UNIVERSITÉ PIERRE ET MARIE CURIE - PARIS 6
Laboratoire d'informatique de Paris 6

HABILITATION À DIRIGER DES RECHERCHES

spécialité : informatique

présentée et soutenue publiquement le 2 décembre 2013

par Stef GRAILLAT

Contribution à l'amélioration de la précision et à la validation des algorithmes numériques

(contribution to the increase of accuracy and validation of numerical algorithms)

après avis des rapporteurs,

Dominique	MICHELUCCI	Professeur à l'université de Bourgogne
Jean-Michel	MULLER	Directeur de recherche CNRS, École normale supérieure de Lyon
Lihong	ZHI	Professeure à l'Académie des sciences de Chine

et devant le jury composé de

Valérie	BERTHÉ	Directrice de recherche CNRS, université Paris Diderot
Jean-Guillaume	DUMAS	Professeur à l'université Joseph Fourier
Laura	GRIGORI	Directrice de recherche Inria, université Pierre et Marie Curie
Dominique	MICHELUCCI	Professeur à l'université de Bourgogne
Jean-Michel	MULLER	Directeur de recherche CNRS, École normale supérieure de Lyon
Mohab	SAFEY EL DIN	Professeur à l'université Pierre et Marie Curie

CONTENTS

INTRODUCTION

On February 25, 1991, during the first Gulf War, an american Patriot Missile battery failed to intercept an Iraqi Scud missile. The Scud killed 28 soldiers. It turns out that the cause was an inaccurate computation of the time since boot due to computer arithmetic errors.

More precisely, the time in tenths of second measured by the internal clock was multiplied by 1/10 in order to compute the time in seconds. The computations were done using a 24 bit fixed-point register. Because the number 1/10 has an infinite binary expansion, it was chopped at 24 bits after the radix point. The small rounding error, when multiplied by the large number giving the time in tenths of a second, lead to a significant error. Indeed, the Patriot battery had been working around 100 hours, and so the resulting time error due to the rounding error was about 0.34 seconds. (The binary expansion of 1/10 is 0.0001100110011001100110011001100.... So 1/10 is stored as 0.00011001100110011001100 in the 24 bits register of the Patriot which lead to an error of $1.1001100... \times 2^{-24}$ in binary, or about 0.000000095 in decimal. Multiplying by the number of tenths of a second in 100 hours gives 0.34). A Scud travels at about 1500 meters per second which is more than a half a kilometer in this time. This was sufficient for the Patriot to fail intercepting the Scud.

As a consequence, a good knowledge of the floating-point arithmetic is very important for computer scientists in order to take into account the existence of rounding errors. One common practice to decrease the rounding errors is to increase the precision of the arithmetic. While in general more robust, this can affect the performance of the program significantly. Moreover, as it will be shown in chapter 5 with Rump's polynomial, increasing the precision does not necessarily increase the accuracy of the computed result. Consequently, it is important to get more information about the accuracy of the computed results.

Our main research topic is then to provide *fast and accurate numerical algorithms*. Of course, when one makes an algorithm more accurate, the cost generally increases. Our aim is to limit this increase. To get a very accurate result, one choice could be to use computer algebra systems. But for some applications where performance is crucial, they may be too slow. So we prefer to use the floating-point capability of modern processors despite the rounding errors that can occur. We mean by "fast" that our algorithms are more efficient than the standard ones performed with multiprecision libraries like MPFR or QD (quad-double). These libraries are easy to use and need very few transformations of the algorithm to be used.

Our method (use of error-free transformations and compensated algorithms) needs more work but provides faster algorithms.

Sometimes it is difficult to rewrite a program to increase the accuracy or even to know the accuracy of the computed result (this is the case for huge codes with millions of lines). Computing a *a priori* general error bound can only be done with small programs.

Indeed in large numerical simulations, the final rounding error is the result of billions of elementary rounding errors. In the general case, it is impossible to carefully describe each elementary error and, then, to compute the right value of the final rounding error. Our aim is, nevertheless, to try to give an insight of the error to the user. This is what is called *validation of numerical codes*. As a deterministic approach is difficult, it is usual to apply a probabilistic model. Of course, one looses the exact description of the phenomena but one may hope to get some global information like order of magnitude. This is the aim of our SAM library.

Another challenge for us is to provide results as accurate as those returned using computer algebra but using numerical algorithms. Testing the nonsingularity of a matrix can be done in computer algebra by exactly computing the determinant. In floating-point arithmetic, it is more difficult due to rounding errors. But using interval arithmetic, it is possible, in some cases, to rigorously prove that a matrix is nonsingular. Moreover, combining interval arithmetic and some fixed-point theorems, it is possible to ensure the existence of the some objects (a multiple root for example) within a certain tolerance.

The document is organized as follows:

- Chapter 1 contains a brief summary of our research since our PhD thesis in 2005. We give a summary of all our publications since 2005. We also provide a list of our publications in international journals and in international refereed conferences.
- Chapter 2 is a short introduction to computer arithmetic (especially floating-point arithmetic) and also presents some methods to perform rounding error analysis. We mainly focus on forward/backward error analysis, interval analysis and stochastic analysis. These are, indeed, the methods we will use to prove the accuracy of our algorithms and to validate or certify some properties of those algorithms.
- In chapter 3, we present the core of our research. Our aim is to increase the accuracy of some algorithms which correspond more or less to polynomial evaluation. For that, we use the so-called *error-free transformations* (algorithms that make it possible to compute the elementary rounding errors). Besides increasing the accuracy, we also provide certified error bounds that can be compute in floating-point arithmetic. We also show that our algorithms are faster than the classic ones using multiprecision libraries sharing the same accuracy.
- In chapter 4, we study some methods to verify mathematical assumptions on computers. We deal with the problem of multiple zeros of nonlinear equations. As it is an ill-posed problem, we change a little bit the problem. We do not show that the equation has a multiple zero but we show that a simple perturbation of the equation has a true multiple zero. For that we use the so-called *self-validating methods* which are base on

fixed-point theory and the use of interval arithmetic.

- In chapter 5, we present a tool called SAM which is a multiprecision implementation of the DSA (Discrete Stochastic Arithmetic). This library makes it possible to perform a validation of huge numerical code. This gives only a confidence interval but it is generally sufficient for all but critical applications.

CHAPTER

ONE

RESEARCH SUMMARY

In this chapter, we present the work we have done since the defense of our PhD thesis in 2005. Our main objective is to compute *accurately* and *as fast as possible* some quantities that are useful in practice. For that, we use the fine properties of the floating-point arithmetic as well as the power of the recent computer architectures (like parallel architectures and superscalar architectures). In order to take avantage of the new architectures (GPU for example), we try to redesign the algorithms in order to get the best performances in term of computing time.

Our work can be divided into four main topics:

- The first one deals with the accurate evaluation of polynomials as well as some related topics like the computation of roots of polynomials or the evaluation of elementary symmetric functions. Our aim is to use the properties of floating-point arithmetic to derive some new algorithms that are accurate and faster than the other state-of-the-art algorithms sharing the same accuracy. This work is described in Section 1.1.
- The second one corresponds to the validation and certification of numerical algorithms. The idea is either to prove in floating-point arithmetic that a result is mathematically true (like the fact that a matrix with floating-point coefficients is nonsingular) or to give any guarantee (like an confidence interval) that the result is sufficiently accurate. For the first argument, we use interval arithmetic and existence results like Brouwer's fixed point theorem. For the second argument, we use statistic and probability theory. This work is described in Section 1.2.
- The third one concerns symbolic-numeric algorithms. The symbolic algorithms give an exact representation of the result but can sometimes be slow. The numerical algorithms (performed in finite precision) are in general faster but give an approximate result. Taking advantage of the power of the floating-point units is a challenge in the computer algebra community motivated by the increase of the complexity of the problems to tackle. One of the main difficulty is to find which part of a symbolic algorithm can be performed in finite precision without leading to a imprecise result due to rounding errors. This work is described in Section 1.3.
- The fourth is concerned with the use of new parallel architectures (for example GPU

and Xeon Phi) to accelerate the resolution of problems necessitating huge power of computation. For instance, the IEEE 754-2008 standard recommends the correct rounding of some elementary functions. This requires to solve the Table Maker's Dilemma which implies a huge amount of CPU computation time. We have considered accelerating such computations on Graphics Processing Units (GPUs) which are massively parallel architectures with a partial SIMD execution (Single Instruction Multiple Data). We have deployed the existing algorithm on GPU but we also redesigned this algorithm to better take into account the architecture of the GPU. This work is described in Section 1.4.

The references in this chapter correspond to our publications and are listed in Section 1.5. The work done after the PhD thesis corresponds to publications [1 – 10] and [17 – 24]. They are shortly described below.

1.1 Accurate polynomial evaluation and applications

In [6], we introduced a compensated Horner scheme, that is an accurate and fast algorithm to evaluate univariate polynomials in floating-point arithmetic. The accuracy of the computed result is similar to the one given by the Horner scheme computed in twice the working precision. This compensated Horner scheme runs at least as fast as existing implementations producing the same output accuracy. We also proposed to compute in pure floating-point arithmetic a valid error estimate that bounds the actual error of the compensated evaluation. Numerical experiments involving ill-conditioned polynomials confirmed these results. All algorithms are performed at a given working precision and are portable assuming the floating-point arithmetic satisfies the requirements of the IEEE-754 standard.
The main tool for achieving such accuracy is *the error-free transformations* (EFT). These are algorithms that make it possible to compute (in pure floating-point arithmetic) the rounding error for the elementary operations (addition, subtraction and multiplication). Indeed, it is possible to show that these elementary rounding errors can be represented exactly as floating-point numbers.

The EFT were first designed only for real floating-point operations. Nevertheless, it is common to deal with complex numbers in numerous applications. In order to be able to use those accurate algorithms with complex numbers, we generalized in [23] the EFT to deal with complex floating-point numbers. Thanks to these new EFT, we were able to generalize all the compensated algorithms (for summation, dot product and Horner scheme mainly) to deal with complex numbers [3].

We also used the EFT to provide a method to improve the accuracy of the product of floating-point numbers [8]. We show that the computed result is as accurate as if computed in twice the working precision. Such an algorithm can be useful for example to compute the determinant of a triangular matrix and to evaluate a polynomial when represented by the root product form. It can also be used to compute an integer power of a floating-point number. Moreover we showed that the result computed by our new algorithm was a *faithful*

rounding of the exact result. Faithful rounding means that the computed result is equal to the exact result if the latter is a floating-point number and otherwise is one of the two adjacent floating-point numbers of the exact result.

As mentioned previously, our algorithms could be useful for evaluating a polynomial when represented by the root product form. If one wants to do that efficiently, it corresponds to a fast and accurate evaluation of elementary symmetric functions. We proposed in [17] a new compensated algorithm by applying error-free transformations to improve the accuracy of the so-called Summation Algorithm, which is used, for example, in MATLAB's `poly` function. We derived a theoretical forward rounding error bound and a validated running error bound for our new algorithm. The theoretical rounding error bound implies that the computed result is as accurate as if computed with twice the working precision and then rounded to the current working precision. The validated running error analysis provides a shaper bound along with the result, without increasing significantly the computational cost. Numerical experiments illustrated that our algorithm runs much faster than the algorithm using the classic `double-double` library while sharing similar error estimates. Such an algorithm can be widely applicable for example to compute characteristic polynomials from eigenvalues. It can also be used into the Rasch model in psychological measurement.

We also investigated the inverse problem: given the floating-point coefficients of a polynomial, compute the zeros of this polynomial in floating-point arithmetic. For that we examined in [9] the local behavior of Newton's method in floating-point arithmetic for the computation of a simple zero of a polynomial assuming that a good initial approximation is available. We allow an extended precision (twice the working precision) in the computation of the residual thanks to our compensated Horner scheme described previously. We proved that, for a sufficient number of iterations, the computed zero is as accurate as if computed in twice the working precision. We provided numerical experiments confirming this. But in double precision, this result is limited to a condition number of 10^{16} for the zeros. If we want to deal with higher condition numbers, it is necessary to accurately evaluate the derivative of the polynomial as well.

For that, we have introduced a compensated algorithm for the evaluation of the k-th derivative of a polynomial in power basis in [2]. The proposed algorithm makes it possible the direct evaluation without obtaining the k-th derivative expression of the polynomial itself, with a very accurate result to all but the most ill-conditioned evaluation. Forward error analysis and running error analysis are performed by an approach based on the data dependency graph. Numerical experiments illustrated the accuracy and efficiency of the algorithm. We have used this algorithm to improve the accuracy of Newton's method for simple roots that are highly ill-conditioned. Our previous result on Newton's method is now valid until a condition number of 10^{32} in double precision.

In order to increase the performances of some algorithms, it is important to take into account the architecture of the processor. In 2009, one of the most promising processors was the Cell. This processor was very fast but was not IEEE-754 compliant. In single precision,

the only rounding mode available was rounding toward zero (truncation). The classic EFT are valid in rounding to the nearest but not with rounding toward zero. Some algorithms were developed to perform error-free transformations in this rounding mode but they were not efficient on the Cell processor. In [22, 7], we then have redesigned these EFT to better fit the architecture of the Cell. This step was a first step toward the implementation of a quadruple precision arithmetic on this processor. We have also studied different implementations of interval arithmetic with only the rounding toward zero available.

Some of the results summarized here are described in detail in Chapter 3.

1.2 Validation and certification of numerical algorithms

In [20, 4], we developed a library called SAM. It is a multi-precision extension of the classic CADNA library. The CADNA library implements a stochastic arithmetic that makes it possible to estimate the propagation of rounding error in scientific codes. Concretely, each variable is replaced by three floating-point numbers and the rounding mode is chosen randomly. The figures that differ in these three numbers are due to rounding errors. The actual version of CADNA only deals with single and double precision numbers defining stochastic numbers in single and double precision. We worked on extending the library so that we can use multiprecision numbers. In CADNA the arithmetic and relational operators are overloaded in order to be able to deal with stochastic numbers. As a consequence, the use of CADNA in a scientific code only needs few modifications. The MPFR Library defines types of multiprecision numbers as well as arithmetic functions acting on multiprecision variables. What is important is that MPFR provides correctly rounded functions and operators in multiprecision. So we used MPFR in CADNA. This new library called SAM makes it possible to dynamically control the numerical methods used and more particularly to determine the optimal number of iterations in an iterative process. This also makes it possible to tackle some problems in the numerical validation of chaotic systems as well as in the dynamic control of approximation methods computing integrals.

This work is presented in Chapter 5.

It is well known that it is an ill-posed problem to decide whether a function has a multiple root. Even for a univariate polynomial an arbitrarily small perturbation of a polynomial coefficient may change the answer from yes to no. In [5], we have proposed an algorithm for computing verified and narrow error bounds with the property that a slightly perturbed system is proved to have a double root within the computed bounds. For a univariate nonlinear function we have given a similar method also for a multiple root. A narrow error bound for the perturbation is computed as well. Computational results for systems with up to 1000 unknowns have demonstrated the performance of the methods. This result is based on the use of interval arithmetic which makes it possible to have validated error bounds together with Brouwer's theorem that can ensure the existence of a value.

This work is presented in Chapter 4.

1.3 Symbolic-numeric algorithms

Polynomials appear in almost all areas in scientific computing and engineering. Most applications need to solve equations involving polynomials and systems of polynomials, often in many variables. A list of the major fields where polynomial systems appear includes for example: Computer Aided Design and Modeling, Mechanical Systems Design, Signal Processing and Filter Design, Civil Engineering, Robotics, Simulation. The wide range of use of polynomial systems needs to have fast and reliable methods to solve them. Roughly speaking, there are two general approaches: symbolic and numeric. The symbolic approach is based either on the theory of Gröbner basis or on the theory of resultants. For the numeric approach, we use iterative methods like Newton's method or homotopy continuation methods. Recently, "hybrid methods", combining both symbolic and numeric methods, began to appear.

In [21], we have provided a new method for computing numerical approximations of the roots of a zero-dimensional system. It works on general systems, even those with multiple roots, and avoids any arbitrary choice of linear combination of the multiplication operators. It works by computing eigenvectors (or a basis of the full invariant subspaces). The sparsity/structure of the multiplication operators by one variable can also be taken into account. The main contribution is to use finite precision computations when possible but still certifying the results through interval computations.

Finite fields are widely used in numerous areas like cryptography, error-correcting codes or computer algebra. Dot products are ubiquitous in all computations especially when dealing with linear algebra. Developing fast libraries for computing dot products in finite fields is a key tool to tackle various problems in scientific computing.

In [19], we have proposed several possibilities for using fast floating-point units for computing dot products in finite fields. The main concern is then to properly manage rounding errors that may appear during the computations. To solve this problem, we used error-free transformations (EFT). Using these EFT on recent processors (with an FMA), we have shown that it is possible to deal with large finite fields. We implemented those algorithms on GPU so that we can exploit the computational capabilities of GPU to their full extent to get the result efficiently. It was possible to reach speedups between 10 or 40 with the parallel versions, with an algorithm using nearly no modular reduction.

The pseudozero set of a polynomial p is the set of complex numbers that are roots of polynomials which are near p. This is a powerful tool to analyze the sensitivity of roots with respect to perturbations of the coefficients. Some applications in algebraic computation and robust control theory have been proposed recently. In [10], we established some topological and geometric properties of the pseudozero set such as boundedness, compactness and convexity.

1.4 The Table Maker's Dilemma and parallel architectures

In floating-point arithmetic, having fully-specified operations is a key-requirement, if one wants portable and predictable numerical software. Since 1985 and the IEEE-754 standard (revised in 2008), the four arithmetic operations $(+, -, \times, /)$ are specified (they must be correctly rounded: the system must return the floating-point number nearest the exact result). This is not fully the case for the basic mathematical functions. The same function may return significantly different results depending on the environment. Hence, numerical programs using these functions suffer from various problems:

- it is almost impossible to estimate their accuracy;
- their portability is difficult to guarantee.

This lack of specification is due to a problem called the *Table Maker's Dilemma*. To compute $f(x)$ in a given format, where x is a floating-point number, we first compute an approximation to $f(x)$ and then we round it to the nearest floating-point number. The problem is: what must the accuracy of the approximation be to ensure that the obtained result is always equal to $f(x)$ rounded to the nearest floating-point number?

To solve that problem, we must locate, for each considered floating-point format and function, what is the hardest to round (HR) point, i.e., the floating-point number x such that $f(x)$ is closest (without being equal) to the exact middle of two consecutive floating-point numbers. Current algorithms (Lefèvre algorithm and SLZ algorithm) to compute such HR points present an exponential complexity with the number of bits of the format.

In order to obtain results in quadruple precision, which is currently a real challenge, these implementations should be able to efficiently operate on current and forthcoming petaflops systems. In the long term, this work should enable to require the correct rounding of some functions in the next versions of the IEEE-754 standard, which will make it possible to completely specify all the components of numerical softwares.

Finding the HR point can be done by exhaustive search. Lefèvre et al. proposed in 1998 an algorithm which improves the exhaustive search by computing a lower bound on the distance between a line segment and a grid. In [18], we presented an analysis of this algorithm in order to deploy it efficiently on GPU (Graphics Processing Units). We managed to obtain a speedup of 15.4 on a NVIDIA Fermi GPU over one single high-end CPU core.

We have also proposed an analysis of the Lefèvre HR argument search using the concept of continued fractions. We then proposed a new parallel search algorithm much more efficient on GPU thanks to its more regular control flow. We also presented an efficient hybrid CPU-GPU deployment of the generation of the polynomial approximations required in Lefèvre's algorithm. In the end, we managed to obtain overall speedups up to 53.4x on one GPU over a sequential CPU execution, and up to 7.1x over a multi-core CPU, which enable a much faster solving of the Table Maker's Dilemma for the double precision format.

1.5 Publications

This section only contains our publications in journals and international refereed conferences. All our publications including articles in unrefereed conference proceedings, abstracts and extended abstracts in international conferences and posters are available at the following webpage : http://www-pequan.lip6.fr/~graillat/.

Book chapters, chapters for encyclopedia

[1] Jean-Marie Chesneaux, Stef Graillat, and Fabienne Jézéquel. *Encyclopedia of Computer Science and Engineering*, volume 4, chapter Rounding Errors, pages 2480–2494. Wiley, 2009.

Articles in journals

[2] Hao Jiang, Stef Graillat, Canbin Hu, Shengguo Lia, Xiangke Liao, Lizhi Cheng, and Fang Su. Accurate evaluation of the k-th derivative of a polynomial. *J. Comput. Appl. Math.*, 191:28–47, 2013.

[3] Stef Graillat and Valérie Ménissier-Morain. Accurate summation, dot product and polynomial evaluation in complex floating point arithmetic. *Information and Computation*, (216):57–71, 2012.

[4] Stef Graillat, Fabienne Jézéquel, Shiyue Wang, and Yuxiang Zhu. Stochastic arithmetic in multiprecision. *Math.comput.sci.*, 5(4):359–375, 2011.

[5] Siegfried R. Rump and Stef Graillat. Verified error bounds for multiple roots of systems of nonlinear equations. *Numer. Algorithms*, 54(3):359–377, 2010.

[6] Stef Graillat, Philippe Langlois, and Nicolas Louvet. Algorithms for accurate, validated and fast polynomial evaluation. *Japan J. Indust. Appl. Math.*, 2-3(26):191–214, 2009.

[7] Diep Nguyen Hong, Stef Graillat, and Jean-Luc Lamotte. Extended precision with a rounding mode toward zero environment. application on the cell processor. *Int. J. Reliability and Safety*, 3(1/2/3):153–173, 2009.

[8] Stef Graillat. Accurate floating point product and exponentiation. *IEEE Transactions on Computers*, 58(7):994–1000, 2009.

[9] Stef Graillat. Accurate simple zeros of polynomials in floating point arithmetic. *Comput. Math. Appl.*, 56(4):1114–1120, 2008.

[10] Stef Graillat. Some topological and geometric properties of pseudozero set. *Appl. Math. E-Notes*, 8:98–108, 2008.

[11] Stef Graillat. Pseudozero set of real multivariate polynomials. *Math.comput.sci.*, 1(2):337–352, 2007.

[12] Stef Graillat and Philippe Langlois. Real and complex pseudozero sets for polynomials with applications. *Theor. Inform. Appl.*, 41(1):45–56, 2007.

[13] Françoise Tisseur and Stef Graillat. Structured condition numbers and backward errors in scalar product spaces. *Electron. J. Linear Algebra*, 15:159–177 (electronic), 2006.

[14] Stef Graillat. A note on structured pseudospectra. *J. Comput. Appl. Math.*, 191(1):68–76, 2006.

[15] Stef Graillat. A note on a nearest polynomial with a given root. *SIGSAM Bull.*, 39(2):53–60, 2005.

[16] Stef Graillat. Computation of pseudozero abscissa. *An. Univ. Timişoara Ser. Mat.-Inform.*, 42(Special issue):115–128, 2004.

Articles in refereed conference proceedings

[17] Hao Jiang, Stef Graillat, and Roberto Barrio. Accurate and fast evaluation of elementary symmetric functions. In *Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, TX, USA, April 7-10*, pages 183–190, 2013.

[18] Pierre Fortin, Mourad Gouicem, and Stef Graillat. Towards solving the table maker dilemma on GPU. In *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2012), Munich, Germany, February 15-17*, pages 407–415, 2012.

[19] Jérémy Jean and Stef Graillat. A parallel algorithm for dot product over word-size finite field using floating-point arithmetic. In *Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, September 23-26*, pages 80–87, 2010.

[20] Stef Graillat, Fabienne Jézéquel, and Yuxiang Zhu. Stochastic arithmetic in multiprecision. In *NSV3, Third International Workshop on Numerical Software Verification, Edinburgh, UK, July 15th,* 7 pages, 2010.

[21] Stef Graillat and Philippe Trébuchet. A new algorithm for computing certified numerical approximations of the roots of a zero-dimensional system. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation, Seoul, Korea, July 28-31*, pages 167–173, 2009.

[22] Stef Graillat and Jean-Luc Lamotte et Diep Nguyen Hong. Error-free transformation in rounding mode toward zero. In *Lecture Notes in Computer Science (LNCS), Numerical Validation in Current Hardware Architectures*, volume 5492/2009, pages 217–229, 2009.

[23] Stef Graillat and Valérie Ménissier-Morain. Compensated horner scheme in complex floating point arithmetic. In *Proceedings of the 8th Conference on Real Numbers and Computers, Santiago de Compostela, Spain, July 7-9*, pages 133–146, 2008.

[24] Stef Graillat and Valérie Ménissier-Morain. Error-free transformations in real and complex floating point arithmetic. In *Proceedings of the International Symposium on Nonlinear Theory and its Applications, Vancouver, Canada, September 16-19*, pages 341–344, 2007.

[25] Stef Graillat and Philippe Langlois. Pseudozero set of interval polynomials. In *Proceedings of the 21th ACM Symposium on Applied Computing SAC'2006, Dijon, France*, pages 1655–1659, Avril 2006.

[26] Stef Graillat, Philippe Langlois, and Nicolas Louvet. Improving the compensated Horner scheme with a Fused Multiply and Add. In *Proceedings of the 21th ACM Symposium on Applied Computing SAC'2006, Dijon, France*, pages 1323–1327, Avril 2006.

[27] Stef Graillat. Pseudozero set of multivariate polynomials. In Jan Draisma and Hanspeter Kraft, editors, *Proceedings of 10th Rhine Workshop on Computer Algebra (RWCA), Basel, Switzerland*, pages 131–141, Mars 2006.

[28] Stef Graillat and Philippe Langlois. A comparison of real and complex pseudozero sets for polynomials with real coefficients. In Christiane Frougny, Vasco Brattka, and Norbert Muller, editors, *RNC-6, Real Numbers and Computer Conference, Schloss Dagstuhl, Germany*, pages 103–112, Novembre 2004.

[29] Stef Graillat. Computation of pseudozero abscissa. In Dana Petcu, Viorel Negru, Daniela Zaharie, and Tudor Jebelean, editors, *Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timişoara, Romania*, pages 176–187, Septembre 2004.

[30] Stef Graillat and Philippe Langlois. Pseudozero set decides on polynomial stability. In Bart de Moor, Bart Motmans, Jan Willems, Paul Van Dooren, and Vincent Blondel, editors, *Proceedings of the Symposium on Mathematical Theory of Networks and Systems, Leuven, Belgium*, Juillet 2004. (CD-ROM, papers/537.pdf).

[31] Stef Graillat and Philippe Langlois. Testing polynomial primality with pseudozeros. In Marc Daumas, editor, *RNC-5, Real Numbers and Computer Conference, Lyon, France*, pages 121–137, Septembre 2003.

CHAPTER

TWO

INTRODUCTION TO COMPUTER ARITHMETIC AND ROUNDING ERROR ANALYSIS

In this chapter, we provide a short introduction to computer arithmetic (especially floating-point arithmetic) and we also present some methods to perform rounding error analysis. We will particularly focus on forward/backward error analysis, interval analysis and stochastic analysis. These are, indeed, the methods we will use to prove the accuracy of our algorithms and to validate or certify some properties of those algorithms. This chapter corresponds mainly to some parts of our paper [16] written for the *Encyclopedia of Computer Science and Engineering*.

