Tools for the Design of Reliable and Efficient Functions Evaluation Libraries

Serge Torres

Numéro National de Thèse : 2016LYSEN030

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée par
l'Ecole Normale Supérieure de Lyon

Ecole Doctorale N° 512
Ecole Doctorale en Informatique et Mathématiques de Lyon

Spécialité de doctorat :
Informatique

Soutenue publiquement le 22/09/2016, par :
Serge TORRES

Tools for the Design of Reliable and Efficient Functions Evaluation Libraries

Outils pour la conception de bibliothèques de calcul de fonctions efficaces et fiables

---

Devant le jury composé de :

Langlois, Philippe	Professeur	Université de Perpignan	Rapporteur
Michelucci, Dominique	Professeur	Université de Bourgogne Dijon	Rapporteur
Berthé, Valérie	Directrice de recherche CNRS	Université Paris Diderot	Examinatrice
Villalba Moreno, Julio	Professeur	Universidad de Màlaga Espagne	Examinateur
Muller, Jean-Michel	Directeur de recherche CNRS	Éns de Lyon	Directeur de thèse
Brisebarre, Nicolas	Chargé de recherche CNRS	Éns de Lyon	Co–encadrant de thèse

Éléments réglementaires en langue française

Introduction

La conception des bibliothèques d'évaluation de fonctions est un activité complexe qui nécessite beaucoup de soin et d'application, particulièrement lorsque l'on vise des niveaux élevés de fiabilité et de performances.

En pratique et de manière habituelle, on ne peut se livrer à ce travail sans disposer d'outils qui guident le concepteur dans le dédale d'un espace de solutions étendu et complexe mais qui lui garantissent également la correction et la quasi-optimalité de sa production.

Dans l'état actuel de l'art, il nous faut encore plutôt raisonner en termes de « boite à outils » d'où le concepteur doit tirer et combiner des mécanismes de base au mieux de ses objectifs plutôt qu'imaginer que l'on dispose d'un dispositif à même de résoudre automatiquement tous les problèmes.

Le présent travail s'attache à la conception et la réalisation de tels outils dans deux domaines:

- la consolidation du test d'arrondi de Ziv utilisé, jusqu'à présent de manière plus ou moins empirique, pour l'implantation des approximations de fonction ;
- le développement d'une implantant de l'algorithme SLZ dans le but de résoudre le « Dilemme du fabricant de table » dans le cas de fonctions ayant pour opérandes et pour résultat approché des nombres flottants en quadruple précision (format `Binary64` selon la norme IEEE-754).

Sommaire

Résumé des chapitres

Chapitre introductif

Ce chapitre introduit les notions, notations et définitions qui seront utilisées dans les deux chapitres suivants.

Mieux comprendre et contrôler le test de Ziv

Au cours de ce chapitre nous donnons plusieurs formules permettant de calculer rigoureusement la « Constante de Ziv » utilisée dans le test éponyme, dans plusieurs contextes numériques (lorsqu'on peut effectuer des calculs en précision arbitraire, en présence ou en l'absence de FMA pour effectuer le test, losqu'on calcule en n'utilisant que la « précision de travail »).

Ces formules sont prouvées et leurs conditions d'emploi précisées. Leur optimalité n'est cependant démontrée que dans quelques cas particuliers. Des test numériques permettent cependant de s'assurer de leur quasi-optimalité.

Dans ce chapitre nous mettons en œuvre des techniques classiques pour résoudre ce genre de problèmes et les difficultés rencontrées sont également très représentatives des celles habituellement rencontrées lorsque l'on travaille dans ce domaine.

La méthode de Coppersmith et le « Dilemme du fabricant de table » (une implémentation de l'algorithme *SLZ*)

Dans ce chapitre nous décrivons l'implantation de l'algorithme que nous avons réalisée. Nous expliquons également une série d'optimisations que nous avons utilisées pour améliorer de manière significative les performances des calculs.

Au regard de ces améliorations, et d'autres pouvant être encore réalisées, nous évaluons également la pertinence de l'algorithme *SLZ* pour résoudre le « Dilemme du fabricant de table » lors de l'approximation de fonctions lorsque les opérandes et les résultats (correctement !) arrondis sont des nombres flottants en quadruple précision.

Conclusions

Les principales conclusions de ce travail consistent en :

- des formules pour le calcul de valeurs sûres et efficaces de la constante de Ziv ; cela permet de continuer d'utiliser ce test dans les meilleures conditions et une totale confiance ; le travail effectué en base 2 pourrait également être reconduit, notamment, en base 10 mais nécessiterait certainement une nouvelle étude complète ;
- la mise en œuvre systématique de l'algorithme *SLZ* est une première ; son utilisation pour résoudre le « Dilemme du fabricant de table » n'est possible qu'avec de très importants moyens de calcul hors il n'est pas certains que ceux-ci puissent être facilement mobilisés pour une telle tâche ; d'autres avancées, plutôt de nature mathématique, seront probablement nécessaires pour pouvoir résoudre ce problème avec des outils de calcul plus banals.

Tools for the Design of Reliable and Efficient Functions Evaluation Libraries

by

Serge Torres

Submitted to the Éns de Lyon - LIP
on 2016-05-31, in partial fulfillment of the
requirements for the degree of
PhD

Abstract

The design of function evaluation libraries is a complex task that requires a great care and dedication, especially when one wants to satisfy high standards of reliability and performance.

In actual practice, it cannot be correctly performed, as a routine operation, without tools that not only help the designer to find his way in a complex and extended solution space but also to guarantee that his solutions are correct and (almost) optimal. As of the present state of the art, one has to think in terms of "toolbox" from which he can smartly mix-and-match the utensils that fit better his goals rather than expect to have at hand a solve-all automatic device.

The work presented here is dedicated to the design and implementation of such tools in two realms:

- the consolidation of Ziv's [1] rounding test that is used, in a more or less empirical way, for the implementation of functions approximation;
- the development of an implementation of the SLZ-algorithm[2] in order to solve the Table Maker Dilemma for the function with quad-precision floating point (IEEE-754 `Binary128` format) arguments and images.

[1] Named after Abraham Ziv.
[2] Named after Vincent Lefèvre, Damien Stehlé and Paul Zimmermann.

Acknowledgments

Saying that this work could not have been performed without the help of many people who deserve to be thanked here is an inescapable cliché because it is perfectly true. Better be a poor stylist than ungrateful. Besides, missing the opportunity to express one's gratitude is not only a moral misstep, it is also, for the failed thanksgiver, a personal deprivation of a great pleasure and a great honor. And there are not so many occasions to write good things about people before their memorial speech as to miss one.

I will start with my advisors, Jean-Michel Muller and Nicolas Brisebarre. Jean-Michel Muller is not only the acclaimed scientist everybody knows, in the floating-point community and beyond, he is also a generous man. He has given a chance into the PhD career to a bunch of improbable contenders, among more regular students, who would not have got this opportunity if it were not because of him. I am proud to be one of these mavericks. Jean-Michel will not only offer the stepping stone: he will make all the journey with you and will not let you down as you stumble. His knowledge, experience and friendliness breathe the needed confidence it takes to make it to the finish line. Nicolas Brisebarre is also a brilliant scientist and a man "who cares". He especially cares about people and the world around him. A cartoonist would portray him as an oversized extended hand. If you are brave enough to grab it, it will make you climb higher than you ever thought you could. But beware, you will have to do what it takes: it is more about lifting oneself than about being pulled up. Should you slip, his hand grip will comfortingly get firmer. Dealing with him brings many happy returns. Personal ones are not the most insignificant. Early 2016 has given him a hard time during which his stamina and dependability have been inspiring and instrumental in the termination of this work. I wish him many and many more PhD students to come.

I am also deeply in debt with my two reviewers, Pr. Philippe Langlois and Pr. Dominique Michelucci. I sent them the initial version of my manuscript at the beginning of summer when people are more in a holiday mood than inclined to arid readings. I must first thank them for not taking this dispatch for an hostile move. But they did more than that. In their emails and reviews they raised unaddressed issues, pointed out errors and suggested betterment. The final state of this document, with regard to accuracy and clarity, owes much to their remarks and questions. These have always been formulated in benevolent terms which alleviated the stress always involved in this important step of the evaluation process. For this too I want to express them my gratitude.

I want to mention my lab bosses for the years this work lasted: Gilles Villard and Guillaume Hanrot. They both have been supportive despite the heavy toll this work took on my schedule at

the expense of some of my other duties. They were more than facilitating: they were inspiring and helpful. “Just do it!” is the customary Gilles’s answer to those who come to him with some blurry idea, waving all the usual excuses for procrastinating about it. If you can get his point, this is an extraordinary kick. Guillaume could be called a “thoughtful ankylophobic”. He wants people and things to keep moving around. But within his deep background and far reaching pondering “moving” has nothing to do with the frantic hustle and bustle so often mistaken nowadays with genuine action. In some important occasions, Guillaume also was directly helpful with specific aspects of this work.

I cannot overlook my colleagues of the technical and administrative teams of the lab. For one, they are my professional folks, I am one of them and forever will as long as my career lasts. Strangely, and while no one has ever said a word about it, I have always felt accountable towards them for the successful termination of this work. For two, they have always been supportive even when, as it happened more often than they deserved, some work I could not do would fall back on their own shoulders. For three they have been helpful just in the same way they always are with all our regular “customers” (PhD students, faculty, interns. . .). And, yes, it makes one really comfortable to be on the “other side of the fence” and to know that dependable people are pulling the organizational strings out there.

Another mention for my fellow PhD students with the AriC team. I find it refreshing to acknowledge that they have accepted me as one of them while I have been young long enough to stand somewhere between their fathers and theirs grandfathers! I want to thank them for this welcome. I am not sure I have always responded adequately and this is possibly my single serious regret in all this experience.

Last but not least: Laurence! To put it mildly, she has never been a great fan of this project. But she has trusted me on this one and I appreciate this trust to its true value. And I also keep remembering that no soothing words will restore my so deeply sunk debit balance. As time will now allow for it, only acts will.

A final work for all those here, in Éns, who have expressed encouragement and helped. In a sense, it is exhilarating to realize that are there so many of them that I cannot recall all the names and all the deeds. My apologies for being so absent-minded and my sincere thanks to everyone.

Lyon, Sep. 2016

Contents

List of Figures

List of Tables

Contributions

This dissertation essentially deals with two apparently loosely connected topics:

- Ziv's rounding test: a technique used in function approximation implementation;
- an implementation of the SLZ-algorithm[1] that addresses the resolution of the Table Maker's Dilemma[2].

But the connection is quite obvious when one has the function approximation framework in mind, and especially the problem of implementing correctly rounded functions.

In the introductory chapter, we classically present a minimal set of notions and definitions used in the two other ones in order to make this work essentially self-contained. It can be skipped by readers familiar with floating-point arithmetic applied to the function approximation field.

In Chapter 2, we give a detailed account of our work on Ziv's rounding test. This material has already been published in [24]. We take advantage here of a less strictly limited space than in a journal paper to give more comprehensive explanations, hopefully easier to follow, than those found in our admittedly rather terse original publication.

The main contribution of our study on Ziv's rounding test is in the sanitation of a technique used for some time now (the original A. Ziv code [111] goes back to 1991) in the realm of function approximation. The practical result we exhibit is quasi-optimal constant values that can be actually used for the test in different contexts[3], backed by rigorous proofs.

The other interest of this presentation is in the exposition of techniques routinely used in floating-point related reasoning and in an attempt to link them, beyond dry calculations, with the underlying informal insights one can have about the problem.

Chapter 3 is devoted to an implementation of the SLZ-algorithm (§3.3 and on) and to some results we were able to obtain with it for the $\exp(x)$ function (§3.7 and §3.9).

[1] An algorithm designed and first published in 2003, by V. Lefèvre, D. Stehlé, and P. Zimmermann, see [102] and [103].

[2] Put very informally and in a nutshell, the Table Maker's Dilemma can be stated as: how accurate should an approximation g to some function f be in order to make sure that we can always compute the correct rounding of $f(x)$ to some precision p from $g(x)$?

[3] With and without FMA, with and without arbitrary precision arithmetic.

The from-the-ground-up developed implementation is independent of that mentioned in [102] for we have at hand tools[4] that were not available back then. In §3.6.2 we give an extended example of its behavior. Many more tests than those presented have been performed to check the validity of the code. Those displayed here are exclusively devoted to the $\exp(x)$ function and the 113-bit precision. Nevertheless these are only specific input parameters since our implementation can deal with any function[5] our tool base can handle and with any sensible precision.

The parameters chosen for our presented experimental data were aimed to a precise target: "solve"[6] the Table Maker's Dilemma for the "quad precision" (aka the IEEE 754 `binary128` format), with the epitomizing $\exp(x)$ function, for this format might become, in a not too distant future, available either in hardware or in efficient software implementations.

More precisely, and beyond the theoretical complexity results obtained for a hardness-to-round in the order of the square of the precision (see [99]), we have wanted to assess if more realistic values[7] could be practically checked. That kind of work has never been performed before. Our ultimate goal here is to validate a possible scheme (exposed in §3.1) for the development, as of today or very soon, of correctly rounded elementary functions in quadruple precision.

In the end, our conclusion is not fully... conclusive! While technically conceivable, the exploration of a function like $\exp(x)$ would require the deployment of an extensive computing infrastructure. In our view this situation should spur two kinds of efforts:

- optimize the Coppersmith technique for our particular context[8];
- develop alternative mathematical approaches to the Table Maker's Dilemma than those explored so far.

Passing, this work also brings in (§3.8.3) a new and dazzling confirmation to the results exposed in [1]. In this paper about possible optimizations in the search of a basis for a lattice with some short-norm vectors, the authors show the gains one can expect from using random projection and LLL-reduction of the projected matrix instead of a straight reduction of the initial matrix representing the lattice, in the so-called "rectangular case". Acknowledging the stunning efficiency[9] of this technique in our particular case, we can only recommend systematically giving it an inexpensive try whenever dealing with Coppersmith-like algorithms.

[4] `fpLLL` [2], Sage [92] and Sollya [14].
[5] There are nevertheless some mathematical limitations exposed in §3.3.3.
[6] What we exactly mean mean here by "solve" is explained with more details in §3.1.2.
[7] In the 6 to 10 times the precision range.
[8] Possible leads are given at sections §3.9.5 and §3.10.
[9] Two orders of magnitude speedups.

Our test results also indicate (§3.7.1.3) that the algorithm (or, at the very least, our implementation) does not perform equally well, by a wide margin, under any arbitrary set of parameters. For some of them, the behavior is markedly better. The practical implication is that careful parameter tuning is a vital preliminary step to any tentative large-scale exploration.

By-products of this work are the codes created for the implementation (see [107]). The two main realized pieces are:

- Sollya[10] to Python/SageMath[11] integration;
- interval exploration with the SLZ-algorithm for an arbitrary precision and function.

To the moment, both should be considered as research prototypes rather than as off-the-shelf products for general audience use.

[10] Sollya [14] is an *ad hoc* designed tool for handling numerical functions and working with arbitrary precision. It is extensively used in our implementation for approximation and infinity norm computations.

[11] SageMath [92] is a free open-source computer algebra system.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

1.1 Topics and motivations

All the work presented here is, one way or another, related to function approximation. This topic is possibly one of the most overworked in Computer Science since its inception. To make the matter worse to those who only swear by absolute novelty, initial work on function approximation predates by centuries the creation of the first computer (whatever the preferred device of the reader is for the title). And yet, as of 2016, it is a very lively investigation field, with groups of researchers all around the World devoted to, scientific congresses and conferences held on, and articles not only written but even paper printed and read about. Such a topical subject, with deep roots in a wealthy history, necessarily has a special appeal of its own.

1.1.1 The landscape

When people find out that one is working in Computer Arithmetic there is often a moment of puzzlement. What? Problems with numerical computations? Is not this supposed to be the most unquestionable realm of computers? What do we call them "number crunchers" for? There can even be a moment of suspicion. Are not these Computer Arithmetic folks comfortably encrusted in some sinecure at the expense of our hard earned tax cash? It does not get any better if one specifies that she is specifically working on function approximation. The elders remember of their trigonometric and logarithmic tables, the youngsters of their pocket calculators: it is all there, is not it?

This lack of knowledge about pending issues is not limited to the public at large (but who could seriously blame it?). More surprisingly, it is shared by a wide fraction of the technical or even scientific community. If a personal memory is relevant here, the author vividly recalls (nowadays,

with a bit of shame) his astonishment when, as an already seasoned computer engineer, he joined his current laboratory, fifteen years ago. He then discovered there was a Computer Arithmetic team and the topics it was working on. All right, by 2000, cars did not fly as had been speculated by overenthusiastic futurists, but still in numerical trouble with computations at the break of the 21st Century? It must be a joke.

We have to acknowledge that there is some truth in the naive views exposed above. Indeed, a lot of work has already been done and achievements are spectacular. Limiting ourselves to recent examples, Higgs's Boson hunt could not have succeeded if computers were performing that bad at numerical tasks. In this realm, as in many others, the "standing on the shoulders of giants" metaphor is perfectly appropriate. Nevertheless many issues are still pending and, worse (or better, depending on one's perspective) new ones pop up at an untamed rate.

We can spot, at least, three driving forces for the need of research in function approximation, to limit ourselves to this question, but without ever forgetting that it is a piece of the much wider "computer numericals" domain.

One is the many-facet evolution of hardware and its use. The improvement of general purpose processors (notably FMA instruction, vector instructions, and soon, rounding mode tagged operations) allows for innovative ways to perform classical computations. The "scissors effect" affecting processing power and memory access performance is also a big incentive for the design of new (or the adaptation of older) algorithms to mitigate it. The widespread use of GPU and innovative architecture chips (e.g. Xeon Phi) for numerical computations rises the pressure for innovation too. The ubiquity of digital devices powered by a wide range of chips offering limited resources (available electric power, instruction set, memory, etc.) that have to perform function computations impose new designs and the automation of their conception as it has to become routine in order to keep the pace of technological evolution and applications. Last but not least, the soaring use of programmable circuits (e.g. FPGA) challenges our ability to devise the best ways to take advantage of their peculiar (and evolving) architectures for a fast pace growing application domain.

Applications are another pusher. "Explosion" is more appropriate than "Evolution" to describe what is going on in this realm. The question is not anymore "What could applications be?" but rather "Where do digital technologies really not belong?". And the answer is not an easy one. This question is intimately connected with hardware issues and asks for an answer in terms of our ability to quickly give *ad hoc* solutions. But high-end applications become more and more challenging. Physical models grow larger, longer time spans without numerical degeneracy are wanted for simu-

lations. This involves not only being able to perform computations faster but also getting better a grip on their numerical quality.

For our third pushing force we have chosen to highlight a question that could belong to the application realm as well. But we think it deserves a special attention. It is safety and, more generally, numerical quality control. The widening spectrum of numerical devices and applications has the consequence that more and more of them are used for critical or near-critical applications. This can happen without the designers being really aware of it with the growing complexity emerging from the cooperation of possibly many unrelated but interconnected systems. This problem affects every one, from hardware architects to software builders and gives a particular responsibility to those in charge of the numerical founding blocks. This comes along with an increased demand for certification and stresses our ability to support our quality claims with unquestionable evidence.

All these challenges have to be addressed and this represents a heavy workload for the Computer Arithmetic community and for all those related to these issues. Indeed, one of the present trends is that problems need to be tackled by new approaches for which specialists of different domains have to cooperate.

We reach now the delicate point when one needs to hinge what can only be a limited contribution presented here to the very wide perspectives and general motivations sketched above and, if possible, avoid a harmful clay pot / iron pot clash.

1.1.2 What is this work about?

Our answer comes in four parts. In this overview we will use terms that are not defined yet. We apologize to the reader for our feeble solution to the chicken-egg problem any introduction faces. Nevertheless all the necessary clarification will be delivered at some point in the sequel.

1.1.2.1 Function approximation

The scope over which digital computations exactly match mathematical ones is relatively narrow. When trying to implement a mathematical object and its behavior in a program [1], one quickly faces the fact that she can not exactly represent them. This is only a half truth if one also considers Computer Algebra, Formal Tools and the quick progresses in these domains. In fact, the above statement stands better when numerical high performance is also required. All that can actually be done, when we can not manipulate the mathematical object, is to use or create a surrogate: an

[1] We do not mean to refute, passing, the sacrosanct Curry-Howard correspondence. The only problem is that if actual programs are mathematical proofs, those, very often, have little to do with what the developer has in mind.

approximation. When it comes to functions, replacing them by approximations is common place. Among the many arbitrary ways of splitting the function approximation realm in two parts, one is to distinguish the situation where the function is known from that where all we have at hand is a "black-box"[2] type function or even a plot made of a finite number of domain $\times$ codomain pairs.

In both situations, tools and methods differ, notwithstanding overlaps. In the first case, Approximation Theory prevails. In the second, Interpolation, Extrapolation, Regression Analysis and Curve Fitting are the most often spoken words. This work is entirely relevant to the first situation. We not only know explicit formulas for the functions we deal with but we also have high accuracy approximations whose main drawbacks are limited performance (computing time) and heavy requirements (hardware, memory consumption, etc.). What we work on are, in a sense, "second order" approximations, to fulfill more limited resources requirements and exhibit decent performance. But they could not be built if it were not for these function initial high accuracy approximation tools (e.g. MPFR[34]).

Another possible ways to bisect the function approximation realm is to make a distinction between *univariate* and *multivariate* functions. Work presented here is almost exclusively dedicated to functions of one variable, even if extensions to other cases could be envisioned.

When we refer to function approximation, what we generally have in mind are *elementary functions*. An elementary function is a function of one variable which is the composition of a finite number of arithmetic operations ($+$, $-$, $\times$, $/$), exponentials, logarithms, constants, and solutions of algebraic equations (a generalization of nth roots). A subset of these functions is usually available from the `libm`-libraries delivered with compilers and those are, by themselves or in composition, the "bread and butter" of scientific computing[3]

But this work will not deal with function approximation at large. It is focused on a limited number of particular questions, even if all are relevant to function approximation. Let us enter into more details.

[2] An object we can only probe by giving it an argument and getting the output image, without any knowledge of its inner workings.

[3] In a meeting with CERN scientists working on numerical efficiency, they reported us that they found out, after instrumenting some pieces of code, that 30 % of the computing time was spent in elementary functions. Of course this result should not be overgeneralized but considered anyway as valuable hint on elementary functions impact and importance.

1.1.2.2 Ziv's rounding test

As will be seen in Chap. 2, p. 79, Ziv's Rounding Test is not news. This is a technique that has been in use for years to validate (or not) the result computed by an approximant under the form of two floating-point numbers. The novelty in this work lies in the fact that this technique had never been analyzed in details. It relies on the use of a constant whose optimal value had never been thoroughly investigated. In this work two near-optimal correct bounds are given with a bundle of proofs for correctness (and optimality when specific contexts allow for it) and extra empirical evidence.

This work is part of a larger sanitization effort to "declutter the attic" that aims to revisit pieces of the worthy heritage left by our forerunners and, possibly, fix them (when proofs were not... proofs!), extend them (to other radices, with new instructions such as FMA, etc.) or, regrettably, dump them to the junkyard (when too badly broken).

1.1.2.3 Quad-Precision and the Table Maker Dilemma

In this part, Chap. 3, p. 135, we reconnect with wide perspectives briefly presented above. At the same time, we do not leave the realm of tool manufacturing. One of the possible improvements in the area of Computer Arithmetic would be the availability of 128-bit floating-point numbers support in hardware. It remains to be seen if manufacturers will incorporate it in a near future, if at all. But there are already software implementations, with variable degrees of compliance with the IEEE 754-2008 standard. There is still room for a dramatic improvement of their performance which is their weakest point. Will betterment reach a point where users will consider to use them extensively? There are no answers yet.

While all these issues are still in debate, a question pops up to mind: will we be able to use 128-bit floating-point numbers "right", beyond basic operations? More precisely will we be able to build correctly-rounded mathematical libraries for that precision? The answer is important at two different moments. One is, of course, when we get there: when our big-irons are capable to crunch quad-precision numbers (almost) as quickly as they do nowadays with double-precision ones. The other is before that ecstatic day. Do we really need to rush for quad-precision if it is to discover that we can not create the tools to really take advantage of it?

This is where the work presented here comes into play. A prerequisite to building correctly rounded libraries for 128-bit floating-point numbers is our ability to solve the Table Maker Dilemma for that precision. We know for a fact that the current toolset we use to deal with the double-precision format will not scale up. We have to explore new avenues. The SLZ-algorithm [102] [103] is such a novel approach.

Our idea here is twofold:

- develop an implementation of the algorithm;
- try to validate an approach that, without actually solving the TMD, will make it possible to design suboptimal but usable correctly-rounded function approximations.

What comes next in this chapter is, in part, what was missed in the introduction bits. We present the necessary concepts to get with the subsequent chapters. Nothing particularly new should be expected here and we will try to limit ourselves to what is absolutely necessary.

To begin with, we will get back where it all started: the womb of the floating-number beast.

1.2 Floating-point numbers: a world of their own

It is nowadays common knowledge that the finite representation of numbers in computers sets up a whole range of issues and that machine floating-point numbers should not indiscriminately be manipulated as elements of $\mathbb{R}$ are unless one is seriously looking for trouble.

Floating-point representation of numbers, as per IEEE 754-2008 standard (see [47]), is widely accepted and most (if not all) of the modern processors offer efficient and compliant machine instructions to process them. It has become the *de facto* standard for number crunching applications.

A key operation when dealing with these floating-point numbers and which can be blamed as the source of (most of[4]) all evil in their use is *rounding*.

According to the *Rounding* article of Wikipedia, rounding a numerical value means:

> replacing it by another value that is approximately equal but has a shorter, simpler, or more explicit representation...

The article goes on with the following example:

> replacing... $\sqrt{2}$ by 1.414.

that epitomizes what is constantly done in scientific computing.

[4] Arithmetic exceptions should not be overlooked. To mitigate our statement, it must be said that *rounding functions* are a great progress: instead of returning some result "not-too-far", in a very fuzzy sense, from the exact value, the concept of function conveys with itself the notion of a *specification* of the difference between the exact and the returned value.

One must notice that rounding does not only take place for the representation of irrational numbers constants or that of experimental data. Rounding is performed after almost any of the basic arithmetic operations $\{+, -, \times, /\}$ (it is common, as in the IEEE-754 standard, to append $\sqrt{}$ to this set) whenever the exact result does not fit into the fixed format of the target representation. Not to speak about rounding when elementary functions (and combinations thereof) are computed. As such, rounding happens every time and everywhere along any non trivial computation.

Let us take a look to a simple example on how things can quickly get weird, even with a short program that toys around with 1.1, 1.0, 0.1 values, addition and subtraction.

```
int main(int argc, char** argv)
{
  double oneDotOne                    =  1.1;
  double one                          =  1.0;
  double oOne                         =  0.1;
  double diff_oneDotOne_one           =  0.0;
  double sum_one_oOne                 =  0.0;
  double sum_one_diff_oneDotOne_one   =  0.0;

  diff_oneDotOne_one                  =  oneDotOne - one;
  sum_one_oOne                        =  one + oOne;
  sum_one_diff_oneDotOne_one          =  one + diff_oneDotOne_one;

  if (diff_oneDotOne_one != oOne) {
    fprintf(stdout, "(1.1 - 1) != 0.1\n");
    if (oneDotOne == sum_one_oOne) {
      fprintf(stdout, "(1.1 - 1) != 0.1 but 1.1  == 1 + 0.1\n");
      if (sum_one_diff_oneDotOne_one == sum_one_oOne) {
        fprintf(stdout, "(1.1 - 1) != 0.1 but (1 + (1.1 - 1)) == (1 + 0.1)\n");
      }
    }
  }
  return 0;
}
```

The output of this program is:

```
(1.1 - 1) != 0.1
(1.1 - 1) != 0.1 but 1.1  == 1 + 0.1
(1.1 - 1) != 0.1 but (1 + (1.1 - 1)) == (1 + 0.1)
```

This is a lot of violations of usual arithmetic rules for such a small piece of code.

There are many situations where floating-point number computations can go totally out of control (no single digit in the result is valid) if some (sometimes, great) care is not taken. Correct (accurate) use of floating-point numbers can lead to quite counter-intuitive algorithms.

From the early days of numerical computing, a large and fruitful research effort has been dedicated to their correct and yet efficient use. It has produced the huge body of results that have allowed for the present situation of general pervasiveness of numerical computations in all realms of science (even Humanities!) and engineering.

Computer scientists, numerical analysts and users have always wanted to quantify the divergence of actual computer calculations from their exact mathematical counterparts. This is the starting point of *error analysis* where one tries to figure out how rounding errors show up and accumulate along computations to evaluate the quality (or lack thereof) of a numerical algorithm or a piece of code.

Floating-point numbers, as defined in the IEEE 754-2008 standard should not only be considered as mere crippled real numbers but as true mathematical objects, with their peculiar (and sometimes disconcerting) properties, as a world of their own.

This is why we will spend some time to present the properties of the IEEE 754-2008 standard, not striving for completeness but trying to limit ourselves to what is really useful to grasp what this work is about. For a full-blown description and analysis one could consider giving a hard look to [78].

1.3 Basic Notions and Definitions

1.3.1 Floating-point numbers

A floating-point representation of numbers is made of a series of parameters and a set of rules that considered together define what we will call here a "Floating-point format" [5]. In particular, for the standard parameters, the set of numbers defined by the format is finite.

We are going to build a floating-point number step by step, one piece at a time. Basically, and commonly[6], a floating-point number can be seen as the signed product of two positive numbers: a

[5] The "Floating-point system" expression is also frequently used.
[6] Other presentations are possible: see [41]. We stick to the "classical" one here.

fixed size integer and a rational number

$$\pm \text{ fixed\_size\_integer } \times \text{ rational\_scaling\_factor.}$$

The integer is represented by a string of digits of a preliminarily specified length. It is called the *integral significand*[7]. It will be denoted by M. Its possible values are the set of integers spanned by the available digits. To represent fractional values or values larger than the maximum integer, a scaling factor is applied: the rational number. A floating-point system specifies, among other things, what are these factors made of through a number of parameters.

The first parameter of a floating-point format is the *radix*, a positive nonzero integer larger than 1, hereby denoted by β. It plays two roles. For one, it is the radix of the numeral system the integral number is written in.

The second role is that, raised to some integer power, it will form the scaling factor of the floating-point number,

$$\pm \text{ integral\_significand } \times \beta^e, \text{ with integral\_significand expressed in base } \beta \text{ and } e \in \mathbb{Z}.$$

The second parameter of a floating-point format is the *precision*, denoted by p. It is the number of digits the integral significand is made of:

$$M = M_{p-1} \dots M_0,\ 0 \leqslant M_i < \beta,\ \beta \in \mathbb{N} \setminus \{0, 1\},\ i \in \{0, \dots, p-1\}.$$

Indexing the digits from right to left allows us to say that digit M_i has *weight* β^i. The values spanned by M are the integers in $\{0, \dots, \beta^p - 1\}$.

To summarize and roughly speaking, a precision-p and radix-β floating-point number X can be represented as:

$$X = \pm\ M\ \times\ \beta^e,$$

where β is the radix, M is X's integral significand and e is the X's exponent corresponding to M[8].

Notice that, to the moment, this fuzzy idea of what a floating-number is does not convey the notion of finite set of numbers since the exponent e can take arbitrary values. To settle the matter, bounds must be enforced on e.

[7] Clarification about significands is just below.

[8] As will be seen there can be several (M_i, e_i) pairs that represent the same floating-point number X. That is why we cannot speak here about "the" exponent of X.

Let us enter into more details.

Definition 1.1. A floating-point format is (partially) [9] characterized by four integers:

- a radix (or *base*), $\beta \in \mathbb{N}$ and $\beta \geqslant 2$;
- a *precision*, $p \in \mathbb{N}$ and $p \geqslant 2$ (roughly speaking, p is the number of "significant digits" of the representation and sets one of the bounds for its finiteness);
- two *extremal exponents*: $e_{\min} \in \mathbb{Z}$ and $e_{\max} \in \mathbb{Z}$ such that:
  - $e_{\min} < e_{\max}$,
  - $e_{\min} > -\infty$ and $e_{\max} < +\infty$, which seals the finiteness property,
  - in all practical cases $e_{\min} < 0 < e_{\max}$.

We will often denote such a format by $\mathbb{F}_{\beta,p,e_{\min},e_{\max}}$ or as $\mathbb{F}$, if the context allows for it, with the doubtful hope that the compactness of this notation will forgive for its ugliness. From time to time, we will also abuse a bit our notation by using $\mathbb{F}$ as the set of the finite numbers [10] which are defined by the format, a subset of $\mathbb{Q}$.

A floating-point number X representing a finite numerical value in such a format is a number for which there exists at least one representation (M,e) such that

$$X = \pm M \times \beta^{e-p+1}, \tag{1.1}$$

where:

- $M \in \mathbb{N}$ and $M \leqslant \beta^p - 1$;
- $e \in \mathbb{Z}$ and $e_{\min} \leqslant e \leqslant e_{\max}$, is called the *exponent* of the representation of X.

We do just recall here that this representation is also signed but since this is a general property of floating-point formats it is not systematically mentioned.

Up to this point, the representation of a number is not necessarily unique. To illustrate that, let us consider the following "toy format": $\mathbb{F}_{2,4,-7,8}$ (radix: $\beta = 2$; precision: $p = 4$; minimum exponent: $e_{\min} = -7$; maximum exponent: $e_{\max} = 8$). In this format, number 8 can receive different representations such as:

[9] The full definition of a format also specifies the binary encoding of the significands and exponents. It also deals with special values: infinities, results of invalid operations, etc.

[10] As we will see, we will append special notations to denote infinities and other special values. Finite numerical values are those that do not refer to these special cases.

- (+,1,6): $1 \times 2^{6-4+1} = 8$;
- (+,2,5): $2 \times 2^{5-4+1} = 8$;
- (+,4,4): $4 \times 2^{4-4+1} = 8$;
- and so forth...

The set of all these equivalent representations is called a *cohort*.

Number β^{e-p+1} is called the *quantum* of the representation of X.

This state of things may not always be desirable. In fact there are many and diverse reasons why we may want to set an extra rule that allows for one, and only one, representation for a number. We give here two examples:

- if we only allow for one representation of a floating-point number, we can have the notion of "the" exponent of a number; in turn, this will allow us to conveniently define concepts as the *unit in the last place* of a number, whose definition will be given below (§1.4.1, p. 39) and is very useful to make reasoning about errors ;
- the authors of the IEEE 754 standards put a lot of dedication into crafting the numbers encoding scheme so that it exhibits interesting properties; one is, for instance, that numbers ordering is consistent with the lexicographic order of their representation; this is extremely convenient if we are to have efficient comparisons; allowing for multiple representations of a number would break this consistency; another one is that if we view representations as integers, that of the successor of some floating-point number can be obtained by adding 1 to the representation of the latter.

Normalization also brings in its own share of problems [11]. But, all in all, normalizers felt the advantages outweighed inconveniences. Hence format is specified further to allow only for a single representation of a number that will be called its *normalized representation*.

1.3.2 Normalized representation

Among the previous different representations we could pick the one, if it exists, such that:

$$\beta^{p-1} \leqslant M \leqslant \beta^p - 1.$$

[11] For instance a sharp cutoff effect for values smaller than the smallest normalized number that can be brutally flushed to 0 at rounding. To mitigate this issue, *subnormal numbers* had to be introduced for very small values, adding an unwelcomed level of complexity to the system.

In our toy format $\mathbb{F}_{2,4,-7,8}$: $8 \leqslant M \leqslant 15$.

With this convention, the favorite representation of number 8 would be (+,8,3): $8 \times 2^{3-4+1} = 8$.

Notice that 2^{-10} does not have such a representation. Even using the smallest possible exponent ($e_{\min} = -7$), what yields $\beta^{e_{\min}} = 2^{-7}$ in our example, there is no integer M such that $M \times 2^{-7-4+1} = 2^{-10}$ and $M \geqslant 8$. As seen above, the smallest allowed value for M in this case is 2^3. The smallest positive number that has a representation conforming to our conventions is 2^{-7}: $2^3 \times 2^{-7-4+1} = 2^{-7}$. Nevertheless, 2^{-10} has the following *denormalized* representation (+,1,−7): $1 \times 2^{-7-4+1} = 2^{-10}$. It is the smallest non zero positive number one can represent in this format.

In this floating-point format, 8 will be called a *normal number* and 2^{-10} a *subnormal number* (sometimes also called a *denormalized number*).

From there, the representation of a normal number X could take form of a triplet (s,m,e) such that:

$$X = (-1)^s \times m \times \beta^e,$$

where:

- $e \in \mathbb{Z}$ is the same as before, "The" exponent (but this time with a capital "T");
- $m = M \times \beta^{1-p}$ is called the *normal significand* or, in short, the *significand*[12] of the representation; it has one digit before the fractional point, and at most $p - 1$ digits after it; also notice that $1 \leqslant m < \beta$ and that m is a rational number;
- $s \in \{0, 1\}$ is the bit sign of X (its sign is $(-1)^s$).

A collateral benefit is that, in radix-2 formats, the first digit of the significand of a normal number is always 1 so it does not need to be stored. It is a way to save one bit in encoding. This technique is called many names: *leading bit convention*, *implicit bit convention*, *hidden bit convention*.

The significand is also frequently (though slightly improperly) called the *mantissa*[13].

For a given floating-point format, the numbers that can be represented in a (unique) normalized form will be called *normal numbers*.

[12] This also justifies the "integral" adjective when is comes to M as a significand.
[13] Accurately, the mantissa of a non-negative number is the fractional part of its logarithm.

Whe can also define some usefull values:

$\omega = \beta^{e_{min}}$ is the smallest absolute value for a normal floating-point number in $\mathbb{F}$;

$\Omega = (\beta^p - 1)\ \beta^{e_{\max}-p+1}$ is the largest normal floating-point number in $\mathbb{F}$.

With the previous normalization convention[14], one quickly sees that there is no normalized way to express number 0, which is, at the very least, embarrassing. We will deal with it later (§1.3.4, p. 37).

1.3.3 Subnormal numbers

For implementation reasons[15], it would be very tempting to limit the set of the floating-point numbers to those that can have a normalized representation. But a less obvious problem arises: a very sharp cut-off effect for the very small absolute values. With the round-to-nearest rounding mode, all the values smaller than $\frac{\beta^{e_{\min}}}{2}$ will be rounded to 0.

1.3.3.1 Subnormal numbers are our children too

An alternative is to allow for subnormal numbers. This has the first beneficial effect of allowing for what W. Kahan calls a *gradual underflow*: loss of precision is progressive instead of being abrupt.

It also has an unexpected interesting property: if $X \neq Y$, with $X \in \mathbb{F}$ and $Y \in \mathbb{F}$ then the computed value of $X - Y \neq 0$. This even applies to subnormal numbers! See Figure 1-1.

When the first version of the IEEE 754 standard was published in 1985, allowing for subnormal numbers was a controversial decision. The terms of the debate were, on the one hand, the difficulty of an implementation in hardware (or conversely the performance impact if implemented in software), and, on the other hand, the beneficial properties brought by the availability of subnormal numbers: ease of development of stable numerical software in some fields, weakening the conditions for some important results as the Sterben'z lemma [104].

1.3.3.2 Underflow

The term can be ambiguous in a variety of ways. A definition that springs to mind is that underflow happens whenever the exact result of an arithmetic operation has an absolute value smaller than the

[14] Notice that the IEEE 754-2008 standard requires a normalized representation in radix 2 but not in radix 10. In the sequel, we will mostly interested in the former case.

[15] As we will see in the part dedicated to IEEE 754-2008 standard, taking subnormal numbers into account leads to a more intricate encoding and, as their implementation is often realized in software, an adverse performance impact is incurred when they show up in a computation. This is not always the case: see [85], p. 39–40 or [32] for counterexamples.

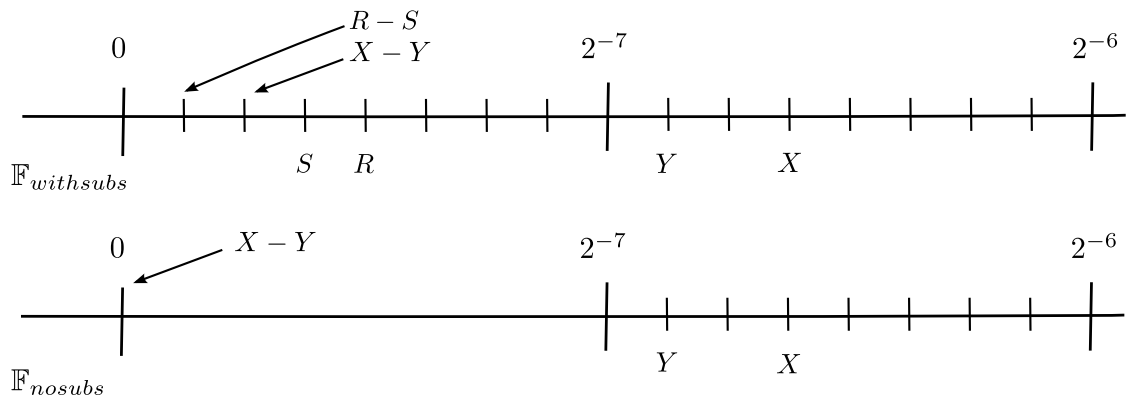

Figure 1-1: Life near 0 with and without subnormal numbers in our toy floating-point system. In the first case there are still numbers between 0 and 2^{-7}, the smallest normal number, and the subtraction of two different nonzero numbers never yields 0 in the rounding-to-nearest mode. This remains true even if the numbers are both picked from the subnormal set. In the second case, the subtraction of two close and small normal numbers may yield 0. Incidentally, we also have the following interesting property: the sum of two subnormal numbers is always exact.

smallest nonzero *representable* number. This is supposed to include subnormal numbers, if they are available. If it were the case underflow would relate to absolute values smaller than $\beta^{e_{\min}-p+1}$. **But this is not the definition adopted by the successive IEEE 754-1985 and IEEE 754-2008 standards**.

Here, underflow is defined in reference to the *smallest normal number*, $\beta^{e_{\min}}$.

But to make the situation more confusing there are indeed two different definitions of underflow and none of the standards made a choice about them.

Definition 1.2. Underflow *before* rounding.
In radix-β, an arithmetic operation or function underflows if the exact result of that operation or function is of absolute value strictly less than $\beta^{e_{\min}}$.

Definition 1.3. Underflow *after* rounding.
In radix-β, precision-p arithmetic, an arithmetic operation or function underflows if the result we would compute with an unbounded exponent range and precision p would be nonzero and of absolute value strictly less than $\beta^{e_{\min}}$.

In practical terms, they only differ on very narrow interval between $\beta^{e_{\min}}$ and its preceding floating-point number [16] whose exact location and width depend on the rounding mode.

[16] If subnormal numbers are available.

We must also notice that, when it comes to actual behavior of a IEEE 754 floating-point implementation, an underflow is signaled (the underflow flag is raised) only if the computed result is inexact. The rationale is that an underflow matters to the user as it indicates that the relative error of the result might be larger than usual. If the computed value is exact, there is not point in needlessly bothering the user.

1.3.3.3 Normal and subnormal range

We will call *normal range* the set of real numbers whose absolute value belongs to the $[\omega, \Omega]$ interval over $\mathbb{R}$.

Along the same lines, we will call *subnormal range* the set of real numbers whose absolute value belongs to the $[0, \omega)$ interval over $\mathbb{R}$.

1.3.4 Special floating-point data

We use here the more generic "data" term rather than "numbers" because some of our special items can not strictly be considered as "numeric". Why are these data needed? The main reason is that we want floating-point numbers to be a stable set: any machine operation has to be well defined[17] and return a floating-point datum. Having a "closed" system has two main advantages:

- it is a must for portability (there should be no case where the results returned by two different implementations diverge);
- it helps for reasoning about floating-point numbers whenever everything is well defined.

One obviously needed element is a datum to represent values whose magnitude is larger (resp. smaller) than that of the largest (resp. smallest) representable floating-point number. Conceptually this can be done in different ways: as an infinity "absolute value" (∞) that can be signed to denote both situations or, directly, as two *signed infinities* ($-\infty$ and $+\infty$). The latter convention is adopted in the IEEE 754-2008 standard. Moreover, the binary representation is built in such a way that the conceptual difference mentioned above has no impact on how things are done in practice: in binary representation, $-\infty$ and $+\infty$ only differ by the leading sign bit.

As the reader may remember, we had no way to give a normalized representation for 0. As a consequence of the previous choice on infinities and for symmetry reasons $\left(\frac{1}{+0} = +\infty \text{ and } \frac{1}{-0} = -\infty\right)$, *signed zeros* ($-0$ and $+0$) have also been appended. For a more in-depth discussion the reader may refer to [53].

[17] Without generating a trap (i.e. a transfer of control to some handler routine).

Eventually any operation over the floating-point numbers, however invalid over the real ones $\left(\text{e.g. } \sqrt{-5} \text{ or } \frac{0}{0}\right)$, should output a result. The result of an invalid operation will be represented by a special datum, denoted by NaN (for *Not a Number*) in IEEE 754-1985 standard and successor, specifically devised for this purpose. As a matter of fact, two kinds of NaNs are defined by the standards: *quiet NaN* (qNaN) and *signaling NaN* (sNaN). A complete explanation can be found in [78].

We have to acknowledge the fact that the introduction of special floating-point data also brings in its own burden of issues. What should be returned when special values such as $-\infty$, $+\infty$ or NaN are themselves the arguments of an operation or a function is still debated for some cases. Sometimes more or less arbitrary decisions have to be taken: in IEEE 754-2008 pow(qNaN,0) and pow(1,qNaN) should both return 1 since that is what the function pow does when qNaN is replaced by any other floating-point number (sNaN excluded) in the above expressions. While this adds up to the arcane reputation of floating-point "theory", all in all, the introduction of special values is unquestionably beneficial.

1.3.5 $\mathbb{F}$ as a finite and (almost) ordered set

$\mathbb{F}$, as we have defined it, is always a finite set. If we exclude NaN's and make a particular case of signed zeroes, this set is totally ordered. For each (but $-\infty$) floating-point number X we can then have notion of the (unique) *predecessor* of X (denoted here as X^-). Similarly, for each (but $+\infty$) floating-point number X we can have the notion of the (unique) *successor* of X (denoted here by X^+).

This allows us to represent $\mathbb{F}$ as a graduated ruler over the $\mathbb{R}$ axis. We will very often add extra graduations to represent, for instance, *midpoints*. These are the numbers that represent the center of the interval defined by two consecutive elements of $\mathbb{F}$. They themselves are not elements of $\mathbb{F}$ and should not be confused with them. But they are instrumental for reasoning about rounding.

1.4 The ulp() function

When dealing with floating-point number operations, one is constantly worried about errors between exact values and rounded (or approximated) ones. Conceptually, we make a distinction between *relative errors* and *absolute errors*.

If $X \in \mathbb{F}$ is an approximation of $x \in \mathbb{R}$ they are defined as follows:

Definition 1.4. Absolute error:

$$\epsilon_{abs} = |X - x|.$$

Definition 1.5. Relative error:

$$\epsilon_{rel} = \left|\frac{X - x}{x}\right|, \text{ if } x \neq 0.$$

If $X = x = 0$ then $\epsilon_{rel} = 0$.

This distinction is of course of paramount importance. Nevertheless there are situations where one would like to discuss about "parametrized" absolute errors. In other words, absolute errors whose value depends on the context.

1.4.1 Definitions

The ulp() function is such a tool. This concept has a somewhat complicated story. See [78] §2.3.1 for an in-depth discussion about this topic. In the sequel we will stick to the option selected in this reference: we will use Goldberg's [36] definition extended to real numbers, since it is the most widely used.

Let us start with the original Goldberg's definition of the units in the last place.

Definition 1.6. If a floating-point number $X = d_0.d_1d_2d_3\ldots d_{p-1} \times \beta^e$ is used to represent a real number x, it is in error by $|d_0.d_1d_2d_3\ldots d_{p-1} - \frac{x}{\beta^e}|$ units in the last last place.

This definition has the following limitations:

- it defines the number of units in the last place for a (floating-point, real number) pair and not as an intrinsic attribute of a (real) number, for a given floating-point system $\mathbb{F}$;
- it does not take subnormal numbers into account.

But from this definition we can derive the following one, more fit to our purposes.

Definition 1.7. Generalized Goldberg's definition : Let $x \in \mathbb{R}$. If $|x| \in [\beta^e, \beta^{e+1})$ and $e \leqslant e_{\max}$ then $\text{ulp}(x) = \beta^{\max(e,e_{\min})-p+1}$.

This ulp() definition stands for any real number, including floating-point numbers. Notice how this function is parametrized by radix β and precision p. This is not shown in notation but it must always be kept in mind. For a floating-point number X, this definition matches that of the *quantum* of X. Subnormal numbers are also taken care of.

Let us give a few examples in our toy floating-point system $\mathbb{F}_{2,4,-7,8}$:

- consider first the normal number $\sqrt{2}$; $\text{ulp}(\sqrt{2}) = 2^{-3}$ since
  - $2^0 \leqslant \sqrt{2} < 2^1$,
  - $(e_{\min} = -7) \leqslant (0 = e_{\sqrt{2}})$,
  - $2^{0-4+1} = 2^{-3}$;
- consider now the subnormal number 2^{-12}; $\text{ulp}(2^{-12}) = 2^{-10}$ since
  - $2^{-12} \leqslant 2^{-12} < 2^{-11}$,
  - $\max(-12,-7) = -7$,
  - $2^{-7-4+1} = 2^{-10}$.

The situation is often a bit complex around the integer powers of the radix β^k, $k \in \mathbb{Z}$, elements of $\mathbb{F}$: there is a sharp change in the value of the ulp() function.

Figure 1-2 depicts such a the situation in radix-2.

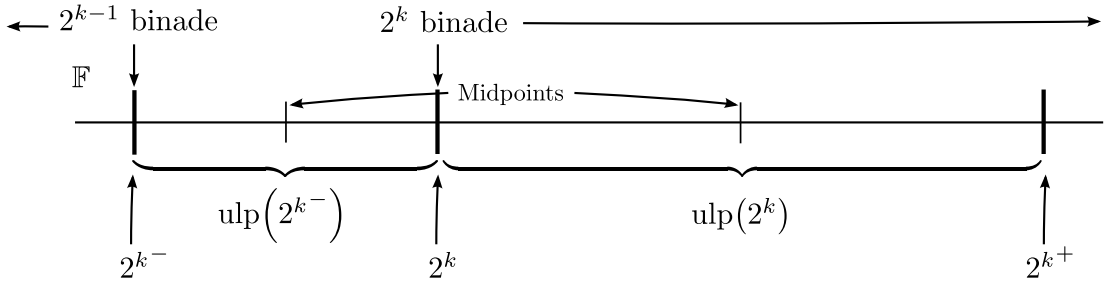

Figure 1-2: For a radix-2 floating-point system, in the neighborhood of a power of 2, the value of the ulp() function experiences a sharp change: it doubles when reaching 2^k "from the left". It remains constant over subsets called *binades*. Also notice how the distance to midpoints also changes making a power of 2 a particular case: its "left" midpoint neighbor is twice as close as its "right" one. This gives an idea as why, in proofs, powers of 2 often have to be dealt with as a separated case.

1.4.2 Some interesting properties

For $X \in \mathbb{F}$ and $x \in \mathbb{R}$, we have the following properties.

Property 1.1. *In radix-2,*

$$|X - x| < \frac{1}{2}\,\text{ulp}(x) \Rightarrow X = \text{RN}(x),$$

where RN() *denotes the round-to-nearest function that will be formally defined in §1.5.2, p. 45.*

Notice that we are reasoning here over ulp(x). This property holds even if X is a power of 2. Depending on the relative position of x to X, the value of ulp(x) "self-adjusts" accordingly in order to make the property hold.

Property 1.2. *For any radix β, and as long as X is not a power of β,*

$$|X - x| < \frac{1}{2}\,\mathrm{ulp}(X) \Rightarrow X = \mathrm{RN}(x).$$

In contrast to the previous property, as we reason over ulp(X), we must exclude the $X = \beta^k, k \in \mathbb{Z}$ case.

Property 1.3. *For any radix,*

$$X = \mathrm{RN}(x) \Rightarrow |X - x| \leq \frac{1}{2}\,\mathrm{ulp}(x).$$

Again, ulp(x) adapts to the right value whatever the value of X is (including a power of β).

Property 1.4. *For any radix,*

$$X = \mathrm{RN}(x) \Rightarrow |X - x| \leq \frac{1}{2}\,\mathrm{ulp}(X).$$

Notice here that the $\frac{1}{2}$ ulp(X) bound is partly an overestimation when X is a normal number and an integer power of β. If both x and X are positive (the both negative case is similar) and X is an integer power of β, if $x \leqslant X$ then $X = \mathrm{RN}(x) \Rightarrow |X - x| \leqslant \frac{1}{4}\,\mathrm{ulp}(X)$.

1.4.3 Binades and the ulp() function

A binade is the interval between two consecutive powers of 2. More precisely:

$$[2^k, 2^{k+1}), \text{ where } k \in \mathbb{Z}, \text{ is a binade.}$$

Notice how we exclude the upper bound from the interval. The reason for it will become obvious in the sequel.

We will often refer to a binade by its lower bound or even only by the exponent of this lower bound: "the 2^4 binade" expression or even shorter, "the 4-binade" expression both denote the $[2^4, 2^5)$

binade. Conversely, 4 will be said to be the exponent of the 4-binade.

If $2^k \leqslant x < 2^{k+1},\ x \in \mathbb{R},\ k \in \mathbb{Z}$, we will say that "$x$ belongs to the 2^k binade" or that "k is the binade of x".

From that we can define a *binade*() function. If we denote by $\lfloor x \rfloor$ the value of the floor(x) function, binade(x) $= \lfloor \log_2(x) \rfloor$. This definition is compatible with the exclusion of the upper bound of the interval. The function is defined over the nonzero positive real numbers. It can be extended to the nonzero negative numbers as well if applied to abs(x) instead of x.

For a $\mathbb{F}_{2,p,e_{\min},e_{\max}}$ floating-point system, the elements of $\mathbb{F}$ that belong to the same 2^k binade are evenly spaced (as depicted in Figure 1-3 for our toy example). If we generalize the notion of binade to that of "β-ade" (as interval $[\beta^k, \beta^{k+1}),\ k \in \mathbb{Z}$), this even spacing property of floating-point numbers in radix β inside each "β-ade" is conserved.

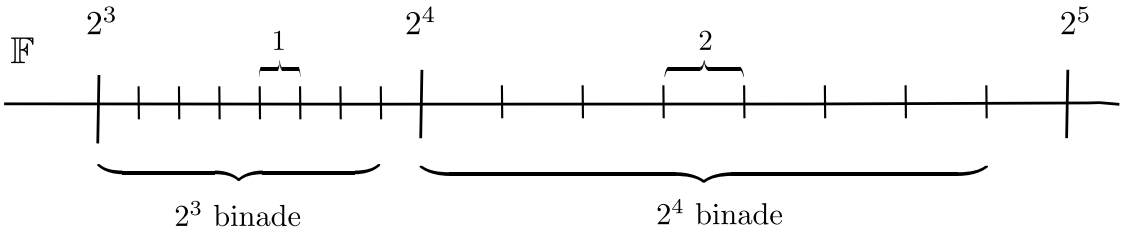

Figure 1-3: Elements of our $\mathbb{F}_{2,4,-7,8}$ toy floating-point format spanning two binades, 2^3 and 2^4, are represented here. Notice how the distance between the elements of the same binade is constant and doubles when we move from one binade to the next. In our examples, for binade 2^k it computes as $2^{k-4+1} = \text{ulp}(2^k)$. For binade 2^3, $\text{ulp}(2^3) = 1$ and for binade binade 2^4, $\text{ulp}(2^4) = 2$.

As stated before, the ulp() function is also defined for the interval spanned by the subnormal numbers $[0, \beta^{e_{\min}})$. The distance between two successive numbers is constant over this interval and its value is $\beta^{e_{\min}-p+1}$. Nevertheless, this interval is not a binade. It spans over an infinity of them!

Notice that while the value of the ulp() function is constant over a binade, this not the case for the relative error $\dfrac{\text{ulp}(X)}{X}$, with $X \in \mathbb{F}$. For normal numbers, it decreases from 2^{-p+1}, for the smallest element of the binade, to almost 2^{-p+2}, for the largest one.

In the subnormal numbers realm considered as a whole, $\dfrac{\text{ulp}(X)}{X}$ decreases from 1 (!), for the

smallest nonzero subnormal number, to almost 2^{-p+1}, for the largest one.

1.4.4 Converting error in ulps to/from relative error

Let us see how absolute errors in ulps help us to build a bridge between absolute and relative errors. Notice that we will have to think in terms of bounds since the value of the ulp() function is the same for all the numbers of the same binade whereas their absolute value continually varies.

Assume first that $X \in \mathbb{F}$ approximates some $x \in \mathbb{R}$ and $x \in [\beta^e, \beta^{e+1})$.

Let us examine first the conversion from the error in ulps to the relative error, assuming that $\epsilon_{abs} = |X - x|$.

We can always rewrite this *absolute* error in ulp terms as: $\epsilon_{abs} = \alpha \times \text{ulp}(x)$.

Assuming no underflow, we can write: $\alpha \times \text{ulp}(x) = \alpha \times \beta^{e-p+1}$.

From our initial assumptions, we have $|x| \geq \beta^e$.

Hence: $\left|\frac{X - x}{x}\right| = \left|\frac{\alpha \times \beta^{e-p+1}}{x}\right| \leq \frac{\alpha \times \beta^{e-p+1}}{\beta^e} = \alpha \times \beta^{-p+1}$.

Let us examine how it works the other way around, assuming that $\epsilon_{rel} = \left|\frac{X - x}{x}\right|$.

Assuming again no underflow, we can state that $\frac{|X - x|}{\beta^{e+1}} \leq \left|\frac{X - x}{x}\right| = \epsilon_{rel}$.

Hence: $\frac{|X - x|}{\beta^{e+1}} \times \beta^{e-p+1} \leq \epsilon_{rel} \times \beta^{e-p+1} = \epsilon_{rel} \times \text{ulp}(x)$.

Finally: $|X - x| \leq \epsilon_{rel} \times \beta^p \times \text{ulp}(x)$.

Expressing errors in ulp's is the usual way for floating-point arithmetic operations and functions. Relative errors are much easier to deal with when performing error calculations. Being able to switch easily between both representations is therefore very convenient. But one must remind that some information is lost along the way. Hence moving back and forth between error in ulps and relative errors is not advisable in error analysis.

1.5 Rounding

Given the importance of this operation (in general, when dealing with floating-point numbers, and in this work as well), we feel compelled to enter into more details about it and introduce some notions that are closely associated with it.

1.5.1 Rounding functions

Rounding is a routine operation when dealing with finite number representation, of which floating-point numbers is a particular case.

To start with, many numbers often used in computations (e.g. constant π) do not have an exact representation in the floating-point numbers.

Moreover, the result of an operation or of the evaluation of a function, even if performed over the floating-point numbers, will often be a value that can not be exactly represented by a floating-point number.

In all cases, one will have to replace them by some approximating floating-point value. This operation is called *rounding*. Let us denote it by the $\circ$ symbol. If $x \in \mathbb{R}$ is some number not representable in a given floating-point system $\mathbb{F}$, $X = \circ(x)$ will be the floating-point number resulting from rounding. Of course, if x is representable in $\mathbb{F}$ then $X = \circ(x) = x$;

Historically, the $\circ$ operation was not always fully specified and could not technically be called a function. For a given x and $\mathbb{F}$ the result could vary from an implementation to another. If several numbers $x_1, x_2, x_3 \ldots, x_n$ were considered the order of $\circ(x_i)$ and x_i and the relative error $\left|\frac{\circ(x) - x}{x}\right|$ could vary in inconsistent ways.

That is why the advent of IEEE 754-1985 helped to level the ground, in particular with the concept of *rounding modes*. It allowed for an exact specification of how any numerical value is rounded to a finite (well, including here the $-\infty$ and $+\infty$ values) floating-point number. Rounding modes are functions and we will stick to the convention of denoting them by $\circ$.

1.5.2 Some definitions

Many rounding modes can be imagined. Four appear in the IEEE 745-2008:

- round toward $-\infty$ or "round down": $\mathrm{RD}(x) = \max_{X \in \mathbb{F}}(X \leqslant x)$; $\mathrm{RD}(x)$ is the largest floating-point number (possibly $-\infty$) less than or equal to x;

- round toward $+\infty$ or "round up": $\mathrm{RU}(x) = \min_{X \in \mathbb{F}}(X \geqslant x)$; $\mathrm{RU}(x)$ is the smallest floating-point number (possibly $+\infty$) greater than or equal to x.
- round toward 0:
  - if $x \geqslant 0$: $\mathrm{RZ}(x) = \mathrm{RD}(x)$,
  - if $x \leqslant 0$: $\mathrm{RZ}(x) = \mathrm{RU}(x)$;

  $\mathrm{RZ}(x)$ is the closest floating-point number to x that is no greater in magnitude than x;
- round-to-nearest:
  - $\mathrm{RN}(x) = X \in \mathbb{F}$ such that $|X - x|$ is minimal; then such X is unique;
  - if there are two consecutive elements of $\mathbb{F}$, X_l and X_u, such that $|X_l - x| = |X_u - x|$, then a *tie-breaking rule* is necessary to select which of the two, X_l or X_u , will be returned.

The different modes are illustrated in Figure 1-4.

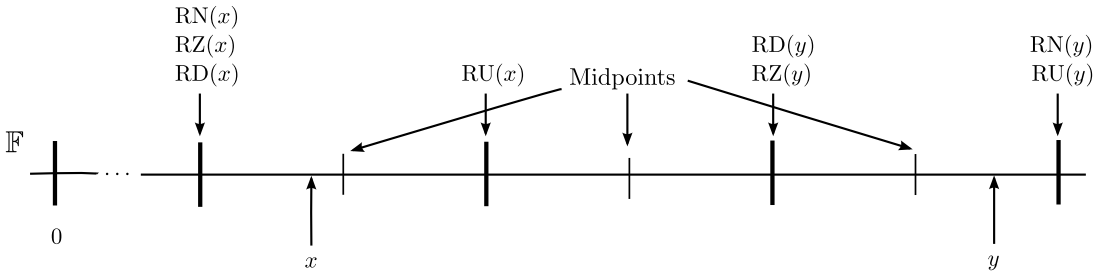

Figure 1-4: The four rounding modes of the IEEE 754-2008 standard [47] are illustrated here. We assume here that both x and y are positive members. Midpoints are not elements of $\mathbb{F}$. They are only shown here to justify the choices for $\mathrm{RN}(x)$ and $\mathrm{RN}(y)$.

For reasons that will soon become obvious, round-to-nearest is of special interest to us. This is why, and because the round-to-nearest mode is not fully specified without it, we will go into more details about the tie-breaking rules. The IEEE 754-2008 standard [47] has chosen the so called *round-to-nearest, tie-break to even* rule as its default mode. It works as follows: since X_l and X_u are consecutive numbers, the integral significand corresponding to their representation [18] are alternatively even or odd (or the other way round). In case of a tie, it is the number with the **even** integral significand that will be returned. The standard makes its availability mandatory for both radix-2 and radix-10 implementations.

[18] This is also true for subnormal numbers.

This is not the only possible choice. One could have taken the symmetrical rule (*round-to-nearest odd*). The IEEE 754-2008 standard [47] defines another tie-breaking rule called *round ties to away*: in the case of a tie, the value with the largest absolute value is returned. The standard make its availability mandatory for radix-10 implementations. It is not the case for binary implementations as this rule is often used in financial computations, the main target of radix-10 implementations.

In the sequel, when we will refer to the "round-to-nearest mode" denoted by RN()), it will always be in its "round-to-nearest even" version, unless otherwise stated of course.

Not any idea of tie-breaking rule that pops to mind can be a useful one. Some properties may be desirable such as symmetry ($\mathrm{RN}(-x) = -\mathrm{RN}(x)$), the lack of statistical bias, the ease of implementation, and reproducibility.

1.5.3 Rounding function properties

For any of the given rounding modes, it is easy to see that a rounding function defined over $\mathbb{R}$ is a step or staircase function. The value of the function is constant over some interval and suddenly changes at or immediately beyond the interval bounds. We will call these sudden change points *rounding breakpoints* or more briefly *breakpoints*. Notice that breakpoints may (in the case of rounding to both infinities or to 0) be elements of $\mathbb{F}$ or may not (in the case of round-to-nearest). As we have seen, in the latter case, they are called midpoints. We can define the midpoint function as

$$\mathrm{midpoint}(Y, Y^+) = \frac{Y + Y^+}{2}.$$

where Y and Y^+ are numerical values floating-point numbers (i.e. not a special datum as NaN) and Y^+ is the successor of Y. Though $\mathrm{midpoint}(Y, Y^+)$ is a rational number (since Y and Y^+ are), it is not an element of $\mathbb{F}$.

An important and instrumental property is that, for all of the IEEE 754-2008 [47] defined modes, the $\circ$ rounding function is a *non-decreasing function* (i.e., $\forall x, y \in \mathbb{R}$, $x \leqslant y \Longrightarrow \circ(x) \leqslant \circ(y)$).

Another property is attached to the round toward 0 or to the round-to-nearest (provided a symmetric tie-breaking rule is adopted) rounding modes is that the symmetries of the function are preserved by the rounding function (i.e., $f(-x) = -f(x) \implies \circ(f(-x)) = -\circ(f(x))$.

It is also interesting to investigate how rounding modes and the relative error due to rounding interact.

When rounding takes place a relative error $\epsilon_{re\ell}$ is committed:

$$\epsilon_{re\ell} \;=\; \left|\frac{x \;-\; \circ(x)}{x}\right|. \tag{1.2}$$

In the normal range and for the round-to-nearest mode

$$\epsilon_{re\ell} \leqslant \frac{1}{2}\beta^{1-p}.$$

In the normal range and for the "directed" rounding modes (toward infinities, toward 0)

$$\epsilon_{re\ell} < \beta^{1-p}.$$

Plotting the value of $\epsilon_{re\ell}$ produces a sawtooth pattern since $\epsilon_{re\ell}$ grows and shrinks as x moves between breakpoints, falling down to 0 each time x can be exactly represented by a floating-point number.

When it comes to subnormal numbers, the relative error can become very large: close to 1! One can possibly comfort herself with the absolute rounding error, if this is what matters for her application:

- $\epsilon_{abs} = |x - \mathrm{RN}(x)| \leqslant \frac{1}{2}\beta^{e_{\min}-p+1}$ (round-to-nearest mode);
- $\epsilon_{abs} = |x - \circ(x)| < \frac{1}{2}\beta^{e_{\min}-p+1}$ (where $\circ$ is one of the directed rounding modes).

We can now combine the two, relative and absolute, errors for some correctly rounded operation $\top$. Let us assume that we have $A, B, Y \in \mathbb{F}$ such that $Y = \circ(A \top B)$. We have

$$Y = (A \top B)(1 + \epsilon) + \epsilon', \tag{1.3}$$

where:

- $|\epsilon| \leqslant \frac{1}{2}\beta^{1-p}$ and $|\epsilon'| \leqslant \frac{1}{2}\beta^{e_{\min}+1-p}$ in round-to-nearest mode:
- $|\epsilon| \leqslant \beta^{1-p}$ and $|\epsilon'| \leqslant \beta^{e_{\min}+1-p}$ in directed rounding modes;
- $\epsilon = 0$ or $\epsilon' = 0$ [19].

The bound on ϵ is often called the *unit roundoff*, *machine epsilon* or even *machine precision* and is often denoted by $\mathbf{u}$.

It relates with ulp as defined in §1.4, p. 38:

[19] If Y is in the normal range, then $\epsilon' = 0$; If Y is a subnormal number, then $\epsilon = 0$.

- $\mathbf{u} = \frac{1}{2}\text{ulp}(1) = \frac{1}{2}\beta^{1-p}$ in round-to-nearest mode;
- $\mathbf{u} = \text{ulp}(1) = \beta^{1-p}$ in directed rounding modes.

This notion is the corner stone of the *Standard Model* of floating-point arithmetic that is widely used for the analysis of numerical algorithms (e.g. see Nicholas J. Higham's book *Accuracy and Stability of Numerical Algorithms* [44]). The model is based, for any arithmetic operation with floating-point numbers $\top \in \{+, -, \times, /\}$, for any rounding mode $\circ \in \{RN, RD, RU, RZ\}$, for all $A, B \in \mathbb{F}$ and as long as $A \top B$ does not underflow or overflow, on the following expression:

$$\circ(A \top B) = (A \top B)(1 + \epsilon_1) = \frac{A \top B}{1 + \epsilon_2}.$$

with $|\epsilon_1| \leqslant \mathbf{u}$ and $|\epsilon_2| \leqslant \mathbf{u}$.

As it has been already stated, a wealth of results has been gathered through this approach and the cited book is acclaimed as a spectacular achievement in this respect.

But it has also been shown, as in [50], that sometimes better results (sharper bounds or even optimal, in the sense of actually attainable) could be obtained if one followed a more precise approach and model, taking into account the minute details of floating-point numbers properties, including the peculiar properties of some "new" basic operations as the Fused Multiply-Add $(x + y \times z)$ with a single final rounding[20].

Remark 1. From time to time one can bump into the *faithful rounding* expression. We feel some clarification is needed at the end of this section devoted to rounding. Let us call y the exact result of some operation or function. Faithful rounding refers to a property of an implementation guaranteed to deliver either $\text{RD}(y)$ or $\text{RU}(y)$, without any further indication. By extension we call such a returned value a *faithful value* and if all operations verify that property, we will speak about a *faithful arithmetic*. But by no means can it be called a rounding mode in the sense of those described above since the result of faithful rounding is not unequivocally defined: it is not a function. It does not mean that this property is worthless and that no reasoning can be made about it.

1.5.4 Fused multiply-add

We will conclude this section devoted to rounding functions by a few words about the *Fused Multiply-Add* machine instruction (FMA).

[20] More details to come about FMA.

1.5.4.1 From MAC to FMA

There are many kinds of computation where the sequence of operations $A + B \times C$, $A, B, C \in \mathbb{F}$ is often performed. One important such application field is signal processing [59]. But this sequence also happens often in scientific computing at large (e.g. dot products, polynomial evaluation with Horner's scheme).

A natural idea among processor architects has been to create a specialized instruction that would perform the sequence at a faster pace than the consecutive executions of a multiplication and an addition, says, almost as quickly as the multiplication. This instruction is known as MAC (for Multiply ACcumulate):

$$a \leftarrow a + b \times c.$$

The name proceeds from a frequent use where successive bc products with different data accumulate into a. In MAC, rounding is performed twice, after each elementary operation. The principal enhancement brought along is an execution speedup.

Later on, a new instruction was introduced in 1990 by IBM in the first scion of its general purpose processors POWER family [75]. Only one rounding takes place here: multiplication is computed (as if) exactly, followed by an (as if) exact addition, completed by a single rounding.

1.5.4.2 FMA and FMAC

The instruction was called *Fused Multiply Add*. Since its introduction most of the other manufacturers have appended it to the instruction set of their own processors. Two variants exist:

- $r \leftarrow a + b \times c$;
- $a \leftarrow a + b \times c$.

In the first one, the result is assigned to a fourth register. In the second one, the result of the multiplication is added with the current value of the left operand. The left operand is eventually overwritten with the final result. This form is also called *Fused Multiply-Accumulate*.

When the instruction made it to the IEEE 754 standard, in its 2008 version, it was specified with the single terminal rounding. The standard did not consider the different variants: what matters to is the single and final rounding.

This specification caught much attention among the numerical computing specialists. It was quickly discovered that, beyond the prosaic idea that the lesser the rounding, the better the result,

new possibilities were opened and old problems or processes could be revisited with opportunities for improvement.

For instance a strikingly short and improved version of the Dekker's multiplication algorithm (see Algorithm 4, p. 72) used, under some conditions, to compute the exact product of two floating-point numbers, A and B, has been devised with the help of FMA.

Provided the product $A \times B$ does not overflow and $e_A + e_B \geqslant e_{\min} + p - 1$, where e_A and e_B are the exponents of A and B, Algorithm 1 computes their exact product as the unevaluated sum of two floating-point numbers Y_1 and Y_2.

Algorithm 1 2MultFMA Algorithm

```
/* Assume a binary floating-point system F_{β,p,emin,emax} and */
/* that the conditions on exponents and overflow are met. */
function 2MultFMA(A, B)
    Y1 ← RN(A × B)
    Y2 ← RN(A × B − Y1)
    /* Y2 is the exact difference between A × B and Y1. */
    return Y1, Y2
end function
```

This algorithm only requires two operations whereas Dekker's Algorithm needs seventeen. Furthermore, it works for any radix and precision. Although presented here it for round-to-nearest mode, it works for the other rounding modes as well.

1.6 Correct rounding

In this section we will elaborate a bit on correct rounding issues in order to be able to discuss at more length the Table Maker's Dilemma.

1.6.1 What is correct rounding?

First of all, the notion of correct rounding is relative to a rounding mode (e.g. rounding-to-nearest, tie-breaking to even) and to a floating-point system: there is no such thing as correct rounding in general[21]. Secondly, this property refers to an implementation f_{imp} of a mathematical function f, not to f itself: it does not actually make sense when one speaks about a "correctly rounded exponential". But, as this is a common expression, we will keep using this misnomer. An implementation $f_{imp} : \mathbb{F}^m \mapsto \mathbb{F}^n$ of an operation or a function $f : \mathbb{R}^m \mapsto \mathbb{R}^n$ is *correctly rounded* for rounding mode $\circ$, if for any input, $X = (X_1, \ldots, X_m) \in \mathbb{F}^m$, for any component Y_i of the output $f_{imp}(X) = Y = (Y_1, \ldots, Y_n) \in \mathbb{F}^n$, either:

[21] Unless we possibly refer to correct rounding for all four rounding modes (RN, RD, RU, RZ) and some implicit $\mathbb{F}$.

- Y_i is a non-numerical value (e.g NaN, an infinity...) for some mathematically sound reason (e.g. $1/\sqrt{0} = +\infty$);

- Y_i is set to some numerical value by definition or by convention (e.g. $\mathrm{pow}(-1, \pm\infty) = 1$);

- if $y = f(X)$, $y = (y_1, \ldots, y_n) \in \mathbb{R}^n$, then $Y_i = \circ y_i$, $i \in \{1, \ldots, n\}$.

The point here is, in the third case, that rounding refers to the mathematically exact value of the function evaluation. It does not imply that one absolutely has to compute this exact value, which very often is out of her reach, and then apply the rounding function. The requirement is that the returned floating-point value is "as if" it had been computed by rounding the exact value. In that case, which is the most common since approximations are used to compute the value intended to be rounded, the developer of a correctly rounded function implementation must make sure this is the case and also have good arguments to support her claims.

Let us give a tricky example of this third case. We want to compute correctly rounded, in round-to-nearest mode, images by function

$$\begin{array}{rcl} f : x & \mapsto & 1/\sqrt{x} \\ \mathbb{F}_{2,53,-1022,1023} & \mapsto & \mathbb{F}_{2,53,-1022,1023} \\ \texttt{binary64} & & \texttt{binary64} \end{array}$$

for the value

$$a = 1.6510283997447916526368771883426234126091003417968 75.$$

While it may not seem absolutely obvious at first glance, a is exactly representable in the `binary64` format and it's value is $3717785442934375/2^{51}$. In the sequel of this example, we will display binary values since they make the problems easier to spot and explain. The target format has 53 bits of precision.

We provide below a high (512 bits) precision of $f(a)$, and complement it with the rounded-to-nearest value in 53-bit precision and the rounded-up value for the same precision:

$$
\begin{aligned}
& \qquad\quad 1 \qquad\quad 2 \qquad\quad 3 \qquad\quad 4 \qquad\quad 5 \qquad\quad 6 \\
& \qquad 12345678901234567890123456789012345678901234567890123456789012\downarrow 4567890 \\
f(a) \; &= 0.11000111001110111101000010001100010100011000010010101011111111111 \; \backslash \\
& \qquad 11111111111111111111111111111111111111111111111010001101000 \; \backslash \\
& \qquad 1010111101101011110\ldots \\
\mathrm{RN}(f(a)) \; &= 0.11000111001110111101000010001100010100011000010010101 \\
\mathrm{RU}(f(a)) \; &= 0.11000111001110111101000010001100010100011000010010110
\end{aligned}
$$

The position of the 53rd bit in the high-precision computed value is tagged with a down arrow.

In a naive approach, one could think that all is needed is to compute an approximation $\widehat{f}$ with an error less than 2^{-55} in order to have bit-54 correct. But this is not enough. One must remember that, by definition, when an approximation is computed, all we have is an upper bound on the difference between the approximation and the exact value. We do not know what the distance to the exact value exactly is. We do not even know which, of the exact value and the approximation, is the largest. For the contrary to be true, one should know the exact value: in that case, what would be the point of computing an approximation at all?

We are constantly forced to consider the worst case, if we are to give any guaranty. In this particular case, with $\epsilon \leqslant 2^{-55}$, we have to suppose, that the approximating value $\widehat{f}(a)$ can be as much as 2^{-55} in excess of the exact value. What will then happen, as far as rounding is concerned? If we add 2^{-55} to the exact value, this will flip bit-55 to 0 with a carry that will propagate to bit-54 and, in turn, flip it to 1. The approximation then is

$$
\begin{aligned}
& \qquad\quad 1 \qquad\quad 2 \qquad\quad 3 \qquad\quad 4 \qquad\quad 5 \qquad\quad 6 \\
& \qquad 12345678901234567890123456789012345678901234567890123456789012\downarrow 4567890 \\
\widehat{f}(a) \; &= 0.11000111001110111101000010001100010100011000010010101 1 \\
\mathrm{RU}(f(a)) \; &= 0.11000111001110111101000010001100010100011000010010110
\end{aligned}
$$

The application of the tie-break even rule will take us to the rounded-up value instead of the correctly rounded-to-nearest one. The same process repeats itself if we compute an approximation with an accuracy of 2^{-56} as we have to assume again that the approximation can be in excess by 2^{-56} to the exact value: bit-56 is flipped to 0 with a carry propagation that will cascade up to bit-54,

with the same consequence as to rounding as above.

How far, in accuracy terms, does one need to go? She must target for an approximation with an error bound that, added to the exact value, will not cause a cascade of carry propagation that will eventually flip bit-54 to 1. And this will take an accuracy as high as $\epsilon \leqslant 2^{-113}$. If this is the case, bit-54 will have the same value in the approximated and exact values. Rounding rules applied to the approximation will then yield the correctly rounded value.

We must point out here that all our reasoning is based on our knowledge of the exact value (or, more precisely, of an extra-accurate approximation). It cannot be the basis for the actual approximation process. To conclude this example, we observe that, fortunately, computing the exact value is not needed. We can do "as if" we had it and yet perform a correct rounding. We have seen that, for this particular value a, it was possible to find an approximation accuracy for which we could blindly round the approximation and yet get the correct rounded value of the exact one. But working out an approximation that guarantees a correct rounding for all possible inputs is by no means an easy task.

1.6.2 Why does correct rounding matter?

We, at the AriC team [3], firmly believe that being able to provide correctly rounded functions is greatly beneficial to those who compute with them. This is not a matter of hair-splitting concern for pervert theorists. It has a lot of practical implications.

Even from the most very basic perspective, it is a way, in the round-to-nearest mode, to provide the best possible accuracy when function f is evaluated with argument X. If the result is in the normal range:

- the absolute error is bounded by $\frac{1}{2}\text{ulp}(f(X))$ (this one is even true for subnormal numbers);
- the relative error is bounded by $\mathbf{u} = 2^{-p}$ (but that does not stand for subnormal numbers).

Generally, in real-world programs, a function is not used all by itself. It is combined with other functions and operations in complex sequences. Accuracy possibly decreases from the very first evaluation and the situation worsens as computation goes on. A good starting point helps to mitigate this accuracy fraying process with, hopefully, the lesser the garbage in, the lesser the garbage out.

Another important issue is software portability: users expect that a program will yield the same results whatever the platform it is run on (at least if they can work with the same precision).

There are many issues that can antagonize this expectation (non-deterministic parallel execution effects, compiler options, etc.). Those are very difficult to control. Using mathematical libraries that provide a consistent level of rounding quality across all platforms can relieve the burden for those who try to achieve portability. This consistent level can be nothing short of correct rounding.

But there is more. Nowadays, the dedicated programmer not only expects that his code will yield the best possible result but he also wants to have more than a vague feeling about what its accuracy is. This is the kind of insight that error analysis provides but it can only work if walking on firm grounds. Having correctly rounded functions allows for the kind of mathematical reasoning that is needed here, especially for formal proofs.

Correct rounding also greatly eases the implementation of interval arithmetic and makes it more useful. Even if, in this respect, it is not the only technique to use, it delays the moment where extravagantly spread apart and useless interval bounds are returned. More generally, any computation where tight bounds around the exact result of a sequence of floating-point operations are needed benefits from correct rounding. This is not only true for accurate modeling of unstable systems over large time spans [60] but also when computations are used for mathematical proofs [56] [42].

One could object that a function guaranteed to be accurate to, say, the last two bits would give the same ability to provide rigorous error bounds. That is true. But certifying rounding is always complex, whatever the accuracy is. Furthermore, we now know for a fact [21] that it is possible to design and implement a correctly rounded library of elementary mathematical functions without incurring a performance penalty. Why should we satisfy ourselves with low grade tools when top-quality ones are at hand?

Last but not least, other interesting properties (preservation of monotonicity, compatibility with symmetry properties of functions, see §1.5.3, p. 46, for more details) of the rounding functions as IEEE 754 standard defines them for the four modes should not be overlooked.

For all these compelling reasons, at the occasion of its 2008 revision, IEEE 754 standard [47], that already mandated correct rounding for arithmetic operations, made it a recommendation for a large set of elementary functions[22].

[22]The function's list is: e^x, $e^x - 1$, 2^x, $2^x - 1$, 10^x , $10x - 1$, $\ln(x)$, $\sqrt{x}$, $\log_2(x)$, $\log_{10}(x)$, $\ln(1 + x)$, $\log_2(1 + x)$, $\log_{10}(1 + x)$, $\sqrt{x^2 + y^2}$, $1/\sqrt{x}$, $(1 + x)^n$, x^n, $x^{1/n}$, (n is an integer), $\sin(\pi x)$, $\cos(\pi x)$, $\arctan(x)/\pi$, $\arctan(y/x)/\pi$, $\sin(x)$, $\cos(x)$, $\tan(x)$, $\arcsin(x)$, $\arccos(x)$, $\arctan(x)$, $\arctan(y/x)$, $\sinh(x)$, $\cosh(x)$, $\tanh(x)$, $\sinh^{-1}(x)$,$\cosh^{-1}(x)$, $\tanh^{-1}(x)$.

1.6.3 The Table Maker's Dilemma

Let us imagine a magic world, where we have at hand an iterative algorithm that computes an approximation of some function f, at the rate of one correct digit of the exact value (possibly made of an infinity of digits) per iteration. To correctly round-to-nearest the exact value of $f(x)$ to a floating-point of precision p, all we have to do is to perform $p+1$ iterations of our algorithm and decide rounding upon the value of the $(p+1)$-th digit according to tie-breaking rules.

But this is not how things happen in Real Life. In mathematical libraries running on computers, functions are not abstractly (mathematically) computed and then (trivially correctly) rounded as above. It turns out that all we have access to are *approximants* that compute *approximations* of function values and that all we know about them is their accuracy. And one should not confuse, for a value close to 1, having an accuracy of, say, $\epsilon \leqslant 2^{-56}$ and having a guarantee that 55 bits or even 54 bits are correct. We have seen above, on an example, that things can get much complicated: there is a subtle alchemy between the accuracy of the approximant, the exact value of the function and the precision we want to round to. Since we do not know exact values, we have to be ready for the worst. At this point, a legitimate question is: what happens when we round these approximations? How can we ensure, in general, that this yield the same result as the rounding of the exact values?

By no means is this an absolutely new problem. Those who computed the now long gone mathematical tables (as trigonometric or log tables) faced the same kind of issues as they also used approximation techniques to provide rounded values of the functions[23]. Oddly enough, it was not before the end of the 20th century that William Kahan, as he tells us the story in a 2004 text [55], coined the suggestive and colorful expression of "Table Maker's Dilemma" to refer to this problem.

We will elaborate a bit more on this question in Chap. 3, p. 135, since solving it is an important motivation for this part of the presented work. In the sequel, we will especially consider the "round to nearest, tie-break to even" mode but the ideas outlined here apply to any of the other modes. In this particular case breakpoints are midpoints.

1.6.3.1 The TMD in pictures

As stated, the starting point is that we do not compute the mathematically exact value of the function but an approximation. And it is this approximation that we will want to round. Correct

[23] We would have gladly provided references to early documents but, unfortunately, the works of Hipparchus of Nicaea (c.190 – c.120 BCE) and Menelaus of Alexandria (c.70 – 140 CE) dubbed as the first authors of trigonometric tables, did not make it to present time.

rounding in this context is making sure that rounding the approximation yields the same result as rounding the exact value.

Our first case shows that rounding is not only a matter of getting a floating-point number.

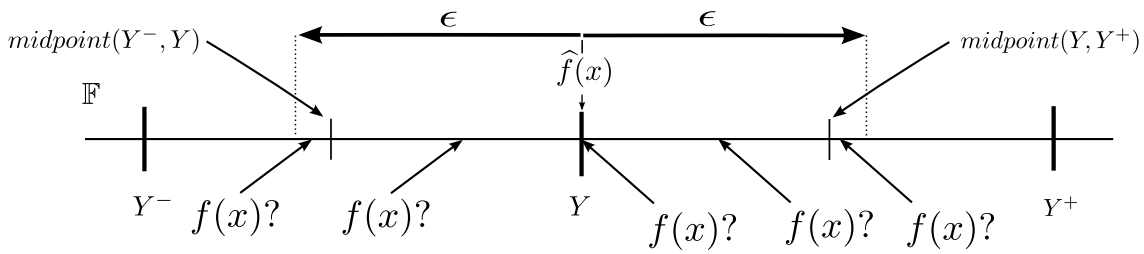

Figure 1-5: In this figure Y denotes a floating-point number, Y^- its predecessor, Y^+ its successor in $\mathbb{F}$; midpoint(Y^-, Y) and midpoint(Y, Y^+) are rational numbers equal to $(Y^- + Y)$ / 2 and $(Y + Y^+)$ / 2 which are not elements of $\mathbb{F}$. The function f is approximated, for some floating-point argument x, by a machine computable function $\hat{f}$ with an error ϵ. Here, $\hat{f}(x)$ can exactly be represented by floating-point number Y. Does $\hat{f}(x)$ require rounding? Cannot we just say RN$(f(x)) = Y$? With such a large value for ϵ (more than half ulp$(\hat{f}(x))$), one has to think twice before returning Y.
In this case, while the approximation is a straight floating-point number, "rounding" is still needed and in this particular proves impossible to perform: we cannot safely pick any of Y^-, Y and Y^+ .

In our second case, illustrated in Figure1-6, $f(x)$ is located "somewhere" in an interval centered on the computed value $\widehat{f}(x)$. In this particular example, rounding is not an issue. The exact value, $f(x)$, lives in the $[Y, \text{midpoint}(Y, Y^+))$ interval. Hence its rounded value is $Y = \text{RN}(f(x)) = \text{RN}(\hat{f}(x))$.

Let us consider now the situation depicted in Figure1-7. Here the exact $f(x)$ value can be smaller or larger than (or even equal to) midpoint(Y, Y^+). Based on the present values of $\widehat{f}(x)$ and ϵ we can not decide whether Y or Y^+ is the correctly rounded value of $f(x)$.

What should we do next? The only reasonable solution (if correct rounding is to be obtained) is to compute a new approximation of $f(x)$ with the another approximant $\widehat{f}^{(1)}(x)$, that allows for a better accuracy (e.g ϵ / 2). Our expectation is that we will reach a situation similar to that exemplified in Figure1-8.

Notice that this process is not guaranteed to terminate. It suffices that $f(x) = \text{midpoint}(Y, Y^+)$ for some $X \in \mathbb{F}$ for it to fall in an infinite loop. This is why it is necessary to make sure that such a situation never happens. We have two options here:

- prove that it is impossible;

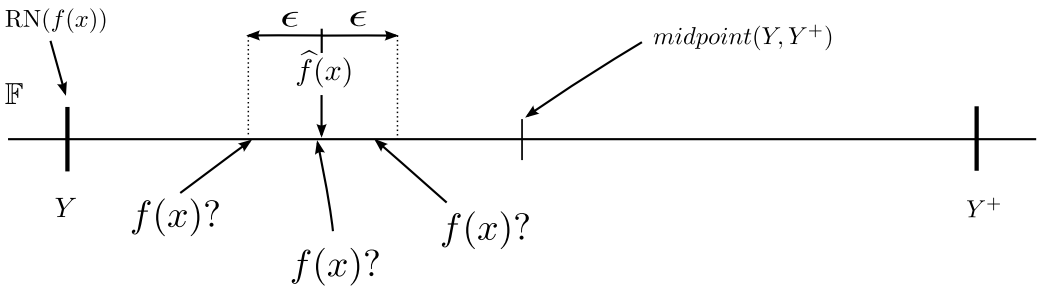

Figure 1-6: In this figure Y denotes a floating-point, number Y^+ its successor in $\mathbb{F}$ and midpoint(Y, Y^+) a rational number equal to $(Y + Y^+)$ / 2, which is not an element of $\mathbb{F}$. Function f is approximated, for some floating-point argument x, by a machine computable function $\hat{f}$ with an error ϵ. The computed value $\hat{f}(x)$ is a rational number that has to be rounded (even if it can be represented by a floating-point number, as seen above). Notice that while we know that $f(x)$ has to be "somewhere" in the $[\widehat{f}(x) - \epsilon, \hat{f}(x) + \epsilon]$ interval we have no clue as to its exact whereabouts. Nevertheless correct rounding is possible here to value Y.

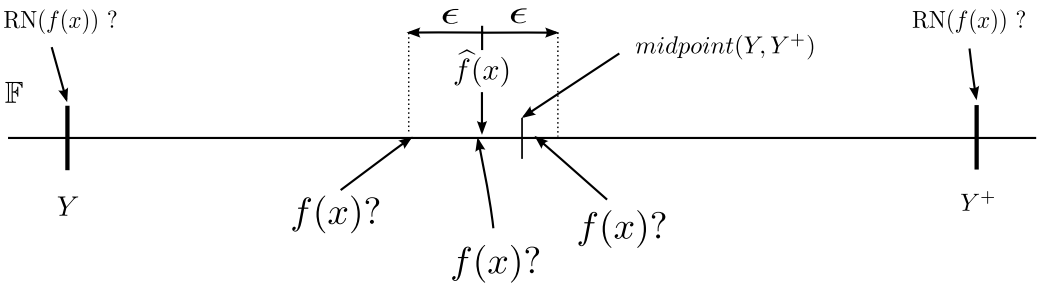

Figure 1-7: In this figure Y, Y^+ and midpoint(Y, Y^+) are the same as in Figure1-6. Function f is approximated, for some floating-point argument x, by a machine computable function $\widehat{f}$ with an error ϵ. The computed value $\widehat{f}(x)$ clearly has to be rounded. Again, if $f(x)$ has to be "somewhere" in the $[\widehat{f}(x) - \epsilon, \widehat{f}(x) + \epsilon]$ interval, we have no idea as to where it actually lives. But this time the interval encompasses midpoint(Y, Y^+). Correct rounding is not possible.

- enumerate all the cases where it happens and deal with them separately.

It is sometimes possible to mathematically assert that the image of any number of the $\mathbb{D} \bigcap \mathbb{F}$ floating-point domain of f (or any other convenient superset) is not a rational number, while midpoint(Y, Y^+), as we recall, always is. This can be done, for instance, by applying Lindemann's Theorem [5] and corollaries for several elementary functions (as exp, log, cos...). They state that:

- the image of any nonzero algebraic number is transcendental for exponential, sine, cosine, tangent, arctangent and hyperbolic functions;
- the logarithm of any algebraic number different from 1 is transcendental too.

If $\nexists\ x \in \mathbb{D} \bigcap \mathbb{F}$ such that $f(x)$ is rational, then we are done with one part of the convergence

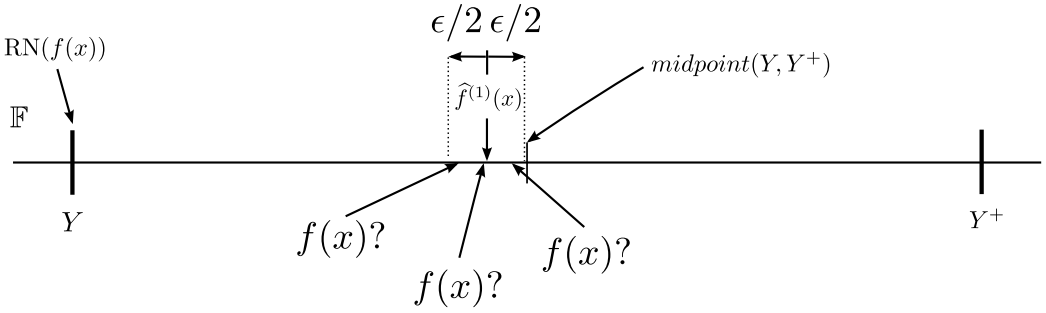

Figure 1-8: In this figure Y, Y^+ and midpoint(Y, Y^+) are the same as in Figure1-6. This figure is a possible sequel to Figure1-7 where function f is now approximated by a machine computable function $\widehat{f}^{(1)}$ with an error $\epsilon/2$. This function is more accurate than the previously used $\widehat{f}(x)$. The computed value $\widehat{f}^{(1)}(x)$ is still a rational number that, in general, does not coincide with any of the numbers of $\mathbb{F}$ and will therefore have to be rounded. Notice that, while $f(x)$ has to be "somewhere" in the $[\widehat{f}^{(1)}(x) - \epsilon/2, \widehat{f}^{(1)}(x) + \epsilon/2]$ interval, we have no clue as to where it actually lives. But this time, as in Figure1-6, all the possible values of $f(x)$ are on "the same side" relatively to midpoint(Y, Y^+) what eventually makes correct rounding (to Y) possible.

impossibility.

If a general result such as Lindemann's can not be obtained, an alternative is to enumerate the rational images (or, if possible, only the midpoint images) and deal with these particular cases in specific branches of the implementation. An example of this kind of characterization can be found in an article by C.-P. Jeannerod, N. Louvet, J.-M. Muller and A. Panhaleux[48].

The other problem is that we must also ensure that either $\widehat{f}(x)$ can be computed to any arbitrary accuracy relatively to $f(x)$ or that we have at hand a family of approximants $\widehat{f}(x)$, $\widehat{f}^{(1)}(x)$, $\widehat{f}^{(2)}(x)\ldots$ that allows for an arbitrary accuracy approximation of $f(x)$.

How far should this process be taken to ensure that $\mathrm{RN}(\widehat{f}(x)) = \mathrm{RN}(f(x))$ is the heart of the Table Maker's Dilemma (TMD).

We should notice that:

1. the elements of the floating-point domain of f where the TMD shows its ugly face are a property of function f itself related to precision p and not one of its approximating functions;

2. the problem does not necessarily happen at high precisions; closeness to a midpoint of an image by f can happen, for any precision, however small, for any value in its floating-point domain ;

3. precision matters a lot as to when points of the domain (or better said, their image by f)

become problematic; we can not say that some floating-point number X_1 is a hard-to-round case for function $f(x)$ in full generality; number X_1 can indeed be a problem when computing $\mathrm{RN}(f(X_1))$ to an image in precision p_1 and totally innocuous if the image is in precision $p_2 \neq p_1$; the respective values of p_1 and p_2 do not matter either: if, for instance, $p_1 < p_2$ and $f(X_1)$ is hard-to-round to precision p_1, one can not automatically conclude that it will also be hard to round to precision p_2 on the grounds of some "hard once, hard forever on" rule.

1.6.3.2 Hardness-to-round: the symptoms

Beyond this pictorial presentation, we can also think about the problem in terms of the precision and accuracy of the approximant. We will still deal, in what comes next, with the "round to nearest, tie-break to even" rounding mode and with a binary representation of the numbers (for both the exact value and its approximation).

Let us suppose that $y^{(1)}$ is the infinite significand of the image of some $X^{(1)} \in \mathbb{F}_{2,p,e_{\min},e_{\max}}$ by function f. This image has an exponent $e_{y^{(1)}}$. Let use suppose the same relationship exists between $y^{(2)}$, $X^{(2)}$ and $e_{y^{(2)}}$. We write $y^{(1)}$ and $y^{(2)}$ under a form, where y_n are the p-most significant bits and u are bits whose value is of no interest:

$$y^{(1)} = y_0.y_1y_2\dots y_{p-1}\overbrace{0111111111111111}^{k\text{ bits}}0uuuuuuuuu\dots$$

and

$$y^{(2)} = y_0.y_1y_2\dots y_{p-1}\overbrace{1000000000000000}^{k\text{ bits}}1uuuuuuuuu\dots.$$

We can easily see that $y^{(1)}$ and $y^{(2)}$ are very close to a midpoint. The larger the length-k string is, the closer we are to a midpoint.

Let us define Y as:

$$Y \;=\; y_0.y_1y_2\dots y_{p-1}.$$

Here $Y\,\beta^{e_{y^{(1)}}}$ and its successor $Y^+\,\beta^{e_{y^{(1)}}}$ in precision p. Furthermore, we assume that $Y\,\beta^{p-1}$ is odd ($y_{p-1} = 1$) and $Y^+\,\beta^{p-1}$ is even ($y^+_{p-1} = 0$).

Obviously we have

$$Y\,\beta^{e_{y^{(1)}}} \;=\; \mathrm{RN}\left(y^{(1)}\right)$$

and

$$Y^{+}\,\beta^{e_{y^{(2)}}} \;=\; \mathrm{RN}\left(y^{(2)}\right).$$

Suppose now that we have at hand an approximant $\widehat{f}^{(1)}$ that is accurate up to 2^{-p-k+1} and yields $p+k$ digits. With this approximant, we compute $\widehat{f}^{(1)}(X^{(1)}) \;=\; \widehat{y}^{(1)}$, an approximation of $y^{(1)}$, whose significand is:

$$\widehat{y}^{(1)} \;=\; \widehat{y}_0.\widehat{y}_1\widehat{y}_2\ldots\widehat{y}_{p-1}1\overbrace{0000000000000000}^{k\text{ bits}}.$$

This value is an approximation of $y^{(1)}$ with an error equal to 2^{-p-k+1}, the maximum allowed by the accuracy of $\widehat{f}^{(1)}$. From this approximation alone we could erroneously conclude that $\mathrm{RN}\left(y^{(1)}\right) \;=\; Y^{+}\beta^{e_{y^{(1)}}}$, since $y^{+}_{p-1}=0$ (break tie to even rule). Remember that, when we compute an approximation, all we know is $\widehat{y}^{(1)}$ and $\epsilon \;=\; 2^{-p-k+1}$. We have no precise idea (to put it vaguely) of what $y^{(1)}$ is nor do we know anything better than $\left|\widehat{y}^{(1)} \;-\; y^{(1)}\right| \leqslant \epsilon$.

We see here that it will take to compute an approximant $\widehat{f}^{(2)}(X^{(1)})$ with an accuracy better than or equal to 2^{-p-k} yielding at least $p+k+1$ digits for the new value of $\widehat{y}^{(1)}$ be able to correctly round it to $Y\beta^{e_{y^{(1)}}}$.

The same applies with the second case where we could compute an approximation of $y^{(2)}$ with $\widehat{f}^{(1)}$ whose significand is:

$$\widehat{y}^{(2)} \;=\; \widehat{y}_0.\widehat{y}_1\widehat{y}_2\ldots\widehat{y}_{p-1}0\overbrace{1111111111111111}^{k\text{ bits}}.$$

This value is an approximation of $y^{(2)}$ with an error equal to 2^{-p+1-k}, the maximum allowed by the accuracy of $\widehat{f}^{(1)}$. From this approximation we could as erroneously well conclude that $\mathrm{RN}\left(y^{(2)}\right) \;= Y\beta^{e_{y_1}}$.

Again, it will take to compute an approximant $\widehat{f}^{(2)}(X^{(2)})$ with an accuracy better than or equal to 2^{-p-k} yielding at least $p+k+1$ digits be able to correctly round the new value of $\widehat{y}^{(2)}$ to $Y^{+}\beta^{e_{y^{(2)}}}$.

While these explanations may help to understand the problem, one must remember that the exact infinite significand values are beyond our reach. One can not make at leisure the above suggested

adjustments in precision and accuracy. The dilemma is still here.

1.6.3.3 Gauging hardness-to-round

Before trying to solve the dilemma, we are going first to try and characterize it. We will first define the notion of *hardness-to-round*.

We have seen that k, the length of the peculiar sequences we depicted above, is a key factor to assess the closeness of the values of function f to midpoints. Similarly, other peculiar sequences can be described for the closeness to breakpoints for other rounding modes. This conditions the accuracy that $\widehat{f}(x)$ approximants should have in relation to $f(x)$ to be able to correctly round it.

Put in another way, we compute $f(X)$, $X \in \mathbb{F}$ (or a subdomain of $\mathbb{F}$) either:

1. $f(X)$ is a midpoint of $\mathbb{F}$;
2. $f(X)$ has one (if next to the bounds) or two neighboring midpoints.

In the latter case, $f(X)$ has a closest midpoint neighbor (we can pick any of the two neighboring midpoints if $f(X)$ has an exact floating-point value). If we consider the normalized significands of $f(X)$ (possibly infinite : $y_0.y_1\ldots$) and that of this closest midpoint ($\mu_0.\mu_1\ldots\mu_p$) there exists $m_X \in \mathbb{N}$ such that :

$$\frac{1}{2^{m_X+1}} \;<\; |y_0.y_1\ldots \;-\; \mu_0.\mu_1\ldots\mu_p| \;\leqslant\; \frac{1}{2^{m_X}}. \tag{1.4}$$

Let us consider now[24] m as the maximum value over all the m_X, for all the elements of $\mathbb{F}$ (at least for those whose image has a numerical value and after a clean up of the elements of the domain whose image is an exact midpoint).

This m is called the hardness-to-round-to-nearest of function f with respect to $\mathbb{F}$ (or a subdomain thereof). This is also the maximum length of a "$011\ldots110$" or "$100\ldots001$" string starting from the p-th position of the infinite significand of $f(X)$.

This definition generalizes to other rounding modes. Hardness-to-round "in general" refers to the largest value for all rounding modes. However, in the context of this work, since we will focus most of the time on the rounding-to-nearest mode, hardness-to-round will only refer to this mode.

Floating-point numbers of the domain whose image by f exhibits near-maximum length "$011\ldots110$" or "$100\ldots001$" strings starting at the p-th position of its infinite significand are called *hard-to-round*

[24] We make here an important change in notation. To the point m was used for significands. From now on it will denote hardness-to-round.

cases. Notice that we make here another abuse of language. What is actually hard to round is not floating-point number X in the domain, which is, by definition, "rounded". It is its image by f, $f(X)$ which is a potential troublemaker. This way of expressing oneself strongly reflects the implementation developer concern: she wants to know beforehand which are those values in the domain whose image will give her a hard time to round. A floating-point number of the domain corresponding to the longest weird string in its image (or those that correspond to it, if several numbers share this property) will be called the (or a) *worst case*.

But yet, we have made no progress in the path towards the computation of m.

Given the importance of this question for the implementation of correct rounding, many efforts and expertise have been dedicated to this computation. A recapitulation of these endeavors and their achievements can be found in Chap. 12 of [78].

The point is to find either the exact value of m or a reasonable upper bound to it. In the latter case, we want this bound to be rather tight since, for actual computations, using unnecessarily large values of m impose needlessly high accuracies for approximations that will possibly take an heavy toll on performance: these computations may be very time (and possibly, memory) consuming.

As we have no access to exact values, a first approach is to make an abstract reasoning on the shape of the digits strings that represents them.

Approaching the problem: a probabilistic model

Let us assume that in y, the infinite length binary significand of $f(X)$, the bits after the p-th position form a sequence of independent random digits. Each of them has a $1/2$ probability of being either "0" or "1".

This is a very crude assumption since one can easily find situations where it does not stand. For instance the quotient of two floating-point numbers (once exact floating-point values and exact midpoint values have been discarded) can have an infinite significand but, at some point, the digits form a non-random at all infinite periodic sequence. Another assumption-debunking situation can be found for the $\exp(x)$ function, when x is so small that $\exp(x) = 1 + x$ with an extremely good accuracy. Nevertheless, let us toy a bit with this idea to see where it takes us.

Standing behind our assumption, the probability of having any particular sequence of some k_0 bits starting from position p is 2^{-k_0}. Of course, this is also true for our peculiar "$100\ldots001$" and

"011...110" strings of interest. Hence, if we consider a set of N floating-point elements of $\mathbb{F}$ (or an interval within it) the number of their image by f having any particular sequence of k_0 bits starting at position p is $N \times 2^{-k_0}$. As soon a k_0 grows larger than $\log_2(N)$, this number of occurrences becomes fractional. It means that such a situation should seldom happen. Furthermore, there is a relationship between N and p. For instance the number of elements of a binade in precision p is 2^p. In this context the k_0 tipping point above which the number of occurrences becomes fractional is $k_0 = p$. For $\mathbb{F}$ in full the number of elements $N \approx 2^p \times 2(e_{\max} - e_{\min}) + 2$. So much for k_0.

The interest of theses relations between N, k_0, and the number of occurrences is that it makes it possible to test (not to prove, we lightheartedly concede it) how well this model does in Real World. They are also helpful for picking adequate parameters for our worst cases search algorithms.

Several checks have been done. Some can be found in Chap. 12 of [78] for the $\sin(x)$ and $\exp(x)$ functions for precisions ranging from 8 to 24, in interval $[1/2, 1)$. The value of these checks is that, at such "low" precisions, exhaustive verification of actual values can be done in a reasonable time.

The experiments show that the expected values of the model closely fit those found in reality. Furthermore, the actual verification of the hardness-to-round by other methods (see below) have also comforted the validity of the model for higher precisions (the 53 bits of the IEEE 754 `binary64` format) and for a full-fledged floating-point system (not only a single binade). Actual hardnesses-to-round have been found in the $2p$ order of magnitude.

This model can be used for three important but different purposes:

1. check our hardness-to-round research tools (they should find a number of cases more or less in line with those of the model);
2. give us a hint about the optimality of bounds to the hardness-to-round found with other methods (a bound constantly larger, by a wide margin, than that expected from the model – regardless of the function, the interval, etc. – deserves to be questioned for its optimality);
3. tune elementary functions algorithms that use Ziv's strategy (see Chap. 2, p. 79) when actual hardness-to-round is still unknown.

But we should never forget that improbable events occur (more often than usually expected[30]) and "unusually" high values for hardness-to-round should not be overlooked as impossible.

Armed with our yardstick, we can come back to the efforts to compute actual hardness-to-round bounds and summarize the situation as follows.

Proved operational bounds for hardness-to-round

For a basic operation such as quotient ($f(x,y) \rightarrow x/y$) of two floating-point numbers, it has been proved [58] that the length of strings "0" or "1" after the $p+1$-th digit is bounded by $p-1$ what gives a hardness-to-round of $2p-1$ (whenever the quotient is not an exact floating-point number, for directed rounding modes and noticing that the quotient can not be a midpoint when the radix is a prime number, for rounding-to-nearest mode). The principal qualities of this bound are that it is mathematically proved and yet allows for fast implementations since it is sharp.

Proved but intractable bounds for hardness-to-round

There is a large bunch of results deriving from the 1851 Liouville Theorem [68] about algebraic functions. For instance, Brisebarre and Muller, in [77] give a bound for the $f(x) \rightarrow x^{a/b}$ function. This bound is, except for some special values of a and b, much larger than that one can expect from the probabilistic model. It can be debated for its optimality and cannot be used to implement efficient approximations since it mandates computing with a very high precision.

We already mentioned the 1882 Lindemann Theorem and corollaries. They assert that no image by $f(x)$ will be a midpoint but do not give any clue as to what the minimal distance to a midpoint is.

Eventually, we have the Nesterenko and Waldschmidt Theorem [79] that, when specialized to rational numbers, allows for the computation of the hardness-to-round of several functions. Their starting point is the following:

- suppose we have a nonzero real number $\theta \in \mathbb{R}^*$;
- suppose that α and α' are any two algebraic numbers;

the $|e^\theta - \alpha| + |\theta - \alpha'|$ sum is lower bounded. Notice that in this formula $e = 2.718\ldots$ represents the base of the natural logarithm and not the exponent of a floating-point number[25].

In other words, if θ is "close" to an algebraic number, e^θ is "far" from any of them. And their contribution goes as far as setting an explicit lower bound to this sum. In terms of floating-point numbers – and since they are rational numbers, hence algebraic numbers – for any floating point number θ there obviously exists an algebraic number α' such that $|\theta - \alpha'| = 0$. It suffices to set $\alpha' = \theta$. Then θ's exponential is necessarily at some lower bounded distance from any algebraic number. This is also a lower bound to the distance of this exponential to any floating-point number and to any of their midpoints too (they are rational numbers as well). Why not to use it as a bound to the hardness-to-round of the exponential? To sort out this question we need to go into more

[25] Exponent of floating-point X will be denoted by e_X. rather than e by itself

details.

In the expression of their bound, Nesterenko and Waldschmidt use the concept of *Weil's height* that we will briefly present here before the theorem itself.

Definition 1.8 (Weil's height)**.** Let α be an algebraic number of degree d and $P(x) = \sum_{i=0}^{d} P_i x^i$ be its minimal polynomial. Let $P(x) = P_d \times \prod_{i=0}^{d}(x - \alpha_i)$ be the factorization of P over the complex numbers. The Weil's height of α is

$$H(\alpha) = \left(|P_d| \times \prod_{i=0}^{d} \max(1, |\alpha_i|) \right)^{\frac{1}{d}}.$$

Let us remind that the minimal polynomial of an irreducible rational number a/b over $\mathbb{Q}$ is $P(x) = bx - a$ and that its factorization is $P(x) = b(x - a/b)$.

With rational numbers, the parameters found in the definition of Weil's height become:

- $d = 1$ (the exponent $1/d$ vanishes, the product is reduced to a single factor);
- $|P_d| = |P_1| = |b|$.

The height itself is:

$$H(a/b) = |b| \max(1, |a/b|). \tag{1.5}$$

This can be expressed more clearly as:

$$H(a/b) = \max(|a|, |b|) = \begin{cases} |a|, \text{ if } \left|\frac{a}{b}\right| \geqslant 1 \\ |b|, \text{ if } 0 < \left|\frac{a}{b}\right| < 1 \end{cases}. \tag{1.6}$$

If we come back now to the normalized floating-point numbers, $X \in \mathbb{F} \setminus \{\text{subnormal numbers}\}$ can be written as $M\beta^{e_X - p+1}$ where M is the integral significand of X. This implies that $\beta^{p-1} \leqslant M < \beta^p$.

If we write X as a rational number, we have:

- if $|X| \geqslant 1$, $|X| = \dfrac{M\beta^{e_X}}{\beta^{p-1}}$, with $M\beta^{e_X} < \beta^p\beta^{e_X}$ and e_X as the exponent of X;
- if $0 < |X| < 1$, $|X| = \dfrac{M}{\beta^{-e_X+p-1}}$.

If we now use these values in 1.6, we have :

- if $|X| \geqslant 1$, $H(X) = M\beta^{e_X} < \dfrac{\beta^p M\beta^{e_X}}{\beta^{p-1}} = \beta^p|X|$;

- if $0 < |X| < 1$, $H(X) = \beta^{p-1} < \beta^{-e_X+p-1}$ as one must remember here that $e_X < 0$; $\beta^{-e_X+p-1} < \frac{\beta^p}{M}\beta^{-e_X+p-1} = \beta^p\frac{\beta^{-e_X+p-1}}{M} = \frac{\beta^p}{|X|}$.

We can conclude that, for any nonzero normalized floating-point number X, we have:

$$H(X) \leqslant \beta^p \max(|X|, |1/X|).$$

Let us come back now to Nesterenko and Waldschmidt's Theorem [79] itself, in a version specialized to rational numbers.

Theorem 1.1. *Let α and α' be rational numbers. Let θ be an arbitrary nonzero real number. Let A, A' and E be positive real numbers with*

$$E \geqslant e,\ A \geqslant \max(H(\alpha), e),\ A' \geqslant H(\alpha').$$

Then

$$\begin{aligned}&|e^\theta - \alpha| + |\theta - \alpha'| \geqslant \\ &\exp\Big(- 211 \times \big(\log(A') + \log(\log(A)) + 2\log(E \times \max(1, |\theta|)) + 10\big) \times \\ &\qquad \big(\log(A) + 2E|\theta| + 6\log(E)\big) \times \big(3.7 + \log(E)\big) \times \log(E)^{-2}\Big).\end{aligned}$$

The proof of the theorem can also be found in [79].

Now we are going to try to apply this theorem to the TMD in the case of the function $\exp(x)$.

Suppose now that α is a precision-p floating-point number in interval $[1, \beta)$. The exact value of $\exp(\alpha)$ belongs to interval $[e, e^\beta)$. Let k be a number such that $[e, e^\beta) \subset [1, \beta^k)$. We use now the Theorem 1.1 with $E = e$ and $\theta = \alpha'$, where α is any precision-p floating-point number in $[1, \beta^k)$. We obtain the following:

$$\begin{aligned}&\Big|e^{\alpha'} - \alpha\Big| \geqslant \\ &\exp\Big(- 992 \times \big((3k+p)\log(\beta) + \log((p+1)\log(\beta)) + 12\big) \times \big((p+1)\log(\beta) + 2e\beta^k + 6\big)\Big).\end{aligned}$$

If we specialize further our formula for radix-2 ($\beta = 2$ and $k = \lceil \log_2(e^2) \rceil = 3$), this gives

$$\Big|e^{\alpha'} - \alpha\Big| \geqslant 2^{-688p^2 - 992p\log(p+1) - 67514p - 71824\log(p+1) - 1283614}.$$

This formula shows that the distance of the exponential of a floating-point number in interval

$[1, 2)$ to a midpoint is lower bounded by a value in the order of 2^{-Cp^2}, where C is a rather big constant. To be even more specific, a final specialization performed by setting $p = 53$ (`double-precision` C type or `binary64` format) gives

$$\left| e^{\alpha'} - \alpha \right| \geqslant 2^{-7290678}.$$

The consequence is that, to ensure that an approximation $\widehat{f}$ of function f allows for correct rounding, its accuracy must be $\epsilon \leqslant 2^{-7290678}$ and gives us an upper bound for the hardness-to-round $m = 7290678$. That is a lot, both compared to our probabilistic yard stick and to actual findings (a maximum of $m = 158$ was found for the exponential function with $p = 53$ and in directed rounding mode) by other methods that we will see thereafter.

This value is way too large for any Real Life implementation attempt.

Generating hard-to-round cases This is another way to tackle the problem. If one can generate hard-to-round cases she gets a lower bound of the hardness-to-round. And if she can generate the Holly Grail "worst case", she has solved the dilemma.

A starting point in this direction could be the seminal work of several researchers (Kahan [54], Parks [82] [83], Cornea [20]...) who have followed this route for the square root function. Their target was not actually to solve the dilemma but to make it is easier to check the implementations by stress testing them with hard-to-round cases. As such they could efficiently create "pretty good" hard-to-round cases but did not try to ensure they created "The" worst case.

Exhaustive search: a dead end? To be fair, this approach, computing and checking the image of all the floating-point numbers of the domain, has been realized for "low" precisions (up to the 24-bit precision significands of the IEEE 754 `binary32` format, see, for instance, [78], Chap. 12). To our knowledge, the first who did that were M. Schulte and E. Swartzlander as they explain in [96] p. 139, 141. Beyond that point, it becomes technically intractable for, says, IEEE 754 `binary64` format even if a promising result, based on an implementation using FPGA's, was published in 2011 [25]. Nevertheless, work done with this technique has allowed to gather data and knowledge about how hardness-to-round actually behaves. It has helped to validate (in the weak sense that it did not disprove it) the statistical model presented above and helps to test (over limited intervals) the algorithms sketched below.

Direct hard-to-round case finding: the L-algorithm This algorithm has been developed, implemented and intensively used ever since by Vincent Lefèvre. It is described in his PhD thesis[57].

In a nutshell, it uses linear approximations (degree-1 polynomials) of function f over small domains and checks that the evaluation of these polynomials never "comes too close" to a breakpoint. If it does, a hard-to-round case has possibly been spotted and has to be checked more thoroughly. The power of the method relies on the ability to make very quick computations and reuse the already done ones as much as possible. Domain splitting makes the problem embarrassingly parallel. Hence, spreading processing over a large number of cores/processors mitigates the adverse effect of using small intervals and eventually allows for a much quicker exploration of the full domain of f.

The smart ideas behind the algorithm go much beyond using affine approximations and breaking the domain into small pieces since those alone would not have sufficed to make the algorithm an effective search tool. Over the time, Vincent Lefèvre has improved upon the original algorithm and implementation. Others have also contributed to improvements as, for instance, in [33] to adapt the L-algorithm to GPUs.

The main achievement (and a still ongoing effort) realized through the use of the L-algorithm is the computation and publication of complete lists of hard-to-round cases for some elementary functions in IEEE-754 `binary64`. Those can be found in [78] and [62].

This knowledge makes it possible to implement correctly rounded functions for, say, the `libm` used as the mathematical library for a C compiler. It has actually been used for this purpose by the authors of `CRlibm` [21].

Hard-to-round case finding beyond `binary64`: SLZ-algorithm?

The L-algorithm, while tractable with the IEEE-754 `binary64` format, at the expense of years of consolidated computing time for a single function, has been found impractical for higher precisions and, more specifically, for IEEE-754 `binary128`, the so-called "quadruple precision" with its 113 bits significand. With the L-algorithm, the number of searches for a single binade is roughly proportional to $2^{2p/3}$. This value is bound to the affine approximation that forces to break the approximation interval (say, again, a binade) into that number of very small sub-intervals.

A possible move out of this complexity trap is to try to use higher degree approximation polynomials to achieve a coarser granularity. This is the idea behind the SLZ-algorithm[102] created by D. Stehlé, V. Lefèvre, and P. Zimmermann. For instance, D. Stehlé showed in [99] and [100] that with degree-3 polynomials, one could limit himself to $2^{p/2}$ intervals per binade (still for the IEEE-754 `binary64` format).

But then the ingenious method used in the L-algorithm to detect "near-misses" with breakpoints does not work anymore and has to be replaced with something different. We will not delve into more details here since the implementation of the SLZ-algorithm is a hefty part of the work presented in Chap.3, p. 135. What can be said for the moment is that sophisticated techniques borrowed from cryptography (Coppersmith's algorithm[18]) are used to detect hard-to-round cases. The basic idea is to transform the computation of the distance to midpoints problem into a modular root finding for bivariate polynomials over the integers problem.

1.7 Some basic algorithms

We will not give an extensive presentation of all the possibly interesting and/or useful basic algorithms on floating-point numbers. For an in-depth coverage see [78].

1.7.1 Accurate computation of the sum of two floating-point numbers

The Fast2Sum algorithm computes the sum of two floating point numbers A and B as the unevaluated sum of two other numbers Y_h and Y_ℓ. At first sight no progress has been made but, as we will see, Y_h and Y_ℓ have interesting properties. The algorithm has a long story (see [78], p. 126, for details).
It has four requirements:

- radix $\beta \leqslant 3$; in practice $\beta = 2$, unless one insists in using a Setun radix-3 computer [9];
- subnormal numbers are available;
- rounding-to-nearest mode is used for operations (the tie-breaking rule is indifferent here);
- the exponent of the first input number, A, is larger than or equal to that of the second input number B; that can be reduced to the often easier to check condition[26] $|A| \geqslant |B|$.

And here are the algorithm's two interesting properties:

- $Y_h + Y_\ell = A + B$ *exactly*;
- Y_h is "the"[27] floating-point number that is closest to $A + B$.

1.7.2 Accurate computation of the product of two floating-point numbers

Before we deal with Dekker's multiplication algorithm, we have to introduce a tool it extensively uses: Veltkamp's splitting algorithm.

[26] Access to exponents and their comparison is more tricky than direct comparison of numbers.
[27] Under certain circumstances there can be two of them.

Algorithm 2 Dekker's Fast2Sum algorithm

```
/* Assume a binary floating-point system F_{2,p,emin,emax} with subnormal numbers. */
/* Let A and B be floating-point numbers. */
function FAST2SUM(A, B)
    Require: the exponent of A is larger than or equal to that of B.
    Y_h    ← RN(A + B)
    Y_temp ← RN(Y_h − A)
    Y_ℓ    ← RN(B − Y_temp)
    /*  A + B = Y_h + Y_ℓ and RN(A + B) = Y_h */
    return Y_h, Y_ℓ
end function
```

1.7.2.1 Veltkamp's splitting

The whole idea is to "split" an element A of some $\mathbb{F}_{\beta,p,emin,emax}$ floating-point system into two floating-point numbers of $\mathbb{F}$, A_h and A_ℓ , such that, for a given integer s, $s < p$ (here "s" stands for "shift", nothing to do with the sign bit), we have:

- the significand of A_h fits into $p - s$ digits;
- the significand of A_ℓ fits into s digits (if the radix is 2, the significand of A_ℓ actually fits into $s - 1$ digits);
- $A = \mathrm{RN}(A_h + A_\ell)$

We owe to Veltkamp Algorithm 3.

Algorithm 3 Veltkamps's splitting algorithm

```
/* Assume a binary floating-point system F_{β,p,emin,emax}. */
function SPLIT(A, s)
    CONST ← β^s + 1
    TEMP_1 ← RN(CONST × A)
    TEMP_2 ← RN(A − TEMP_1)
    A_h    ← RN(TEMP_1 + TEMP_2)
    A_ℓ    ← RN(A − A_h)
    return A_h, A_ℓ
end function
```

It must be noted that nothing guarantees that A_h and A_ℓ have the same sign. Veltkamp's algorithm is not equivalent, for A_h, to masking the least significant digits of the significand of A, and, for A_ℓ, to masking the most significant digits of the significand of A. This is why it may seem a bit convoluted. This also explains why, in radix-2, we can have a $(p-s)$-digit A_h and a $(s-1)$-digit A_ℓ. The latter method could hardly be formulated in terms of operations that could work with any floating-point system.

1.7.2.2 Dekker's multiplication algorithm

Discovered by Dekker (see [29]), Algorithm 4, under the conditions presented below, allows for the exact computation of the product of finite[28] floating-point numbers, A and B, to be returned as an unevaluated sum of two numbers Y_h and Y_ℓ (floating-point numbers themselves). Let e_A and e_B respectively be the exponents of A and B.

Let us assume that for our floating-point system $\mathbb{F}_{\beta,p,emin,emax}$ and the values of A and B the four following conditions are met:

- $p \geqslant 3$;
- $\beta^{e_{\min}-p+1} \leqslant 1$;
- $\beta = 2$ or p is even;
- there is no overflow when splitting A and B nor in the computation of $Y_h = \mathrm{RN}(A \times B)$ nor in that of $\mathrm{RN}(A_h \times B_h)$.

The first two conditions hold in any sensible floating-point system. The two others may seem more baroque, see [78] p. 137 for a detailed explanation.

Under these conditions, the numbers Y_h and Y_ℓ computed by Dekker's algorithm have two remarkable properties:

1. if $e_A + e_B \geqslant e_{\min} - p + 1$ then $AB = Y_h + Y_\ell$ exactly;
2. in any case,
$$|AB - (Y_h + Y_\ell)| \leqslant \frac{7}{2}\beta^{e_{\min}-p+1}.$$

Behind (and far from) their superficial simplicity, the workings of these two algorithms make it difficult to establish rigorous proofs. As for Dekker's algorithm, the reiterated efforts of several authors culminated in Sylvie Boldo's transcription to the Coq theorem prover and proof completion as reported in [7].

1.7.2.3 Error Free Transformations

The previous examples are typical of a class of algorithms collectively known as *Error Free Transformations* (EFT). Let us denote as $\bullet$ an operation defined on $\mathbb{F}$. Very often, for $A \in \mathbb{F}$, $B \in \mathbb{F}$, $A \bullet B \notin \mathbb{F}$ even if $\mathrm{RN}(A \bullet B)$ is a numerical value in $\mathbb{F}$. But it is known for basic operations ($\bullet \in \{+, -, \times\}$) that the rounding error itself ($(A \bullet B) - \mathrm{RN}(A \bullet B)$) is an element of $\mathbb{F}$.

For $A \in \mathbb{F}$ and $B \in \mathbb{F}$, $\mathrm{RN}(A \bullet B) = Y_1$, $A \bullet B = Y_1 + Y_2$ with $Y_1 \in \mathbb{F}$ and $Y_2 \in \mathbb{F}$

[28] In the sense of different from the special values $-\infty$ and $+\infty$. No surreptitious smuggle in of a new kind of floating-point numbers on sight!

Algorithm 4 Dekker's multiplication algorithm

```
/* Assume a binary floating-point system F_{β,p,emin,emax} and */
/* that the four conditions in main text are met. */
function DekkerMult(A, B)
    Require: s == ⌈ p/2 ⌉
    /* Split the two arguments so that the "halves" can be exactly "cross multiplied". */
    (A_h, A_ℓ) ← Split(A, s)
    (B_h, B_ℓ) ← Split(B, s)
    /* Compute the main term: a "quick and dirty", possibly inexact, approximation for A × B. */
    Y_h        ← RN(A × B)
    /* From now on, compute the correcting term: subtract from the inexact value Y_h the four */
    /* exact "cross terms" computed as when using the school-grade multiplication algorithm to */
    /* perform A × B as (A_h, A_ℓ) × (B_h, B_ℓ). Intermediate computations values are all exact */
    /* under the condition e_A + e_B ⩾ e_min − p + 1. */
    TMP_1      ← RN(−Y_h + RN(A_h × B_h))
    TMP_2      ← RN(TMP_1 + RN(A_h × B_ℓ))
    TMP_3      ← RN(TMP_2 + RN(A_ℓ × B_h))
    Y_ℓ        ← RN(TMP_3 + RN(A_ℓ × B_ℓ))
    /* Eventually, Y_ℓ is the exact difference between A × B and Y_h. */
    return Y_h, Y_ℓ
end function
```

There are generally some extra conditions (e.g. no underflow is assumed for multiplication). But the interesting point is that the values of Y_1 and Y_2 can be computed exactly within floating-point arithmetic itself. In general, the first value, Y_1, is straightforwardly computed, with a single floating-point instruction. The computation of Y_2 is sometimes more complicated, as can be seen from our previous examples.

Many authors [40][39][66][81][84][91][90] other than those already cited have developed such error free algorithms. The introduction of the FMA instruction has strongly stimulated this kind of research.

1.8 Function approximation

In this section we present the principal steps taken to perform a function approximation and how this thesis work fits into this framework. We must first reassert that the driving spirit of our (as a research team) endeavor in the realm of function approximation is combining numerical quality (correct rounding, method error bounding, certified evaluations) with computation efficiency (speed, memory footprint). This is necessary to explain the emphasis we put on some not-so-obvious steps for unworldly approaches. For each of them we will give a reference to a PhD thesis work realized in the team. It has nothing to do with an AriC-centric bias, neither with a PhD-thesis-centric one.

Our only point here is to demonstrate the constancy of this team's dedication in this field and how the present work fits in a long term collective endeavor.

1.8.1 The steps

We can single out six main steps in the approximation of some function f in some floating-point format $\mathbb{F}$:

0. compute hard-to-round cases for function f for $\mathbb{F}$;
1. design argument(s) range reduction of f;
2. devise a "machine efficient" polynomial approximation p_{opt};
3. set up an evaluation scheme;
4. compute a rigorous approximation error $||f - p_{opt}||$;
5. compute a certified evaluation error for p_{opt}.

1.8.2 Compute the hard-to-round cases

We give this step a special attention by numbering it as "0" since it is not classically identified, *per se*, as a part of function approximation. The reason is that not everyone is interested in correct rounding. If, for instance, faithful rounding (see Remark 1, p. 48) is all that is wanted, this step can be avoided.

Hard-to-round cases, as already stated in §1.6.3, p. 55, are closely connected to the TMD. One must notice that the list of the hard-to-round cases is a mathematical property of both the function f and the target floating-point format, independently of our intent to approximate it. But it is also important to understand that, for a given function f, there is no such thing as a "generic" list of hard-to-round cases. For that matter, we must also consider what the floating-point domain and image formats are in.

When analyzed for hard-to-round cases, each triplet (f, $\mathbb{F}_d$, $\mathbb{F}_i$) (where f is a function, $\mathbb{F}_d$ the floating-point format domain of f and $\mathbb{F}_i$ the set where the images of the elements of the domain must live in) will yield its own list of cases.

As stated above (§1.6.3.3, p. 67), two principal approaches have been designed for a systematical research of hard-to-round cases. The first one is the so-called "L-algorithm" due to V. Lefèvre and is described at length in his PhD thesis [63]. The second one, known as the "SLZ-algorithm", was designed by Damien Stehlé, Vincent Lefèvre and Paul Zimmermann and exposed in [102] and [103].

A part of this PhD thesis (Chap. 3, p. 135) will be devoted the work done for the implementation and the exploration of the properties of this algorithm.

1.8.3 Argument/range reduction

The usual techniques for function approximation rely on polynomials. The reasons for picking this option are technical and mathematical as well. From the technical point of view, the main appeal of polynomials is that – when the only operations at hand are addition, subtraction, multiplication, and comparison operators – piecewise polynomials are the only functions in one variable that can be computed. This may still be the case in simple microcontrollers. Yet, with more sophisticated processors, these operations can be computed at blazing speed which sustains the appeal of polynomials, even on these advanced architectures.

To be more specific here, for those written in the so called "standard monomial basis", classical evaluation schemes as Horner's algorithm (see [45]) are very often used since they only rely on addition and multiplication (and for some time now, FMA) that are highly optimized (small cycle number, pipelining) in nowadays hardware. Other efficient schemes are also available for other bases.

Mathematically, polynomial approximation has a long and rich history. Important mathematicians such as Weierstrass [109], Chebyshev [10], La Vallée Poussin [27], Remez [86] [88], to name only a few, have not only leveled the theoretical ground and but have also created tools that allow for the design of robust polynomial approximation. Unfortunately these techniques are plagued by the narrowness of the intervals over which they are accurate for a reasonable [29] degree.

When it comes to the approximation of some non-trivial function f defined over an interval as wide as (approximately) $[-2^{1024}, +2^{1024}]$ (the span of the `binary64` format) there are very small chances that a polynomial with a degree, say, smaller than 100 will provide an accurate enough approximation. It could be necessary to resort to a set of polynomials, each defined over a somehow limited interval. But then the cardinal of the set could be prohibitively large.

But even a cursory observation of, say, periodical functions, can lead one to think that it is not necessary to compute an approximation polynomial that works over the whole domain of f. If we can reduce the computation of many elements image of the domain to those of a much more limited interval, a single or an itself limited number of polynomials can be sufficient. This is *argument reduction*.

[29] In the sense that the polynomial can be stored in a reasonable amount of memory space and evaluated in a reasonable time span.

When designing the argument reduction scheme one can first set apart from the initial domain some values or even subsets for which elements have an obvious image that can be returned without further evaluation. For instance, the image by the $\exp(x)$ function of large positive argument values will, above some easy to compute once-for-all threshold, be assigned to the positive infinity in the target floating-point format.

For what is left, the solution is to switch, through smart mathematical transformations, from the initial function f, defined on domain $\mathbb{D}_f$, to another (or limited number of) function(s) g defined over a much narrower domain $\mathbb{D}_g$. The function g is somehow equivalent (in a very broad sense, since it can actually be very different) to f, but can be approximated, over its own domain, by a single (or limited number of) polynomial(s).

The result of the evaluation of g will possibly go through another series of transformations to take into account the effects of range reduction before being returned as the result of the evaluation of f. The (somehow intricate) process is sketched in Figure 1-9. It has not been the main topic of any of the thesis defended at LIP but cursory remarks can be found in the work of David Defour [28], who wrote one of the first thesis that ever covered the correctly-rounded function approximation development process from start to end. A few years later Christoph Lauter developed a more systematic approach to the question with an opening to its automatic management in his own work [61]. The design of range reduction is mathematically challenging as soon as we leave the relatively well-marked path of *additive range reduction*[30]. It must be done with great care to keep errors under control (see [78] Chap. 11). As such, its automation is a really difficult challenge.

1.8.4 Computing polynomial p_{opt}

Before we compute an optimal approximation polynomial, we must clarify how we gauge accuracy.

1.8.4.1 Norms and accuracy

We have defined above (def. 1.4 and def. 1.4) what absolute and relative error are for a single value.