2.1 Introduction

In the introduction of this document, we have shown that a good understanding of floating-point arithmetic is necessary to avoid some catastrophic consequences (example of the Patriot missile). It is then important to know how the computations in floating-point arithmetic work. Moreover, it is necessary to have some tools making it possible to get some informations about the numerical behavior of a scientific code. For that, there exist some methods that can help to evaluate the impact of rounding errors on the final results.

In Section 2.2, we present some computer arithmetics, mainly the floating-arithmetic described in the IEEE 754 standard. In Section 2.3, we describe some methods to analyse rounding errors. We present mainly forward/backward error analysis, interval analysis and stochastic analysis.

2.2 Computer arithmetic

2.2.1 Representation of numbers

In a position number system, numbers are represented by a sequence of symbols. The number of distinct symbols which can be used is called the radix (or the base). In the binary

system, which is used on most computers, the radix is 2; hence numbers are represented with sequences of 0s and 1s.

Several formats exist to represent numbers on a computer. The representation of integer numbers differs from the one of real numbers. Using a radix b, if unsigned integers are encoded on n digits, they can range from 0 to $b^n - 1$. Hence an unsigned integer X is represented by a sequence $a_{n-1}a_{n-2}...a_1a_0$ with $X = \sum_{i=0}^{n-1} a_i b^i$ and $a_i \in \{0,...,b-1\}$. With a radix 2 representation, signed integers are usually represented using two's complement. With this rule, signed integers range from $-b^{n-1}$ to $b^{n-1} - 1$ and the sequence $a_{n-1}a_{n-2}...a_1a_0$ with $a_i \in \{0,...,b-1\}$ represents the number $X = -a_{n-1}b^{n-1} + \sum_{i=0}^{n-2} a_i b^i$. The opposite of a number in two's complement format can be obtained by inverting each bit and adding 1.

In numerical computations, most real numbers are not exactly represented because only a finite number of digits can be stored in memory. Two representations exist for real numbers:

- the fixed-point format, available on most embedded systems
- the floating-point format, available on classical computers.

In fixed-point arithmetic, a number is represented with a fixed number of digits before and after the radix point. Using a radix b, a number X which is encoded on m digits for its magnitude (*e.g.* its integer part) and f digits for its fractional part is represented by $a_{m-1}...a_0 \,.\, a_{-1}...a_{-f}$, with $X = \sum_{i=-f}^{m-1} a_i b^i$ and $a_i \in \{0,...,b-1\}$.

If $b = 2$, unsigned values range from 0 to $2^m - 2^{-f}$ and signed values, which are usually represented with the two's complement format, range from -2^{m-1} to $2^{m-1} - 2^{-f}$.

In a floating-point arithmetic using radix b, a number X is represented by:

- its sign ε_X which is encoded on one digit that equals 0 if $\varepsilon_X = 1$ and 1 if $\varepsilon_X = -1$,
- its exponent E_X, a k digit integer,
- its significand M_X, encoded on p digits.

Therefore $X = \varepsilon_X M_X b^{E_X}$ with $M_X = \sum_{i=0}^{p-1} a_i b^{-i}$ and $a_i \in \{0,...,b-1\}$. The mantissa M_X can be written as $M_X = a_0 \,.\, a_1 ... a_{p-1}$.

Floating-point numbers are usually normalized. In this case, $a_0 \neq 0$, $M_X \in [1, b)$ and the number zero has a special representation. Normalization presents several advantages, such as the uniqueness of the representation (there is exactly one way to write a number in such a form) and the easiness of comparisons (the signs, exponents and mantissas of two normalized numbers can be tested separately).

2.2.2 The IEEE 754 standard

At the beginning, most computers had different characteristics for their floating-point arithmetic. As a consequence, simulation programs could provide different results from one computer to another, because of different floating-point representations. Indeed different values could be used for the radix, the width of the exponent, the width of the mantissa, etc. So, in 1985, the IEEE 754 standard [34] was elaborated to define floating-point formats and rounding modes. It was revised in 2008 into the IEEE 754-2008 standard [1]. It mainly specifies two basic formats, both using radix 2.

- With the *single precision* format (binary32), numbers are stored on 32 bits: 1 for the sign, 8 for the exponent and 23 for the significand.

- With the *double precision* format (binary64), numbers are stored on 64 bits: 1 for the sign, 11 for the exponent and 52 for the significand.

Extended floating-point formats also exist; the standard specifies a binary128 format.

Because of the normalization, the first bit in the mantissa must be 1. As this implicit bit is not stored, the precision of the mantissa is actually 24 bits in single precision and 53 bits in double precision.

The exponent E is a k digit signed integer. Let us denote its bounds by E_{min} and E_{max}. The exponent which is actually stored is a biased exponent E_Δ such that $E_\Delta = E + \Delta$, Δ being the bias. Table 2.1 specifies how the exponent is encoded.

precision	length	bias	non-biased		biased	
	k	Δ	E_{min}	E_{max}	$E_{min}+\Delta$	$E_{max}+\Delta$
single	8	127	-126	127	1	254
double	11	1023	-1022	1023	1	2046

Table 2.1: Exponent coding in single and double precision

The number zero is encoded by setting to 0 all the bits of the (biased) exponent and all the bits of the mantissa. Two representations actually exist for zero: $+0$ if the sign bit is 0 and -0 if the sign bit is 1. This distinction is consistent with the existence of two infinities. Indeed $1/(+0) = +\infty$ and $1/(-0) = -\infty$. These two infinities are encoded by setting to 1 all the bits of the (biased) exponent and to 0 all the bits from the mantissa. The corresponding non-biased exponent is therefore $E_{max}+1$.

NaN (Not a Number) is a special value which represents the result of an invalid operation such as $0/0$, $\sqrt{-1}$ or $0\times\infty$. NaN is encoded by setting all the bits of the (biased) exponent to 1 and the fractional part of the mantissa to any non-zero value.

Denormalized numbers (also called subnormal numbers) represent values close to zero. Without them, as the integer part of the mantissa is implicitly set to 1, there would be no representable number between 0 and $2^{E_{min}}$ but 2^{p-1} representable numbers between $2^{E_{min}}$ and $2^{E_{min}+1}$. Denormalized numbers have a biased exponent set to 0. The corresponding values are: $X = \varepsilon_X M_X 2^{E_{min}}$ with $\varepsilon_X = \pm 1$, $M_X = \sum_{i=1}^{p-1} a_i 2^{-i}$ and $a_i \in \{0,1\}$. The mantissa M_X can be written as $M_X = 0.a_1 ... a_{p-1}$. Therefore the lowest positive denormalized number is $\underline{u} = 2^{E_{min}+1-p}$.

Let us denote by $\mathbf{F}$ the set of all floating-point numbers, *i.e.* the set of all machine representable numbers. This set, which depends on the chosen precision, is bounded and discrete. Let us denote its bounds by X_{min} and X_{max}. Let x be a real number which *is not machine representable*. If $x \in (X_{min}, X_{max})$, there exists $\{X^-, X^+\} \subset \mathbf{F}^2$ such that $X^- < x < X^+$ and $(X^-, X^+) \cap \mathbf{F} = \emptyset$. A rounding mode is a rule which, from x, provides X^- or X^+. This rounding occurs at each assignment and at each arithmetic operation. The IEEE 754 standard imposes a correct rounding for all arithmetic operations ($+$, $-$, $\times$, $/$) and also for the square root. The result must be the same as the one obtained with infinite precision and then rounded. The IEEE 754 standard defines four rounding modes:

- rounding towards $+\infty$ (or upward rounding), x is represented by X^+,
- rounding towards $-\infty$ (or downward rounding), x is represented by X^-,

- rounding towards 0, if x is negative, then it is represented by X^+, if x is positive, then it is represented by X^-,
- rounding to the nearest, x is represented by its nearest machine number. If x is at the same distance of X^- and X^+, it is represented by the machine number which has a mantissa ending with a zero. With this rule, rounding is said to be *tie to even*.

Let us denote by X the number obtained by applying one of these rounding modes to x. By definition, an *overflow* occurs if $|X| > \max\{|Y| : Y \in \mathbf{F}\}$ and an *underflow* occurs if $0 < |X| < \min\{|Y| : 0 \neq Y \in \mathbf{F}\}$ and $x \neq X$ (the operation is inexact). Gradual underflow denotes the situation where a number is not representable as a normalized number, but still as a denormalized one.

The reference for floating-point arithmetic is [50].

2.2.3 Rounding error formalization

2.2.3.1 Notion of exact significant digits

In order to correctly quantify the accuracy of a computed result, the notion of exact significant digits must be formalized. Let R be a computed result and r the corresponding exact result. The number $C_{R,r}$ of exact significant decimal digits of R is defined as the number of significant digits which are in common with r:

$$C_{R,r} = \log_{10} \left| \frac{R+r}{2(R-r)} \right|. \tag{2.1}$$

This mathematical definition is in accordance with the intuitive idea of decimal significant digits in common between two numbers. Indeed Equation (5.1) is equivalent to

$$|R-r| = \left| \frac{R+r}{2} \right| 10^{-C_{R,r}}. \tag{2.2}$$

If $C_{R,r} = 3$, the relative error between R and r is of the order of 10^{-3}. Then R and r have therefore three common decimal digits.

2.2.3.2 Rounding error occurring at each operation

A formalization of rounding errors generated by assignments and arithmetic operations is proposed below. Let X be the representation of a real number x in a floating-point arithmetic respecting the IEEE 754 standard. This floating-point representation of X may be written as $X = \mathrm{fl}(x)$.

The *machine epsilon* is the distance ϵ from 1.0 to the next larger floating-point number. Clearly, $\epsilon = 2^{1-p}$, p being the length of the mantissa including the implicit bit. The relative error on X is no larger than the *unit round-off* $\mathbf{u}$:

$$X = x(1+\delta) \text{ with } |\delta| \leqslant \mathbf{u} \tag{2.3}$$

where $\mathbf{u} = \epsilon/2$ with rounding to the nearest and $\mathbf{u} = \epsilon$ with the other rounding modes.

To carry out rounding error analysis of an algorithm we need to make some assumptions about the accuracy of the basic arithmetic operations. For that, we will use the standard model of arithmetic. For each operation $\circ \in \{+,-,\times,/\}$, the computed result satisfies

$$\mathrm{fl}(x \circ y) = (x \circ y)(1+\delta), \quad |\delta| \leqslant \mathbf{u}.$$

The computed operation are often written with a circle $\{\oplus, \ominus, \otimes, \oslash\}$. This model ignores the possibility of underflow. To take it into account, the model can be modified as follows,

$$\mathrm{fl}(x \circ y) = (x \circ y)(1+\delta) + \eta, \quad |\delta| \leqslant \mathbf{u}, \quad \eta \leqslant \underline{u}/2.$$

We always have $\delta\eta = 0$. If there is underflow then $\delta = 0$ else $\eta = 0$. For addition and subtraction, we always have $\eta = 0$. Indeed, if x and y are floating-point numbers and $\mathrm{fl}(x \pm y)$ underflows then $\mathrm{fl}(x \pm y) = x \pm y$.

2.2.3.3 Rounding error propagation

A numerical program is a sequence of arithmetic operations. The result R provided by a program after n operations or assignments can be modelled to the first order in 2^{-p} as:

$$R \approx r + \sum_{i=1}^{n} g_i(d) 2^{-p} \alpha_i \tag{2.4}$$

where r is the exact result, p is the number of bits in the mantissa, α_i are independent uniformly distributed random variables on $[-1,1]$ and $g_i(d)$ are coefficients depending exclusively on the data d and on the code [14].

The number $C_{R,r}$ of exact significant bits of the computed result R is

$$C_{R,r} = \log_2 \left| \frac{R+r}{2(R-r)} \right|. \tag{2.5}$$

$$C_{R,r} \approx -\log_2 \left| \frac{R-r}{r} \right| = p - \log_2 \left| \sum_{i=1}^{n} g_i(d) \frac{\alpha_i}{r} \right|. \tag{2.6}$$

The last term in Equation (2.6) represents the loss of accuracy in the computation of R. This term is independent of p. Therefore, assuming that the model at the first order established in Equation (2.4) is valid, the loss of accuracy in a computation is independent of the precision used.

2.3 Methods for rounding error analysis

In this section, different methods of analyzing rounding errors are reviewed.

2.3.1 Forward/Backward analysis

This subsection is heavily inspired from [32, 43]. The historical reference is [76]. Other important references are [13, 61].

Let X be an approximation to a real number x. The two common measures of the accuracy of X are its *absolute error*

$$E_a(X) = |x - X|, \tag{2.7}$$

and its *relative error*

$$E_r(X) = \frac{|x - X|}{|x|} \tag{2.8}$$

(which is undefined if $x = 0$). When x and X are vectors, the relative error is usually defined with a norm as $\|x - X\|/\|x\|$. This is a *normwise relative error*. A more widely used relative error is the *componentwise relative error* defined by $\max_i \frac{|x_i - X_i|}{|x_i|}$. It makes it possible to put the individual relative errors on an equal footing.

2.3.1.1 Well-posed problems

Let us consider the following mathematical problem (P)

$$(P) : \text{given } y, \text{ find } x \text{ such that } F(x) = y,$$

where F is a continuous mapping between two linear spaces (in general $\mathbf{R}^n$ or $\mathbf{C}^n$). One will say that the problem (P) is *well-posed* in the sense of Hadamard if the solution $x = F^{-1}(y)$ exists, is unique and F^{-1} is continuous in the neighborhood of y. If it is not the case, one says that the problem is *ill-posed*. An example of ill-posed problem is the solution of a linear system $Ax = b$ where A is singular. It is difficult to deal numerically with ill-posed problems (this is generally done via regularization techniques, see Chapter 4 for working with ill-posed problems). That is why we will focus only on well-posed problems in the sequel.

2.3.1.2 Conditioning

Given a well-posed problem (P), one wants now to know how to measure the difficulty of solving this problem. This will be done via the notion of *condition number*. Roughly speaking, the condition number measures the sensitivity of the solution to perturbation in the data. Since the problem (P) is well-posed, one can write it as $x = G(y)$ with $G = F^{-1}$.

The input space (data) and the output space (result) are denoted respectively by $\mathscr{D}$ and $\mathscr{R}$; the norms on these spaces will be denoted $\|\cdot\|_{\mathscr{D}}$ and $\|\cdot\|_{\mathscr{R}}$. Given $\varepsilon > 0$, let $\mathscr{P}(\varepsilon) \subset \mathscr{D}$ be a set of perturbation Δy of the data y satisfying $\|\Delta y\|_{\mathscr{D}} \leqslant \varepsilon$, the perturbed problem associated with problem (P) is defined by

$$\text{Find } \Delta x \in \mathscr{R} \text{ such that } F(x + \Delta x) = y + \Delta y \text{ for a given } \Delta y \in \mathscr{P}(\varepsilon).$$

x and y are assumed to be non-zero. The *condition number* of the problem (P) in the data y is defined by

$$\text{cond}(P, y) := \lim_{\varepsilon \to 0} \sup_{\Delta y \in \mathscr{P}(\varepsilon), \Delta y \neq 0} \left\{ \frac{\|\Delta x\|_{\mathscr{R}}}{\|\Delta y\|_{\mathscr{D}}} \right\}. \tag{2.9}$$

Example (summation). Let us consider the problem of computing the sum $x = \sum_{i=1}^{n} y_i$ assuming that $y_i \neq 0$ for all i. One will take into account the perturbation of the input data that are the coefficients y_i. Let $\Delta y = (\Delta y_1, \ldots, \Delta y_n)$ be the perturbation on $y = (y_1, \ldots, y_n)$. It follows that $\Delta x = \sum_{i=1}^{n} \Delta y_i$. Let us endow $\mathscr{D} = \mathbf{R}^n$ with the relative norm

$$\|\Delta y\|_{\mathscr{D}} = \max_{i=1,\ldots,n} |\Delta y_i|/|y_i|$$

and $\mathscr{R} = \mathbf{R}$ with the relative norm $\|\Delta x\|_{\mathscr{R}} = |\Delta x|/|x|$. Since

$$|\Delta x| = |\sum_{i=1}^{n} \Delta y_i| \leqslant \|\Delta y\|_{\mathscr{D}} \sum_{i=1}^{n} |y_i| \,[1]$$

one has

$$\frac{\|\Delta x\|_{\mathscr{R}}}{\|\Delta y\|_{\mathscr{D}}} \leqslant \frac{\sum_{i=1}^{n} |y_i|}{|\sum_{i=1}^{n} y_i|}. \tag{2.10}$$

This bound is reached for the perturbation Δy such that $\Delta y_i / y_i = \text{sign}(y_i)\|\Delta y\|_{\mathscr{D}}$ where sign is the sign of a real number. As a consequence,

$$\text{cond}\left(\sum_{i=1}^{n} y_i\right) = \frac{\sum_{i=1}^{n} |y_i|}{|\sum_{i=1}^{n} y_i|}. \tag{2.11}$$

Now one has to interpret this condition number. A problem is considered as ill-conditioned if it has a large condition number. Otherwise, it is well-conditioned. It is difficult to give a precise frontier between well-conditioned and ill-conditioned problems. This statement will be clarified in 2.3.1.4 thanks to the rule of thumb. The larger the condition number is, the more a small perturbation on the data can imply a greater error on the result. Nevertheless, the condition number measures the *worst case* implied by a small perturbation. As a consequence, it is possible for an ill-conditioned problem that a small perturbation on the data also implies a small perturbation on the result. Sometimes, such a behavior is even typical.

Remark. It is important to note that the condition number is independent of the algorithm used to solve the problem. It is only a characteristic of the problem.

2.3.1.3 Stability of an algorithm

Problems are generally solved using an *algorithm*. This is a set of operations and tests similar to the function G defined above given the solution of our problem. Because of the rounding errors, the algorithm is not the function G itself but rather a function $\widehat{G}$. Therefore, the algorithm does not compute $x = G(y)$ but $\widehat{x} = \widehat{G}(y)$.

The *forward analysis* tries to study the execution of the algorithm $\widehat{G}$ on the data y. Following the propagation of the rounding errors in each intermediate variables, the forward analysis tries to estimate or to bound the difference between x and $\widehat{x}$. This difference between the exact solution x and the computed solution $\widehat{x}$ is called the *forward error*.

1. the Cauchy-Schwarz inequality $|\sum_{i=1}^{n} x_i y_i| \leqslant \max_{i=1,\ldots,n} |x_i| \times \sum_{i=1}^{n} |y_i|$ is used

It is easy to see that it is pretty difficult to follow the propagation of all the intermediate rounding errors. The *backward analysis* makes it possible to avoid this problem by working with the function G itself. The idea is to seek for a problem that is actually solved and to check if this problem is "close to" the initial one. Basically, one tries to put the error on the result as an error on the data. More theoretically, one seeks for Δy such that $\widehat{x} = G(y + \Delta y)$. Δy is said to be the *backward error* associated with $\widehat{x}$. A backward error measures the distance between the problem that is solved and the initial problem. As $\widehat{x}$ and G are known, it is often possible to obtain a good upper bound for Δy (generally, it is easier than for the forward error). Figure 2.1 sums up the principle of the forward and backward analysis.

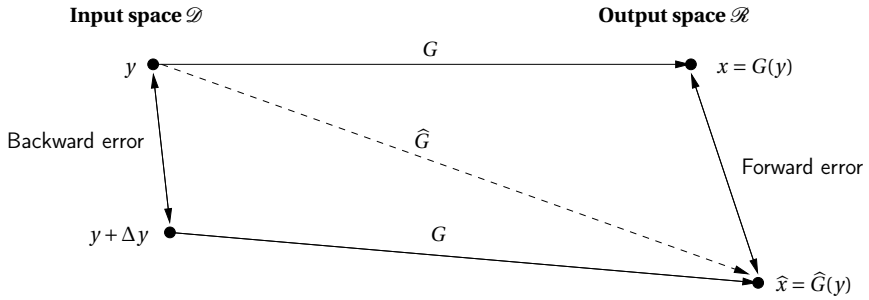

Figure 2.1: Forward and backward error for the computation of $x = G(y)$.

Sometimes, it is not possible to have $\widehat{x} = G(y + \Delta y)$ for some Δy but it is often possible to get Δx and Δy such that $\widehat{x} + \Delta x = G(y + \Delta y)$. Such a relation is called a *mixed forward-backward error*.

The *stability* of an algorithm describes the influence of the computation in finite precision on the quality of the result. The backward error associated with $\widehat{x} = \widehat{G}(y)$ is the scalar $\eta(\widehat{x})$ defined by, when it exists,

$$\eta(\widehat{x}) = \min_{\Delta y \in \mathscr{D}} \{\|\Delta y\|_{\mathscr{D}} : \widehat{x} = G(y + \Delta y)\}. \tag{2.12}$$

If it does not exist, $\eta(\widehat{x})$ is set to $+\infty$. An algorithm is said to be *backward-stable* for the problem (P) if the computed solution $\widehat{x}$ has a "small" backward error $\eta(\widehat{x})$. In general, in finite precision, "small" means of the order of the rounding unit $\mathbf{u}$.

Let us study an example for summation. The addition is supposed to satisfy the following property:

$$\widehat{z} = z(1 + \delta) = (a + b)(1 + \delta), \quad \text{with } |\delta| \leqslant \mathbf{u}. \tag{2.13}$$

It should be noticed that this assumption is satisfied by the IEEE arithmetic. The following algorithm to compute the sum $\sum y_i$ will be used.

Algorithm 2.1. Computation of the sum of floating-point numbers

function `res` = `Sum`(y)
  $s_1 = y_1$
  for $i = 2 : n$
    $s_i = s_{i-1} \oplus y_i$
  `res` = s_n

Thanks to relation (2.13), one can write

$$s_i = (s_{i-1} + y_i)(1+\delta_i) \quad \text{with } |\delta_i| \leqslant \mathbf{u}. \tag{2.14}$$

For convenience, $1+\theta_j = \prod_{i=1}^{j}(1+\varepsilon_i)$ is written, for $|\varepsilon_i| \leqslant \mathbf{u}$ and $j \in \mathbf{N}$. Iterating the previous equation yields

$$\texttt{res} = y_1(1+\theta_{n-1}) + y_2(1+\theta_{n-1}) + y_3(1+\theta_{n-2}) + \cdots + y_{n-1}(1+\theta_2) + y_n(1+\theta_1). \tag{2.15}$$

One can interpret the computed sum as the exact sum of the vector z with $z_i = y_i(1+\theta_{n+1-i})$ for $i = 2:n$ and $z_1 = y_1(1+\theta_{n-1})$.

As $|\varepsilon_i| \leqslant \mathbf{u}$ for all i and assuming $n\mathbf{u} < 1$, it can be proved that $|\theta_i| \leqslant i\mathbf{u}/(1-i\mathbf{u})$ for all i. Consequently, one can conclude that the backward error satisfies

$$\eta(\widehat{x}) = |\theta_{n-1}| \lesssim n\mathbf{u}. \tag{2.16}$$

Since the backward error is of the order of $\mathbf{u}$, one concludes that the classic summation algorithm is backward-stable.

2.3.1.4 Accuracy of the solution

How is the accuracy of the computed solution estimated? The accuracy of the computed solution actually depends on the condition number of the problem and on the stability of the algorithm used. The condition number measures the effect of the perturbation of the data on the result. The backward error simulates the errors introduced by the algorithm as errors on the data. As a consequence, at the first order, one has the following *rule of thumb*:

$$\boxed{\text{forward error} \lesssim \text{condition number} \times \text{backward error.}} \tag{2.17}$$

If the algorithm is backward-stable (that is to say the backward error is of the order of the rounding unit $\mathbf{u}$) then the rule of thumb can be written as follows

$$\text{forward error} \lesssim \text{condition number} \times \mathbf{u}. \tag{2.18}$$

In general, the condition number is hard to compute (as hard as the problem itself). As a consequence, there are some estimators that make it possible to compute an approximation of the condition number with a reasonable complexity.

The rule of thumb makes it possible to be more precise about what was called ill-conditioned and well-conditioned problems. A problem will be said to be ill-conditioned if the condition number is greater than $1/\mathbf{u}$. It means that the relative forward error is greater than 1 just saying that one has no accuracy at all for the computed solution.

In fact, in some cases, the rule of thumb can be proved. For the summation, if one denotes by $\widehat{s}$ the computed sum of the vector y_i, $1 \leqslant i \leqslant n$ and $s = \sum_{i=1}^{n} y_i$ the real sum, then (2.15) implies

$$\frac{|\widehat{s}-s|}{|s|} \leqslant \gamma_{n-1}\,\mathrm{cond}\left(\sum_{i=1}^{n} y_i\right) \tag{2.19}$$

with γ_n defined by

$$\gamma_n := \frac{n\mathbf{u}}{1-n\mathbf{u}} \quad \text{for } n \in \mathbf{N}. \tag{2.20}$$

Since $\gamma_{n-1} \approx (n-1)\mathbf{u}$, it is almost the rule of thumb with just a small factor $n-1$ before $\mathbf{u}$.

2.3.1.5 The LAPACK library

The LAPACK library [4] is a collection of subroutines in Fortran 77 designed to solve major problems in linear algebra: linear systems, least square systems, eigenvalues and singular values problems.

One of the most important advantages of LAPACK is that it provides error bounds for all the computed quantities. These error bounds are not rigorous but are mostly reliable. To do this, LAPACK uses the principles of backward analysis. In general, LAPACK provides both componentwise and normwise relative error bounds using the rule of thumb (2.17).

In fact, the major part of the algorithms implemented in LAPACK are backward-stable meaning that the rule of thumb (2.18) is satisfied. As the condition number is generally very hard to compute, LAPACK uses estimators. It may happen that the estimator is far from the right condition number. In fact, the estimation can arbitrarily be far from the true condition number. The error bounds in LAPACK are only qualitative markers of the accuracy of the computed results.

Linear algebra problems are central in current scientific computing. Getting some good error bounds is therefore very important and is still a challenge.

2.3.2 Interval arithmetic

Interval arithmetic [2, 41] is not defined on real numbers, but on closed bounded intervals. The result of an arithmetic operation between two intervals, $X = [\underline{x}, \overline{x}]$ and $Y = [\underline{y}, \overline{y}]$, contains all values that can be obtained by performing this operation on elements from each interval. The arithmetic operations are defined below.

$$X + Y = [\underline{x} + \underline{y},\ \overline{x} + \overline{y}]. \tag{2.21}$$

$$X - Y = [\underline{x} - \overline{y},\ \overline{x} - \underline{y}]. \tag{2.22}$$

$$X \times Y = [\min(\underline{x}\times\underline{y}, \underline{x}\times\overline{y}, \overline{x}\times\underline{y}, \overline{x}\times\overline{y}),\ \max(\underline{x}\times\underline{y}, \underline{x}\times\overline{y}, \overline{x}\times\underline{y}, \overline{x}\times\overline{y})]. \tag{2.23}$$

$$X^2 = [\min(\underline{x}^2, \overline{x}^2),\ \max(\underline{x}^2, \overline{x}^2)] \text{ if } 0 \notin [\underline{x},\ \overline{x}], \quad [0, \max(\underline{x}^2, \overline{x}^2)] \text{ otherwise.} \tag{2.24}$$

$$1/Y = [\min(1/\underline{y}, 1/\overline{y}),\ \max(1/\underline{y}, 1/\overline{y})] \text{ if } 0 \notin [\underline{y}, \overline{y}]. \tag{2.25}$$

$$X/Y = [\underline{x},\ \overline{x}] \times (1/[\underline{y},\ \overline{y}]) \text{ if } 0 \notin [\underline{y},\ \overline{y}]. \tag{2.26}$$

Arithmetic operations can also be applied to interval vectors and interval matrices by performing scalar interval operations componentwise.