When is comes to the accuracy of an approximation we need a more general definition based on norms on the real-valued bounded functions. In this work we will consider the *supremum norm*

[30] Let us assume X is the initial floating-point argument and c the range reduction constant. In additive reduction, we are looking for an integer k and a real number $y \in [0, c]$ or $y \in [-c/2, c/2]$ such that

$$y = X - kc.$$

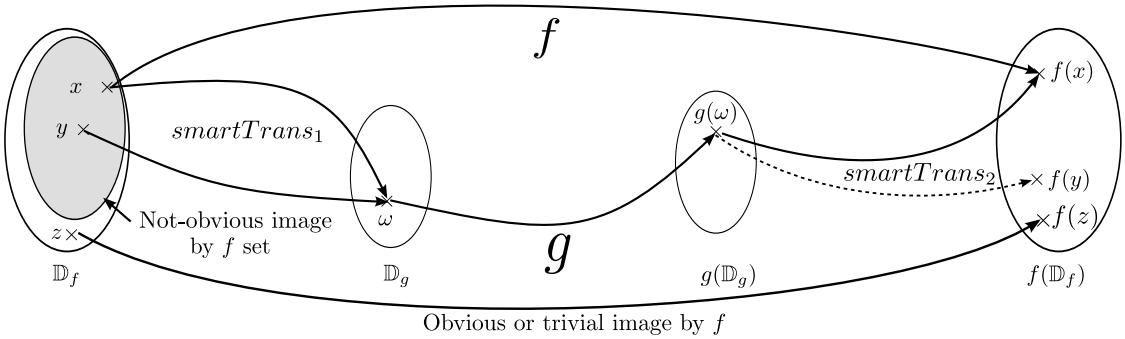

Figure 1-9: f is the function to approximate. Some part of its domain $\mathbb{D}_f$ (gray area) can be mapped, through a "smart transformation" $smartTrans_1$, into some narrow interval (for instance, x and y will be mapped to the same value ω) denoted by $\mathbb{D}_g$. This is the domain of a function g (we only consider here one g function, for clarity), "equivalent" to f in the sense that, at a final (optional) "smart transformation" $smartTrans_2$ short, $f(x) = g(\omega)$. Hopefully function g can be approximated with a single (or limited number of) polynomial(s). Value y can possibly have a different image (dashed line) than x whenever the final transformation $smartTrans_2$ takes into account the initial values (x or y). The representation of two arguments case for $smartTrans_2$ is omitted for clarity. Ultimately, there is a subset of $\mathbb{D}_f$ whose elements image by f can be trivially computed (bottom arrow).

(also called uniform norm, infinity norm or maximum norm[31]);

Definition 1.9. The supremum norm of the function $f(x)$ over the interval $[a, b] \subset \mathbb{R}$ is defined as:

$$\|f\|_{\infty}^{[a,b]} \;=\; \sup\{|f(x)|,\; x \in [a,b]\}.$$

We will often denote this norm simply by $\|f\|$ when the interval is obvious.

When we say that the function $p(x)$ (possibly a polynomial) is an approximation of the function $f(x)$ with an absolute error accuracy ϵ_{abs} over the interval $[a, b]$ we mean that:

$$\|f - p\|_{\infty}^{[a,b]} \;=\; \sup\{|f(x) - p(x)|,\; x \in [a,b]\} \;<\; \epsilon_{abs}.$$

Almost identically, when it comes the relative error accuracy ϵ_{rel}. what we have in mind is:

$$\left\|1 - \frac{p}{f}\right\|_{\infty}^{[a,b]} \;=\; \sup\left\{\left|1 - \frac{p(x)}{f(x)}\right|,\; x \in [a,b]\right\} \;<\; \epsilon_{rel}.$$

[31] When the function is continuous over a closed interval.

1.8.4.2 Approximation polynomial properties

At this point an approximation polynomial p_{opt} matching all the user desired properties will be crafted. These can be many and sometimes conflicting:

- accuracy of the result ("small" absolute or relative error);
- numerical properties (e.g. correct rounding);
- polynomial structural properties (e.g. precision and/or specific values for the coefficients);
- execution time/memory footprint optimization;
- etc.

Many approaches are possible to realize this step. The already mentioned Christoph Lauter's thesis pays a sustained attention to this question [61]. In his own text [12], Sylvain Chevillard's develops an in-depth analysis with a special emphasis on certification. Both are the main authors of the Sollya [13] software.

1.8.5 Setting up an evaluation scheme

As already mentioned in our plea for polynomials, those expressed in the "standard basis" can be efficiently evaluated using Horner's algorithm. Other schemes exist for these, as Estrin's [76], which allows for more parallelism. For Chebyshev polynomials one can efficiently use Clenshaw's algorithm [15]. De Casteljau's algorithm [26][31] can be applied to Bernstein's form polynomials.

While the major incentive for using evaluation schemes is that they are much faster (and/or more memory-saving) than naive evaluation of polynomials, performance is not the only criterion. Numerical properties of the scheme matter too. Mathematically equivalent methods on real numbers may, now to no surprise, exhibit diverging accuracies when applied to floating-point ones. Insights on the application of different schemes can be found in Guillaume Révy's work [89].

1.8.6 Certified approximation error

Tools used at the previous steps often deliver quite satisfactory results. Some properties of the approximants (size/structure) are easy to check. Nevertheless approximation error checking is more complicated and can not be left in some "almost-OK" nebulous state, specially when critical applications are envisioned. Techniques and tools that allow for a certified error have to be called in. This is not a matter of hair-splitting perfectionism. First of all, as already noted, absolute certitude is mandatory for critical systems. But, even in less stringent applications, an approximant is often one single piece of the possibly many that make up some artifact. Making any relevant reasoning about

its properties as a whole requires that each element is rigorously qualified (certified). Mioara Joldeş's thesis manuscript [51] is largely devoted to that question. She is also an important contributor to the Sollya software.

1.8.7 Computing a certified evaluation error

The previous step gives certified bounds on the committed error when replacing function f by its approximation polynomial p_{opt}. But there are more errors to come. The evaluation of polynomial p_{opt} will be implemented under the form of some machine computable program. Its execution will, in turn, generate its own string of numerical errors. These have to be rigorously characterized too. Investigating the numerical properties of computer codes is an arduous problem. Computations and proof about them are particularly tedious and error-prone. That is why the capability of automatic extraction of these properties from programs code and their handling with formal tools as proof assistants strongly fortifies the trust one can have in the approximants and their implementations.

In our team the GAPPA tool [71] [73], was developed by Guillaume Melquiond during his thesis [72] work, and the Coq [19] theorem prover, whose application to proofs on floating-points theorems is a large body of Sylvie Boldo's thesis [6], are routinely used for that certification task.

This final step bridges back between high performance computations and mathematics.

After all these steps are completed, the resulting computing modules do not only have an intrinsic high quality to be directly called from user code (for some scientific application) but, as already stated, can also be confidently used as building blocks to assemble more complex tools.

It is time now for us to put to use all this introductory material and, with the next chapter, to take on the core of the presented works.

Chapter 2

A better understanding and control of Ziv's rounding test

2.1 Motivation

Developing algorithms and software for floating-point function approximation is a many-facet activity. The complexity and the diversity of the issues can spark many different kinds of preoccupations.

Sometimes it is a matter of creating new concepts or importing methods developed in other fields into the realm of function approximation. This is for instance the case when Coppersmith's algorithm is applied to the Table Maker's Dilemma.

Other times it can be the creation of new devices in order to supplement the "function approximator's toolbox" with extra and more efficient machinery. Combining linear-programming, Remez algorithm and LLL-reduction as to optimize the polynomial approximation of a function can be viewed from this perspective.

But there is another aspect, possibly less glamorous, that deserves a close attention. Interest in floating-point computation is not a new whim: it has a long and tumultuous history. Our predecessors have accumulated a large and valuable "bag of tricks". Some of them have, from the start, been established on firm grounds or, later on, thoroughly checked under rigorous mathematical scrutiny. Others have not and have nevertheless made their way into the informal floating-point trade Credo.

We feel it a necessary work to revisit this heritage and to get pending clarifications done. An important step in this direction, among other things, has been taken with the publication of the

Handbook of Floating-Point Arithmetic [78]. In a sense it has opened a new research program, possibly very mundane at first glance, but that has clearly important implications. In fact, it does not only involve the verification of admitted rules but also the investigation of their possible extensions (radix-dependency is a big issue in this field, as is the possible availability of not-so-new FMA processor instruction whose implication are yet overlooked). Studying their limitations is also instrumental for their safe use. Rather than the useless rumination of old chewed bones, this exercise can be a way to deepen our understanding of the behavior of floating-point computations and extend our ability to perform them safely.

This chapter is an inroad in this direction, in the realm of techniques used in function approximation computation. The main results present here were already published in a more terse version in [24].

2.2 The "main + correcting term" technique

Let us assume that we want to compute or approximate the exact value, noted y, of, say, an arithmetic operation, a function evaluation at some point or any other accurate numerical computation.

There are many instances where this y, if it can actually be computed or, if not, its approximation, is returned as the unevaluated sum of two floating-point numbers noted y_1 and y_2. Another quite similar situation is when a single floating-point is eventually returned but is computed as the sum of two numbers at a very last stage of the algorithm.

Let us take a first example of a result under the form of such an unevaluated sum, that of Dekker's Fast2Sum algorithm, whose detailed description can be found in §1.7.1, p. 69.

We quickly remind the algorithm here.

Algorithm 5 Dekker's Fast2Sum algorithm

```
/* Assume a binary floating-point system F_{2,p,emin,emax}. */
/* Let a and b be floating-point numbers of F . */
function FAST2SUM(a, b)
    Require: The exponent of a is larger than or equal to that of b.
    y1    ← RN(a + b)
    ytmp  ← RN(y1 − a)
    y2    ← RN(b − ytmp)
    /*  a + b = y1 + y2 and RN(a + b) = y1 */
    return y1, y2
end function
```

Notice that no further operation is requested here: $y_1 = \mathrm{RN}(a + b)$. Should one compute $y_1 + y_2$, no rounding test would be needed since one of the nice properties of Dekker's Fast2Sum is

that $y_1 = \mathrm{RN}(y_1 + y_2)$.

When the order of magnitude of the elements in the pair differ largely, as in our first example (we have here $y_2 < \mathrm{ulp}(y_1)$), this result is generally more valuable (i.e. more accurate) than the single value $\mathrm{RN}(y_1 + y_2)$.

In this case, the largest of the two in magnitude noted y_1, will be called the "main term" (providing, roughly speaking, for the most significant bits of the value) and the other one noted y_2, will be called the "correcting term" (providing, when added to the main term, for the least significant bits, and possibly affecting the most significant ones due to carries propagation), that "adjusts" the main term value closer to that of the exact value of y.

In our next example, the algorithm returns a single value, as in function approximation. But what is interesting here is that at the very last stage the algorithm performs the sum of a "main + correcting term" pair. This second example is taken from [40]. That paper deals with accurate polynomials evaluation using a so called "Horner's Compensated Algorithm". In §3, it is described as follows in Algorithm 6. Error Free Transformations are introduced in §1.7.2.3, p. 71.

Algorithm 6 Compensated Horner's algorithm

```
/* p is a floating-point coefficient polynomial evaluated for floating-point argument x. */
function CompHorner(p, x)
    /* EFTHorner is a custom version of Horner's algorithm that uses error-free transformations */
    /* and returns a value r̂ (the main term) and two new polynomials, p_π and p_σ. */
    /* These polynomials convey the information needed to compute a correcting term. */

    [r̂, p_π, p_σ]   ← EFTHorner(p, x)

    /* ĉ (the correcting term) is computed by applying the classic Horner's algorithm on */
    /* polynomial RN(p_π + p_σ) and x. */

    ĉ   ← Horner(RN(p_π + p_σ), x)

    /* Eventually, the result is computed as the rounded sum of r̂ and ĉ. */

    r̄   ← RN(r̂ + ĉ)

    return r̄
end function
```

Here, the point of the authors of [40] is to guarantee that the computed result, $\bar{r}$, is approximately as good as if Horner's algorithm had been performed with twice the precision of the available floating-point arithmetic system and eventually rounded to this working precision. This is achieved, by the end of the algorithm, by adding to the main term of the result $\hat{r}$ the correcting term $\hat{c}$.

But still, correct rounding is not the final target of the authors since Horner's algorithm is sensitive to the condition number of the problem and it can, in very adverse situations, output very poor results, wrong up to the order of magnitude of the correct value!

2.3 When correct rounding matters: the "main + correcting term" technique in function approximation

Suppose now one wants to compute the correctly rounded value of some function f in some floating-point system $\mathbb{F}_{\beta,p,e_{\min},e_{\max}}$[1]. Here, we will limit ourselves to the "round to nearest, ties-to-even" rounding mode which is the most common.

As stated in §1.8, p. 72, we will approximate function f by some many-steps "composite function" $\mathcal{G}$ such that $\forall\ X\ \in\ \mathbb{F}$, $\mathcal{G}(X)\ =\ \mathrm{RN}(f(X))$, if $f(X)$ is mathematically defined, or $\mathcal{G}(X)\ =\ \mathrm{NaN}$ otherwise.

Sections within the computer program implementing $\mathcal{G}$ will filter out the input elements whose image is not defined or can be trivially determined or read from a table, those whose image is a midpoint (when possible [2]), some hard-to-round cases, in short, all the special cases. Another section will take care of argument reduction, if any. Eventually we will end up with a residual domain $\mathbb{D}_r\ \subset\ \mathbb{F}$ whose element must have their image computed by some function g, more or less deeply buried inside $\mathcal{G}$.

Function g may possibly be very different from f (due, for instance, to argument reduction). This function g is the one we have to approximate over $\mathbb{D}_r$ since (generally) we are not able to compute it exactly for each value of its domain. For that purpose we will devise an approximating function h_ϵ (or a family of these, if $\mathbb{D}_r$ is still too large and must be broken into smaller subdomains) such that[3]:

$$\epsilon\ >\ \max_{X\ \in\ \mathbb{D}_r}\left|\frac{h_\epsilon(X)\ -\ g(X)}{g(X)}\right|.$$

In practical terms, h_ϵ will approximate g over a tight real number interval $[a,b]$ including $\mathbb{D}_r$[4] such that

[1] We assume here, for the sake of simplicity, that both the domain and the image are subsets of the same floating-point system $\mathbb{F}$.

[2] see [49] for a discussion when midpoints are subnormal numbers.

[3] We assume here that $\forall\, X\ \in\ \mathbb{D}_r$, $g\,(X)\ \neq\ 0$. If not so, further refinement and/or splitting must be done on $\mathbb{D}_r$.

[4] where $\mathbb{D}_r\ =\ [a,b]\ \bigcap\ \mathbb{F}$.

$$\epsilon \; > \; \left|\left| \frac{h_\epsilon(x) \; - \; g(x)}{g(x)} \right|\right|_\infty, \text{ for } x \in [a,b] \subset \mathbb{R},$$

on the grounds that what stands for real numbers also stands for floating-point ones. Ideally, for a given ϵ, the actual values of the maximum or of the infinity norm above should not be much smaller than ϵ. Otherwise it would mean that h_ϵ approximates g with an excessively high accuracy, possibly at the expense of a long evaluation time.

Notice that the values returned by h_ϵ are not only theoretical real numbers but are computed floating-point ones.

What should be the value of ϵ? In very broad terms, it should be small enough to allow for the resolution of the Table Maker's Dilemma for all the elements in $\mathbb{D}_r$ and their image in the target floating-point system.

But this is where grim reality catches up. A sufficiently small value of ϵ to go past the TDM in all cases will generally be too expensive in terms of execution time. And very often this time will be wasted because for most of the cases a milder value of ϵ would have been enough.

This opens the door to different strategies for real-world approximation. One, for instance, is to perform a very quick approximation, with a relatively large value for ϵ but nevertheless small enough for most of the elements of the domain. Then make a check to assess whether the accuracy was sufficient (more on this to come very soon). If the check fails, performs another, possibly slower, approximation with a value of ϵ known to be sufficient to avoid the TMD. The idea is to have the second round performed so seldom that the average time of execution of the approximation is not much larger than that of the quick method. This strategy is currently implemented in `CRlibm` [21].

Another strategy somehow generalizes the previous one. Why limit ourselves to two stages? Let us start with some h_ϵ approximation function that has the same properties as in in our preceding example. Perform a check. If it fails, make another try with, for instance, a $h_{\frac{\epsilon}{2^k}}$, with $k \in \mathbb{N}$, function [5]. Check again. If it still fails give $h_{\frac{\epsilon}{2^{2k}}}$ a try, and so on until the check succeeds. The idea is the same though we do not need to know what the ultimate value of ϵ is. Notice that elements whose image is a midpoint must have been dealt with beforehand to avoid an endless loop (or to resort to some bound on ϵ to force exit from the loop but then take the chance to return an incorrect value). This is, in a nutshell, the "onion peel" strategy developed by Abraham Ziv in [111].

[5] with $k > 1$ to allow for a fast enough decrease of the error to make the new attempt worth it.

What can we say about the precision of the values returned by the approximation functions? It will have indeed to be larger than that of the target floating-point system, if we want any actual rounding to be performed at all. Even if g is easy to round (i.e. $g(X)$ is far from a midpoint), the accuracy ϵ will be such that $|h_\epsilon(X) - g(X)|$, upper-bounded by $|g(X)| \times \epsilon$, will be smaller than $\mathrm{ulp}(\mathrm{RN}(g(X)))$. Therefore it will take more than p bits (say, p') to express the floating-point number $h_\epsilon(X)$ if it is to "come close enough" to $g(X)$.

How could this precision-p' number (with $p' \geqslant p$) be expressed?

Among the many possibilities, one is to use a pair of floating-point numbers of $\mathbb{F}$. Indeed, if the numbers do not "overlap" and are not too much "spread apart" (that is their exponents exactly differ by $p+1$) one can view this pair as a precision-$(2p+1)$ number represented by the unevaluated sum of its members. Possible ways to view a pair of floating-point numbers as higher precision number as represented in Figure 2-1.

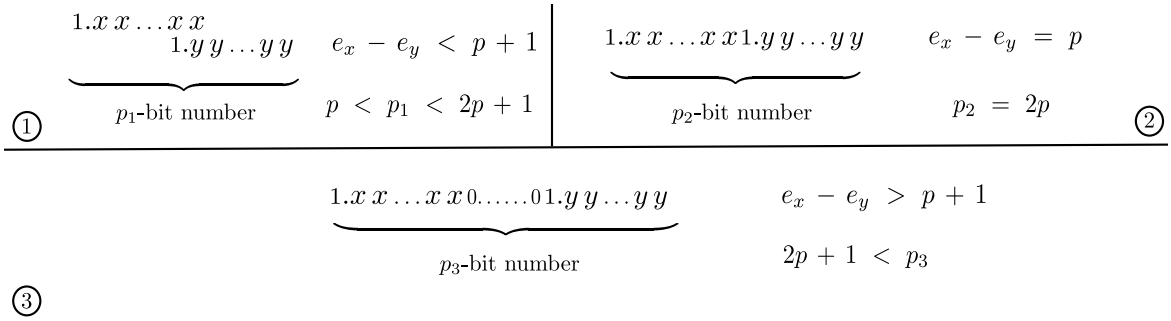

Figure 2-1: A floating-point numbers of precision p pair can be viewed as a precision-p_i number, with $p_i \geqslant p$, if considered as an unevaluated sum of the pair members. Two normal numbers (notice the fractional point) are represented here "aligned" with respect to their relative exponents. In situation 1, the two numbers "overlap", the precision p_1 of the pair is larger than that, p, of the single member but smaller or equal than $2p$. In the extreme situation 2, the two numbers are "exactly aligned". We can represent all the possible values over $2p$ bits. In situation 3, a much higher precision-p_3 number can be represented but not all p_3-possible numbers can: the "central part" of the significand must be all 0s. Subnormal numbers could also be used as the second element of the pair.

Returning such pair is obviously an instance of "main + correcting term" technique. If we call the elements y_h and y_ℓ they are such that:

$$y_h + y_\ell \;=\; h_\epsilon(X) \;=\; g(X) \times (1+\alpha), \text{ with } |\alpha| \;<\; \epsilon. \tag{2.1}$$

Notice that α is unknown. All we know about it is a bound on its absolute value.

Suppose, without any loss of generality[6], that

$$\mathrm{RN}(y_h + y_\ell) = y_h. \tag{2.2}$$

This is a shorthand for the following more lengthy and complex set of conditions:

- if y_h is not a power of 2 and its integral significand (see §1.3.2, p. 33) is odd then $|y_\ell| < \frac{1}{2}\mathrm{ulp}(y_h)$;
- if y_h is not a power of 2 and its integral significand is even then $|y_\ell| \leqslant \frac{1}{2}\mathrm{ulp}(y_h)$;
- if y_h is a power of two (its significand is, almost per definition, a power of two) and $y_h > 0$ then $-\frac{1}{4}\mathrm{ulp}(y_h) \leqslant y_\ell \leqslant \frac{1}{2}\mathrm{ulp}(y_h)$.
- if y_h is a power of two (again, its significand is a power of 2) and $y_h < 0$ then $-\frac{1}{2}\mathrm{ulp}(y_h) \leqslant y_\ell \leqslant \frac{1}{4}\mathrm{ulp}(y_h)$.

The length and complexity of the set of conditions pertain to both the behavior of the "round to nearest, ties-to-even" rounding mode and the peculiarities of the definition we use for the ulp() function (see §1.4.1, p. 39).

Under these assumptions, (2.1) and (2.2), we would like to know when we can safely state that $y_h = \mathrm{RN}(y)$.

To get acquainted with the problem, assume first that y_h and y_ℓ are both positive.
We may find ourselves in one of the two situations depicted by Figures 2-2 and 2-3.

If in the first situation (Figure 2-2), we can safely state that $y_h = \mathrm{RN}(y)$.

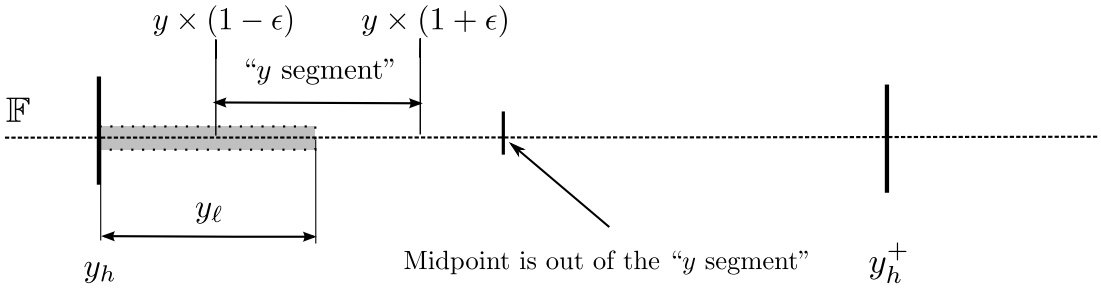

Figure 2-2: y can't be a midpoint nor beyond it: correct rounding is possible.

If in the second one (Figure 2-3), this is not the case and $g(x)$ should be approximated by

[6]It suffices to consider instead y'_h and y'_ℓ such that $(y'_h, y'_\ell) = \mathrm{Fast2Sum}(y_h, y_\ell)$.

another, more accurate, function [7], $h_{\epsilon'}(x)$, such that $\epsilon^{'} < \epsilon$. Should we fall in the second situation again, another attempt would be made with an even better accuracy $\epsilon^{''}$, such that $\epsilon^{''} < \epsilon^{'}$, and so forth, until a final $(y_{h_{final}}, y_{\ell_{final}})$ pair is computed from which we can safely conclude[8] that $y_{h_{final}} = \mathrm{RN}(y)$.

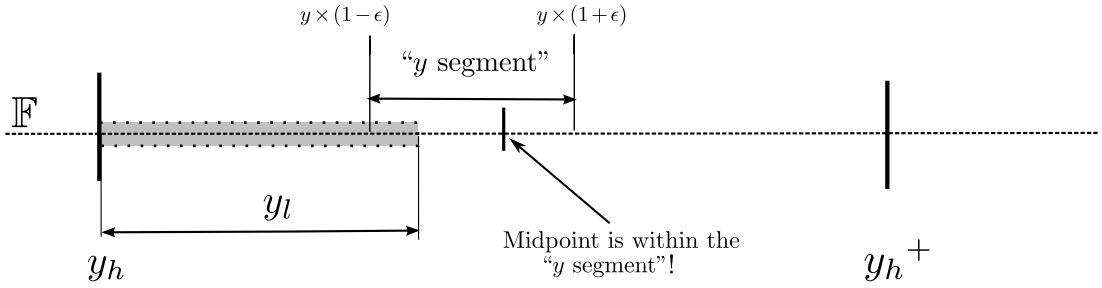

Figure 2-3: y can be a midpoint or beyond it: correct rounding is impossible.

Equivalent figures could be drawn should the signs of y_h and y_ℓ be both negative.

The problem can become quite intricate if, more realistically, we allow for different signs for y_h and y_ℓ, and even worse, if y_h is a power of 2, as in Figure 2-4. Here Condition 2.2 can't anymore be restated as $|y_\ell| < \frac{1}{2}\mathrm{ulp}(y_h)$. It takes at least $|y_\ell| \leqslant \frac{1}{4}\mathrm{ulp}(y_h)$ to make sure that $y_h = \mathrm{RN}(y_h + y_\ell)$. Again, and despite the smaller length of the "y segment", another attempt at higher precision should be made in the situation depicted in Figure 2-4.

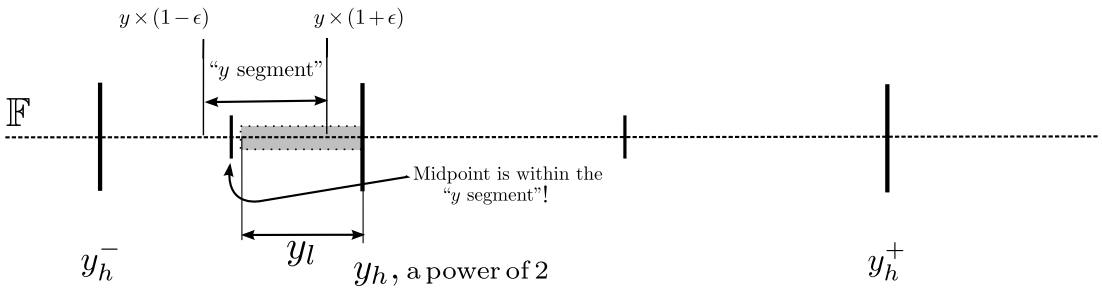

Figure 2-4: Numbers y_h and y_ℓ can have different signs and, to make it worse, y_h can be a power of 2. Here while the length of the "y segment" is smaller than in the previous cases, y can be a midpoint or beyond it, in the direction of y_h^-.

How could one possibly make the check?

[7] It can use the same algorithm but with a better accuracy.

[8] We optimistically assume here that the cases where $f(X)$ is exactly a midpoint of $\mathbb{F}$ have been dealt with beforehand.

From the figures, we see that one could add[9] the length of the "y segment" to y_ℓ and run Algorithm 7.

Algorithm 7 Correct rounding test 1

```
/* y is the exact value of f(X), X ∈ D_r. */
/* (y_h, y_ℓ) is a pair of numbers computed to approximate y. */
/* ε is the relative error of the approximation. */
Require: y × (1 − ε) < y_h + y_ℓ < y × (1 + ε)
/* (by definition of the relative error) */
Require: y_h = RN(y_h + yl)
/* ysl is the "y segment length". */
ysl ← 2ε × y
if y_h == RN(y_h + RN(y_ℓ + ysl)) then
    True
else
    False
end if
```

But this algorithm is doomed to fail: the computation of variable ysl (the "y segment length") is impossible since we do not know y. On the other hand the relative error bound ϵ is known (it should be for without this knowledge any attempt to round is inutile!).

Nevertheless it points us toward a possible solution: add "*something*" to y_ℓ and check whether

$$y_h \;=\; \mathrm{RN}(y_h \,+\, \mathrm{RN}(y_\ell \,+\, \mathit{something})).$$

Intuitively, we see that "*something*" depends on the absolute value of y or, at least, on its order of magnitude. This forces us to compute a different "*something*" for each order of magnitude for all possible y which is an annoyance. Indeed we would like the test to perform as quick as possible since it can possibly be taken many times in a computation. We have also seen that, depending on the respective signs of y_h and y_ℓ we may have to subtract "*something*" instead of adding it.

A different approach could be to multiply y_ℓ by some constant $e > 1$. Ideally e would only depend on the parameters of $\mathbb{F}$ and on the value of ϵ. It could then be computed once for all, for a given algorithm.

The test would then conform to Algorithm 8.

With the very loose requirements put on e, Algorithm 8 is not guaranteed to give correct results. These can fall in four categories:

- *positives*: $y_h \;=\; \mathrm{RN}(y_h + \;\mathrm{RN}(y_\ell \times e))$ and $y_h \;=\; \mathrm{RN}(y)$;
- *negatives*: $y_h \;\neq\; \mathrm{RN}(y_h + \;\mathrm{RN}(y_\ell \times e))$ and $y_h \;\neq\; \mathrm{RN}(y)$;

[9] Well, or could subtract, depending on the sign of y_ℓ !

Algorithm 8 Correct rounding test 2

```
/* y is the exact value of f(X), x ∈ D_r. */
/* (y_h, y_ℓ) is a pair of numbers computed to approximate y. */
/* ε is the relative approximation error. */
Require: y × (1 − ε) < y_h + y_ℓ < y × (1 + ε)
/* (by definition of the relative error) */
Require: y_h = RN(y_h + y_ℓ)
Require: e = some_function(F, ε) and e > 1
if y_h == RN(y_h + RN(y_ℓ × e)) then
    True
else
    False
end if
```

- *false positives*: $y_h = \mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e))$ but $y_h \neq \mathrm{RN}(y)$;
- *false negatives*: $y_h \neq \mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e))$ but $y_h = \mathrm{RN}(y)$.

Of course, desirable properties of the test (and of the value of e) are:

- make no false positives;
- yield the least possible number of false negatives.

The first property is self-evident. We want the test to be absolutely safe: we should never return a wrong value for $\mathrm{RN}(y)$.

The second has to do with performance issues. Test failure mandates *i)* another computation of an approximation, *ii)* with a better accuracy (that will unavoidably take longer, otherwise why not use it from the very start?). The false negative cases must then be kept to a minimum for this (quick) test to be acceptable. If false negatives were too frequent, other, possibly more time consuming but more accurate, options would become relevant.

Figures 2-2, 2-3 and 2-4 are in fact a bit misleading since the length of the represented "y segment" is small, smaller indeed than $1/2 \times \mathrm{ulp}(y_h)$. This is not necessarily true and this length depends, for a given y, on the value of ϵ. As depicted in Figure 2-5, when the "y segment" is too wide, any attempt to round $y_h + y_\ell$ is futile.

The very notion of the correct rounding of the value of a function only makes sense if the accuracy of the approximation is sufficient.

If the relative error of the approximation is larger than $\frac{1}{2}2^{-p+1}$ (or $\frac{1}{2}\ \mathrm{ulp}(y)) = 2^{-p}$ (as in Figure 2-5) it makes it impossible to correctly round. This gives us an absolute lower bound on ϵ.

In reality, we will impose an even more stringent condition on ϵ for reasons that will become clear in the sequel:

$$\epsilon \ < \ 2^{-p-1}. \tag{2.3}$$

This is not a particular problem since, in all practical situations, ϵ will be smaller than that[10] .

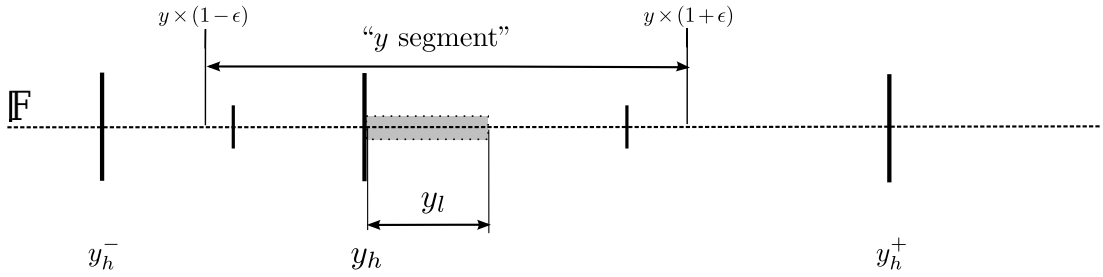

Figure 2-5: Striving for correct rounding is futile under these conditions

We are ready now to state what we could call the "main + correcting term" problem in more formal terms.

Problem 2.1. *Let a real number $y \in \mathbb{R}$, be approximated by two binary floating-point numbers $y_h, y_\ell \in \mathbb{F}_{2,p,emin,emax}$ such that:*

$$|(y_h + y_\ell) - y| \ < \ \epsilon \times |y| \tag{2.4}$$

and

$$\epsilon \ < \ 2^{-p-1},$$

$$y_h = \ RN(y_h + y_\ell). \tag{2.5}$$

Can we find a quick test that allows us to determine if $y_h = RN(y)$?

Notice that (2.4) implies that both y and y_h have the same sign. In the sequel, without loss of generality, we will assume that they are both positive.

Up to now we have essentially been toying with the idea of a rounding test. We will see in the following how it has actually been used (and still is) in the context of the correct rounding of function approximation.

[10]Indeed, other issues such as the Table Maker Dilemma may make it necessary to approximate y with much smaller relative errors

2.4 Ziv's rounding test, `crlibm` and the "magic constant"

In 1991 A. Ziv published the so-called "Accurate Portable Mathlib" (or `libultim`) library of correctly ("to the last bit") rounded elementary functions and the companion paper [111].

In that article, he elaborates on a (possibly) two-staged process to compute an approximation:

- a "quick" and "low precision" algorithm that yields two numbers f_ℓ and b (in Ziv's notations); b being a few orders of magnitude smaller than f_ℓ and the result being $\mathrm{RN}(f_\ell + b)$, if the exact sum $f_\ell + b$ does not exhibits the stigmas of a hard-to-round case (see §1.6.3.2, p. 59 for hardness-to-round semiology);
- a set of increasingly high precision but slower algorithms, until a sufficient precision is reached.

The article does not enter into details on how to perform the check in the first step.

The problem is mentioned, under the form we have stated it, and is dealt with more details in another article [23] by F. de Dinechin, A. V. Ershov and N. Gast in 2005. They trace it back to Ziv's work on `libultim`, referring to the Ziv inspired source code of the "IBM Accurate Mathematical Library" (see, for a contemporary instance, the `e_asin.c` file in version 2.20 of the GNU C Library[106]), rather than to Ziv's article itself.

Their solution to the problem is a test in the form of a product with a "magic constant" e:

$$y_h = \mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e)) \Rightarrow y_h = \mathrm{RN}(y) \tag{2.6}$$

In the sequel, we will call it *Ziv's rounding test*.

It is discussed also at length again in the `crlibm` [21] documentation. Possible values of e and proofs are given.

Nevertheless, this could not be considered as the last word on the issue. Notably because:

- the question was only examined in the case of IEEE `binary64` values for y_h and y_ℓ (and the rounding to IEEE `binary64`);
- the subnormal situation was not dealt with;
- the documentation itself mentioned that FMA usage should be investigated (and erroneously leaned to the opinion that it should not be used at all); for a presentation about FMA, see §1.5.4, p. 48.

Before proceeding to the next section we will make the following remarks, based on the monotonicity of the function $y \to \mathrm{RN}(y)$.

Remark 2. if (y, y_h, y_ℓ) is a false negative for Ziv's rounding test with constant e, then it will be a false negative with any constant $e^{'} > e$ (i.e. the larger the constant e, the more false negatives will show up).

Remark 3. if (y, y_h, y_ℓ) is a false positive for Ziv's rounding test with constant e, then it will be a false positive with any constant $e^{'} < e$ (i.e. the smaller the constant e, the more false positives will show up).

It follows that our goal is, given ϵ and $\mathbb{F}_{2,p,emin,emax}$, to find the smallest value of e that yields no false positive.

The reader has noticed that in our goal definition we surreptitiously limit ourselves to the radix-2 case. A general theory (at least an extension to radix-10) of Ziv's rounding test would of course be more intellectually satisfactory. Indeed, as will be seen in the sequel, this matter is very radix-dependent. We face here a contradiction between tight bounds on e for some particular radix or "universal" but larger ones. Aside from its theoretical interest, the clarification of Ziv's rounding test has practical applications, mainly in the field of scientific computing. At the moment (and for the foreseeable future) these will essentially be carried on in a radix-2 context. Nevertheless the outlined approach will be as general as possible (and as practical) in three directions:

- the floating-point precision p;
- the relative error ϵ;
- the consideration given to the subnormal case.

Furthermore, insights are given for situations where the FMA operation is available.

All together, this explains why we focus on this case. At the same time the acknowledgement of this limitation points towards possible directions for future work.

2.5 Breaking the magic spell: mostly provable near optimal constants

As we have seen, Ziv's rounding test is used in some form or another in various correctly-rounded elementary functions floating-point libraries. We would like to improve on the efficiency and reliability of these libraries by providing an optimal (or, at least, near-optimal) value of constant e. This could also be used in future (or other existing) libraries that would contemplate using Ziv's rounding

test as it is particularly appealing thanks to its simplicity: three[11] elementary operations (two only, when an FMA instruction is available).

For $y \in \mathbb{R}$, y_h, $y_\ell \in \mathbb{F}_{2,p,e_{\min},e_{\max}}$ and $\epsilon < 2^{-p-1}$ verifying:

- $|y_h + y_\ell - y| < |\epsilon \times y|$; Notice: this implies that y_h and y have the same sign; without loss of generality will assume them positive;
- $y_h = \mathrm{RN}(y_h + y_\ell)$.

We want the value of e to be such that:

$$y_h = \mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e)) \Rightarrow y_h = \mathrm{RN}(y)$$

More precisely, from a constructive perspective, we are looking for an expression that links e to the other global parameters of the problem (as p and ϵ), and such that we can compute its (near) optimal value once for all for this set of parameters.

2.5.1 Preliminary properties

We will start by a set of properties that will be used in the forthcoming developments. In most of them we try to find bounds on different aspects of the problem that will only use what is actually known to us: y_h, y_ℓ, p, and ϵ.

Notice first that, in our introductory sketches and discussion, we extensively used the $[y\times(1-\epsilon), y\times(1+\epsilon)]$ segment (the "y segment") in a somewhat omniscient "God's view" of the problem. As "mere mortals" we cannot do that here since this segment is unknown to us, just as the value of y is. Our first task will be to give a bound on $|y_h - y|$ based on what we actually know.

$$|y_h - y| = |y_h + y_\ell - y - y_\ell| \leqslant |(y_h + y_\ell) - y| + |y_\ell|.$$

Hence, from (2.4):

$$|y_h - y| < \epsilon \times |y| + |y_\ell|. \tag{2.7}$$

Notice now that from the definition of the relative error, $\left|\dfrac{y-(y_h+y_\ell)}{y}\right|$, and as we assume $y \geqslant 0$:

$$y < \frac{y_h + y_\ell}{1-\epsilon}. \tag{2.8}$$

[11] One product, one addition, one comparison. Product and addition are fused in the presence of FMA

Combining (2.7) and (2.8), we can deduce:

$$|y_h - y| < \frac{\epsilon}{1-\epsilon} \times (y_h + y_\ell) + |y_\ell|. \tag{2.9}$$

This gives us the wanted bound on $|y_h - y|$.

This bound is the sum of two terms. The second one is an absolute term (it also is an absolute value but this is not the point here). The first one is a product between the computed value, $y_h + y_\ell$, and the factor $\frac{\epsilon}{1-\epsilon}$ that we could call the "reverse relative error" (as opposed to the "direct relative error" ϵ).

Put in another way, we could say that $\frac{\epsilon}{1-\epsilon}$ is a relative error in the "$y_h + y_\ell$ perspective" whereas ϵ is the relative error in the "y perspective". It really makes a difference, as can be seen in the following example, also illustrated by Figure 2-6.

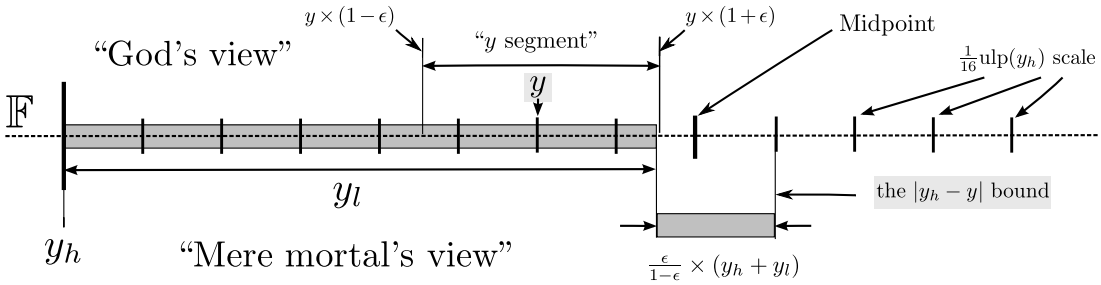

Figure 2-6: "God's view" (at the top) vs "mere mortal's view" (at the bottom) of an approximation and rounding problem with $y \geqslant 0$. In this extreme case $y_h + y_\ell$ is very close to the upper bound of the "y segment". Since the bound computed in the "mortal's view" realm by adding $\frac{\epsilon}{1-\epsilon} \times (y_h + y_\ell)$ to $y_h + y_\ell$ is beyond the midpoint, we can do nothing but erroneously conclude that the computation should be restarted with a higher accuracy. This layout shows that false negatives can not be totally avoided.

Let us now examine a numerical example, with $y \geqslant 0$, based on the following parameters:

- precision $p = 53$ (IEEE-754 binary64) ;
- $\epsilon = 2^{-56}$ (unrealistically large, but for demonstration purposes).

Suppose now that we have the following values:

- $y_h = 1.5$, or $1.1 \cdot 2^0$, in "binary";
- $y_\ell = 2^{-54} + 2^{-55} + 2^{-56} + 2^{-57}$, i.e. $1.111 \cdot 2^{-54}$;

- $y = 1.5 + 2^{-54} + 2^{-55}$.

It is very easy to verify that the following conditions:

- $\epsilon < 2^{-53-1}$;
- $(1 - \epsilon)\, y < y_h + y_\ell < (1 + \epsilon)\, y$;
- $y_h = \mathrm{RN}(y_h + y_\ell)$

are matched.

We have:

- $\epsilon \times y \approx 1.1 \cdot 2^{-56} + 1.1 \cdot 2^{-110} \approx \frac{3}{32}\mathrm{ulp}(y_h)$;
- $\frac{\epsilon}{1-\epsilon}(y_h + y_\ell) \approx 1.1 \cdot 2^{-56} + 1.001 \cdot 2^{-109} \approx \frac{3}{32}\mathrm{ulp}(y_h)$.

The difference between these two values is small but cannot be ignored. But the most important element to take into consideration is the starting point of the bound computation, in "y segment" case $\epsilon \times y$ is added to the value of y while in the other case, $\frac{\epsilon}{1-\epsilon}(y_h + y_\ell)$ is added at the "tip" of $y_h + y_\ell$.

As can be seen from Figure 2-6 reasoning on bounds will, in our example, produces a false negative (a situation where $y_h = \mathrm{RN}(y)$ but where we conclude that the computation should be restarted with a higher accuracy). It also shows why, in a very informal and pictorial form, without additional information, a perfect (no false positives, no false negatives) rounding test is impossible.

We can also see why some increase in the accuracy of the approximation may pay for itself as the number of false negative situations (and the number of extra computations associated with them) decreases. We will come back to this topic with experimental results.

2.5.1.1 The different cases

Our goal is to find an expression to compute e. From our problem statement, we know that $e \geqslant 1$. As we will see, different situations regarding:

- the signs of $y - y_h$ and y_ℓ are different;
- if y_h is a power of 2 or not when the signs of $y - y_h$ and y_ℓ are the same

will give us different lower bounds for e. The value we will actually use will be the maximum of all these bounds.

If $y - y_h$ and y_ℓ have different signs

The following property simplifies the problem, but it should not be regarded *per se* as a possible test. In fact, as we will discover, this situation is of no help for the computation of e since it places no new constraint upon it.

Suppose $y - y_h$ and y_ℓ have different signs and, as stated before, $y_\ell < \frac{1}{2} \times \text{ulp}(y_h)$ and $\epsilon < 2^{-p-1}$.

Property 2.1. *if $y - y_h$ and y_ℓ have different signs then $y_h = RN(y)$.*

Proof. If $y - y_h$ and y_ℓ have different signs then either:

- $y \leqslant y_h \leqslant y_h + y_\ell$, if $y_h < 0$ and $y_\ell \geqslant 0$;
- $(y_h + y_\ell) \leqslant y_h \leqslant y$, if $y_h > 0$ and $y_\ell \leqslant 0$ see Figure 2-7.

At any rate y is closer to y_h than to $(y_h + y_\ell)$. Combined with (2.4), this also implies that $|y_h - y| < |y| \times \epsilon$.

Since $|\epsilon| < 2^{-p-1}$, this implies that $y_h = \text{RN}(y)$. □

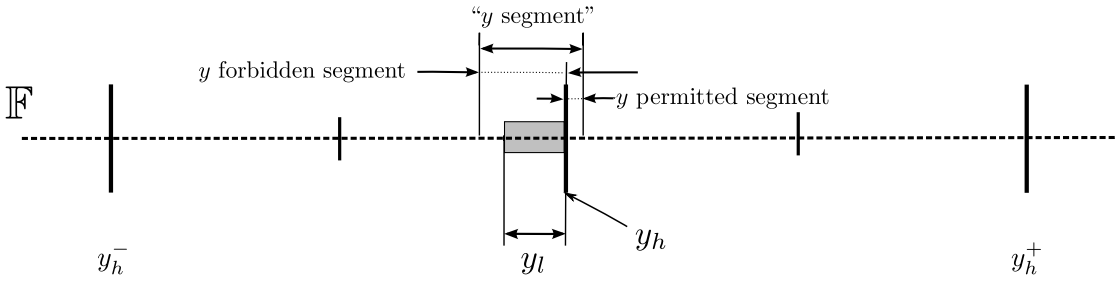

Figure 2-7: We are in the second case mentioned in the proof of Property 2.1: $(y_h + y_\ell) \leqslant y_h \leqslant y$, if $y_h > 0$ and $y_\ell \leqslant 0$. In this situation, y can only live in the "y permitted segment". Hence, and despite a large value of ϵ, $y_h = \text{RN}(y)$.

As a consequence, in this situation, with any value of $e \geqslant 1$

$$y_h = \text{RN}(y_h + \text{RN}(y_\ell \times e)) \Rightarrow y_h = \text{RN}(y).$$

No new lower bound, other than that already present in the problem statement, can be computed from this situation.

When y_h is not a power of 2

From now on, we consider that $y - y_h$ and y_ℓ have the same sign. Let us give a set of properties that apply when y_h is not a power of two. Another one for the case where y_h is a power of two will be given in the next subsection.

The first one gives a bound on y_ℓ, solely based on the values of y_h and ϵ, leaving y out of the picture, for $y_h = \mathrm{RN}(y)$.

Property 2.2. *Assume that y_h is not a power of two. If*

$$|y_\ell| \leqslant \frac{1}{2}\mathrm{ulp}(y_h) \times (1 - \epsilon) - y_h \times \epsilon$$

then $y_h = \mathrm{RN}(y)$

See Figure 2-8

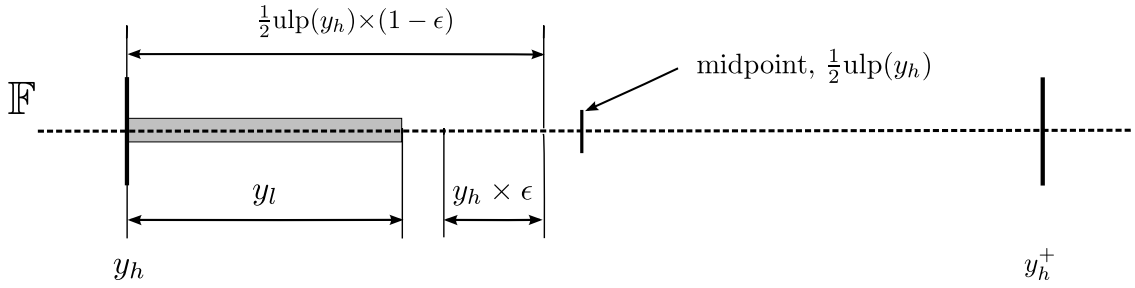

Figure 2-8: $|y_\ell| \leqslant \frac{1}{2}\mathrm{ulp}(y_h) \times (1 - \epsilon) - y_h \times \epsilon$. Here y_h and y_ℓ are both > 0.

Proof. If $|y_\ell| \leqslant \frac{1}{2}\mathrm{ulp}(y_h) \times (1 - \epsilon) - y_h \times \epsilon$, we can also say that

$$(1 - \epsilon + \epsilon) \times |y_\ell| \leqslant \frac{1}{2}\mathrm{ulp}(y_h) \times (1 - \epsilon) - y_h \times \epsilon.$$

Then

$$\left(1 + \frac{\epsilon}{1 - \epsilon}\right) \times |y_\ell| \leqslant \frac{1}{2}\mathrm{ulp}(y_h) - \frac{\epsilon}{1 - \epsilon} \times y_h.$$

In turn

$$\left(\frac{\epsilon}{1 - \epsilon}\right) \times y_\ell + |y_\ell| \leqslant \frac{1}{2}\mathrm{ulp}(y_h) - \frac{\epsilon}{1 - \epsilon} \times y_h.$$

This implies:

$$\left(\frac{\epsilon}{1 - \epsilon}\right) \times (y_h + y_\ell) + |y_\ell| \leqslant \frac{1}{2}\mathrm{ulp}(y_h).$$

Eventually, using (2.9):

$$|y_h - y| < \frac{1}{2}\mathrm{ulp}(y_h).$$

Hence:

$$y_h \;=\; \mathrm{RN}(y).$$

□

Are there many y_ℓ values left with the bound we impose on it?

It is easy to see how the value of ϵ impacts that of the $\frac{1}{2}\mathrm{ulp}(y_h) \times (1-\epsilon)$ term. Since $\epsilon < 2^{-p-1}$, under no circumstance it can be smaller than $\frac{1}{2}\mathrm{ulp}(y_h) - (\frac{1}{2}\mathrm{ulp}(y_h) \times 2^{-p-1})$. This is indeed smaller than $\frac{1}{2}\mathrm{ulp}(y_h)$ but by a narrow margin, for practical values of p.

Notice conversely how $|y_h \times \epsilon|$ is bounded. Suppose y_h belongs to the $[2^m, 2^{m+1})$ binade and $\epsilon \;=\; 2^{-a}$ where $a \;>\; p+1$. We have:

$$\mathrm{ulp}(y_h) \;=\; 2^{m-p+1}.$$

We also have

$$|y_h \times \epsilon| \;<\; 2^{m+1} \times 2^{-a} \;<\; 2^{m+1} \times 2^{-p-1} \;=\; 2^{m-p} \;=\; \frac{1}{2}\mathrm{ulp}(y_h).$$

For very large values of ϵ (a close to $-p-1$) and large values of y_h (in the sense of close to the binade upper bound 2^{m+1}) there may be no acceptable value of y_ℓ left . But as soon a ϵ decreases, this gives more and more room for y_ℓ.

We are going now to find a bound on $|y_\ell|$ based on some hypothetical values of $e \;>\; 1$, the constant used in Ziv's rounding test, should it exist. The other parameters of the bound expression will be problem parameters as well (precision p, approximation accuracy ϵ).

At some point, our reasoning only works if we assume that $\frac{1}{2}\mathrm{ulp}(y_h)$ is in the normal range. That's why this condition appears in our hypothesis list.

Property 2.3. *Assume that y_h is not a power of 2. Let e be a nonnegative floating-point number. If $\frac{1}{2}\mathrm{ulp}(y_h)$ is in the normal range, then if*

$$y_h \;=\; \mathrm{RN}(y_h \,+\, \mathrm{RN}(y_\ell \times e)),$$

then

$$|y_\ell| \;\leqslant\; \frac{(1 \,+\, 2^{-p}) \times \mathrm{ulp}(y_h)}{2e}.$$

Proof. From $y_h = \mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e))$, since y_h is not a power of two and from Property 1.4, p. 41:

$$y_h - \frac{1}{2}\mathrm{ulp}(y_h) \leqslant y_h + \mathrm{RN}(y_\ell \times e) \leqslant y_h + \frac{1}{2}\mathrm{ulp}(y_h).$$

This implies

$$|\mathrm{RN}(y_\ell \times e)| \leqslant \frac{1}{2}\mathrm{ulp}(y_h). \tag{2.10}$$

From Property 1.3, p. 41, we evidently have

$$|\mathrm{RN}(y_\ell \times e)| - \frac{1}{2}\mathrm{ulp}(\mathrm{RN}(y_\ell \times e)) \leq |y_\ell \times e| \leq |\mathrm{RN}(y_\ell \times e)| + \frac{1}{2}\mathrm{ulp}(\mathrm{RN}(y_\ell \times e)).$$

As obviously, we have:

$$\frac{1}{2}\mathrm{ulp}\left(\mathrm{RN}(y_\ell \times e)\right) \leqslant \frac{1}{2}\mathrm{ulp}\left(\frac{1}{2}\mathrm{ulp}\left(y_h\right)\right).$$

From the definition of the ulp() function (def. 1.7, p. 39) and (2.10), and since $\frac{1}{2}\mathrm{ulp}(y_h)$ is in the normal range

$$\mathrm{ulp}\left(\frac{1}{2}\mathrm{ulp}(y_h)\right) = 2^{-p+1}\frac{1}{2}\mathrm{ulp}\left(y_h\right).$$

Finally

$$\frac{1}{2}\mathrm{ulp}\left(\mathrm{RN}(y_\ell \times e)\right) \leqslant 2^{-p} \times \frac{1}{2}\mathrm{ulp}\left(y_h\right). \tag{2.11}$$

From (2.10) and (2.11), it follows:

$$|y_h \times e| \leqslant (1 + 2^{-p}) \times \frac{1}{2}\mathrm{ulp}(y_h).$$

□

Combining Properties 2.2 and 2.3 if

$$\frac{(1 + 2^{-p}) \times \mathrm{ulp}(y_\ell)}{2e} \leqslant \frac{1}{2}\mathrm{ulp}(y_h)\,(1 - \epsilon) - y_h \times \epsilon, \tag{2.12}$$

then $y_h = \mathrm{RN}(y_h + \mathrm{RN}(y_h \times e)) \Rightarrow y_h = \mathrm{RN}(y)$.

Condition 2.12 is equivalent to:

$$e \geqslant \frac{(1 + 2^{-p}) \times \mathrm{ulp}(y_h)}{\mathrm{ulp}(y_h)\,(1 - \epsilon) - 2y_h \times \epsilon}.$$

Notice that this lower bound on e still depends on the value of y_h. It would be convenient to get rid of it in order to obtain one that would only depend on parameters of the problem. The

maximum value of this bound would give us a minimum value for e.

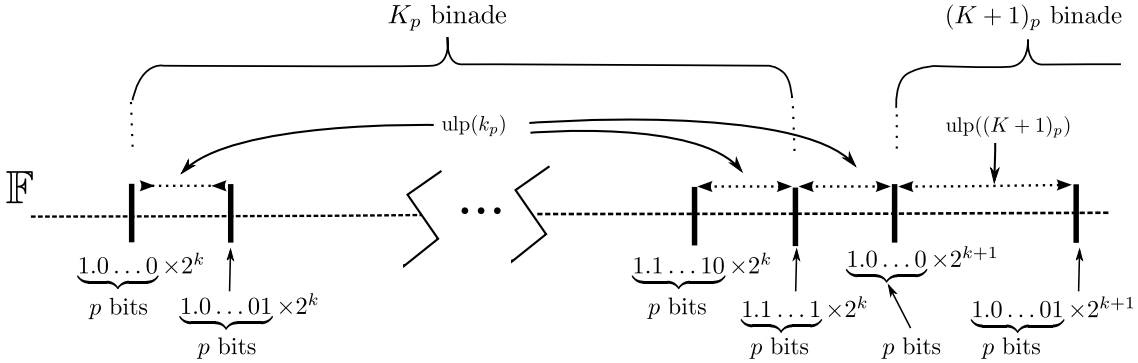

Figure 2-9: Both ends of the positive binade 2^k in precision-p (referenced as the K_p binade) are represented here. The value of the ulp() function is constant among the members X of this binade and equal to 2^{k-p+1}. If we define ξ as $\text{ulp}(X) = \xi \times X \times 2^{-p}$, its possible values are 2, for the first element (2^k) of the binade and almost 1 for the last element ($2^{k+1} - 2^{k-p+1}$), with a set of intermediate values in between.

If y_h is in the normal range, we can write:

$$\text{ulp}(y_h) = \xi \times y_h \times 2^{-p}, \text{ with } 1 < \xi \leqslant 2.$$

See Figure 2-9.

Hence:

$$\frac{(1 + 2^{-p}) \times \text{ulp}(y_h)}{\text{ulp}(y_h) \times (1 - \epsilon) - 2y_h \times \epsilon} = \frac{(1 + 2^{-p}) \times \xi}{\xi \times (1 - \epsilon) - 2^{p+1} \times \epsilon}.$$

Our lower bound can be rewritten as:

$$e \geqslant \frac{(1 + 2^{-p}) \times \xi}{\xi \times (1 - \epsilon) - 2^{p+1} \times \epsilon}.$$

And we can try to find for which values of ξ the right hand expression is maximized.

First of all, we want to make sure our fractional bound is always defined and positive, hence the denominator of the expression must be strictly positive (the numerator always is), setting conditions as necessary.

In other words:

$$\xi\,(1 - \epsilon) - 2^{p+1} \times \epsilon > 0.$$

The worst case is when ξ is close to 1. Let us set $\xi = 1$. We then have:

$$1 - \epsilon > 2^{p+1} \times \epsilon$$

We can deduce:

$$\frac{1}{2^{p+1} + 1} > \epsilon. \tag{2.13}$$

Incidentally, this condition is very close to that on ϵ in the problem statement. The small difference comes from the fact that we identified ξ to 1, which is never actually the case.

Let us consider now function $\Xi(\xi)$ defined as:

$$\Xi(\xi) = \frac{(1 + 2^{-p})\,\xi}{(1 - \epsilon)\,\xi - 2^{p+1} \times \epsilon}. \tag{2.14}$$

We can rewrite it as

$$\Xi(\xi) = \frac{a\xi + b}{c\xi + d}$$

where:

- $a = 1 + 2^{-p}$;
- $b = 0$;
- $c = 1 - \epsilon$;
- $d = -2^{p+1}\,\epsilon$.

From calculus rules, we have

$$\Xi'(\xi) = \frac{\begin{vmatrix} a & b \\ c & d \end{vmatrix} \xi}{(c\xi + d)^2}.$$

We are only interested in the sign of the derivative. The denominator is positive, which leaves us with the question of the sign of the numerator, made of the single $ad\xi$ term since $b = 0$, to deal with. We have:

$$ad\xi = (1 + 2^{-p}) \times (-2^{p+1}\,\epsilon)\,\xi < 0$$

for

$$1 + 2^{-p} \geqslant 0,\ -2^{p+1}\,\epsilon < 0 \text{ and } \xi \in (1, 2].$$

Under Condition 2.13, function Ξ is decreasing over the interval $[1, 2]$.
Hence it reaches its maximum at 1.

From the preceding we can deduce the following property:

Property 2.4. *Assume :*

- y_h *is not a power of* 2*;*
- $\frac{1}{2}\text{ulp}(y_\ell)$ *is in the normal range.*

If

$$e \;\geq\; \frac{1+2^{-p}}{1-\epsilon-2^{p+1}\epsilon}$$

then

$$y_h \;=\; \text{RN}(y_h + \text{RN}(y_\ell \times e)) \quad\Rightarrow\quad y_h \;=\; \text{RN}(y)$$

When y_h is a power of 2

This case will be broken down into a string of sub-cases.

On some instances detailed below the previous result can be applied straight away.

We will start with the situation where $y \;\geqslant\; y_h$ and $y_\ell \;\geqslant\; 0$, still remembering of the $y \;\geqslant\; 0$ and $y_h \;\geqslant\; 0$ assumptions (see Figure 2-10). This "right hand side" of a power of 2 configuration does not differ from the non-power of 2 situation. We have:

$$y - y_h \;\leqslant\; \frac{1}{2}\text{ulp}(y_h) \quad\Rightarrow\quad y_h \;=\; \text{RN}(y) \tag{2.15}$$

and, whatever the value of $e \;\geqslant\; 1$ is,

$$\text{if } y_h \;=\; \text{RN}(y_h + \text{RN}(y_\ell \times e)) \text{ then } y_h \;\leqslant\; y_h + \text{RN}(y_\ell \times e) \;\leqslant\; y_h + \frac{1}{2}\text{ulp}(y_h). \tag{2.16}$$

Expression (2.15) allows for the validity of Property 2.2.
Expression (2.16) allows for that of Property 2.3.
Hence Property 2.4 still holds.

What happens now when

- $y \;\leqslant\; y_h$ and $y_\ell \;\geqslant\; 0$ or
- $y \;\geqslant\; y_h$ and $y_\ell \;\leqslant\; 0$?

We are in the situation where $y - y_h$ and y_ℓ have different signs. Property 2.1 applies: $y_h \;=\; \text{RN}(y)$.

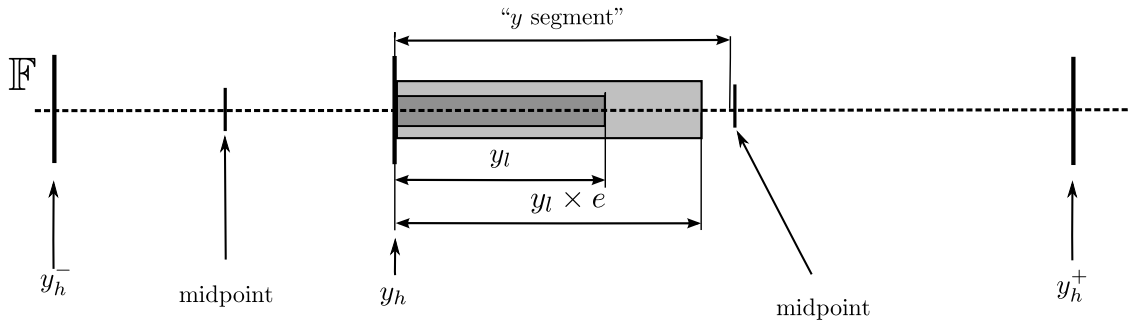

Figure 2-10: y_h is a power of 2. Here $y \geqslant y_h \geqslant 0$ and $y_\ell \geqslant 0$. Remember $e \geqslant 1$.
In the depicted situation $y - y_h \leqslant \frac{1}{2}\mathrm{ulp}(y_h)$ hence $y_h = \mathrm{RN}(y)$.
We also have: if $(y_h = \mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e))$ *then* $y_h \leqslant y_h + \mathrm{RN}(y_\ell \times e) \leqslant y_h + \frac{1}{2}\mathrm{ulp}(y_h)$.

We are left now with the case where $y \leqslant y_h$ and $y_\ell \leqslant 0$. Since y_h is a power of 2, we are in the peculiar instance where $\mathrm{ulp}(y_h) = 2^{-p+1}y_h$.

Property 2.5. *Assume that y_h is a power of 2, that $y \leqslant y_h$, and $y_\ell \leqslant 0$. If*

$$|y_\ell| \leqslant y_h \times \frac{2^{-p+1} - \dfrac{\epsilon}{1-\epsilon}}{1 - \dfrac{\epsilon}{1-\epsilon}} \tag{2.17}$$

then $y_h = \mathrm{RN}(y)$.

This property plays the same part as Property 2.2 in the non-power of 2 case. But since the "rounding distance to y_h" is twice as small on the "left side" as it is on the "right side", the condition on y_ℓ has to be stiffer (see Figure 2-11).

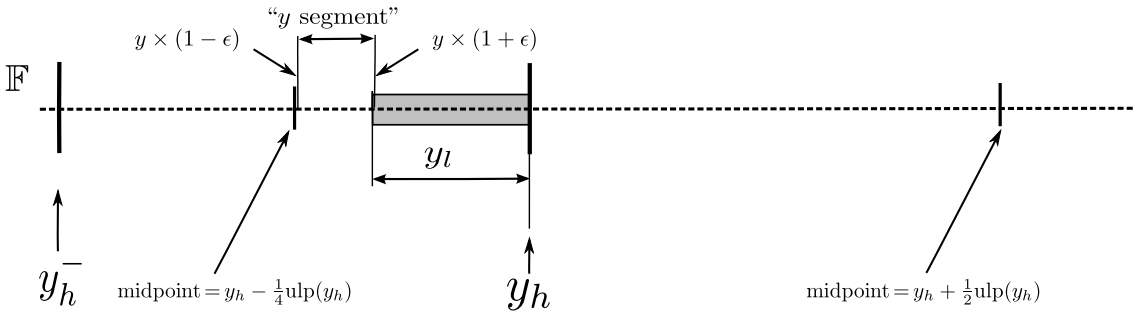

Figure 2-11: Here y_h is a power of 2, $y \leqslant y_h$ and $y_\ell \leqslant 0$. We also have $|y_\ell| \leqslant y_h \times \dfrac{2^{-p+1} - \dfrac{\epsilon}{1-\epsilon}}{1 - \dfrac{\epsilon}{1-\epsilon}}$.
This guarantees, even in the "worst case" depicted here, where y_ℓ is very close to the upper bound of the "y segment" (but still inside it), that we have $y_h = \mathrm{RN}(y)$.

Notice that Condition 2.17 can not be satisfied if $2^{-p+1} - \dfrac{\epsilon}{1-\epsilon} \leqslant 0$. It implies again, see (2.13), that ϵ should be smaller than $\dfrac{1}{1+2^{p+1}}$.

Proof. If (2.17) is satisfied, then:

$$|y_\ell| \times \left(1 - \frac{\epsilon}{1-\epsilon}\right) \leqslant y_h \times \left(2^{-p-1} - \frac{\epsilon}{1-\epsilon}\right).$$

In other terms:

$$|y_\ell| + (y_h - |y_\ell|) \times \frac{\epsilon}{1-\epsilon} \leqslant y_h \times 2^{-p-1}.$$

Notice that, from our assumptions, $|y_\ell| = -y_\ell$, so we can state:

$$|y_\ell| + (y_h + y_\ell) \times \frac{\epsilon}{1-\epsilon} \leqslant y_h \times 2^{-p-1}.$$

That implies, using (2.9) and $\mathrm{ulp}(y_h) = 2^{-p+1}y_h$ (as y_h is exactly a power of two), that:

$$0 \leqslant y_h - y \leqslant y_h \times 2^{-p-1} = \frac{1}{4}\mathrm{ulp}(y_h).$$

Finally:

$$y_h = \mathrm{RN}(y).$$

Notice that condition $y_h - y \leqslant \frac{1}{4}\mathrm{ulp}(y_h)$ is absolutely necessary here, since from the milder (and "usual") condition $y_h - y \leqslant \frac{1}{2}\mathrm{ulp}(y_h)$ we could not conclude $y_h = \mathrm{RN}(y)$, as y_h is exactly a power of two. □

We have obtained a bound on $|y_\ell|$ in terms of y_h, the precision p and the approximation accuracy ϵ. We are going to give another bound on $|y_\ell|$ in terms of y_h, the precision p, and on some hypothetical e that would be used in Ziv's rounding test.

Property 2.6. *Assume that y_h is a power of 2, $y \leqslant y_h$, $y_\ell \leqslant 0$, and that $2^{-p-1}y_h = \frac{1}{4}\mathrm{ulp}(y_h)$ is a normal number. If*

$$y_h = \mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e)), \tag{2.18}$$

then

$$|y_\ell| \leqslant \frac{(1+2^{-p}) \times 2^{-p-1} \times y_h}{e}. \tag{2.19}$$

Proof. As y_h is a power of 2, (2.18) implies:

$$y_h - \frac{1}{4}\mathrm{ulp}(y_h) \leqslant y_h + \mathrm{RN}(y_\ell \times e) \leqslant y_h.$$

Since $\text{ulp}(y_h) = 2^{-p+1}y_h$ exactly, as y_h is a power of 2, it follows:

$$y_h - 2^{-2} \times 2^{-p+1} \times y_h \leqslant y_h + \text{RN}(y_\ell \times e) \leqslant y_h.$$

That is:

$$-2^{-p-1} \times y_h \leqslant \text{RN}(y_\ell \times e) \leqslant 0.$$

If $2^{-p-1}y_h$ is a normal number, then:

$$|y_\ell \times e| \leqslant 2^{-p-1}y_h + \frac{1}{2}\text{ulp}(2^{-p-1}y_h).$$

More precisely:

$$|y_\ell \times e| \leqslant 2^{-p-1}y_h + 2^{-p} \times 2^{-p-1}y_h = (1 + 2^{-p}) \times 2^{-p-1}y_h.$$

Finally (remembering that $e > 0$):

$$|y_\ell| \leqslant \frac{(1 + 2^{-p}) \times 2^{-p-1}y_h}{e}.$$

□

We are ready now, combining Properties 2.5 and 2.6 and previous remarks, to give a bound on e in terms of precision p and accuracy ϵ.

Property 2.7. *Assume that y_h is a power of 2, that $\frac{1}{4}\text{ulp}(y_h)$ is a normal number, and that $\epsilon < \frac{1}{1 + 2^{p+1}}$. If*

$$e \geqslant \frac{(1 + 2^{-p}) \times (1 - 2\epsilon)}{1 - \epsilon - 2^{p+1}\epsilon} \tag{2.20}$$

then $y_h = \text{RN}(y_h + \text{RN}(y_\ell \times e)) \Rightarrow y_h = \text{RN}(y)$.

Proof. Since most of the cases have been dealt with before by equating them to generic cases or non-power of 2 cases, all we have to do here is to check the $y \leqslant y_h$ and $y_\ell \leqslant 0$ case.
Properties 2.5 and 2.6 both give an upper bound on $|y_\ell|$. We are going to combine them in order to find the wanted lower bound on e. Since e is in the denominator of Expression (2.19), if we let it shrink to arbitrary values, the upper bound on y_ℓ given in (2.19) will overrun that given by (2.17). To prevent that we have to bound Expression (2.19) by Expression (2.17):

$$\frac{(1 + 2^{-p}) \times 2^{-p-1}y_h}{e} \leqslant y_h \times \frac{2^{-p+1} - \dfrac{\epsilon}{1 - \epsilon}}{1 - \dfrac{\epsilon}{1 - \epsilon}}.$$

The idea is, from that inequality, to find a lower bound for e. From there, it turns out that with

elementary calculus, we fall back on Expression (2.20) which is such a bound. □

2.5.2 The values of e that allow for a correct Ziv's rounding test

The bounds obtained for e in Properties 2.4 (when y_h is not a power of 2) and 2.7 (when y_h actually is a power of 2) are different, $e \geqslant \frac{(1 + 2^{-p})}{1 - \epsilon - 2^{p+1}\epsilon}$ and $e \geqslant \frac{(1 + 2^{-p}) \times (1 - 2\epsilon)}{1 - \epsilon - 2^{p+1}\epsilon}$ respectively. From that, taking the largest of the two, we can now deduce

Theorem 2.1. *Assume that:*

- $\epsilon < \frac{1}{1 + 2^{p+1}}$;
- $|(y_h + y_\ell) - y| < \epsilon \times |y|$;
- $y_h = \mathrm{RN}(y_h + y_\ell)$;
- y_h *is a floating-point number such that* $\frac{1}{4}\mathrm{ulp}(y_h)$ *is in the normal range;*

if

$$e \geqslant \frac{(1 + 2^{-p})}{1 - \epsilon - 2^{p+1}\epsilon}$$

then

$$y_h = \mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e)) \Rightarrow y_h = \mathrm{RN}(y).$$

This upper bound is somewhat theoretical in the sense that we did not care to check whether, for some floating-point system $\mathbb{F}$, $\frac{(1 + 2^{-p})}{1 - \epsilon - 2^{p+1}\epsilon}$ actually is an element of $\mathbb{F}$.

It turns out that this expression can not be decently computed inside $\mathbb{F}$. For instance, in this context, sub-expressions like $(1 + 2^{-p})$ will always round to 1. We can't rely on the $2^{p+1}\epsilon$ to save our neck for "small" values of ϵ ($\epsilon \leqslant 2^{-2p-1}$). Neither shall we exactly compute $1 - \epsilon$ for any value of $\epsilon < \frac{1}{1+2^{p+1}}$, as the hypothesis of Theorem 2.1 mandate.

Expression $\frac{1 + 2^{-p}}{1 - \epsilon - 2^{p+1}\epsilon}$ will then have to be computed in a higher precision floating-point arithmetic system (or rational arithmetic system, provided that ϵ is expressed as a rational number) than that of $\mathbb{F}$ and then the result will be rounded up into $\mathbb{F}$ to yield an acceptable value for e (that is an element of $\mathbb{F}$ larger than or equal to $\frac{(1 + 2^{-p})}{1 - \epsilon - 2^{p+1}\epsilon}$).

Hence, in practice, the rounding test will be performed with:

$$e = \mathrm{RU}\left(\frac{1 + 2^{-p}}{1 - \epsilon - 2^{p+1}\epsilon}\right)$$

as computed in some high precision arithmetic system at hand and rounded to nearest to an element of $\mathbb{F}$. This is not actually an issue since, in the context of function approximation, one must make use of some extended precision framework (relatively to that of $\mathbb{F}$) to compute the function approximants with a sufficient accuracy, and that computation is done once for all when designing a function evaluation program that, itself, will likely run millions of times.

2.5.3 How good is this value?

We will not make the suspense last too long. The bound is excellent but we cannot prove it to be optimal in any circumstance, even under the conditions we have given (such as $\frac{1}{4}\text{ulp}(y_h)$ should be in the normal range). However we will provide here some evidence to support our near-optimality claims. Experimental result will add up later on.

It is important to stress here that corner cases matter a lot and should not be seen as mere curiosities. Ziv's rounding test is meant to be quick. Going through a contrived nest of conditionals thwarts the whole purpose. The values for e should work without any other case splitting than the one we introduced by requesting that the value of $\frac{1}{4}\text{ulp}(y_h)$ be in the normal range. We will come back later to this subject.

2.5.3.1 For some values of ϵ and a generic value for p it is possible to prove our e near-optimal

We will first consider a somewhat particular form of ϵ. The peculiarities stem from technical reasons that will become clear in the sequel.

Let us consider an ϵ of the form

$$\epsilon \;=\; \frac{2^{-p-k+1} \;+\; 2^{-3p}}{2 \;-\; 2^{-p} \;+\; 2^{-3p}},$$

for $1 \;<\; k \;<\; \dfrac{p}{2}$.

Let us set $\upsilon \;=\; 2^{-p}$.

We can rewrite our expression for ϵ as

$$\epsilon \;=\; \frac{2^{-k+1}\upsilon \;+\; \upsilon^3}{2 \;-\; \upsilon \;+\; \upsilon^3}.$$

For that particular expression of ϵ and using the above convention for υ, the minimum value of e, as given in Theorem 2.1 becomes

$$e_{\min} \;=\; \frac{1 + 2^{-p}}{1 - \epsilon - 2^{p+1}\epsilon} \;=\; \frac{1+\upsilon}{1 - \left(1 + \dfrac{2}{\upsilon}\right) \times \dfrac{2^{-k+1}\upsilon + \upsilon^3}{2 - \upsilon + \upsilon^3}}.$$

This expression can be considered as a function of υ. Its Taylor development can be (hopefully) computed at $\upsilon = 0$ with a computer algebra system.

As it turns out, after some simplifications,

$$e_{\min} \;=\; \frac{(1 + 2^{-p})}{1 - \epsilon - 2^{p+1}\epsilon} \;=\; \left(\frac{2^k}{2^k - 2}\right) + \left(\frac{4^k}{(2^k - 2)^2}\right) \times \upsilon - \left(\frac{-4^k + 2^{1+k} - 8^k}{(2^k - 2)^3}\right) \times \upsilon^2 + \ldots$$

After replacing υ by its value, one gets

$$e_{\min} \;=\; \frac{(1 + 2^{-p})}{1 - \epsilon - 2^{p+1}\epsilon} \;=\; \left(\frac{2^k}{2^k - 2}\right) + \left(\frac{4^k}{(2^k - 2)^2}\right) \times 2^{-p} - \left(\frac{-4^k + 2^{1+k} - 8^k}{(2^k - 2)^3}\right) \times 2^{-2p} + \ldots$$

The first term $\left(\dfrac{2^k}{2^k - 2}\right)$ can be rewritten as

$$\left(\frac{2^k}{2^k - 2}\right) \;=\; \frac{1}{1 - 2^{1-k}}$$

that itself can be expanded to

$$\frac{1}{1 - 2^{1-k}} \;=\; 1 + 2^{1-k} + 2^{2-2k} + 2^{3-3k} + \ldots.$$

The difference between $e_{\min}$ and $1 + 2^{1-k}$ is essentially packed into the 2^{2-2k} term as the subsequent terms are quickly decaying.
The same reasoning (very quick decrease) applies to next terms of the development of $e_{\min}$, $\dfrac{4^k}{(2^k - 2)^2} \times 2^{-p}$ and its successors.
It turns out that we can consider $1 + 2^{1-k}$ as a very close, and yet smaller, approximation of $e_{\min}$.

What happens if we take it for e for a rounding test?

Let us check with the following example where:

- $y_h \;=\; 2 - 2^{-p+1}$;
- $y_\ell \;=\; 2^{-p} - 2^{-p-k+1}$;
- $y \;=\; 2 - 2^{-p} + 2^{-3p}$;
- $e \;=\; 1 + 2^{-k+1}$, as suggested above.

We have:

- $|y_h + y_\ell - y| < \epsilon \times |y|$;
- $y_h = \mathrm{RN}(y_h + y_\ell)$;
- $y_h = \mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e))$;

and yet $y_h \neq \mathrm{RN}(y)$. This demonstrates the value of e given by Theorem 2.1, if not optimal, is very close to it.

This result is generic (for it is for arbitrary values of p, though limitations are imposed on k). It can also be interesting to scrutinize what happens in particular cases.

2.5.3.2 For some particular values of p, even more can be said

Let us consider the following example:

- the floating-point format is IEEE binary64 (AKA "double precision") with $p = 53$;
- $y = \dfrac{1461983273612937874357096965722}{776934764230052409376713600323}$, with $\mathrm{RN}(y) = \dfrac{8474569075036949}{4503599627370496}$;
- $\epsilon = 2^{-80}$ (not an unrealistic value in the context of function approximation);
- $y_h = \dfrac{8474569075036948}{4503599627370496}$;
- $y_\ell = \dfrac{9007199188662643}{81129638414606681695789005144064}$.

Notice that $y_h \neq \mathrm{RN}(y)$.
It is easy (if tedious) to check that

$$|(y_h + y_\ell) - y| < \epsilon \times |y|,$$

and that $|(y_h + y_\ell) - y|$ is very close to the allowed maximum since

$$\frac{|(y_h + y_\ell) - y|}{\epsilon \times |y|} \approx 1 - 2.54811 \times 10^{-37}.$$

Let us call e^* the minimum given by Theorem 2.1

$$e^* = \frac{(1 + 2^{-p})}{1 - \epsilon - 2^{p+1}\epsilon} = \frac{120892581961462930892390 4}{120892580160023066522419 1}.$$

Again, it is easy to check that, correctly,

$$y_h \neq \mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e^*))$$

If we take for e a slightly smaller value than e^*

$$e \;=\; \frac{4503599649443365}{4503599627370496}$$

with

$$\frac{e}{e^*} \;=\; 0.9999999900000000987640\ldots$$

then we have

$$y_h \;=\; \mathrm{RN}(y_h \,+\, \mathrm{RN}(y_\ell \,\times\, e^*))$$

while $y_h \neq \mathrm{RN}(y)$: an outrageous false positive!

From the two previous examples, we can conclude that Theorem 2.1 gives rather sharp approximations of the optimal bound.

2.5.4 A precision-p computable bound for e

As stated before, Theorem 2.1 assumes some high precision (higher than p) floating-point system is available to correctly compute e. We said this is not a real issue, since to approximate the function with a sufficient accuracy, one must use some high-precision tool that could also be used to compute e. But one may wonder if it is possible to give a bound, let us call it e_{simple} because it does not rely on fancy high-precision arithmetic, that could be computed using solely the target arithmetic system and yet be an accurate one.

2.5.4.1 Assumptions and general form

The quick answer is that it can be done, under not too stringent assumptions. Possible use cases are situations where the high-precision arithmetic is not available. This often happens to the users of a floating-point mathematical library where the high-precision primitives are deeply buried into it and are not directly accessible to them for their own usage. Or if one wants to quickly compute e on the fly for values different of ϵ.

Our principal assumption is that $1 - 2^{p+1}\epsilon$ is exactly representable in precision-p floating-point arithmetic. This can be translated into more specific terms:

- ϵ must be a power (or the sum of a small number of powers) of 2;
- the smallest power of 2 involved in the decomposition of ϵ must be larger than 2^{-2p}.

In the most simple situation, as it often happens in real life, ϵ is a power of 2 larger than 2^{-2p}.

We will give first a general form a p-precision computable bound for e

$$e_{simple} \;=\; \frac{1 \,+\, 2^{-p+1}}{1 \,-\, 2^{p+1}\epsilon}$$

and discuss in more details our assumptions.

Notice that, in general, e_{simple} is not a floating-point number. It must be rounded. This issue will be dealt with below.

One can easily see the reason for our main assumption. If $1 - 2^{p+1}\epsilon$ is not exactly representable, the computation of e goes out of control and there is not much that can mathematically be said about it.

The requirement that $1 - 2^{p+1}\epsilon$ has an exact representation has a cascade effect on the elements of the expression: both $2^{p+1}\epsilon$ and ϵ have to be exactly representable.

If we follow the chain the other way around, we start with the fact ϵ must be a sum of powers of 2 to be exactly representable in p bits. If this is the case $2^{p+1}\epsilon$ will also be since this computation is merely a change of exponent of the corresponding floating-point number (no overflow can happen for any sensible value of ϵ). For $1 - 2^{p+1}\epsilon$ to be exactly representable, it takes the smallest power of 2 in the binary representation of the product of ϵ and 2^{p+1} to be larger than or equal to ulp(1) for the $\mathbb{F}_{2,p,emin,emax}$ system.

Let us check with the following examples.

2.5.4.2 Examples on how our assumptions matter

Example 1

Let us assume the following values:

- The floating-point number system is IEEE 754 **binary64** ($\mathbb{F}_{2,53,-1022,1023}$);
- $\epsilon \;=\; 2^{-100} + 2^{-101}$.

In this case ϵ is exactly representable in $\mathbb{F}$.

$$\epsilon \;=\; 1.10\ldots0 \times 2^{-100}.$$

We then have

$$2^{53+1} \times \epsilon \;=\; 2^{-46} + 2^{-47}.$$

We can therefore exactly compute $1 - 2^{53+1} \times \epsilon$ since $2^{-47} \geqslant 2^{-52} = \text{ulp}(1)$.

Example 2

Let us suppose now that, with the same floating-point system, $\epsilon = 2^{-105} + 2^{-106} + 2^{-107}$.

We then have

$$2^{53+1} \times \epsilon = 2^{-51} + 2^{-52} + 2^{-53}.$$

Now, we can't exactly compute $1 - 2^{53+1} \times \epsilon$ since $2^{-53} \leq 2^{-52} = \text{ulp}(1)$.

2.5.5 More precise properties

From the general form given above, we will consider two more specific forms of e_{simple}:

$$e^{up}_{simple} = \text{RU}\left(\frac{1 + 2^{-p+1}}{1 - 2^{p+1}\epsilon}\right), \text{ and}$$

$$e^{nearest}_{simple} = \text{RN}\left(\frac{1 + 2^{-p+1}}{1 - 2^{p+1}\epsilon}\right).$$

We will give here two properties about them. In the sequel the optimal (high-precision computed) value of e will be called e^*, which is not a floating-point number.

Lets us start with a rather general property of e^{up}_{simple}. For technical reasons that will become obvious in the proof, we need to (slightly) further restrict the upper bound for ϵ to 2^{-p-3}.

Property 2.8. *If $\epsilon \leqslant 2^{-p-3}$ and $1 - 2^{p+1}\epsilon$ is exactly representable (see above for a more precise statement) then $e^{up}_{simple} \geqslant e^*$ and $\frac{e^{up}_{simple}}{e^*} < 1 + 4 \times 2^{-p} = 1 + 2\,\text{ulp}(1)$.*

More informally stated, under Property 2.8 hypothesis, e^{up}_{simple} is a safe (no false positives) and good (very close to) approximation of e^*.

Proof. From the definition of RU () and that of ulp(1) we obviously have

$$\frac{1 + 2^{-p+1}}{1 - 2^{p+1}\epsilon} \leqslant \text{RU}\left(\frac{1 + 2^{-p+1}}{1 - 2^{p+1}\epsilon}\right) \leqslant (1 + 2^{-p+1}) \times \frac{1 + 2^{-p+1}}{1 - 2^{p+1}\epsilon} = \frac{(1 + 2^{-p+1})\,(1 + 2^{-p+1})}{1 - 2^{p+1}\epsilon}$$

Let us define ρ as $\frac{e^{up}_{simple}}{e^*}$. It is clear that

$$\frac{\frac{1 + 2^{-p+1}}{1 - 2^{p+1}\epsilon}}{\frac{1 + 2^{-p}}{1 - \epsilon - 2^{p+1}\epsilon}} \leqslant \rho \leqslant \frac{\frac{(1 + 2^{-p+1}) \times (1 + 2^{-p+1})}{1 - 2^{p+1}\epsilon}}{\frac{1 + 2^{-p}}{1 - \epsilon - 2^{p+1}\epsilon}}$$

To make our expression more legible, let us define now ω as $1 - 2^{p+1}\epsilon$. With this notation

$$1 - \epsilon - 2^{p+1}\epsilon = \omega - \epsilon = \omega\left(1 - \frac{\epsilon}{\omega}\right).$$

With this new notation and the above remark, our previous expression becomes now

$$\frac{\dfrac{1 + 2^{-p+1}}{\omega}}{\dfrac{1 + 2^{-p}}{\omega\left(1 - \dfrac{\epsilon}{\omega}\right)}} \leqslant \rho \leqslant \frac{\dfrac{(1 + 2^{-p+1}) \times (1 + 2^{-p+1})}{1 - 2^{p+1}\epsilon}}{\dfrac{1 + 2^{-p}}{\omega\left(1 - \dfrac{\epsilon}{\omega}\right)}}.$$

From this expression, simplifying by ω whenever possible,

$$\frac{1 + 2^{-p+1}}{1 + 2^{-p}}\left(1 - \frac{\epsilon}{\omega}\right) \leqslant \rho \leqslant \frac{(1 + 2^{-p+1})^2}{1 + 2^{-p}}\left(1 - \frac{\epsilon}{\omega}\right).$$

We are going now to prove the upper bound of Property 2.8. Notice that

$$1 - \frac{\epsilon}{\omega} = \frac{\epsilon}{1 - 2^{p+1}\epsilon} = \frac{1}{\dfrac{1}{\epsilon} - 2^{p+1}}.$$

As $\epsilon < \dfrac{1}{1 + 2^{p+1}}$ we also have

$$0 < \frac{1}{\dfrac{1}{\epsilon} - 2^{p+1}} < \frac{1}{1 + 2^{p+1} - 2^{p+1}} = 1.$$

Hence

$$1 - \frac{\epsilon}{\omega} < 1.$$

Then

$$\frac{(1 + 2^{-p+1})^2}{1 + 2^{-p}}\left(1 - \frac{\epsilon}{\omega}\right) \leq \frac{(1 + 2^{-p+1})^2}{1 + 2^{-p}}.$$

Notice that, on the one hand,

$$(1 + 4 \times 2^{-p})\,(1 + 2^{-p}) = 1 + 5 \times 2^{-p} + 4 \times 2^{-2p} = 1 + 5 \times 2^{-p} + 2^{-2p+2},$$

and that, on the other hand,

$$(1 + 2^{-p+1})^2 = 1 + 4 \times 2^{-p} + 2^{-2p+2}.$$

As $1 + 2^{-p} > 1$, whatever p is, we have

$$\frac{(1 + 2^{-p+1})^2}{1 + 2^{-p}} < 1 + 4 \times 2^{-p},$$

and consequently

$$\frac{1 + 2^{-p+1}}{1 + 2^{-p}} \left(1 - \frac{\epsilon}{\omega}\right) \leqslant \rho < 1 + 4 \times 2^{-p}.$$

Now, let us make sure that $\rho = \frac{e^{up}_{simple}}{e^*} \geqslant 1$. For all $p \geqslant 1$

$$\frac{1 + 2^{-p+1}}{1 + 2^{-p}} = \frac{1 + 2^{-p} + 2^{-p}}{1 + 2^{-p}} = \frac{(1 + 2^{-p}) + 2^{-p}}{1 + 2^{-p}} = 1 + \frac{2^{-p}}{1 + 2^{-p}}.$$

Let us concentrate on the last term of the final sum on the previous line. We can see that

$$\frac{2^{-p} \times (1 - 2^{-p})}{(1 + 2^{-p}) \times (1 - 2^{-p})} = \frac{2^{-p} - 2^{-2p}}{1 - 2^{-2p}}.$$

Since $1 - 2^{-2p} < 1$

$$\frac{2^{-p} - 2^{-2p}}{1 - 2^{-2p}} \geqslant 2^{-p} - 2^{-2p}$$

and, putting all together,

$$\frac{1 + 2^{-p+1}}{1 + 2^{-p}} \geqslant 1 + 2^{-p} - 2^{-2p}.$$

From our hypothesis $\epsilon \leqslant 2^{p-3}$. This implies that, as $\omega = 1 - 2^{p+1}\epsilon$, $\omega \geqslant \frac{3}{4}$ and, consequently, $1 - \frac{\epsilon}{\omega} \leqslant 1 - 3 \times 2^{-p-1}$.

Combining the first lower bound and the second upper bound

$$(1 + 2^{-p} - 2^{-2p}) \times (1 - 3 \times 2^{-p-1}) = 1 + \frac{5}{6}2^{-p} - \frac{7}{6}2^{-2p} + \frac{1}{6}2^{-3p}$$

which is larger than 1 as soon as $p \geqslant 1$.

Ultimately

$$1 \leqslant \rho < 1 + 4 \times 2^{-p}.$$

□

Our second property is a bit different and possibly has more practical applications. We will first limit ourselves to particular values of p, those that are very often used in actual computations: the binary interchange formats of the IEEE Standard for Floating-Point Arithmetic and the INTEL "double-extended precision" format. Secondly our property will relate to $e^{nearest}_{simple}$ which is easier and quicker to compute since it does involve any time-consuming rounding-mode change (round-to-

nearest being usually the default mode).

Property 2.9. *If* $p \in \{11, 24, 53, 64, 113\}$, *and* $\epsilon = \gamma \times 2^{-p-k}$, *with* $\gamma = 1, \frac{3}{4}, \frac{5}{8}$, *or* $\frac{7}{8}$, *and* $3 \leqslant k \leqslant p+1$, *then*

$$e^* \leqslant e^{nearest}_{simple} < e^* + \mathrm{ulp}(e^*).$$

We remind here that generally e^* is not a floating-point number. The proof of the property can be done by an enumeration of all cases and computing the numbers. Within the hypothesis of Property 2.9 it is not possible to compute a better Ziv's test constant by using Theorem 2.1. The "simple" yields a result as good as that of the "regular" one.

For the sake of completeness we will give a third and last property of the simple bound. We will also give some insights into its apparent limitations when accurate (i.e. very small values of ϵ) approximations are computed.

Property 2.10. *If* $\epsilon \leqslant 2^{-2p-2}$, *then*

$$e^* \leqslant e^{nearest}_{simple}.$$

Proof. If $\epsilon \leqslant 2^{-2p-2}$, then

$$1 = \mathrm{RN}(1) \geqslant \mathrm{RN}(1 - 2^{-p+1}\epsilon) \geqslant \mathrm{RN}(1 - 2^{-p+1} \times 2^{-2p-2}) = \mathrm{RN}(1 - 2^{-p-1}) = 1.$$

Hence, for any $\epsilon \leqslant 2^{-2p-2}$, $\mathrm{RN}\left(\dfrac{1 + 2^{-p+1}}{1 - 2^{p+1}\epsilon}\right) = 1 + 2^{-p+1}$.

How does it compare with $e = \dfrac{1 + 2^{-p}}{1 - \epsilon - 2^{p+1}\epsilon}$?

Since $1 \geqslant 1 - \epsilon - 2^{p+1}\epsilon$ whatever the value of ϵ matching our hypothesis and whatever the (high) precision used to compute the last expression, we are done if we can prove that

$$(1 + 2^{-p+1})(1 - \epsilon - 2^{p+1}\epsilon) \geqslant 1 + 2^{-p}$$

Assigning ϵ to its maximum value 2^{-2p-2}, elementary computations give:

$$\begin{aligned}(1 + 2^{-p+1})(1 - \epsilon - 2^{p+1}\epsilon) &= (1 + 2^{-p+1})(1 - 2^{-2p-2} - 2^{p+1} \times 2^{-2p-2}) \\ &= 1 + 2^{-p+1} - 2^{-3p-1} - 2^{-p-1} - 5 \times 2^{-2p-2} \\ &= 1 + 2^{-p} + 2^{-p} - 2^{-3p-1} - 2^{-p-1} - 5 \times 2^{-2p-2}.\end{aligned}$$

At this point, the verification boils down to checking that:

$$2^{-p} - 2^{-3p-1} - 2^{-p-1} - 5 \times 2^{-2p-2} \geqslant 0,$$

or

$$2^{-p-1} + 2^{-p-1} - 2^{-3p-1} - 2^{-p-1} - 5 \times 2^{-2p-2} \geqslant 0.$$

Eventually we can easily check that:

$$2^{-p-1} > 2^{-3p-1} + 8 \times 2^{-2p-2} = 2^{-3p-1} + 2^{-2p+1}$$

□

But this is not the end of the story about $e_{simple}^{nearest}$ and high accuracy approximation. It must be noted it is stuck to the constant value $1 + 2^{-p+1}$ as soon as, when ϵ decreases, it reaches 2^{-2p-1}. One could think that this is not the case with e^* which may "freely" continue to decrease and consider that, $e_{simple}^{nearest}$, though safe (no false positives) becomes less and less efficient as a Ziv's rounding constant since its divergence from e^* leads to an increasing number of false negatives as ϵ plunges under 2^{-2p-2}.

This is not the case since $\mathrm{RU}(e^*)$ itself, the actually used value, is lower bounded by 1 and is also a floating-point number. The value immediately above 1 is $1 + \mathrm{ulp}(1)$, which also happens to be $1 + 2^{-p+1}$, the limit value of $e_{simple}^{nearest}$. When one gets approximation accuracies close to 2^{-2p}, the final rounding in the computation of ϵ^* sweeps out the refinement brought in by the previous high precision evaluations.

At this point, one could rather question the relevance of Ziv's rounding test in these conditions. In fact, this questioning relates to the very idea of computing very sharp approximations ($\epsilon < 2^{-2p-2}$) with (only) a "main term" and a single "correcting term" that together provide, at most (if no "gap" is allowed, see Figure 2-1(3)), $2p + 1$ bits of precision.

If we elaborate a bit on one of our initial assumptions about y_h and y_ℓ (namely $y_h = \mathrm{RN}(y_h + y_\ell)$) we have seen that it has twofold implication:

- $|y_\ell| < \frac{1}{2}\mathrm{ulp}(y_h)$, if y_h and y_ℓ have the same signs or y_h is not a power of 2;
- $|y_\ell| < \frac{1}{4}\mathrm{ulp}(y_h)$, if y_h is a power of 2 and y_h and y_ℓ have different signs.

We will not care further about the $\frac{1}{2}$ and $\frac{1}{4}$ difference here as we will think in terms of orders of magnitude and only consider the first case. We will not deal with the subnormal issues either.

With these restrictions the $y_h + y_\ell$ sum can then be seen as some kind of $(2p+1)$-precision number that we can represent, when y_h and y_ℓ have the same sign, in most situations (for "large" values of y_h) as:

$$y_h + y_\ell = \underbrace{1.hh\cdots hhh01\ell\ell\cdots\ell\ell\ell}_{2p+1 \text{ bits}} \times 2^{e_{y_h}}, \text{ where } e_{y_h} \text{ is the exponent of } y_h,$$

assuming here that $y_\ell \geqslant 1/4\text{ulp}(y_h)$.

Here the $hh\cdots hhh$ and the $\ell\ell\cdots\ell\ell\ell$ strings represent, respectively, the bits of the fraction of y_h and those of the fraction of y_ℓ. We can also say that $2^{e_{y_h}}$ is the order of magnitude of y_h and that of y as well.

Had y_h and y_ℓ different signs, the values of the bits would differ, the central bit would possibly not be 0, but the structure should remain the same.

Let us consider now the "ulp()"[12]:

$$\text{ulp}(y_h + y_\ell) = 2^{(-2p-1)+1} \times 2^{e_{y_h}} = 2^{-2p} \times 2^{e_{y_h}}.$$

With such a value for $\text{ulp}(y_h + y_\ell)$, it does not make much sense to try and compute approximations of y with an accuracy sharper than 2^{-2p-1}. It comes to no surprise that this limitation extends its shadow to Ziv's rounding test itself.

2.6 Additional results

2.6.1 If test fails, lose no hope!

As we will see soon, when the Ziv's test fails, it does not imply that $\text{RN}(y)$ can go wild, neither that $|y_h - y|$ can take any arbitrary value and, finally, neither that the computation of y_h and y_ℓ was a complete waste of time.

What happens when the Ziv's test fails?

[12] Quotes are necessary here since one has to remember that the ulp() function is fully specified only in the context of a floating-point system. Here, we deal with the blurry $y_h + y_\ell$ unevaluated sum object.

Our hypothesis is that

$$y_h + y_\ell = y \times (1 + \alpha) \text{ with } |\alpha| \leqslant \epsilon. \tag{2.21}$$

And we have

$$y_h \neq \mathrm{RN}(y_h + \mathrm{RN}(e \times y_\ell)).$$

In other words, we have

$$\mathrm{RN}(y_h + \mathrm{RN}(e \times y_\ell)) = y_c \text{ with } y_c \neq y_h.$$

In fact, under reasonable assumptions, the values of y_c and $\mathrm{RN}(y)$ are quite constrained as stated in the following property:

Theorem 2.2. *If*

- $p \geqslant 2$ *(a very mild condition);*
- $\epsilon < \dfrac{1}{2^{p+3} + 9}$ *(this weird value has technical reasons that will explain by themselves in the proof);*
- $e = \mathrm{RN}\left(\dfrac{1 + 2^{-p}}{1 - \epsilon - 2^{p+1}\epsilon}\right)$ *(the smallest value of e that is guaranteed for Ziv's test to be correct)*

then, when the test fails (remember: $y_h \neq y_c = \mathrm{RN}(y_h + \mathrm{RN}(e \times y_\ell))$):

1. *y_h and y_c are two consecutive floating-point numbers;*
2. *$y_h \leqslant y \leqslant y_c$ or $y_c \leqslant y \leqslant y_h$;*
3. *$y_h = \mathrm{RN}(y)$ or $y_c = \mathrm{RN}(y)$.*

Notice that **(3)** is only a self-evident consequence of **(2)**, when **(1)** holds.

Proof. We will first check that e never gets larger than 2. Let us consider the following family of functions $E_p(\epsilon) = \dfrac{1 + 2^{-p}}{1 - \epsilon - 2^{p+1}\epsilon}$ of ϵ, each defined in domain $\left[0, \dfrac{1}{2^{p+3} + 9}\right]$ with p in the $[2, +\infty) \bigcap \mathbb{N}$ set.

We can easily see that these are increasing functions in ϵ.

Let us consider now the $E_m(p)$ function, which computes the maximum of each of the $E_p(\epsilon)$ family members when they reach their respective $\frac{1}{2^{p+3}+9}$ value:

$$E_m(p) \;=\; \frac{1+2^{-p}}{1-\dfrac{1+2^{p+1}}{2^{p+3}+9}}$$

One can check that the value of this function is always smaller than 2.

Among the members of the family, $E_2(\epsilon)$ reaches the largest value for any given ϵ.
The variable ϵ reaches its upper bound $\frac{1}{2^{p+2}+8} = \frac{1}{2^{2+2}+8}$. We then have

$$E_2\left(\frac{1}{24}\right) \;=\; 2.$$

As soon as $p \geqslant 2$ and $\epsilon < \frac{1}{2^{p+2}+8}$ then $e \leqslant 2$. As we will see, the stronger condition on ϵ in our hypothesis will be needed later on. Yet the property $e \leqslant 2$ will still stand.

We can assume, now on and without loss of generality, that y_h and y are both positive numbers.

Since, under our assumptions, $e \leqslant 2$, we can deduce that

$$\begin{aligned} &\text{if} \quad |y_\ell| \leqslant \frac{1}{8}\mathrm{ulp}(y_h) \\ &\text{then } |\mathrm{RN}(e \times y_\ell)| \leqslant \frac{1}{4}\mathrm{ulp}(y_h). \end{aligned}$$

If so, then $y_c = \mathrm{RN}(y_h + \mathrm{RN}(e \times y_\ell)) = y_h$ because either:

- y_h is not a power of 2 and this is straightforward;
- y_h is a power of 2 and this is a consequence of the "round to nearest, ties-to-even" rule (a power of 2 always has an even integral significand, almost per definition).

In this situation, $y_c \neq y_h$ implies that $|y_\ell| > \frac{1}{8}\mathrm{ulp}(y_h) > 2^{-p-3}y_h$. And, consequently

$$\left|\frac{y_h}{y_\ell}\right| \;<\; 2^{p+3}. \tag{2.22}$$

From our hypothesis (2.21), we can derive

$$y - y_h \;=\; y_\ell\left(1 - \alpha\frac{y}{y_\ell}\right) \tag{2.23}$$

with $|\alpha| \leqslant \epsilon$.

We also have, as our fundamental hypothesis, $\mathrm{RN}(y_h + y_\ell) = y_h$ which implies

$$\left|\frac{y_h}{y_\ell}\right| \geqslant 2^p. \tag{2.24}$$

From (2.21) we have

$$y = \frac{y_h + y_\ell}{1+\alpha},$$

from which, by using (2.24), we can deduce (remember we still assume that both y_h and y are positive numbers and $1 + \alpha \geqslant 1 - \epsilon$)

$$\left|\frac{y}{y_\ell}\right| \leqslant \frac{y_h + |y_\ell|}{|y_\ell|(1+\alpha)} \leqslant \frac{y_h + |y_\ell|}{|y_\ell|(1-\epsilon)} \leqslant \frac{y_h(1+2^{-p})}{|y_\ell|(1-\epsilon)} = \frac{y_h}{|y_\ell|} \times \frac{1+2^{-p}}{1-\epsilon}.$$

Therefore, using (2.22) we can state

$$\left|\frac{y}{y_\ell}\right| < 2^{p+3}\frac{1+2^{-p}}{1-\epsilon},$$

finally

$$|\alpha| \times \left|\frac{y}{y_\ell}\right| < 2^{p+3} \times (1+2^{-p}) \times \frac{\alpha}{1-\epsilon} \leqslant 2^{p+3} \times (1+2^{-p}) \times \frac{\epsilon}{1-\epsilon}. \tag{2.25}$$

If we express ϵ as

$$\epsilon = \frac{1}{2^{p+3}+k}, \text{ with } k \in \mathbb{N},$$

then

$$\frac{\epsilon}{1-\epsilon} = \frac{1}{2^{p+3}+k-1}.$$

We can rewrite the right hand side of (2.25) as

$$2^{p+3} \times (1+2^{-p}) \times \frac{1}{2^{p+3}+k-1} = \frac{2^{p+3}+8}{2^{p+3}+k-1}. \tag{2.26}$$

We have (2.26) < 1 as soon as $k > 9$. Consequently, under the same conditions and if $y_c \neq y_h$, we have $|\alpha| \times \left|\frac{y}{y_\ell}\right| < 1$, $1 - |\alpha| \times \left|\frac{y}{y_\ell}\right| > 0$. Combining with (2.23) we can deduce two things:

- $y - y_h$ and y_ℓ have the same sign;
- $y_h^- \leqslant y$ and $y \leqslant y_h^+$.

Therefore:

- if $y_\ell > 0$ then $y > y_h$ and (straightforwardly), $y_c > y_h$;

- if $y_\ell < 0$ then $y < y_h$ and (straightforwardly), $y_c < y_h$.

Eventually, when the test fails, y is between the two consecutive floating-point numbers, y_c and y_h. From the definition of the RN() function, either:

- $\mathrm{RN}(y) = y_c$;
- $\mathrm{RN}(y) = y_h$.

□

Theorem 2.2 is particularly useful for the implementation of an (at least) locally monotonic elementary function f at point X. If one can approximate $f^{-1}\left(\frac{y_h + y_c}{2}\right)$ with a good enough accuracy[13], it is possible to determine if $\mathrm{RN}(f(X)) = y_h$ or $\mathrm{RN}(f(X)) = y_c$. See Figure 2-12 for more details.

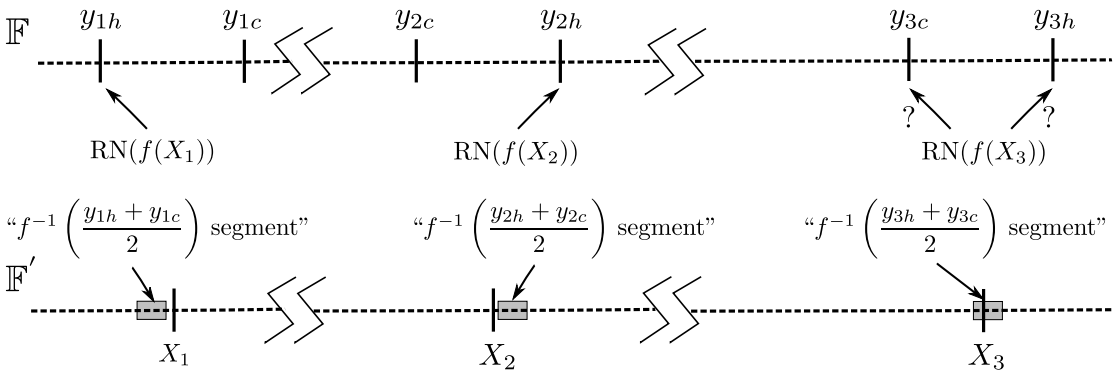

Figure 2-12: We have here three examples of a situation where Ziv's test has failed and we were able to compute $f^{-1}\left(\frac{y_h + y_c}{2}\right)$. We assume that f is locally increasing. In example 1, $f^{-1}\left(\frac{y_{1h} + y_{1c}}{2}\right)$ computes a value smaller than X_1 and error on f^{-1} rises no issue here. We can safely conclude that $\mathrm{RN}(f(X_1)) = y_{1_h}$. In example 2, the ordering of y_{2c} and y_{2h} is different (there is no reason one should prevail over the other). This time $f^{-1}\left(\frac{y_{2h} + y_{2c}}{2}\right)$ computes a value larger than X_2 and still, the error on f^{-1} rises no issue. We can safely conclude that $\mathrm{RN}(f(X_2)) = y_{2h}$. In the third example the accuracy of the approximation of f^{-1} is not sufficient to make a safe choice between y_{3h} or y_{3c}. Worse, $f(X_3)$ could be a midpoint of $\mathbb{F}$!

[13] if the relative error ϵ is such that $x \notin \left(f^{-1}\left(\frac{y_h + y_c}{2}\right)(1-\epsilon),\ f^{-1}\left(\frac{y_h + y_c}{2}\right)(1+\epsilon)\right)$, when $f^{-1}\left(\frac{y_h + y_c}{2}\right) > 0$.

2.6.2 When an FMA is available

The FMA instruction properties are presented in §1.5.4, p. 48. If an FMA instruction is available Ziv's test can be modified with two beneficial consequences:

- the value of e can possibly be smaller (less false negatives);
- the test still works when $\frac{1}{4}y_h$ is a subnormal.

We will use here the property that the FMA instruction evaluates expressions of the form $\mathrm{RN}(a + b \times c)$, that is with a single rounding.

Fundamentally, we replace the $y_h \stackrel{?}{=} \mathrm{RN}(y_h + \mathrm{RN}(e \times y_\ell))$ test by the $y_h \stackrel{?}{=} \mathrm{RN}(y_h + e \times y_\ell)$ test. The analysis of this new test will follow the same lines as that of the original one.

When y_h is not a power of 2 and y_h is in the normal range, we can both get rid of the $(1 + 2^{-p})$ term and drop the $\frac{1}{2}\mathrm{ulp}(y_h)$ in the normal range condition from Property 2.3.

Property 2.11. *Assume that y_h is not a power of 2. Let e be a nonnegative floating-point number. If*

$$y_h = \mathrm{RN}(y_h + y_\ell \times e),$$

then

$$|y_\ell| \leqslant \frac{\mathrm{ulp}(y_h)}{2e}.$$

We then have an analogous property to Property 2.4.

Property 2.12. *Assume y_h is not a power of 2. If*

$$e \geq \frac{1}{1 - \epsilon - 2^{p+1}\epsilon}$$

then

$$y_h = \mathrm{RN}(y_h + y_\ell \times e) \Rightarrow y_h = \mathrm{RN}(y).$$

Similarly, when y_h is a power of 2, Property 2.6 is modified as follows.

Property 2.13. *Assume that y_h is a power of 2, $y \leqslant y_h$, and $y_\ell \leqslant 0$. If*

$$y_h = \mathrm{RN}(y_h + y_\ell \times e), \tag{2.27}$$

then

$$|y_\ell| \leqslant \frac{2^{-p-1} \times y_h}{e}. \tag{2.28}$$

Additionally, Property 2.7 is also altered into the following statement.

Property 2.14. *Assume that y_h is a power of 2, and that $\epsilon < \frac{1}{1+2^{p+1}}$. If*

$$e \geq \frac{1-2\epsilon}{1-\epsilon-2^{p+1}\epsilon} \tag{2.29}$$

then $y_h = \mathrm{RN}(y_h + y_\ell \times e) \Rightarrow y_h = \mathrm{RN}(y)$.

From all these properties we can deduce the following theorem.

Theorem 2.3. *Assume that:*

- $\epsilon < \frac{1}{1+2^{p+1}}$;
- $|(y_h + y_\ell) - y| < \epsilon \times |y|$;
- $y_h = \mathrm{RN}(y_h + y_\ell)$;

if

$$e \geqslant \frac{1}{1-\epsilon-2^{p+1}\epsilon}$$

then

$$y_h = \mathrm{RN}(y_h + y_\ell \times e) \Rightarrow y_h = \mathrm{RN}(y).$$

Another difference is that for values of e very close or equal to the FMA bound, e can eventually (when $\epsilon < 2^{-2p}$) reach 1. At such an approximation accuracy, if $|(y_h + y_\ell) - y| \leqslant |y \times \epsilon|$ we are in one of the two following situations:

- $|y_\ell|$ is much smaller than $\frac{1}{2}\mathrm{ulp}(y_h)$ (or $\frac{1}{4}\mathrm{ulp}(y_h)$) and then there is not the slightest chance that y, even if placed at the "tip" of the $(y_h + y_\ell) \times \left(1 + \frac{\epsilon}{1-\epsilon}\right)$ segment, can reach or be beyond a midpoint; necessarily $\mathrm{RN}(y) = y_h$;
- $|y_\ell| \approx \frac{1}{2}\mathrm{ulp}(y_h)$ (or $\frac{1}{4}\mathrm{ulp}(y_h)$), but nevertheless smaller; the (y_h, y_ℓ) pair can be viewed as a kind of $2p+1$ "floating-point number"; we must consider ourselves very lucky that $y_h + y_\ell$ is so close to y with such a limited precision; any other value for y_ℓ, say y'_ℓ, even if only differing from y_ℓ by $\mathrm{ulp}(y_\ell)$, will violate the $|(y_h + y'_\ell) - y| \leqslant |y \times \epsilon|$ fundamental accuracy assumption.

For an approximation accuracy $\epsilon < 2^{-2p}$, the fact that e reaches 1 could be be interpreted as "we have found a right (y_h, y_ℓ), rounding test is inutile since we are sure that $\mathrm{RN}(y) = y_h$". But it is impossible to consistently approximate a function at such an accuracy with only two floating-point numbers. If one targets for a higher accuracy, she must resort to other techniques (e.g. triple-double numbers – see [21] for an implementation – if she insists on using groups of floating-point numbers). Ziv's rounding test is not relevant in these contexts.

2.6.3 What if $\frac{1}{4}\mathrm{ulp}(y_h)$ is subnormal and no FMA is available?

We know that the relative error committed in rounding to subnormal numbers can be very large. In the worst case, if we call $sub_{min} = 2^{e_{\min}-p+1}$ the smallest subnormal number in $\mathbb{F}$, and if $\mathrm{RN}(x) = sub_{min}$ then $|x| \in ((1-\frac{1}{2}) \times sub_{min}, (1+\frac{1}{2}) \times sub_{min})$. The relative error is almost as large as $\frac{1}{2}$!

Hence, if we revisit the proof of Property 2.3, all we can deduce from

$$|\mathrm{RN}(y_\ell \times e)| \leqslant \frac{1}{2}\mathrm{ulp}(y_h)$$

is

$$|\mathrm{RN}(y_\ell \times e)| \leqslant \frac{3}{4}\mathrm{ulp}(y_h).$$

Following the reasoning that led us to Theorem 2.1, we end up with the following bound

$$e \geqslant \frac{3/2}{1-\epsilon-2^{p+1}\epsilon} \tag{2.30}$$

which is quite discouraging. To add insult to injury this rather bad bound is not the result of poor calculus: this would have, at least, left hope for improvement. It is indeed possible to find examples where a value slightly lower than our bound leads to false positives.

Consider the following extreme (but nevertheless possible) situation:

- $y_h = 2^{e_{\min}+1}$;
- $y_\ell = 2^{e_{\min}-p+1}$ (which is the smallest subnormal);
- $y = 2^{e_{\min}+1} + 2^{e_{\min}-p+1} + 2^{e_{\min}-2p} = y_h + y_\ell + 2^{e_{\min}-2p}$;
- $\epsilon = 2^{-2p-1}$.

We have, while y_ℓ is a midpoint in the y_h binade but the integral significand of y_h is even,

$$y_h = \mathrm{RN}(y_h + y_\ell)$$

and

$$|(y_h + y_\ell) - y| \leqslant |y \times \epsilon|.$$

If we compute the bound given by (2.30), we get

$$e \geqslant \frac{3/2}{1 - \epsilon - 2^{p+1}\epsilon} \approx \frac{3}{2}(1 + 2^{-p}).$$

Let us pick a slightly smaller value for e:

$$e = \frac{3}{2} - 2^{-p+1} = \underbrace{1.011\ldots111}_{p \text{ bits}}.$$

Multiplying y_ℓ by e and rounding gives y_ℓ again:

$$\mathrm{RN}(y_\ell \times e) = \mathrm{RN}(2^{e_{\min}-p+1} \times \underbrace{1.011\ldots111}_{p \text{ bits}}) = 2^{e_{\min}-p+1}.$$

We then still have

$$\mathrm{RN}(y_h + \mathrm{RN}(y_\ell \times e)) = y_h.$$

But yet

$$\mathrm{RN}(y) = 2^{e_{\min}+1} + 2^{e_{\min}-p+1} \neq y_h.$$

But what would have happened here had an FMA instruction been available (and used)?

In this case, the first multiplication issues a number larger than y_ℓ, hence also larger than $\frac{1}{2}\mathrm{ulp}(y_h)$. The upcoming addition is performed with this number and will output $y_h^+ \neq y_h$: the test will fail calling for a new computation for the (y_h, y_ℓ) pair under more stringent accuracy conditions. But this may also require a change in precision since is it very unlikely, given the value of y, that two floating-point numbers provide enough precision to approximate it with a sufficient accuracy.

The order of magnitude of y is that of y_h. Hence, the order of magnitude of the absolute error, $|y \times \epsilon|$ is $2^{e_{\min}+1} \times 2^{-p-1} = 2^{e_{\min}-p}$, which, in turn, is smaller that $\mathrm{ulp}(\mathit{subnormals}) = 2^{e_{\min}-p+1}$ for this floating-point system.

As for Ziv's test, the FMA version is without controversy the way to go. The increasing availability of this instruction on mainstream devices[14] will probably make the previous discussion history

[14] At least in the realm of numerical and/or high-performance computing but presently also in mobile processors as the recent ARM Cortex series.

in a near future.

But there is still a very large installed base of efficient hardware in use and more important, a huge body of code devoted to numerical computations and approximations. How can all these assets be properly used?

2.6.4 What can be done, for function approximation, if no FMA instruction is available?

If we consider the problem of the correctly-rounded evaluation of the usual transcendental functions, we can often escape from the trap of the subnormal values.

Suppose we want to compute some $f(X)$ for domain values where $f(X) \approx 0$ but still, $f(X) \neq 0$ (and of course, $\mathrm{RN}(f(X)) \neq 0$).

This situation may happen in, at least, two cases:

- we compute $f(X)$ for tiny arguments for which $f(X) \approx 0$;
- we compute $f(X)$ for large arguments (in the sense of far from the subnormals range) that are very close to a root of $f(X)$.

Let us start with an example of the first case. We want to compute, for instance, $\sin(X)$ where X is a very small floating-point number and $X > 0$.

One possible way to evaluate a function for very small arguments is to compute its Taylor series expansion in the vicinity of $X = 0$.

In the case of the sine function this expansion is

$$f(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots + \frac{(-1)^n x^{2n+1}}{(2n+1)!} + O(x^{2n+3}). \tag{2.31}$$

Let $X = m_X \times 2^{e_X}$, with $1 \leqslant m_X < 2$ and let us assume that X is not a power of two (this case can be dealt with separately). If

$$X \leqslant \sqrt{3} \times 2^{\frac{-p}{2}},$$

then

$$m_X \times X^2 < 2 \times X^2 < 2 \times \left(\sqrt{3} \times 2^{\frac{-p}{2}}\right)^2 < 6 \times 2^{-p}.$$

From that, we can deduce

$$X^3 \;=\; {m_X}^3 \times 2^{3e_X} \;<\; 6 \times 2^{e_X} \times 2^{-p},$$

and then

$$\frac{X^3}{6} \;<\; 2^{e_X - p} \;=\; \frac{1}{2}\mathrm{ulp}(X).$$

This is an alternate series whose successive terms have a decreasing absolute value. The error generated by truncating it at some term is upper bounded by the value of the first dropped term. No check is needed for any of the subsequent terms.

$$\mathrm{RN}\left(X \;-\; \frac{X^3}{6}\right) \;=\; X$$

Consequently, for $X \;\leqslant\; \sqrt{3} \times 2^{\frac{-p}{2}}$, we can say that $\mathrm{RN}\left(\sin(X)\right) \;=\; X$.

Similar reasoning can be held for other functions and even better bounds can be found for specific values of p. Table 2.1 shows how small values can be handled for a set of usual functions for $p = 53$.

Another situation is when X is a "large" floating-point number. In practice, $f(X)$ is always far away from the subnormal range, even if, theoretically, X is close to a root of f. For instance, in the case of the trigonometric functions, using a continued-fraction argument presented first by W. Kahan [52] (a Maple® implementation can be found in [78]), one can show that the double-precision/IEEE-binary64 number larger than 1 which is nearest to an integer multiple of $\frac{\pi}{2}$ is

$$6381956970095103 \times 2^{797} \;\approx\; k * \pi/2 + 2^{-60.8879} \text{ with } k \in \mathbb{N}.$$

If we deal with the $\cos(x)$ function, we find out that $\cos(6381956970095103 \times 2^{797}) \;\approx\; -4.69 \times 2^{-19}$. We cannot get any closer to 0 with double-precision arguments larger than one with the cosine function. And this $\frac{1}{4}\mathrm{ulp}(-4.69 \times 2^{-19})$ value is very far from the subnormal range. Hence one could safely use the non-FMA version of Ziv's test for the design of a correctly-rounded approximation of the $\cos(x)$ function.

2.7 Tests and checks

Tests and checks were performed here, not so much as a support or as a substitute for mathematical proofs, but because numerical experiments are instrumental for documenting some interesting behaviors.

This function...	can be replaced by...	when...
$\exp(x) - 1$	x	$\lvert x\rvert < \mathrm{RN}(\sqrt{2}) \times 2^{-53}$
$\exp(x) - 1$, $x \geqslant 0$	x^+	$\mathrm{RN}(\sqrt{2}) \times 2^{-53} \leqslant x < \mathrm{RN}(\sqrt{3}) \times 2^{-52}$
$\ln(1 + x)$	x	$\lvert x\rvert < \mathrm{RN}(\sqrt{2}) \times 2^{-53}$
$\sin(x)$, $\sinh(x)$, $\sinh^{-1}(x)$	x	$\lvert x\rvert \leqslant \mathrm{RN}(3^{\frac{1}{3}}) \times 2^{-26}$
$\arcsin(x)$	x	$\lvert x\rvert < \mathrm{RN}(3^{\frac{1}{3}}) \times 2^{-26}$
$\sin(x)$, $\sinh^{-1}(x)$	$x^- = x - 2^{-78}$	$\mathrm{RN}(3^{\frac{1}{3}}) \times 2^{-26} < x \leqslant 2^{-25}$
$\sinh(x)$	$x^+ = x + 2^{-78}$	$\mathrm{RN}(3^{\frac{1}{3}}) \times 2^{-26} < x < 2^{-25}$
$\arcsin(x)$	x^+	$\mathrm{RN}(3^{\frac{1}{3}}) \times 2^{-26} \leqslant x < 2^{-25}$
$\tan(x)$, $\tanh^{-1}(x)$	x	$\lvert x\rvert < \mathrm{RN}(12^{\frac{1}{3}}) \times 2^{-27}$
$\tanh(x)$, $\arctan(x)$	x	$\lvert x\rvert \leqslant \mathrm{RN}(12^{\frac{1}{3}}) \times 2^{-27}$
$\tan(x)$, $\tanh^{-1}(x)$	x^+	$\mathrm{RN}(12^{\frac{1}{3}}) \times 2^{-27} \leqslant x \leqslant 1.650 \times 2^{-26}$
$\arctan(x)$, $\tanh(x)$	x^-	$\mathrm{RN}(12^{\frac{1}{3}}) \times 2^{-27} < x \leqslant 1.650 \times 2^{-26}$

Table 2.1: Rounding to nearest is assumed (this table is extracted from that, more comprehensive, in [78]). They can be extended to other precisions. They make evaluating function for tiny argument straightforward and allow one to avoid using the non-FMA version of Ziv's test in areas where it is of no value. Number $\mathrm{RN}(3^{\frac{1}{3}}) \times 2^{-26} \approx 1.4422\cdots \times 2^{-26}$, and number $\mathrm{RN}(12^{\frac{1}{3}}) \times 2^{-27} \approx 1.1447 \times 2^{-26}$. If x is a floating-point number, x^- denotes the largest floating-point number strictly less than x while x^+ denotes the smallest floating-point number strictly larger than x. The 2^{-78} skew from x given here for x^- and x^+ only makes sense in the corresponding 2^{-26} binade and for double-precision format.

2.7.1 General remarks

We have limited our tests to IEEE binary64 floating-point numbers ($p = 53$): both y_h and y are of the type "double" of the C language. We (allegedly) tried to approximate an arbitrary precision number y, with relative error bounded by ϵ. We also have limited ourselves to peculiar values of y_ℓ, where it is positive (the negative case is symmetric), and close to $\frac{1}{2}\text{ulp}(y_h)$ since, considering the small values of e that we obtain for reasonable values of ϵ, nothing special happens for values of y_ℓ notably smaller than $\frac{1}{2}\text{ulp}(y_h)$. The value of y is chosen extremely close to the limit imposed by ϵ. When y_ℓ is very close to $\frac{1}{2}\text{ulp}(y_h)$, y is almost guaranteed to be larger than or equal to $y_h + \frac{1}{2}\text{ulp}(y_h)$ unless ϵ is very small. Therefore our experiments can be viewed as mimicking a worst case situation, where y is always cornered close to the upper bound rather than having its value scattered all over the interval $((y_h + y_\ell)\ (1 - \epsilon), (y_h + y_\ell)\ (1 + \epsilon))$.

Let us remind here the semantics associated with our results typology:

- **positive:** $(y_h = \text{RN}(y_h + \text{RN}(y_\ell \times e))$ and $y_h = \text{RN}(y)$; in such a case, we are done: Ziv's rounding test correctly indicates that $y_h = \text{RN}(y)$;
- **false positive:** $(y_h = \text{RN}(y_h + \text{RN}(y_\ell \times e))$ and $y_h \neq \text{RN}(y)$; that case should never occur since it means that the rounding test is not correct;
- **negative:** $(y_h \neq \text{RN}(y_h + \text{RN}(y_\ell \times e)$ and $y_h \neq \text{RN}(y)$; in such a case, a different strategy (e.g. a tighter approximation) should be used;
- **false negative:** $(y_h \neq \text{RN}(y_h + \text{RN}(y_\ell \times e))$ and $y_h = \text{RN}(y)$, which causes us to unduly switch to a more expensive strategy; this case must occur as infrequently as possible.

More precisely, the parameters involved in our experiments satisfy:

- $p = 53$;
- $y_h \in [1, 2)$;
- $y_\ell \in [\frac{15}{32}\text{ulp}(1), \frac{16}{32}\text{ulp}(1))$;
- $\epsilon \in (2^{-81}, 2^{-55}]$;
- as a consequence, $e \in [1.0000000074505808, 2.0000000000000004]$.

2.7.2 Stressing Ziv's rounding test under various relative error values

Our experiments consist in 100,000,000 ϵ values randomly picked in each interval $[2^{-k}, 2^{-k+1})$, for k varying from -81 to -56.

For each value ϵ we pick:

- one random $y_h \in [1, 2)$;
- one random $y_\ell \in \left[\frac{15}{32}\mathrm{ulp}(y_h), \frac{16}{32}\mathrm{ulp}(y_h)\right]$.

The results are presented in Table 2.2.

ϵ	Positives	False Positives	Negatives	False negatives
$[2^{-55}, 2^{-56}[$	0	0	100 000 000	0
$[2^{-56}, 2^{-57}[$	0	0	99 997 603	2397
$[2^{-57}, 2^{-58}[$	23 284 922	0	76 116 019	599 059
$[2^{-58}, 2^{-59}[$	61 962 104	0	37 740 947	296 949
$[2^{-59}, 2^{-60}[$	80 926 366	0	18 924 027	149 607
$[2^{-60}, 2^{-61}[$	90 593 819	0	9 332 714	73 467
$[2^{-61}, 2^{-62}[$	95 167 993	0	4 794 367	37 640
$[2^{-62}, 2^{-63}[$	97 625 196	0	2 356 226	18 578
$[2^{-63}, 2^{-64}[$	98 776 533	0	1 213 736	9 731
$[2^{-64}, 2^{-65}[$	99 399 403	0	596 044	4 553
$[2^{-65}, 2^{-66}[$	99 703 144	0	294 513	2 343
$[2^{-66}, 2^{-67}[$	99 853 799	0	145 113	1 088
$[2^{-67}, 2^{-68}[$	99 924 441	0	74 914	645
$[2^{-68}, 2^{-69}[$	99 962 425	0	37 261	314
$[2^{-69}, 2^{-70}[$	99 981 576	0	18 286	138
$[2^{-70}, 2^{-71}[$	99 990 768	0	9 154	78
$[2^{-71}, 2^{-72}[$	99 995 282	0	4 677	41
$[2^{-72}, 2^{-73}[$	99 997 740	0	2 239	21
$[2^{-73}, 2^{-74}[$	99 998 842	0	1 148	10
$[2^{-74}, 2^{-75}[$	99 999 413	0	584	3
$[2^{-75}, 2^{-76}[$	99 999 715	0	284	1
$[2^{-76}, 2^{-77}[$	99 999 853	0	147	0
$[2^{-77}, 2^{-78}[$	99 999 928	0	70	2
$[2^{-78}, 2^{-79}[$	99 999 955	0	44	1
$[2^{-79}, 2^{-80}[$	99 999 981	0	19	0
$[2^{-80}, 2^{-81}[$	99 999 991	0	9	0

Table 2.2: Results of our experiments, for $\epsilon \in (2^{-81}, 2^{-55}]$.

For the largest values of ϵ, the values of e are so large that the test fails systematically (yielding

a "negative" result) because, due to our "worse case" bias for computing it, on the one hand, y is certain to be larger than or equal to $y_h + \text{ulp}(y_h)$ and, on the other hand $\text{RN}(y_\ell \times e) \geqslant \frac{1}{2}\text{ulp}(y_h)$. This indicates that the theoretical bound on ϵ in Theorem 2.1 is probably not a good practical one. Consequently, at least in the $p = 53$ case, one should favor a sharper initial approximation accuracy to avoid a too frequent resort to high-accuracy approximation.

As ϵ decreases, so does e and y has less latitude to wander beyond $y_h + \text{ulp}(y_h)$. As a consequence, the number of "negatives" decreases too and so does the number of "false negatives". At some point, the "false negatives" seem to vanish totally: the probability of a "false negative" is too low for them to show up among the tested cases, and all we have left is a small and decreasing (by a factor 2 each time ϵ itself decreases by a factor 2) number of "negatives".

The comforting result is that, at no point, do we have a "false positive".

2.7.3 Near optimality of the value of e provided by Theorem 2.1

In order to minimize the number of "false negatives" e must be as small as possible (and yet large enough to allow for a correct Ziv's test). Hence, it is important to assess how optimal is the value of e given by Theorem 2.1). Let us call it e^*. As we could not prove it mathematically to be minimal, we will try to shave off tiny amounts from e^* and check when we start getting "false positives". In several experiments, in the $p = 53$ context, we started getting them at $e^* - 2^{-31}$. One should not conclude from this experiment that the "Real and True Optimal Constant" is shyly ensconced somewhere in the $[e^* - 2^{-31}, e^*]$ interval. Neither could one lightheartedly pick any value in this interval to brew an "optimized" and yet safe Ziv's rounding test of their own. We will not venture further than saying that the bound given by Theorem 2.1 is rather efficient for practical purposes. Table 2.3 provides a handful of examples of failures at $e = e^* - 2^{-31}$.

Example 1		
$y_h = 1.9947587312896187$	$y_\ell = 1.1102229216237452 \times 10^{-16}$	$\epsilon = 5.1757461373254229 \times 10^{-24}$
$\mathrm{RN}(y) = 1.9947587312896189$	$y = 1.9947587312896187805577597160974603068928590135237284347920284065279757344416324$	$e = 1.000000092772301$
Example 2		
$y_h = 1.9883894880686108$	$y_\ell = 1.1102229932848 77 \times 10^{-16}$	$\epsilon = 1.5927638864921235 \times 10^{-24}$
$\mathrm{RN}(y) = 1.988389488068611$	$y = 1.98838948806861093299147572569526564594932805547535275484086046830843325121040$	$e = 1.0000000282270229$
Example 3		
$y_h = 1.9802155978993843$	$y_\ell = 1.1102230085276806 \times 10^{-16}$	$\epsilon = 8.2224179116152375 \times 10^{-25}$
$\mathrm{RN}(y) = 1.9802155978993845$	$y = 1.980215597899384394331434596375631826389445396327677003067177359863566352$	$e = 1.0000000143465304$

Table 2.3: Results of our experiments, for $\epsilon \in (2^{-81}, 2^{-55}]$.

2.7.4 How do the simplified and the accurate formulas for e compare in practice?

As we noted in §2.5.4, p. 109, we were able to figure out a formula to compute a bound for e that only uses the target p-precision floating-point arithmetic. Two formulas were given

$$e^{up}_{simple} = \mathrm{RU}\left(\frac{1 + 2^{-p+1}}{1 - 2^{p+1}\epsilon}\right), \text{ and}$$

$$e^{nearest}_{simple} = \mathrm{RN}\left(\frac{1 + 2^{-p+1}}{1 - 2^{p+1}\epsilon}\right).$$

The latter, always smaller or equal to the former, is preferable since it yields a more efficient test. Unfortunately, we were not able to prove it larger or equal to the bound computed from the accurate formula, in the general case. This has only been done in particular cases (see Property 2.9).