An interval extension of a function f must provide all values that can be obtained by applying the function to any element of the interval argument X:

$$\forall x \in X,\ f(x) \in f(X). \tag{2.27}$$

For instance, $\exp[\underline{x}, \overline{x}] = [\exp \underline{x}, \exp \overline{x}]$ and $\sin[\pi/6, 2\pi/3] = [1/2, 1]$.

The interval obtained may depend on the formula chosen for mathematically equivalent expressions. For instance, let $f_1(x) = x^2 - x + 1$. $f_1([-2,1]) = [-2,7]$. Let $f_2(x) = (x - 1/2)^2 + 3/4$. The function f_2 is mathematically equivalent to f_1, but $f_2([-2,1]) = [3/4, 7] \neq f_1([-2,1])$. One

can notice that $f_2([-2,1]) \subseteq f_1([-2,1])$. Indeed a power set evaluation is always contained in the intervals resulting from other mathematically equivalent formulas.

Interval arithmetic enables one to control rounding errors automatically. On a computer, a real value which is not machine representable can be approximated by a floating-point number. It can also be enclosed by two floating-point numbers. Real numbers can therefore be replaced by intervals with machine representable bounds. An interval operation can be performed using directed rounding modes, in such a way that the rounding error is taken into account and the exact result is necessarily contained in the computed interval. For instance, the computed results, with guaranteed bounds, of the addition and the subtraction between two intervals $X = [\underline{x}, \overline{x}]$ and $Y = [\underline{y}, \overline{y}]$ are

$$X + Y = [\nabla(\underline{x} + \underline{y}), \Delta(\overline{x} + \overline{y})] \supseteq \{x + y | x \in X, y \in Y\} \tag{2.28}$$

$$X - Y = [\nabla(\underline{x} - \overline{y}), \Delta(\overline{x} - \underline{y})] \supseteq \{x - y | x \in X, y \in Y\} \tag{2.29}$$

where ∇ (respectively Δ) denotes the downward (respectively upward) rounding mode.

Interval arithmetic has been implemented in several libraries or softwares like, for instance, C-XSC [2] [33], a C++ class library, or INTLAB [3] [31, 58], a Matlab toolbox.

The main advantage of interval arithmetic is its reliability. But the intervals obtained may be too large. The intervals width regularly increases with respect to the intervals that would have been obtained in exact arithmetic. With interval arithmetic, rounding error compensation is not taken into account.

The overestimation of the error can also be due to the loss of information on variable dependency. In interval arithmetic, several occurrences of the same variable are considered as different variables. For instance, let $X = [1,2]$,

$$\forall x \in X,\ x - x = 0, \tag{2.30}$$

but

$$X - X = [-1,1]. \tag{2.31}$$

Another source of overestimation is the "wrapping effect" due to the enclosure of a non-interval shape range into an interval. For instance, the image of the square $[0,\sqrt{2}] \times [0,\sqrt{2}]$ by the function

$$f(x,y) = \frac{\sqrt{2}}{2}(x+y, y-x) \tag{2.32}$$

is the rotated square S_1 with corners $(0,0),(1,-1),(2,0),(1,1)$. The square S_2 provided by interval arithmetic operations is: $f([0,\sqrt{2}],[0,\sqrt{2}]) = ([0,2],[-1,1])$. The area obtained with interval arithmetic is twice the one of the rotated square S_1.

As the classical numerical algorithms can lead to over-pessimistic results in interval arithmetic, specific algorithms, suited for interval arithmetic, have been proposed (for example for an accurate inclusion for the determinant of a matrix in [61, p.214]).

2. http://www.xsc.de
3. http://www.ti3.tu-harburg.de/rump/intlab

2.3.3 Probabilistic approach

Here, a method for estimating rounding errors is presented. For the mathematical model, remember the formula at the first order (2.4). Concretely, the rounding mode of the computer is replaced by a random rounding mode, *i.e.* at each elementary operation, the result is rounded towards $-\infty$ or $+\infty$ with probability 0.5. The main interest of this new rounding mode is to run a same binary code with different rounding error propagations because one generates for different runs different random draws. If rounding errors affect the result, even slightly, for N different runs, one obtains N different results on which a statistical test may be applied. This is the basic idea of the CESTAC method (Contrôle et Estimation STochastique des Arrondis de Calcul). Briefly, the part of the N mantissas that is common to the N results is assumed not to be affected by rounding errors, contrary to the part of the N mantissas which is different from one result to an other.

The implementation of the CESTAC method in a code providing a result R consists in:

- executing N times this code with the random rounding mode, which is obtained by using randomly the rounding mode towards $-\infty$ or $+\infty$; then, an N-sample (R_i) of R is obtained,
- choosing as the computed result the mean value $\overline{R}$ of R_i, $i = 1, ..., N$,
- estimating the number of exact decimal significant digits of $\overline{R}$ with

$$C_{\overline{R}} = \log_{10}\left(\frac{\sqrt{N}\left|\overline{R}\right|}{\sigma\tau_\beta}\right), \tag{2.33}$$

where

$$\overline{R} = \frac{1}{N}\sum_{i=1}^{N} R_i \quad \text{and} \quad \sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(R_i - \overline{R}\right)^2. \tag{2.34}$$

τ_β is the value of Student's distribution for $N-1$ degrees of freedom and a probability level $1-\beta$.

From equation (2.4), if the first order approximation is valid, one may deduce that:

1. the mean value of the random variable R is the exact result r,
2. under some assumptions, the distribution of R is a quasi-Gaussian distribution.

It has been shown that $N = 3$ is an adequate choice. The estimation with $N = 3$ is more reliable than with $N = 2$ and increasing the size of the sample does not improve the quality of the estimation. The complete theory can be found in [15, 74].

This leads to the synchronous implementation of the method, *i.e.* to the parallel computation of the N results R_i. In this approach, a classical floating-point number is replaced by a 3-sample $X = (X_1, X_2, X_3)$ and an elementary operation $\Omega \in \{+, -, \times, /\}$ is defined by $X\Omega Y = (X_1 \omega Y_1, X_2 \omega Y_2, X_3 \omega Y_3)$ where ω represents the corresponding floating-point operation followed by a random rounding.

A new important concept has also been introduced: the computational zero.

Definition 2.1. During the run of a code using the CESTAC method, an intermediate or a final result R is a computational zero, denoted by @.0, if one of the two following conditions holds:

- $\forall i, R_i = 0$,
- $C_{\overline{R}} \leqslant 0$.

Any computed result R is a computational zero if either $R = 0$, R being significant, or R is non significant. In other words, a computational zero is a value that cannot be differentiated from the mathematical zero because of its rounding error. From this new concept of zero, one can deduce new order relationships that take into account the accuracy of intermediate results. For instance,

Definition 2.2. X is stochastically strictly greater than Y if and only if:

$$\overline{X} > \overline{Y} \quad \text{and} \quad X - Y \neq @.0.$$

or

Definition 2.3. X is stochastically greater than or equal to Y if and only if:

$$\overline{X} \geqslant \overline{Y} \quad \text{or} \quad X - Y = @.0.$$

The joint use of the CESTAC method and these new definitions is called DSA (Discrete Stochastic Arithmetic) [75]. Elements of the DSA, which are named *stochastic numbers*, are N-sets provided by the CESTAC method. DSA enables to estimate the impact of rounding errors on any result of a scientific code and also to check that no anomaly occurred during the run, especially in branching statements. The CADNA (Control of Accuracy and Debugging for Numerical Applications) software [4] [38] is a library which implements DSA in any code written in C++ or in Fortran and allows to use new numerical types: the stochastic types. The library contains the definition of all arithmetic operations and order relations for the stochastic types. The control of the accuracy is only performed on variables of stochastic type. When a stochastic variable is printed, only its exact significant digits appear. For a computational zero, the symbol @.0 is printed. The CADNA library allows, during the execution of any code:

- the estimation of the error due to rounding error propagation,
- the detection of numerical instabilities,
- the checking of the sequencing of the program (tests and branchings),
- the estimation of the accuracy of all intermediate computations.

4. http://www-pequan.lip6.fr/cadna/

CHAPTER

THREE

INCREASING THE ACCURACY OF NUMERICAL ALGORITHMS

This chapter corresponds mainly to some parts of our papers [16, 37, 29, 26, 25, 36].

3.1 Introduction

In this chapter, different methods that increase the accuracy of the computed result of an algorithm are presented. Far from being exhaustive, two classes of methods are presented. The first class is algorithms using multiprecision arithmetic. The second class is the class of compensated methods. These methods consist in estimating the rounding error of individual operations and then adding them later on to the computed result.

Throughout this chapter, one assumes that the floating-point arithmetic adhers to IEEE 754 floating-point standard, in rounding to the nearest. One also assume that no overflow nor underflow occurs.

In Section 3.2, we present the classic error-free transformations. They are a basis block for our compensated algorithms. They are also used in some multiprecision libraries we present in Section 3.3. We will show that the use of algorithms with multiprecision libraries is less efficient that using compensated algorithms for doubling the precision. Compensated algorithms for summation and dot product from Ogita, Rump and Oishi [51] are presented in Section 3.4. In Section 3.5, we present our compensated Horner scheme. In Section 3.6 we modify this algorithm to also compute accurately the derivatives of a polynomial. We use this algorithm to increase the accuracy of simple roots by Newton's iterations in Section 3.7. Section 3.8 is devoted to the opposite problem. Knowing the roots, we accurately compute the coefficients of the corresponding polynomial. Sometimes, more accuracy is needed. In Section 3.9, we present what a faithful rounding is. In Section 3.10, we propose an algorithm for computing a faithful rounding of the product of floating-point numbers.

3.2 Error-free transformations (EFT)

One can notice that $a \circ b \in \mathbf{R}$ for $\circ \in \{+,-,\times,/\}$ and $a \circledcirc b \in \mathbf{F}$ for $\circledcirc \in \{\oplus,\ominus,\otimes,\oslash\}$ but in general $a \circ b \in \mathbf{F}$ does not hold. It is known that for the basic operations $+,-,\times$ in rounding to nearest assuming no underflow nor overflow, the approximation error of a floating-point operation is still a floating-point number:

$$\begin{array}{lcll} x = a \oplus b & \Rightarrow & a + b = x + y & \text{with } y \in \mathbf{F}, \\ x = a \ominus b & \Rightarrow & a - b = x + y & \text{with } y \in \mathbf{F}, \\ x = a \otimes b & \Rightarrow & a \times b = x + y & \text{with } y \in \mathbf{F}, \\ x = a \oslash b & \Rightarrow & a = x \times b + y & \text{with } y \in \mathbf{F}, \\ x = \bigcirc\!\!\!\!\surd(a) & \Rightarrow & a = x^2 + y & \text{with } y \in \mathbf{F}. \end{array} \tag{3.1}$$

These are *error-free* transformations of the pair (a,b) into the pair (x,y). The floating-point number x is the result of the floating-point operation whereas y is the rounding term. Fortunately, the quantities x and y in (3.1) can be computed exactly in floating-point arithmetic. For the algorithms, Matlab-like notations are used. For addition, one can use the following algorithm by Knuth.

Algorithm 3.1 (Knuth [40])**.** Error-free transformation of the sum of two floating-point numbers

function $[x,y] = \texttt{TwoSum}(a,b)$
$x = a \oplus b$
$z = x \ominus a$
$y = (a \ominus (x \ominus z)) \oplus (b \ominus z)$

Another algorithm to compute an error-free transformation is the following algorithm from Dekker. The drawback of this algorithm is that $x + y = a + b$ provided that $|a| \geqslant |b|$. Generally, on modern computers, a comparison followed by a branching and 3 operations costs more than 6 operations. As a consequence, `TwoSum` is generally more efficient than `FastTwoSum` plus a branching.

Algorithm 3.2 (Dekker [17])**.** Error-free transformation of the sum of two floating-point numbers.

function $[x,y] = \texttt{FastTwoSum}(a,b)$
$x = a \oplus b$
$y = (a \ominus x) \oplus b$

Concerning the error-free transformation of a product, it can be written in a very simple way if a Fused-Multiply-and-Add (`FMA`) operator is available on the target architecture. Some computers have a *Fused-Multiply-and-Add* (`FMA`) operation that enables a floating-point multiplication followed by an addition to be performed as a single floating-point operation. The Intel IA-64 architecture, implemented in the Intel Itanium processor, has an `FMA` instruction as well as the IBM RS/6000 and the PowerPC before it. On the Itanium processor, the `FMA` instruction enables a multiplication and an addition to be performed in the same number

of cycles than one multiplication or one addition. As a result, it seems to be advantageous for speed as well as for accuracy. The FMA is now available on GPU, Intel Haswell and AMD Bulldozer processors.

Theoretically, this means that for given $a, b, c \in \mathbf{F}$, the result of $\mathtt{FMA}(a, b, c)$ is the nearest floating-point number of $a \times b + c \in \mathbf{R}$. The FMA satisfies

$$\mathtt{FMA}(a, b, c) = (a \times b + c)(1 + \varepsilon_1) = (a \times b + c)/(1 + \varepsilon_2) \text{ with } |\varepsilon_v| \leqslant \mathbf{u}.$$

Thanks to the FMA, the TwoProduct algorithm can be written as follows which costs only 2 flops.

Algorithm 3.3. Error-free transformation of the product of two floating-point numbers using an FMA.

function $[x, y] = \mathtt{TwoProduct}(a, b)$
  $x = a \otimes b$
  $y = \mathtt{FMA}(a, b, -x)$

If no FMA is available, one first needs to split the input argument into two parts. Let p be given by $\mathbf{u} = 2^{-p}$ and let us define $s = \lceil p/2 \rceil$. For example, if the working precision is IEEE 754 double precision, then $p = 53$ and $s = 27$. The following algorithm due to Dekker splits a floating-point number $a \in \mathbf{F}$ into two parts x and y such that

$$a = x + y \quad \text{with } |y| \leqslant |x|. \tag{3.2}$$

Both parts x and y have at most $s - 1$ non-zero bits.

Algorithm 3.4 (Dekker [17])**.** Error-free split of a floating-point number into two parts

function $[x, y] = \mathtt{Split}(a, b)$
  $\mathtt{factor} = 2^s \oplus 1$
  $c = \mathtt{factor} \otimes a$
  $x = c \ominus (c \ominus a)$
  $y = a \ominus x$

The main point of Split is that both parts can be multiplied in the same precision without error. With this function, an algorithm attributed to Veltkamp by Dekker enables to compute an error-free transformation for the product of two floating-point numbers. This algorithm returns two floating-point numbers x and y such that

$$a \times b = x + y \quad \text{with } x = a \otimes b. \tag{3.3}$$

Algorithm 3.5 (Veltkamp [17])**.** Error-free transformation of the product of two floating-point numbers

function $[x, y] = \texttt{TwoProduct}(a, b)$
  $x = a \otimes b$
  $[a_1, a_2] = \texttt{Split}(a)$
  $[b_1, b_2] = \texttt{Split}(b)$
  $y = a_2 \otimes b_2 \ominus (((x \ominus a_1 \otimes b_1) \ominus a_2 \otimes b_1) \ominus a_1 \otimes b_2)$

The performance of the algorithms is interpreted in terms of floating-point operations (flops). The following theorem summarizes the properties of algorithms `TwoSum` and `TwoProduct`.

Theorem 3.1 (Ogita, Rump, Oishi [51])**.** *Let $a, b \in \mathbf{F}$ and let $x, y \in \mathbf{F}$ be given such that $[x, y] =$ `TwoSum`(a, b) (Algorithm 3.1). Then,*

$$a + b = x + y, \quad x = a \oplus b, \quad |y| \leqslant \mathbf{u}|x|, \quad |y| \leqslant \mathbf{u}|a + b|. \tag{3.4}$$

The algorithm `TwoSum` *requires* 6 *flops.*
Let $a, b \in \mathbf{F}$ and let $x, y \in \mathbf{F}$ such that $[x, y] =$ `TwoProduct`(a, b) (Algorithm 3.5). Then,

$$a \times b = x + y, \quad x = a \otimes b, \quad |y| \leqslant \mathbf{u}|x|, \quad |y| \leqslant \mathbf{u}|a \times b|. \tag{3.5}$$

The algorithm `TwoProduct` *requires* 2 *flops with* `FMA` *and* 17 *flops otherwise.*

3.3 Multiple precision arithmetic

One possibility to increase the accuracy is to increase the working precision. For this purpose, some multiprecision libraries have been developed. One can divide the libraries into four categories.

Arbitrary precision libraries using a *multiple-digit* format

In these libraries a number is expressed as a sequence of digits coupled with a single exponent. Examples of this format are Bailey's ARPREC [1] [6], Brent's MP [2] [8] or MPFR [3] [22]. ARPREC and MPFR are briefly described below.

The ARPREC library, which is entirely written in C++, supports high-precision real, integer and complex datatypes. Both C++ and Fortran-90 translation modules modules are also provided that allow one to convert an existing C++ or Fortran-90 program to use the library with only minor changes to the source code. In most cases only the type statements and (in the case of Fortran-90 programs) read/write statements need be changed.

The ARPREC package also includes "The Experimental Mathematician's Toolkit", which is a complete interactive high-precision arithmetic computing environment. One enters

1. http://crd.lbl.gov/~dhbailey/mpdist/
2. http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub043.html or http://wwwmaths.anu.edu.au/~brent/pub/pub043.html
3. http://www.mpfr.org/

expressions in a Mathematica-style syntax, and the operations are performed using the ARPREC package, with a level of precision that can be set from 100 to 1000 decimal digit accuracy. This program supports all basic arithmetic operations, common transcendental and combinatorial functions, high-precision quadrature, summation of series,...

The MPFR library is written in C and is based on the GNU MP library (GMP for short). The internal representation of a floating-point number x by MPFR is

- a mantissa m;
- a sign s;
- a signed exponent e.

If the precision of x is p, then the mantissa m has p significant bits. The mantissa m is represented by an array of GMP unsigned machine-integer type and is interpreted as $1/2 \leqslant m < 1$. As a consequence, MPFR does not allow denormalized numbers.

MPFR provides the four IEEE rounding modes as well as some elementary functions (*e.g.* exp, log, cos, sin), all correctly rounded. The semantic in MPFR is as follows: for each instruction $a = b + c$ or $a = f(b,c)$ the variables may have different precisions. In MPFR, the data b and c are considered with their full precision and a correct rounding to the full precision of a is computed.

Applications using MPFR inherit the same properties as programs using the IEEE 754 standard (portability, well-defined semantics, possibility to design robust programs and prove their correctness) with no significant slowdown on average with respect to multiple precision libraries with ill-defined semantics.

Arbitrary precision libraries using a *multiple-component* format

In these libraries a number is expressed as unevaluated sums of ordinary floating-point words. Examples using this format are Priest's [4] and Shewchuk's [5] libraries. Such a format is also introduced in [66]. An expansion x is then a unevaluated sums of several floating point numbers

$$x = x_1 + x_2 + \cdots + x_m.$$

Each x_i is called a component of x and is a floating point number. To impose some structure on expansions, they are required to be nonoverlapping and ordered by magnitude ($|x_1| \leqslant |x_2| \leqslant \cdots \leqslant |x_m|$). Priest says that x and y nonoverlap (with $|x| \leqslant |y|$) if the weight of the most significant bit of x is less than the one of the least signicant of y. This is the main difference with Shewchuck's expansions where x and y nonoverlap if the weight of the most significant nonzero bit of x is less than the one of the least significant nonzero bit of y. Note that the number m of floating point numbers is not fixed. For exemple if $x = x_1 + x_2 + \cdots + x_m$ and $y = x_1 + x_2 + \cdots + x_n$ then the product xy is a expansion that can have up to nm terms.

4. ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z
5. http://www.cs.cmu.edu/~quake/robust.html

Fixed precision libraries using a *multiple-component* format

These libraries use the *multiple-component* format but with a limited number of components. Examples of this format are Bailey's *double-double*[1] (double-double numbers are represented as an unevaluated sum of a leading double and a trailing double) and *quad-double*[1].

The double-double library is presented now. For our purpose, it suffices to know that a double-double number a is a pair (a_h, a_l) of IEEE-754 double precision floating-point numbers that satisfies $a = a_h + a_l$ and $|a_l| \leqslant \mathbf{u}|a_h|$. In the sequel, algorithms for

- the addition of a double number and a double-double number;
- the product of a double-double number by a double number;
- the addition of a double-double number and a double-double number

will only be presented. Of course, it is also possible to implement the product of a double-double by a double-double as well as the division of a double-double by a double, etc.

Algorithm 3.6. Addition of the double number b and the double-double number (a_h, a_l)

function $[c_h, c_l] = \texttt{add\_dd\_d}(a_h, a_l, b)$
  $[t_h, t_l] = \texttt{TwoSum}(a_h, b)$
  $[c_h, c_l] = \texttt{FastTwoSum}(t_h, (t_l \oplus a_l))$

Algorithm 3.7. Product of the double-double number (a_h, a_l) by the double number b

function $[c_h, c_l] = \texttt{prod\_dd\_d}(a_h, a_l, b)$
  $[s_h, s_l] = \texttt{TwoProduct}(a_h, b)$
  $[t_h, t_l] = \texttt{FastTwoSum}(s_h, (a_l \otimes b))$
  $[c_h, c_l] = \texttt{FastTwoSum}(t_h, (t_l \oplus s_l))$

Algorithm 3.8. Addition of the double-double number (a_h, a_l) and the double-double number (b_h, b_l)

function $[c_h, c_l] = \texttt{add\_dd\_dd}(a_h, a_l, b_h, b_l)$
  $[s_h, s_l] = \texttt{TwoSum}(a_h, b_h)$
  $[t_h, t_l] = \texttt{TwoSum}(a_l, b_l)$
  $[t_h, s_l] = \texttt{FastTwoSum}(s_h, (s_l \oplus t_h))$
  $[c_h, c_l] = \texttt{FastTwoSum}(t_h, (t_l \oplus s_l))$

Algorithms 3.6 to 3.8 use error-free transformations and are very similar to compensated algorithms we will study in the next section. The difference lies in the step of renormalization. This step is the last one in each algorithm and makes it possible to ensure that $|c_l| \leqslant \mathbf{u}|c_h|$. This is why compensated algorithms are faster than the ones that use double-double library.

There are several implementations for the double-double library. The difference is that the lower-order terms are treated in a different way. If a, b are double-double numbers and $\circledcirc \in \{+, \times\}$, then one can show [44] that

$$\mathrm{fl}(a \circledcirc b) = (1 + \delta)(a \circledcirc b),$$

with $|\delta| \leqslant 4 \cdot 2^{-106}$.

The MPFI library for multiprecision interval arithmetic

MPFI (Multiple Precision Floating-point Interval) [6] [54] is intended to be a portable library written in C for arbitrary precision interval arithmetic with intervals represented using MPFR reliable floating-point numbers. It is based on the GNU MP library and on the MPFR library. The purpose of an arbitrary precision interval arithmetic is on the one hand to get guaranteed results, thanks to interval computation, and on the other hand to obtain accurate results, thanks to multiple precision arithmetic. The MPFI library is built upon MPFR in order to benefit from the correct roundings provided by MPFR. Further advantages of using MPFR are its portability and compliance with the IEEE 754 standard for floating-point arithmetic.

3.4 A compensated summation and dot product algorithm

Hereafter, a compensated scheme to evaluate the sum of floating-point numbers is presented, *i.e.* the error of individual summation is somehow corrected.

Indeed, with Algorithm 3.1 (`TwoSum`), one can compute the rounding error. This algorithm can be cascaded and sum up the errors to the ordinary computed summation. For a summary, see Figure 3.1 and Algorithm 3.9.

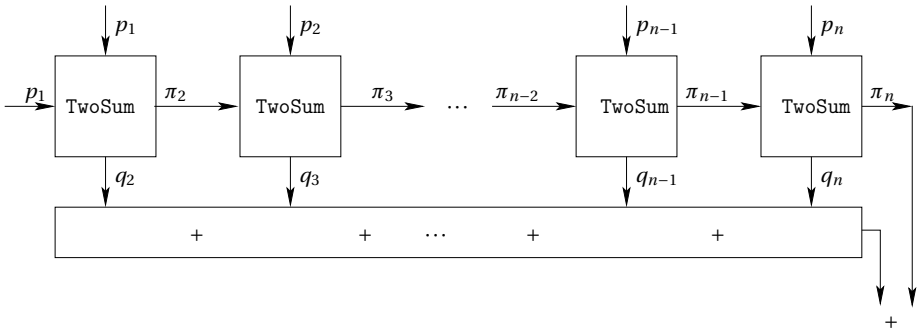

Figure 3.1: Compensated summation algorithm

Algorithm 3.9 (Ogita, Rump, Oishi [51])**.** Compensated summation algorithm

function `res` = `CompSum`(p)
  $\pi_1 = p_1$; $\sigma_1 = 0$;
  for $i = 2 : n$
    $[\pi_i, q_i] =$ `TwoSum`(π_{i-1}, p_i)
    $\sigma_i = \sigma_{i-1} \oplus q_i$
  `res` $= \pi_n \oplus \sigma_n$

The following proposition gives a bound on the accuracy of the result. The notation γ_n defined by Equation (2.20) will be used. When using γ_n, $n\mathbf{u} \leqslant 1$ is implicitly assumed.

6. http://perso.ens-lyon.fr/nathalie.revol/mpfi_toc.html

Proposition 3.1 (Ogita, Rump, Oishi [51]). *Suppose Algorithm* `CompSum` *is applied to floating-point number* $p_i \in \mathbf{F}$, $1 \leqslant i \leqslant n$. *Let* $s := \sum p_i$, $S := \sum |p_i|$ *and* $n\mathbf{u} < 1$. *Then, one has*

$$|\mathtt{res} - s| \leqslant \mathbf{u}|s| + \gamma_{n-1}^2 S. \tag{3.6}$$

In fact, the assertions of Proposition 3.1 are also valid in the presence of underflow. One can interpret Equation (3.6) in terms of the condition number for the summation (2.11). Since

$$\mathrm{cond}\left(\sum p_i\right) = \frac{\sum |p_i|}{\left|\sum p_i\right|} = \frac{S}{|s|}, \tag{3.7}$$

inserting this in Equation (3.6) yields

$$\frac{|\mathtt{res} - s|}{|s|} \leqslant \mathbf{u} + \gamma_{n-1}^2 \,\mathrm{cond}\left(\sum p_i\right). \tag{3.8}$$

Basically, the bound for the relative error of the result is essentially $(n\mathbf{u})^2$ times the condition number plus the rounding $\mathbf{u}$ due to the working precision. The second term on the right hand side reflects the computation in twice the working precision ($\mathbf{u}^2$) thanks to the *rule of thumb*. The first term reflects the rounding back in the working precision.