Table 2.4 displays the results of some tests done in the IEEE-binary64 format, for approximation accuracy bounds varying from 2^{-56} to 2^{-90}. Notice that in all the aforementioned developments, we have considered $\epsilon \leqslant 2^{-p-2}$. In this context we should start at 2^{-55}. But when $\epsilon \in (2^{-56}, 2^{-55}]$ we

have found many instances where $e_{simple}^{nearest} \leqslant e^*$.This is a limited annoyance since, as can be found up from §2.7.2, p. 128, in this approximation accuracy interval the test is far from pertinent (all our corner cases are "negative", but that could also happen for many of the more "conventional" cases). Again, these experiments do not fill the void of the absence of proof. Nevertheless they foster our intuition about the validity of the simplified bound.

Maximum relative error $\frac{e_{simple}^{nearest} - e^*}{e^*}$	$2.2204460492339477 \times 10^{-16}$
$e_{simple}^{nearest} = e^*$ cases	1 514 615 533
Total number of cases	3 500 000 000

Table 2.4: Accurate (e^*) and simplified ($e_{simple}^{nearest}$) bounds comparison

2.8 Conclusion

First of all, we have exhibited formulas that allow for the computation of bounds for the e constant in the Ziv's rounding test. We have proved that, under some realistic conditions, if one uses a value of e larger or equal to these bounds, she can perform reliable (i.e. no "false positives" at all) and yet effective (with a low probability of "false negatives") tests.

We have also shown that, by using state-of-the-art technologies like FMA instructions, the safety domain of the test extends, to put it in a bit over- simplistic terms, to the subnormal range (see Section 2.6.2 for a more delineated description).

Most of the results are precision-independent, which broaden their generality. This in-depth analysis pulls out this utensil from the realm of "Black Magic" and anoints it as a dependable tool for practitioners. At the same time it confirms the intuitions and practices of our forerunners (starting with Abraham Ziv himself) who designed and used the test.

We have extensively mentioned correctly-rounded (usual elementary) functions as the main application field. Others exist, as, mainly, the implementation of "heterogeneous" operations (more information about it can be gathered from [24]) and others may be devised.

Is it the end of the story for Ziv's rounding test, from a research perspective? Not yet, could we answer. As noticed before, some holes in the proofs (optimality or, possibly, a better bound, correctness in general cases) still need to be plugged up.

Besides, our investigation limited itself to radix 2. It may be interesting to produce equivalent results for, at least, radix-10, as it will possibly be used more and more extensively, out of it original "financial" application domain. Nevertheless, as can be seen from our proofs, the issue is still highly radix-dependent and we do not envision yet (even if we can not totally dismiss it) the possibility of a general, radix-independent and nevertheless optimal solution.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Coppersmith's Method and the TMD

An implementation of the SLZ-algorithm.

3.1 Motivation: floating-point numbers beyond 64 bits

The development of floating-point number systems is still a work in progress. The last major normalization round held in 2008 (IEEE 754-2008) did not speak the final word about it.

In particular, the standard stepped ahead in the direction of a larger precision representation than that of the "historical" 64-bit (aka `double precision` with its companion `binary64` interchange format[1]) and 80-bit (aka `double-extended precision`, for which no interchange format was defined) representations that still are the bread-and-butter of today's numerical computations. A 128-bit interchange format, `binary128`, was defined in order to anticipate the users expectations in this realm. Rules to define interchange formats were even set for even larger precision, and it is then possible to figure out; for instance, what `binary256` or `binary1024` would look like. Nevertheless, only `binary128` was incorporated into the *basic formats* list [47].

While no mainstream hardware implementation has shown yet [2] support for 128-bit floating-point arithmetic, some processor architectures, notably SPARC and IBM POWER, have dedicated instructions for 128-bit formats but, when called, they trigger a trap to software emulation rather

[1] For the sake of brevity we will refer to the floating-point systems by the name of their interchange format. The former define the precision, the minimal and maximal exponents, the bias and other characteristics (rounding modes and rules, NaN's, subnormal numbers, etc.). The latter merely describe how data should be laid out for interchange between systems. This does not rule out the idea that interchange formats can be cleverly designed in order to make them a sensible choice for internal representation. But the distinction should remain clear.

[2] One should not mistake 128-bit registers and AVX vector operations in Intel processors for the support of a "true" 128-bit floating-point arithmetic.

than being performed at full metal speed. Even then, the support of IEEE 754-2008 requirements cannot be taken for granted throughout.

Nevertheless, many other initiatives have been taken, mainly in software, to mention a few :

- the `gcc` compiler `__float128` type and the companion `libquadmath` library [67];
- some version of the `icl` Intel compiler `_Quad` type and the companion MKL library (apparently discontinued);
- the C++ Boost `float128` type in the multi-precision library [69] that is a thin wrapper over the previous two;
- the IBM 128-bit `long double` type, as supported in its AIX®operating system, that exhibits a limited compatibility with IEEE 754 standard.

Space and dedication lack to make an accurate census of all the attempts to implement `binary128` in software. We are not considering here the generic arbitrary precision libraries that, on their way, can also be used with 113-bit significands[3]. We refer here to specialized 128-bit libraries. Notice that a great confusion exists since there are also libraries that offer higher precision arithmetic as "`double-double`" (or even a higher number of `double`'s [4] [21]) to represent "higher precision" floating-point numbers but they are a different beast than IEEE 754 `binary128`. The very number of these initiatives is an indication that this topic is becoming hotter and hotter.

Here, we place ourselves in the perspective of the full support of the IEEE 754-2008 standard and, more particularly, of correctly-rounding. The aforementioned initiatives try to deal, achieving variable degrees of compliance with IEEE 754-2008, with arithmetic operations. The next step is correct rounding of the elementary functions for which the key point is solving the Table Maker's Dilemma. The TaMaDi Project [105] was started in 2010 to tackle several issues regarding floating-point numbers arithmetic. One of its main goals was to evaluate options for the search of hard-to-round cases in the `binary128` and `decimal128` formats in order to allow for the development of correctly rounded libraries.

As explained in §1.6.3.3, p. 68, while, thanks to the L-algorithm, the solution of the TMD is at hand reach for `binary64` it is known that this solution will not work efficiently enough for `binary128`.

In this context, the work presented in this chapter stems of two kinds of motivations:

[3]Notice, passing, that arbitrary precision libraries do not mimic the exact behavior of `binary128` when precision is set to 113 bits since, for instance, they do not have the same exponent range or lack support of subnormal numbers.

1. "solve", in possibly different senses detailed below, the Table Maker Dilemma;

2. check if, with the SLZ-algorithm, we are able to tackle problems that are out of reach of our most elaborated tool to date, the L-algorithm and its implementations.

We are not sure either that the SLZ-algorithm will allow us to solve the TMD satisfactorily, in the sense of setting exactly the hardness-to-round for a function or at least attain a very sharp upper bound. This bound should live in the neighborhood of $2p$ (in precision p) as the probabilistic model suggests. But this case is very difficult to test.

D. Stehlé, in his PhD thesis [99], gives a theorem that states that hard-to-round cases can be found in polynomial time complexity of the input for a hardness-to-round in the order of $O(p^2)$. This can be seen as a great improvement over Y. Nesterenko and M. Waldschmidt's bound (it guarantees a distance larger than $2^{-18,563,653}$ between the image of any rational number corresponding to a 113-bit floating-point number (a number in the domain) by the exp function and a rational number in 113-bit precision (a number in the image)), it is still impractical for an implementation. In 113-bit precision, it takes an approximation accuracy a bit better than $2^{-18,563,653}$.

For the interval $[3/2 - 2^{-7}, 3/2 + 2^{-7}]$, no known tool[4] could compute the degree of a sharp enough approximation polynomial (not to speak about the polynomial itself!).

Of course, there are other (more mathematical) methods that allow for the computation of the necessary degree. Such an evaluation may be interesting to get a clear idea of how intractable the Nesterenlo and Waldschmidt's bound is.

If we assume that we use Taylor polynomials (as will be seen in §3.6, p. 160, this is what we actually do) to approximate the $\exp(x)$ function, between 0 and 1, the approximation error of a degree-n such polynomial is given by the Taylor-Lagrange formula:

$$\exp(x) = \sum_{k=0}^{n} \frac{x^k}{k\,!} + \frac{x^{n+1}}{(n+1)\,!}\,\exp(t), \text{ with } t \in [0,1]. \tag{3.1}$$

The error term here is

$$\frac{x^{n+1}}{(n+1)\,!}\,\exp(t) \leqslant \frac{e}{(n+1)\,!}.$$

where e denotes the base of the $\exp(x)$ function.

[4] In particular, we gave Sollya a very hard try. We had to put down the process running Sollya after 12 636 minutes (near 9 days), at the point where, using over 82 GB of memory and still ballooning, it was about to trigger the system into swapping.

The Stirling formula implies that:

$$(n+1)\,! \geqslant \sqrt{2\pi(n+1)}\left(\frac{n+1}{e}\right)^{n+1}.$$

Therefore, the approximation error of the degree-n Taylor polynomial is upper-bounded by

$$\frac{e}{\sqrt{2\pi(n+1)}\left(\frac{n+1}{e}\right)^{n+1}}$$

This value become less than $2^{-18,563,653}$ when $n \geqslant 1,003,750$.

Even if our bound is not absolutely tight, it is safe to say that using a polynomial with such a degree and with the huge coefficients implied, is not a reasonable way to go for approximation. In the case of the exponential function, other techniques can be used, based, for instance, on the AGM for the evaluation of the $\log(x)$ function and the $x_{n+1} = 1 + a - \log(x_n)$ Newton iteration where a is an initial approximation of $\exp(x)$. Even then, the evaluation of the approximation would require either the development of a specialized high precision arithmetic library or, and more safely, the use of an existing arbitrary precision library as MPFR with the associated time and memory costs.

But we are somehow comparing apples to oranges here. On the one hand, the Nesterenko and Waldschmidt's bound is an absolute result. How discouraging as it is, no worse surprise is to be expected beyond that. On the other hand, with a polynomial complexity, very large constant terms and/or coefficients in the complexity formula can make things much more difficult than they seem at first sight.

If a direct attack in the p^2 precision is beyond our possibilities, what can be tried instead, and succeeding would not be a trivial result, is to make sure that hardness-to-round is smaller than, say, $6p$, $8p$, $10p$ or even $12p$.

This would break ground for a three-step (once all special cases have been dealt with) viable function implementation strategy. Steps would differ in accuracy, expressed here as a function of precision p. A first approximation could be made at an accuracy of $2^{-(p+20\,\%\,\mathrm{of}\,p)}$. This would correspond, for `binary128`, to what is done in the "fast phase" approximation currently used in the implementation of `libm`'s. Should this fail, another step could be taken at $2^{-(2p+a\_small\_fraction\_of\_p)}$ accuracy (that, from our probabilistic model, should work most of the time) but it would be more expensive to compute. It corresponds, more or less, to the "slow phase" of an implementation in 53-bit precision since the actually known hard-to-round cases for this format are all in this precision frame.

Eventually a very high accuracy (and possibly very expensive to compute) approximation (with an accuracy in the 2^{-8p} - 2^{-12p} range) that would be (hopefully) seldom, if ever, called, should the first two fail. The question of the implementation technique of this last step is left open to debate: should one use an arbitrary precision library as MPFR or create from scratch a dedicated high precision computing framework?

That is mainly what we are trying to do here: work out an implementation of SLZ that would allow us to check values in the 2^{6p} to 2^{12p} hardness-to-round bracket.

Before we go further into the motivations, let us reset the stage.

3.1.1 Setting the stage

Most of the description can be done in general terms, we will specialize it only when it becomes absolutely necessary. Let us start with some floating-point system $\mathbb{F}_{2,p,emin,emax}$. Our interest is in the rounding-to-nearest, tie-break to even rounding mode. Nevertheless we must say here that, from Lindeman's Theorem and corollaries (see [5]), ties never happen for the $\exp(x)$ function and many other common ones. Most of what we say here can be applied to other tie-breaking rules. We study the hardness-to-round, as defined in §1.6.3.3, p. 61 for this mode, of some function $f : \mathbb{F} \to \mathbb{F}$.

From now on, we assume that we have dealt with or eliminated all the troublesome parts such as:

1. parts of $\mathbb{F}$ where $f(x)$ is not defined;
2. parts of $\mathbb{F}$ such that their image by $f(x)$ is an infinity;
3. argument reduction;
4. parts of $\mathbb{F}$ whose image by $f(x)$ does not require complex computations (e.g. $\mathrm{RN}(\exp(x)) = 1$, when x is "small enough", depending on precision);
5. elements of $\mathbb{F}$ whose image by $f(x)$ are midpoints [5].

At this point, we end up with a domain $\mathcal{D} \subset \mathbb{F}$. This is where we actually want to study the hardness-to-round of $f(x)$ for rounding-to-nearest mode. As expressed in our motivations, we want to make it in a "reasonable" amount of time which rules out exhaustive search of the possible options.

[5] For some functions, as explained in the introduction, §1.6.3.1, p. 58, no image is a rational number (e.g. $\exp(x)$). For others we assume some way can be devised to get rid of them. As we will see SLZ-algorithm could also catch these cases but if there were overwhelmingly many of them they could jam the algorithm. Hence trying to spot them beforehand is still a good option.

3.1.2 "Solving" the TMD

In the round-to-nearest mode we are essentially interested in the distance from the images to the midpoints of $\mathbb{F}$. We should always keep in mind that they are not elements of $\mathbb{F}$.

Let us call $\mathcal{M}$ the set of the midpoints corresponding to the $\mathbb{F} \cap [1,2)$ set. In other words,

$$\mathcal{M} \;=\; \left\{ \mu \in \mathbb{R} \text{ such that, } \exists\, X \in \mathbb{F} \cap [1,2) \text{ such that } \mu = \frac{X^+ + X}{2} \right\},$$

where, as usual in this document, X^+ denotes the successor of X in $\mathbb{F}$ [6].

Let us call **nn** (for Normalized Notation) a function that maps a real number x to the absolute value of its "scientific" notation in base-2 deprived of its scaling term.

$$\forall\, x \in \mathbb{R},\, \mathbf{nn}(x) \;=\; x_{nn} \;=\; \begin{cases} \left| \dfrac{x}{2^{\lfloor \log_2(|x|) \rfloor}} \right|, & \text{if } x \neq 0 \\ \\ 0, & \text{otherwise.} \end{cases}$$

In more fuzzy terms, the function **nn** operates similarly to the computation of the normalized significand of a floating-point number, only generalized to real numbers.

Strictly speaking, solving the TMD over domain $\mathcal{D} \subset \mathbb{F}$ consists in finding the smallest $m \in \mathbb{N}$ such that, $\forall\, X \in \mathcal{D}$, and for all $\mu \in \mathcal{M}$, $|\mathbf{nn}\,(f\,(X)) - \mu| \geqslant \dfrac{1}{2^m}$.

Hunting directly for the smallest m is a difficult task and no one has yet devised an algorithm that shoots straight into the bullseye[7]. What can be done instead is to figure out some value $\widetilde{m}$, and try to check it.

We have assumed, when we have defined $\mathcal{D}$, that all $X \in \mathcal{D}$ such that $f(X) \in \mathcal{M}$ had been cleaned up beforehand. Depending on the inner workings of the checking algorithm, this may not be absolutely necessary if it can detect such a situation without falling into an infinite loop. Of course, and as already stated, these points should be dealt with specifically.

When checking if $\widetilde{m}$ is over the hardness-to-round, we can get into one of the two following situations:

[6] For the largest midpoint, $X \in \mathbb{F} \cap [1,2)$ while $X^+ \notin \mathbb{F} \cap [1,2)$.
[7] Even if this what the L-algorithm indirectly does.

- the first is:
$$\forall\, X \,\in\, \mathcal{D},\, \forall\, \mu \,\in\, \mathcal{M},\, |\mathbf{nn}\,(f(X)) \,-\, \mu| \,\geqslant\, \frac{1}{2^{\widetilde{m}}};$$
- the second is:

  there exists some X (or several of them) for which $\exists\, \mu \,\in\, \mathcal{M}$ such that $|\mathbf{nn}\,(f(X)) \,-\, \mu| \,<\, \frac{1}{2^{\widetilde{m}}}$.

In the first case, $\widetilde{m}$ is an upper bound to m. Though not optimal this information can be very interesting since, as noted above, if $\widetilde{m}$ is relatively small (sharp, in regard to m), a reasonably efficient implementation of a correctly rounded approximation can be devised on these grounds.

The second case, $\widetilde{m}$ is a lower bound to m. It is interesting only if $\widetilde{m}$ is a tight lower bound, especially if the cases where $|f(X) \,-\, \mu| \,<\, \frac{1}{2^{\widetilde{m}}}$ can be precisely spotted[8] and if there are not to many of them, to avoid hampering the algorithm operation with the special precessing these situations require. Indeed, it can be worth computing high-precision values of $f(x)$ for these particular cases, make an *ad hoc* analysis that would culminate in the recovery of "the" exact value of m when all these cases have been dealt with.

3.1.3 Back to motivations

In this work, the SLZ-algorithm will be mainly used to work out upper bounds (in the $6p$ to $12p$ bracket) for `binary128` floating-point numbers system. Why do we not try to compute an exact value for the hardness-to-round? As it will be seen, the complexity of the algorithm grows when $\widetilde{m}$ shrinks and searching for a very sharp value can still be out of the range of our tools.

3.2 History serves: the L-algorithm

Having a look to the L-algorithm is not only a not a ritual "hats off" to a dignified predecessor. For one, the L-algorithm is not a mummy in display in some computer science mausoleum. It is alive and running. The ever-improved original strain is still hunting for hard-to-round cases in the evaluation of elementary functions with `binary64` arguments. A full description of the seminal work can be found in [63]. As already mentioned, it has also given birth to recent investigations in order to adapt it to state-of-the-art parallel computation technologies (on CPU and GPU, see [37]).

This quick look back is also for us an opportunity to introduce some of the concepts shared by the L-algorithm and the SLZ-algorithm and, of course, to point out some differences between them.

[8] The algorithm returns a more precise answer than " Well, there is one or several X out there for which $|f(X) \,-\, \mu| \,<\, \frac{1}{2^{\widetilde{m}}}$".

As we have already said, the very first method to solve the TMD is checking all the possible inputs for the closeness of their image to break point. But we have also noticed that this process becomes intractable when the precision p grows. This is still the case for `binary64` with $p = 53$ despite the pioneer work realized in 2011, based on FPGA (see [25]). The L-algorithm, and the SLZ-algorithm as well, are methods that improve on enumeration by avoiding pinpoint checking as much as possible.

3.2.1 The TMD as curve-to-grid distance problem

As shown on Figure 3-1, a hard-to-round case can be viewed as a situation where the vertical distance between a curve (the graph of the function f), and a grid (made of the points that represent, in abscissa, the floating-point numbers and, in ordinate, the breakpoints for the different rounding modes[9]) is very small.

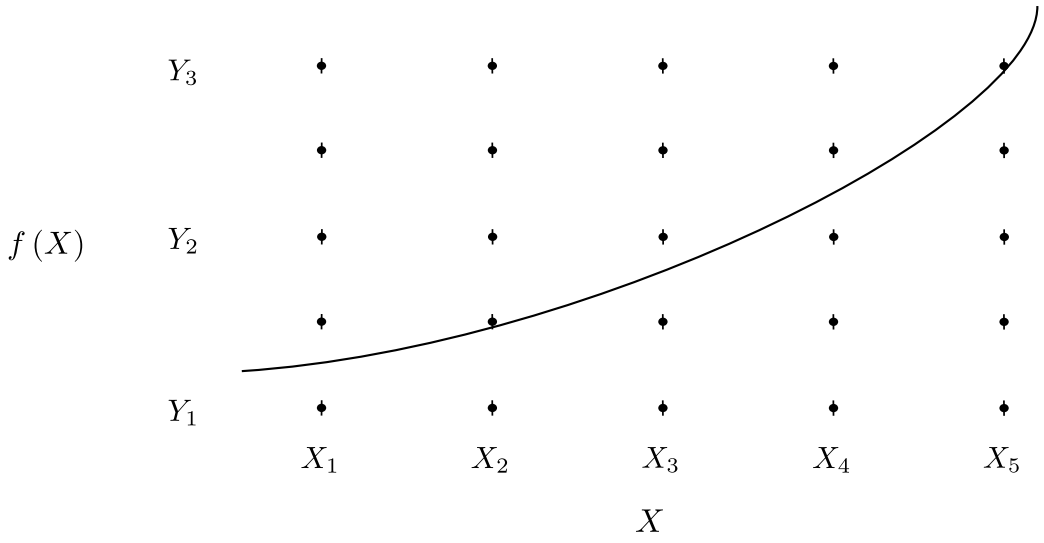

Figure 3-1: Hard-to-round cases for the function f happen when its graph hits one of the small vertical bars place on the dotted grid.

The tiny (but still very exaggerated in length) vertical bars on grid points in Figure 3-1 indicate the threshold that, when "hit" by the curve, triggers a close examination for hardness-to-round. In Figure 3-1, we have this situation for $f(X_2)$ (for the rounding-to-nearest mode) and $f(X_5)$ (for the directed rounding modes). Solving the TMD comes down to this process of (hopefully scarce)

[9]Floating-points are breakpoints for directed rounding modes, midpoints are for rounding-to-nearest.

hard-to-round case candidates quick insulation and their scrutiny.

It turns out that this process can be enhanced and regularized (from our initial rectangular grid to a square one) if we consider the function f and its reciprocal f^{-1} at the same time, as on Figure 3-2. The extra horizontal bars represent now the threshold for the reciprocal function. Here, $f^{-1}(Y_2)$ can possibly be problematic.

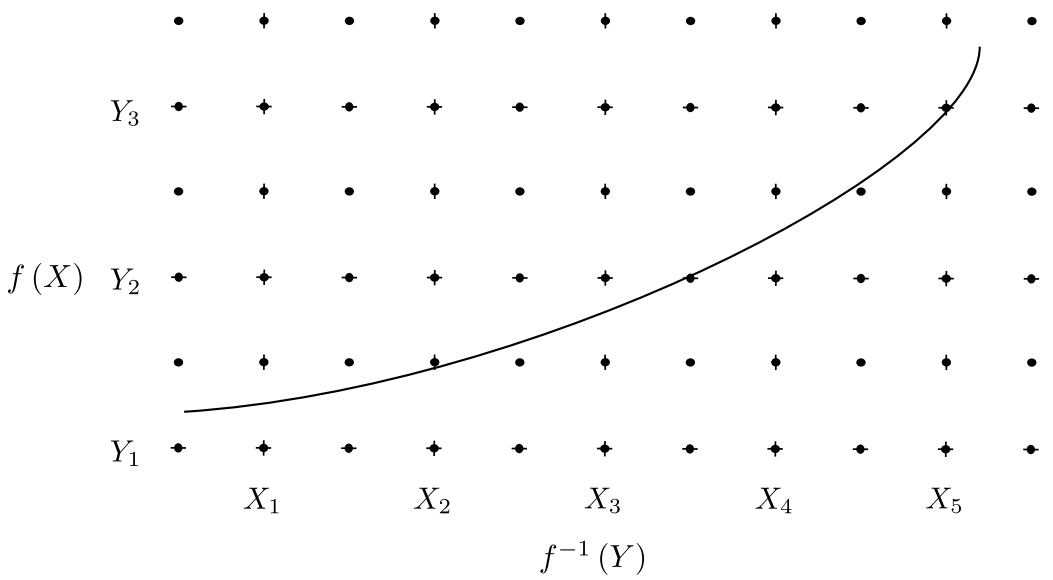

Figure 3-2: We can consider both the function f and its reciprocal with a new "square" grid.

We can give more formal expression of our pictorial observation. Let us call X an element of the break-points set of the domain of the function f. This includes the floating-point numbers (as they are breakpoints for the directed rounding modes) and the midpoints (breakpoints for the round-to-nearest rounding mode). Let us call Y an element of the set of floating-point numbers and the corresponding midpoint in the image of the function f, such that Y is, in this set, the closest number to $f(X)$. Checking if X is a hard-to-round case can be done by verifying if $|Y - f(X)| < \delta$, where δ is a small positive real number. Let us also assume, without any loss of generality [10] that the function f is strictly increasing and that $X < f^{-1}(Y)$.

If we assume that X is a hard-to-round case for the function f and applying the Mean Value

[10] For other situations, all we have to do is switch the bounds of the intervals accordingly.

Theorem for this function, we can write:

$$|Y - f(X)| = \left| f'(\xi) \left(f^{(-1)}(Y) - X \right) \right| < \delta, \tag{3.2}$$

where $\xi \in (X, f^{-1}(Y))$.

From Equation 3.2 we can see that if X is hard-to-round case for the function f, and provided $|f'(\xi)|$ is not too small, then the value of the expression $\left|f^{-1}(Y) - X\right|$ is also (or can also be) small, since, by hypothesis, the value $\left|Y - f^{(-1)}(X)\right|$ is small. Hence, Y is a good candidate for being a hard-to-round case for the function f^{-1} and worth checking.

Considering all the points of our new grid duplicates the number of necessary checks but does the work for two functions at the same time. It is not more expensive than considering both functions independently since other gains can be expected from, for instance, computing only one function approximation. Nevertheless, it may lead us to useless checks for points that do not make sense as, for instance, the point at coordinates $(\text{midpoint}(X_4, X_5), \text{midpoint}(Y_2, Y_3))$. Useless check can also be induced if our hypothesis on the smallness of the derivative is not verified.

But there is more. One can carefully pick which function (f or f^{-1}) one will use to make the checks. In the configuration presented in Figure 3-2 and for an enumerative method, it is quicker to consider the reciprocal function f^{-1} than the function f since there are less cases to deal with. For a more convincing and real-world detailed example, see [78], p. 435.

3.2.1.1 Working with a linear approximation

As already stated, function f will not be routine-evaluated with some arbitrary precision library as MPFR. We absolutely want to limit the invocation of such a library to checking only serious contenders for hard-to-round case because of the involved overhead. The most common approximating functions family is polynomials and among them, apart from the trivial case of a mere constant, their simplest form is degree-1. This is what is used in the case of the L-algorithm for reasons that will become clearer latter latter on. Our previous curve-to-grid distance becomes a segment-to-grid distance one.

But before we go to the approximation step, let us, in order to simplify the expression of the problem, scale the input and the output values so that we work with the first non-negative integers instead of the original rational values as the indeterminate of our polynomial. More precisely, the scaling takes the following form. We assume that:

- the distance between two consecutive input values is β^{-v}, where β is the floating-point system radix and $v \in \mathbb{Z}$;
- the distance between two consecutive breakpoints in the image is $\beta^{-w}/2$, where $w \in \mathbb{Z}$.

Instead of the restriction of the function f to the set $\{X_{\ell o}, \ldots, X_{up}\}$, with $X_{\ell o} < X_{up}$ and $X_{\ell o} \in \mathbb{F}$, $X_{up} \in \mathbb{F}$, we consider now the function g defined over the set of integers $\{0, \ldots, N-1\} \subset \mathbb{N}$ such that:

$$g(k) = 2 \times \beta^w f\left(X_{\ell o} + \beta^{-v} k\right).$$

What corresponds to a hard-to-round case is now a non negative integer $k < N$ such that $g(k)$ is very close to an integer. Finding one such case comes down, for some small number $\delta_0 \in \mathbb{R}^+$, to solving:

$$\exists u \in \mathbb{Z}, |g(k) - u| < \frac{\delta_0}{2}$$

What we do here is approximate the function $g(k)$ (and not $f(x)$) by a degree-1 polynomial $P(k) = b_0 - ak$, with $a \in \mathbb{R}$ and $b_0 \in \mathbb{R}$. Taking into account the approximation error with a new small real number $\delta \in \mathbb{R}^+$, the problem becomes:

$$\exists u \in \mathbb{Z}, |(b_0 - ak) - u| < \frac{\delta}{2}.$$

In other words, in order to get rid of the absolute value, we have:

$$\exists u \in \mathbb{Z}, -\frac{\delta}{2} < (b_0 - ak) - u < \frac{\delta}{2}.$$

With this formulation, we have to check the distance of $b_0 - ak$ to an integer in both directions, up and down. We can easily transform the problem into a single check, to the "below" integer, by slightly shifting "upward" the $y = b_0 - ak$ segment, using $b = b_0 + \delta$ as the intercept in our linear function. The problem becomes:

$$\exists\, u \in \mathbb{Z}, (b - ak) - u < \delta.$$

With a classical definition (see [38], p. 70 and [43], p. 110) of the *fractional part* function (also often noted $\{x\}$):

$$\text{frac}(x) = x - \lfloor x \rfloor,$$

we can restate the problem as finding the non-negative integers $k < N$ such that $\text{frac}(b - ak) < \delta$, keeping in mind that $\forall\, x \in \mathbb{R},\ \text{frac}(x) > 0$. As noted above, this simplifies somehow the search.

Figure 3-3 illustrates this last formulation. We have a regular square grid, where, in abscissa,

we find the first values of k (from 0 to 8). In ordinate we have successive integer values (from n to $n+4$). The lower row displays the floating-point numbers of the domain that are represented by the values of k. The outer left column displays the floating-points numbers of the image, and the midpoints in between as well.

The partial graph of linear function $b-ak$ is also displayed and the vertical arrows represent the values of the function $\text{frac}(b-ak)$ for the different values of k. For $k=2$ and $k=7$ these fractional parts are so small that they are hardly legible on the figure. For $k=4$ the fractional part is almost equal to 1. Depending on the value of the constant δ, $u=n+1$ and $u=n+3$, for, respectively, $k=2$ and $k=7$ are possible hard-to-round candidates.

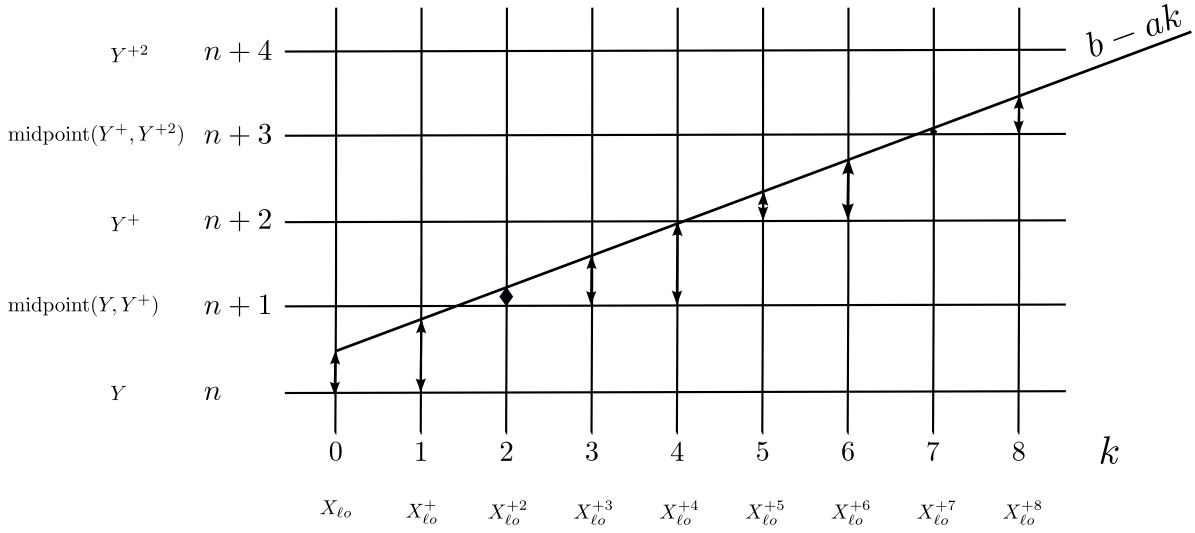

Figure 3-3: Hard-to-round cases search reformulated as the search of values of $\text{frac}(b-ak)$ smaller than a bound depending on the investigated hardness to round.

3.2.1.2 Avoiding an exhaustive search

An obvious way to check the values of k for which a hard-to-round case exists is to enumerate the values of $\text{frac}(b-ak)$ for $k \in \{0,\ldots,N-1\}$. In fact this is the only option if one wants an exact result. Nevertheless it is possible to make checking faster, yielding an approximate result in the sense that more possible hard-to-round cases are detected than necessary but none of the actual ones is missed (see [78] p.440-442).

It happens that the succession of $\text{frac}(ka)$, with $k \in \{0,\ldots,N-1\}$ and a an irrational number,

has a special structure[11]. The position of the successive values of frac(ka) on the $[0, 1)$ interval do not happen at random but according to a, if not simple yet predictable, pattern.

Exploiting this particular structure allows for the search of a lower bound on frac($b-ak$) without the enumeration the k's. If, from the value of this bound, one deduces that a possible hard-to-round case is buried in the interval, she can resort to exhaustive search to localize it. The lower bound search does not generate false negatives (intervals pronounced free of hard-to-round cases but actually containing one).

The general shape of the algorithm strongly reminds that of the subtractive euclidean one for the GCD (see [78], p. 441). One of the niceties (among others) of the algorithm used for the lower bound computation is that, at the expense of minor modifications, it yields the one used for pinpoint searches. In the actual implementation, one also has to deal with other issues as, for instance, the fact that our hypothesis on the irrationality of a is not relevant for machine numbers.

3.2.1.3 Quick approximation computation

The above sketched technique for the search of the hard-to-round cases, relies on degree-1 polynomial approximations. While this warrants the exploitation of a structure on frac(ka), it generates annoyances of its own.

Degree-1 polynomials have a limited approximation ability. This implies that the domain of the function must be split into many small subdomains (remember: in the order of $2^{2p/3}$ per binade, as analyzed in [99], p. 164-165). As a consequence, as many approximation polynomials have to be computed. These computations do not come cheap even with quick techniques as Taylor truncated series (see §3.3.3, p. 159). The very narrowness of the intervals makes them quick to check and by contrast the overhead occasioned by the computation of the approximants becomes more detrimental in this context.

Here too some optimization is possible when one remarks that if a new approximant must be computed for each new interval, it is not very different from the just used one. If one can devise a method that cheaply (in computing terms) derives the next polynomial from the current one (or the former ones), instead of computing it from scratch, some gains may be expected.

This is what is achieved through the *Finite Difference Method* (used here to compute successive

[11] A particular and different structure also exists when $a \in \mathbb{Q}$ but we are more interested in the $a \in \mathbb{R} \setminus \mathbb{Q}$ case.

values of a polynomial without evaluating it through traditional methods such as Horner's rule) and the application of this method to the computation of the coefficients of the next polynomial from those of the previous ones. Detailed explanations can be found in [63] and [64].

The dependence on degree-1 approximation polynomials has been seen as the main limitation of the L-algorithm. Hence using higher degree ones was the starting point for the development of the SLZ-algorithm. But this change in approximation technique has had a deep impact on the subsequent parts of the computations since, as we have seen, the L-algorithm heavily relies on the fact that the sequence of the approximations from one point to the next form an arithmetic progression that allows for exploration methods inspired from the euclidean GCD algorithm. This property is lost with higher degree approximation polynomials. Using the Coppersmith method is the response to these new conditions.

After this historical and technical introduction, it is time to get to the heart of our matter.

3.3 First steps towards the SLZ-algorithm

This section is very extensive for two main reasons:

1. the algorithm is itself rather complex;
2. a large batch of preparatory work is necessary to level the ground before a single algorithm can be used to deal with the whole problem.

We can not directly dive into the heart of the problem before some housekeeping chores are performed and we have the problem formulated in proper terms for the techniques that will be used thereafter.

3.3.1 Hair Splitting

We will often discuss splitting here. But what we are about to perform here is a first step that has more to do with problem conditioning than with the usual reasons for domain splitting in approximation (e.g. operate on a sufficiently narrow interval so that a polynomial of a given degree d approximates the function accurately enough).

These reasons, and more specific ones, will be invoked when we talk again about interval splitting.

What we eventually want is to work on "normalized" domains. All our endeavor here is to prepare this normalization and try to make sure it is correctly done.

We mentioned that we were dealing now with a subset of $\mathbb{F}$ that we call $\mathcal{D}$. It is a finite set and, as such, it has a smaller (resp. larger) element we shall call lb (resp. ub). Most of the time, we will denote this set as $[\![lb, ub]\!]$. From now on, we shall make a distinction between real intervals (e.g. $[a, b]$) and finite subsets of $\mathbb{F}$, $\mathbb{Z}$ or $\mathbb{N}$ denoted by their bounds (e.g. $x \in [\![lb, ub]\!] \Leftrightarrow (x \in \mathbb{F}$ and $lb \leq x \leq ub)$). In addition, and to be more accurate, notice that in particular situations, some "trouble making" elements of $\mathbb{F}$ may be missing from $[\![ub, lb]\!]$. What we have in mind here are subsets of $\mathbb{F}$ from which some element(s) X has (have) been removed for specific reasons (e.g. $f(X)$ is not numerical).

3.3.1.1 Monotonic splitting

We will first split $\mathcal{D}$ into a family of subsets $\{\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_r\}$ such that $f(x)$ is monotonic on each of them; if no splitting takes place (i.e. the function is monotonic), we end up with a single subset $\mathcal{D}_0$.

3.3.1.2 Subdomain binade-wise splitting

We will, if necessary, split each of the domains $\mathcal{D}_\rho$ obtained at the previous step into a family of subsets $\{\mathcal{D}_{\rho,0}, \mathcal{D}_{\rho,1}, \ldots, \mathcal{D}_{\rho,s}\}$ such that each of them fits into a single binade; if some $\mathcal{D}_\rho$ is not split, it yields a single $\mathcal{D}_{\rho,0}$ set.

3.3.1.3 Subdomain image binade-wise image splitting

Eventually, and if necessary, we will split each of the $\mathcal{D}_{\rho,\sigma}$ sets into a family of subsets $\{\mathcal{D}_{\rho.\sigma.0}, \mathcal{D}_{\rho,\sigma,1}, \ldots, \mathcal{D}_{\rho,\sigma,t}\}$ such that $f(\mathcal{D}_{\rho,\sigma,k})$ fits into a single binade; again, if some $\mathcal{D}_{\rho,\sigma}$ is not split, it yields a single $\mathcal{D}_{\rho,\sigma,0}$.

3.3.1.4 Splitting in pictures

Notice that these splitting operations are always possible since $\mathcal{D}$ (and offspring) and $f(\mathcal{D})$ (and offspring) are all finite sets.

Notice also that this can lead to unnecessary splitting if, on some subset $\mathcal{D}$ included in a single binade, $f(x)$ is not monotonic over $\mathcal{D}$ but $f(\mathcal{D})$ "lives" in a single binade: think of, for instance, some periodic function as in Figure 3-4.

From Figure 3-5 we can see why it is first necessary to split into monotonic domains if we want the image of each final domain to fit into a single binade.

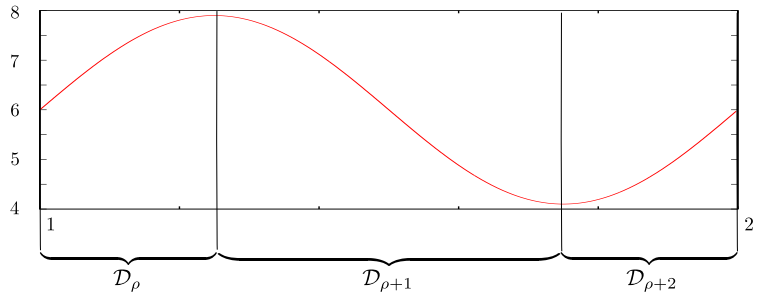

Figure 3-4: The $[1, 2)$ interval domain is split in three subdomains $\mathcal{D}_\rho$, $\mathcal{D}_{\rho+1}$, $\mathcal{D}_{\rho+2}$ based on monotonicity. This is not absolutely relevant here, at least at this stage, since both the domain and the image live in a single binade.

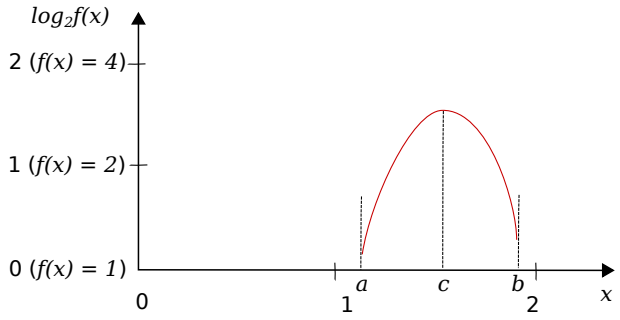

Figure 3-5: While $[a, b]$ domain of $f(x)$ is contained in single binade, $f([a, b])$ spans several ones. Both $f(a)$ and $f(b)$ belong to the same binade, and yet $f(c)$ belongs to a different one.

The fact alone that the image of the lower bound $f(lb)$ and the image of the upper bound $f(ub)$ belong to the same binade does not guarantee that the image of any intermediate values does. In our example $f(c)$ does not.

The full splitting process is illustrated and recapped in Figure 3-6.

From now on, we will assume that we are working on some interval $\mathcal{D}_{x,y,z}$ obtained from the process described above.

What comes next is a reformulation of the hard-to-round cases in terms that will be more suited to the techniques we want to use hereafter.

3.3.2 Hardness-to-round revisited

Let us assume we work with a precision-p floating-point system $\mathbb{F}_{2,p,emin,emax}$. We introduce here new notations about binades.

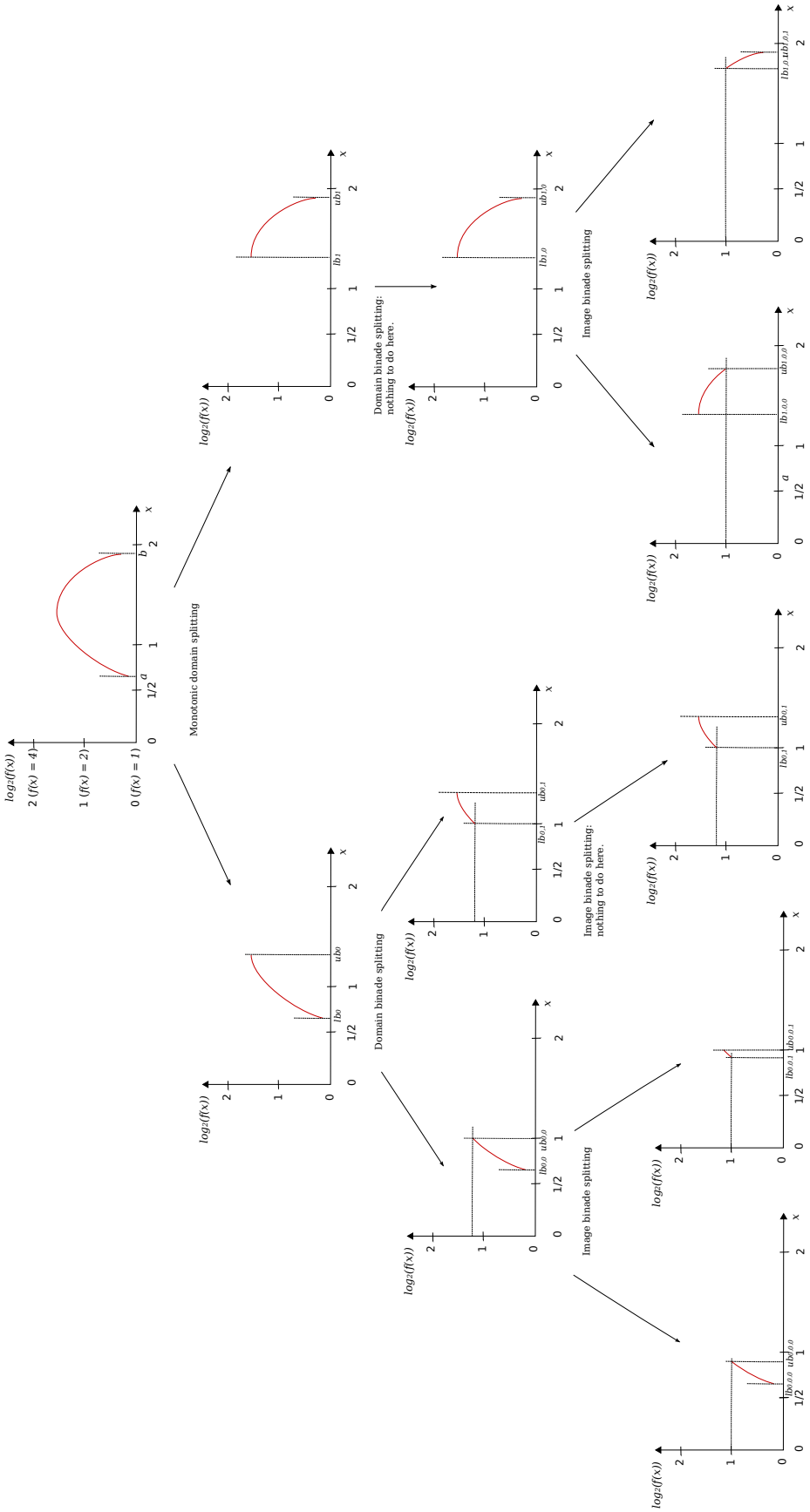

Figure 3-6: Domain and image splitting.
The domain and image splitting according, first, to monotonicity, and then to the binade spanning of both the domain and the image.

3.3.2.1 Leveling the playground: toying with binades

As defined in the introduction (§1.4.3, p. 41), let us denote ℓ-binade the following interval over $\mathbb{R}$: $[2^\ell, 2^{\ell+1})$ (e.g. the 0-binade is the $[1, 2)$ interval).

Let us denote as k-fbinade, the "restriction to $\mathbb{F}$ of the k-binade" that holds all the $X \in \mathbb{F}$ such that $X \in [2^k, 2^{k+1})$ (e.g the 0-fbinade is all the $X \in \mathbb{F}$ such that $X \in [1, 2)$).

If p is the precision of the floating-point system, then

$$k\text{-fbinade} = \left[\!\!\left[\frac{2^{p-1+k}}{2^{p-1}}, \frac{2^{p+k} - 2^k}{2^{p-1}} \right]\!\!\right].$$

In our notation $-k$-fbinade will denote

$$\text{all the } X \in \mathbb{F} \text{ such that } X \in \left[\!\!\left[-\frac{2^{p+k} - 2^k}{2^{p-1}}, -\frac{2^{p-1+k}}{2^{p-1}} \right]\!\!\right].$$

Notice that k is an element of $\mathbb{Z}$. Nevertheless, when $k < 0$, $-k$-fbinade $\neq$ $|k|$-fbinade. The initial explicit minus sign and the implicit one attached to the negative integer k do not relate to the same thing and can not be simplified according to the usual signed numbers multiplying and dividing rules. The initial minus sign indicates that we are dealing with negative floating-point numbers. The implicit sign of k relates to the magnitude of these numbers. Fortunately we will never get caught in such a confusing "clash of signs" situation in the forthcoming developments.

We will also add a *subnorm-fbinade*, that, for the positive range, will hold all the elements of $\mathbb{F}$ smaller than $2^{e_{\min}}$ and larger or equal to 0. At this point our definition diverges from that used for the "regular" fbinades (this fbinade span an infinity of binades as defined over $\mathbb{R}$) but it allows for a practical decomposition of $\mathbb{F}$.

Let us now decompose $\mathbb{F}$ into the list of the fbinades it is made of [12] as follows:

$$\mathbb{F} = \dot{\bigcup_{i \in [\![-(e_{\max}), -(e_{min})]\!] \cup [\![e_{\min}, e_{\max}]\!]}} i\text{-fbinade}$$
$$\dot{\bigcup}$$
$$-\text{ subnorm-fbinade } \dot{\cup} \text{ subnormal-fbinade.}$$

[12] We acknowledge that infinities and all not-a-number are missing here.

3.3.2.2 Stretching and shrinking

We consider now some function $f : \mathbb{F} \to \mathbb{R}$ that we want to correctly round into $\mathbb{F}$[13].

From our previous splitting work, the search of the smallest m (or checking of some $\widetilde{m}$) is well specified since any $\mathcal{D}_{\rho,\sigma,\tau}$ fits into a single (f)binade as well as $f(\mathcal{D}_{\rho,\sigma,\tau})$ does. But, for a given hardness-to-round m, the actual distance between $f(X)$ and the midpoints one has to check differs from one binade to another, as shown in Figure 3-7.

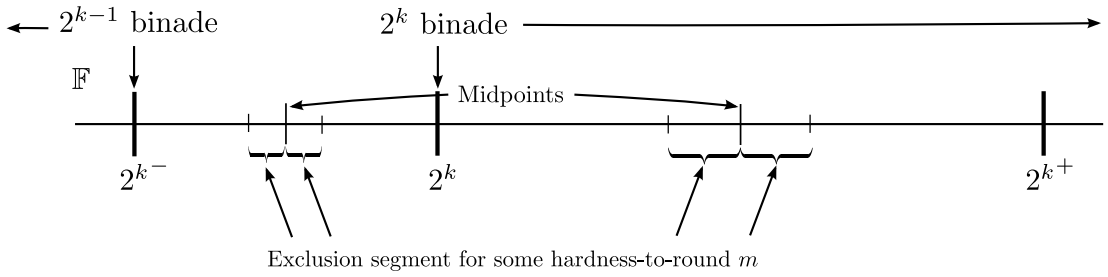

Figure 3-7: We have represented here the "exclusion segments" where $f(X)$ should not live if we were to prove that some integer m is the hardness-to-round of $f(x)$. If we think of it in absolute terms, the width of the segments changes when we change binades.

One may want to simplify the problem by creating the conditions where the distances to consider, between the floating-point numbers and the midpoints, are the same for all subsets $\mathcal{D}_{\rho,\sigma,\tau}$, (in the domain of $f(x)$ or in its image). In a sense, this is what we did for our definition of the hardness-to-round. To this end, we will now take an extra step in order to normalize the subdomains (each included in some k-fbinade) and their images (each included in some ℓ-fbinade) as if they were both included in the 0-fbinade.

This operation will allow us to have a single formulation of the algorithm that deals with different binades, either for the domain or for the image. Normalizing to the 0-fbinade is a natural choice, but any other is possible, as long as bookkeeping for denominators, approximation accuracy in relation to hardness-to-round, etc. is ensured.

We can easily realize that by scaling any k-fbinade or ℓ-fbinade to the 0-fbinade. Remember that, in a precision-p floating-system, all fbinades hold the exact same number of elements: 2^{p-1}. A one-to-one correspondence between the elements of two fbinades boils down to a multiplication by

[13] We could want to have an image in different floating-point system. Conceptually, it would not make any difference, at the cost of limited, but terse, notations adaptations.

a power of 2. This is what we do, back and forth, between any k-fbinade and the 0-binade.

Let us call $\underset{m\to n}{scale}(x)$ the scaling from the m-fbinade to the n-fbinade.

With this definition, $\underset{0\to k}{scale}(x)$ maps all the elements of the 0-fbinade to those of the k-binade, as does $\underset{\ell\to 0}{scale}(x)$ for all the elements of the ℓ-binade to those of the 0-fbinade.

From function $f(x)$, that computes images from the k-binade into the ℓ-binade, let us define a new function $g:$ 0-binade $\to$ 0-binade as:

$$g(x) = \underset{\ell\to 0}{scale}\left(f\left(\underset{0\to k}{scale}(x)\right)\right).$$

If we set $y = \underset{0\to k}{scale}(x)$, the function $g(x)$ behaves exactly the same as $f(y)$ in terms of hardness-to-round:

- the initial scaling from the domain of $g(x)$ to that of $f(y)$ maps a floating-point, x_0, onto another one, x_k, exactly (no rounding ever takes place) since the mapping is a multiplication by a power of 2;
- nothing changes for $f(y)$, that computes images from the k-binade into the ℓ-binade;
- the final scaling from the ℓ-binade to the 0-binade changes the value $f(x_k)$ but not its behaviour in terms of hardness-to-round; more accurately this scaling is necessary to make the application of our definition of hardness-to-round (§1.6.3.3, p. 61) possible; this definition considers the "normalized significands"[14] both for the exact value of $f(x_k)$ and for the midpoints; all these are elements of the 0-binade; this is exactly what the final scaling does.

As everything is now properly scaled back and forth, without any loss of information, we will operate, from now on, only over the 0-(f)binade for both the domain and the image.

Obviously each fbinade, or chunk of fbinade, in the domain will require a different $g(x)$ function since scaling changes from one binade to another (this is how they were built in the first place). Nevertheless we can use the same function for different chunks as long all their elements (resp. the images thereof by $f(y)$) belong to the same binade (resp. the images all belong to the same binade), what remains possible since splitting can happen for other reasons (e.g. monotonicity) than binade change.

We have reduced our problem of finding the hard-to-round cases of function $f(y)$ over $\mathbb{F}$ (or a large subset of it) into multiple sub-problems of finding the hard-to-round cases of a family of func-

[14] An almost unbearable abuse of language since we speak of a possibly infinite string of digits (for $f(x_k)$) and a "significand" written in $p+1$ digits for the midpoints!

tions $g_1, g_2, \ldots, g_n$ with their domain in the 0-fbinade (remember we are only interested in images of floating-point numbers) and their image in the 0-binade (these images can be anywhere in the 0-binade, all the problems stem from that). We could even perform the rounding of the images in the 0-fbinade (to be accurate, marginally, from time to time, some rounding would be to 2, and $2 \notin$ 0-fbinade). Scaling the rounded value back to the image domain is equivalent to performing rounding on the image itself.

At this point, we have to expand our analysis with a new issue. Indeed, we will not be actually working with the "real" function $g(x)$ itself (which is, admittedly, only a very thin wrapper around $f(y)$), but we will rather have to use a machine computable approximation function. In the case of the SLZ-algorithm this function will be a polynomial. The problem shifts now from the hardness-to-round of an arbitrary function to that of a polynomial. It will allow us both to use the same technique for any function and to leverage on some particular properties of polynomials

Let $P(x)$ be such an approximation of $g(x)$. For the moment, we do not specify the accuracy of this approximation, this will come a bit later and will be related to the value of m one wants to test. We introduce $P(x)$ here in order to stabilize the notations in this new part of our analysis.

3.3.2.3 From inequality to modular equations

In the 0-fbinade and in precision p, floating-point numbers are rational numbers of the form $\frac{i}{2^{p-1}}$ with $i \in [\![2^{p-1}, 2^p - 1]\!]$.

The following relations stand, as long as the accuracy of the approximation of $g(x)$ by $P(x)$ is sufficient[15]:

$$g\left(\left[\!\left[\frac{2^{p-1}}{2^{p-1}}, \frac{2^p - 1}{2^{p-1}}\right]\!\right]\right) \subset \text{0-binade},$$
$$P\left(\left[\!\left[\frac{2^{p-1}}{2^{p-1}}, \frac{2^p - 1}{2^{p-1}}\right]\!\right]\right) \subset \text{0-binade}.$$

From now on, we completely remove $g(x)$ out of the picture (and the long gone $f(y)$ as well). We can restate the previous in terms of the approximation polynomial $P(x)$ as:

$$\forall\, i \in [\![2^{p-1}, 2^p - 1]\!]\,,\ \exists\, j \in [\![2^{p-1}, 2^p - 1]\!] \text{ such that } \frac{j}{2^{p-1}} \leqslant P\left(\frac{i}{2^{p-1}}\right) < \frac{j+1}{2^{p-1}}$$

since all the $\frac{j}{2^{p-1}}$ belong to the 0-binade.

[15] Another abuse of language here. Of course, the domain of $P(x)$ may not be the full binade, we spent enough time in splitting to be well aware of that. The same is true for the image, but it does not change anything here and acting this way helps to keep notations more legible. Also notice that, with a (too) low accuracy approximation, we could have many situations where for element $X \in \left[\!\left[\frac{2^{p-1}}{2^{p-1}}, \frac{2^p-1}{2^{p-1}}\right]\!\right]$, $P(X) \notin$ 0-binade.

We can now define the midpoint that is immediately larger than $\frac{j}{2^{p-1}}$ as $\frac{2\,j\,+\,1}{2^p}$.

At this point we have two options for $P\left(\frac{i}{2^{p-1}}\right)$:

- $P\left(\frac{i}{2^{p-1}}\right) \;=\; \frac{2j\,+\,1}{2^p}$, in other words $P\left(\frac{i}{2^{p-1}}\right)$ is an exact midpoint;
- $P\left(\frac{i}{2^{p-1}}\right)$ is not such a midpoint.

In the latter case, $\exists\, m_{i,j} \in \mathbb{N}$ such that

$$\frac{1}{2^{m_{i,j}}} \;\leqslant\; \left| P\left(\frac{i}{2^{p-1}}\right) - \frac{2\,j\,+\,1}{2^p} \right| \;<\; \frac{1}{2^{m_{i,j}\,+1}}.$$

This situation, for some i, is depicted in Figure 3-8.

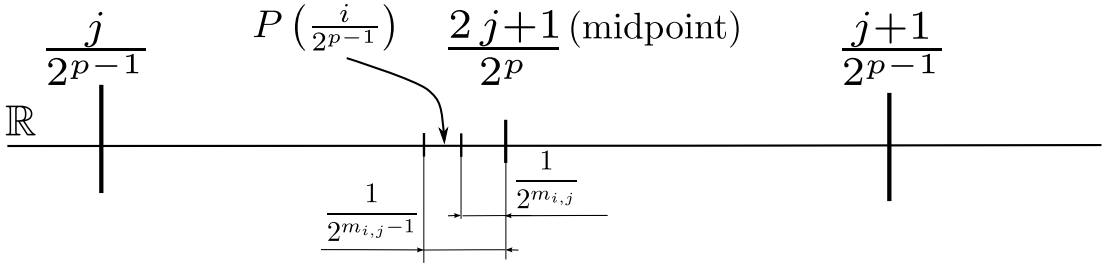

Figure 3-8: The image by polynomial $P(x)$ of some $\frac{i}{2^{p-1}}$ lives between $\frac{1}{2^m}$ and $\frac{1}{2^{m-1}}$ from a midpoint.

If, from $[\![2^{p-1}, 2^p\,-\,1]\!]$, we get rid of all the cases where $P\left(\frac{i}{2^{p-1}}\right)$ is an exact midpoint, we obtain what we will denote $[\![2^{p-1}, 2^p\,-\,1]\!]^*$. One could think that the this distinction is pointless in the case of the $\exp(x)$ function and of all the others Liouville's Theorem and subsequent lemmas deal with. But we must remember here that $P(x)$ is an approximation. We then have to perform the midpoints clean-up in order to have our definition make sense.

We can now define m, the hardness-to-round of $P(x)$ over the 0-fbinade as:

$$m \;=\; \max(m_{i,j}), \text{ for all } i \in [\![2^{p-1}, 2^p\,-\,1]\!]^*, \text{ with } j, \in [\![2^{p-1}, 2^p\,-\,1]\!].$$

3.3.2.4 Back to approximation precision

This is where the $g(x)$ function makes a short comeback. As mentioned above, we left aside the problem of the accuracy of the approximation of $g(x)$ by $P(x)$. We even took for granted the fact that $g(x)$ could be approximated by a polynomial. At this point we come back to these issues. To all the restriction we have placed on $g(x)$, considered as a function over the real numbers on some interval $[a, b]$, we must append the two following conditions :

1. the $[a, b]$ interval is closed and bounded;

2. the function $g(x)$ is continuous over this interval.

In turn, these conditions can rise new issues. For the first one, we must remark that we are only interested in approximations over bounded intervals (we do not care about what happens beyond the minimum and the maximum numerical values represented in $\mathbb{F}$) and we can always have closed intervals of interest with floating-point (rational) numbers. So the condition always holds. For the second, if $g(x)$ is at least piecewise continuous, the solution is extra splitting into subdomains where the function is continuous.

Why are these conditions appended? In order to be able to apply the Weierstrass Approximation Theorem[109], [110] that states that any continuous function defined over a closed and bounded interval can be uniformly approximated on that interval by polynomials and to any degree of accuracy. Or put in more formal way: If $g(x)$ is a continuous real-valued function on $[a, b]$, for any given $\epsilon > 0$ there exists a polynomial of degree d, p_d, such that

$$\forall\, x \in [a, b],\ |g(x) - p_d(x)| < \epsilon. \tag{3.3}$$

We will deal soon with the question of actually computing approximations. For the time being, we just recall the results that make it theoretically possible. First, we can use to our advantage another result stating that, for a given degree d, there is a unique optimal polynomial p_d^* in the sense that $||g - p_d^*||_\infty$ is minimal (e.g. equal to some ϵ^*). The reader may refer to [11] for a detailed discussion and a proof.

Furthermore, there is a constructive algorithm that, for a given degree d, computes this optimal polynomial[16]. This is the famous Remez Algorithm [86], [88], [87] that was followed by several improvements.

Notice that p_d^* is a real number coefficient polynomial. But in practice, as we will machine-compute it with a tool like Sollya and as we want it to be machine-evaluable for further processing, what we will eventually get is a polynomial whose coefficients are rational numbers with bounded numerator and denominator. Let us call it $P_d(x)$ or more briefly $P(x)$ if the we do not need to embed the degree in the notation.

The difference between p_d^* and P_d that is most relevant to our present discussion is that the latter will generally be less accurate than the former relatively to $g(x)$. If the rational number coefficients

[16] In fact the optimal polynomial with a degree lower or equal to d.

polynomial is not accurate enough, we will still have options:

- allow for more bits for the coefficients of P_d, by performing the approximation with a better precision;
- switch to a higher degree polynomial;
- do some extra interval splitting.

What stands for real numbers of the domain of $g(x)$ is also true for rational ones. We switch now from the interval $[a, b]$ to the rational numbers set $[\![\frac{lb}{2^{p-1}}, \frac{ub}{2^{p-1}}]\!]$ where $\frac{lb}{2^{p-1}} = a$ and $\frac{ub}{2^{p-1}} = b$ [17]. This set holds all the elements of $\mathbb{F}$ comprised between $\frac{lb}{2^{p-1}}$ and $\frac{ub}{2^{p-1}}$. At the same time we will replace p_d^* by its rational surrogate $P(x)$:

$$\forall i \ \in \ \left[\!\!\left[\frac{lb}{2^{p-1}}, \frac{ub}{2^{p-1}}\right]\!\!\right], \ \left| g\left(\frac{i}{2^{p-1}}\right) - P(i) \right| < \epsilon.$$

As stated before, we are not going to compute "the" hardness-to-round of $g(x)$ directly but rather check if a given integer value m is larger or equal to this (unknown) hardness-to-round[18].

Let us assume now, without loss of generality, that $g(x)$ is an increasing function[19]. With our present notation and state of affairs (i.e. after all the above presented substitutions are made) we are trying to prove about the above a given m that:

$$\text{If } \forall\, i \ \in \ [\![lb, ub]\!], \ \forall\, j \ \in \ \left[\!\!\left[\left\lfloor g\left(\frac{lb}{2^{p-1}}\right) 2^{p-1} \right\rfloor, \left\lceil g\left(\frac{ub}{2^{p-1}}\right) 2^{p-1} \right\rceil\right]\!\!\right], \ g\left(\frac{i}{2^{p-1}}\right) \ \neq \ \frac{2j+1}{2^p},$$

then

$$\forall i \in [\![lb, ub]\!], \forall j \ \in \left[\!\!\left[\left\lfloor g\left(\frac{lb}{2^{p-1}}\right) 2^{p-1} \right\rfloor, \left\lceil g\left(\frac{ub}{2^{p-1}}\right) 2^{p-1} \right\rceil\right]\!\!\right],$$

$$m \ \in \mathbb{N} \text{ is such that } \left| g\left(\frac{i}{2^{p-1}}\right) - \frac{2j+1}{2^p} \right| \geqslant \frac{1}{2^m}.$$

We are ready now to connect the approximation accuracy of $P(x)$ to the value m, as an upper bound to the actual hardness-to-round of $g(x)$.

[17] Since we are interested in images of elements of $\mathbb{F}$, the considered intervals should also have bounds in $\mathbb{F}$: our equalities are not overenthusiastic.

[18] We make it clear here that the value m we are dealing with is not "the" hardness-to-round as defined above but, as we want the former to have the same properties as the latter, we get along with the same notation.

[19] Remember we did split the domain first in intervals where the initial function $f(x)$ is increasing or decreasing. The function $g(x)$ behaves the same. If $g(x)$ is decreasing all we have to change are the bounds j lives in.

Lemma 3.1.

If $\forall i \in [\![lb, ub]\!]$, $\forall j \in \left[\!\left[\left\lfloor g\left(\frac{lb}{2^{p-1}}\right) 2^{p-1} \right\rfloor, \left\lceil g\left(\frac{ub}{2^{p-1}}\right) 2^{p-1} \right\rceil \right]\!\right]$, $m \in \mathbb{N}$ is such that

$$\left| P\left(\frac{i}{2^{p-1}}\right) - \frac{2j+1}{2^p} \right| \geqslant \frac{1}{2^{m+1}} \tag{3.4}$$

and

$$\left| g\left(\frac{i}{2^{p-1}}\right) - P\left(\frac{i}{2^{p-1}}\right) \right| < \frac{1}{2^{m+1}}, \tag{3.5}$$

then

$$m \in \mathbb{N} \text{ is such that } \left| g\left(\frac{i}{2^{p-1}}\right) - \frac{2j+1}{2^p} \right| \geqslant \frac{1}{2^m}.$$

Proof. From Expression 3.4 we have

$$\left| P\left(\frac{i}{2^{p-1}}\right) - g\left(\frac{i}{2^{p-1}}\right) \right| + \left| g\left(\frac{i}{2^{p-1}}\right) - \frac{2j+1}{2^p} \right| \geqslant \left| P\left(\frac{i}{2^{p-1}}\right) - g\left(\frac{i}{2^{p-1}}\right) + g\left(\frac{i}{2^{p-1}}\right) - \frac{2j+1}{2^p} \right| \geqslant \frac{1}{2^{m+1}}.$$

Combining with Expression 3.5, we have

$$\frac{1}{2^{m+1}} + \left| g\left(\frac{i}{2^{p-1}}\right) - \frac{2j+1}{2^p} \right| \geqslant \frac{1}{2^{m+1}}$$

Ultimately, we have

$$\left| g\left(\frac{i}{2^{p-1}}\right) - \frac{2j+1}{2^p} \right| \geqslant \frac{1}{2^m}$$

□

In order words, provided that

1. polynomial $P(x)$ approximates function $g(x)$ with an absolute accuracy smaller than $\frac{1}{2^{m+1}}$,
2. integer $m+1$ is an upper bound on the hardness-to-round of polynomial $P(x)$,

then integer m is an upper bound on the hardness-to-round of function $g(x)$.

The first condition sets the accuracy of the polynomial approximation of function $g(x)$ we will have to compute. The second describes the hardness-to-round problem we are going to solve about polynomial $P(x)$ instead of the initial one we had about function $g(x)$.

Let us examine now how polynomial approximation will be performed.