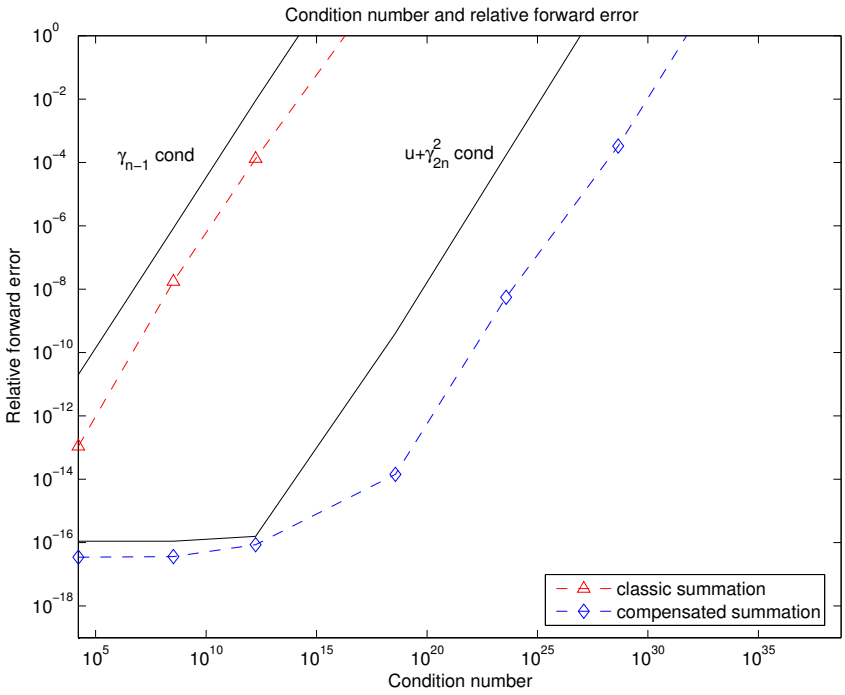

Figure 3.2: Compensated summation algorithm

The compensated summation on ill-conditioned sum was tested; the condition number varying from 10^4 to 10^{40}.

Figure 3.2 shows the relative accuracy $|\texttt{res} - s|/|s|$ of the computed value by the two algorithms 2.1 and 3.9. The *a priori* error estimations (2.19) and (3.11) are also plotted.

As one can see in Figure 3.2, the compensated summation algorithm exhibits the expected behavior, that is to say, the compensated rule of thumb (3.11). As long as the condition number is less than $\mathbf{u}^{-1}$, the compensated summation algorithm produces results with full precision (forward relative error of the order of $\mathbf{u}$). For condition numbers greater than $\mathbf{u}^{-1}$, the accuracy decreases and there is no accuracy at all for condition numbers greater than $\mathbf{u}^{-2}$.

Thanks to the `TwoProduct` algorithm, we already have an error-free transformation of a product of two floating-point numbers into the sum of two floating-point numbers. Combining `TwoProduct` with the compensated algorithm `CompSum`, we can provide a compensated algorithm for computing the dot product of two floating-point vectors x, y of length n. For each elementary operation (here addition and substraction), we can compute the rounding error thanks to error-free transformations. We can then accumulate the errors and add them to the final result. This is what is done in the following algorithm. We recall that given two vectors of size n of floating point numbers, the dot product is defined by

$$x^T y = \sum_{i=1}^{n} x_i y_i.$$

Algorithm 3.10 (Ogita, Rump, Oishi [51])**.** Dot product in twice the working precision

function $\texttt{res} = \texttt{CompDot2}(x, y)$
  $[p, s] = \texttt{TwoProduct}(x_1, y_1)$
  for $i = 2 : n$
    $[h, r] = \texttt{TwoProduct}(x_i, y_i)$
    $[p, q] = \texttt{TwoSum}(p, h)$
    $s = s \oplus (q \oplus r)$
  end
  $\texttt{res} = p \oplus s$

The following theorem gives an *a priori* error bound on the error.

Proposition 3.2 (Ogita, Rump, Oishi [51])**.** *Let floating point numbers $x_i, y_i \in \mathbf{F}, 1 \leqslant i \leqslant n$, be given and denote by $\texttt{res} \in \mathbf{F}$ the result computed by Algorithm* `CompDot2`*. Then occurs,*

$$|\texttt{res} - x^T y| \leqslant \mathbf{u}|x^T y| + \gamma_n^2 |x^T||y|. \tag{3.9}$$

One can interpret Equation (3.9) in terms of the condition number for dot product. Since

$$\operatorname{cond}\left(x^T y\right) = \frac{|x|^T |y|}{|x^T y|}, \tag{3.10}$$

inserting this in Equation (3.9) yields

$$\frac{|\texttt{res} - x^T y|}{|x^T y|} \leqslant \mathbf{u} + \gamma_n^2 \operatorname{cond}\left(x^T y\right). \tag{3.11}$$

As a consequence, we conclude that the result is as accurate as if computed in twice the working precision and then rounded to the current working precision. This is the same phenomena as for the compensated summation algorithm.

3.5 A compensated Horner scheme

We now want to accurately compute the evaluation of a polynomial at a given point. We present hereafter a compensated algorithm for Horner scheme. One can find a more detailed description of the algorithm in [30, 29]. We first recall the classic algorithm for Horner scheme and give a error bound. We then present the compensated Horner scheme together with an error bound.

The classical method for evaluating a polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

is the Horner scheme which consists on the following algorithm.

Algorithm 3.11. Polynomial evaluation with Horner's scheme

function `res` = `Horner`(p, x)
$s_n = a_n$
for $i = n-1 : -1 : 0$
  $s_i = s_{i+1} \otimes x \oplus a_i$
end
`res` $= s_0$

A forward error bound for the result of Algorithm 3.11 is (see [32, p.95]):

$$|p(x) - \texttt{res}| \leqslant \gamma_{2n} \sum_{i=0}^{n} |a_i||x|^i = \gamma_{2n}\widetilde{p}(|x|)$$

where $\widetilde{p}(x) = \sum_{i=0}^{n} |a_i|x^i$. It is very interesting to express and interpret this result in terms of the condition number of the polynomial evaluation defined by

$$\text{cond}(p, x) = \frac{\sum_{i=0}^{n} |a_i||x|^i}{|p(x)|} = \frac{\widetilde{p}(|x|)}{|p(x)|}. \tag{3.12}$$

Thus we have

$$\frac{|p(x) - \texttt{res}|}{|p(x)|} \leqslant \gamma_{2n}\,\text{cond}(p, x).$$

If an FMA instruction is available, then the statement $s_i = s_{i+1} \otimes x \oplus a_i$ in Algorithm 3.11 can be re-written $s_i = \texttt{FMA}(s_{i+1}, x, a_i)$ which slightly improves the error bound. Using an FMA this way, the computed result now satisfies

$$\frac{|p(x) - \texttt{res}|}{|p(x)|} \leqslant \gamma_{n}\,\text{cond}(p, x).$$

We can modify the Horner scheme to compute the rounding error at each elementary operation that are a sum and a product. This is done in Algorithm 3.12.

Algorithm 3.12. Polynomial evaluation with a compensated Horner's scheme

function res = CompHorner(p, x)
$s_n = a_n$
$r_n = 0$
for $i = n-1:-1:0$
  $[p_i, \pi_i] =$ TwoProduct(s_{i+1}, x)
  $[s_i, \sigma_i] =$ TwoSum(p_i, a_i)
  $r_i = r_{i+1} \otimes x \oplus (\pi_i \oplus \sigma_i)$
end
res $= s_0 \oplus r_0$

If we denote by p_π and p_σ the two following polynomials

$$p_\pi = \sum_{i=0}^{n-1} \pi_i x^i, \qquad p_\sigma = \sum_{i=0}^{n-1} \sigma_i x^i,$$

then one can show, thanks to error-free transformations that

$$p(x) = s_0 + p_\pi(x) + p_\sigma(x).$$

If one looks at the previous algorithm closely, it is then clear that $s_0 =$ Horner(p, x). As a consequence, we can derive a new error-free transformation for polynomial evaluation

$$p(x) = \text{Horner}(p, x) + p_\pi(x) + p_\sigma(x).$$

The compensated Horner scheme first computes $p_\pi(x) + p_\sigma(x)$ which correspond to the rounding errors and then add the obtained value to the result of the classic Horner scheme Horner(p, x).

We will show that the results computed by Algorithm 3.12 admit significantly better error-bounds than those computed with the classical Horner scheme. We argue that Algorithm 3.12 provides results as if they were computed using twice the working precision. This is summed up in the following theorem.

Theorem 3.2. *Consider a polynomial p of degree n with floating point coefficients, and a floating point value x. The forward error in the compensated Horner algorithm is such that*

$$|\text{CompHorner}(p, x) - p(x)| \leqslant \mathbf{u}|p(x)| + \gamma_{2n}^2 \widetilde{p}(x). \tag{3.13}$$

It is interesting to interpret the previous theorem in terms of the condition number of the polynomial evaluation of p at x. Combining the error bound (3.13) with the condition number (3.12) for polynomial evaluation gives

$$\frac{|\text{CompHorner}(p, x) - p(x)|}{|p(x)|} \leqslant \mathbf{u} + \gamma_{2n}^2 \operatorname{cond}(p, x). \tag{3.14}$$

In other words, the bound for the relative error of the computed result is essentially γ_{2n}^2 times the condition number of the polynomial evaluation, plus the unavoidable term $\mathbf{u}$ for rounding the result to the working precision. In particular, if $\operatorname{cond}(p, x) < \gamma_{2n}^{-1}$, then the

relative accuracy of the result is bounded by a constant of the order of $\mathbf{u}$. This means that the compensated Horner algorithm computes an evaluation accurate to the last few bits as long as the condition number is smaller than $\gamma_{2n}^{-1} \approx (2n\mathbf{u})^{-1}$. Besides that, (3.14) tells us that the computed result is as accurate as if computed by the classic Horner algorithm with twice the working precision, and then rounded to the working precision.

We test the expanded form of the polynomial $p_n(x) = (x-1)^n$. The argument x is chosen near to the unique real root 1 of p_n, and with many significant bits so that a lot of rounding errors occur during the evaluation of $p_n(x)$. We increment the degree n from 1 until a sufficiently large range has been covered by the condition number $\mathrm{cond}(p_n, x)$. Here we have

$$\mathrm{cond}(p_n, x) = \frac{\widehat{p_n}(x)}{|p_n(x)|} = \left|\frac{1+x}{1-x}\right|^n,$$

and $\mathrm{cond}(p_n, x)$ grows exponentially with respect to n. In the experiments reported on Figure 3.3, $\mathrm{cond}(p_n, x)$ varies from 10^2 to 10^{40} (for $x = \mathrm{fl}(1.333)$, that corresponds to the degree range $n = 3, \ldots, 42$). These huge condition numbers have some meaning since here the coefficients of p and the value x are chosen to be exact floating point numbers.

We experiment both `Horner` and `CompHorner`. For each polynomial p_n, the exact value $p_n(x)$ is approximate with a high accuracy. Figure 3.3 presents the relative accuracy $|y - p_n(x)|/|p_n(x)|$ of the evaluation y computed by the two algorithms.

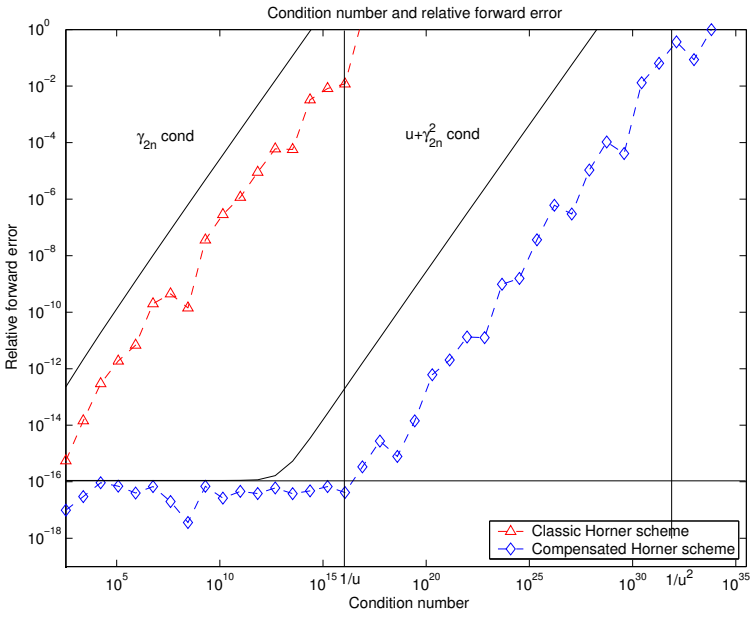

Figure 3.3: Comparison of the classic Horner scheme with the Compensated Horner scheme

We observe that the compensated algorithm exhibits the expected behavior. The full precision solution is computed as long as the condition number is smaller than $\mathbf{u}^{-1} \approx 10^{16}$. Then, for condition numbers between $\mathbf{u}^{-1}$ and $\mathbf{u}^{-2} \approx 10^{32}$, the relative error degrades to

no accuracy at all. However, when the condition number is beyond $\mathbf{u}^{-1}$, the *a priori* error estimate 3.14 is always pessimistic by 2 or 3 orders of magnitude.

We now demonstrate the practical efficiency in terms of running time comparing our algorithms and up-to-date challengers on several significant computing environments.

Since Bailey's double-double are usually considered as the most efficient portable library to double the IEEE-754 double precision, we consider it as a reference in the comparisons with `CompHorner`. We denote by `DDHorner`, our implementation of the Horner algorithm with the double-double format. We also denote by `MPFRHorner` the Horner algorithm with 106-bits precision arithmetic provided by the MPFR library.
We implemented the three algorithms `CompHorner`, `DDHorner`, and `MPFRHorner` in a C code to measure their overhead compared to the `Horner` algorithm. We programed these tests straightforwardly with no other optimization than the ones performed by the compiler. All timings were done with the cache warmed to minimize the memory traffic over-cost.

The measures were performed with polynomials whose degree vary from 5 to 200 by step of 5. For every algorithm, we measured the ratio of its computing time over the computing time of the classic Horner algorithm. It turned out that our compensated algorithm `CompHorner` is about 3 times slower than the classic Horner scheme. The same slowdown factor is about 8 for algorithm `DDHorner`. From a practical point of view, we can state that our algorithm is more than twice faster than the Horner scheme with double-doubles. The `MPFRHorner` routine exhibits a slowdown factor of more than 80. That comparison with the MPFR library is not entirely fair in this context. Indeed It is not surprising since the MPFR library is specially designed to handle floating point numbers with extremely large significand.

3.6 A compensated algorithm for accurate evaluation of the k-th derivative of a polynomial

The Horner Derivative (`HD`) algorithm is the classic method for the evaluation of the k-th derivative of a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$. This algorithm 3.13 makes the direct evaluation possible without obtaining the k-th derivative of the polynomial itself at any point x (see [32, p.97]).

Algorithm 3.13. Horner Derivative algorithm

function `res=HD`(p, x, k)
  $y_i^j = 0$, for $i = 0:1:k$, and $j = n+1:-1:0$
  $y_{-1}^{j+1} = a_j$, for $j = n:-1:0$

  for $j = n:-1:0$
    for $i = \min(k, n-j):-1:\max(0, k-j)$
      $y_i^j = x \times y_i^{j+1} + y_{i-1}^{j+1}$
    end

end
res $= k! \times y_k^0$

The condition number for the k-th derivative evaluation of a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ at entry x is given by

$$\mathtt{cond}(p,x,k) = \frac{k!\sum_{m=k}^{n} C_m^k |x|^{m-k}|a_m|}{|k!\sum_{m=k}^{n} C_m^k x^{m-k} a_m|} = \frac{\widetilde{p}^{(k)}(x)}{|p^{(k)}(x)|}, \quad C_m^k = \binom{m}{k}.$$

The following theorem gives an error bound analysis for the computed result by Algorithm 3.13.

Theorem 3.3. *Let $p(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial of degree n with floating-point coefficients, and x a floating-point value (with $p^{(k)}(x) \neq 0$). The* relative forward error bound *in the HD algorithm satifsfies:*

$$\frac{|\mathtt{HD}(p,x,k) - p^{(k)}(x)|}{|p^{(k)}(x)|} \leqslant \gamma_{2n}\mathtt{cond}(p,x,k).$$

This theorem is stated and proved in [32, p.97] for the first derivative only and using linear algebra. In [37], we introduced a data dependency graph as a convenient technique for error analysis. We have proved Theorem 3.3 for all k.

In Algorithm 3.14, the rounding errors generated by Algorithm 3.13 in each step are computed. These errors are accumulated and finally added to the computed result. We then obtain the compensated Horner Derivative algorithm (CompHD).

Algorithm 3.14. Compensated Horner Derivative algorithm

function res=CompHD(p,x,k)
$y_i^j = 0, \widehat{\epsilon y}_i^j = 0$, for $i = 0:1:k$, and $j = n+1:-1:0$
$y_{-1}^{j+1} = a_j, \widehat{\epsilon y}_{-1}^{j+1} = 0$, for $j = n:-1:0$
for $j = n:-1:0$
for $i = \min(k, n-j):-1:\max(0, k-j)$
$[s, \pi_i^j] = \mathtt{TwoProd}(x, \widehat{y}_i^{j+1})$
$[\widehat{y}_i^j, \sigma_i^j] = \mathtt{TwoSum}(s, \widehat{y}_{i-1}^{j+1})$
$\widehat{\epsilon y}_i^j = x \otimes \widehat{\epsilon y}_i^{j+1} \oplus \widehat{\epsilon y}_{i-1}^{j+1} \oplus (\pi_i^j \oplus \sigma_i^j)$
end
end
res $= (\widehat{y}_k^0 \oplus \widehat{\epsilon y}_k^0) \otimes k!$

Theorem 3.4 shows that the result of the compensated Horner derivative algorithm is nearly as accurate as if computed by the original Horner derivative algorithm with twice the working precision and then roughly rounded to the working precision.

Theorem 3.4. *Let $p(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial of degree n with floating-point coefficients, and x a floating-point value (with $p^{(k)}(x) \neq 0$). The relative forward error bound in CHD algorithm is such that*

$$\frac{|\texttt{CompHD}(p,x,k) - p^{(k)}(x)|}{|p^{(k)}(x)|} \leqslant 2u + (k+1)\underbrace{\gamma_{2n}\gamma_{3n}}_{\approx 6n^2\mathbf{u}^2}\texttt{cond}(p,x,k). \tag{3.15}$$

In the first experiment, we evaluate the 3rd derivative of the polynomial $p(x) = (x - 0.75)^5(x-1)^{11}$, given in its expanded form, in the neighborhood of its multiple roots 0.75 and 1. Figure 3.4 presents the evaluation for 400 equally spaced points in the intervals $[0.74995, 0.75005]$ and $[0.9935, 1.0065]$. It is clear that the CompHD algorithm (Algorithm 3.14) gives much more smooth drawing than the original HD algorithm (Algorithm 3.13). The results show that the compensated algorithm is an effective method of accurate evaluation to recover the expected curve.

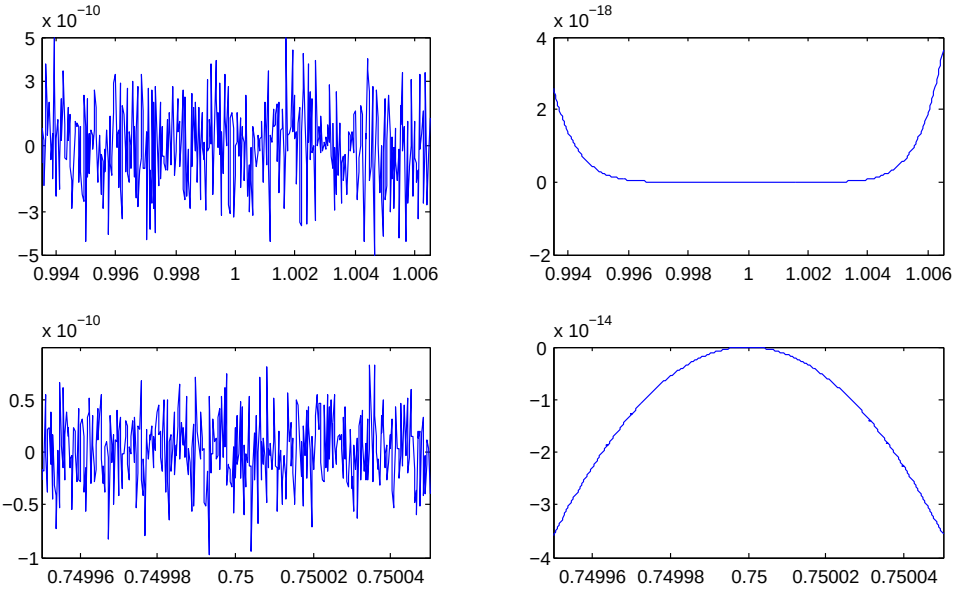

Figure 3.4: Evaluation the 3rd derivative of the polynomial $p(x) = (x-0.75)^5(x-1)^{11}$ in the neighborhood of its multiple roots, using HD (left) and CompHD (right)

In the second experiment, we focus on the forward error bound of our compensated Horner Derivative algorithm. We consider the 3rd derivative evaluation of $p(x) = (x-1)^n$ for $n = 5, \cdots, 45$ in expanded form at the entry $x = \text{fl}(1.333)$. The results of the tests performed with the CompHD and HD algorithm are reported on Figure 3.5. As expected, when the condition number is smaller than $1/\mathbf{u}$, the relative error of the result by the CompHD algorithm (Algorithm 3.14) is equal to or smaller than $2\mathbf{u}$. This relative error increases nearly linearly for the condition number between $1/\mathbf{u}$ and $1/\mathbf{u}^2$.

It is interesting to compare the compensated Horner derivative algorithm with other approaches to obtain high-precision. A standard way is by using multiple precision libraries,

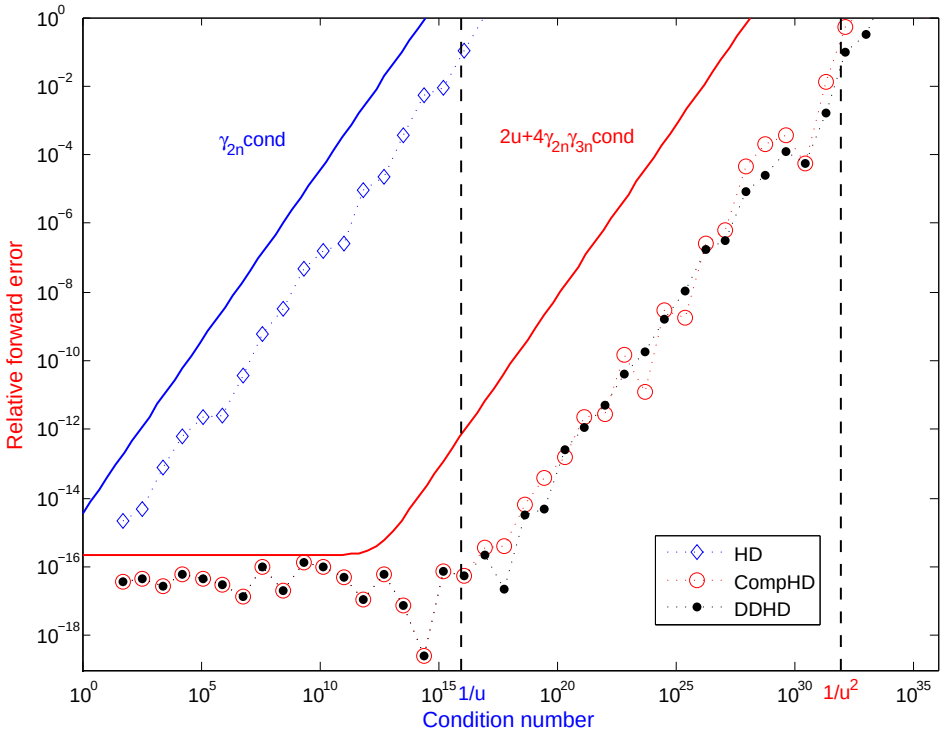

Figure 3.5: Accuracy of the evaluation of the 3rd derivative of $p(x) = (x-1)^n$ for $n = 5, \cdots, 45$ in expanded form at the entry $x = \mathrm{fl}(1.333)$ with respect to the condition number.

but if we just want to double the IEEE-754 double precision, the most efficient way is by using the double-double arithmetic. Thus we compare `CompHD` algorithm with the Horner derivative algorithm written in double-double arithmetic (`DDHD` algorithm). Here the result of `DDHD` should be rounded to the working precision (double precision). As we see in Figure 3.5, `CompHD` algorithm has nearly the same accuracy as `DDHD` algorithm.

In a practical computation, we usually wish to compute a corresponding error bound along with the result. The *a priori* error bound (3.15) in Theorem 3.4 is entirely adequate for theoretical purposes, but lacks sharpness. We have performed a running error analysis of the compensated Horner Derivative algorithm, which provides a sharper and a posteriori error bound in Theorem 3.5. Hence, we can obtain a running error bound by Algorithm 3.15.

Algorithm 3.15. Compensated Horner Derivative algorithm with validated running error bound

function [`res`, μ]=`CompHDWErr`(p, x, k)
$y_i^j = 0,\ \widehat{\epsilon y}_i^j = 0,\ \widehat{w}_i^j = 0,$ for $i = 0:1:k,\ j = n+1:-1:0$
$y_{-1}^{j+1} = a_j,\ \widehat{\epsilon y}_{-1}^{j+1} = 0,\ \widehat{w}_{-1}^{j+1} = 0,$ for $j = n:-1:0$
for $j = n:-1:0$
for $i = \min(k, n-j):-1:\max(0, k-j)$

$[s,\pi_i^j] = \texttt{TwoProd}(x,\widehat{y}_i^{j+1})$
$[\widehat{y}_i^j,\sigma_i^j] = \texttt{TwoSum}(s,\widehat{y}_{i-1}^{j+1})$
$\widehat{\epsilon y}_i^j = x\otimes\widehat{\epsilon y}_i^{j+1}\oplus\widehat{\epsilon y}_{i-1}^{j+1}\oplus(\pi_i^j\oplus\sigma_i^j)$
$\widehat{w}_i^j = |x|\otimes\widehat{w}_i^{j+1}\oplus\widehat{w}_{i-1}^{j+1}\oplus(|\pi_i^j|\oplus|\sigma_i^j|)$
end
end
$[s,c]$=FastTwoSum$(\widehat{y}_k^0,\widehat{\epsilon y}_k^0)$
[res, e]=TwoProd$(s,k!)$
$\widehat{\alpha} = (\widehat{\gamma}_{3n-k-1}\otimes\widehat{w}_k^0)\oslash(1\ominus(3n+1)u)$
$\widehat{\beta} = |c\otimes k!\oplus e|$
$\mu = (\widehat{\alpha}\otimes k!\oplus\widehat{\beta})\oslash(1\ominus 4u)$

Theorem 3.5. *A running error bound of* CompHD *algorithm is given by*

$$|\texttt{CompHD}(p,x,k)-p^{(k)}(x))| \leqslant \mathrm{fl}\Big(\frac{\widehat{\alpha}\otimes k!\oplus\widehat{\beta}}{1-4u}\Big) =: \mu, \tag{3.16}$$

where $\widehat{\alpha} = (\widehat{\gamma}_{3n-k-1}\otimes\widehat{w}_k^0)\oslash(1\ominus(3n+1)u)$ *and* $\widehat{\beta}$ *is obtained from the following equality*

$$\widehat{\beta} = |c\otimes k!\oplus e|$$

with $[s,c]$=TwoSum$(\widehat{y}_k^0,\widehat{\epsilon y}_k^0)$ *and* $[\overline{p}^{(k)}(x),e]$=TwoProd$(s,k!)$.