3.3.3 Taylor truncated series/polynomials

Before we try to deal with our final goal, we have to focus a moment on how we are going to compute the polynomial approximations. We have mentioned the Remez algorithm to assert the

theoretical and, in some way, practical possibility to compute polynomial approximations satisfying our accuracy requisites. But, in practice we take a different route. As the reader may have noticed we have been heavily splitting intervals above. The consequence is that we possibly end up with a very large set of subdomains. And there will be many $g(x)$ functions (up to possibly one for each subdomain obtained after the decomposition depicted in Figure 3-6). This can sum up to a lot of approximations to compute and, possibly, to a hefty amount of time if an expensive approximation computation technique is picked up. The Remez algorithm execution does not always come cheap. But there are other ways to compute function approximations that we can use. Taylor polynomials is one of them.

3.3.3.1 Taylor theorem

The following is a well known result. Full discussion and proof can be found in many textbooks such as [98], from p. 391*et seq.*

Theorem 3.2. *Suppose that function $g : \mathbb{R} \longmapsto \mathbb{R}$ is $n+1$ times differentiable on interval $[a, x]$ and that $R_{n,a}(x)$ is defined by*

$$g(x) \;=\; g(a) \,+\, g'(a)(x-a) \,+\, \ldots \,+\, \frac{g^{(n)}(a)}{n!} \,+\, R_{n,a}(x). \tag{3.6}$$

Then

$$R_{n,a}(x) \;=\; \frac{g^{(n+1)}(t)}{n!}\,(x-t)^n\,(x-a) \text{ for some } t \,\in\, (a,x),$$

$$R_{n,a}(x) \;=\; \frac{g^{(n+1)}(t)}{(n+1)!}\,(x-a)^{n+1} \text{ for some } t \,\in\, (a,x).$$

Moreover, if $g^{(n+1)}$ is integrable on $[a, x]$, then

$$R_{n,a}(x) \;=\; \int_a^x \frac{g^{(n+1)}(t)}{n!}\,(x-t)^n \,\mathrm{d}t.$$

Expression 3.6 is often called *Taylor's Formula*. The $g(a) + g'(a)(x-a) + \ldots + \frac{g^{(n)}(a)}{n!}$ expression is called the *Taylor polynomial of degree n for $g(x)$ at a*. The $R_{n,a}(x)$ expression is called the *remainder*. The three above forms of the remainder are respectively called Cauchy, Lagrange and integral form. With a simple change of variable $x = a + h$ we can obtain the following form of Taylor's formula

$$g(a \,+\, h) \;=\; g(a) \,+\, \frac{g'(a)}{1!}\,h \,+\, \frac{g^{(2)}(a)}{2!}\,h^2 \,+\, \ldots \,+\, \frac{f^{(n)}(a)}{n!}\,h^n \,+\, R_{n,a}(h)$$

Let us set a real positive value r and define the interval $[a - r, a + r]$ centered on a. Under this

last form of the Taylor's formula, we obtain a polynomial in h that computes approximate values of $g(a+h)$ in the $[a-r, a+r]$ interval. The maximum absolute value of the remainder $R_{n,a}(h)$ over $[a-r, a+r]$ is the approximation error. We can pick the Taylor polynomial as our approximation polynomial $P(x)$, if we make a final domain change.

Initially, we had set $\left[\frac{lb}{2^{p-1}}, \frac{ub}{2^{p-1}}\right]$ as the approximation polynomial domain. Let us compute $a = \left\lfloor \frac{ub+lb}{2} \right\rfloor$, $rl = lb - a$ and $ru = ub - a$. We will use $\left[\frac{rl}{2^{p-1}}, \frac{ru}{2^{p-1}}\right]$ as the domain for our Taylor's polynomial that computes an approximation to our initial $P(x)$. In the sequel, and in order to limit the notation bloat, we will, also call $P(x)$ the Taylor approximation polynomial.

As one can see, it can be computed very quickly for a given interval and a given degree. This functionality is available in many tools and, while less obvious from the formula alone, the approximation error can also be provided without the need of an extra expensive computation.

The choice of Taylor polynomials is not one of extra performance, but mainly for its convenience. We can always compute one, for a given degree d, that has the required approximation accuracy. If this not immediately the case, we resort to our trusted splitting technique: if the error is too large, the interval is split into two (or more) subdomains where, in turn, new truncated Taylor series are computed until a sufficient accuracy is achieved.

Another advantage stems from the change of variables imposed by the form we chose for Taylor polynomials. In general the maximum absolute value of the elements in set $[\![rl..ru]\!]$ will be smaller than that of the elements in set $[\![lb..ub]\!]$. We will see latter that this a desirable property.

This choice, of course, introduces an extra constraint on function $g(x)$: it must be differentiable to (at least) order d, where d is the degree of the polynomial we want to compute. Other properties may be desirable, as analyticity, if we want the approximation to have a sufficient accuracy and the polynomial to have a reasonable degree. Notice that, nevertheless, what comes next is agnostic (notations excepted) to the technique that is used to compute polynomial approximations. It could be reformulated to support other approximation options.

3.3.4 Hardness-to-round: the definitive reformulation

We have now to combine together all the elements we have introduced in the above sections to specify the exact problem we are going to work on. Those are essentially:

- a shift in targets from checking if m is an upper bound to the hardness-to-round of function

$g(x)$ to checking if $m+1$ is such an upper bound for the approximation polynomial $P(x)$;

- having the domain of our polynomial defined differently than that of function $g(x)$.

3.3.4.1 The image of the domain by the polynomial

Let us note $\min\limits_{i\in[\![rl..ru]\!]}\left(\mathrm{RD}\left(P\left(\frac{i}{2^{p-1}}\right)\right)\right)=\frac{j_l}{2^{p-1}}$ and $\max\limits_{i\in[\![rl..ru]\!]}\left(\mathrm{RU}\left(P\left(\frac{i}{2^{p-1}}\right)\right)\right)=\frac{j_u}{2^{p-1}}$.

Rounding here is considered relatively to the rationals $\frac{j}{2^{p-1}}(j\in\mathbb{Z})$.

We assume here that both $\frac{j_l}{2^{p-1}}$ and $\frac{j_u}{2^{p-1}}$ belong to the same binade. This assumption is not a thorny one since the domain we use for $P(x)$ is designed in such way that the image of its elements by $g(x)$, live all in the same binade. Of course, since $P(x)$ is only an approximation of $g(x)$, the image by $P'x$ of at least one such element could live outside the common binade. Should this happen, extra splitting of the interval in order to isolate the problematic cases could solve this uncommon issue.

If we now view $\left[\frac{j_l}{2^{p-1}},\frac{j_u}{2^{p-1}}\right]$ as an interval over $\mathbb{R}$, we have:

$$P\left(\left[\!\left[\frac{rl}{2^{p-1}},\frac{ru}{2^{p-1}}\right]\!\right]\right)\subset\left[\frac{j_l}{2^{p-1}},\frac{j_u}{2^{p-1}}\right].$$

3.3.4.2 Hardness-to-round reformulated

Checking if $m+1$ is the hardness-to-round for the round-to-next rounding mode of $P(x)$ (or at least an upper bound to it) over the $\left[\!\left[\frac{rl}{2^{p-1}},\frac{ru}{2^{p-1}}\right]\!\right]$ domain can eventually be restated as:

$$\forall i\in[\![rl..ru]\!],\ \forall j\in[\![j_l,j_u]\!],\ \left|\ P\left(\frac{i}{2^{p-1}}\right)-\frac{2j+1}{2^p}\ \right|\geqslant\frac{1}{2^{m+1}}. \tag{3.7}$$

We said in several occasions that we had to get rid first of all the cases where the image an element in the domain was an exact midpoint. It is important to stress this point again here. Let us suppose that one such a point of the domain, i_μ, was not filtered out. Its image by $P(x)$, $P\left(\frac{i_\mu}{2^{p-1}}\right)$, will be very close to some midpoint. But as $P(x)$ is an approximation, the image will not be exactly equal to this midpoint (and even if it were, this information would be nothing more than a hint). At this point in the computations, it may be very difficult to decide if, either, i_μ is a special case (an element whose image is a midpoint) or a particularly hard-to-round case.

Unless we have at hand, because of some simple mathematical property, an easy test to check if the image of element of the domain is a midpoint, the presence, even if residual, of special cases can make it difficult to verify that $m+1$ is or is not the actual hardness-to-round.

Out of the situation where one is absolutely sure that all special cases have been removed, and if the value of $m+1$ is large relatively to that of p, each element of the domain whose image by $P(x)$ is closer than $\frac{1}{2^{m+1}}$ to a midpoint deserves a very close scrutiny before concluding.

3.3.5 From rational polynomials to integer polynomials

Our next stage now is to transform our rational inequalities into integer inequalities. All we have to do is to clear out the denominators.

This is done in two steps:

- a simple change of variable $x = \frac{y}{2^{p-1}}$, with $y \in \mathbb{Z}$, yields a new polynomial

  $P_{rat,int}(y) = P\left(\frac{y}{2^{p-1}}\right)$ where $P_{rat,int}(y)$ has rational number coefficients and integer arguments.

- clear denominators by multiplying the terms that show up in the inequality of (3.7) by the least common multiple of the denominators of the coefficients of $P_{(rat,int)}$, 2^p and 2^{m+1}.

Now we have $P_{(int,int)}(i) \in \mathbb{Z}[i]$, with $i \in \mathbb{Z}$ (where i, by an unfortunate choice in notations, is our indeterminate for integer coefficients polynomials).

From that we can deduce a new inequality:

$$\left| P_{(int,int)}(i) - 2^K(2j+1) \right| \leqslant 2^k, \tag{3.8}$$

where:

- $P_{(int,int)}$ comes from the transformation (general reduction to the same denominator and denominators elimination) of $P_{(rat,int)}$;
- K and k have been computed in the course of the same reduction/elimination, for $\frac{1}{2^p}$ and $\frac{1}{2^{m+1}}$, respectively.

Notice that we still have $i \in [\![rl, ru]\!]$ and $j \in [\![j_l, j_u]\!]$.

3.3.6 From inequality to modular equation

Inequality 3.8 is not easy to solve. We now turn it into a new form that, though this is not obvious at first glance, will be easier to deal with.

As we want our unknowns to be small[20], we will replace $2j+1$ by $P_{(int,int)}(i) - 2^K(2j+1)$ as an unknown.

Indeed, since the coefficients of $P_{(int,int)}$ are all integers, for any i and j (themselves integers) the expression $P_{(int,int)}(i) - 2^K(2j+1)$ always evaluates to some integer t. Hence, Inequality 3.8 reduces to:

$$P_{(int,int)}(i) - 2^K(2j+1) = t, \text{ with } t \in \mathbb{Z} \text{ and } |t| < 2^k.$$

For $i \in [\![rl, ru]\!]$ and $j \in [\![j_l, j_u]\!]$, we want:

$$P_{(int,int)}(i) - 2^K(2j+1) = t, \text{ with } t \notin \{-2^k..2^k\}.$$

We can write $P_{(int,int)}(i) - t = 2^K(2j+1)$.

From which we can derive $(P_{(int,int)}(i) - t) \mod 2^K = (2^K(2j+1)) \mod 2^K = 0 \mod 2^K$.

Let us now define the polynomial $Q(i,t) \in \mathbb{Z}$ as $Q(i,t) = P_{(int,int)}(i) - t$, with $i,\ t \in \mathbb{Z}$. Let us also set $\mathcal{N} = 2^K$.

We have now $Q(i,t) \in \mathbb{Z}[i,t]$, with $i,\ t \in \mathbb{Z}$. The hardness-to-round check problem becomes:

$$\text{prove that } Q(i,t) \mod \mathcal{N} = 0 \text{ has no solution for } i \in [\![rl, ru]\!] \text{ and } t \in [\![-2^k, 2^k]\!]. \tag{3.9}$$

It may seem that no progress has been done so far since we are grappling now with a bivariate polynomial modular equation which is not at all easy to solve. But due to some particular characteristics of the problem[21], we will manage to transform it into a set of bivariate polynomial equations over the integers, which is a more amenable one.

3.3.7 From a modular equation to an equation over the integers

We are going to examine here the conditions under which the above announced transformation can take place. For this purpose, we are going to use an adaptation of the Coppersmith techniques devised for cryptanalysis purposes. When it comes to polynomial roots, Coppersmith has developed two different techniques:

- one for finding small roots of univariate modular polynomial equations [18];
- another for small roots of bivariate polynomial equations over the integers [17].

[20] The explanation for this "small is beautiful" preference will be given soon.

[21] Mainly the fact that we are looking for bounded and small solutions and not trying to solve the problem in general. More about this to come next.

Our main interest is in the first provided it is extended to cope with two variables since Coppersmith's seminal work is only about univariate polynomials. Furthermore, Coppersmith's initial algorithm was seen as a bit involved. Fortunately, clarification and extensions to bivariate polynomials have been realized in two steps by N. Howgrave-Graham (univariate untangled version[46]) and D. Boneh and G. Durfee (bivariate version[8]).

Regrettably, a property was lost in the extension process: while Coppersmith's algorithm is proved to actually yield all the small roots, bivariate extensions can only be dubbed as "heuristic" for a reason that will be detailed below.

These tools were initially developed in the context of the cryptanalysis of RSA, for solving what is known as "The small inverse problem" in this community. But their scope has been extending ever since.

We are going here to rephrase their formalization in the language of our own context. It all starts with the following definition and theorem.

Let $h(i,t) = \sum_{\rho,\sigma} a_{\rho,\sigma} i^{\rho} t^{\sigma}$. Assume that the polynomial $h(i,t) \in \mathbb{Z}[i,t]$ is a sum of, at most, w monomials.

Let us define:

Definition 3.1. $\|h(i,t)\|^2 := \sum_{\rho,\sigma} a_{\rho,\sigma}^2$.

Theorem 3.3. *Suppose that:*

1. $h(i_0,t_0) \mod \mathcal{N} = 0$ *where*
   $|i_0| < I$ *and* $|t_0| < T$ *(I and T are some positive integer constants), and*
2. $\|h(iI,tT)\| < \dfrac{\mathcal{N}}{\sqrt{w}}$.

Then $h(i_0,t_0) = 0$ holds over the integers.

In other words, if (i_0,t_0) a solution of the modular equation and two conditions (one on the absolute values of i_0 and t_0, another one (often referred to as the *Coppersmith condition*) on the norm of $h(i,t)$) hold, (i_0,t_0) is also a solution over the integers. The similarity with our problem is pretty obvious. The conditions on the absolute values of i_0 and t_0 translate almost transparently into the $i \in [\![rl, ru]\!]$ and $t \in [\![-2^k, 2^k]\!]$ ones. We will see below what can be done for the more

intricate condition on the polynomial norm.

Let us start first with the proof of Theorem3.3.

Proof.

$$|h(i_0, t_0)| \; = \; \left|\sum a_{\rho,\sigma} i^\rho t^\sigma\right| \; \leqslant \; \sum |a_{\rho,\sigma}||i|^\rho |t|^\sigma \; \leqslant \; \sum |a_{\rho,\sigma}| I^\rho T^\sigma.$$

By the Cauchy-Schwarz inequality:

$$\begin{aligned}
\sum |a_{\rho,\sigma} I^\rho T^\sigma| \quad & \leqslant \; \|(|a_{\rho_1,\sigma_1} I^{\rho_1} T^{\sigma_l}| \;, \ldots, \; |a_{\rho_w,\sigma_w} I^{\rho_w} T^{\sigma_w}|)\| \; \times \; \|(1 \;, \ldots, \; 1)\| \\
& \qquad\qquad\qquad \| \\
& \leqslant \; \sqrt{\sum |a_{\rho,\sigma} I^\rho T^\sigma|^2} \; \times \; \sqrt{\sum_1^w 1^2} \\
& \qquad\qquad\qquad \| \\
& \leqslant \; \sqrt{\sum |a_{\rho,\sigma} I^\rho T^\sigma|^2} \; \times \; \sqrt{w}.
\end{aligned}$$

In this case, we have $\sqrt{\sum |a_{\rho,\sigma} I^\rho T^\sigma|^2} = \|h(iI, tT)\|$ (if we aggregate I and T, at their respective powers, to the coefficients).

Hence:

$$|h(i_0, t_0)| \; \leq \; \|h(iI, tT)\|\sqrt{w}.$$

From condition 2 of Theorem 3.3:

$$|h(i_0, t_0)| \; \leq \; \|h(iI, tT)\|\sqrt{w} < \mathcal{N}.$$

Hence:

$$|h(i_0, t_0)| < \mathcal{N}.$$

Since, from condition 1 of Theorem 3.3,

$$h(i_0, t_0) = 0 \bmod \mathcal{N},$$

therefore

$$h(i_0, t_0) = 0.$$

□

3.3.8 Families of modular equations

Let us return to our problem (and notations) and assume now that (i_0, t_0) is a solution to our modular equation $Q(i,t) = 0 \mod \mathcal{N}$.

For reasons that will become clearer below, it is necessary to derive other modular equations from our original one, actually a full family of them, whose solutions, if any, include all those of $Q(i,t) = 0 \bmod \mathcal{N}$.

For instance, the above mentioned pair (i_0, t_0) is also a solution of $Q(i,t)^2 = 0 \bmod \mathcal{N}^2$. But there are many other modular equations that also share this property. To mention a few:

- $\mathcal{N}\ Q(i,t) = 0 \bmod \mathcal{N}^2$ also accepts (i_0, t_0) as a solution;
- as does $\ell\ \mathcal{N}\ Q(i,t) = 0 \bmod \mathcal{N}^2, \forall\, \ell\ \in\ \mathbb{Z}$, and, particularly, $i\ \mathcal{N}\ Q(i,t) = 0 \bmod \mathcal{N}^2$ and $t\ \mathcal{N}\ Q(i,t) = 0 \bmod \mathcal{N}^2$;
- as also do $i^2\ \mathcal{N}\ Q(i,t) = 0 \bmod \mathcal{N}^2$, $i\ t\ \mathcal{N}\ Q(i,t) = 0 \bmod \mathcal{N}^2$, $t^2\ \mathcal{N}\ Q(i,t) = 0 \bmod \mathcal{N}^2$ and so forth, since i and t are integers;
- as will also do any modular equation formed, for its left member, by a linear combination of the left members, $l_j(i,t)$, of the above given equations, $\sum \lambda_j\ l_j(i,t) = 0 \bmod \mathcal{N}^2, \lambda_j \in \mathbb{Z}$.

We can generalize our previous examples as follows. Let α be an integer parameter, if (i_0, t_0) is a solution to $Q(i,t) = 0 \bmod \mathcal{N}$, then any modular equation of the following form:

$$i^r\ t^s\ \mathcal{N}^u\ Q(i,t)^{\alpha-u} = 0 \bmod \mathcal{N}^\alpha, \text{ with } r, s \in \mathbb{N} \text{ and } u \in [\![0, \alpha]\!], \tag{3.10}$$

will also accept (i_0, t_0) as a solution.

We can even generalize further, to linear combinations of equations of the above form:

$$\sum_j \lambda_j\ i^{r_j}\ t^{s_j}\ \mathcal{N}^{u_j}\ Q(i,t)^{\alpha-u_j} = \mod \mathcal{N}^\alpha, \text{ with } \lambda_j \in \mathbb{Z}, r_j, s_j \in \mathbb{N} \text{ and } u_j \in [\![0, \alpha]\!]$$

The idea is, using these linear combinations, to build two polynomials of small norm so the second condition of Howgrave-Graham's Theorem holds for each of them. If this is the case, we have a system of two polynomials and two unknowns which we can try to solve over the integers.

If we can find roots over the integer – i.e. non-modular roots – (which is much easier here than solving modular equations), we will check them against our initial modular equation:

$$Q(i,t) \mod 2^K = 0 \text{ for } i \in [\![rl, ru]\!] \text{ and } t \in [\![-2^k, 2^k]\!].$$

In order to have the problem fully specified, we finally set the I and T constants as :

$$I = \max(-rl, ru) \text{ (recall } rl < 0 < ru) \text{ and } T = 2^k.$$

How are we going to generate polynomials of a sufficiently small norm? This where the LLL-algorithm comes into play.

3.4 Connecting the LLL-algorithm to our problem

While we use the LLL-algorithm essentially as a "black box" under the form of the `fpLLL` implementation [2], we feel it necessary to give here a few indications of what it does and of what are the main parameters involved in the *reduction* operation which is central to solve our problem.

3.4.1 A quick reminder about lattices

We will not dwell on lattices at large. We will only expose basic definitions and facts needed to solve our problem.

Definition 3.2. A lattice L is the set of all integral linear combinations of a family of $m \leqslant n$ linearly independent vectors $\mathbf{b}_i \in \mathbb{R}^n$:

$$L = \sum_{i=1}^{m} x_i \mathbf{b}_i,\ x \in \mathbb{Z}. \tag{3.11}$$

The vectors $\mathbf{b}_i$ are often referred to as a basis of L. The dimension of the lattice $\dim(L)$, is m (the number of vectors in the basis). Its degree, $\deg(L)$, is n (the dimension of each vector). When the degree and the dimension match, the lattice is said *full-dimensional*. Obviously, the degree of the lattice L cannot be smaller than its dimension.

As soon as $m \geqslant 2$, a lattice can be generated by an infinite number of different bases.

The previous definition can formulated under a matrix form:

Definition 3.3. Let $\mathbf{B} = [\mathbf{b}_1 | \dots | \mathbf{b}_m]$ the matrix made of m linearly independent column vectors $\mathbf{b}_i, \dots, \mathbf{b}_m$ in $\mathbb{R}^n$.
The lattice generated by $\mathbf{B}$ is the set:

$$L(\mathbf{B}) = \{\mathbf{B}x,\ x \in \mathbb{Z}^m\}.$$

In this context, $\mathbf{B}$, is called the basis of L and L the lattice generated by $\mathbf{B}$.

The rank of the lattice, $\text{rank}(L)$ is the dimension of the linear span of the basis of L, m, since we have assumed that the $\mathbf{b}_i$ vectors are linearly independent. When $\text{rank}(L) = n$ the lattice is said to be *full-rank*.

Another very common presentation is based on linearly independent row vectors. Several tools (e.g. `fplll` [2]) assume this representation. We will exclusively use it now on to keep the presentation and the programs in line.

3.4.2 From integer coefficient polynomials to lattices and back

The general idea is to use a family of polynomials built along the lines proposed in §3.3.8, p. 167. These polynomials are linearly independent. From these polynomials, we are going extract linearly independent vectors that can form the basis of a lattice:

1. we set an order on the monomials found in the polynomials;
2. from each polynomial we derive a vector whose elements are the coefficients;
3. more precisely, to build a vector, we pick the coefficients of the monomials in the polynomial, in the order set above;
4. by stacking the obtained row vectors, we can create a matrix representing a basis of some lattice L;
5. this basis is then LLL-reduced (see the proper definition coming soon below);
6. we use the shortest-norm (hopefully satisfying Coppersmith's condition) vectors pair of the LLL-reduced basis to recreate polynomials (reversing the above process) and compute the roots (by techniques exposed below) of this polynomial equations system;
7. if the set of the roots is not empty, check these against our initial polynomial equation ($Q(i,t) = 0 \bmod \mathcal{N}$), since if the latter equation has roots they will necessarily show up in the former set.

This informal sketch still leaves many unresolved issues that we will tackle in the upcoming sections. We will try first to clarify what an LLL-reduced basis is. For that purpose the first needed concept is that of Gram-Schmidt orthogonalization.

3.4.3 Gram-Schmidt orthogonalization

Orthogonality of a lattice basis is measured by comparing it with that of the matching orthogonal basis.

Let us first remind here the Gram-Schmidt[22] orthogonalization process that, starting from a basis of a subspace of $\mathbb{R}^n$, computes an *orthogonal basis* with the following properties.

Input: a family $(b_1, \ldots, b_m)$ of linearly independent vectors of $\mathbb{R}^n$, with $n \geqslant m$

Output: a family $(b_1^*, \ldots, b_m^*)$ of linearly independent vectors of $\mathbb{R}^n$ such that

- $(b_1^*, \ldots, b_m^*)$ is orthogonal;
- $\text{Vect}(b_1^*, \ldots, b_k^*) = \text{Vect}(b_1, \ldots, b_k)$ for all k, $1 \leqslant k \leqslant m$.

We first set $b_1^* = b_1$. Next, each b_k^* is computed by projecting b_k orthogonally along the subspace $\text{Vect}(b_1, \ldots, b_{k-1})$.

We search for b_k^* as $b_k - \sum_{i=1}^{k-1} \mu_{k,i} b_i^*$; as $\langle b_k^*, b_i^* \rangle$ has to be 0, we deduce $\mu_{k,i} = \langle b_k, b_i^* \rangle / \|b_i^*\|^2$.

Notice that for all $i = 1, \ldots, m$, $||b_i^*|| \leqslant ||b_i||$.

See [108] for a detailed description of Gram-Schmidt orthogonalization.

Notice that if the vector family $(b_1, \ldots, b_m)$ defines a lattice, there is no reason why the family $(b_1^*, \ldots, b_m^*)$ should define the same one.

3.4.4 Determinant of a lattice

Let now L be the lattice generated by a family $(b_1, \ldots, b_m)$ of linearly independent vectors of $\mathbb{R}^n$, with $m \leq n$.

Let us consider the matrix M formed by the m row vectors, as explained above, and define the volume of lattice L as $\text{vol}(L) = \sqrt{\det(M \times M^T)}$

The quantity $\det(M \times M^T)$ is often called the *Gram determinant* of M or its *Gramian* (see [35] p. 246–256).

This definition of the volume is less common than the more usual $\text{vol}(L) = |\det(M)|$ that only applies to square matrices. But since M needs not be square (when $m \neq n$), we rather consider the

[22] After Jørgen Pedersen Gram and Erhard Schmidt.

square root of the Gramian of the family $(b_1, \ldots, b_m)$.

When the matrix M is square, of course, both definitions of the volume coincide, we have

$$\det M = \pm\sqrt{\det(M \times M^T)}.$$

Theorem 3.4. *If $(b_1^*, \ldots, b_m^*)$ is the result of the Gram-Schmidt orthogonalization of $(b_1, \ldots, b_m)$ and L is the lattice generated by $(b_1, \ldots, b_m)$, then vol$(L) = \prod_{i=1}^{m} ||b_i^*||$.*

Proof. Since the vectors of the $(b_1^*, \ldots, b_m^*)$ family are mutually orthogonal and if we consider the M^* matrix formed by these row vectors, we have:

$$M^* \times M^{*T} = \begin{bmatrix} ||b_1^*||^2 & 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ 0 & ||b_2^*||^2 & 0 & \ldots & \ldots & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \ldots & 0 & ||b_i^*||^2 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \ldots & \ldots & \ldots & 0 & ||b_{m-1}^*||^2 & 0 \\ 0 & \ldots & \ldots & \ldots & \ldots & 0 & ||b_m^*||^2 \end{bmatrix},$$

since $\langle b_i^*, b_j^* \rangle = 0$ for $i \neq j$ and $||b_i^*|| = \sqrt{\langle b_i^*, b_i^* \rangle}$.

As $M^* \times M^{*\intercal}$ is a square diagonal matrix: $\det(M^* \times M^{*\intercal}) = \prod_{i=1}^{m} ||b_i^*||^2$ (the product of the diagonal elements).

We also have, from the description of the GSO,

$$\begin{bmatrix} \\ M^* \\ \\ \end{bmatrix} = \begin{bmatrix} 1 & * & \ldots & * & * \\ 0 & 1 & \ldots & * & * \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & * \\ 0 & 0 & \ldots & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \\ M \\ \\ \end{bmatrix},$$

where we have an upper triangular matrix formed with the previously defined $\mu_{i,j}$ in the orthogonalization process (denoted here by "$*$" as we do not care about their actual value) and whose diagonal elements are all 1s.

When is comes to the transposed matrices, we also have

$$\begin{bmatrix} \\ M^{*\intercal} \\ \\ \end{bmatrix} = \begin{bmatrix} \\ M^{\intercal} \\ \\ \end{bmatrix} \times \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ * & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ * & * & \dots & 1 & 0 \\ * & * & \dots & * & 1 \end{bmatrix}.$$

We can form the $M\,M^{\intercal}$ product as

$$\begin{bmatrix} \\ M^{*} \\ \\ \end{bmatrix} \times \begin{bmatrix} \\ M^{*\intercal} \\ \\ \end{bmatrix} =$$

$$\begin{bmatrix} 1 & * & \dots & * & * \\ 0 & 1 & \dots & * & * \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & * \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \\ M \\ \\ \end{bmatrix} \times \begin{bmatrix} \\ M^{\intercal} \\ \\ \end{bmatrix} \times \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ * & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ * & * & \dots & 1 & 0 \\ * & * & \dots & * & 1 \end{bmatrix}.$$

As the determinant of our upper and lower triangular matrices is 1, we can deduce

$$\mathrm{vol}(L) = \sqrt{\det(M^{*} \times M^{*\intercal})} = \prod_{i=1}^{m} ||b_i^{*}||.$$

□

3.4.5 *LLL*-algorithm

The *LLL*-algorithm, created by A. Lenstra, H. Lenstra and L. Lovász in 1982, performs an operation called *lattice basis reduction*. Given a basis $(v_1, \dots, v_m)$ of L, the *LLL*-algorithm[65] computes in polynomial time a new basis of L, $(b_1, \dots, b_m)$, (called the *LLL-reduced basis*), from which we can derive the the orthogonal $(b_1^{*}, \dots, b_m^{*})$ basis by GSO, with the following properties:

- $\forall k$, if $b_k = b_k^{*} + \sum_{\ell=1}^{k-1} \mu_{k,\ell}\, b_\ell^{*}$, then $|\mu_{k,\ell}| \leq \frac{1}{2}, \forall \ell \in [\![1, k-1]\!]$;
- $||b_k^{*} + \mu_{k,k-1} b_{k-1}^{*}||^2 \geqslant \frac{3}{4}||b_{k-1}^{*}||^2$, $\forall\, k \in \{2..m\}$.

The constant $\frac{3}{4}$ above is somewhat arbitrary and is a parameter of the algorithm that can be replaced by any fixed real number δ such that $\frac{1}{4} < \delta < 1$. The 1 strict upper bound must be honored

to guarantee polynomial time.

Notice that the second property applies to the elements of the set of vectors[23] produced by the Gram-Schmidt orthogonalization of the LLL-reduced basis and not to the reduced basis itself.

This property tells us more than one could think at first sight about the reduced basis itself. If we combine both properties, we obtain $||b_k^*|| \geqslant \frac{||b_{k-1}^*||}{2}$. This indicates that the norm of the Gram-Schmidt orthogonal vectors does not decrease too quickly. Intuitively this means that the vectors of the reduced basis are near-orthogonal or at least not too far from being orthogonal. Indeed, it is when vectors of the basis are the closest to be collinear that the sharpest reductions of norm are seen along the Gram-Schmidt orthogonalization process. And this near-orthogonality property has also the interesting consequence that the first (and often the first follow-up) vector(s) is(are) have relatively short norms.

Another theorem, derived from the original LLL article[65] by Alexander May (the proof can be found in his PhD thesis[70]), puts more emphasis on LLL-reduced basis vectors ordering and on the individual upper bound of their norm.

Theorem 3.5. *Let L be a lattice of dimension m. In polynomial time, the LLL-algorithm outputs a reduced basis $(b_1, \ldots, b_m)$ such that:*

$$\|b_1\| \leqslant \|b_2\| \leqslant \cdots \leqslant \|b_i\| \leqslant 2^{\frac{m(m-1)}{4(m+1-i)}} \det(L)^{\frac{1}{m+1-i}}.$$

We will not enter into much details about the LLL-algorithm as we will mainly use as a black box.

However, the original LLL-algorithm is mainly of theoretical interest, since, although it has polynomial time complexity, it remains quite slow in practice. Instead, floating-point variants are used, where the underlying Gram-Schmidt orthogonalization is performed with floating-point numbers rather than with rational arithmetic (which produces huge numerators and denominators).

In the floating-point versions of the algorithm, another parameter η is often introduced. We recall here the following theorem under a formulation authored by D. Stehlé in [101] that rephrases in the floating-point context the one found in the *editio princeps*[65] of the Lenstra, Lenstra and Lovász article.

Theorem 3.6. *Let $\delta \in (1/4, 1]$ and $\eta \in [1/2, \sqrt{\delta})$. Let $(b_1, \ldots, b_m)$ be a (δ, η)-LLL-reduced basis*

[23] We prefer not to speak of a basis here since the set of vectors produced by the orthogonalization process is not basis of the L lattice.

of the L lattice. Then:

$$\|b_1\| \leqslant \left(\frac{2}{\delta - \eta^2}\right)^{\frac{m-1}{4}} \text{vol}(L)^{\frac{1}{m}}.$$

$$\prod_{k=1}^{m} \|b_k\| \leqslant \left(\frac{2}{\delta - \eta^2}\right)^{\frac{m(m-1)}{4}} \text{vol}(L).$$

The value of δ is as above ($0.25 < \delta < 1.0$) and that of η is $0.5 \leq \eta < \sqrt{\delta}$. In practice, we use $\delta = 0.99$ and $\eta = 0.501$.

All these elements should convince ourselves that the norms of the first vectors of an LLL-reduced basis of a lattice are shorter than those from a random basis.

It may be useful to remind here that the vectors of the reduced basis represent polynomials (according to the mapping exposed in §3.4.2, p. 169), each element of a vector being a coefficient for the monomial at the matching index in the corresponding polynomial. Hence, computing the ℓ^2-norm of the vectors is equivalent to computing the norm of the polynomials as defined for Theorem 3.3.

And this is where we eventually wanted to get: have vectors a small enough norm so that they can satisfy Condition 2 of Theorem 3.3?

Hence, after reduction, we will filter out the vectors of the basis whose norm is too large (thanks to the nice norm ordering property of the reduced basis, it suffices to stop as soon as we get a vector that does not match the norm condition) and transform vectors back to polynomials. This involves getting rid of I and T, the bounds for i and t, respectively, taking into account the exponent according to which monomial we are dealing with.

3.5 Back to the general algorithm

We had to make this rather long detour through the LLL-algorithm and its application to our problem because it is a key element in the overall strategy. We wanted to make its contribution clear and show how it fits in the whole process. It is time now to return to the main algorithm, and make use of the results obtained through LLL-reduction.

3.5.1 Chasing the roots of the polynomials

Let us call $Q_1(i,t), \ldots Q_n(i,t)$ the polynomials reconstructed from the vectors of the LLL-reduced basis that comply with Condition 2 of Theorem 3.3. We know for sure now all the roots (i_0, t_0) such

that $|i_0| \leqslant I$ and $|t_0| \leqslant T$ of $Q(i,t)$ are also roots of any of the Q_k's, with $k \in \{1,\ldots,n\}$.

At this point, a possible technique would be to search for the integral roots of any of these Q_k's and check them against the I and T bounds and, if they successfully take the test, eventually check them against the $Q(i,t) \mod \mathcal{N}$ expression to verify if they are also modular roots of this polynomial.

But finding the zero integer set of a single bivariate polynomial (which forms an underdetermined system) is by no means easy. The problem is indeed geometrically ill-posed as, generically, a single polynomial defines a curve.

Fortunately this is not necessary as we do not need all the integral roots of any of the Q_k's by itself. Since the roots of polynomial $Q(i,t)$ are common to all the Q_k's, it suffices to find the roots common to a pair of the latter to get all we want (and possibly more: a superset) for the former. In contrast with our single polynomial roots search, a system of two bivariate polynomial equations is, since it defines the intersection of two curves, often a finite set of points, a well-posed problem. By "often", we mean as soon as the two polynomials forming the system are coprime. Unfortunately, the technique used to generate them only guarantees linear independence over $\mathbb{Z}$ but not over $\mathbb{Z}[x,y]$.

As a consequence, we can not just pick any two random Q_k's and go for their common roots.

The fact that pairs of polynomials returned after basis reduction and norm checking may still have a non-trivial GCD is the reason why the algorithm we use for root finding of modular polynomial equations is dubbed as "heuristic" rather than proved. This property (or lack thereof) is often mentioned in literature because, in contrast, Coppersmith's algorithm for univariate polynomials is proved.

In our context, the fact that, at some step, we use a heuristic algorithm does not imply that our own algorithm, considered as as a whole, can itself be called "heuristic". For our application, in case we cannot get two polynomials with a non-trivial GCD, we always have the option to split the interval we are currently working on. The number of polynomials matching Coppersmith's condition will generally be larger on this reduced interval (essentially because of a smaller value for constant I), increasing therefore the odds to find among them a pair that comply with the coprime condition.

Should splitting continue all the way down to a single floating-point number, direct checking of this value for hardness-to-round would solve the problem. Of course, we do not want this to happen

often, if at all: it would turn down into an overwhelmingly expensive way to perform exhaustive search! In practice, it does not and splitting to smaller but still reasonably large intervals always saves the day.

Nevertheless, the complexity of the method is heuristic, since all the analysis rests on the fact that the first two first polynomials are coprime.

3.5.1.1 Root finding with the resultant method

We will not recall resultant theory at large which is a vast and intricate domain. A more detailed approach and the proofs of several lemmas used here can be found in von zur Gathen and Gerhard's work [108] from which we borrow and specialize the following results.

Theorem 3.7. *(Gauss) If R is a unique factorization domain (UFD), then R[x] is a UFD.*

A proof of the theorem can be found in the above cited reference. As $\mathbb{Z}$ is a UFD and after Theorem 3.7, so is $\mathbb{Z}[x]$. In our problem, we have bivariate polynomials in $\mathbb{Z}[x, y]$. They can be viewed as polynomials in $\mathbb{Z}[x][y]$ that is to say polynomials in y with coefficients in $\mathbb{Z}[x]$. They also be viewed as polynomials in x with coefficients in $\mathbb{Z}[y]$.

Nevertheless, and by same token, $\mathbb{Z}[x, y]$ is a UFD.

From the coefficients of two polynomials p and q of $\mathbb{Z}[x, y]$, one can build their Sylvester matrix $Syl(p, q)$. It can be constructed in different ways, according to our view of the polynomials as elements of $\mathbb{Z}[x][y]$ or of $\mathbb{Z}[y][x]$.

In the first case the elements of the Sylvester matrix are polynomials in x. In the second case they are polynomials in y.

We will focus now on the first case, for reasons that will become clear later on. Nevertheless what comes next also applies to the second case.

One can compute the determinant of the Sylvester matrix, that, in our first case, is called the *resultant of p and q with respect to y*, $\mathrm{res}_y(p(x, y), q(x, y))$. It is also a polynomial in x.

We borrow again[24] this second result from [108].

[24] With a minor notation modification in the name of the indeterminate.

Theorem 3.8. *Let R be a UFD and $p, q \in R[y]$ not both zero. Then* $\gcd(p, q)$ *is nonconstant in* $R[y]$ *if and only if* $\operatorname{res}(p, q) = 0$ *in* R.

The proof of this theorem can also be found in the above cited reference.

We can use it to our advantage if we identify R with $\mathbb{Z}[x]$ and $\operatorname{res}(p, q)$ with $\operatorname{res}_y(p, q)$.
This result has two important consequences:

- it provides us with a criterion to check if two polynomials are coprime (their resultant is nonzero);
- when the resultant is nonzero, we can search for its roots in x (remember the resultant is a polynomial in x); if any, those form a superset of the roots p and q have in common.

At this point let us come back to our original problem and recover our initial notation. Here i (resp. t) corresponds to x (resp. y). From now on, we will use the former.

As a reminder, we were looking for roots of polynomial $Q(i, t) = 0 \bmod \mathcal{N}$. Those, if any, are common to Coppersmith's condition compliant $(Q_j(i, t), Q_k(i, t))$ pairs, if any, obtained after the LLL-reduction. This way, we can find all the roots for $Q(i, t) = 0 \bmod \mathcal{N}$ that are bounded as follows: $|i| < I$ and $|t| < T$.

To summarize, once $\operatorname{res}_t(Q_j(i, t), Q_k(i, t))$ is computed[25], we have four options:

- $\operatorname{res}_t(Q_j(i, t), Q_k(i, t)) = 0$: we can say nothing about any common roots; as we have seen Q_j and Q_k have at least a common factor; for our problem this implies a complete new search from the start on a smaller interval;
- $\operatorname{res}_t(Q_j(i, t), Q_k(i, t))$ is a nonzero constant: there is no root common to $Q_j(i, t)$ and $Q_k(i, t)$; in our case it follows that no hard to round case exists in the considered interval;
- in the remaining case $\operatorname{res}_t(Q_j(i, t), Q_k(i, t))$ has a degree in i larger or equal to one, we try to compute its integral roots; according to the result we have two sub-cases:
  - no integral root is found; again, there is no integral root common to $Q_j(i, t)$ and $Q_k(i, t)$, no hard-to-round case exists,
  - the number of roots found is nonzero, value checking is now necessary.

As stated above, root checking is only necessary if, for two polynomials $Q_j(i, t)$ and $Q_k(i, t)$ with $j \neq k$, polynomial $\operatorname{res}_t(Q_j(i, t), Q_k(i, t)) \in \mathbb{Z}[i]$ has integral roots. If there are such $i_r \in \mathbb{Z}$, $r \in \mathbb{N}$,

[25] When $\operatorname{res}_t(Q_j(i, t), Q_k(i, t)) \neq 0$ we also have the following inclusions chain:
$\operatorname{roots}_{(|x_0|<I)}(Q(i, t)) \subseteq \operatorname{roots}_{x_0}(Q_j(i, t), Q_k(i, t)) \subseteq \operatorname{roots}_{x_0}(\operatorname{res}_t(Q_j(i, t), Q_k(i, t)))$.

the next logical step could be, for each root i_r, to compute the corresponding value(s) $t_{r,s} \in \mathbb{Z}$, $s \in \mathbb{N}$ to form as many as possible $(i_r, t_{r,s})$ root pairs common to $Q_j(i,t)$ and $Q_k(i,t)$.

There are several ways to do it but we must notice here that it is pointless. Indeed, computing values for t involves more or less expensive resultant computations and/or root finding searches. These roots should also be checked for bounds conformity ($|i_r| <$I and $|t_{r,s}| < T$, which, admittedly is computationally cheap) and against $Q(i,t)$ and $\mathcal{N}$) ($Q(i_r, t_{r,s}) = 0 \bmod \mathcal{N}$) for validity (remember we have possibly computed a superset of the roots of $Q(i,t)$).

We can bypass all of these root computations and checks to go straight ahead towards what really matters to us: do these values i_r correspond to hard-to-round cases? Of course, we fork here from the classical descriptions found in cryptography-related literature because our target is different. In fact, we do not care one iota about the $t_{r,s}$ values.

We can do then straightforwardly:

- scale back i_r to the corresponding floating-point value X_r in the $g(x)$ function domain;
- compute $g(X_r)$ with enough accuracy (e.g. using an arbitrary precision package like MPFR);
- check whether the obtained image is a hard-to-round case.

3.5.2 Producing the initial basis

We deal, at the bottom of this section, with a question that chronologically should rest on the top. If we have commented at length on how to manipulate (essentially, reduce) our initial basis and what to do with the output of these transformations (root search and check), we have not said much about how we build it in the first place. It is about time to fill this gap.

We will start with a small reminder on what the initial basis is made of. We noticed that if (i_0, t_0) is a solution of $Q(i,t) = 0 \mod \mathcal{N}$, we can build new polynomial modular equations based on $Q(i,t)$, i, t, and $\mathcal{N}$ that also have (i_0, t_0) among their own roots (if any).

They could have, for instance, the following general form:

$$i^r\, t^s\, Q(i,t)^u\, \mathcal{N}^v = 0 \mod \mathcal{N}^{u+v}, \text{ with: } r, s, u, v \in \mathbb{N}.$$

We have two groups of exponents here:

- u and v that set the modulus of the modular equations: $\mathcal{N}^{u+v}$;

- r and s whose different values allow for the creation of new polynomials but have no effect on the modulus.

If we are to build a consistent basis, the polynomials we use should all be related to the same modulus in modular equations. As we will see later, the exponent of $\mathcal{N}$ in the modulus, as well as d (the degree in i of polynomial $Q(i,t)$), is one of the key parameters of the method. Let us call that exponent α.

Our previous general form goes now as:

$$i^r\, t^s\, Q(i,t)^u\, \mathcal{N}^{\alpha-u} = 0 \mod \mathcal{N}^\alpha, \text{ with: } r,s,u,\alpha \in \mathbb{N} \text{ and } u \leqslant \alpha.$$

Following D. Boneh and G. Durfee in [8] but using our own notations in the sake of presentation consistency, we will create two new kinds of polynomials:

- $is_{r,u}(i,t) = i^r\, Q(i,t)^u\, \mathcal{N}^{\alpha-u}$, called *i-shifts* by D. Boneh and G. Durfee;
- $ts_{s,u}(i,t) = t^s\, Q(i,t)^u\, \mathcal{N}^{\alpha-u}$, that are called *t-shifts*;

Remark
One may notice that there are none of what we could call mixed "*it*-shifts" where both i and t show up with a nonzero exponent. In fact, those are not always necessary when, depending on the form of $Q(i,t)$, any polynomial of the $i^r\, t^s\, Q(i,t)^u\, \mathcal{N}^{\alpha-u}$ form can be obtained as linear combination of i-shifts and t-shifts.

What we are after are small-norm (remember Theorem 3.3) integral linear combinations of polynomials $is_{r,u}(iI,tT)$ and $ts_{s,u}(iI,tT)$. We recall here that I (resp. T) is an upper bound on $|i|$ (resp. $|t|$) mentioned in the latter theorem. To get them, we form a lattice spanned by the corresponding coefficients vectors that has sufficiently small vectors and we use the LLL-algorithm to find those small vectors.

As can be seen in, for instance, Theorem 3.6, their norm is connected to the volume (or the determinant) of the lattice. Hence we want to start, ideally, with the smallest possible volume since this property is conserved by a change of basis. Starting with a large volume lattice hampers the search of very short-norm vectors. This should incline us to limit the number of polynomials.

On the other hand we must provide the LLL-algorithm with a sufficient number of vectors to shuffle in order to compute efficiently short vectors This may require to append more polynomials

at the expense of a volume expansion.

The balance is not easy to set and, as will be seen, there are no clear rules on how it can be done.

When α and d are set, all that is left to set, before starting the generation of the polynomials list, are the rules for r and s. This is the point we are going to examine now.

3.5.2.1 When making it simple is not enough

The question of the initial basis production is present in the very early papers that describe the above sketched method for polynomial modular equations root finding. They start with construction rules that allow for a basis structure that facilitates the computations used in proofs.

But it soon emerges that these bases are not optimal. New construction rules are needed that produce better bases, with the consequence that proofs get more involved. We describe here two cases that especially matter to us:

- D. Bonneh and G. Durfee's approach, as they laid the foundations of the method;
- the SLZ-algorithm approach, as it is, after all, the algorithm we implement here.

.

3.5.2.2 Boneh and Durfee's approach

In their first paper [8], where the extension of Coppersmith's method is performed for the bivariate case, D. Boneh and G. Durfee offer two different presentations of their initial basis. One, with a rather simple layout, is used to present the problem and prove preliminary results. A second one, much more convoluted, is used for improved results.

If we rephrase their description in our own notations, they initially propose the following rules:

- for all $is_{r,u}(iI, tT)$ and $ts_{s,u}(iI, tT)$, u is such that $0 \leqslant u \leqslant \alpha$;
- for each u we build $is_{r,u}(iI, Tt)$, r is such that $0 \leqslant r \leqslant \alpha - u$;
- for each u we build $ts_{s,u}(iI, tT)$, s is such that $0 \leqslant r \leqslant S$, for some S determined later;

For $\alpha = 3$ and $S = 1$ the result is listed in Table 3.1.

We build the lattice matrix, from the coefficients of the polynomials as follows:

- each column corresponds to a monomial as it shows up in the polynomials;

$\mathcal{N}^3$	$iI\,Q(iI,tT)\,\mathcal{N}^2$	$iI\,Q(iI,tT)^2\,\mathcal{N}$	$tT\,Q(iI,tT)^2\,\mathcal{N}$
$iI\,\mathcal{N}^3$	$Q(iI,tT)^2\,\mathcal{N}$	$Q(iI,tT)^3$	$tT\,Q(iI,tT)^3$
$Q(iI,tT)\,\mathcal{N}^2$	$(iI)^3\,\mathcal{N}^3$	$tT\,\mathcal{N}^3$	
$(iI)^2\,\mathcal{N}^3$	$(iI)^2\,Q(iI,tT)\,\mathcal{N}^2$	$tT\,Q(iI,tT)\,\mathcal{N}^2$	

Table 3.1: List of polynomials obtained with D. Boneh and G. Durfee's first set of rules for $\alpha = 3$ and $S = 1$.

- for each polynomial, a matrix row is formed with its coefficients each placed in the column corresponding to the monomial it refers to.

We give an example in Table 3.2, with the same polynomials as those listed in Table 3.1 and with an initial polynomial in the $Q(i,t) = ai + it - 1$ form, as it is in D. Boneh and G. Durfee's paper. Here $Q(iI,tT)$ is abbreviated to Q, iI to i and tT to t in order to save space.

One nice feature of this polynomials set is that, when their coefficients are used to build the lattice matrix, the latter is lower triangular.

	1	i	it	i^2	i^2t	i^2t^2	i^3	i^3t	i^3t^2	i^3t^3	t	it^2	i^2t^3	i^3t^4
$\mathcal{N}^3$	$\mathcal{N}^3$	0	0	0	0	0	0	0	0	0	0	0	0	0
$i\,\mathcal{N}^3$	-	$\mathcal{N}^3 I$	0	0	0	0	0	0	0	0	0	0	0	0
$Q\,\mathcal{N}^2$	-	-	$\mathcal{N}^2 IT$	0	0	0	0	0	0	0	0	0	0	0
$i^2\,\mathcal{N}^3$	-	-	-	$\mathcal{N}^3 I^2$	0	0	0	0	0	0	0	0	0	0
$i\,Q\,\mathcal{N}^2$	-	-	-	-	$\mathcal{N}^2 I^2 T$	0	0	0	0	0	0	0	0	0
$Q^2\,\mathcal{N}$	-	-	-	-	-	$\mathcal{N} I^2 T^2$	0	0	0	0	0	0	0	0
$i^3\,\mathcal{N}^3$	-	-	-	-	-	-	$\mathcal{N}^3 I^3$	0	0	0	0	0	0	0
$i^2\,Q\,\mathcal{N}^2$	-	-	-	-	-	-	-	$\mathcal{N}^2 I^3 T$	0	0	0	0	0	0
$i\,Q^2\,\mathcal{N}$	-	-	-	-	-	-	-	-	$\mathcal{N} I^3 T^2$	0	0	0	0	0
Q^3	-	-	-	-	-	-	-	-	-	$I^3 T^3$	0	0	0	0
$t\,\mathcal{N}^3$	-	-	-	-	-	-	-	-	-	-	$\mathcal{N}^3 T$	0	0	0
$t\,Q\,\mathcal{N}^2$	-	-	-	-	-	-	-	-	-	-	-	$\mathcal{N}^2 IT^2$	0	0
$t\,Q^2\,\mathcal{N}$	-	-	-	-	-	-	-	-	-	-	-	-	$\mathcal{N} I^2 T^3$	0
$t\,Q^3$	-	-	-	-	-	-	-	-	-	-	-	-	-	$I^3 T^4$

Table 3.2: Matrix generated by polynomials of Table 3.1. Going from one polynomial to the next introduces only one new monomial. Hence the matrix is square and lower triangular. Computing its determinant is very easy: all we need to care about are the elements on the main diagonal.

The authors of [8] carry on a first analysis with this kind of basis to conclude that they achieve some level of effectiveness to solve their cryptography problem[26] Their target is to get the largest possible upper bound on one of the parameters of the problem, denoted by δ. With this first approach they achieve $\delta < 0.284$. But, at the same time, they notice two important facts :

- they will not be able to go any further with this kind of lattice: once $\delta > 0.284$, the determinant of the matrix is always too large;

[26] That is not relevant for us to discuss in detail here.

- their idea is to get rid of some of the rows whose element on the diagonal is an heavy contributor to determinant growth.

They go into some details in the technique they use for rows selection. In a nutshell, they add extra t-shifts (the value they come with for S is larger than 1, as used to build the Table 3.2) and they eliminate those that yield a line vector that has an element on the diagonal larger that some threshold.

Of course, this breaks the comfortable squareness of the matrix and they devote a lot of time to the exposition of sophisticated techniques to bound (since it cannot be exactly computed anymore) the volume of the lattice spanned by their new rectangular matrix. But they eventually succeed in improving on their initial bound on δ to 0.292, which is more of an achievement than the small numerical difference might lead to think.

No extra details will be given here as they are not relevant to our problem. The reader interested in an in-depth study is invited to refer to [8].

3.5.2.3 SLZ and D. Stehlé own's approach

We discuss here the solutions proposed for initial basis construction found in the 2005 original paper of D. Stehlé, V. Lefèvre and P. Zimmermann [103] and a later evolution, proposed by D. Stehlé all by himself [100], in 2006.

In their first work, the authors consider only what we could call, in D. Boneh and G. Durfy's parlance reworked in our own notations, i-shifts. This is due, in part, to the specific shape of their initial polynomial. In this case:

$$Q(i,t) = P(i) + t, \text{ with } P(i) \in \mathbb{Z}[i], \text{ of degree } d \text{ in } i.$$

The considered i-shifts are those, for some constant $\alpha \in \mathbb{N}$ and d, the degree in i of $Q(i,t)$, such that

$$is_{r,u}(i,t) = i^r Q(i,t)^u \mathcal{N}^{\alpha-u}, \text{ where } 0 \leqslant r + du \leqslant d\alpha.$$

With this setting, they obtain a square matrix of dimension $\dfrac{(\alpha+1)(d\alpha+2)}{2}$. Table 3.3 displays the matrix $M_{12,12}$ obtained for $d = 3$ and $\alpha = 2$.

Later on, D. Stehlé found out that some of the polynomials (hence vectors in the lattice basis) did not add much information and that they cluttered the operation of the LLL-algorithm.

	1	i	i^2	i^3	i^4	i^5	i^6	t	it	i^2t	i^3t	t^2
$\mathcal{N}^2$	$\mathcal{N}^2$	0	0	0	0	0	0	0	0	0	0	0
$i\mathcal{N}^2$	-	$\mathcal{N}^2I$	0	0	0	0	0	0	0	0	0	0
$i^2\mathcal{N}^2$	-	-	$\mathcal{N}^2I^2$	0	0	0	0	0	0	0	0	0
$i^3\mathcal{N}^2$	-	-	-	$\mathcal{N}^2I^3$	0	0	0	0	0	0	0	0
$i^4\mathcal{N}^2$	-	-	-	-	$\mathcal{N}^2I^4$	0	0	0	0	0	0	0
$i^5\mathcal{N}^2$	-	-	-	-	-	$\mathcal{N}^2I^5$	0	0	0	0	0	0
$i^6\mathcal{N}^2$	-	-	-	-	-	-	$\mathcal{N}^2I^6$	0	0	0	0	0
$Q\mathcal{N}$	-	-	-	-	-	-	-	$\mathcal{N}T$	0	0	0	0
$iQ\mathcal{N}$	-	-	-	-	-	-	-	-	$\mathcal{N}IT$	0	0	0
$i^2Q\mathcal{N}$	-	-	-	-	-	-	-	-	-	$\mathcal{N}I^2T$	0	0
$i^3Q\mathcal{N}$	-	-	-	-	-	-	-	-	-	-	$\mathcal{N}I^3T$	0
Q^2	-	-	-	-	-	-	-	-	-	-	-	T^2

Table 3.3: Matrix generated from polynomials built according to the SLZ given rules. Going from one polynomial to the next introduces only one new monomial. Once again, the matrix is square and lower triangular. Computing its determinant is very easy: all we need to care about are the elements on the main diagonal.

Retaining the original monomials list of length $\frac{(\alpha+1)(d\alpha+2)}{2}$, he proposed in [100] a different polynomial construction rule. The considered i-shifts are now those, for some constant $\alpha \in \mathbb{N}$[27] such that

$$is_{r,u}(i,t) = i^r Q(i,t)^u \mathcal{N}^{\alpha-u}, \text{ where } 0 \leqslant r + u \leqslant \alpha.$$

This rule yields $\frac{(\alpha+1)(\alpha+2)}{2}$ polynomials.

Table 3.4 displays the matrix obtained with the same parameters as for Table 3.3: $\alpha = 2$ and the degree of $Q(i,j)$ in i is 3, to make a comparison possible with the previous building rule.

	1	i	i^2	i^3	t	i^4	it	i^5	i^6	i^2t	i^3t	t^2
$\mathcal{N}^2$	$\mathcal{N}^2$	0	0	0	0	0	0	0	0	0	0	0
$i\mathcal{N}^2$	-	$\mathcal{N}^2I$	0	0	0	0	0	0	0	0	0	0
$i^2\mathcal{N}^2$	-	-	$\mathcal{N}^2I^2$	0	0	0	0	0	0	0	0	0
$Q\mathcal{N}$	-	-	-	$\mathcal{N}I^3$	$\mathcal{N}T$	0	0	0	0	0	0	0
$iQ\mathcal{N}$	-	-	-	-	-	$\mathcal{N}I^4$	$\mathcal{N}IT$	0	0	0	0	0
Q^2	-	-	-	-	-	-	-	I^5	I^6	I^2T	I^3T	T^2

Table 3.4: Matrix generated by polynomials build according to D. Stehlé modified rules. The matrix is not square any more and a different technique is used to get, at least, an upper bound on the lattice determinant.

The reasons that explain this modified basis building rule are similar in principle, if technically different, to those used by D. Boneh and G. Durfee. The details can be found in [100] as Theorem 2; its proof is in the Appendices of the article. At some point D. Stehlé remarks that his polynomial/basis building rule does not allow, when applied to D. Boneh and G. Durfee's problem, for the

[27] Notice that d, the degree of $Q(i,t)$ in i, does not show up anymore in the formula.

same effectiveness. In fact, building rules are very problem-dependent. Up to the date there is no general model that, for a given problem, allows one to derive the optimal initial basis building rules.

As for the implementation presented here, we will stick to the rule presented by D. Stehlé in [100].

3.5.3 Still journeying: from an interval to the next

Solving the Table Maker's Dilemma for a function and some $\mathbb{F}$ requires the exploration of all the fbinades of $\mathbb{F}$. As it has been seen, the technique used here does not allow to make it in one shot. Once the "special cases" removed (elements of the domain that map to non numeric values, those for which the function is not defined, etc.) what is left is split into small sub-domains. Dealing with those may lead to extra splitting since the technique used here heavily relies on polynomial approximation. For our hardness-to-round investigations this requires high accuracy approximations that can only be obtained for relatively narrow intervals if the degree of the polynomial is to be kept at a reasonable level.

3.5.3.1 The general case

Running the algorithm is another source of splitting. Even if the width of an interval is in line with what the accuracy of the approximation polynomial allows for, problems may surface after the LLL-reduction. The number of short-norm vectors may be insufficient or the polynomials derived from those may share a non trivial common factor. This is often overlooked in literature as an uncommon situation never happening in practice. Our own experience differs on that: it does happen. The only remedy to this situation is, again, extra splitting.

This kind of failure is very expensive because, as will be seen, reductions are time consuming and performing them in vain is hardly acceptable. But, as we have noted, for a given interval, it is difficult to come up with a "right" set of parameters. It does not only take thinking but also tedious, frustrating and time-consuming testing.

For a given function, precision and hardness-to-round, one would like to get the best possible return of investment on one's expensively gained parameters for a given interval. Ideally, they should apply equally well wherever another tested interval of the same relative width is located on the function domain.

At this point, the question is: is such a requirement only wishful thinking or is there any sound foundation to support it?

One problem comes from function approximation. After higher-level domain slicing into subdomains where the function is monotonic, these are in turn split into lower level subdomains that span, at most, one binade. Each of these is then mapped to the 0-fbinade ($[\![1, 2[\![$). More splitting is done for the images to fit into a single fbinade too. But this does not mean that the scaled functions we try to approximate are similar because they share the same domain. Scaling does not blur the differences in behavior of the function in different fbinades of its domain. This precisely what makes this scaling legitimate.

One must also take into account the following effect: equal interval lengths in the 0-fbinade, where different fbinades are mapped to, do not correspond to equal interval lengths in the original fbinades.

Let us take an example. We can make a one-to-one map between the floating-point numbers in $[\![\frac{4}{16}, \frac{5}{16}]\!]$ (a subset of the -2-fbinade) and those in $[\![1, \frac{5}{4}]\!]$ (a subset of the 0-fbinade) by a 4-scaling up.

The same is true for the floating-point numbers in $[\![\frac{4}{8}, \frac{5}{8}]\!]$ (a subset of the -1-fbinade) but this time for a scaling up of 2.

We will not study the hardness-to-round of the $\exp(x)$ function on the original sets $[\![\frac{4}{16}, \frac{5}{16}]\!]$ or $[\![\frac{4}{8}, \frac{5}{8}]\!]$ but instead will will study the hardness-to-round of the corresponding scaled functions ($\exp(\frac{x}{4})$ for the first one and $\exp(\frac{x}{2})$ for the second one) on their (same) mapping: $[\![1, \frac{5}{4}]\!]$.

In this mapping, the distance between the bounds is the same: $\frac{1}{4}$. But in the first case the actual distance "represented" by $\frac{1}{4}$ is $\frac{1}{16}$ and in the second case the distance is $\frac{1}{8}$. The only real constant here is the number of floating-point numbers that are represented.

It come as no surprise that the approximation of the scaled functions behaves differently even if it is performed over the same interval, $\left[1, \frac{5}{4}\right]$, just as much as the approximation of the $\exp(x)$ function behaves differently in $\left[\frac{4}{16}, \frac{5}{16}\right]$ and in $\left[\frac{4}{8}, \frac{5}{8}\right]$.

As a consequence, when it comes to approximation, the accuracy obtained with a polynomial of a given degree over a given interval width in the mapping may greatly vary according to the considered mapped portion of the domain.

Let us give actual data gathered in the case of the $\exp(x)$ function. When we want to make a search for a hardness-to-round of $6p$ with precision $p = 113$ (`binary128`), we need an approximation accuracy smaller than $3.99e-205$. Let us suppose that we want an approximation polynomial of degree 17 and that we want each interval to contain 2^{80} floating-point numbers, in order to explore the different binades at a steady pace. Let us examine the situation when the studied interval is astride the center of the binade [28].

Data is presented in Table 3.5.

Binade	Approx. error		Binade	Approx. error
-3	1.61e-211		3	5.38e-179
-2	5.10e-206		4	1.75e-173
-1	9.73e-201		5	3.54e-168
0	2.69e-195		6	1.10e-162
1	7.93e-190		7	4.09e-157
2	2.61e-184		8	1.07e-151

Table 3.5: Approximation error for the exp function by degree-17 in several binades, for intervals holding 2^{80} floating-point numbers.

We can see that the approximation accuracy of a degree-17 polynomial is sufficient for binades -3 and -2. We can even say that the chosen degree is exaggeratedly conservative for binade -3. But for the next binades, this accuracy is insufficient.

One can get an intuition as to the reasons for this difference by examining the plots of two extremal cases of scaled functions as presented in Figure 3-9. The first (square-dotted line, almost horizontal) is the scaled function used for binade -2. The second (ball-dotted line, canted) is used for binade 8 (notice that the plot of this function was largely offset down in order to get both plots side by side and observe them on the same figure). Functions used in other binades, as presented in Table 3.5, would plot as almost straight lines but with increasing slopes as the binade index grows.

Contrarily to what pops up at first sight, these plots are not straight lines but slightly bent curves. Intuitively, a degree-17 polynomial has a much harder time at sticking to the curvature of the scaled $\exp(x)$ function and at dealing with the (relatively) high difference in image values from the beginning to the end of the interval, all at the same time (as in ball-dotted line plot), than

[28] The exact location in the binade does not especially matter here, but we had to pick the same one for each binade to make the comparison meaningful.

dealing with the curvature alone (as in the square-dotted line plot).

This is peculiar to the $\exp(x)$ function, where the value of the derivative increases towards positive values. The situation would be different with some other function, but the problem remains the same.

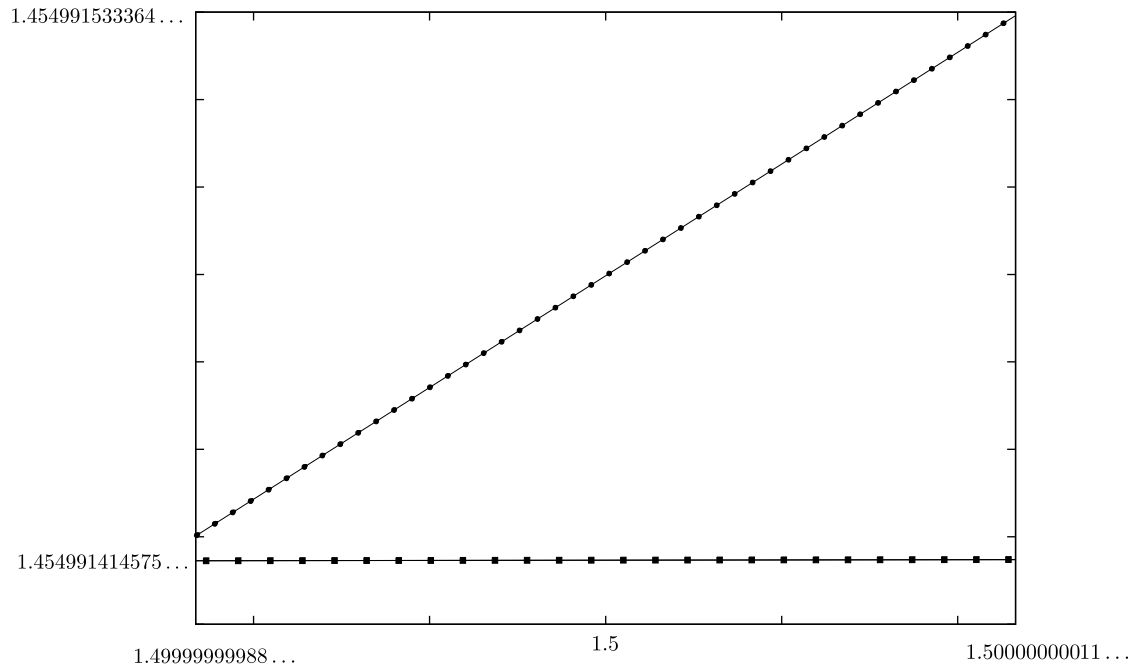

Figure 3-9: The ball-dotted line is the plot of the scaled $\exp(x)$ function in a small portion of the 8-binade. The square-dotted is the plot of the scaled exp function in an equivalent portion of the -2-binade.