In the next experiment, we illustrate the advantage of the running error bound (3.16) over the *a priori* one (3.15) in the accuracy of the error bound. We evaluate the 3rd derivative of $p(x) = (x-1)^8$ in expanded form for 400 equally-spaced points in the interval $[0.9935, 1.0065]$. The results are reported on Figure 3.6. The running error bound is more significant than the *a priori* one especially near $x = 1$.

We now present the computational complexity of algorithms HD, CompHD, CompHDwErr and DDHD, and then show the practical efficiency of our algorithm in terms of running time. The computational cost of the algorithms in terms of flops is:

- HD: $2(n-k)(k+1)+k-1+4n$ flops,
- CompHD: $23(n-k)(k+1)+k+4+4n$ flops,
- CompHDwErr: $29(n-k)(k+1)+k+43+4n$ flops,
- DDHD: $38(n-k)(k+1)+k+2+4n$ flops.

We measured the flop count ratios among HD, CompHD, CompHDwErr and DDHD and display the average ratios for $n = 50:5:1000$ and $k = 1:1:8$ in Table 3.1. We can observe that CompHD is as accurate as DDHD but only requires on average only about 39% less flop count. We also see that the over-cost due to the running error analysis for CompHDwErr is quite reasonable.

We also compared HD, CompHD, CompHDwErr and DDHD in terms of measured computing time. The experiments are performed on a laptop with a Intel(R) Core(TM) i5-2520M processor, with two cores each at 2.50Ghz.

The average time ratio are reported in Table 3.2. In contrast with the data in Table 3.1, we see that the measured computing time ratio CompHD/DDHD is better than the theoretical one.

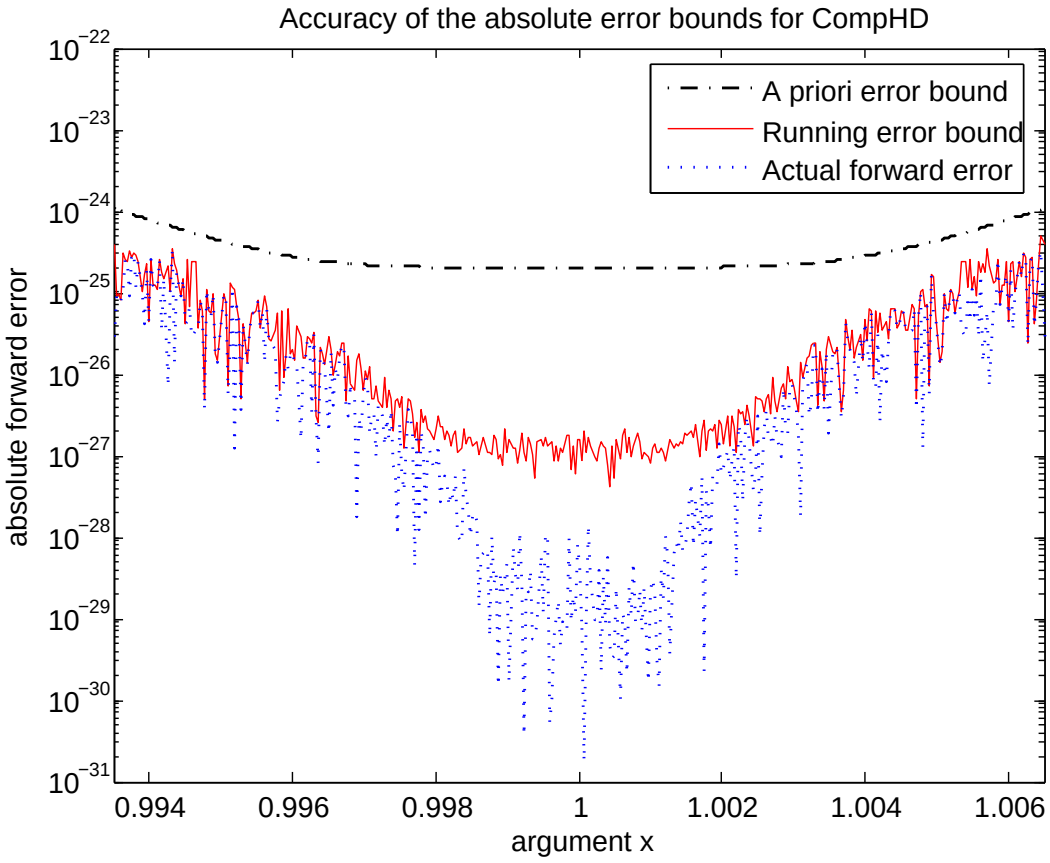

Figure 3.6: Significance of the running error bound

Table 3.1: Average ratios of the floating-point operations

$\frac{\texttt{CompHD}}{\texttt{HD}}$	$\frac{\texttt{CompHDwErr}}{\texttt{HD}}$	$\frac{\texttt{DDHD}}{\texttt{HD}}$	$\frac{\texttt{CompHD}}{\texttt{CompHDwErr}}$	$\frac{\texttt{CompHD}}{\texttt{DDHD}}$
8.35	10.46	13.60	79.87%	61.47%

Thanks to the analysis in terms of instruction level parallelism (ILP), one can see that this phenomenon is surprising, but reasonable. Briefly speaking, avoiding the renormalization step needed for double-double computations, the `CompHD` algorithm presents more ILP than its counterpart `DDHD` algorithm. Also note that `CompHD`, `CompHDwErr` and `DDHD` have much more instructions than `HD`, then they will introduce some more instruction-level parallelism. This partly explains the phenomenon that the first three ratios of running time are smaller than those of the theoretical flop count. Considering that `CompHDwErr` has some more procedures for computing the running error bound than `CompHDwErr`, which limits exploiting the ILP. Then it is reasonable that the measured running time ratio between `CompHD`

and CompHDwErr is smaller than the theoretical flop count one.

Table 3.2: Measured running time ratios

	$\frac{\texttt{CompHD}}{\texttt{HD}}$	$\frac{\texttt{CompHDwErr}}{\texttt{HD}}$	$\frac{\texttt{DDHD}}{\texttt{HD}}$	$\frac{\texttt{CompHD}}{\texttt{CompHDwErr}}$	$\frac{\texttt{CompHD}}{\texttt{DDHD}}$
Linux gcc 4.4.5	3.85	6.44	8.14	61.76%	47.42%

3.7 Accurate Newton's methods for finding simple roots of polynomials

In this section, we present an application to show the effectiveness of the proposed compensated Horner Derivative algorithm. We consider Newton's method in floating-point arithmetic [69] for solving the equation $p(x) = 0$ where p is an univariate polynomial. We want to find simple roots. In that case, we improved the accurate Newton's method we proposed in [25], by using the CompHD algorithm to accurately compute the derivative. The classic Newton method and the accurate Newton method are presented as follows:

Algorithm 3.16. The classic Newton method

$x_0 = \xi$

$x_{i+1} = x_i - \frac{\texttt{Horner}(p,x_i)}{\texttt{HD}(p,x_i,1)}$

Algorithm 3.17 ([25])**.** The accurate Newton method

$x_0 = \xi$

$x_{i+1} = x_i - \frac{\texttt{CompHorner}(p,x_i)}{\texttt{HD}(p,x_i,1)}$

For a polynomial $p(x)$ with a simple zero x, x is not a zero of p', however, sometimes the evaluation of p' near x can be still ill-conditioned. In such a case, it is necessary to accurately evaluate $p'(x_i)$, then we choose the CompHD algorithm to modify Algorithm 3.17 and obtain the following algorithm.

Algorithm 3.18. The new accurate Newton method

$x_0 = \xi$

$x_{i+1} = x_i - \frac{\texttt{CompHorner}(p,x_i)}{\texttt{CompHD}(p,x_i,1)}$

The condition number for finding a simple root of an univariate polynomial is given as follows

$$\texttt{cond}_{\texttt{root}}(p,x) = \frac{\widetilde{p}(|x|)}{|x||p'(x)|}. \tag{3.17}$$

The following error analysis and numerical example proves and illustrates, respectively, that the convergence of iteration strongly depends on the accuracy of the derivative's evaluation when the problem of finding simple root is too ill-conditioned, and that the accuracy of the final iteration result depends on the accuracy with which the residual is computed.

Using a theorem of Tisseur [69], we have shown the following theorem for the accurate Newton's method (Algorithm 3.17).

Theorem 3.6. *Assume that there is an x such that $p(x) = 0$ and $p'(x) \neq 0$ is not too small. Assume also that $\mathbf{u} \cdot \mathrm{cond}(p,x) \leqslant 1/8$ for all i.*
Then, for all x_0 such that $\beta |p'(x)^{-1}||x_0 - x| \leqslant 1/8$, Newton's method in floating point arithmetic with Algorithm 3.17 generates a sequence of $\{x_i\}$ whose relative error decreases until the first i for which

$$\frac{|x_{i+1} - x|}{|x|} \approx \mathbf{u} + \gamma_{2n}^2 \, \mathrm{cond}_{\texttt{root}}(p,x).$$

We have tested the classic Newton's iteration (Algorithm3.16) and the accurate Newton's iteration (Algorithm 3.17) with $p_n(x) = (x-1)^n - 10^{-8}$ and $x = 1 + 10^{-8/n}$ for $n = 1:40$. In that case, the condition number $\mathrm{cond}_{\texttt{root}}(p_n, x)$ varies from 10^4 to 10^{22}. The result are presented in Figure 3.7.

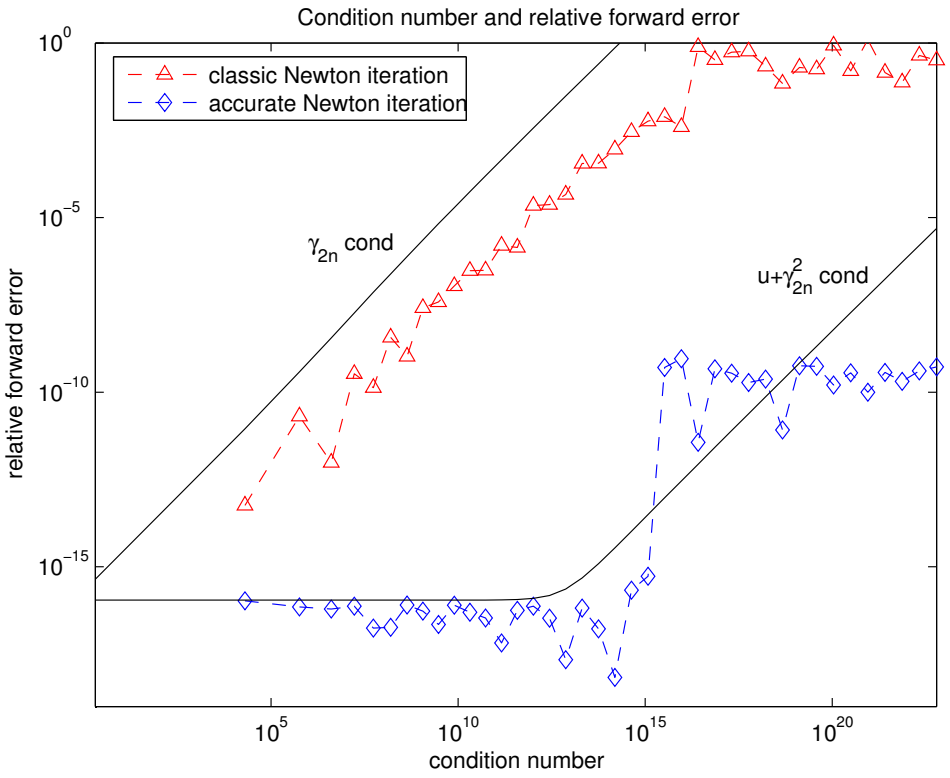

Figure 3.7: Accuracy of the classic Newton's iteration and of the accurate Newton's iteration

As one can see in Figure 3.7, the root is computed with full accuracy until a condition number of $10^{16} = 1/\mathbf{u}$. For a greater condition number, less accuracy is obtained which cannot be estimated. This is due in part to the rounding errors in the evaluation of the derivative. To tackle this problem, we used the `CompHD` in Algorithm 3.18.

The following theorem summarizes the results on the accuracy of Algorithms 3.16, 3.17 and 3.18.

Theorem 3.7. *Assume that the simple root is α such that $f(\alpha) = 0$, $f'(\alpha) \neq 0$ and that f is continuously differentiable in a neighborhood of the root, and in floating point arithmetic the computation of the derivative satisfies*

$$\text{Assumption 1}: \qquad \left|\frac{\widehat{f}'(v) - f'(v)}{f'(v)}\right| < \omega < \frac{1}{2}, \tag{3.18}$$

where ω is a given upper bound when v is closed to α, which shows the relative error bound of $f'(v)$. Assume also that for any v, obtained from the iteration from the initial value v_0 sufficiently close to the root α, satisfies

$$\text{Assumption 2}: \qquad 0 < \frac{f(v)}{f'(v)(v-\alpha)} < \mu_1. \tag{3.19}$$

Here, μ_1 is an upper bound which partly shows the ratio bound between the secant and the tangent. In the iterative process, $f'(v) \neq 0$ and $\widehat{f}'(v) \neq 0$, meanwhile ω and μ_1 satisfy

$$\text{Assumption 3}: \qquad \mu_1 + 2\omega \leqslant 2. \tag{3.20}$$

Newton's method (Algorithm 3.16) or its improved versions (Algorithms 3.17 and 3.18) in floating-point arithmetic generates a sequence $\{\widehat{v}_i\}$ converging to v_*. Then assume that, when the iteration converges, there is

$$\text{Assumption 4}: \qquad 0 < \mu_2 < \frac{f(v_*)}{f'(v_*)(v_*-\alpha)}. \tag{3.21}$$

μ_2 means a lower bound of equation in (3.19) for the final iterated result v_*. The parameters ω, μ_1 and μ_2 used in Assumption 1-4 will help to obtain the accuracies guaranteed by the algorithms as follows.

In case of Algorithm 3.16:

$$\left|\frac{\alpha - v_*}{v_*}\right| < C\gamma_{2n}\mathtt{cond}_{\mathtt{root}}(p, v_*). \tag{3.22}$$

In case of Algorithm 3.17 and 3.18,:

$$\left|\frac{\alpha - v_*}{v_*}\right| < Ku + D\gamma_{2n}^2\mathtt{cond}_{\mathtt{root}}(p, v_*). \tag{3.23}$$

where C, K and D are constants depending on ω and μ_2.

Assumption 1 (3.18) is necessary and reasonable. When the relative error is larger than 1/2, the computed result maintains nearly no more than one bit precision, which means there is nearly no useful information left. Assumption 2 (3.19) and Assumption 3 (3.20) will guarantee the convergence of the iteration. These assumptions are not strong that even when the derivative evaluation is too ill-conditioned they can still hold. We deem that the

convergence depends on the accuracy of the function's derivative but not that of the function itself. When the evaluation of derivative is too ill-conditioned, such that $\mathbf{u}\cdot\texttt{cond}(p,x,1)>1$, Algorithm 3.18 still converges but Algorithm 3.17 does not.

To illustrate the effectiveness and accuracy of Algorithm 3.18, we compare Algorithms 3.16, 3.17 and 3.18 by computing the simple real zero of the expanded form of the polynomial $p_n(x)=(x-1)^n-2^{-31}$, for $n=2:55$. The condition number of the real zero varies roughly from 10^4 to 10^{32}. The result is presented in Figure 3.8. Note that, if n is even, there are two real roots: $1\pm 2^{-31/n}$; if n is odd, there is only one real root $1+2^{-31/n}$. We set the initial value $v_0=2$, then considering the local convergence property of Newton method, we deem that the iteration sequence will converge to the real root $\alpha=1+2^{-31/n}$.

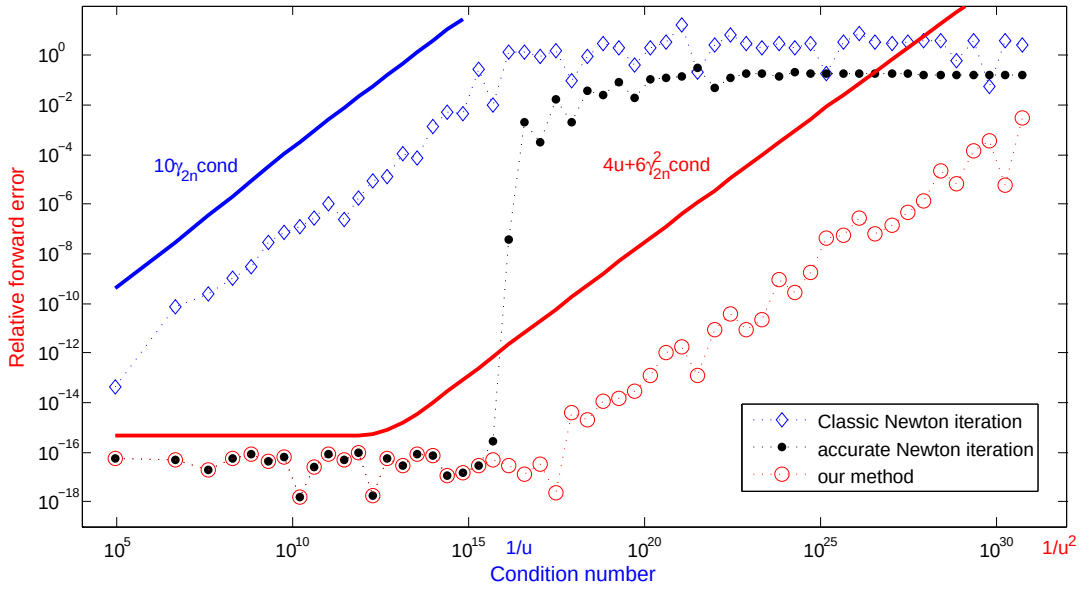

Figure 3.8: Accuracy of the three algorithms with respect to the condition number.

3.8 Accurate and fast evaluation of elementary symmetric functions

The kth Elementary Symmetric Function (ESF) associated with a vector $X=(x_1,\ldots,x_n)$ of n numbers is defined by

$$S_k^{(n)}(X)=\sum_{1\leqslant\pi_1<\ldots<\pi_k\leqslant n} x_{\pi_1}x_{\pi_2}\ldots x_{\pi_k},\quad 1\leqslant k\leqslant n, \tag{3.24}$$

which consists of $\binom{n}{k}$ summands. For $k=0$, $S_0^{(n)}(X)=1$. Throughout this section, we assume that the inputs $X=(x_1,\ldots,x_n)$ are floating-point numbers.

The classic and widely-used method to compute the elementary symmetric functions (3.24) is the so-called Summation Algorithm, which is essentially the algorithm used by MATLAB's `poly` function. The error analysis of this algorithm has been considered in [53], and the result implies that the algorithm is stable. However, as mentioned in [53] "due to cancellation from subtraction", for some too ill-conditioned problems, the computed result by the Summation Algorithm in floating-point arithmetic may be still little accurate. Then a higher accurate algorithm is required.

As an application, the ESFs appear when expanding a linear factorization of a polynomial

$$\prod_{i=1}^{n}(x-x_i)=\sum_{i=0}^{n}c_i x^i=\sum_{i=0}^{n}(-1)^{n-i}S_{n-i}^{(n)}(X)x^i. \tag{3.25}$$

With the Summation Algorithm, one can evaluate polynomial's coefficients $\{c_i\}_{i=0}^n$ from its zeros $\{x_i\}_{i=1}^n$, specially compute characteristic polynomials from eigenvalues (see [11], [21] and [53]). Our algorithm can be used to enhance the accuracy for some ill-conditioned polynomials' coefficients evaluation.

The computation of ESFs is also an important part of the conditional maximum likelihood estimation (CMLE) of item parameters under the Rasch model in psychological measurement [7]. It is promising that our algorithm, improving the numerical accuracy, can allow much more items to be calibrated.

It is very instructive to study the condition number of the kth ESF evaluation (3.24). One defines

$$\mathtt{cond}(S_k^{(n)}(X))=\lim_{\varepsilon\to 0}\sup\Big\{\frac{|S_k^{(n)}(X+\triangle X)-S_k^{(n)}(X)|}{\varepsilon|S_k^{(n)}(X)|}:|\triangle X|<\varepsilon|X|\Big\},$$

where absolute value and comparison are to be understood componentwise. A direct calculation yields that

$$\mathtt{cond}(S_k^{(n)}(X))=\frac{kS_k^{(n)}(|X|)}{|S_k^{(n)}(X)|}. \tag{3.26}$$

In particular, $\mathtt{cond}(S_1^{(n)}(X))=\mathtt{cond}(\sum_{i=1}^n x_i)=\frac{\sum_{i=1}^n|x_i|}{|\sum_{i=1}^n x_i|}$, and $\mathtt{cond}(S_n^{(n)}(X))=\mathtt{cond}(\prod_{i=1}^n x_i)=n$.

The Summation Algorithm, represented by Algorithm 3.19 below, is the same as the one in [53], except that it only computes the kth ESF rather than computing all of ESFs.

Algorithm 3.19 ([53]). Summation Algorithm

```
function S_k^(n)=SumESF(X,k)
  S_0^(i)=1, 1≤i≤n-1;      S_j^(i)=0, j>i;      S_1^(1)=x_1;
  for i=2:n
    for j=max{1,i+k-n}:min{i,k}
      S_j^(i)=S_j^(i-1)+x_i S_{j-1}^(i-1);
    end
  end
```

If we substitute $j = 1:i$ for $j = \texttt{max}\{1, i+k-n\}:\texttt{min}\{i,k\}$, we can compute all ESFs simultaneously. For the simplification of the error analysis, we only consider the computation of the kth ESF. However, in practical calculation such as computing characteristic polynomial from eigenvalue, this substitution is often required.

The following theorem exhibits the roundoff error bounds of Algorithm 3.19.

Theorem 3.8. *If $X = (x_1, \ldots, x_n)$ is a vector of floating-point numbers, the computed k-th elementary symmetric function $\widehat{S}_k^{(n)} = \widehat{S}_k^{(n)}(X)$ by Algorithm 3.19 in floating-point arithmetic satisfies*

$$\begin{aligned}
&|\frac{\widehat{S}_k^{(n)} - S_k^{(n)}}{S_k^{(n)}}| \leqslant \frac{1}{k}\gamma_{2(n-1)}\texttt{cond}(S_k^{(n)}),\ 2 \leqslant k \leqslant n-1,\\
&|\frac{\widehat{S}_1^{(n)} - S_1^{(n)}}{S_1^{(n)}}| \leqslant \gamma_{n-1}\texttt{cond}(S_1^{(n)}) = \gamma_{n-1}\frac{\sum_{i=1}^n |x_i|}{|\sum_{i=1}^n x_i|},\ k = 1,\\
&|\frac{\widehat{S}_n^{(n)} - S_n^{(n)}}{S_n^{(n)}}| \leqslant \frac{1}{n}\gamma_{n-1}\texttt{cond}(S_n^{(n)}) = \gamma_{n-1},\ k = n.
\end{aligned} \tag{3.27}$$

We present hereafter a compensated scheme to evaluate the kth elementary symmetric function.

Algorithm 3.20. Compensated Summation Algorithm

```
function S̄_k^(n) =CompSumESF(X, k)
  Ŝ_0^(i) = 1, 1 ⩽ i ⩽ n−1; Ŝ_j^(i) = 0, j > i; Ŝ_1^(1) = x_1; ε̂S_j^(i) = 0, ∀ i, j
  for i = 2 : n
    for j = max{1, i+k−n} : min{i, k}
      [p, β_j^(i)] = TwoProd(x_i, Ŝ_{j−1}^(i−1));        % S_j^(i) = S_j^(i−1) + x_i S_{j−1}^(i−1)
      [Ŝ_j^(i), σ_j^(i)] = TwoSum(Ŝ_j^(i−1), p);
      ε̂S_j^(i) = ε̂S_j^(i−1) ⊕ (β_j^(i) ⊕ σ_j^(i)) ⊕ x_i ⊗ ε̂S_{j−1}^(i−1)
    end
  end
  S̄_k^(n) = Ŝ_k^(n) ⊕ ε̂S_k^(n)
```

The following theorem exhibits the accuracy of Algorithm 3.20. As for other compensated algorithms, we show that the result is as accurate as if computed with twice the working precision and then rounded to the working precision.

Theorem 3.9. *For a vector of n floating-point numbers $X = (x_1, \ldots, x_n)$, the relative forward*

error bound in Algorithm satisfies

$$\left|\frac{\overline{S}_k^{(n)} - S_k^{(n)}}{S_k^{(n)}}\right| \leqslant \mathbf{u} + \frac{1}{k}\gamma^2_{2(n-1)}\mathtt{cond}(S_k^{(n)}(X)),$$
$$\left|\frac{\widehat{S}_1^{(n)} - S_1^{(n)}}{S_1^{(n)}}\right| \leqslant \mathbf{u} + \gamma^2_{n-1}\mathtt{cond}(S_1^{(n)}),$$
$$\left|\frac{\widehat{S}_n^{(n)} - S_n^{(n)}}{S_n^{(n)}}\right| \leqslant \mathbf{u} + \frac{1}{n}\gamma_n\gamma_{2n}\mathtt{cond}(S_n^{(n)}),$$

with $2 \leqslant k \leqslant n-1$, $k = 1$, $k = n$, *respectively.*

In practical calculations, it is desirable to obtain a corresponding error bound at the same time as the computed value. The *a priori* error bound of Theorem 3.9 is entirely adequate for theoretical purposes, but lakes sharpness. For this requirement, we perform a running error analysis of the `CompSumESF` algorithm, which provides a sharper and *a posteriori* error bound.