At this point two options are open:

- pick a larger degree, if we really insist on having even intervals;
- have a variable (in this case, decreasing) interval width, if sticking to degree-17 is what matters most to us.

None of them is fully satisfactory. Changing degrees may involve changing the value of α if one wants a minimal reduction time (see §3.7), p. 219. It probably requires some experimental tweaking, what we do not really want. Changing the interval width is not necessarily a smart move. As will be seen, it can be more expensive to scan through several short intervals than to process a larger one that contains them all.

3.5.3.2 A special case: the $\exp(x)$ function on a single binade

We have seen that parameters have to change from one binade to another. But can we have some relief when we stay inside a single binade? There is no general answer to this question but a response can be given in the particular case of the $\exp(x)$ function.

We have said that we used Taylor polynomials of some degree d at point a, the center of the interval we want to explore for hard-to-round cases for function $f(x)$. The general form of the polynomial is:

$$p_{d,a}(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots + \frac{f^{(d)}(a)}{d!}(x-a)^d.$$

If we specialize it for the exponential function case, using the property that $\exp^{(d)}(x) = \exp(x)$ for $d \in \mathbb{N}^*$, we have:

$$p_{d,a}(x) = \exp(a) + \exp(a)(x-a) + \frac{\exp(a)}{2!}(x-a)^2 + \ldots + \frac{\exp(a)}{d!}(x-a)^d.$$

We can rewrite this expression as

$$p_{d,a}(x) = \exp(a)\left(1 + (x-a) + \frac{(x-a)^2}{2!} + \ldots + \frac{(x-a)^d}{d!}\right).$$

The Taylor polynomial at some other point b such that $b > a$ (what comes next is also true in the trivial situation where $b = a$ but is of no interest here).

$$p_{d,b}(x) = \exp(b)\left(1 + (x-b) + \frac{(x-b)^2}{2!} + \ldots + \frac{(x-b)^d}{d!}\right).$$

In both polynomials, the expressions $(x-a)$ and $(x-b)$ actually mean "distance to the center of the interval". For any pair of points x_a and x_b such that $(x_a - a) = (x_b - b)$ we can write

$$\begin{aligned}
p_{d,b}(x_b) &= \exp(b)\left(1 + (x_b-b) + \frac{(x_b-b)^2}{2!} + \ldots + \frac{(x_b-b)^d}{d!}\right),\\
p_{d,b}(x_b) &= \exp(b)\left(1 + (x_a-a) + \frac{(x_a-a)^2}{2!} + \ldots + \frac{(x_a-a)^d}{d!}\right),\\
p_{d,b}(x_b) &= \exp(b-a)\, p_{d,a}(x_a).
\end{aligned}$$

For the actual functions, we build our lattice basis for the interval centered at a, a change of

variable has been made: $x \longmapsto y = x - a$. The actual polynomial is thus

$$P_a(y) = \exp(a) \left(1 + y + \frac{y^2}{2} + \ldots + \frac{y^d}{d!}\right).$$

For a y point located in the interval centered on b we can compute its image either with P_a or P_b if we use the following equality

$$P_b(y) = \exp(b - a)\, P_a(y).$$

We detailed above the transformations we made on these polynomials in order to obtain all integer coefficients and arguments. But these transformations amounted to scalings: they do not change the P_b/P_a ratio. If the only thing we are interested in is the ratio $\dfrac{P_b(y)}{P_a(y)}$ we can leave the polynomials under this form even when we change the real y variable to the i integer variable as was described at length in §3.3.5, p. 163. We can still write

$$P_b(i) = \exp(b - a)\, P_a(i).$$

As we remember, we introduced another integer variable t to form polynomial $Q(i, t) = P(i) - t$. If we introduce this variable to form polynomials derived from the previous ones, we can write

$$\begin{aligned} Q_a(i,t) &= P_a(i) - t \\ &= \exp(a) \left(1 + i + \frac{i^2}{2} + \ldots + \frac{i^d}{d!}\right) - \frac{\exp(a)\, t}{\exp(a)} \\ &= \exp(a) \left(1 + i + \frac{i^2}{2} + \ldots + \frac{i^d}{d!} - \frac{t}{\exp(a)}\right). \end{aligned}$$

Let us now examine how we build our lattice basis or, equivalently, our lattice matrix. Our notation will be plagued with abuses of language. But what we are about to write remains correct as long as our interest stays focused on ratios rather than on exact values.

To be more clear, let us use an example. For that end we will reuse the lattice matrix presented in §3.5.2.3, p. 182. Let us examine first a slightly different presentation of the matrix built for the interval centered at a in Table 3.6.

We have extracted to a new column, outside and to the left of the matrix table, the $\exp(a)^k$ $k \in \{1, 2\}$ factors that are common to all the term of the corresponding row. We can notice that $\exp(a)$ is risen to the same power as the one of the polynomial $Q(i, t)$. Let us call this matrix M_a. We put the label in the uppermost/leftmost table cell.

	M_a	1	i	i^2	i^3	t	i^4	it	i^5	i^6	i^2t	i^3t	t^2
	$\mathcal{N}^2$	$\mathcal{N}^2$	0	0	0	0	0	0	0	0	0	0	0
	$i\mathcal{N}^2$	-	$\mathcal{N}^2 I$	0	0	0	0	0	0	0	0	0	0
	$i^2\mathcal{N}^2$	-	-	$\mathcal{N}^2 I^2$	0	0	0	0	0	0	0	0	0
$\exp(a)$	$Q\mathcal{N}$	-	-	-	$\mathcal{N}I^3$	$\mathcal{N}T$	0	0	0	0	0	0	0
$\exp(a)$	$iQ\mathcal{N}$	-	-	-	-	-	$\mathcal{N}I^4$	$\mathcal{N}IT$	0	0	0	0	0
$\exp(a)^2$	Q^2	-	-	-	-	-	-	-	I^5	I^6	I^2T	I^3T	T^2

Table 3.6: Matrix generated by polynomials for the exp function at point a.

We can repeat the same process for the lattice matrix we built for the interval centered on b. We present it in Table 3.6

	M_b	1	i	i^2	i^3	t	i^4	it	i^5	i^6	i^2t	i^3t	t^2
	$\mathcal{N}^2$	$\mathcal{N}^2$	0	0	0	0	0	0	0	0	0	0	0
	$i\mathcal{N}^2$	-	$\mathcal{N}^2 I$	0	0	0	0	0	0	0	0	0	0
	$i^2\mathcal{N}^2$	-	-	$\mathcal{N}^2 I^2$	0	0	0	0	0	0	0	0	0
$\exp(b)$	$Q\mathcal{N}$	-	-	-	$\mathcal{N}I^3$	$\mathcal{N}T$	0	0	0	0	0	0	0
$\exp(b)$	$iQ\mathcal{N}$	-	-	-	-	-	$\mathcal{N}I^4$	$\mathcal{N}IT$	0	0	0	0	0
$\exp(b)^2$	Q^2	-	-	-	-	-	-	-	I^5	I^6	I^2T	I^3T	T^2

Table 3.7: Matrix generated by polynomials for the exp function at point b.

But, if we remember well, rows, in both matrices, correspond to polynomials and using the relationship between the polynomials we established above, we can rewrite M_b as a function of M_a. The result is displayed in in Table 3.8

	M_a	1	i	i^2	i^3	t	i^4	it	i^5	i^6	i^2t	i^3t	t^2
	$\mathcal{N}^2$	$\mathcal{N}^2$	0	0	0	0	0	0	0	0	0	0	0
	$i\mathcal{N}^2$	-	$\mathcal{N}^2 I$	0	0	0	0	0	0	0	0	0	0
	$i^2\mathcal{N}^2$	-	-	$\mathcal{N}^2 I^2$	0	0	0	0	0	0	0	0	0
$\exp(b-a)\ \exp(a)$	$Q\mathcal{N}$	-	-	-	$\mathcal{N}I^3$	$\mathcal{N}T$	0	0	0	0	0	0	0
$\exp(b-a)\ \exp(a)$	$iQ\mathcal{N}$	-	-	-	-	-	$\mathcal{N}I^4$	$\mathcal{N}IT$	0	0	0	0	0
$\exp(b-a)^2\ \exp(a)^2$	Q^2	-	-	-	-	-	-	-	I^5	I^6	I^2T	I^3T	T^2

Table 3.8: Matrix generated by polynomials for the exp function at point b as derived from that at point a.

Notice that we start here from the matrix built for the interval centered at a to form the one for the interval centered at b. Our interest turns now to the ratio between the volumes of the lattices generated by the two matrices M_a and M_b. The matrices are quite similar[29]. They differ in that, in M_b, some lines of M_a, are multiplied by $\exp(b-a)$ raised to a suitable power.

[29]Not in the technical sense of *similarity* but the natural language one.

We recall that, if we denote by L_a the lattice generated by the M_a matrix, the volume of the lattice is given by

$$\mathrm{vol}\,(L_a) \;=\; \sqrt{\det\,(M_a \,\times\, M_a^{\intercal})}.$$

where $\det\,(M_a \,\times\, M_a^{\intercal})$ is the Gram determinant[30].

We can represent the above matrices relationship by the following matrix expression

$$\begin{bmatrix} \\ M_b \\ \\ \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \exp(b-a) & 0 & 0 \\ 0 & 0 & 0 & 0 & \exp(b-a) & 0 \\ 0 & 0 & 0 & 0 & 0 & \exp(b-a)^2 \end{bmatrix} \times \begin{bmatrix} \\ M_a \\ \\ \end{bmatrix}.$$

Notice that, in the above expression, M_a and M_b look squarer than they should (they actually are outstretched rectangular) while the term multiplying M_a is actually a diagonal (hence square) matrix. Let us call this matrix Λ.

When it comes to the computation of the volume of the generated lattices, which involves transposed matrices, we can notice that

$$M_b^{\intercal} = M_a^{\intercal} \;\; \Lambda.$$

If we call $\mathrm{vol}(L_a)$ (resp. $\mathrm{vol}(L_b)$) the volume of the lattice generated by M_a (resp. M_b), we have

$$\begin{aligned} \mathrm{vol}\,(L_b) &= \sqrt{\det\,(M_b \times M_b^{\intercal})}, \\ \mathrm{vol}\,(L_b) &= \sqrt{\det\,(\Lambda \times M_a \times M_a^{\intercal} \times \Lambda)}, \\ \mathrm{vol}\,(L_b) &= \sqrt{\det\,(\Lambda) \times \det\,(\Lambda)}\ \sqrt{\det\,(M_a \times M_a^{\intercal})}. \end{aligned}$$

Since Λ is a diagonal matrix, we have

$$\det\,(\Lambda) = \exp(b-a)\;\; \exp(b-a)\;\; \exp(b-a)^2 = \exp(b-a)^4.$$

Consequently

$$\mathrm{vol}\,(L_b) \;=\; \exp(b-a)^4 \;\times\; \mathrm{vol}\,(L_a)\,.$$

This is for our example. But what is the factor in the general case?

[30]Some care is needed with volume definition as read in literature. In some instances it is presented as equal to the determinant. These cases, often implicitly, refer to square matrices were the absolute value of the "classical" determinant is interpreted as the volume. With rectangular matrices we use Gram's determinant, hence the square root.

We recall the building rule

$$is_{r,u}(i,t) = i^r Q(i,t)^u \mathcal{N}^{\alpha-u}, \text{ where } 0 \leqslant r + u \leqslant \alpha.$$

The factors $\exp(b-a)$ appear for row corresponding to $u \geqslant 1$. For a given α and u, the number of factors $\exp(b-a)^u$ will be $\alpha + 1 - u$. Their total contribution to the volume augmentation will be $\exp(b-a)^{u\,(\alpha+1-u)}$.

Their total contribution is

$$\exp(b-a)^{\sum_{u=0}^{\alpha} u\,(\alpha+1-u)}.$$

Elementary calculations show that

$$\sum_{u=0}^{\alpha} u\,(\alpha+1-u) \;=\; \frac{\alpha\,(\alpha+1)\,(\alpha+2)}{6}.$$

Hence, when it comes to lattice volumes

$$\mathrm{vol}(L_b) \;=\; \exp(b-a)^{\frac{\alpha\,(\alpha+1)\,(\alpha+2)}{6}}\,\mathrm{vol}(L_a).$$

But what eventually interests us are the relationships between the lengths of the short-norm vectors computed at a and at b. We use here a different formula than that introduced in §3.4.5, p. 172 for the bound on the first vector on a LLL-reduced basis. It is less technical (the references to the parameters used to configure the LLL-algorithm are removed), more legible and sufficient for our present needs:

$$||b_1|| \;\leqslant\; 2^{\frac{w}{2}}\;\mathrm{vol}(L)^{\frac{1}{r}},$$

where w is the number of monomials produced by all the shifts and r is the dimension (the number of vectors of the basis) of the lattice. In our case, we have:

$$||b_1|| \;\leqslant\; 2^{\frac{w}{2}}\;\mathrm{vol}(L)^{\frac{2}{(\alpha+1)(\alpha+2)}}.$$

If we want to compare now the bounds on the first vector of the reduced basis computed at point a and at point b, we have

$$\underset{@b}{||b_1||} \;\leqslant\; 2^{\frac{w}{2}}\;\mathrm{vol}(L_b)^{\frac{2}{(\alpha+1)(\alpha+2)}} \;=\; 2^{\frac{w}{2}}\;\left(\exp(b-a)^{\frac{\alpha\,(\alpha+1)\,(\alpha+2)}{6}}\,\mathrm{vol}\,(L_a)\right)^{\frac{2}{(\alpha+1)(\alpha+2)}}.$$

From which we have

$$||\underset{@b}{b_1}|| \leqslant 2^{\frac{w}{2}} \ \mathrm{vol}(L_b)^{\frac{2}{(\alpha+1)(\alpha+2)}} \ = \ 2^{\frac{w}{2}} \ \exp(b-a)^{\frac{2\alpha\,(\alpha+1)\,(\alpha+2)}{6(\alpha+1)(\alpha+2)}} \ \mathrm{vol}\,(L_a)^{\frac{2}{(\alpha+1)(\alpha+2)}} \ .$$

And, eventually

$$||\underset{@b}{b_1}|| \leqslant 2^{\frac{w}{2}} \ \mathrm{vol}(L_b)^{\frac{2}{(\alpha+1)(\alpha+2)}} \ = \ 2^{\frac{w}{2}} \ \exp\left(\frac{\alpha}{3}(b-a)\right) \ \mathrm{vol}\,(L_a)^{\frac{2}{(\alpha+1)(\alpha+2)}} \ .$$

If we examine now what the values of a and b can be, we remind that we scale everything to the 0-fbinade whose bounds are 1 and $2-2^{-p+1}$. Hence, $b-a < 1$. For the $\alpha = 6$ case, a typical value when one searches for a hardness-to-round of $6p$,

$$\exp(1)^2 \approx 7.39.$$

This is the ratio between the bounds on the shortest-norm vectors of the LLL-reduced basis computed at points $a = 1$ and $b = 2-2^{-p+1}$. If parameters are set in order to get an efficient operation at the upper end of a binade, one is guaranteed that these settings will work all the way down to its lower end.

Using the same parameter values for several binades is more problematic. The ratio computed above increases by 7.39 per binade. If the settings adopted for the worst case are used all along a large interval, it will very quickly happen that they are far from optimal for the lowest values. As we have seen, other considerations, as approximation error, come into play when exploring far apart intervals.

As a final note, the above analysis only applies, as is, to the $\exp(x)$ function.

3.6 The Implementation

In this section, we will give a closer look to the implementation itself. The transition from paper and pencil algorithms to an actual running code should never be overlooked. It is especially true here, where practical goals matter so much to us. In any case, and regardless of the efforts made by the developer to stick with the specification, the practical conditions of the implementation always introduce twists of their own that have an impact on the behavior of the code. One is not always able to report them completely. At some point, the tools that are used for the implementation are black boxes that are opaque to the developer himself. This is unavoidable and even, in a way, desirable.

3.6.1 The tools of the trade

We will first introduce the tools used for the implementation since some of its peculiarities are related with the features (and lack thereof) of the tools.

3.6.1.1 Sollya[13]

This is our first tool. The simplest and most accurate description is found in the introduction to the reference manual:

> Sollya is an interactive tool for handling numerical functions and working with arbitrary precision. It can evaluate functions accurately, compute polynomial approximations of functions, automatically implement polynomials for use in math libraries, plot functions, compute infinity norms, etc. Sollya is also a full-featured script programming language with support for procedures etc.

As it will be seen, it is mainly used here for its ability to compute polynomial approximations as Taylor forms (approximation errors are provided as well) and infinite norms. The second interesting trait is that most of the features of Sollya are also available through calls to a `C` language library which allows us to harness it to other tools. Besides, it is free software and is continuously enhanced by three researchers who are very active in the realm of approximation at large and who use it for their own research. A wider, if small yet, community of users is building around the tool.

3.6.1.2 fpLLL

The implementation of the LLL-algorithm we use is `fpLLL`[2] This incarnation of the LLL-algorithm is disseminated as a `C` language library. Thanks to this format it can also be called from other tools. As such, it is one of the two default implementations of the LLL-algorithm in SageMath.

Its other outstanding feature is very good performance. As reductions are to be performed very often and take the lion's share of computing time in this implementation this aspect cannot be overlooked.

Besides LLL-reduction, `fpLLL` offers another reduction algorithm, BKZ, and a function for the computation of the shortest nonzero vector of a lattice. A set of data type templates are also provided to allow for the manipulation different types (e.g. different integer types) in a uniform way.

In fact, state-of-the art libraries do not expose a function that performs "the" LLL-algorithm. For one, using floating-point numbers is a departure from original LLL-algorithm. For two, even in this realm, innovation has constantly been popping up, notably Schnorr-Euchner's (both proved [93],

Schnorr by himself, and heuristic [94][95]) and L^2 [80] algorithms. Developers of implementations feel compelled to offer several options to their users because of the possible execution time, problem complexity and results safety trade-offs.

If we come back to the `fpLLL` library, several options are offered for reduction. Beyond those that control its quality (at the expense of more computation time), $1/4 < \delta \leqslant 1$ and $1/2 < \eta < \sqrt{\delta}$, aforementioned in the LLL section, the two main controls are on *method* and *floatType*. A *precision* argument can be used when computation are to be performed with MPFR floating-point numbers.

The following is excerpted from `fpLLL` documentation and gives some details.

First, about the available methods:

Method	Description
LM_WRAPPER	Tries to reduce the matrix with a combination of the following methods with increasing precision. Then, runs the proved version with the default precision. The floatType must be FT_DEFAULT and precision must be 0. The result is guaranteed.
LM_PROVED	Proved method. Uses integers to compute dot products. The result is guaranteed if floatType=FT_DEFAULT/FT_MPFR and precision=0 (see above).
LM_HEURISTIC	Heuristic method. Uses floating-point numbers to compute dot products. The result is not guaranteed. It is more efficient that the proved version when the coefficients of the input are large (the threshold depends on the floating-point type and precision).
LM_FAST	Same as LM_HEURISTIC with floatType=FT_DOUBLE, with a special trick to handle larger inputs.

Next, about floating-point number types (FP_DOUBLE and FT_MPFR are self-explanatory) in relation to methods:

	LM_WRAPPER	LM_PROVED	LM_HEURISTIC	LM_FAST
FT_DEFAULT	yes	yes, same as FT_MPFR	yes, same as FT_DPE	yes, same as FT_DOUBLE
FT_DOUBLE	no	yes	yes	yes
FT_DPE (large exponent floating-point numbers)	no	yes	yes	no
FT_MPFR	no	yes	yes	no

As `fpLLL` is invoked from SageMath, one has to check the arguments provided in this environment. Indeed SageMath offers a choice between the `fpLLL` methods and an alternative implementation from Victor Shoup's NTL[97].

We invoke `fpLLL` with the default parameters in SageMath which correspond to *LM_ WRAPPER* and *FT_ DEFAULT*. This method is dubbed a "Thoughtful Wrapper" as it relieves the user from the burden of selecting and transitioning between variants. What end-users (as we characterize ourselves in this implementation) want here is the execution to terminate and to return a good quality answer. It is not an easy task when it is to be done the proper way (i.e. trying to speed up things when it is possible and not unconditionally following a safe but slow route). It involves making use of heuristics, detecting and interpreting misbehavior that may happen in such a way that the cheapest variant that is likely to safely work is picked.

Important notice about LLL

From *LLL*-algorithm analysis (as in D. Stehlé's work [101]), one should not overlook the fact that the bit-complexity of the algorithm is not only bound to the dimension of lattice basis. It also depends on the bit size of input (see §3.6.1.5, p. 198)

3.6.1.3 SageMath

A bird's eye view of SageMath can be given by quoting the front page of the SageMath Web Site:

> SageMath is a free open-source mathematics software system licensed under the GPL. It builds on top of many existing open-source packages: NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R and many more. Access their combined power through a common, Python-based language or directly via interfaces or wrappers.

The target and ambition of the project can be read from SageMath's FAQ as:

> The stated mission of SageMath is to be viable free open source alternative to Magma, Maple, Mathematica, and Matlab.

We can augment this description with a few words about the features we particularly relied upon in this work.

First of all, we will be manipulating multivariate polynomials. Since SageMath has a more than decent support for these objects, there was no point in trying to develop a new library from scratch. Using excellent embedded libraries as `Maxima` was a wiser choice. And that's what we actually do through SageMath.

The second important element is the ability to use SageMath to integrate scattered pieces of code. This is one of its key features since, for lots of its operations on many different mathematical objects, it relies on best-of-breed open source specialized libraries. At its inception, SageMath specific code was mostly written to plug up the holes between existing libraries and glue them together. Nowadays many new developments take place directly in Python/SageMath.

This leads to a third element which is the ability to squeeze different performance levels from a given SageMath code. Initially, code written in "pure" SageMath runs at the speed allowed by the underlying Python interpreter[31]. While it is not particularly slouch, it is still possible to improve upon it in terms of performance. A possibility is to have the initial code transformed into C and compiled with the help of Cython[22]. This code executes more efficiently than native Python code. Most of SageMath's core is nowadays written in Cython.

Another interesting feature of the use of Cython for SageMath performance improvement is that the shift is not an all-or-nothing move. Initially, Cython code is almost Python code translated to C. One can obtain an increasing performance boost by enlightening the original Python code with annotations that provide information about the manipulated objects. The more details are given, the more the compiler has room for improvement. But, at the same time, the further it gets from the original Python/SageMath code. The process can be incrementally performed: when time allows, or when performance bottlenecks urge, a higher level of code improvement can be reached with more work on the already running code. Tiny bits of our bindings of Sollya were written in Cython, not so much for performance improvement but rather to access the underlying *C* language structures of some arbitrary precision numbers from MPFR, MPI, and GMP libraries.

[31] We refer here to code written by an application developer, not SageMath code itself

Together these reasons explain why SageMath was picked as the integration platform all along this project.

3.6.1.4 All together now!

As explained above, the integration of `fpLLL` to SageMath had already been realized by the authors of the former. What was missing yet was an access to Sollya from SageMath. An important effort was devoted to this task. While publicly available[107], this is still a work in progress or even a draft since only we concentrated on the part that was absolutely needed for this work: large parts remain to be coded for a complete binding. Hence the existing wrapper cannot be considered yet as a full integration of Sollya. Completing it (probably at the cost of a full rewrite taking into account the gathered experience) is future work and our experience with Sollya power harnessed to SageMath has convinced us that it would be worth the effort. Many developers in SageMath but also in Python could benefit from this work. It could also offer a wider audience to Sollya thanks the momentum SageMath is picking up.

3.6.1.5 Close shaved polynomials look nicer

The LLL-algorithm complexity depends on multiple factors. In the initial A. K. Lenstra and al. article [65] the complexity of the algorithm is upper bounded by

$$O(n^4 \log B) \text{ arithmetic operations on integers of binary length } O(n \log B) \text{ where}$$

- n is the number of vectors in the lattice basis
- B is the maximum of the L^2 norms of the vectors of the input matrix.

Which gives a total complexity of $O(n^6 \log^3 B)$ of bit operations.

Better bounds may have been obtained with new variants of the algorithm and/or for special cases, but these data give an idea of the execution time dependency on the input data size. Since basis reduction grabs the largest, by far, share of overall execution time, all legitimate transformations on input data that have a favorable impact are definitely welcome. We have already seen in our discussion about the initial basis (§3.5.2, p. 178) how dropping some vectors that add not much value kills two birds with one stone:

- get a lattice with a smaller average volume;
- reduce the lattice dimension.

The first allows generically for shorter-norm vectors in the output basis. The second makes reductions quicker. But as can be clearly seen in the above formula, vectors bit size has also a large

impact on complexity.

It happens that when we compute our approximation polynomial $P(x)$ of function $g(x)$ output coefficients span all the available bits in working precision of the tool. Indeed, it tries very hard to give the best possible approximation and has no particular reason to optimize coefficients size. It turns out, especially for high degree polynomials, that all this precision in coefficients is superfluous and that shorter ones may yield very decent approximants.

Of course, reduction operated at this early stage may not fully withstand all the transformations the initial polynomial will suffer (from floating-point numbers to integers and the incorporation of possibly large parameters, as the bounds on i and t or $\mathcal{N}$). Nevertheless, as will be seen from our experiments, the result is worth the effort.

The question is how can it be done in an efficient and yet safe way?

The main intuition is that rounding should be different among coefficients: it should prune more bits from the coefficients of the higher degree monomials that from the low degree ones.

Indeed, the intervals we work on always have a radius smaller than 1 (remember we map any f-binade into the 0-fbinade $= [\![1, 2[\![$). As we move along to higher degree monomials, the powers of the argument get smaller and smaller. Their multiplication with the coefficients yields products whose least significant bits bring in only a very small contribution to the final value of the evaluation of the polynomial. The same does not apply, for instance, to the constant coefficient. But even for this one, rounding may still be desirable since it has possibly been computed at a very high precision, more than is actually needed for the expected accuracy of the approximation.

We first approximate function $g(x)$ with polynomial $P(y)$ (with floating-point coefficients we will view here as rational numbers) of degree d over the interval $[a, b]$, with an accuracy of 2^{-m-1}. We must remember here that $P(y)$ is a Taylor polynomial that approximates $g(x)$ with an argument $y = x - x_0$, where x_0 is the center of interval $[a, b]$.

We then have:

$$|g(x) - P(y)|^{[a,b]} \leqslant \frac{1}{2^{m+1}},$$
$$\text{with } P(y) = p_0 + p_1 y + \ldots + p_d y^d$$
and where m is the hardness-to-round we want to check.

In order to work with smaller coefficients, we want to replace $P(y)$ by some polynomial $R(y)$ ("R" stands for rounded here) of the same degree d, derived from $P(x)$, such that:

$$|g(x) - R(y)|^{[a,b]} \leqslant \frac{1}{2^{m+1}}, \text{ with } R(y) = r_0 + r_1 y + \ldots + r_d y^d.$$

The succession of events is the following:

1. $P(y)$ is derived from $g(x)$;
2. $R(y)$ is derived from $P(y)$.

How can this be done in order to satisfy the final conditions on $|g(x) - R(y)|^{[a,b]}|$? More precisely, what are the conditions on $P(y)$ and $R(y)$ that make it possible?

Since $|g(x) - P(y)| \leqslant |g(x) - P(y)| + |P(y) - R(y)|$, an obvious and acceptable choice is

$$|g(x) - P(y)| < \frac{1}{2^{m+2}},$$
$$|P(y) - R(y)| \leqslant \frac{1}{2^{m+2}}.$$

Notice that if polynomial rounding is to be used, we must start with a $P(y)$ polynomial that is a closer approximation to function $g(x)$ than we initially anticipated.

In order to derive polynomial $R(y)$ from $P(y)$ we are writing our polynomials a bit differently,

$$\left| \sum_{k=0}^{d} p_k\, y^k \; - \; \sum_{k=0}^{d} r_k\, y^k \right| \leqslant \frac{1}{2^{m+2}}$$

We write the coefficients of polynomial $R(y)$ as rational numbers with a denominator that is a power of 2:

$$\left| \sum_{k=0}^{d} p_k\, y^k \; - \; \sum_{k=0}^{d} \frac{R_k\, y^k}{2^{m_k}} \right| \leqslant \frac{1}{2^{m+2}}, \text{ with } R_k \in \mathbb{Z} \text{ and } m_k \in \mathbb{N},$$

$$\left| \sum_{k=0}^{d} \left(p_k - \frac{R_k}{2^{m_k}} \right) y^k \right| \leqslant \frac{1}{2^{m+2}}, \text{ with } R_k \in \mathbb{Z} \text{ and } m_k \in \mathbb{N}.$$

What we want now is to compute the m_k's. Once those set, we will be able to compute the R_k's from the p_k's.

At this point if we can find a sharp bound on y, we can simplify a bit the problem. It is time to use the fact that both $P(y)$ and $R(y)$ are polynomials on the same variable whose domain is centered on zero: $y \in [-\rho, \rho]$, where ρ is the radius of interval $[a, b]$. Put in another way: $|y| \leqslant \rho$.

We can then write

$$\left|\sum_{k=0}^{d}\left(p_k - \frac{R_k}{2^{m_k}}\right) y^k\right| \leqslant \left|\sum_{k=0}^{d}\left(p_k - \frac{R_k}{2^{m_k}}\right) \rho^k\right|.$$

If we can bound our new sum with $\frac{1}{2^{m+2}}$, the initial one will be bounded as well. That is what we do in the following lemma.

Lemma 3.9. *If,* $\forall k \in 0, \ldots, d$, *we set* $R_k = \lfloor 2^{m_k} p_k \rceil$ *and* $m_k = \left\lfloor \log_2\left(\frac{2^{m+2}(d+1)}{\rho^{-k}}\right)\right\rfloor$ *then* $\left|\sum_{k=0}^{d}\left(p_k - \frac{R_k}{2^{m_k}}\right)\rho^k\right| \leqslant \frac{1}{2^{m+2}}$, *with* $R_k \in \mathbb{Z}$ *and* $m_k \in \mathbb{N}$.

Proof. Let us consider each term of the above sum, using the value we have set the R_k's to in our hypothesis. We first have

$$\left|p_k - \frac{R_k}{2^{m_k}}\right| = \left|p_k - \frac{\lfloor 2^{m_k} p_k \rceil}{2^{m_k}}\right| = \left|\frac{2^{m_k} p_k}{2^{m_k}} - \frac{\lfloor 2^{m_k} p_k \rceil}{2^{m_k}}\right| = \left|\frac{2^{m_k} p_k - \lfloor 2^{m_k} p_k \rceil}{2^{m_k}}\right| \leqslant \frac{1}{2^{m_k+1}}. \tag{3.12}$$

Considering the assignment of the m_k in our hypothesis, we also have

$$\frac{1}{2^{m_k}} = \frac{1}{2^{\left\lfloor \log_2\left(\frac{2^{m+2}(d+1)}{\rho^{-k}}\right)\right\rfloor}} \geqslant \frac{1}{2^{\log_2\left(\frac{2^{m+2}(d+1)}{\rho^{-k}}\right)}} \geqslant \frac{1}{2^{\left\lceil \log_2\left(\frac{2^{m+2}(d+1)}{\rho^{-k}}\right)\right\rceil}} \geqslant \frac{1}{2^{m_k+1}}.$$

From here we can deduce

$$\frac{1}{2^{\log_2\left(\frac{2^{m+2}(d+1)}{\rho^{-k}}\right)}} = \frac{1}{\frac{2^{m+2}(d+1)}{\rho^{-k}}} = \frac{\rho^{-k}}{2^{m+2}(d+1)} \geqslant \frac{1}{2^{m_k+1}}. \tag{3.13}$$

Combining both (3.12) and 3.13, we then have

$$\left|p_k - \frac{R_k}{2^{m_k}}\right| \leqslant \frac{\rho^{-k}}{2^{m+2}(d+1)} \text{ and } \left|p_k - \frac{R_k}{2^{m_k}}\right| \rho^k \leqslant \frac{1}{2^{m+2}(d+1)}. \tag{3.14}$$

Since $\rho > 0$, we remark that

$$\left|\frac{R_k}{2^{m_k}}\right| \rho^k = \left|\frac{R_k}{2^{m_k}} \rho^k\right|.$$

If we return now to our sums, we can notice the $d+1$ factor in the denominator of (3.14). We then

have

$$\left| \sum_{k=0}^{d} \left(p_k - \frac{R_k}{2^{m_k}} \right) \rho^k \right| < \frac{1}{2^{m+2}}.$$

□

Our assignment of the m_k's in the hypothesis of Lemma 3.9 finally amounts to an even split of the $\frac{1}{2^{m+2}}$ bound among all the terms of the polynomial. For a given approximation interval, the different powers ρ^k throttle the corresponding values for the m_k's. In general, since $\rho < 1$, the value of the m_k will shrink with as k grows larger.

Some tests confirm the intuition that polynomial coefficients "shaving" can improve execution time.

A test result

We performed our test with the following parameters, typical of our usage of the algorithm:

- function: $\exp(x)$;
- interval: $[1.5 - 2^{-35}, 1.5 + 2^{-35}]$;
- polynomial approximation degree: 17;
- α : 6;
- precision: 113 (`binary128`).

The run time reduction of polynomial "shaving", put parallel with the effect it has on data size reduction, measured according to different metrics, is presented in Table 3.9.

Execution time ratio (rounded/non-rounded)	0.25
Maximum vector L^2 norm ratio (rounded/non-rounded)	0.67
Mean vector element size ratio (rounded/non-rounded)	0.70
Maximum vector size ratio (rounded/non-rounded)	0.66

Table 3.9: Reduction of execution time with cutback of Taylor approximation polynomial coefficients initial binary size.

As can be seen, the impact of polynomial coefficients rounding on execution time largely exceeds that it has on raw data size.

Starting from 1024, the final bit size of each polynomial coefficient is represented in Figure 3-10. As remarked above, all the coefficients could benefit from this optimization, including the constant

term.

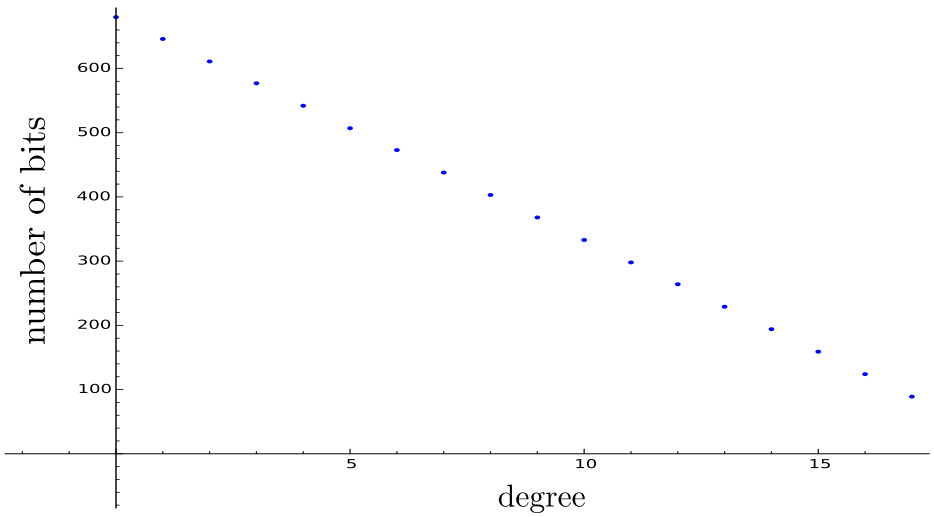

Figure 3-10: Starting from a uniform 1024 bit size, the coefficients of the test polynomial in degree 17 are rounded to the presented values.

Incidentally, due to a programming error of Equation 3.12, factor $d+1$ was replaced by k.

This led to an even better reduction, confirming that our assignement of the m_k's in Lemma 3.9 was somehow pessimistic. Further testing showed that this extra-aggressive reduction sometimes failed (the approximation accuracy was not preserved to an acceptable level).

Nevertheless, as it is worth spending some time in data size reduction, we use a two steps coefficients rounding:

- we first apply the very aggressive formula, and check the approximation accuracy;
- if it fails, we fall back to the proved one.

3.6.2 Touring the implementation

What we mean to do here is to give a taste of what the implementation looks like through a small example. As we go, we will provide some information about technical details of interest and about some algorithmic aspects as well.

In this example we deal with the exp function. It has been used all around this work to develop the implementation and perform the tests that will be reported later. The parameters used in our examples make it a bit "toyish" (even if these parameters correspond more or less to those used in the "fast phase" of a two-phase function implementation, see [61]). Our main justification is that we

had to keep data size moderate in order to be able to legibly display them on paper.

In this tour, we will display accuracy check of the polynomial approximation after each of the many conversions we perform along the way. These checks do not happen in real code since we have thoroughly tested it in this respect and have taken into account the fact that many of the transformations are trivial. Checks in actual code do not go beyond polynomial rounding. They are presented here to support our above claim of accuracy conservation.

3.6.2.1 Initial data

$$
\begin{aligned}
\text{precision: } & p & &= 53 \\
\text{hardness-to-round: } & m & &= p + 10 = 63 \\
\text{approximation polynomial accuracy: } & 2^{-m-1} & &< 2.71050543121376108501863 20e - 20 \\
\text{approximation polynomial hardness-to-round: } & m + 1 & &= 64 \\
\text{function: } & f & &= \exp(x) \\
\text{alpha: } & \alpha & &= 2 \\
\text{degree: } & d & &= 2 \\
\text{interval center: } & ic & &= 3/2 \\
\text{ulp}(ic)\text{: } & ulpic & &= 2^{-53+1} \\
\text{interval width in ulpic: } & iwu & &= 2^9 ulpic \rightarrow [ic - 2^8 ulpic, ic + 2^8 ulpic] \\
\text{lower bound: } & lb & &= 1.4999999999999431565811391919851303100585937 5 \\
\text{upper bound: } & ub & &= 1.5000000000000568434188608080148696899414062 5 \\
\text{Sollya internal pecision: } & prec & &= 192 \text{ bits.}
\end{aligned}
$$

3.6.2.2 Initial interval splitting

There is nothing to do here, since :

- function $\exp(x)$ is increasing in the $[lb, ub]$ interval;
- bounds lb and ub belong to the same binade (0-binade);
- both bound images belong to the same binade (2-binade).

3.6.2.3 Scaling

In order to map the images in the 0-binade, we use the following scaled function:

$$g : x \mapsto \frac{\exp(i)}{4} \tag{3.15}$$

instead of $f(x)$.

3.6.2.4 Taylor expansion computation

Taylor expansion is computed with the Sollya `taylorform` function in absolute error mode. We picked `taylorform` instead of the quickest `taylor` function because the former also returns bounds on the error term. Testing the highest absolute value for the bound guarantees the accuracy of the approximation.

Taylor expansion

P_{so} denotes the approximation polynomial output by Sollya.

$$\begin{aligned} P_{so}(i) = \quad & 1.120422267584516205650513865029818954751437467092416764193 \\ + \ & 1.120422267584516205650513865029818954751437467092416764193\ i \\ + \ & 0.560211133792258102825256932514909477375718733546208382096 5\ i^2 \end{aligned}$$

Taylor expansion related interval splitting

As mentioned, Sollya `taylorform` function also returns an interval for the approximation error of $g(x)$ by P_{so}. In this case :

$$\begin{aligned} [&-3.429817828336139277551692123280460109461922906718027756905 7e-41, \\ &3.429817828336236742914815206710768736364658335326293820682e-41]. \end{aligned}$$

To avoid taking a risk, we use the upper bound of the interval (more precisely the maximum of the absolute values of the bounds) to check that the approximation matches our target accuracy

$$\epsilon_{P_{so}} \approx \quad 3.429817828336236742914815206710768736364658335326293820682e-41.$$

For the given accuracy ($5.421010862427522170037264 0e-20$) and the given degree (2), one polynomial is sufficient to approximate $g(x)$ with the expected accuracy over all the interval $[lb, ub]$. No

further splitting is needed.

We must notice that one can not use the same inputs for the $g(x)$ function and the P_{so} polynomial. For the truncated Taylor polynomial, argument 0 represents the center of the interval $[lb, ub]$. When one wants to check P_{so} against function $g(x)$ she must make a variable change:

$$x \text{ (for } g(x)) \;\mapsto\; y = x - ic \text{ (for } P(y))$$

At this point, we can also notice that the approximation error of polynomial P_{so} is needlessly good. On the other hand, all its coefficients are expressed with 192 bits of precision, what is way too large for our purposes. In order to reduce the input size, we can try to cut back the coefficients binary size in an orderly way.

Indeed, we can reduce our the size of our polynomial coefficients as follow:

$$\begin{aligned} P_{so_{round}} = \quad & 1.120422267584516205665527699419925511392648331820964813232 4 \text{ (64-bit number)} \\ & + 1.12042236328125\ i \qquad \text{(15-bit number)} \\ & + 0.5625\ i^2 \qquad \text{(4-bit number)}, \end{aligned}$$

for an approximation error

$$\epsilon_{P_{so_{round}}} \;\approx\; 2.045356392018573155951789273211367895641067062486318163894 9e-20$$

which is more than sufficient but not by an extraordinary wide margin. As can be seen, the reduction in size is noticeable.

Recovering the polynomial in SageMath

The expression tree of $P_{so_{round}}$ is traversed to find out which is the maximum precision used to represent the coefficients (as we have seen, precision can differ from one coefficient to another in Sollya). Conversely, in SageMath, polynomials are defined over a ring of floating-point numbers. Such rings are indexed by the precision of their elements, hence all the coefficients of a polynomial must be represented with the same number of bits. The maximum value (here 64 bits) is used to convert the Sollya polynomial into a SageMath polynomial with no information loss: all coefficients are converted into SageMath with the broadest format.

There is no visible different in the output of the two polynomials. Nevertheless, the size reduction operated above is not lost: it relates to the informational content, not to the representation. In the

sequel, we will be able to actually use the fact that the coefficient of i^2 is a 4-bit number despite the fact it is represented, in the initial conversion, by a 64-bit precision floating-point number. $P_{sa,ff}$ denotes the floating-point coefficients (first `f`) polynomial for floating-point arguments (second `f`) in SageMath.

$$\begin{aligned} P_{sa,ff}(i) = \quad & 1.12042226758451621 \\ & + \ 1.12042236328125000 \ i \\ & + \ 0.562500000000000000 \ i^2 \end{aligned}$$

From now on, all computations (but infinity norm of errors) are performed in SageMath. We want to make sure that, at any point, the just performed transformation has not spoiled the approximation accuracy achieved previously. Using the variable change proposed above we can check with the Sollya `dirtyinfnorm` function that

$\|g - P_{sa,ff}\|_\infty$ over $[lb, ub] \approx 2.045356392018573155951789273211367895655212982466770090132e{-}20$

This is marginally different from the initial value in Sollya and still better than enough for our purpose.

3.6.2.5 The Long March to the integral polynomial

In this section, through a series of transformations, we will derive an integral polynomial from our floating-point initial one. This will be the starting point to set up our small roots finding problem.

From a floating-point coefficients polynomial to a rational coefficients one

Indeed floating-point coefficients and arguments are self-ignorant rational numbers. The transformation are then trivial. What matters here is to get things done with diligence. In the next step we make this correspondence explicit by transforming the original $P_{sa,ff}$ into a polynomial over the rational numbers. Let us call this polynomial $P_{sa,rr}$. Its coefficients are rational, hence the first `r` and its possible arguments are floating-point numbers or rationals (we picked `r` in the designation).

$$\begin{aligned} P_{sa,rr}(i) = \quad & 10334071412308445887/9223372036854775808 \\ & + \ 18357/16384 \ i \\ & + \ 9/16 \ i^2 \end{aligned}$$

Again an approximation error check can be done with the previously done change of variable.

$\|g - P_{sa,rr}\|_\infty$ over $[lb, ub] \approx 2.0453563920185731559517892732113678956410670624863181638949e{-}20$

The small discrepancies in infinity norms show up at about the 40th number in the fractional part are due to decimal to binary conversion at slightly different precisions in Sollya.

From floating-point/rational arguments to integral arguments

$P_{sa,rr}$ accepts floating-point and rational arguments. What we do now is to transform it in order to accept only integral arguments: the numerators. This implies that the denominators corresponding to the argument get incorporated into the coefficients, taking into account the degree of the argument in the corresponding monomial.

We obtain the following polynomial $P_{sa,ri}$ for coefficients are rational numbers and integral arguments.

$$\begin{aligned} P_{sa,ri}(i) = \quad & 1033407141230844 5887/9223372036854775808 \\ + \ & 18357/73786976294838206464\ i \\ + \ & 9/3245185536584267267831560205762 56\ i^2 \end{aligned}$$

If we want to check the accuracy of the approximation, we have to use the following variable change:

$$x \text{ (for } g(x)) \ \mapsto \ y = (x - ic)/ulpic \text{ (for } P(y))$$

When, after the variable change, the check is performed we get:

$\|g - P_{sa,ri}\|_\infty$ over $[lb, ub] \approx 2.0453563920185731559517892732113678956410670624863181638949e{-}20$

which should not surprise anyone.

From rational coefficients to integer coefficients

Our final target is an all integral expression. For that purpose, what we want to do now is to eliminate the denominators. Not only those of the polynomial coefficients but also those of the other terms that show up in our inequalities as well. Let us remind our main inequality here:

$$\forall i \in [\![rl..ru]\!],\ \forall j \in [\![j_{rl}, j_{ru}]\!],\ \left| \ P\,(i) - \frac{2j+1}{2^p} \ \right| > \frac{1}{2^{m+1}}.$$

We also have to remove the denominator of $\frac{2j+1}{2^p}$ and $\frac{1}{2^{m+1}}$ terms.

This involves first reducing all rational numbers (polynomial coefficients and other terms) to the same denominator by computing their least common multiple. Let us call it $scaling\_factor$ (we will see why below). We will store it for book keeping reasons. Of course, we adjust all numerators accordingly. When done with it, we can remove the (now common) denominator.

Let us call $P_{sa,ii}$ the obtained polynomial (integer coefficients and integer argument).

The previous inequality becomes:

$$\forall i \in [\![rl..ru]\!],\ \forall j \in [\![j_{rl}, j_{ru}]\!],\ \left| P_{sa,ii}(i) - \frac{scaling\_factor\,(2j+1)}{2^p} \right| > \frac{scaling\_factor}{2^{m+1}}.$$

The $\frac{scaling\_factor}{2^p}$ and $\frac{scaling\_factor}{2^{m+1}}$ terms are integral, since, as we remember, $scaling\_factor$ is the least common multiple of a list of numbers that includes 2^p and 2^{m+1}. In our algorithm description we denoted $2^K = \frac{scaling\_factor}{2^p}$ and $2^k = \frac{scaling\_factor}{2^{m+1}}$, based on the theoretical assumption that the denominators of polynomial[32] $P_{sa,ri}$ coefficients were all powers of 2.

In real world computations this may not necessarily be the case at this stage since the computer algebra system (SageMath for that matter) may have reduced floating-points coefficients to fractions on its own way. It turns out that SageMath does it in a very hostile way to us[33] by approximating floating-points with continued fractions. The consequence is that, if nothing is done about it, the odds are very high that the denominators become mutually prime, and that $scaling\_factor$ becomes apocalyptically huge. So much for polynomial coefficients shaving. Performing the coefficient transformation by ourselves was straightforward and left us in total control of the transformation. As a consequence, all our initial assumptions were still standing and the binary size of data was kept at a manageable level.

Those are the results for our example:

$$\begin{aligned} P_{sa,ii}(i) = \quad & 3635978137632219705282618290339 84 \\ & + \ 80734939804336128\ i \\ & + \ 9\ i^2 \end{aligned}$$

[32] A polynomial with rational coefficients that derived, one way or another, from floating-point numbers.

[33] We do not take it personally: SageMath has very good reasons to adopt this default behavior. For many applications, it is the most sensible way to convert floating-point numbers into rational ones.

For the record, the other computed data are:

$$\begin{aligned} \mathcal{N} = &\quad 36028797018963968 \\ iBound = &\quad 256 \text{ (remember the interval radius is } 2^8 ulpic) \\ tBound = &\quad 35184372088832 \\ scaling\_factor = &\quad 324518553658426726783156020576256 \end{aligned}$$

With the same variable change as before, but this time dividing the output of $P_{sa,ii}$ by $scaling\_factor$ (hence the variable name), we can still check the quality of the approximation accuracy:

$$\left\| g - \frac{P_{sa,ii}}{scaling\_factor} \right\|_\infty \text{ over } [lb, ub] =$$

$$2.045356392018573155951789273211367895641067062486318163894 9e-20.$$

No change, nor surprise.

From $P_{sa,ii}(i)$ we can easily derive what we denote as $Q(i,t) = P_{sa,ii}(i) - t$, $Q(i,t) \in \mathbb{Z}[i,t]$.

At this point we want to solve the modular equation:

$$Q(i,t) \mod \mathcal{N} = 0$$

We are now ready to create the family of polynomials used to build the basis of lattice L that will be input into an LLL-algorithm implementation for reduction.

3.6.2.6 Creating polynomials for a lattice basis

The choice of the polynomial set is very important, as discussed in §3.5.2, p. 178. We use here the selection proposed by D. Stehlé in his PhD thesis [99] and his 2006 article [100].

We remind here that we set the α parameter to 2, essentially here for legibility reasons .

We compute here all the polynomials such that:

$$Q_{r,s}(i,t) = i^r Q^s(i,j) \mathcal{N}^{\alpha - s}, r \in \mathbb{N},\ s \in \mathbb{N},\ r + s \leqslant \alpha$$

We obtain the following list of polynomials:

$$
\begin{aligned}
Q_{0,0}(i,t) &= & & & \mathcal{N}^2 \\
Q_{1,0}(i,t) &= & i & & \mathcal{N}^2 \\
Q_{2,0}(i,t) &= & i^2 & & \mathcal{N}^2 \\
Q_{0,1}(i,t) &= & & Q(i,t) & \mathcal{N} \\
Q_{1,1}(i,t) &= & i & Q(i,t) & \mathcal{N} \\
Q_{0,2}(i,t) &= & & Q(i,t)^2 &
\end{aligned}
$$

Once these polynomials are created we must make the following change of variables:

$$i \ \mapsto \ iBound\, i \ \text{ and } \ t \mapsto tBound\, t,$$

in order to obtain the elements for the lattice L generating matrix. In fact $iBound$ and $tBound$, raised to the power corresponding to the degree of the monomial where the variable change is made, are incorporated into the coefficients.

The obtained polynomials are listed below.

$$
\begin{aligned}
Q_{0,0}(i,t) = \quad & 1298074214633706907132624082305024 \\
Q_{1,0}(i,t) = \quad & 33230699894622896822595176507008614 4\, i \\
Q_{2,0}(i,t) = \quad & 85070591730234615865843651857942052864\, i^2 \\
Q_{0,1}(i,t) = \quad & 21250649172913403461632\, i^2 \\
& + 74464838618846742794479938371479142 4\, i \\
& - 12676506002282294014967032053 76\, t \\
& + 130999918286141877471\ 78429227109016058669943488512 \\
Q_{1,1}(i,t) = \quad & 54401661882658312861777 92\, i^3 \\
& + 19062998686424766155386864223098660454 4\, i^2 \\
& - 32451855365842672678315602057625 6\, i\, t \\
& + 335359790812523206327767788213990811101950553305907 2\, i \\
Q_{0,2}(i,t) = \quad & 347892350976\, i^4 \\
& + 24381135429198209209073664\, i^3 \\
& - 41505174165846491136\, i^2\, t \\
& + 85608963459958528905168183156041947545 6\, i^2 \\
& - 14543913792743504452046862963179 52\, i\, t \\
& + 12379400392853802748991242 24\, t^2 \\
& + 15029784374866915279684651582627512986812759538663424\, i \\
& - 25585921540262085443707869584197296989589733376\, t \\
& + 13220337017339464821776068775331593809528549087129014099462691225 6
\end{aligned}
$$

3.6.2.7 Creating the lattice matrix

In many of the classical presentations of matrix generated lattices, the basis of is described in term of column vectors. Nevertheless, this is not a general rule among authors. Besides, the implementation of the *LLL*-algorithm used by SageMath, `fplll`, uses a row oriented matrix. In order to keep the exposition and the code in line, this presentation will be row-oriented.

Here the row vectors of the matrix are based on the polynomials obtained above. They are built as follows:

1. we create Ω, an ordered list of all the monomials included in the $Q_{r,s}(i,t)$ polynomials (not taking here the coefficients into account);

2. for each $Q_{r,s}(i,t)$ polynomial we create a vector $V_{r,s}$ which holds, for each element of Ω, its coefficient in $Q_{r,s}(iBound\ i, tBound\ t)$; notice how $iBound$ and $tBound$ make their way into the vectors; logically, if some monomial $\omega_j \in \Omega$ is missing in $Q_{r,s}(i,t)$ the value for the corresponding element of the $V_{r,s}$ is 0 at position j.

3. Matrix M is built with all $V_{r,s}$'s as it's rows.

In our example the ordered list of monomials is the following :

$$\Omega = [1, i, t, i^2, it, t^2, i^2t, i^3, i^4].$$

For instance, the vector corresponding the first (constant) polynomial $Q_{0,0}(i,t) = \mathcal{N}^2$ is $V_{0,0} = (\mathcal{N}^2, 0, 0, 0, 0, 0, 0, 0, 0,)$. This vector will be the first row of the M matrix.

Displaying the matrix on paper is difficult because its elements are really large. As they are the coefficients of the polynomials listed above, this would not add much information either.

3.6.2.8 Reducing the matrix

Once the matrix has been built, it is input to the `fplll` package for reduction. There is not much to say about it, since `fplll` is essentially used as a black box with the default optional parameters that are set to produce a guaranteed result. The very small matrix built above is reduced in a blink.

3.6.2.9 Checking the Coppersmith condition

The `fplll` package has done its best to reduce the lattice basis. The corresponding vectors can be found as rows of the output matrix. But we cannot be sure that the basis has been reduced enough to comply with Coppersmith's condition. This can be easily checked when the result is still in it's matrix form. The norm we are supposed to compute over the polynomials is the 2-norm of the corresponding row vector in the matrix.

For each row r_i, we must check:

$$\|r_i\|_2 \sqrt{w} < \mathcal{N}, \text{ where } w \text{ is the cardinal of the monomials set } \Omega.$$

In our example $w = 9$ and $\mathcal{N}^{\alpha=2} = 1298074214633706907132624082305024$.

We filter out all the row vectors that match the condition. Here, 2 vectors out of 6 successfully take the test. With these "winner row vectors" (and only these), we are going to build back polynomials.

Again, numbers are too large to be output here.

3.6.2.10 Polynomials (re)building

We can first built "raw polynomial" $Rraw_m(i,t)$ using the m-th row r_m of the reduced matrix as follows:

$$\text{For each } j \in \{0, |\Omega|\},\ Rr_m = \sum r_m[j] \times \omega_j,\ \omega_j \in \Omega.$$

The second step, to produce $R_m(i,t)$ "reduced polynomials", is to remove $iBound$ and $tBound$ from the $Rraw_m(i,t)$ polynomials.

It is simply performed by making the following changes of variables:

$$i \mapsto i/iBound \text{ and } t \mapsto t/tBound,$$

The obtained polynomials are elements of $\mathbb{Z}[i,t]$ and are:

$$\begin{aligned} R_0 = \quad & 2954961\ i^4 + 166325322956931072\ i^3 - 656658\ i^2\ t \\ & + 2340480402378261472372326400\ i^2 - 18480591439659008\ i\ t \\ & + 36481\ t^2 + 439159228936488652520464318464\ i \\ & - 1733815487793463296\ t + 2060056019374801315459632621158 4 \end{aligned}$$

$$\begin{aligned} R_1 = & - 8029449\ i^4 - 127692882403393536\ i^3 + 1784322 * i^2\ t \\ & + 276602223514676564097341849 6\ i^2 + 14188098044821504\ i\ t - 99129\ t^2 \\ & - 2732752636722476956839816724 48\ i + 4459056212306034688\ t \\ & - 4998430496622579935960193879 2448 \end{aligned}$$

.

3.6.2.11 Finding integral roots of the reduced polynomials

We now have a set, R, whose elements are the polynomials $R_m(i,t)$. The roots of these polynomials over the integers are also solution of the modular equation $R_m(i,t) = 0 \mod \mathcal{N}^\alpha$.

On the one hand, we do not know if they have a common non trivial factor. On the other hand they are bivariate polynomials. One possible method to take care of both aspects together is to compute the pairwise resultant in one variable (t is the preferred eliminated variable in our context) for all the polynomials.

If two polynomials have a non trivial common root, their resultant will be null. If not, the resultant will be a univariate polynomial in i whose roots, if any, can be directly computed by applying the appropriate function in SageMath. This is the technique we use here.

First of all we form all the possible polynomial pairs among those whose coefficients comply with the Coppersmith Condition. In the general case this yields $\frac{|R|\,(|R|-1)}{2}$ pairs since $\mathrm{Res}_t(R_m, R_n) = \pm\mathrm{Res}_t(R_n, R_m)$. Out of our 2 polynomials we build only a single pair.

We only retain the pairs whose resultant (in one arbitrarily selected variable, here t) is not null.

In our example, for the single pair of polynomial, the result is null. An extra check (only performed for our exposition needs but no time is wasted at it in actual code) shows that their gcd is:

$$gcd(R_0, R_1) = -1719\ i^2 - 48378511622144\ i + 191\ t - 4538783999459328$$

This situation forces us to reconsider the bounds we are working on. We will have to start over all the process (that we will spare to the reader) with a new (narrower) interval. In our implementation, we apply a slightly more than 1/2 reduction factor. This also mean that covering all the interval will take at least two iterations instead of one.

To make a long story short, at the next iteration, we obtain 3 Coppersmith's condition compliant polynomials. All of the $\frac{3\,(3-1)}{2} = 3$ pairs of polynomials yield again null resultants in t. After several failures, we reach a point (with $iBound = 81$, remember we started at 256) where a non null resultant is found for two polynomials $R_i(i,t)$, $R_j(i,t)$ (that we do not specify further):

$$\begin{aligned} Res_t(R_i, R_j = & -643668726678221525097674985202593682333086823088156167986348694700032\ i^3 \\ & - 160917181669555381274418746300648420583271705772039041996587173675008 00\ i^2 \\ & + 563660703952118589528033984541911287619084130978298356305645551948818022 4\ i \\ & + 258026838794771984544379873643178526843182181684526074904519614895707127808 \end{aligned}$$

At the next step, we check if the resultant is a constant (has degree in $i < 1$), which is obviously

not the case here. If the resultant were a constant, we would pronounce the interval empty of hard-to-round cases, since the resultant would have not root in i. In our case, we will have to search for integral roots in i of the resultant.

An integral root exists: 168. Interestingly, it is larger than the $iBound = 81$. It is also a root of the original $Q(i,t)$ equation (remember that the obtained roots set is a superset of the $Q(i,t)$'s roots set). This is not abnormal. It only shows that the upper bound condition in Theorem 3.3 is somehow pessimistic.

The root test in polynomial $Q(i,t)$ was only done here for the needs of the exposition. In actual code, as soon as we have an i root of the resultant we directly test it for hardness-to-round. This is the process we detail below.

3.6.2.12 Analyzing a root

For that end, we map back the integral value to the floating-point number it represents in the scaled function domain (0-fbinade).

This is done in three steps:

1. change coordinates from those relative to the center of the interval to those relative to the $[\![2^{p-1}, 2^p-1]\!]$ enumeration, (here $[\![4503599627370496, 9007199254740991]\!]$), by adding the value corresponding to the interval center (here 6755399441055744) to 168;
2. transform the previous integer into a floating-point number that will live in the 0-fbinade by a division by 2^{p-1} (here 4503599627370496);
3. transform the floating-point number in the 0-fbinade into a floating-point number in its original binade (here the scaling is straightforward since the original binade is the 0-fbinade).

The outcome of all these transformations is the following value:

- 1.01111111111111111111111111111111111111111111111111001 (binary);
- 1.49999999999999844568776552478084340691566467285156 25 (decimal).

Computing, at a very high precision, the image of this number by our original function $\exp(x)$ outputs (in binary):
100.01111011010011111111100110010011111100010100111001100000000000111110011100110011101 . . .
123.4567890123456789012345678901234567890123456789012↑567890123↑567890123456789

The line of digits just beneath the number in binary is not a numerical value. It is a "bit position scale" that will help us to visualize what happens. Two tiny vertical arrows are placed, respectively,

at positions 54 and 64. One can see that past position 54, occupied by a 1-bit, starts a string of 0-bits. That string is not finished yet at position 64, occupied by a 0-bit. That signs, indeed, for our initial data, that we have found an hard-to-round case. In the implementation, the technique actually used to identify hard-to-round cases is different, more efficient if less pictorial.

This concludes the work on the current sub-interval. Since the set of roots of any pair of Coppersmith's condition compliant polynomials is a superset of that of $Q(i,t)$, no other pair needs to be examined. All we could expect from such examination is either:

- find the same root;
- find other roots that are guaranteed not to be roots of $Q(i,t)$, since they should otherwise have been found with the first pair.

Nevertheless, the considered resultant only has one root. Should it have several, one should investigate them all as hard-to-round cases candidates.

3.6.2.13 Mission accomplished

Execution of our example continues with the examination of the next sub-intervals, in order to fully cover the initial interval.

As one expects, when the next sub-interval is dealt with, the same hard-to-round case is found but, of course, for a different root since the bounds are different. This time the root lives inside the $[\![-iBound, iBound]\!]$. As anticipated as it is, this result is nevertheless comforting: the implementation does not only catch roots outside the assigned sub-interval but it can also find them inside too!

In the subsequent computations, other outcomes are possible. In several occasions, false candidates are found:

- 1.50000000000001953992523340275511145591735839843 75;
- 1.50000000000005595524044110788963735103607177734375.

This illustrates the fact that the polynomials rebuilt from the LLL-reduced basis have roots that are not roots of $Q(i,t)$.

Eventually, the program terminates with a grand total of two hard-to-round cases found:

- 1.49999999999999844568776552478084340691566467285156 25 (already mentioned, that was found first, outside of the assigned search interval that was the explored);

- 1.4999999999999560351682248438009992241859436035156 25 (that though smaller than the previous one, was found later, quietly waiting to be found in its "home sub-interval").

Anecdotally, both cases are found in the same narrow interval, when the probabilistic model predicted only 1 case for the whole interval. The reduced sample size (512 numbers!) dampens any strong conclusion one would like to draw from this example.

An exhaustive search absolutely confirms the above findings.

The total number of iterations of the algorithm is 8 since, in several occasions and especially at the beginning, there were not enough polynomials (as in iteration #1) to compute a resultant or the resultant of all the pairs of polynomials formed at some iteration was null. Interval splitting was necessary to obtain a meaningful decision about the status (free or infested of hard-to-round cases) of a given interval.

As a concluding remark about this example, we must notice that the result was obtained in 2.5 seconds (which includes a "warm up" phase until the "right" interval stride was found) whereas exhaustive search took less than a second (0.73 s).

3.7 First set of experimental results

For our experiments, and as mentioned in the example section, we have worked with the $\exp(x)$ function. One of the main reasons for this choice is that, thanks to previous work by V. Lefèvre, hard-to-round cases are known for `binary64`. The very least that one could expect from an implementation of the SLZ-algorithm was that it would find back what is already known (let alone improved performance).

Experiments were a bit tedious to perform for several reasons. The first one is that there is no clearly established relationship, once precision p is set, between the main parameters of the algorithm (α, the degree – d – of the approximation polynomials, the target hardness-to-round, the width of the intervals) in actual practice. Of course, it is perfectly clear that the width of the interval should not be larger than what the degree of the polynomial allows for in terms of approximation accuracy in order to verify the constraints dictated by the hardness-to-round one wants to check.

The consequence is that testing involves a lot of groping. To make the matter worse, when performing realistic tests is wanted (remember, in the $6p$ to $12p$ bracket), each run takes a lot of time. Under these conditions, one may have the uncomfortable feeling of groping with boxing gloves.

Tests were all performed on servers powered by dual Intel® Xeon®CPU E5-2695 v2 @ 2.40GHz, with 128 Gb of RAM.

3.7.1 The data set

We present here our first experimental data set obtained with implementation as described above. Tables 3.10 through 3.13 list the information for, respectively, a hardness-to-round of $6p$ (for the first three ones) and $8p$ (for the last one). Despite several attempts, we could not run our expected tests for $10p$. More exactly they all failed at some points, sometimes after almost a week spent in computations, because of a program crash (a segfault error).

HTRN	Degree	α	#floats	Min. r. t.	Max. r. t.	Avg. r. t.	total r. t.	total t.	#iter.
$6p$	15	3	2^{75}	6	27	19	33,519	34,856	1755
$6p$	15	4	2^{75}	57	89	60	668	678	8
$6p$	15	5	2^{75}	279	279	279	279	280	6
$6p$	15	6	2^{75}	1,370	1,370	1,370	1,370	1,374	6
$6p$	16	4	2^{77}	47	224	153	5,845	5,880	32
$6p$	16	5	2^{77}	303	303	303	303	304	1
$6p$	16	6	2^{77}	1,482	1,482	1,482	1,482	1,485	1

Table 3.10: Experimental results with `binary128` with floating-point numbers intervals centered on 3/8.
All timings are in seconds (with commas as thousands separator). The header's definitions, for the not so obvious ones, are as follows:

- HTRN: tested hardness-to-round to nearest, in bits, given here as a multiple of precision $p = 113$;
- #floats: the count of 113-bit floating-point numbers included in the searched interval;
- Min. r. t.: minimum time spent in a reduction;
- Max. r. t.: ditto, for the maximum time;
- Avg. r. t.: average time for one reduction;
- total r. t.: total time spent in reductions;
- total t.: total time spent in search;
- #iter.: the number of iteration (reductions) to span all the interval; some of them end up in failures (not enough polynomials nor any valid pairs).

HTRN	Degree	α	#floats	Min. r. t.	Max. r. t.	Avg. r. t.	total r. t.	total t.	#iter.
$6p$	17	3	2^{80}	Too many interval splits, giving up after 78 hours spent in computations.					
$6p$	17	4	2^{80}	53	254	237	66,372	66,644	280
$6p$	17	5	2^{80}	290	738	457	5,030	5,045	8
$6p$	17	6	2^{80}	1,642	1,642	1,642	1,642	1,646	1
$6p$	17	7	2^{80}	6,483	6,483	6,483	6,483	6,496	1
$6p$	18	5	2^{82}	313	2,023	1,327	51763	51,825	39
$6p$	18	6	2^{82}	1,771	2,155	1,844	11,069	11,090	6
$6p$	18	7	2^{82}	7,028	7,028	7,028	7,028	7,043	1
$6p$	18	8	2^{82}	24,111	24,111	24,111	24,111	24,154	1

Table 3.11: Main experimental results with `binary128` (cont'd). Headings and conventions are the same as in Table 3.10.