Algorithm 3.21. Compensated Summation Algorithm with running error bound

`function` $[\overline{S}_k^{(n)}, \mu]$=`CompSumESFwErr`$(X, k)$
$\widehat{S}_0^{(i)} = 1, 1 \leqslant i \leqslant n-1;$ $\widehat{S}_j^{(i)} = 0,\ j > i;$ $\widehat{S}_1^{(1)} = x_1;$ $\widehat{\epsilon S}_j^{(i)} = 0, \widehat{ES}_j^{(i)} = 0, \forall\ i, j$
`for` $i = 2 : n$
`for` $j = \max\{1, i+k-n\} : \min\{i, k\}$
$[p, \beta_j^{(i)}] = \mathtt{TwoProd}(x_i, \widehat{S}_{j-1}^{(i-1)});$
$[\widehat{S}_j^{(i)}, \sigma_j^{(i)}] = \mathtt{TwoSum}(\widehat{S}_j^{(i-1)}, p);$
$\widehat{\epsilon S}_j^{(i)} = \widehat{\epsilon S}_j^{(i-1)} \oplus (\beta_j^{(i)} \oplus \sigma_j^{(i)}) \oplus x_i \otimes \widehat{\epsilon S}_{j-1}^{(i-1)}$
$\widehat{ES}_j^{(i)} = \widehat{ES}_j^{(i-1)} \oplus |\beta_j^{(i)} \oplus \sigma_j^{(i)}| \oplus |x_i| \otimes \widehat{ES}_{j-1}^{(i-1)}$
`end`
`end`
$[\overline{S}_k^{(n)}, c] = \mathtt{FastTwoSum}(\widehat{S}_k^{(n)}, \widehat{\epsilon S}_k^{(n)})$
$\widehat{\alpha} = (\widehat{\gamma}_{2(n-1)} \otimes \widehat{ES}_k^{(n)}) \oslash (1 - 3nu);$
$\mu = (|c| \oplus \widehat{\alpha}) \oslash (1 - 2u)$

The following theorem proves that this is a certified error bound computed in floating-point arithmetic.

Theorem 3.10. *A running error bound of Algorithm 3.21 is given by*

$$|\overline{S}_k^{(n)} - S_k^{(n)}| \leqslant \mathrm{fl}(\frac{|c| \oplus \widehat{\alpha}}{1-2u}) := \mu. \tag{3.28}$$

As we can see on Figure 3.9, `CompSumESF` (Algorithm 3.20) exhibits the expected behavior. When the condition number is smaller than $1/\mathbf{u}$, the relative error of `CompSumESF` is equal to

or smaller than **u**. This relative error degrades to no precision at all for the condition number between $1/\mathbf{u}$ and $1/\mathbf{u}^2$. Meanwhile, it is shown that `CompSumESF` and `DDSumESF` (Summation Algorithm with double-double numbers) nearly have the same accuracy. We also present the forward error bound of `DDSumESF` shown as the dashed line `TheoBoundDD`. In fact, `DDSumESF` may be a little more accurate than `CompSumESF`, however it is not significant from Figure 3.9. It is also shown that the forward error bound from Theorem 3.9 is valid, but pessimistic compared with the relative running error bound from Algorithm 3.21, which is exhibited as `RunErrBound`. Besides of `DDSumESF`, we also compare our algorithm with `LejaSumESF`, which uses Leja ordering of the zeros in conjunction with the original Summation Algorithm (see [11]). However, it does not give significantly higher accuracy of the results in our numerical tests, partly due to the fact that the inputs are ordered randomly in the generation algorithm.

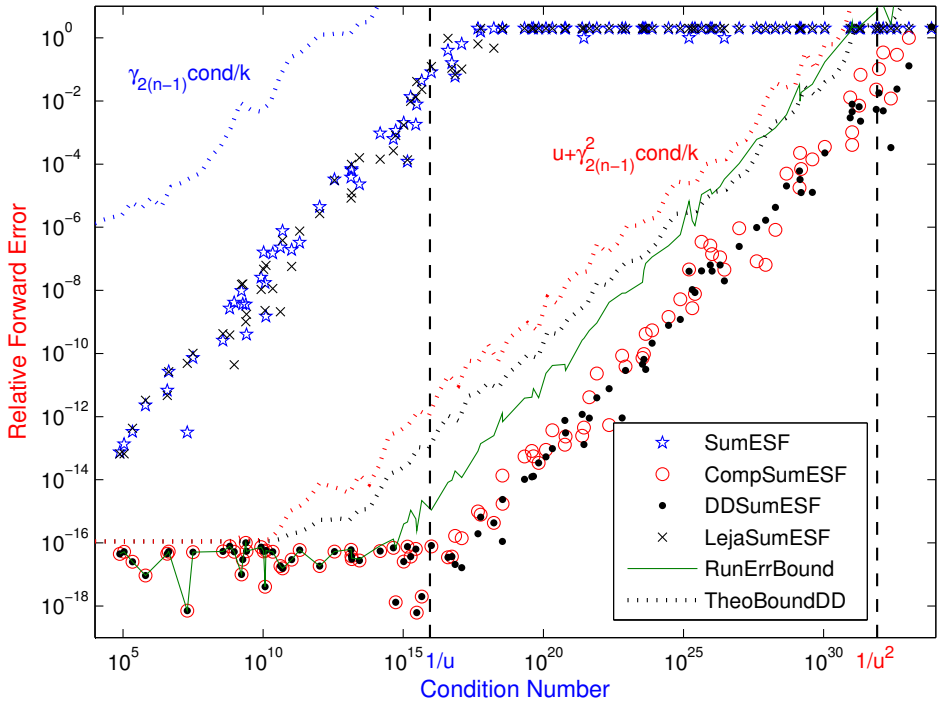

Figure 3.9: Accuracy of evaluation with respect to the condition number

When performing the running time tests, we optimize all algorithms in C code by reversing the computing sequence of j to reduce the required storage location. Similar technique can be seen in [11] and be used in MATLAB's `poly`. We generate the tested random inputs in the interval $[-1,1]$ with n varying from 10 to 30.

We perform tests in two cases. In Case 1, we perfom timing on the algorithms that only compute the kth ESF. Then, in Case 2 we perform timing of the modified algorithms that compute all ESFs simultaneously, which only change the line of code $j = \max\{1, i+k-n\}$:

Table 3.3: Time ratios of computing k-th ESF and all ESFs

	CompSumESF / SumESF	DDSumESF / SumESF	CompSumESF / DDSumESF	CompSumESF / CompSumESFwErr
Case 1	3.05	5.42	57.42%	69.91%
Case 2	3.91	7.48	52.97%	68.02%

$\min\{i,k\}$ in each algorithm to $j = 1:i$. We exhibit the measured running time ratios in two cases in Table 3.3. Case 2 corresponds to the application of computing the coefficients of polynomial from zeros. For simplification, we still denote these algorithms by the same names as before. In both cases, it seems that `CompSumESF` is significantly faster than `DDSumESF` while the results share the same accuracy, and that the over-cost due to the running error bound supported by `CompSumESFwErr` is quite reasonable.

We also consider the flop counts ratios of the algorithms in Case 2 (there are too many comparison operations in Case 1 to be suitable for flop counting). The theoretical ratio between `CompSumESF` and `SumESF` in the optimized C code is approximatly 11.5, which is much smaller than the running time ratios 7.48 shown in Table 3.3. Thanks to the analysis in terms of instruction level parallelism (ILP), this phenomenon is surprising, but reasonable. Moreover, since the renormalization steps in `DDSumESF` may break ILP, the measured running time ratio between `CompSumESF` and `DDSumESF` is usually smaller than the theoretical one ($\approx 61\%$).

As a consequence, it seems that `CompSumESF` is a fast and accurate algorithm to compute elementary symmetric functions and can be well used in computing the coefficients of polynomial from zeros.

3.9 K-fold, faithfully rounded and rounded to nearest results

In the previous sections, we have presented some compensated algorithms that make it possible to compute a result as accurate as if computed with twice the working precision and then rounded to the current working precision.

Let us generalize this for a general problem. Let $\widehat{x}$ be the computed solution of a problem (P) whose exact solution is x. Suppose that the computations have been done with floating-point arithmetic with unit round-off $\mathbf{u}$. We will say the $\widehat{x}$ is as accurate as if computed with twice the working precision if

$$\frac{|\widehat{x}-x|}{|x|} \leqslant \mathbf{u} + C\mathbf{u}^2\,\mathrm{cond}(P). \tag{3.29}$$

where C is a moderate constant, $|\cdot|$ is a norm on the space of the solution and $\mathrm{cond}(P)$ is the condition number of the problem (P). In the right-hand side of inequality (3.29), the second term reflects the computation in twice the working precision and the first one the rounding into the working precision. Relation (3.29) is what we called the compensated rule of thumb,

the classic rule of thumb being [32, p.9]

$$\frac{|\widehat{x}-x|}{|x|} \leqslant C\mathbf{u}\,\mathrm{cond}(P).$$

We will say that the computed result $\widehat{x}$ is of the same quality as if computed in K-fold precision and rounded to working precision if

$$\frac{|\widehat{x}-x|}{|x|} \leqslant \mathbf{u} + (C\mathbf{u})^K\,\mathrm{cond}(P).$$

One can find some K-fold precision algorithm for summation and dot product in [51].

Sometimes, it is needed to get even more accuracy. The floating-point predecessor and successor of a real number r satisfying $\min\{f : f \in \mathbf{R}\} < r < \max\{f : f \in \mathbf{F}\}$ are defined as

$$\mathrm{pred}(r) := \max\{f \in \mathbf{F} : f < r\} \quad \text{and} \quad \mathrm{succ}(r) := \min\{f \in \mathbf{F} : r < f\}.$$

Definition 3.1. A floating-point number $f \in \mathbf{F}$ is called a *faithful rounding* of a real number $r \in \mathbf{R}$ if

$$\mathrm{pred}(f) < r < \mathrm{succ}(f).$$

We denote this by $f \in \square(r)$. For $r \in \mathbf{F}$, this implies that $f = r$.

Faithful rounding means that the computed result is equal to the exact result if the latter is a floating-point number and otherwise is one of the two adjacent floating-point numbers of the exact result (see Figure 3.10).

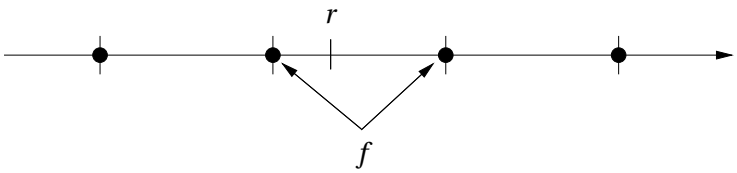

Figure 3.10: Faithful rounding

We can be more accurate requiring the rounded to nearest number. A floating-point number $f \in \mathbf{F}$ is called a *rounded to nearest* of a real number $r \in \mathbf{R}$ if

$$|r-f| = \min\{|r-f'| : f' \in \mathbf{F}\}$$

The tie can be rounded in anyway, for example to even.

Recently, some new algorithms for summation and dot product has been proposed [65, 66]. They make it possible to compute a faithful rounding or a rounding to nearest.

3.10 Accurate floating-point product and exponentiation

One of the examples frequently used in Sterbenz's book [68] is the computation of the product of some floating-point numbers. Such algorithms can be used, for instance, to compute the determinant of a triangle matrix

$$T = \begin{bmatrix} t_{11} & t_{12} & \cdots & t_{1n} \\ & t_{22} & & t_{2n} \\ & & \ddots & \vdots \\ & & & t_{nn} \end{bmatrix}.$$

Indeed, the determinant of T is

$$\det(T) = \prod_{i=1}^{n} t_{ii}.$$

Another application is for evaluating a polynomial when represented by the root product form $p(x) = a_n \prod_{i=1}^{n}(x - x_i)$. It can also apply to compute the integer power of a floating-point number.

In this section, we present an accurate algorithm to compute the product of floating-point numbers [26].

The classic method for evaluating a product of n numbers $a = (a_1, a_2, \ldots, a_n)$

$$p = \prod_{i=1}^{n} a_i$$

is the following algorithm.

Algorithm 3.22. Product evaluation

function `res` = `Prod`(a)
  $p_1 = a_1$
  for $i = 2 : n$
    $p_i = p_{i-1} \otimes a_i$
  end
  `res` = p_n

This algorithm requires $n-1$ flops. It is easy to show that a forward error bound is given by

$$|a_1 a_2 \cdots a_n - \texttt{res}| = |a_1 a_2 \cdots a_n - \mathrm{fl}(a_1 a_2 \cdots a_n)| \leqslant \gamma_{n-1} |a_1 a_2 \cdots a_n|. \tag{3.30}$$

We present hereafter a compensated scheme to evaluate the product of floating-point numbers, i.e. the error of individual multiplication is somehow corrected.

Algorithm 3.23. Product evaluation with a compensated scheme

function res = CompProd(a)
  $p_1 = a_1$
  $e_1 = 0$
  for $i = 2 : n$
    $[p_i, \pi_i] =$ TwoProduct(p_{i-1}, a_i)
    $e_i = e_{i-1} \otimes a_i \oplus \pi_i$
  end
  res $= p_n \oplus e_n$

This algorithm requires $19n - 18$ flops if we use TwoProduct. It only requires $3n - 2$ flops if we use TwoProductFMA instead of TwoProduct (if, of course, an FMA is available) and $e_i =$ FMA(e_{i-1}, a_i, π_i) instead of $e_i = e_{i-1} \otimes a_i \oplus \pi_i$.

The acccuracy of the computed result is given by the following theorem.

Theorem 3.11. *Suppose Algorithm 3.23 is applied to floating-point number $a_i \in \mathbf{F}$, $1 \leqslant i \leqslant n$, and set $p = \prod_{i=1}^{n} a_i$. Then,*

$$|\texttt{res} - p| \leqslant \mathbf{u}|p| + \gamma_n\gamma_{2n}|p|. \tag{3.31}$$

In fact, if $p \neq 0$, we can rewrite Equation (3.31) in the following form

$$\frac{|\texttt{res} - p|}{|p|} \leqslant \mathbf{u} + \gamma_n\gamma_{2n}.$$

Since $\gamma_n\gamma_{2n} \approx 2n^2\mathbf{u}^2$, for n not too large, it follows that $\gamma_n\gamma_{2n}$ is negligible compared to $\mathbf{u}$. As a consequence, the relative error $|\texttt{res} - p|/|p|$ is of the order of $\mathbf{u}$, that is to say, the result has nearly full accuracy. To precise this, we use the following lemme.

Lemma 3.1 (Rump, Ogita and Oishi [65, lem. 2.4])**.** *Let $r, \delta \in \mathbf{R}$ and $\tilde{r} := \mathrm{fl}(r)$. Suppose that $2|\delta| < \mathbf{u}|\tilde{r}|$. Then $\tilde{r} \in \square(r + \delta)$, that means $\tilde{r}$ is a faithful rounding of $r + \delta$.*

It is then possible to show that under mild assumptions, the computed result is a faithful rounding of the exact result.

Lemma 3.2. *If $n < \frac{\sqrt{1-\mathbf{u}}}{\sqrt{2}\sqrt{2+\mathbf{u}}+2\sqrt{(1-\mathbf{u})\mathbf{u}}}\mathbf{u}^{-1/2}$ then* res *is a faithful rounding of p.*

We have just shown that if $n < \alpha\mathbf{u}^{-1/2}$ where $\alpha \approx 1/2$ then the result is faithfully rounded. More precisely, in double precision where $\mathbf{u} = 2^{-53}$, if $n < 2^{25} \approx 5 \cdot 10^7$, we get a faithfully rounded result.

We can propose a weaker form of Lemma 3.2 but with a nicer constant if we suppose, for instance, that $\mathbf{u} \leqslant 2^{-7}$. This is not a strong assumption since in general $\mathbf{u} = 2^{-53}$ or $\mathbf{u} = 2^{-24}$. We can easily show that if $\mathbf{u} \leqslant 2^{-7}$ and $n < (4/9)\mathbf{u}^{-1/2}$ then res is a faithful rounding of p.

3.11 Conclusion

In this chapter, we have shown that error-free transformations could be use to efficiently increase the accuracy of numerical algorithms. The computed results obtained with our

compensated algorithms are as accurate as if computed with twice the working precision and then rounded to that working precision. Our algorithms are at least twice as fast as the classic algorithms implemented with the double-double library while sharing the same accuracy. Nevertheless, our algorithms need, for the moment, to be rewrite significantly whereas the use of the double-double library is easier.

CHAPTER

FOUR

VERIFYING ASSUMPTIONS OF THEOREMS ON THE COMPUTER

This chapter is mainly based on the paper [64]. A more complete review on self-validating methods can be found in [61] and [62].

4.1 Introduction

In this chapter, our aim is to present some work on the possibility of verifying assumptions of mathematical theorems on the computer. Making mathematical proofs with computers needs to get verified results, that could be:

- an interval enclosure of the true result or
- an approximate result with a rigorous error bound.

If possible, we would also like to have a proof of uniqueness of a solution. This must be performed quickly and accurately. We would also like to tackle some "ill-posed problems". This could be done

- with computer algebra systems: exact results but sometimes not efficient or
- with floating-numbers: fast but often wrong results due to rounding errors.

A possible solution could be to compute with floating-point arithmetic but taking into account all rounding errors.

Let us take an example. We want to prove that a floating-point matrix $A \in \mathbf{F}^{n\times n}$ is nonsingular. The following theorem is well-known.

Theorem 4.1. *Let A be a matrix and R another matrix such that $\|I - RA\| < 1$. Then A is nonsingular.*

Proof. By contraposition, if A is singular, there exists $x \neq 0$ such that $Ax = 0$. Then $(I - RA)x = x$ and so $\|I - RA\| \geqslant 1$. □

On a computer, one can compute a floating-point approximation $R \approx A^{-1}$ of the inverse of the matrix A. This can be done with a Gaussian elimination procedure. Then using interval

arithmetic, it is possible to compute a verified (mathematically true) error bound for $\|I - RA\|$. So proving that a matrix is nonsingular with INTLAB can be done via the following algorithm.

Algorithm 4.1. Let A be a matrix of dimension n

```
R = inv(A)
C = eye(n) - R*intval(A)
nonsingular = ( norm(C,1) < 1 )
```

If, at the end of the algorithm, `nonsingular` = 1, then A is nonsingular. This is a mathematical result. But if `nonsingular` = 0, then we can say nothing; A may or may not be nonsingular.

Another approach can be used with some fixed point theorems. Let $f : \mathbf{R}^n \to \mathbf{R}^n$ and $\widehat{x} \in \mathbf{R}^n$ unknown such that $f(\widehat{x}) = 0$. Suppose we know $\widetilde{x} \approx \widehat{x}$ such that $f(\widetilde{x}) \approx 0$. Our purpose is to find a bound for $\widetilde{x}$, that is to say, an interval X containing $\widetilde{x}$ such that $\widehat{x} \in X$. It is clear that $f(x) = 0 \Leftrightarrow g(x) = x$ with $g(x) := x - Rf(x)$ and $\det(R) \neq 0$. We will now use Brouwer's fixed point theorem.

Theorem 4.2 (Brouwer, 1912)**.** *Every continuous function from a closed ball of a Euclidean space to itself has a fixed point.*

Let $X \in \mathbf{IR}^n$ such that $g(X) \subseteq X$. By Brouwer's theorem, there exists $\widehat{x} \in X$, such that $g(\widehat{x}) = \widehat{x}$ and so $f(\widehat{x}) = 0$. As a consequence, if we know an interval X such $g(X) \subseteq X$ and $\det(R) \neq 0$ then we have a verified error bound for $\widehat{x}$. Checking if R is nonsingular can be done by the previous algorithm 4.1.

But naive approach fails since in general $g(X) \subseteq X - Rf(X) \not\subseteq X$. To overcome this problem, it is useful to use the Mean Value Theorem : if $f \in \mathscr{C}^1$ then $f(x) = f(\widetilde{x}) + M(x - \widetilde{x})$ with $M = (\frac{\partial f}{\partial x}(\xi_i))_i$.

Let us denote $Y := X - \widetilde{x}$. Then we have

$$x \in X \quad \Rightarrow \quad g(x) - \widetilde{x} = x - \widetilde{x} - Rf(x) = -Rf(\widetilde{x}) + (I - RM)(x - \widetilde{x}) \in -Rf(\widetilde{x}) + (I - RM)Y.$$

As a consequence, if $-Rf(\widetilde{x}) + (I - RM)Y \subseteq Y$ then $g(X) - \widetilde{x} \subseteq Y$ and so $g(X) \subseteq X$. This is the strategy we will follow in the rest of the chapter.

In Section 4.2 we present the problem of multiple roots of systems of nonlinear equations. In Section 4.3 we briefly summarize how to compute verified error bounds for a (simple) solution of a system of nonlinear equations. In Section 4.4 we develop methods to compute verified error bounds for a double root of a univariate nonlinear function, and in Section 4.5 we treat double roots of systems of nonlinear equations. We close the chapter with numerical results.

4.2 Multiple roots of systems of nonlinear equations

It is well-known that to decide whether a univariate polynomial has a multiple root is an ill-posed problem: an arbitrary small perturbation of a polynomial coefficient may change

the answer from yes to no. In particular a real double root may change into two simple (real or complex) roots.

Therefore it is hardly possible to verify that a polynomial or a nonlinear function has a double root unless the entire computation is performed without any rounding error, i.e. using methods from Computer Algebra.

A typical so-called *verification method* is based on a theorem the assumptions of which are verified on computers. Typically such theorems are in turn based on some kind of fixed point theorem (see the introduction above). The verification of the assumptions is performed using floating-point arithmetic with rigorously estimating all intermediate rounding errors. The computed results have a mathematical certainty. Some of those methods are collected in INTLAB [58], the Matlab Toolbox for Reliable Computing.

The computing time of such a verification method is often of the order of a comparable pure approximative (floating-point) algorithm, whereas the latter does not provide the kind of guaranty of the correctness of the result. A main reason is that verification methods use floating-point arithmetic as well, combined with suitable error estimations.

In case of an exactly given (real or complex floating-point) matrix, the verification of nonsingularity is, of course, possible as well. As a drawback however, in contrast to Computer Algebra methods, the verification of *singularity* is by principle outside the scope of verification methods because this is an ill-posed problem: An arbitrarily small perturbation of a singular matrix may produce a regular matrix changing the answer discontinuously from "yes" to "no".

In the rest of this chapter we describe a verification method for computing guaranteed (real or complex) error bounds for double roots of systems of nonlinear equations. To circumvent the principle problem of ill-posedness we prove that a slightly perturbed system of nonlinear equations has a double root. For example, for a given univariate function $f : \mathbf{R} \to \mathbf{R}$ we compute two intervals $X, E \subseteq \mathbf{R}$ with the property that there exists $\widehat{x} \in X$ and $\widehat{e} \in E$ such that $\widehat{x}$ is a double root of $\overline{f}(x) := f(x) - \widehat{e}$. If the function f has a double root, typically the interval E is a very narrow interval around zero. For complex discs and system of equations assertions are similar.

4.3 Verified solution of nonlinear systems

In the following we assume that we use IEEE 754 double precision floating-point arithmetic for which relative rounding error unit $\mathbf{u} = 2^{-53} \approx 1.11 \cdot 10^{-16}$. The i-th row or column of a matrix $A \in \mathbf{R}^{n\times n}$ are denoted by $A_{i,:}$ or $A_{:,i}$, respectively, similar to Matlab notation.

We recall that $\mathbf{IR}$ denotes the set of real intervals, and by $\mathbf{IR}^n$ and $\mathbf{IR}^{n\times n}$ the set of real interval vectors and interval matrices, respectively.

Standard verification methods for systems of nonlinear equations are based on the following theorem [67].

Theorem 4.3. *Let $f : \mathbf{R}^n \to \mathbf{R}^n$ with $f = (f_1, \ldots, f_n) \in \mathscr{C}^1$, $\widetilde{x} \in \mathbf{R}^n$, $X \in \mathbf{IR}^n$ with $0 \in X$ and $R \in \mathbf{R}^{n\times n}$ be given. Let $M \in \mathbf{IR}^{n\times n}$ be given such that*

$$\{\nabla f_i(\zeta) : \zeta \in \widetilde{x} + X\} \subseteq M_{i,:} \,. \tag{4.1}$$

Denote by I the $n \times n$ identity matrix and assume

$$-Rf(\widetilde{x}) + (I - RM)X \subseteq int(X). \tag{4.2}$$

Then there is a unique $\widehat{x} \in \widetilde{x} + X$ with $f(\widehat{x}) = 0$. Moreover, every matrix $\widetilde{M} \in M$ is nonsingular. In particular, the Jacobian $J_f(\widehat{x}) = \frac{\partial f}{\partial x}(\widehat{x})$ is nonsingular.

An implementation of a verification method for nonlinear systems based on Theorem 4.3 is algorithm `verifynlss` in INTLAB [58].

Part of the assertions of Theorem 4.3 is the nonsingularity of the Jacobian $J_f(\widehat{x})$. Naturally this restricts the application to simple roots because it is proved that the root is simple. Next we will derive verification methods to prove existence of a truly multiple root of a slightly perturbed function.

4.4 The univariate case

The typical scenario in the univariate case is a function $f : \mathbf{R} \to \mathbf{R}$ with a double root $\widehat{x}$, i.e. $f(\widehat{x}) = f'(\widehat{x}) = 0$ and $f''(\widehat{x}) \neq 0$. Consider, for example,

$$\begin{array}{rcl} f(x) & = & 18x^7 - 183x^6 + 764x^5 - 1675x^4 + 2040x^3 - 1336x^2 + 416x - 48 \\ & = & (3x-1)^2(2x-3)(x-2)^4 \end{array} \tag{4.3}$$

In [60] verification methods for multiple roots of polynomials are presented. Here, for example, a set containing k roots of a polynomial is computed, but no information on the true multiplicity can be given. A hybrid algorithm based on the methods in [60] is implemented in algorithm `verifypoly` in INTLAB.

To compute inclusions of the roots of f we need rough approximations. Computing inclusions `X1`, `X2` and `X3` of the simple root $x_1 = 1.5$, the double root $x_2 = 1/3$ and the quadruple root $x_3 = 2$ of f in (4.3) by algorithm `verifypoly` in INTLAB we obtain the following (the polynomial is, of course, specified in expanded form, not the factored form). Note that only rough approximations of the roots are necessary.

```
>> X1 = verifypoly(f,1.3), X2 = verifypoly(f,.3), ...
X3 = verifypoly(f,2.1)
intval X1 =
[   1.49999999999904,   1.50000000000078]
intval X2 =
[   0.33333316656015,   0.33333343640539]
intval X3 =
[    1.99741678159164,    2.00363593397305]
```

The accuracy of the inclusion of the double root $x_2 = 1/3$ is much less than that of the simple root $x_1 = 1.5$, and this is typical. If we perturb f into $\widetilde{f}(x) := f(x) - \varepsilon$ for some small real constant ε and look at a perturbed root $\widetilde{f}(\widehat{x} + h)$ of $\widetilde{f}$, then

$$0 = \widetilde{f}(\widehat{x} + h) = -\varepsilon + \frac{1}{2} f''(\widehat{x}) h^2 + \mathcal{O}(h^3) \tag{4.4}$$

implies

$$h \sim \sqrt{2\varepsilon / f''(\hat{x})}. \tag{4.5}$$

In general floating-point computations are afflicted with a relative error of size $\varepsilon \approx 10^{-16}$. This has the same effect as a perturbation of the given function f into $\widetilde{f}$. Therefore we may compute an inclusion of two roots of a nonlinear function, but by (4.4) and (4.5) we cannot expect this inclusion to be of better relative accuracy than $\sqrt{\varepsilon} \approx 10^{-8}$. This corresponds to the inclusion `X2` above and to the results in [3, 59, 60].