HTRN	Degree	α	#floats	Min. r. t.	Max. r. t.	Avg. r. t.	total r. t.	total t.	#iter.
$6p$	19	7	2^{84}	7,078	9,431	8,009	48,055	48,119	6
$6p$	19	8	2^{84}	26,164	25,836	25,944	77,833	77,923	3
$6p$	19	9	2^{84}	75,169	75,169	75,169	75,169	75,287	1
$6p$	19	10	2^{84}	195,906	195,906	195,906	195,906	196,195	1
$6p$	20	7	2^{84}	7,425	10,085	8,526	51,159	51,228	6
$6p$	20	8	2^{84}	27,502	27,635	27,619	82,858	82,956	3
$6p$	20	9	2^{84}	80,148	80,148	80,148	80,148	80,278	1
$6p$	20	10	2^{84}	208,762	208,762	208,762	208,762	209,117	1

Table 3.12: Main experimental results with `binary128` (cont'd). Headings and conventions are the same as in Table 3.10.

HTRN	Degree	α	#floats	Min. r. t.	Max. r. t.	Avg. r. t.	total r. t.	total t.	#iter.
$8p$	30	7	2^{87}	20,118	26,189	22,513	135,083	135,448	6
$8p$	30	8	2^{87}	71,254	71,254	71,254	71,254	71,363	1
$8p$	30	9	2^{87}	202,280	202,280	202,280	202,280	202,549	1
$8p$	30	10	2^{87}	524,858	524,858	524,858	524,858	525,485	1
$8p$	31	8	2^{88}	69,707	73,365	72275	216,826	217,056	3
$8p$	31	9	2^{88}	210,906	210,906	210,906	210,906	211,196	1

Table 3.13: Main experimental results with `binary128` (cont'd). Headings and conventions are the same as in Table 3.10.

As it can be seen, our tests concentrated on an hardness-to-round of $6p$ (678 bits). The reason is obvious: our incursions into the $8p$ realm was chastised with a tenfold execution time penalty. We did not overly insist. However scanty they are, $8p$ data give nevertheless some interesting indications.

Before we try to give an analysis of the data, we must first pay tribute to the rock-steady stability of `fpLLL`. As can be seen, some runs last more than two days, with large volumes of data. At high-degree/high-alpha, the typical size of the file holding the matrix to reduce, expressed in decimal numbers, is several hundreds of megabytes. And the larger the data set, the longer the execution time is. As stated before we experienced segfaults when running reductions for hardness-to-round of $10p$ after more than a week of execution time. Which element of the software chain should take the blame for these failures remains unclear.

3.7.1.1 Reduction grabs the lion's share of execution time

Without any question, reduction is the main contributor, by a wide margin, to the execution time of the SLZ-algorithm, at least in our implementation. This is why we put so much emphasis on reduction in our tables. This is especially true for our search domain ($6p$ and $8p$ hardness-to-round). For smaller precisions (e.g. `binary64`) and the corresponding hardness-to-round, reduction execution time can be compared with that of other parts of the algorithm (e.g. polynomials computation and rounding, resultants computation).

This situation highlights the importance of the construction rules of the initial lattice. We believe that much remains to be done in this area. As preliminary and not reported here results show, shifting from the initial formulation of the basis construction from that found in [102] to that of [100] was very useful move in this respect. But still, an elucidation of the optimal lattice basis

building rule for a given problem structure could bring in great improvements for the reduction of execution time. It would, at the very least, lift the doubts we still have in this respect.

3.7.1.2 The weird reduction time drift

For a given degree, the width of the approximation interval is initially constrained by the accuracy of the approximation. We make sure the interval is narrow enough so that a single polynomial of the chosen degree can approximate the function with the necessary accuracy.

As it turns out, when a low value is used for α, one can stumble on another problem: the reduction does not yield two short enough norm vectors (and free of any non-trivial common factor) to be able perform the hunt for roots. In this case, we have to break the interval in two and carry on with the smaller sub-interval. Of course, we may be forced to make several of those splits in cascade until the interval is small enough in order to obtain the needed couple of polynomials. Our experience is, once a satisfactory interval size is found, that we can carry on with it for further checks, at least for a limited part of the domain of the function. This has been analyzed in detail in §3.5.3.2, p. 188, for the $\exp(x)$ function. We systematically attempted to enlarge interval size and those attempts have systematically failed as well: even for a minimal stretching factor over the "ideal" interval width, the short-norm vectors search fails again. Reductions are then successfully [34] performed on smaller intervals of growing width until the initial width is fully covered.

When multiple such searches are performed they string together inside the same system process. We experience then a lengthening of reductions execution time from the first to the last ones in a somewhat jagged pattern but with a clear growth trend. That can be seen from our minimum and maximum reduction timings. The former are generally observed at the beginning of the execution while the latter are at the end.

When really many (over a few dozens) reductions are preformed, the drift eventually stabilizes at some relatively high value (compared with the initial ones). This has nothing to do with a real reduction time increase along the interval: when one of these last intervals is processed directly (without any forerunner), its reduction times match that of the first intervals. Our best guess is that the tools have to be regarded as responsible for that reduction time drift but we could not pinpoint (our own programming, Python/SageMath, fpLLL?) where put the blame on. This lays a shade of suspicion on our comparisons between one time reduction over a single "large interval" and aggregated reductions over smaller ones.

[34] In the sense that they yield short-norm vectors and those are free from non-trivial common factor.

3.7.1.3 The "sweet spot" effect

We have not represented all of our testing to spare place and reader's attention. For degree-17, degree-18 and $6p$, the best explored area, our data is presented around a particular case ($d = 17$, $\alpha = 6$; $d = 18$, $\alpha = 6$). It has two main characteristics:

- it is the smallest α for which we do not need to split the interval because of a Coppersmith's condition failure (two valid short-norm vectors are found);
- the execution time is minimal.

Indeed, for a smaller value of α, the interval has to be split. The aggregated time of the multiple necessary searches is larger (even if we set aside the "time drift" effect) than that of a single, wider search. On the other hand, using a larger value of α, which considerably grows the volume of the input data for *LLL*-reduction, only leads to a vastly increased reduction time. To our dismay, we are not able to give an informed explanation to this phenomenon.

Our only conclusion so far, from a practical point of view, is that, if one is to use this algorithm, she must look for the "sweet spot" the parameters and perform all her computations with this setting. For degree-19 and degree-20, we are inclined to think that we have not worked out the sweet spot yet. Apparently, the effect shows up again at degree-30 and $\alpha = 8$ for a $8p$ hardness-to-round.

3.7.1.4 Higher degrees do not fare better

For a given value of α, once an optimum in polynomial degree is reached (see §3.7.1.3, p. 224) increasing the degree of the polynomial does not help. On the one hand higher degree polynomials allow for larger intervals (from the approximation standpoint). But, on the other hand, the search process fails when searching for small-norm, non-trivial common factor free polynomial pairs on such wider interval. This forces us to come back to an interval width that is the same as that we used for our optimal set of parameters.

It also turns out that polynomial with higher degrees yield larger basis data size that causes a time penalty. From our degree-17 and $\alpha = 6$ best performer case, moving to higher degrees does not show any improvement. There is no real gain in interval width and should there be some, it would be overweighted by the increase in execution time. Hence, when α is set, it is very important, for optimal performance, to pick the "right" polynomial degree.

3.7.1.5 Larger values of hardness-to-round are easier to check

This result was asymptotically expected from theoretical results. Experiments seem (considered to the limited number of tests, we cannot be absolutely conclusive on this one) to follow the trend for

the small values (small compared with p^2 for which property is proved, see [99], Theorem 27) we use. Our best scorer in $6p$ can process a 2^{80} number interval in 1,646 s. Its sibling in $8p$ performs the same task on 2^{87} numbers in 71,254 s. The ratio between the two intervals is $2^7 = 128$. The speedup is almost 3 in $8p$ and we can not discard the possibility that even better timings are obtained for this hardness-to-round value.

3.8 Accelerating operations lattice basis: from reduction to transformation

Our above detailed example and initial experimental results already raised some doubts on the capacity of the initial implementation to get the work done in a reasonable time span. Those doubts are solely related with execution time, since we could recover the results found with the L-algorithm (using small intervals around the already known hard-to-round cases) or meet other challenges, as the one we set in §3.6.2, p 203, where we controlled the behavior of our implementation with exhaustive search over a necessarily limited-width interval.

We could try to comfort ourselves by thinking that the execution time data of the detailed example are not relevant since our test parameters placed us in a situation where the SLZ-algorithm is at a disadvantage with other techniques. Nevertheless, the first test in 113-bit precision and with a hardness-to-round of $8p$ cleared up our doubts and confirmed that, even in better conditions, the execution time was way too long. All the timings are given in the previous section: with our best performer, it would take 224,310 years to explore a full binade on a single processor. Even if our problem is embarrassingly parallel and its processing can be split in smaller tasks executed independently, the exploration of a single binade could be intractably long to perform.

Where is the bottleneck? Without any debate, in lattice reduction. From our first experimental data, lattice reduction always represents more than 99 % of the execution time. These lengthy reductions also set another problem such as the actual possibility to deal with a "large" hardness-to-round. In our context "large" means $10p$ and above. In most occasions, with these parameters, reduction crashed before completion after running for rather long time spans (more than a week). In other occasions we faced a spectacular memory ballooning that threatened the stability of the (shared) system we used. We had to abort the execution in order to avoid a memory swapping that would ultimately have swept away all our performance hopes.

At this point, "Can reduction time be shrunk?" is a legitimate question.

3.8.1 Two (momentarily?) discarded options

A first approach could have been to search for "tactical workarounds" for our problem such as:

- find a faster reduction library;
- call the library with different parameters.

We have felt that there was no point in changing software. The `fpLLL` library used in SageMath is already one of the quickest implementations around. Switching to another library would have possibly given us, at best and being optimistic, a factor 2 in time reduction: not a radical change at that point.

Another option could be to use different parameters in the `fpLLL` invocation. As we have seen, we have used the default ones that ensure that we get an LLL-reduced basis. But, if we think about it, what we actually need is a basis that has some (at least 2) sufficiently small-norm vectors to comply with the Coppersmith's condition. We LLL-reduce our initial basis with the expectation that the output will provide us with such vectors and that, furthermore, those will be the first ones that are returned. One can imagine that calling the `fpLLL` library with less aggressive parameters would be sufficient to fulfill our objective. We gave this option a lazy[35] try. The initial tests did not lean us to think that we could obtain a major acceleration factor that could completely change the picture.

It does not mean that this optimization leads are discarded forever. All we say here that we view them as useful second line options. This leaves us with more "strategic moves": interfere with the reduction process itself, keeping in mind that what we actually want are small-norm vectors.

3.8.2 Reducing the Gram matrix

The matrices that define the lattices we want to reduce have a very eye-catching property: they are very "rectangular". In other words, the dimension of the lattice is small compared to its degree. More precisely, and with the building rules presented in §3.5.2.3, p. 182, the number of rows (polynomials) of the matrix is given by formula $\frac{(\alpha+1)(\alpha+2)}{2}$ while the number of columns (monomials) is computed with formula $\frac{(\alpha+1)(d\alpha+2)}{2}$, where d is the degree of the polynomial. For our best performing set of parameter hardness-to-round $8p$ ($\alpha = 8$, $d = 30$) the matrix has 45 rows for 1089 columns.

[35] We must confess here that we did not make a systematic and thorough investigation.

3.8.2.1 Defining a lattice by its Gram matrix

We already met the Gram matrices when we defined the volume of a lattice in §3.4.4, p 170. It turns out that this object is connected to deeper aspects of lattice theory than only being a convenient mean to be able to compute a volume-like magnitude for lattices the defined by rectangular matrices.

One can find an in-depth and yet accessible exposition in [16] Chap. 2, §2.5. In this theoretical perspective, the Gram matrix is a more fundamental object than a basis for the definition of a lattice. For instance, several different bases of the same lattice can yield the same Gram matrix.

More precisely, if from two bases, $\boldsymbol{b}_1$, and $\boldsymbol{b}_2$ we form the $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$ such that:

- $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$ define the same lattice,
- $\boldsymbol{B}_1 \times \boldsymbol{B}_1^\intercal = \boldsymbol{B}_2 \times \boldsymbol{B}_2^\intercal = \boldsymbol{G}$, where $\boldsymbol{G}$ is a Gram matrix [36],

then $\boldsymbol{B}_1 = \boldsymbol{K} \times \boldsymbol{B}_2$, where $\boldsymbol{K}$ is an orthogonal matrix.

In other words, the Gram matrix $\boldsymbol{G}$ of a lattice with basis $\boldsymbol{b}$ determines this basis uniquely up to isometry.

It comes from the above that a Gram matrix is a very natural way to define a lattice and that the algorithms to perform lattice reduction from a Gram matrix are very similar to those used from a basis-built matrix. There are theoretical arguments to support the claim of a greater efficiency of this course of action (see [16] p. 89, Remark 2).

It also comes that the result of the reduction of a lattice defined by a Gram matrix, cannot be, as in the case of a lattice defined by a basis, a new (reduced) basis. Here the output of the algorithm is a transformation matrix $\boldsymbol{T}$ such that

$$\boldsymbol{T}^\intercal \times \boldsymbol{B} = \boldsymbol{B}_{LLL-r}$$

where $\boldsymbol{B}$ is a matrix formed from a basis $\boldsymbol{b}$ defining a lattice $\boldsymbol{L}$, and $\boldsymbol{B}_{LLL-r}$ is formed from the LLL-reduced basis $\boldsymbol{b}_{LLL-r}$ of the same lattice [37].

[36] We have swapped the order of the matrices in the products since Cohen's exposition is based on matrices formed with the vectors of the basis as column vectors and we stick to our initial choice of row-vector based matrices.

[37] Again, matrix transposition and order of multiplication depend on the choice among the columns/row alternatives for the presentation.

3.8.2.2 Gram matrix dimensions

Our interest in the Gram matrix was not initially triggered by the aforementioned theoretical advantages in the reduction process. The extent to which they materialized in actual implementations remained unclear. But what was immediately and particularly appealing was that the Gram matrix opened the possibility to work with more amenable objects than our initial extraordinarily stretched matrices.

Indeed from a, say, 45×1089 initial matrix we could get a 45×45 Gram matrix. Of course, this reduction in matrix dimension did not come for free: the bit size of the matrix elements was vastly extended (essentially by a factor 2). Matrix multiplications also took their toll on performance when the global process was considered but, in actual tests, it was negligible.

3.8.2.3 A dead end

The result in actual tests, replacing the straight basis reduction by the above sketched process in our implementation, were not particularly rewarding. We do not give a detailed record of these experiments here since the timings were within 5 to 10 % of our initial results.

We insist here on the fact that going through the Gram matrix process also delivered an *LLL*-reduced basis. The output was not essentially different than that we initially obtained with basis reduction alone.

3.8.3 Random projections and projected matrices reduction

Does the lack of improvement noticed when reducing the Gram matrix negatively seal the faith of our effort to squeeze the matrix? No, since another path has been opened by a work of A. Akhavi and D. Stehlé [1] published in 2008.

3.8.3.1 A bird's eye view of the process

This global description does not settle all the issues that will be dealt with in subsequent sections. The general scheme is essentially the same as the one described in the previous section:

1. start with a basis $\boldsymbol{b}$ of some lattice $\boldsymbol{L}$ made of m vectors of n elements, with $n \geqslant m$;
2. from $\boldsymbol{b}$ build a $m \times n$ matrix $\boldsymbol{B}$ (we remind our convention that the vectors of $\boldsymbol{b}$ are the rows of $\boldsymbol{B}$);
3. from $\boldsymbol{B}$ derive some $m \times m$ square matrix $\boldsymbol{B}_p$;

4. perform a LLL-reduction of $\boldsymbol{B}_p$;

5. this operation does not only yield some matrix $\boldsymbol{B}_{pr}$, corresponding to a LLL-reduced basis we do not care about, but also a $m \times m$ transformation matrix T such that $T \times \boldsymbol{B}_p = \boldsymbol{B}_{pr}$;

6. perform $T \times \boldsymbol{B} = \boldsymbol{B}_t$ where $\boldsymbol{B}_t$ is a $m \times n$ matrix; from $\boldsymbol{B}_t$ straightforwardly extract the basis $\boldsymbol{b}_t$;

7. we can even LLL-reduce the matrix $\boldsymbol{B}_t$ to obtain the matrix $\boldsymbol{B}_{tr}$ since, as we shall see, the reduction of the former full $m \times n$ matrix is much shorter than a reduction from scratch because the "preparatory work" realized by the multiplication $T \times \boldsymbol{B}$ is a very efficient step towards a "full" LLL-reduction.

The last step differs from that of the Gram matrix reduction technique. Whereas the transformation matrix obtained in this case allowed for the computation of an actually LLL-reduced basis, this is not the case anymore with $\boldsymbol{T}$: $\boldsymbol{b}_t$ is not guaranteed to be LLL-reduced anymore. But as expressed in different occasions, we are not interested in LLL-reduction *per se*. All we want are short-norm vectors in the basis.

But wait a minute, how can we ensure that $\boldsymbol{b}_t$ is a basis and that it defines the same lattice as the one defined by the basis $\boldsymbol{b}$?

3.8.3.2 More reminders on lattices

The above question forces us to extend a bit our reminders. As they we only be used here, will feel it relevant to introduce them only at this point of our exposition. Let us start with a classical definition and lemma.

Definition 3.4. A matrix $U \in \mathbb{Z}^{m \times m}$ is called *unimodular* if $\det U = \pm 1$.

A classical result is the following lemma.

Lemma 3.10. *If $U \in \mathbb{Z}^{m \times m}$ is unimodular, then U^{-1} is also unimodular.*

Saying that U^{-1} is unimodular also conveys the idea that $U^{-1} \in \mathbb{Z}^{m \times m}$.

A more lattice-specific and also classical result is the following lemma.

Lemma 3.11. *Let two matrices $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$ derive respectively from the two bases $\boldsymbol{b_1}$ and $\boldsymbol{b_2}$. They define the same lattice $\boldsymbol{L}$, if and only if $\boldsymbol{B}_1 = U \times \boldsymbol{B}_2$ for some unimodular matrix U.*

The demonstration of this lemma can be found, for instance, in [16].

The transformation matrix output by the `fplll` implementation we use is precisely such a unimodular matrix.

It comes from the above that the matrix T is unimodular. It follows that the set of vectors $\boldsymbol{b_t}$ computed at step (6) is a basis that defines the same lattice $\boldsymbol{L}$ as $\boldsymbol{b}$.

Let us enter into more details about the majors steps of the process.

3.8.3.3 Reducing the number of columns: matrix projection

Having anticipated the shortcoming of Gram's matrix reduction (in the Gram matrix, elements have essentially a doubled bit-size compared to those of a matrix formed with a basis), A. Akhavi and D. Stehlé have devised another method that tries to avoid this flaw. The first idea is to use a projecting matrix whose left multiplication by the initial matrix will yield a square matrix.

Let us call $\boldsymbol{B}$ the $m \times n$ with $n \geqslant m$, the matrix built from a basis of the lattice. We remind here our row-vector construction convention from the vectors of the basis (m is the dimension of the lattice, n is the degree). Let us also call P an $n \times m$ projecting matrix[38].

We obviously have:

$$\left[\begin{array}{c} \\ \boldsymbol{B} \\ \\ \end{array}\right] \times \left[\begin{array}{c} \\ P \\ \\ \end{array}\right] = \left[\begin{array}{c} \\ \boldsymbol{B}_p \\ \\ \end{array}\right]$$

where the *projected matrix* $\boldsymbol{B}_p$ is an $m \times m$ square matrix.

If P is built in such a way that its elements have a small bit-size, that of the elements of $\boldsymbol{B}_p$ will not be much larger than that of the elements of $\boldsymbol{B}$.

[38] We resist the temptation of using the term *projector* here since it has precise technical meaning that does not fit with what the matrix P actually is.

But then two other issues immediately pop up:

- how do we pick the elements of the projecting matrix P (beyond our intention to have small bit-sized elements)?
- are $\boldsymbol{B}_p$ and the LLL-reduction of $\boldsymbol{B}_p$ relevant to our problem with the basis $\boldsymbol{B}$?

3.8.3.4 How do we use the results of A. Akhavi and D. Stehlé?

Our starting point is the situation we are in after §3.7:

- our timings are unbearably long;
- LLL-reduction is not an end but a mean to get short-norm vectors.

We see the A. Akhavi and D. Stehlé proposition as a possible option that, given their experimental results, deserve more than a casual try in order to exit from the performance pit were stuck in.

Of course we have taken into account their analysis about complexity and the quality of the transformed basis. On the latter we have also noticed that, in the available form of the paper which is dubbed as "an extended abstract", continuous random models are used and that the full analysis involving the effect of quantization is left for the "full version".

We then use their method as an oracle to obtain a basis with some short-norm vectors.

We validate afterward the obtained data with our own criteria:

- as for the complexity: the actual improvement in timings;
- as for the quality of the basis: our yardstick is the Coppersmith condition.

3.8.3.5 Building a random projecting matrix

What is proposed in [1] is several probabilistic models to pick the elements of the projecting matrix P. We will not enter into more details about these models here.

This discussion would have been relevant for us if one model had shown a clear advantage over the others. This is not the case in the experimental results presented in [1].

We have picked for our own tests two simple models:

- one where the elements of the matrix a uniformly and independently picked from $\{-1, 1\}$;

- another where the entries are uniformly and independently picked from $\{-2^N, ..., 0, ...2^N - 1\}$, where N is a small positive integer (typically, the experimental results of [1] where computed for $N = 3$ and $N = 10$; we used similar values).

As one can see, in both cases, our transformation matrix elements are integers.

But what is not sure yet is that $\boldsymbol{B}_p$ is still a basis. Actually, this will be the case with a high probability. If not, this will occasion a problem and it will be detected during LLL-reduction causing its failure. All we have to do is to gracefully recover from this misstep, compute another projection matrix P with the expectation that this time the product $\boldsymbol{B}_p \times P$ will actually yield a basis.

The situation where $\boldsymbol{B}_p$ does not represent a basis has never happened in any of our tests. We had to make up artificial cases to test the behavior of `fpLLL` (linear independence of the rows of the matrix is checked when a Gram-Schmidt orthogonalization is performed) and check the handling of the generated exceptions in our implementation.

3.8.3.6 The implementation

We only give here a few elements of the implementation of the matrix projection and its behavior. More extensive experimental results will be given in a section to come.

As explained above the projecting matrix P is formed by picking its elements in $\{-1, 1\}$ or in $\{-2^N, ..., 0, ...2^N - 1\}$, with $N \in \{4, 8, 12\}$. To make a long story short, we did not notice any real improvement when switching from the first distribution to the second. As for the second, we observed instead a performance degradation as N was growing larger. Nevertheless a more systematic exploration of values for $N \leqslant 4$ may possibly help to scrap extra bits of execution time.

Using the same notations as above, we compute $\boldsymbol{B} \times P = \boldsymbol{B}_p$ where $\boldsymbol{B}_p$ is the (square) projected matrix.

The function of the `fpLLL` library we use to perform reductions does not only compute an LLL-reduced basis but it also returns, at almost no extra cost, the transformation $m \times m$ matrix T that when right multiplied by the original matrix (here $\boldsymbol{B}$) yields the reduced matrix:

$$T \times \boldsymbol{B}_p = \boldsymbol{B}_{pr},$$

where $\boldsymbol{B}_{pr}$ is the (square) LLL-reduced projected matrix.

To trigger the return of the transformation matrix T by the LLL-reduction function in SageMath, we have, instead of just calling with the matrix to reduce as an argument, to invoke it with an augmented matrix crafted as follow :

$$\left[\begin{array}{c|c} I & \boldsymbol{B}_p \end{array}\right],$$

where I is an $m \times m$ identity matrix. The final result is an $m \times 2m$ matrix that will be the actual argument for the `fpLLL` call.

After reduction is performed, the matrix returned by `fpLLL` is laid out as follows

$$\left[\begin{array}{c|c} T & \boldsymbol{B}_{pr} \end{array}\right],$$

such that $T \times \boldsymbol{B}_p = \boldsymbol{B}_{p,r}$.

The $\boldsymbol{B}_{pr}$ matrix is of no use to us, it is indeed the transformation matrix T that interests us. We are going to left multiply our initial matrix by T with the hope that $T \times M$ will be a matrix that corresponds to a basis of our lattice that has vectors with a small enough norm to comply with Coppersmith's condition.

$$\left[\begin{array}{c} T \end{array}\right]\left[\begin{array}{c} \boldsymbol{B} \end{array}\right] = \left[\begin{array}{c} \boldsymbol{B}_t \end{array}\right],$$

where $\boldsymbol{B}_t$ is the transformed matrix. We call it "transformed" rather than "reduced" since it is not LLL-reduced and a simple check (performed here for the sake of completeness but not in actual code) with the `is_LLL_reduced` function of `fpLLL` immediately confirms it. But, as it has been written in different occasions, it is not LLL-reduction *per se* that interests us but short-norm vectors.

We leave now the realm of the reduction (or transformation acceleration) to look at another optimization possibility. We have often referred to the Coppersmith condition as the pythonissa that blessed or cursed our short-norm vectors candidates. But can we challenge the form we gave it in our initial presentation in §3.3.7, p. 164 to improve the efficency?

3.8.4 Changing the Coppersmith condition

What we are going to deal with now cannot be strictly aggregated under the umbrella of the acceleration of operations on lattice basis. But it is so strongly connected to this question and pertains so tightly to the enhancements to the algorithm topic that we feel it relevant to discuss it here.

This implementation is based on Theorem 3.3. We have used it to obtain the experimental results given below. Nevertheless, as has been seen in our detailed example, the bounds given are not absolutely tight since we could find a hard to round case outside the interval is was supposed to live in.

3.8.4.1 The new condition

D. Stehlé was first to remark that the analysis given in the proof of Theorem 3.3 could be somehow simplified and yield a different condition on the small-norm vectors, involving a change in the considered norm.

Let us quickly remind the context of Theorem 3.3:

- we have polynomial $h(i,t) \in \mathbb{Z}[i,t]$, made of (at most) w monomials, such that $h(i,t) = \sum h_{\rho,\sigma}\, i^\rho\, t^\sigma$
- we have a positive integer constant $\mathcal{N}$;
- two integers, i_0 and t_0, are such that
  - $h(i_0,t_0) \bmod \mathcal{N} = 0$,
  - $|i_0| < I$ and $|t_0| < T$, with $I \in \mathbb{N}$ and $T \in \mathbb{N}$;

Initially, Theorem 3.3 states that if $||h(iI,tT)|| < \dfrac{\mathcal{N}}{\sqrt{w}}$ then $h(i_0,t_0) = 0$ also stands over the integers.

We claim here that, under the above listed conditions, if $|h(iI,tT)| < \mathcal{N}$ then $h(i_0,t_0) = 0$ also stands over the integers.

Proof. We start from the same point as for the proof of Theorem 3.3 but quickly fork to a different

direction

$$|h(i_0,t_0)| = \left|\sum h_{\rho,\sigma}\, i^\rho\, t^\sigma\right|,$$
$$\left|\sum h_{\rho,\sigma}\, i^\rho\, t^\sigma\right| \leqslant \sum |h_{\rho,\sigma}|\,|i^\rho|\,|t^\sigma| \leqslant \sum |h_{\rho,\sigma}|\, I^\rho\, T^\sigma = |h(iI,tT),$$
$$|h(i_0,t_0)| \leqslant |h(iI,tT)| < \mathcal{N} \text{ (from our initial conditions).}$$

Hence

$$|h(i_0,t_0)| < \mathcal{N}$$

Since

$$h(i_0,t_0) = 0 \bmod \mathcal{N}.$$

We eventually have

$$h(i_0,t_0) = 0.$$

□

3.8.4.2 How well does this condition behave in our case?

What is expected from this new condition ($|h(iI,tT)| < \mathcal{N}$) is that $|h(iI,tT)|$ will be smaller than $||h(iI,tT)||\sqrt{w}$ in order to filter out more polynomials. The assumption here is that among these polynomials we will find more non trivial common factor free pairs and that we will be able to avoid some interval splitting, speeding up the whole process.

But, probably due to the peculiar shape of the polynomials we use, and after many tests, the $|h(iI,tT)| < ||h(iI,tT)||\sqrt{w}$ condition was never satisfied. Since our interval splitting policy is very aggressive (division by 2 when not enough condition compliant polynomials are found) we did not notice any real difference in overall execution time. Using what happens to be a more stringent condition did not cause extra splitting. It eventually happened as often with both conditions.

It also remains to be seen, when the $|h(iI,tT)| < ||h(iI,tT)||\sqrt{w}$ condition stands and more polynomials are filtered out, if we can find, among those, pairs of non trivial common factor free ones. If no such pairs are found, as already explained, all we can do is shrink the interval and run a new iteration. It turns out that what is gained at Coppersmith's condition checking stage could be lost at the roots finding one. In other words, does the use of a different condition affect the "quality" of the selected polynomials?

Unfortunately we cannot answer this question since our test cases never gave us a chance to investigate it. But this does not imply that there is no point, in other contexts, in using this

modified condition. To the moment, we consider it as an open option. When using the Coppersmith method, one may give it a try and check if, in her case, this condition gives better results (more "interesting" polynomials) than the "classical one".

3.9 Projection in action: a second set of experimental results

This second set is computed using projection as explained in §3.8.3. The conditions are otherwise identical (platform, function $\exp(x)$) as those under which our initial data were observed.

The experimental area has been extended since, from the quicker execution time, we had the opportunity to explore a broader parameter space. We could lead to a normal ending computations that collapsed in our first test round.

The projection technique has also allowed for the completion of some extra computations that systematically failed without it ($10p$ and over).

Two sets of results are listed in Table 3.14 through Table 3.18.

HTRN	Degree	α	#floats	Min. r. t.	Max. r. t.	Avg. r. t.	total r. t.	total t.	#iter.
$6p$	15	3	2^{75}	6.0	27.0	19.0	33,519	34,856	1755
$6p$	15	4	2^{75}	1.4	3.2	2.4	19.0	31.0	8
$6p$	15	5	2^{75}	3.9	4.9	4.5	27.0	39.1	6
$6p$	15	6	2^{75}	11.0	14.8	12.7	76.0	102.0	6
$6p$	16	4	2^{77}	0.9	2.4	1.8	56.0	89.0	32
$6p$	16	5	2^{77}	2.6	2.6	2.6	2.6	3.8	1
$6p$	16	6	2^{77}	8.7	8.7	8.7	8.7	12.1	1

Table 3.14: Main experimental results with `binary128` with floating-point numbers intervals centered on 3/8.
All timings are in seconds (with commas as thousands separator). The header's definitions, for the not so obvious ones, are as follows:

- HTRN: tested hardness-to-round to nearest, in bits, given here as a multiple of precision $p = 113$;
- #floats: the count of 113-bit floating-point numbers included in the searched interval;
- Min. r. t.: minimum time spent in a reduction;
- Max. r. t.: ditto, for the maximum time;
- Avg. r. t.: average time for one reduction;
- total r. t.: total time spent in reductions;
- total t.: total time spent in search;
- #iter.: the number of iteration (reductions) to span all the interval; some of them end up in failures (not enough polynomials nor any valid pairs).

HTRN	Degree	α	#floats	Min. r. t.	Max. r. t.	Avg. r. t.	total r. t.	total t.	#iter.
$6p$	17	3	2^{80}	Crashes after about 48 hours spent in computations, with $2^{64.5}$ floats intervals.					
$6p$	17	4	2^{80}	0.8	3.3	1.9	528.6	809.2	280
$6p$	17	5	2^{80}	3.4	4.3	3.9	31.2	45.2	8
$6p$	17	6	2^{80}	7.3	7.3	7.3	7.3	11.7	1
$6p$	17	7	2^{80}	28.3	28.3	28.3	28.3	41.4	1
$6p$	18	5	2^{82}	2.3	6.1	4.1	158.9	220.9	39
$6p$	18	6	2^{82}	8.7	10.6	9.5	56.9	77.5	6
$6p$	18	7	2^{82}	27.0	27.0	27.0	27.0	41.0	1
$6p$	18	8	2^{82}	86.3	86.3	86.3	86.3	126.6	1

Table 3.15: Main experimental results with `binary128` (cont'd). Headings and conventions are the same as in Table 3.14.

HTRN	Degree	α	#floats	Min. r. t.	Max. r. t.	Avg. r. t.	total r. t.	total t.	#iter.
$6p$	19	7	2^{84}	24.8	33.0	28.0	168.2	231.2	6
$6p$	19	8	2^{84}	75.7	91.5	81.9	245.6	333.2	3
$6p$	19	9	2^{84}	232.9	232.9	232.9	232.9	345.5	1
$6p$	19	10	2^{84}	747.4	747.4	747.4	747.4	1040.7	1
$6p$	20	7	2^{84}	28.8	35.2	29.7	178.0	249.0	6
$6p$	20	8	2^{84}	93.70	105.4	98.6	295.9	396.8	3
$6p$	20	9	2^{84}	262.0	262.0	262.0	262.0	391.2	1
$6p$	20	10	2^{84}	743.0	743.0	743.0	743.0	1052.1	1

Table 3.16: Main experimental results with `binary128` (cont'd). Headings and conventions are the same as in Table 3.14.

HTRN	Degree	α	#floats	Min. r. t.	Max. r. t.	Avg. r. t.	total r. t.	total t.	#iter.
$8p$	30	7	2^{87}	59.5	74.2	65.2	391.1	550.2	6
$8p$	30	8	2^{87}	177.7	177.7	177.7	177.7	285.4	1
$8p$	30	9	2^{87}	525.9	525.9	525.9	525.9	798.8	1
$8p$	30	10	2^{87}	1447.3	1447.3	1447.3	1447.3	2076.5	1
$8p$	31	8	2^{88}	218.1	255.9	239.4	718.0	955.2	3
$8p$	31	9	2^{88}	525.9	525.9	525.9	525.9	877.9	1

Table 3.17: Main experimental results with `binary128` (cont'd). Headings and conventions are the same as in Table 3.14.

HTRN	Degree	α	#floats	Min. r. t.	Max. r. t.	Avg. r. t.	total r. t.	total t.	#iter.
$10p$	44	9	2^{92}	998.2	1,253.7	1,133.2	4,532.6	6,874.6	4
$10p$	44	10	2^{92}	2,849.3	2,849.3	2,849.3	2,849.3	4,158.8	1
$10p$	44	11	2^{92}	7,923.1	7,923.1	7,923.1	7,923.1	10,548.6	1
$10p$	44	12	2^{92}	15,654.0	15,654.0	15,654.0	15,654.0	20,574.3	1
$10p$	44	13	2^{92}	39,808.5	39,808.5	39,808.5	39,808.5	48,883.1	1
$10p$	45	10	2^{93}	2,634.4	2,798.8	2,716.6	5,433.2	8,097.9	2
$10p$	45	11	2^{93}	7,732.1	7,732.1	7,732.1	7,732.1	10,471.9	1
$10p$	45	12	2^{93}	15,467.0	15,467.0	15,467.0	15,467.0	20,591.8	1
$10p$	46	10	2^{93}	2,808.2	2,914.2	2,861.2	5,722.4	8,583.1	2
$10p$	46	11	2^{93}	7,040.0	7,040.0	7,040.0	7,040.0	9,819.8	1
$10p$	46	12	2^{93}	16,400.7	16,400.7	16,400.7	16,400.7	21,815.0	1

Table 3.18: Main experimental results with `binary128` (cont'd). Headings and conventions are the same as in Table 3.14.

We analyze here our second set of results obtained with an the improved transformation technique described in §3.8.3, p. 228. In general terms, these results are, from a practical perspective, a powerful stimulus to persist in the optimization tracks we are on. At the same time, at a more conceptual level, they rise more issues than they solve.

For our $10p$ experiments and specifically for the higher degrees, we run our tests for the maximal interval width for which we did not incur a failure in non-trivial common factor free polynomial

pairs search. When this kind of failure happens, we have to split the interval and run our search on the sub-intervals. This situation calls for the following comments:

- we have noticed that when an experiment is run with a set of parameters that force interval splitting, we can find a different set of parameters that, either, avoids it entirely or runs faster on the sub-interval; in both cases the aggregated time is smaller;

- we also experienced lots of unexplained program crashes when interval splitting was involved; this is peculiar to $10p$ experiments and does not happen in $8p$ or $6p$ or only if a very large number (> 1000) of intervals is involved.

3.9.1 Substantially reduced lattice basis transformation timings

We speak here about lattice basis transformation and not reduction since, as noted, the obtained basis is not necessarily LLL-reduced. Reduction in transformation timings is of course the most striking difference between the two sets of data. This decrease can amount to a factor 300 in most favorable cases (for a hardness-to-round of $8p$). It could perhaps have been even larger if we had been able to succeed in the "full" LLL-reductions for hardness-to-round $10p$.

Decreasing the time transformation of the lattice is good but not at any price. How do the vectors obtained after transformation compare with those we get after a "regular" LLL-reduction?

As a matter of fact our tests show that most of the time (above 99 %) the transformed matrix provides us with polynomials whose quality is comparable (in terms of failures to comply with Coppersmith's condition) with those the LLL-reduction outputs. While performing a full LLL-reduction of the transformed matrix is relatively quick we do not find it very helpful.

Most of the time, when the first transformation fails, performing the full LLL-reduction does not provide us with the vectors we need. It turns out that, instead of stubbornly persevering with the current attempt, it is better to give up and to immediately split the interval and work on the two subintervals.

This performance enhancement has many consequences that will be detailed in subsequent sections. The most outstanding one is that it takes us to the verge of considering that the exploration in the initial hardness-to-round bracket we have considered ($6p$ to $12p$) is practically doable.

3.9.2 The vanished time drift

This remark may seem of anecdotal interest but we think it can be put at use to discover the reason for this phenomenon. This allows to discard the hypothesis of a steady accumulation of "friction" in the SageMath runtime even if we do not have any alternative explanation yet.

3.9.3 Confirmation of the "sweet spot" effect

We witness the same kind of behavior with this second set of data, even it is not as sharp was with the first set. This probably indicates that our previous observations are not an artifact generated by our reduction process but probably reveal a more profound property of the lattice basis, even if we are not able to characterize it yet.

Our present lack of clues certainly relates to our flawed knowledge about what the underlying structure of the problem is. This shortcoming has already emerged in different occasions: lattice building rules, optimal choice of parameters, etc. One possible extension of this work is to gather more and more precise, empirical data, thanks to the existing implementation, in order to figure out how what is actually going on here.

3.9.4 Other conserved trends

For one, the ease to explore the higher hardness-to-round receives a second confirmation since, this time, we have timings for $10p$. In the listed results the effect is not as striking that for the $6p$ to $8p$ transition, but is still visible (a factor 2). A possible explanation is also that the transition from $6p$ to $8p$ is proportionally larger than that from $8p$ to $10p$.

Another conserved trend is that pertaining higher degrees. For a given value of constant α, the gains in interval width obtained thanks to a better approximation accuracy with higher-degree polynomials are more than outbalanced by the failures in non-trivial common factor free polynomial pairs search. Losses in execution time occasioned by the growth of lattice basis data size worsen the picture. In some tests for $10p$ we even witnessed a kind of regressive behavior. For instance, for a polynomial degree in 46 or 47 (not reported in our tables) and with $\alpha = 10$, we could not cover a 2^{93}-floating-point-numbers interval with a single iteration while it was done with a polynomial in degree 45 and with the same value for α.

3.9.5 A new target: Optimize in other areas

The results of the first set of data focused our attention on the reduction step. As can be seen with the second set, lattice basis transformation "only" represents from 2/3 to 3/4 of all the execution

time. This means that other parts of the implementation deserve now our optimization zeal.

While not shown in our data, resultants computation and roots finding grab most of the remaining execution time. This opens several paths in this realm:

- use more effective implementations than those presently available in SageMath;
- use different algorithms (as, for instance, Hensel lifting, see [108], Chap. 15, instead of resultants).

More generally, now that lattice basis transformation optimization is a less obsessive matter, all other up to now neglected enhancement trails may require our attention.

3.9.6 When do we get there?

From the timings and the interval width we can try to figure out how long it takes to cover a complete binade. We think that a single but full binade processing time is the right metric to judge on the practicability of the exploration. For a full function domain, many parts can be dealt with in such a way that we never have to run the SLZ-algorithm for them. But their size differs from function to function.

3.9.6.1 Not all the binades always have to be explored

In our case for instance, with the function $\exp(x)$ and `binary128` format, the situation is depicted in Figure 3-11.

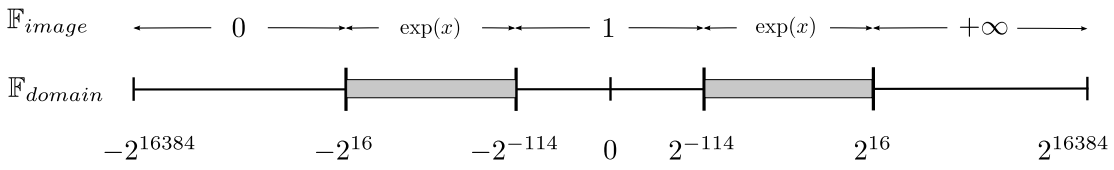

Figure 3-11: The top line indicates the value of the image of the elements of the bottom line domain by the $\exp(x)$ function. Only gray shaded domain intervals need to be checked for hard-to-round cases. The boundaries of the domain intervals where all the elements have a constant image (0, 1, $+\infty$) do not coincide with binades limits. For instance, the value of $\exp(x)$ transitions to $+\infty$ "somewhere" in 15-fbinade. Hence, at least some part of this fbinade should be explored. Strictly speaking the actual extremal boundaries of the domain are one floating-point number short of those indicated on the figure.

It turns out that only 258 fbinades need to be explored out of the 65534 that the full domain comprises (a reduction factor of 254!).

3.9.6.2 Computing fantasies

As it goes, and using our best performing parameters (hardness-to-round checked = $10p$, $\alpha = 10$, degree = 45, number of floating point numbers per interval = 2^{93}), the exploration of a full binade would take 66 years on a single of our E5-2695 cores. Nevertheless one can easily break this task in a set of embarrassingly parallel jobs that can be simultaneously run on the adequate infrastructure.

As of April 2016, the top500 [74] first ranking supercomputer is the Chinese Tianhe-2 (MilkyWay-2). It has as much as 32,000 Intel®Xeon®E5-2692 @ 2.2GHz and 48,000 Intel®Xeon Phi^{TM}31S1P @ 1.1GHz processors. These amount to 3,120,000 cores.

Under the (mildly optimistic) hypothesis that these cores would fare as well as the ones we tested with, and the ludicrous one that we could borrow it in full from its present users (and provided we could master the beast), we could obtain the following throughput:

- single binade exploration: the timing is lower bounded by that of the exploration of a single interval (as it has been seen, splitting to smaller intervals is a waste of resources); in our example it would take 1.1 h but using only 524,288 of the 3,120,000 cores;
- full function $\exp(x)$ exploration: it would take a bit less than 2 days (a bit more than 17,000 core $\times$ year);
- full generic worst-case function exploration (without any binade elimination): a bit less than 1.4 year (around $4,319,895$ core $\times$ year).

3.9.6.3 Back to Real World

A large part of the hardest-to-round cases output from the work performed by V. Lefèvre were computed in our lab, over years, on a set of servers and desktops whose total core count amounts to about 500. These computers are not fully dedicated to that task: a scheduler gathers the computing time of unused cores for low priority jobs, such as those that search for hard-to-round cases. There are many different processor models (even if they are all 64-bit) in this infrastructure and the ones we used to set our own timings are among the fastest available.

Neglecting all these details and assuming we have 500 powerful cores at hand, exploring a single binade would take about $1,150$ hours, (48 days). Checking for the full $\exp(x)$ function, would require

34 years of computations and for a generic (without binade elimination, a very unlikely situation though) function, one should have the patience to wait $8,639$ years.

We must imagine a better than 3 optimization factor (by taking full advantage of all the above proposed leads?) and a $5,000$-core cluster (a 6 to 7 full racks set and for an about $2,500,000$ € price tag with commodity hardware) to be able to perform the checking of the $\exp(x)$ function in a single year.

3.9.7 No animals were harmed during the experiments

But no hard-to-round cases were found either... which was expected!

3.10 Conclusion

As a Table Maker's Dilemma solver, the SLZ-algorithm fits in the category of (enhanced) exhaustive search methods. It shares this characteristic with the L-algorithm. Scorn is not the point here, far from it. Both algorithms rely on much more smart ideas and clever methods than the mere enumeration of the floating-point numbers. The implementation of the L-algorithm by V. Lefèvre has given important fruits: complete lists of hard-to-round cases for that are actually used [21] for correctly rounded implementations of elementary functions. This long-run effort is still in progress nowadays.

The results presented in this work are the first for the $6p$ to 10 hardness-to-round bracket for the $\exp(x)$ function. The fact that this exploration is not totally out of our reach is not, despite of our reluctance to admit it, a trivial result.

The ultimate question in debate here is can exhaustive search methods, even in sophisticated clothing, stand the hit on the violent combinatorial explosion of the problem in `binary128`? If one compares with `binary64`, one must remember that it is not only the number of floating-point numbers in each binade that balloons to unprecedented levels (from 2^{52} from 2^{112}) but that this problem is made worse by the widely increased exponent range (from $2,047$ fbinades to $32,767$) even if, as we have seen for the function $\exp(x)$, the actual number of fbinades can be squeezed to more tractable levels.

The present work has tried to investigate this question. The answer it gives, though encouraging, is not absolutely conclusive:

- we cannot infer from our current results that exploration is impossible; there are still what we have called "second line" optimizations that can be introduced and while we cannot expect time reduction factors as dramatic as those obtained with projection, even single-digit integer divisors would improve very much the practicability of the exploration;
- our timings based on the availability of Tianhe-2 are a bit unfair and provocative; such an infrastructure will never be devoted to this kind of investigation, however important we may think it is; (but) even building or borrowing more modest (but still a very powerful one, as sketched above) platforms dedicated to this task is hard to imagine nowadays;
- resorting to, as once envisioned, volunteer computing and other kinds of lightweight computation units aggregation would require a deep rewrite of the implementation (what may be needed anyway) in order to alleviate or fully remove the burdening dependencies computing nodes have over huge pieces of software (SageMath, essentially, but also Sollya, fpLLL and the luggage they carry along with them).

There are still too many misunderstood issues. To name a few:

- the rules of optimal initial basis construction for a given polynomial pattern;
- a clear and detailed understanding of the relationship between the main parameters of the method (α, the polynomial degree, the precision, the target hardness-to-round);
- what can be done, if at all, and gained with a more aggressive reduction of the precision of the coefficients.

All these questions have to be examined at an operational level. In other words, for values of the parameters that can be used in actual computations. This involves extensive experimentation and mathematical advances as well.

One could also question the use of an LLL-algorithm implementation as a black box. As D. Stehlé suggests himself in [101], adapting the implementation of the LLL-algorithm (we should more accurately speak now of a *family of algorithms*) to a particular type of input can be a rewarding move. He also opens more general options for the improvement of the LLL-algorithm implementations at large.

But we can also ask ourselves if this effort is worth it. Is the order of magnitude of the gain one can expect from the above suggested improvements enough to make the method actually usable in practice and at what cost? Are there not other routes that should be explored? Did Mathematics speak their last word with the works of Y. Nesterenko and M. Waldschmidt? These questions are still open and we hope that in the forthcoming years they will receive the attention their practical, and possibly theoretical, importance deserves.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] A. Akhavi and D. Stehlé. Speeding-up Lattice Reduction with Random Projections. In E. S. Laber, C. F. Bornstein, L. T. Nogueira, and L. Faria, editors, *LATIN 2008: Theoretical Informatics, 8th Latin American Symposium, Búzios, Brazil, April 7-11, 2008, Proceedings*, volume 4957 of *Lecture Notes in Computer Science*, pages 293–305. Springer, 2008.

[2] M. Albrecht, D. Stehlé, D. Cade, and X. Pujol. fplll-4.0, a floating-point LLL implementation. `http://perso.ens-lyon.fr/damien.stehle`.

[3] AriC research team. `http://www.ens-lyon.fr/LIP/AriC/`. The AriC team and project is the sequel, one renewed perspective, of the Arénaire team.

[4] D. H. Bailey. Qd. `http://crd-legacy.lbl.gov/~dhbailey/mpdist/`.

[5] A. Baker. *Transcendental number theory*. Cambridge mathematical library. Cambridge University Press, Cambridge, New York, 1990. Reprint of the 1975 edition with updates.

[6] S. Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, Nov. 2004.

[7] S. Boldo. Pitfalls of a Full Floating-Point Proof: Example on the Formal Proof of the Veltkamp/Dekker Algorithms. In *Proceedings of the third International Joint Conference on Automated Reasoning (IJCAR)*, pages 52–66, Seattle, USA, Aug. 2006.

[8] D. Boneh and G. Durfee. Cryptanalysis of RSA with Private Key d less than $N^{0.292}$. *IEEE Transactions on Information Theory*, 46(4): 1339–1349, July 2000.

[9] N. P. Brusentsov and J. R. Alvarez. Ternary Computers: The Setun and the Setun 70. In J. Impagliazzo and E. Proydakov, editors, *SoRuCom*, volume 357 of *IFIP Advances in Information and Communication Technology*, pages 74–80. Springer, 2006.

[10] P. L. Chebyshev. Sur les questions de minima qui se rattachent á la représentation approximative des fonctions. In *Mémoires de l'Académie impériale des sciences de Saint. Pétersbourg*, volume VII of *VI*, pages 199–291. Académie impériale des sciences de Saint. Pétersbourg, 1859.

[11] E. W. Cheney. *Introduction to approximation theory.* AMS Chelsea Publishing, Providence, RI, 2d edition, 1998. (Reprint of 1982 edition).

[12] S. Chevillard. *Évaluation efficace de fonctions numériques – Outils et exemples.* PhD thesis, École normale supérieure de Lyon – Université de Lyon, 46, allée d'Italie, 69 364 Lyon Cedex 07, 2009.

[13] S. Chevillard, M. Joldeş, and C. Lauter. Sollya. `http://sollya.gforge.inria.fr/`.

[14] S. Chevillard, M. JoldeşS, and C. Lauter. Sollya. `http://sollya.gforge.inria.fr/`. Il s'agit d'un outil pour aider au développement de code sûr utilisant la virgule flottante. Il dispose, en particulier, d'une norme infinie certifiée et d'une implantation libre de l'algorithme de Remez.

[15] C. W. Clenshaw. A note on the summation of Chebyshev series. *Math. Tables Aids Comput.*, 9(51):118–120, 1955.

[16] H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[17] D. Coppersmith. Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known. In U. Maurer, editor, *Advances in Cryptology – EUROCRYPT '96*, volume 1070 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin Heidelberg, 1996.

[18] D. Coppersmith. Finding a Small Root of a Univariate Modular Equation. In U. Maurer, editor, *Advances in Cryptology – EUROCRYPT '96*, volume 1070 of *Lecture Notes in Computer Science*, pages 155–165. Springer Berlin Heidelberg, 1996.

[19] The coq proof assistant. `http://coq.inria.fr/`.

[20] M. Cornea-Hasegan. Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.22.2472&rep=rep1&type=pdf`, 1998.

[21] CRLibm. `http://lipforge.ens-lyon.fr/www/crlibm/`.

[22] Cython: C-Extensions for Python. `http://cython.org/`.

[23] F. de Dinechin, A. V. Ershov, and N. Gast. Towards the Post-Ultimate Libm. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ARITH '05, pages 288–295, Washington, DC, USA, 2005. IEEE Computer Society.

[24] F. de Dinechin, C. Lauter, J.-M. Muller, and S. Torres. On Ziv's Rounding Test. *ACM Trans. Math. Softw.*, 39(4):25:1–25:19, July 2013.

[25] F. De Dinechin, J.-M. Muller, B. Pasca, and A. Plesco. An FPGA architecture for solving the Table Maker's Dilemma. In *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, pages 187–194, Santa Monica, United States, Sept. 2011. IEEE Computer Society. Received the Best Paper Award at the conférence.

[26] P. de Faget de Casteljau. Outillages méthodes de calcul. Technical report, André Citroën Automobiles SA, Paris, France, 1959.

[27] C.-J. de La Vallée Poussin. *Leçons sur l'approximation des fonctions d'une variable réelle.* Gauthier-Villars, Paris, 1919.

[28] D. Defour. *Fonctions élémentaires: algorithmes et implémentations efficaces pour l'arrondi correct en double précision.* PhD thesis, École normale supérieure de Lyon, Lyon, France, 2003.

[29] T. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971/72.

[30] J. Ellenberg. *How not to be wrong: the hidden maths of everyday life.* Allen Lane, 2014.

[31] G. Farin. *Curves and Surfaces for CAGD: A Practical Guide.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2002.

[32] *A floating-point library for integer processors*, volume 5559, 2004.

[33] P. Fortin, M. Gouicem, and S. Graillat. Solving the table maker's dilemma by reducing divergence on gpu. In *Proceedings of the 15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN'2012)*, Novosibirsk, Russia, September 2012.

[34] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[35] F. R. F. R. Gantmacher. *The Theory of Matrices*, volume 1. Chelsea Publishing Company, New York, NY, USA, 1959.

[36] D. Goldberg. What Every Ccomputer Scientist Should Know About Floating-Point arithmetic. *ACM Computing Surveys*, 23:5–48, Mar. 1991.

[37] M. Gouicem. *Conception et implantation d'algorithmes efficaces pour la résolution du dilemme du fabricant de tables sur architectures parallèles.* PhD thesis, Université Pierre et Marie Curie - Paris 6, Paris (France), 2013.

[38] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[39] S. Graillat, J.-L. Lamotte, and D. Hong. Error-free transformation in rounding mode toward zero. In A. Cuyt, W. Krämer, W. Luther, and P. Markstein, editors, *Numerical Validation in Current Hardware Architectures*, volume 5492 of *Lecture Notes in Computer Science*, pages 217–229. Springer Berlin Heidelberg, 2009.

[40] S. Graillat, P. Langlois, and N. Louvet. Algorithms for Accurate, Validated and Fast Computations with Polynomials. *Japan Journal of Industrial and Applied Mathematics*, Special issue on Verified Numerical Computation. 26(2,3):191–214, 2009.

[41] J. Gustafson. *The End of Error: Unum Computing*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2015.

[42] T. C. Hales. The Kepler Conjecture. arXiv:math.MG/9811078, 1998.

[43] J. Havil and F. Dyson. *Gamma: Exploring Euler's Constant*. Princeton Science Library. Princeton University Press, 2009.

[44] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002.

[45] W. G. Horner. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, 109:308–335, 1819.

[46] N. Howgrave-Graham. Finding Small Roots of Univariate Modular Equations Revisited. In M. Darnell, editor, *Cryptography and Coding, 6th IMA International Conference, Cirencester, UK, December 17-19, 1997, Proceedings*, volume 1355 of *Lecture Notes in Computer Science*, pages 131–142. Springer, 1997.

[47] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, New York, NY, USA, 29 Aug. 2008.

[48] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Panhaleux. Midpoints and Exact Points of Some Algebraic Functions in Floating-Point Arithmetic. *IEEE Transactions on Computers*, 60(2):228–241, 2011.

[49] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Panhaleux. Midpoints and exact points of some algebraic functions in floating-point arithmetic. *IEEE Trans. Comput.*, 60(2):228–241, Feb. 2011.

[50] C.-P. Jeannerod and S. M. Rump. On relative errors of floating-point operations: optimal bounds and applications. Research report, INRIA, Laboratoire LIP (CNRS, ENS de Lyon, INRIA, UCBL) and Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstraße, Jan. 2014.

[51] M. Joldeş. *Approximations polynomiales rigoureuses et applications*. PhD thesis, École normale supérieure de Lyon – Université de Lyon, Lyon, France, 2011.

[52] W. M. Kahan. Minimizing q*m-n. `http://www.eecs.berkeley.edu/~wkahan/testpi/nearpi.c`, Mar. 1983. Starting at line 67 of the nearpi.c file.

[53] W. M. Kahan. Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit. In A. Iserles and M. J. D. Powell, editors, *On The State of the Art in Numerical Analysis*, New York, NY, USA, 1987. Oxford University Press, Inc.

[54] W. M. Kahan. A test for correctly rounded sqr. `http://www.cs.berkeley.edu/~wkahan/SQRTest.ps`, May 1996.

[55] W. M. Kahan. A Logarithm Too Clever by Half. `http://www.eecs.berkeley.edu/~wkahan/LOG10HAF.TXT`, 9 Aug. 2004.

[56] J. C. Lagarias. *The Kepler conjecture : the Hales-Ferguson proof*. Springer Science+Business Media, LLC, New York, NY, 2011.

[57] L-algorithm. `http://www.vinc17.net/research/papers/these.ps.gz`. Initialy developed by Vincent Lefèvre in his PhD work.

[58] T. Lang and J.-M. Muller. Bound on run of zeros and ones for algebraic functions. In Burgess and Ciminiera, editors, *Proc. of the 15th IEEE Symposium on Computer Arithmetic (Arith-15)*. IEEE Computer Society Press, 2001.

[59] P. Lapsley, J. Bier, E. A. Lee, and A. Shoham. *DSP Processor Fundamentals: Architectures and Features*. Wiley-IEEE Press, 1st edition, 1996.

[60] J. Laskar and M. Gastineau. Existence of collisional trajectories of Mercury, Mars and Venus with the Earth. *Nature*, 459:817–819, June 2009.

[61] C. Lauter. *Arrondi correct de fonctions mathématiques*. PhD thesis, École normale supérieure de Lyon – Université de Lyon, Lyon, France, 2008.

[62] V. Lefevre. Hardest-to-round cases, part 2. `http://tamadiwiki.ens-lyon.fr/tamadiwiki/images/c/c1/Lefevre2013.pdf`, Oct. 08 2013.

[63] V. Lefèvre. *Moyens arithmétiques pour un calcul fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, Jan. 2000.

[64] V. Lefèvre. Génération de coefficients de polynômes d'approximation sur des sous-intervalles, avec bornes d'erreur garanties. `http://tamadiwiki.ens-lyon.fr/tamadiwiki/images/c/c8/Vincent.pdf`, Feb. 2011. Accessed: 2016-05-02.

[65] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982. `http://www.math.elte.hu/~lovasz/scans/lll.pdf`.

[66] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo. Design, Implementation and Testing of Extended and Mixed Precision blas. *ACM Trans. Math. Softw.*, 28(2):152–205, June 2002.

[67] Gcc libquadmath. `https://gcc.gnu.org/onlinedocs/libquadmath/`. The GCC Quad-Precision Math Library Application Programming Interface (API).

[68] J. Liouville. Sur des classes très étendues de quantités dont la valeur n'est ni algébrique, ni même réductible à des irrationnelles algébriques. *J. Math. Pures Appl.*, 16:133–142, 1851.

[69] J. Maddock and C. Kormanyos. Boost.Multiprecision. `www.boost.org/doc/libs/`.

[70] A. May. *New RSA Vulnerabilities Using Lattice Reduction Methods*. PhD thesis, University of Paderborn, 2003.

[71] G. Melquiond. Gappa : Génération Automatique de Preuves de Propriétés Arithmétiques. `http://gappa.gforge.inria.fr/`.

[72] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École normale supérieure de Lyon, Lyon, France, 2006.

[73] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, Lyon (France), 2006.

[74] H. Meuer, E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. top500 Supercomputercomputing Sites. `http://www.top500.org/`.

[75] R. K. Montoye, E. Hokenek, and S. L. Runyon. Design of the IBM RISC System/6000 Floating-point Execution Unit. *IBM J. Res. Dev.*, 34(1):59–70, Jan. 1990.

[76] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhauser Boston, Inc., Secaucus, NJ, USA, 2006.

[77] J.-M. Muller and N. Brisebarre. Correct rounding of algebraic functions. *RAIRO - Theoretical Informatics and Applications*, 41(1):71–83, Apr. 2007.

[78] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[79] Y. V. Nesterenko and M. Waldschmidt. On the approximation of the values of exponential function and logarithm by algebraic numbers (in Russian). In Y. V. Nesterenko, editor, *Diophantine Approximations - Proceeding of papers dedicated to Prof. N.I. Feldman*, volume 2 of *Mat. Zapiski*, pages 23–42, Moscow, Russia, 1996. Center for applied research under Mech. Math. Faculty of MSU. Available in English at `http://arxiv.org/pdf/math/0002047.pdf`.

[80] P. Q. Nguyen and D. Stehlé. Floating-Point LLL Revisited. In R. Cramer, editor, *Advances in Cryptology– EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 215–233. Springer Berlin Heidelberg, 2005.

[81] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, June 2005.

[82] M. Parks. Number-theoretic test generation for directed rounding. *IEEE Transactions on Computers*, 49(7):651–658, 2000.

[83] M. Parks. Unifying tests for square root. In G. Hanrot and P. Zimmermann, editors, *Proceedings of the 7th Conference on Real Numbers and Computers (RNC'7)*, page 151 pages. None., 2006.

[84] D. M. Priest. *On Properties of Floating-Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California, berkeley, Nov. 1992.

[85] S. K. Raina. *FLIP: a Floating-Point Library for Integer Processors*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, Sept. 2006.

[86] E. J. E. R. Remez. Sur la détermination des polynômes d'approximation de degré donné. *Communications de la Société mathématique de Kharkov*, 10:41–63, 1934.

[87] Remez, E. Ja. (Eugène Remes). Sur le calcul effectif des polynômes d'approximation de Tchebichef. *Comptes rendus hebdomadaire des séances de l'Académie des sciences*, 199:337–340, 1934. Is a sequel of [88]. Available online at `http://gallica.bnf.fr/ark:/12148/bpt6k3151h.image`.

[88] Remez, E. Ja. (Eugène Remes). Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *Comptes rendus hebdomadaire des séances de*

l'Académie des sciences, 198:2063–2064, 1934. Available online at `http://gallica.bnf.fr/ark:/12148/bpt6k31506.image`.

[89] G. 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, 46 allée d'Italie, F-69364 Lyon cedex 07, France, Dec. 2009.

[90] S. M. Rump, T. Ogita, and S. Oishi. Accurate Floating-Point Summation. Part I: Faithful Rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[91] S. M. Rump, T. Ogita, and S. Oishi. Accurate Floating-Point Summation. Part II: Sign, K-Fold Faithful and Rounding to Nearest". *SIAM Journal on Scientific Computing*, 31(2):1269–1302, 2008.

[92] Sage: Open source mathematics software. `http://www.sagemath.org/`.

[93] C. Schnorr. A more efficient algorithm for lattice basis reduction. *J. Algorithms*, 9(1):47–62, 1988.

[94] C. Schnorr and M. Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. In *Fundamentals of Computation Theory, 8th International Symposium, FCT '91, Gosen, Germany, September 9-13, 1991, Proceedings*, pages 68–85, 1991.

[95] C. Schnorr and M. Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. In *Math. Programming*, pages 181–191, 1993.

[96] M. J. Schulte and E. E. Swartzlander. Exact rounding of certain elementary functions. In E. E. Swartzlander, M. J. Irwin, and G. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 138–145. IEEE Computer Society Press, Los Alamitos, CA, June 1993.

[97] V. Shoup. NTL: A Library for Doing Number Theory.

[98] M. Spivak. *Calculus.* Cambridge University Press, Cambridge, [Eng.], 3d edition, 1994.

[99] D. Stehlé. *Algorithmique de la réduction de réseaux et application à la recherche de pires cas pour l'arrondi de fonctions mathématiques.* PhD thesis, Université Henri Poincaré – Nancy 1, Nancy (France), 2005.

[100] D. Stehlé. On the randomness of bits generated by sufficiently smooth functions. In F. Hess, S. Pauli, and M. E. Pohst, editors, *Algorithmic Number Theory, 7th International Symposium,*

ANTS-VII, Berlin, Germany, July 23-28, 2006, Proceedings, volume 4076 of *Lecture Notes in Computer Science*, pages 257–274. Springer, 2006.

[101] D. Stehlé. Floating-point lll: Theoretical and practical aspects. In P. Q. Nguyen and B. Vallée, editors, *The LLL Algorithm*, Information Security and Cryptography, pages 179–213. Springer Berlin Heidelberg, 2010.

[102] D. Stehlé, V. Lefèvre, and P. Zimmermann. Worst Cases and Lattice Reduction. In J.-C. Bajard and M. Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic, Santiago de Compostela, Spain*, pages 142–147, Los Alamitos, CA, Mar. 2003. IEEE Computer Society Press. `http://http://www.acsel-lab.com/arithmetic/arith16/papers/ARITH16_Stehle.pdf`.

[103] D. Stehlé, V. Lefèvre, and P. Zimmermann. Searching Worst Cases of a One-Variable Function Using Lattice Reduction. *IEEE Transactions on Computers*, 54(3):340–346, Mar. 2005. `http://csdl.computer.org/comp/trans/tc/2005/03/t0340abs.htm`.

[104] P. H. Sterbenz. *Floating-point computation*. Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[105] TaMaDI, an ANR project involving the AriC, Marelle and PEQUAN teams. `http://tamadiwiki.ens-lyon.fr/`. Project leader: J.-M. Muller.

[106] The GNU Project. The GNU C Library. `http://ftp.gnu.org/gnu/glibc/glibc-2.20.tar.gz`.

[107] S. Torres. PoBySo: Powered By Sollya, Python-Sage bindings. `http://forge.cbp.ens-lyon.fr/redmine/projects/pobyso/`, 2015.

[108] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, second edition, 2003.

[109] K. Weierstrass. Über die analytische Darstellbarkeit sogenannter willkürlicher Functionen einer reellen Veränderlichen. *Sitzungsberichte Akad. Berlin*, pages 633–639, 789–805, 1885.

[110] K. Weierstrass. Sur la possiblité d'une représentation analytique des fonctions dites arbitraires d'une variable réelle. *Journal de mathématiques pures et appliquées*, 2:105–115, 1886. Translation to french by L. Laugel of [109]. Available on line at `http://gallica.bnf.fr/ark:/12148/cb343487840/date`.

[111] A. Ziv. Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit. *ACM Transactions on Mathematical Software*, 17(3):410–423,

Sept. 1991. http://dl.acm.org/citation.cfm?id=114697.116813&coll=DL&dl=ACM&CFID=453974937&CFTOKEN=40298601.