Similarly it is known that the sensitivity of a k-fold root is of the order $\varepsilon^{1/k}$, so that for the quadruple root $x_3 = 2$ of f we cannot expect a better relative accuracy than $\sqrt[4]{\varepsilon} \sim 10^{-4}$. This corresponds to the accuracy of `X3`.

Instead we consider for a double root the nonlinear system $G : \mathbf{R}^2 \to \mathbf{R}$ with

$$G(x,e) = \begin{pmatrix} f(x) - e \\ f'(x) \end{pmatrix} = 0 \tag{4.6}$$

in the two unknowns x and e. The Jacobian of this system is

$$J_G(x,e) = \begin{pmatrix} f'(x) & -1 \\ f''(x) & 0 \end{pmatrix}, \tag{4.7}$$

so that the nonlinear system (4.6) is well-conditioned for the double root $x_2 = 1/3$ of f in (4.3). Now we can apply a verification algorithm for solving general systems of nonlinear equations based on Theorem 4.3 such as algorithm `verifynlss` in INTLAB. Note that the system of nonlinear functions is provided by a Matlab subroutine for computing the function values. No more information is necessary; in particular derivatives are computed by means of automatic differentiation. Indeed, applying algorithm `verifynlss` to (4.6) we obtain

```
>> Y2 = verifynlss(G,[.3;0])
intval Y2 =
[  3.333333333333328e-001,   3.333333333333337e-001]
[ -2.131628207280424e-014,   2.131628207280420e-014]
```

This proves that there is a constant ε with $|\varepsilon| \le 2.14 \cdot 10^{-14}$ such that the nonlinear equation $f(x) - \varepsilon = 0$ has a double root $\hat{x}$ with $0.3333333333333328 \le \hat{x} \le 0.3333333333333337$. In contrast to the previous inclusion `X2` the new inclusion `Y2` is very accurate. The reason is that only double roots are taken into account, and this removes the high sensitivity of the root. It is a kind of regularization.

4.5 The multivariate case

Let a suitably smooth function $f : \mathbf{R}^n \to \mathbf{R}^n$ and $\hat{x} \in \mathbf{R}^n$ be given such that $f(\hat{x}) = 0$ and the Jacobian of f at $\hat{x}$ is singular. A standard verification method such as `verifynlss` must fail because with an inclusion of a root the nonsingularity of the Jacobian at the root is proved as well. Again it is an ill-posed problem and we need some regularization technique.

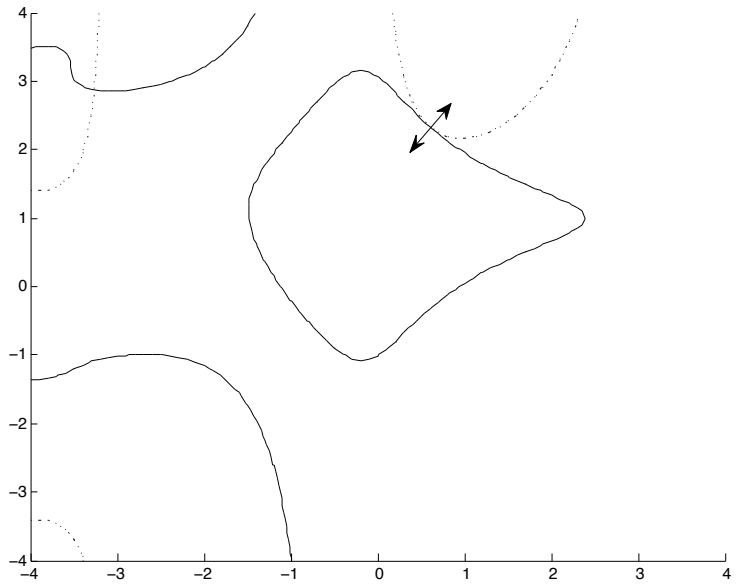

Figure 4.1: Contour lines of $f_1(x) = 0$ (solid) and $f_2(x) = 0$ (dashed)

Consider the model problem

$$f(x,y) = \begin{pmatrix} f_1(x,y) \\ f_2(x,y) \end{pmatrix} = \begin{pmatrix} x^2 + (x+1)(y-1)^2 - \operatorname{asinh}((x+3)^3 + y^2)\cos(x - xy) \\ (x + 1.908718874061618)^2 - \sin(x)(y+1)^2 \end{pmatrix} = 0\,. \tag{4.8}$$

In Figure 4.1 the zero contour lines of f are displayed. Near $(x,y) = (0.60, 2.34)$ the tangents of the contour lines are nearly parallel so that the Jacobian of f at the nearby root is nearly singular. As a regularization we add, similar to the univariate case, a smoothing parameter e and rewrite (4.8) into

$$F(x,y,e) = \begin{pmatrix} f_1(x,y) - e \\ f_2(x,y) \\ \det J_f(x,y) \end{pmatrix} = 0\,. \tag{4.9}$$

The third equation forces the tangents of the zero contour lines to be parallel at the solution, whereas the first equation introduces a perturbation to f_1 so that the root becomes a double root. Locally the zero contour lines behave linearly, so that the smoothing parameter expands or shrinks the zero line for f_1 as depicted by the double arrow in Figure 4.1. Each point of the contour line moves locally normal to the contour line itself. Obviously this is optimal for the regularization.

This approach may work for two or three unknowns, however, an explicit formula for the determinant of the Jacobian is prohibitive for larger dimensions. Consider the following way to ensure the Jacobian to be singular.

Let a function $f = (f_1, \ldots, f_n) : \mathbf{R}^n \to \mathbf{R}^n$ be given and let $\hat{x} = (\hat{x}_1, \ldots, \hat{x}_n)$ be such that $f(\hat{x}) = 0$ and the Jacobian $J_f(\hat{x})$ of f at $\hat{x}$ is singular. Adding a smoothing parameter e we

arrive with $g : \mathbf{R}^{n+1} \to \mathbf{R}^n$ and

$$g(x,e) = \begin{pmatrix} f_1(x) - e \\ f_2(x) \\ \cdots \\ f_n(x) \end{pmatrix} = 0 \tag{4.10}$$

at n equations in $n+1$ unknowns. We force the Jacobian to be singular by

$$J_f(x)\, y = 0 \tag{4.11}$$

for some vector y in the kernel of J_f. In order to make y unique we normalize some component of y to 1. For the moment we choose the first component so that $y = (1, y_2, \ldots, y_n)$. In practice we have to choose a suitable component for normalization. Now (4.11) adds another n equations in $n-1$ unknowns, so that we arrive at a system of $2n$ equations (4.10) and (4.11) in $2n$ unknowns $(x_1, \ldots, x_n, e, y_2, \ldots, y_n)$. Note that the new equations (4.11) only ensure the Jacobian to be singular and have no influence on the described regularization technique.

Theorem 4.4. *Let $f = (f_1, \ldots, f_n) : \mathbf{R}^n \to \mathbf{R}^n$ with $f \in \mathscr{C}^2$ be given. Define $F : \mathbf{R}^{2n} \to \mathbf{R}^{2n}$ by (4.10) and*

$$F(x,e,y) = \begin{pmatrix} g(x,e) \\ J_f(x)\, y \end{pmatrix} = 0\,, \tag{4.12}$$

where $x = (x_1, \ldots, x_n)$, $e \in \mathbf{R}$ and $y = (1, y_2, \ldots, y_n)$. Suppose Theorem 4.3 is applicable to F and yields inclusions for $\widehat{x} \in \mathbf{R}^n, \widehat{e} \in \mathbf{R}$ and $\widehat{y} \in \mathbf{R}^{n-1}$ such that $F(\widehat{x}, \widehat{e}, \widehat{y}) = 0$. Then $g(\widehat{x}, \widehat{e}) = f(\widehat{x}) - (\widehat{e}, 0, \ldots, 0)^T = 0$, and the rank of the Jacobian $J_f(\widehat{x})$ of f at $\widehat{x}$ is $n-1$.

The Jacobian of F computes to

$$J_F(x,e,y) = \begin{pmatrix} J_f(x) & I_{:,1} & \mathscr{O}_{n,n-1} \\ H & \mathscr{O}_{n,1} & J_f(x)_{:,2..n} \end{pmatrix}, \tag{4.13}$$

where I denotes the $n \times n$ identity matrix and $\mathscr{O}_{k,l}$ the $k \times l$ zero matrix. The i-th row of H computes to

$$H_{i,:} = (1, y_2, \ldots, y_n) \cdot \mathrm{Hessian}(f_i(x))\,.$$

Two problems remain. The first is how to choose a suitable component for normalizing the vector in the kernel of J_f. For a given matrix $A \in \mathbf{R}^{n \times n}$, Gaussian elimination with partial pivoting yields LU-factors and a permutation matrix. Applying this to A^T yields $PA^T = LU$. Total pivoting guarantees that the rank of A is $n-1$ or less if and only if $U_{nn} = 0$ (cf. [32]), and except extraordinary circumstances this is also true for partial pivoting. Then $Ax = 0$ for $L^T P x = I_{:,n}$. Applying this to the Jacobian J_f and taking a component of x with largest absolute value is a suitable choice for the component to be normalized to 1.

The second problem is that an inclusion cannot be computed if the rank of the Jacobian J_f is less than $n-1$. More precisely, we proved that if an inclusion of a multiple root is computed, then the rank of the Jacobian is $n-1$, and it would be nice to have the converse, namely that

for a root $f(\widehat{x}) = 0$ and Jacobian $J_f(\widehat{x})$ of rank $n-1$ an inclusion can be computed by applying Theorem 4.3 to (4.10) and (4.11). This is not true as by

$$f(x_1, x_2) = \begin{pmatrix} x_1 - x_2^2 \\ x_1^2 - x_2^2 \end{pmatrix} = 0\,. \tag{4.14}$$

Obviously the Jacobian has rank 1 at $x_1 = x_2 = 0$, but the Jacobian (4.13) of the augmented system (4.12) computes to

$$J_F(x, e, y) = \begin{pmatrix} 1 & -2x_2 & -1 & 0 \\ 2x_1 & -2x_2 & 0 & 0 \\ 0 & -2 & 0 & 1 \\ 2y & -2 & 0 & 2x_1 \end{pmatrix}, \tag{4.15}$$

which is singular for $x_1 = x_2 = 0$. This means that it is not possible to compute an inclusion of the multiple root $(0,0)$. However, in this case the reason is that the wrong equation was regularized. Exchanging the two equations in (4.14) into

$$f(x_1, x_2) = \begin{pmatrix} x_1^2 - x_2^2 \\ x_1 - x_2^2 \end{pmatrix} = 0 \tag{4.16}$$

yields

$$J_F(x, e, y) = \begin{pmatrix} 2x_1 & -2x_2 & -1 & 0 \\ 1 & -2x_2 & 0 & 0 \\ 0 & -2 & 0 & 1 \\ 2y & -2 & 0 & 2x_1 \end{pmatrix}, \tag{4.17}$$

as the Jacobian of the augmented system, which is nonsingular for $x_1 = x_2 = 0$. Thus an inclusion is in principle possible, and indeed

```
>> f=inline('[x(1)^2-x(2)^2;x(1)-x(2)^2]'), ...
verifynlss2(f,[0.002;0.001])
f =
     Inline function:
     f(x) = [x(1)^2-x(2)^2;x(1)-x(2)^2]
intval ans =
  1.0e-323 *
[  -0.66666666666666,   0.66666666666666]
[  -1.00000000000000,   1.00000000000000]
[  -1.00000000000000,   1.00000000000000]
```

However, we mention that in this case the iteration is sensitive to the initial approximation as by

```
>> verifynlss2(f,[0.001;0.001])
intval ans =
[   0.49999999999999,   0.50000000000001]
[   0.70710678118654,   0.70710678118655]
[  -0.25000000000001,  -0.24999999999999]
```

which finds the double root $(0.5, 1/\sqrt{2})$ of $x^2 - y^2 + 0.25 = 0$ and $x - y^2 = 0$.

We might hope that there is always a renumbering of the equations such that for Jacobian $J_f(\widehat{x})$ of rank $n-1$ an inclusion of the root $\widehat{x}$ can be computed. Unfortunately this is not the case. Consider

$$f(x_1, x_2) = \begin{pmatrix} x_1^2 x_2 - x_1 x_2^2 \\ x_1 - x_2^2 \end{pmatrix} = 0\,. \tag{4.18}$$

The Jacobian of the augmented system computes to

$$J_F(x, e, y) = \begin{pmatrix} 2x_1x_2 - x_2^2 & x_1^2 - 2x_1x_2 & -1 & 0 \\ 1 & -2x_2 & 0 & 0 \\ 2(x_2y + x_1 - x_2) & 2(x_1 - x_2)y - 2x_1 & 0 & 2x_1x_2 - x_2^2 \\ 0 & -2 & 0 & 1 \end{pmatrix}, \tag{4.19}$$

Obviously the third row is entirely zero for $x_1 = x_2 = 0$, and this does not change when interchanging the two equations in (4.18). Note that the Jacobian J_f at the root is $\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ and forces the kernel vector to be $\begin{pmatrix} y_1 \\ 1 \end{pmatrix}$. Summarizing an inclusion for the root $(0,0)$ of (4.18) is in principle not possible by our method. But this situation is rare.

4.6 Numerical results

We add some numerical examples for the univariate and the multivariate case. We implemented the methods using (4.6) and (4.10), (4.11) in Algorithm `verifynlss2` in INTLAB. Following we display results of this algorithm.

First consider

$$f(x) = (\sin(x) - 1)(x - \alpha) \quad \text{for } \alpha := \frac{\pi}{2}(1 + \varepsilon)\,. \tag{4.20}$$

The function f has a double root $\widehat{x} = \pi/2$ with another simple root α of relative distance ε to $\pi/2$. Hence in any case we expect the inclusion E of the offset e for regularization to be a narrow inclusion of zero. Table 4.1 displays the results for different values of ε.

Table 4.1: Inclusions for the double root $\widehat{x} = \pi/2$ and a nearby simple root α for f as in (4.20).

ε	X	E
10^{-2}	$1.5707963267949 \pm 1.8 \cdot 10^{-14}$	$[-3.5, 1.8] \cdot 10^{-18}$
10^{-3}	$1.5707963267948 \pm 1.7 \cdot 10^{-13}$	$[-3.5, 1.8] \cdot 10^{-19}$
10^{-4}	$1.570796326795 \pm 1.6 \cdot 10^{-12}$	$[-3.5, 1.8] \cdot 10^{-20}$
10^{-5}	$1.57079632679 \pm 1.2 \cdot 10^{-10}$	$[-3.5, 1.8] \cdot 10^{-21}$
10^{-6}	$1.5707963268 \pm 1.5 \cdot 10^{-9}$	$[-3.5, 1.8] \cdot 10^{-22}$
10^{-7}	$1.570796327 \pm 1.6 \cdot 10^{-8}$	$[-3.5, 1.8] \cdot 10^{-23}$
10^{-8}	failed	

As can be seen for decreasing relative distance of α to the double root $\hat{x}$ the quality of the inclusion decreases. An inclusion is possible until about a relative error $10^{-8} \sim \sqrt{\varepsilon}$. This corresponds to the sensitivity of the double root $\hat{x}$: If there is another root α of relative distance $\sqrt{\varepsilon}$, then numerically the three roots cannot be distinguished in a floating-point arithmetic with relative rounding error unit ε. This effect can also be observed when changing f into

$$f(x) = (\sin(x) - 1)(x - \alpha)^2 \quad \text{for } \alpha := \frac{\pi}{2}(1 + \varepsilon)\,, \tag{4.21}$$

so that now there is a double root α near the double root $\hat{x}$. For a relative distance ε of about $\sqrt[4]{\varepsilon} \sim 10^{-4}$ the four roots behave like a quadruple root. This is confirmed by the results in Table 4.2.

Table 4.2: Inclusions for the double root $\hat{x} = \pi/2$ and a nearby double root α for f as in (4.21).

ε	X	E
10^{-2}	$1.57079632679488 \pm 1.2 \cdot 10^{-14}$	$[-2.8, 5.5] \cdot 10^{-20}$
10^{-3}	$1.5707963267948 \pm 2.4 \cdot 10^{-13}$	$[-2.8, 5.5] \cdot 10^{-22}$
10^{-4}	$1.570796326794 \pm 2.8 \cdot 10^{-12}$	$[-2.8, 5.5] \cdot 10^{-24}$
10^{-5}	failed	

Note that in both cases the inclusions of the offset for the regularization are very accurate inclusions of zero.

Next we test a system of nonlinear equations. The test function is

$$f(x_1, x_2) = \begin{pmatrix} \mathrm{e}^{x_1 x_2} - \sin(x_1^2 - 2x_1 x_2) \\ x_1(x_1 - \cosh(x_2)) + x_1 \operatorname{atan}(x_2) - \alpha \end{pmatrix} = 0\,, \tag{4.22}$$

where we choose the parameter α such that the system has a nearly double root. For $\alpha = 0.40031204474074$ there is an almost double root near $(1.329, -0.0273)$. The parameter α is chosen such that we can just separate the nearly double root into two single roots. The results are displayed in Table 4.3.

Table 4.3: Inclusions X_1, X_2 for two single roots and X for a nearly double root for f as in (4.22) and $\alpha = 0.4003120447407$.

X_1	X_2	X	E
$1.32889962\,1^{86}_{28}$	$1.3288995\,1^{57}_{48}$	1.32889956839071^{6}_{5}	
$-0.0272980\,5^{67}_{59}$	$-0.0272979\,2^{98}_{88}$	$-0.027297992758\,79^{41}_{34}$	[-5.2,-5.0]$\cdot 10^{-14}$

The inclusions of the two simple roots are separated by about 10^{-7} which is almost $\sqrt{\varepsilon}$, and the quality of the inclusion is about $\sqrt{\varepsilon}$ as well. Subtracting a constant $\varepsilon \in E$ from the first equation generates a truly double root. Note that $|\varepsilon| < 6 \cdot 10^{-14}$. As expected the quality

of the inclusion of the double root is much better than those of the separated simple roots, almost of maximum accuracy.

For $\alpha = 0.35653033083794$ there is another almost double root of f as in (4.22) near $(-0.292, 1.195)$. Again the parameter α is chosen such that we can just separate the nearly double root into to two single roots. The results are displayed in Table 4.4. The quality of the results is very similar to the previous example.

Table 4.4: Inclusions X_1, X_2 for two single roots and X for a nearly double root for f as in (4.22) and $\alpha = 0.35653033083794$.

X_1	X_2	X	E
-0.29197330^{91}_{44}	-0.2919733^{61}_{57}	$-0.291973333127^{64}_{29}$	
1.1950051^{23}_{00}	1.1950048^{69}_{53}	1.19500498575^{09}_{87}	[-1.17,-0.96]$\cdot 10^{-14}$

4.7 Conclusion

In this chapter we showed an efficient algorithms for computing verified and narrow error bounds with the property that a slightly perturbed system is proved to have a double root within the computed bounds. We have applied those to univariate polynomials, to multivariate polynomials and also to eigenvalue problems. Numerical experiments have confirmed the performance of our algorithms.

CHAPTER

FIVE

VALIDATION OF NUMERICAL CODES WITH MULTIPRECISION STOCHASTIC ARITHMETIC

In this chapter, we present a software designed for the validation of numerical codes with the use of multiprecision stochastic arithmetic. This chapter is based on the papers [28, 27].

5.1 Introduction

The increasing power of current computers makes it possible to solve more and more complex problems. Then it is necessary to perform a high number of floating-point operations, each one leading to a rounding error. Because of rounding error propagation, some problems must be solved with a longer floating-point format. This is the case, especially, for applications which carry out very complicated and enormous tasks in scientific fields, for example (see http://crd.lbl.gov/~dhbailey/dhbpapers/hpmpd.pdf):

- Quantum field theory
- Supernova simulation
- Semiconductor physics
- Planetary orbit calculations
- Experimental and computational mathematics.

Even if other techniques, methods or algorithms are employed to increase the accuracy of numerical results, some extended precision is still required to avoid severe numerical inaccuracies. Therefore some versions of scientific software carry out multiprecision computation. For instance, XBLAS routines [44] consist of a multiprecision version of basic linear algebra routines (BLAS). Moreover some libraries implement arbitrary precision arithmetic. MPFR [22] is an arbitrary precision library in C language. Freely available on various platforms, MPFR uses very efficient algorithms. Multiprecision libraries were described in Section 3.3 of Chapter 3.

Nevertheless, even with multiprecision, there are still some rounding errors and so it is still necessary to get some information about the numerical quality of the results. Validation

of numerical computations has been studied for a long time (see for example [77, 32, 20, 12]). Several approaches exist to control rounding error propagation: backward analysis [77] (used in LAPACK), stochastic backward analysis [12] (used in PRECISE), interval arithmetic [46, 2, 47], stochastic arithmetic [15, 74], and abstract interpretation-based static analysis [24]. These methods were described in Chapter 2. Based on MPFR, the MPFI library [55] provides arbitrary precision interval arithmetic. The problem is that it is difficult to validate huge scientific codes with interval arithmetic. If such a code is not rewritten to deal with intervals, its results would often be so overestimated that they would provide no information at all.

In this chapter, we present the SAM (Stochastic Arithmetic in Multiprecision) library which is based on MPFR and extends the features of discrete stochastic arithmetic by enabling arbitrary precision computation. It can be used to validate real-life scientific codes with few modifications of the code.

In Section 5.2, we describe the SAM library. Some examples of applications and numerical experiments are presented in Section 5.3.

5.2 The SAM library

The SAM library implements in arbitrary precision the features of DSA (see Section 2.3.3 of Chapter 1): the stochastic types, the concept of computational zero and the stochastic operators. The particularity of SAM (compared to CADNA) is the arbitrary precision of stochastic variables. The SAM library with 24-bit (resp. 53-bit) mantissa length is similar to CADNA in single (resp. double) precision, except the range of the exponent is only limited by the machine memory. In SAM, the number of exact significant digits of any stochastic variable is estimated with the probability 95 %, whatever its precision. Like in CADNA, the arithmetic and relational operators in SAM take into account rounding error propagation. All numerical instabilities which occur at run time are detected. Such instabilities are usually generated by an operation involving a computational zero.

The SAM library is written in C++ and is based on MPFR. In the SAM library, all operators are overloaded. Consequently for a program written in C++ to be used with SAM, only a few modifications are needed: mainly changes in type declarations. Classical variables have to be replaced by multiprecision stochastic variables (of `mp_st` type) which consist of three variables of MPFR type and one integer variable to store the accuracy. In SAM, for each stochastic operation, three MPFR operations are performed using different rounding modes and the numerical instability that may be generated is detected. As a remark, in order to avoid using the same rounding mode for the three operations, the first two operations are performed using a rounding mode randomly chosen and the third rounding mode is the opposite of the second one. For instance, if the result of the second operation is rounded up, then the result of the third operation is rounded down.

The use of the SAM library in a C++ program involves five steps described below.

1. The SAM library has to be included in the C++ program using the `#include "sam.h"` directive.
2. The `sam_init` initialization function has to be called once. In the following instruction
   `sam_init(nb_instabilities, nb_bits);`
   `nb_instabilities` is the maximum number of numerical instabilities to be detected and `nb_bits` is the mantissa length in bits. If the value of the first argument is -1, all the instabilities will be detected.
3. In variable declarations, the `float` or `double` type has to be replaced by the `mp_st` stochastic type.
4. The `strp` function returns a string which contains the exact significant digits of a stochastic result, i.e. its significant digits not affected by rounding errors. If the result is insignificant, the `strp` function returns "@.0". Therefore output statements should be modified to print stochastic results with their accuracy using the `strp` function.
5. The `sam_end` function has to be called to print a report on the numerical instabilities which possibly occurred during the execution.

As an example, a simple program which uses the SAM library is given in 5.3.1.

5.3 Numerical experiments

5.3.1 Computing a rational function of two variables

In the following example proposed by S. M. Rump [57], the rational function

$$f(x,y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + \frac{x}{2y}$$

is computed with $x = 77617$, $y = 33096$. The associated program written using the SAM library is:

```
#include "sam.h"
#include <stdio.h>
int main() {
   sam_init(-1,122);
   mp_st x = 77617, y = 33096, res;
   res=333.75*y*y*y*y*y*y*y+x*x*(11*x*x*y*y-y*y*y*y*y*y*y
      -121*y*y*y*y-2.0)+5.5*y*y*y*y*y*y*y*y*y+x/(2*y);
   printf("res=%s\n",strp(res));
   sam_end();
}
```

Because the first argument in the `sam_init` function is -1, the SAM library will detect all the numerical instabilities which will possibly occur at run time. Using SAM with 122 bits, one obtains:

```
------------------------------------------------------------
SAM software --- University P. et M. Curie --- LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
------------------------------------------------------------
res=-0.827396059946821368141165095479816292
------------------------------------------------------------
SAM software --- University P. et M. Curie --- LIP6
No instability detected
------------------------------------------------------------
```

Using SAM with 53 bits, the result obtained has no exact significant digits. Furthermore SAM detects a cancellation, which is a severe loss of accuracy due to the subtraction of two nearly equal numbers. Let us decompose the expression into three terms: $f(x,y) = a+b+c$ with $a = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2)$, $b = 5.5y^8$ and $c = x/(2y)$. SAM with 53 bits provides two opposite results for a and b:
`a=-0.791711134066896E+037`
`b= 0.791711134066896E+037`.
The computed results a and b have the same 15 exact significant digits. Their sum is insignificant, because it is only due to rounding errors. The three samples representing this sum are:
`-1.1805916207174113E+021`, `-1.1805916207174113E+021`, `-0.0000000000000000E+000`.
The computed result c has 15 exact significant digits:
`c= 0.117260394005317E+001`.
The final result $a+b+c$ is insignificant. It is represented by the three samples:
`-1.1805916207174113E+021`, `-1.1805916207174112E+021`, `+1.1726039400531785E+000`.

Table 5.1 shows that if computations carried out with different precisions lead to similar approximations, it does not mean that the results have a good accuracy.

single precision	1.172603
double precision	1.1726039400531
extended precision	1.172603940053178
Variable precision interval arithmetic	[−0.827396059946821368141165095479816292005, −0.827396059946821368141165095479816291986]
SAM, $\leqslant$121 bits	@.0
SAM, 122 bits	−0.827396059946821368141165095479816292

Table 5.1: Computation of $f(x,y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$ with $x = 77617$ and $y = 33096$.

Indeed, it is shown that the evaluations of Rump's polynomial f in single, double and extended precision share the same first digits. But in fact, the correct evaluation is totally

different from that result. This is detected by SAM. Until 121 bits, SAM says that the result has no exact significant digits. It is only starting with 122-bit mantissa length that SAM says that we have full accuracy. It is verified by computing the exact result with the Maple computer algebra system. It is noticeable that the result provided by SAM with 122-bit mantissa length is included in the interval computed using variable precision interval arithmetic.

5.3.2 Computing a second order recurrent sequence

This sequence was proposed by J.-M. Muller [49]. The first 30 iterations of the following recurrent sequence are computed:

$$U_{n+1} = 111 - \frac{1130}{U_n} + \frac{3000}{U_n U_{n-1}}$$

with $U_0 = 5.5$ and $U_1 = \frac{61}{11}$. The exact limit is 6.

Using IEEE double precision arithmetic with rounding to the nearest, one obtains:

```
U(3) = +5.590163934426237e+00
(...)
U(11) = +5.861018785996283e+00
U(12) = +5.882524608269310e+00
U(13) = +5.918655323805488e+00
U(14) = +6.243961815306110e+00
U(15) = +1.120308737284091e+01
U(16) = +5.302171264499677e+01
U(17) = +9.473842279276452e+01
U(18) = +9.966965087355071e+01
U(19) = +9.998025776093678e+01
U(20) = +9.999882245337588e+01
(...)
U(30) = +1.000000000000000e+02
```

Until the 13th iteration, the sequence seems to converge to 6. Then it seems to converge to 100, although the exact limit is 6.

Using SAM in double precision (with 53-bit mantissa length), one obtains:

```
-----------------------------------------------------------------
SAM software --- University P. et M. Curie --- LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
-----------------------------------------------------------------
U(3) =  0.5590163934426E+1
U(4) =  0.563343108504E+1
U(5) =  0.56746486205E+1
```

```
U(6) =  0.571332905E+1
U(7) =  0.57491209E+1
U(8) =  0.5781811E+1
U(9) =  0.581131E+1
U(10) =  0.58376E+1
U(11) =  0.586E+1
U(12) =  0.59E+1
U(13) =  0.6E+1
U(14) =  @.0
U(15) =  @.0
U(16) =  @.0
U(17) =  @.0
U(18) =  0.9E+2
U(19) =  0.99E+2
U(20) =  0.999E+2
(...)
U(30) =  0.100000000000000E+3
------------------------------------------------------------
SAM software --- University P. et M. Curie --- LIP6
CRITICAL WARNING: the self-validation detects major problem(s).
The results are NOT guaranteed.
There are 12 numerical instabilities
9 UNSTABLE DIVISION(S)
3 UNSTABLE MULTIPLICATION(S)
------------------------------------------------------------
```

First the sequence seems to converge to 6, losing regularly exact significant digits to such an extent that iterates 14 to 17 are insignificant. Then the sequence converges to 100, improving regularly its numerical quality. The accuracy of the 30th iterate (15 exact significant digits) is optimal for a computation carried out in double precision. Unfortunately results from the 18th iteration are incorrect. The instabilities reported at the end of the execution are unstable divisions and unstable multiplications which may invalidate the first order approximation in the CESTAC method. A warning informs the user that the results may not be reliable.

Using SAM with 100 bits, one obtains:

```
------------------------------------------------------------
SAM software --- University P. et M. Curie --- LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
------------------------------------------------------------
U(3) =  0.559016393442622950819672131E+1
(...)
U(22) =  0.5979E+1
```

```
U(23) =  0.598E+1
U(24) =  0.59E+1
U(25) =  @.0
U(26) =  @.0
U(27) =  @.0
U(28) =  @.0
U(29) =  0.9E+2
U(30) =  0.99E+2
U(31) =  0.999E+2
(...)
U(52) =  0.10000000000000000000000000000E+3
----------------------------------------------------------------
SAM software --- University P. et M. Curie --- LIP6
CRITICAL WARNING: the self-validation detects major problem(s).
The results are NOT guaranteed.
There are 12 numerical instabilities
9 UNSTABLE DIVISION(S)
3 UNSTABLE MULTIPLICATION(S)
----------------------------------------------------------------
```

The sequence obtained using SAM with 100 bits has the same behavior as the previous one (computed with 53 bits). The sequence seems to converge to 6, then four iterates have no exact significant digits. These insignificant results appear later, at iterations 25 to 28. From the 29th iteration, the sequence converges to 100, improving regularly its accuracy. Like in the previous execution, a self-validation of SAM has been performed during the run. Because unstable divisions and unstable multiplications occurred, the results are not reliable.

5.3.3 Computation of a chaotic dynamical system

It is always difficult to simulate chaotic dynamical systems [19, 5] with a computer because rounding errors can change dramatically their behavior. Nevertheless, some results can be proved on chaotic systems using rigorous numerical computations and in particular interval computations (see for example [71, 23, 72]). In this subsection, we study the computation of the logistic map using stochastic arithmetic with SAM and we compare with MPFI the results obtained.

Let us consider the logistic iteration [19] defined by $x_{n+1} = ax_n(1-x_n)$ with $a > 0$ and $0 < x_0 < 1$.

- When $a < 3$ this sequence converges to a unique fixed point, whatever the initial condition x_0 is.
- When $3 \leqslant a \leqslant 3.57$ this sequence is periodic, whatever the initial condition x_0 is, the periodicity depending only on a. Furthermore the periodicity is multiplied by 2 for some values of a called “bifurcations”.
- When $3.57 < a < 4$ this sequence is usually chaotic, but there are certain isolated values of a that appear to show periodic behavior.
- Beyond $a = 4$, the values eventually leave the interval $[0,1]$ and diverge for almost all initial values.

The logistic map has been computed with $x_0 = 0.6$ using SAM and MPFI. In stochastic arithmetic, iterations have been performed until the current iterate is a computational zero, *i.e.* all its digits are affected by rounding errors. In interval arithmetic, iterations have been performed until the two bounds of the interval have no common significant digits. In Tables 5.2 and 5.3, we report the number N of iterations performed for two ways of computing the logistic map. We also report the execution times. With SAM two types of executions have been performed:

- without the detection of numerical instabilities
- with the detection of all numerical instabilities which may occur at run time.

In Table 5.2, we have computed the logistic map using the formula

$$x_{n+1} = a x_n (1 - x_n) \quad \text{with} \quad x_0 = 0.6. \tag{5.1}$$

During the computation performed using SAM with $a = 3.57$, no computational zero has been detected. The program has been stopped after one million iterations. The run time reported in Table 5.2 has been measured for one million iterations. Whereas using MPFI with the same value for a, at a certain iteration the stopping criterion is satisfied: the bounds of the last interval have no common digits. Using 500 000 bits, fewer than 300 000 iterations are performed in about 6 hours. Since the number N of iterations performed increases linearly as the precision increases, about 2 millions bits would be required to perform one million iterations. If $a = 3.575$, $a = 3.6$ or $a = 3.7$, more iterations are performed with SAM than with MPFI for the stopping criterion to be satisfied. However it must be pointed out that SAM is based on an estimation of rounding errors. Results obtained with MPFI are more pessimistic, but are guaranteed. If $a = 10$, the sequence diverges. Similar results are obtained with SAM and MPFI. The run times measured with SAM are longer than those measured with MPFI. The main reason is the number of iterations, higher with SAM than with MPFI. In the numerical experiment presented in 5.3.4, the same computation is carried out using SAM and MPFI in order to compare their performance. The execution time is higher with SAM when the detection of instabilities is enabled. However, in this application, the cost of this detection remains reasonable.

In Table 5.3, we have computed the logistic map using the formula

$$x_{n+1} = -a\left(x_n - \frac{1}{2}\right)^2 + \frac{a}{4} \quad \text{with} \quad x_0 = 0.6. \tag{5.2}$$

Concerning SAM, the results are very similar to those in Table 5.2 (similar N). Nevertheless, the results are better with MPFI compared to Table 5.2. This can be explained by the so-called *dependency problem* which is a major drawback of interval arithmetic. Indeed, if an interval occurs several times in an expression, each occurrence is taken independently and then can lead to an unwanted over-estimation of the resulting interval. This is the case in equation (5.1) where the variable x_n appears twice whereas x_n appears only once in equation (5.2). As a remark, if $a = 3.57$, no computational zero has been detected using SAM. The run time reported in Table 5.3 has been measured for one million iterations. Using MPFI with $a = 3.57$, more than 15 000 bits are required to perform one million iterations.

a		# bits	N	Execution time (in s)	
3.57	SAM			without detection	with all detections
		24	$> 10^6$	6.668	12.093
		53	$> 10^6$	7.192	13.565
		100	$> 10^6$	9.045	17.485
		200	$> 10^6$	11.565	23.277
		2000	$> 10^6$	160.39	360.65
	MPFI	24	11	$< 10^{-3}$	
		53	27	$< 10^{-3}$	
		100	53	$< 10^{-3}$	
		200	108	0.004	
		2000	1088	0.040	
		5 000	2 722	0.316	
		50 000	27 232	97.270	
		500 000	272 341	20901.6	
3.575	SAM			without detection	with all detections
		24	110	0.008	0.008
		53	324	0.012	0.012
		100	722	0.016	0.020
		200	1526	0.020	0.044
		2000	15896	2.585	5.804
	MPFI	24	12	$< 10^{-3}$	
		53	27	$< 10^{-3}$	
		100	53	$< 10^{-3}$	
		200	108	0.004	
		2000	1087	0.032	
3.6	SAM			without detection	with all detections
		24	52	$< 10^{-3}$	0.008
		53	150	0.008	0.008
		100	338	0.011	0.016
		200	718	0.012	0.028
	MPFI	24	12	$< 10^{-3}$	
		53	27	$< 10^{-3}$	
		100	53	$< 10^{-3}$	
		200	107	0.004	
3.7	SAM			without detection	with all detections
		24	37	$< 10^{-3}$	0.008
		53	97	0.004	0.008
		100	193	0.008	0.012
		200	387	0.012	0.020
	MPFI	24	11	$< 10^{-3}$	
		53	27	$< 10^{-3}$	
		100	52	$< 10^{-3}$	
		200	105	0.004	
10	SAM			without detection	with all detections
		24	21	$< 10^{-3}$	0.004
		53	29	0.008	0.008
		100	29	0.008	0.008
		200	29	0.008	0.008
	MPFI	24	29	$< 10^{-3}$	
		53	29	$< 10^{-3}$	
		100	29	$< 10^{-3}$	
		200	29	$< 10^{-3}$	

Table 5.2: Logistic map: number N of iterations performed with SAM and MPFI, $x_{n+1} = ax_n(1-x_n)$ with $x_0 = 0.6$.

a		# bits	N	Execution time (in s)	
3.57	SAM			without detection	with all detections
		24	$> 10^6$	10.017	20.357
		53	$> 10^6$	10.565	22.669
		100	$> 10^6$	12.665	28.706
		200	$> 10^6$	15.389	37.154
		2000	$> 10^6$	164.85	541.14
	MPFI	24	394	$< 10^{-3}$	
		53	1990	0.004	
		100	5044	0.016	
		200	11310	0.036	
		2000	123220	5.376	
		5 000	311 828	55.691	
		15 000	935 556	913.40	
		16 500	$> 10^6$	1064.9	
		50 000	$> 10^6$	5818.7	
3.575	SAM			without detection	with all detections
		24	110	0.008	0.012
		53	354	0.012	0.016
		100	722	0.016	0.032
		200	1548	0.032	0.068
		2000	15890	2.636	8.673
	MPFI	24	108	$< 10^{-3}$	
		53	324	$< 10^{-3}$	
		100	722	$< 10^{-3}$	
		200	1534	0.008	
		2000	15882	0.696	
3.6	SAM			without detection	with all detections
		24	56	$< 10^{-3}$	0.008
		53	152	0.008	0.012
		100	342	0.008	0.020
		200	722	0.020	0.036
	MPFI	24	56	$< 10^{-3}$	
		53	152	$< 10^{-3}$	
		100	338	$< 10^{-3}$	
		200	722	0.004	
3.7	SAM			without detection	with all detections
		24	39	0.004	0.008
		53	106	0.008	0.008
		100	198	0.012	0.012
		200	396	0.016	0.024
	MPFI	24	39	$< 10^{-3}$	
		53	103	$< 10^{-3}$	
		100	197	$< 10^{-3}$	
		200	389	0.004	
10	SAM			without detection	with all detections
		24	21	$< 10^{-3}$	0.008
		53	29	0.008	0.008
		100	29	0.012	0.012
		200	29	0.012	0.020
	MPFI	24	29	$< 10^{-3}$	
		53	29	$< 10^{-3}$	
		100	29	$< 10^{-3}$	
		200	29	$< 10^{-3}$	

Table 5.3: Logistic map: number N of iterations performed with SAM and MPFI, $x_{n+1} = -a(x_n - \frac{1}{2})^2 + \frac{a}{4}$ with $x_0 = 0.6$.

Even if SAM has been designed for arbitrary precision, it can be compared with CADNA if the chosen precision is 24 bits or 53 bits. The results are, in general, similar with CADNA in double precision (53 bits) and SAM with 53 bits. It is normal since the operations are correctly rounded with the same precision in both cases. When there is a difference, it is due to the fact that the exponent of the floating-point numbers is not limited in SAM, whereas it is limited to 1023 in double precision. The same explanation applies for single precision and SAM with 24 bits (here the exponent in single precision is limited to 127).

5.3.4 Performance test

The performance of SAM has been compared with that of MPFI. The matrix multiplication $M \cdot M$, where $M_{i,j} = i + j - 1$ ($1 \leqslant i \leqslant N$, $1 \leqslant j \leqslant N$) has been computed using SAM and MPFI with different precisions and different values of N. Three types of executions have been performed using SAM:

- without the detection of numerical instabilities
- with the detection of instabilities that may occur in multiplications, divisions or calls to the power function and may invalidate the CESTAC method, as mentioned in Section 3; this detection enables a self-validation of the SAM library
- with the detection of all instabilities.

The run times reported in Table 5.4 have been measured for $N = 100$ on an Intel Q9550 2.83GHz processor using the g++ 4.4 compiler.

# bits	24	53	100	500	1000	5000
MPFI	0.292	0.320	0.432	0.504	0.648	2.216
SAM without detection	0.892	0.936	1.076	1.120	1.372	2.616
SAM with self-validation	0.896	0.940	1.092	1.160	1.380	2.624
SAM with all detections	7.168	8.357	10.461	27.254	69.588	903.528
SAM with self-validation/MPFI	3.07	2.94	2.53	2.30	2.13	1.18

Table 5.4: Run time (in seconds) for a matrix multiplication of size 100

From Table 5.4, it can be noticed that MPFI performs better than SAM. When self-validation is activated, the time ratio varies from 1.2 to 3.1, depending on the working precision. The cost of self-validation in SAM (like in CADNA) is negligible. The detection of all numerical instabilities is an interesting feature of SAM, unfortunately it may be very costly, depending on the application. This cost is mainly due to the detection of cancellations which may occur in additions or subtractions. From tests carried out for various values of N (from 50 to 1000) it is noticeable that the time ratio between SAM and MPFI is independent of N. Anyway our main objective was to show that the resources required by SAM with self-validation are of the same order of magnitude as MPFI. This makes SAM useful for the validation of huge numerical codes.

5.4 Conclusion

We have demonstrated that SAM can be very useful to study the behavior of chaotic dynamical system. We have only studied the behavior of the logistic map. We plan to do a similar work with other systems like the Hénon map [52] or the Lorenz attractor [78].

We have also compared in terms of efficiency and computing times the SAM library with MPFI. As mentioned previously, SAM and MPFI do not give the same answer since MPFI leads to a certified answer whereas SAM gives an answer true within a given probability. Anyway, it is sometimes sufficient to know the answer only with high probability. The main advantage of SAM is that it can be used to validate huge numerical scientific codes.

CONCLUSION AND FUTURE WORK

Conclusion

In this HDR thesis, we have presented some of our recent results concerning the increase of accuracy and the validation of numerical algorithms.

We first proposed an introduction to computer arithmetic (more precisely floating-point arithmetic) and to some methods for rounding error analysis (this includes forward and backward error analysis, interval arithmetic and stochastic arithmetic). These are the tools we frequently use in our researches.

We then described our main research topic that is increasing the accuracy of numerical algorithms. For that, we use the so-called *error-free transformations*. We compute the elementary rounding errors and we correct the result with an approximation of all the rounding errors that have occurred during the computation. Such algorithms are called *compensated algorithms*. If the first one dates back to Kahan, such method was extensively studied by Rump, Ogita and Oishi for summation and dot product. We have applied this method to nonlinear problems like polynomial evaluation, derivative evaluation mainly. We have then used those algorithms for some applications like Newton's method to accurately compute simple roots of polynomials. Our algorithms run faster than the classic ones used with multiprecision arithmetic libraries.

We have also presented some results on the verification of mathematical assumptions on computers. We first described the classic one concerning the verification of the nonsingularity of a matrix in finite precision. Then we tackled an ill-posed problem concerning the multiplicity of roots of nonlinear equations. Indeed, it is well known that deciding whether a univariate polynomial has a multiple root is an ill-posed problem: an arbitrary small perturbation of a polynomial coefficient may change the answer from yes to no. We described a verification method for computing guaranteed error bounds for double roots of systems of nonlinear equations. To circumvent the principle problem of ill-posedness we proved that a slightly perturbed system of nonlinear equations has a double root.

Finally, we presented the SAM (Stochastic Arithmetic in Multiprecision) library which

is based on MPFR and extends the features of discrete stochastic arithmetic by enabling arbitrary precision computation. It can be used to validate real-life scientific codes with few modifications of the code.

Future work

We present now the research axis we would like to develop in the next few years. It mainly concerns computer arithmetic and the validation of numerical algorithms. An axis concerning parallel algorithms is also considered.

The project research is divided into 4 parts:

- increasing the accuracy;
- validation and certification;
- symbolic-numerical algorithms;
- numerical reproducibility and parallel algorithms.

Increasing the accuracy

The accuracy of a computed result depends mainly on 3 factors: the condition number of the problem, the stability of the algorithm and the precision of the arithmetic. It is also necessary to take into account the approximation errors due to the numerical scheme used. The condition number measures the difficulty to solve a problem independently of the algorithm used to solve this problem.

The stability of an algorithm describes the influence of the computations in finite precision on the accuracy of the computed result. Finally, concerning the arithmetic, it is often the IEEE 754 floating-point arithmetic which makes it possible a maximal precision of about 16 decimal digits in double precision (binary64). This precision is sometimes not sufficient for solving accurately some ill-conditioned problems.

As we have shown in chapter 3, there are several solutions to increase the precision. Some are hardware solutions like 80 bits registers in x87. Others are software solutions like multiprecision libraries with MPFR [22] for example.

Another way to increase the precision of the computations is to estimate the rounding errors that occurred during the computation and accumulate them in order to correct the computed result. This is the choice we used in our researches. We have shown in chapter 3 that these methods are faster than using multiprecision libraries.

We plan to continue working on that topic. Indeed for computing accurately roots of polynomials, it is necessary to accurately evaluate a polynomial. This is particularly true when the root is a multiple root or is very close to a multiple one. In that case, it will be useful to be able to compute a faithfully rounded evaluation of a polynomial. We think this is possible at a mild cost by iterating our compensated Horner scheme [6]. What is difficult is to find a tight error bound to stop as soon as possible.

The computation of the 2-norm (Euclidean norm) of a vector is intensively used in numerical linear algebra codes. Even if this computation is well-conditioned, the size of the problems (and so the size of the vectors) is growing and implies that the rounding errors are more and more important (the bound on the error is proportional to the size of the vector). Our aim is to propose some new algorithms to efficiently and accurately compute the 2-norm of a vector (useful for example in QR decomposition). By accurate, we mean a faithful rounding of the exact result and if needed a rounded to nearest result with a limited cost. Similar results for vectors of length 2 has already been obtained [10].

Our long term goal should be to provide some accurate building blocks with a limited cost for lots of classic algorithms. They could be used in applications where accurate results are needed (linear algebra, equation solving, and so on).
Moreover, we would like to automatize the design of compensated algorithms. In our papers, we describe the use of a graph that could be used to automatically design a compensated algorithm together with a proof on the accuracy. The use of proof checker like Coq could be considered as well. The tool would take in input code and would output a compensated version of this code that could also give a certified error bound computed in floating-point arithmetic.

Recently, Jeannerod and Rump [35, 63] have shown that for $x_i \in \mathbf{F}$,

$$\left| \mathrm{fl}(\sum_{i=1}^{n} x_i) - \sum_{i=1}^{n} x_i \right| \leqslant (n-1)\mathbf{u} \sum_{i=1}^{n} |x_i|$$

which is better than the previous bound

$$\left| \mathrm{fl}(\sum_{i=1}^{n} x_i) - \sum_{i=1}^{n} x_i \right| \leqslant \gamma_{n-1} \sum_{i=1}^{n} |x_i|$$

with $\gamma_n = n\mathbf{u}/(1-n\mathbf{u}) = n\mathbf{u} + \mathcal{O}(\mathbf{u}^2)$. It would be interesting to know if we have a similar relation for polynomial evaluation, that is to say,

$$\left| \mathrm{fl}(\sum_{i=0}^{n} a_i x^i) - \sum_{i=1}^{n} a_i x^i \right| \leqslant 2n\mathbf{u} \sum_{i=1}^{n} |a_i x^i|$$

instead of

$$\left| \mathrm{fl}(\sum_{i=0}^{n} a_i x^i) - \sum_{i=1}^{n} a_i x^i \right| \leqslant \gamma_{2n} \sum_{i=1}^{n} |a_i x^i|.$$

We did some intensive random tests with no counter-example found in single precision.

So we focused on a more simple relation. Do we have, for positive $x \in \mathbf{F}$, $|\mathrm{fl}(x^n) - x^n| \leqslant (n-1)\mathbf{u}x^n$? We have no mathematical proof but we rigorously checked for all $x \in [1;2[$ in `binary32` (2^{23} numbers to test) until overflow for x^n. For $x = 1+2\mathbf{u}$, $n \approx 7.5 \times 10^8$ is needed to reach overflow. For that we used of interval arithmetic with 100 bits in MPFI.

Numerical validation and certification

We have recently proposed in [5] an algorithm that makes it possible to detect some singularities (for exemple to detect a multiple root) in the neighborhood of a polynomial system. This method can be used to detect multiple eigenvalues in the neighborhood of a given matrix. Our result is certified in the sense that we give a rigorous error bound concerning the size of the perturbation needed to make the system singular. We plan to extend this technique to detect singular matrices under structured perturbations. This could be used to detect non-coprimeness of polynomials known with uncertainty (using the singularity of Sylvester matrix).

Some problems can be reduce to a computation of polynomial roots. The *pseudozero set* [48, 70] represents the roots of all the polynomials that are closed to a given one. This concept naturally arises in applications where polynomials are only known with uncertainty on their coefficients or when computations are performed in finite precision. In general it is difficult to test if a number is a pseudozero because of rounding error (it needs to evaluate a polynomial near a root). Our aim is to propose a certified method in floating-point arithmetic to test if a number is really a pseudozero.

Moreover, the computation of pseudozero can generally be done in polynomial time. But the complexity depends on the type of perturbations we consider. For some types of perturbations, there exists, for the moment, no polynomial time algorithm. We think in that case that this is a NP-hard problem. Some recent results on complexity theory let us think that there exist some tools that could lead to a proof of the NP-hardness of such problems.

In [56], the authors presented a tool called PRECIMONIOUS which is a dynamic program analysis tool that makes it possible to help developers for tuning the precision of floating-point programs. PRECIMONIOUS makes a search on the types of the floating-point variables to try to lower their precision without degrading the final accuracy. For that, they use the *delta-debugging* analysis. We plan to provide a similar tool but we will replace delta-debugging by our tool SAM. Indeed, contrary to delta-debugging we can have access with high probability to the loss of accuracy for each variables. This could lead to choose the initial precision needed for each variables.

Symbolic-numeric algorithms

The accurate computation of elementary functions needs the use of methods from computer algebra especially during the phase of approximation. To certify the errors, it is sometimes possible to use sum of square (SOS). Some symbolic-numeric algorithms exist to compute SOS (floating-point computation by projection and then lifting to the rationals [39]).

The sequence begins with the function and go to a certified numerical code through the use of symbolic-numeric tools. Our long term goal is to be able to have the whole sequence in a single tool.

More generally, we can code functions in different ways. One possible way is to use

ODE (ordinary differential equations). This gives a general framework for the evaluation of functions. Indeed, lots of functions can be defined as a solution of an ODE of the form $x'(t) = f(x(t), t)$. As a consequence, their evaluations need to solve precisely these equations (see the PhD thesis of Marc Mezzarobba [45] on the evaluation of D-finite functions). A possible method of resolution consists in using an expansion in power series of the solution $x(t)$ and of the function f. We then obtain a recursive sequence that needs to be evaluated (see the book of Warwick Tucker [73, chap. 6]).

The use of intervals to bound the remainder in the expansion in power series makes it possible to obtain a certified error bound on the solution. A research axis could be to look at what happens if we use other basis for the expansion in power series. We could use for example the Chebychev polynomials [9] or other orthogonal polynomials. Indeed, some solutions can be expressed in a basis of eigenvectors which are orthogonal polynomials (see spectral methods [9]).

It is then natural to look for solutions in such basis. Our researches could then be on the numerical stability of the computation of the recursive sequence. A change of basis is then necessary to rewrite the solution in the power basis which is in general ill-conditioned. A possible solution could be to use symbolic-numeric computation for this part.

Reproducibility and accuracy for Exascale computing

The increasing power of current computers enables one to solve more and more complex problems. Then it is necessary to perform a high number of floating-point operations, each one leading to a round-off error. Because of round-off error propagation, some problems must be solved with a longer floating-point format. This is the case, especially, for applications which carry out very complicated and enormous tasks in scientific fields, like quantum field theory, or planetary orbit calculations for instance.

Exascale computing (10^{18} operations per second) is likely to be reached within a decade. With such computers, getting accurate results in floating-point arithmetic will be a challenge. But another challenge will be the reproducibility of the results. Indeed, due to non-associativity of floating-point operations, it is very difficult to obtain the same result in different runs of a code on parallel computers (due for example to dynamic scheduling).

By reproducibility, we mean getting a bitwise identical floating-point result from multiple runs of the same code. This is an important property very useful for debugging or checking the correctness of codes. Reproducibility is becoming so important that Intel proposed a "Conditional Numerical Reproducibility" (CNR) in its MKL (Math Kernel Library). But CNR is slow and does not give any guarantee about the accuracy of the result. A very recent result was obtain by James Demmel and Hong Diep Ngyuen from UC Berkeley [18]. They propose a new algorithm for reproducible summation in floating-point arithmetic. Their algorithm is fast and always returns the same answer. Nevertheless, no information is given on the accuracy of the result.

Our aim is to provide a reproducible, accurate and fast library for classic operations

(summation, dot product) on new parallel architectures that are GPU and Intel MIC (Xeon Phi). Indeed, we plan to take into account the hardware specificity of those architectures. For that, we will use a *long accumulator* proposed by Ulrich Kulisch [42]. This accumulator makes it possible to accurately (exactly) compute a summation or a dot product. As a consequence, the results will be exact and so reproducible (no rounding errors). An Exact Dot Product (EDP) is also a useful basic tool for accurate interval arithmetic. As the use of long accumulators need memory transfers, it will be necessary to efficiently use the memory hierarchy and mainly cached memory.

BIBLIOGRAPHY

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–58, 2008.

[2] G. Alefeld and J. Herzberger. *Introduction to interval analysis*. Academic Press, 1983.

[3] G. Alefeld and H. Spreuer. Iterative improvement of componentwise error bounds for invariant subspaces belonging to a double or nearly double eigenvalue. *Computing*, 36(4):321–334, 1986.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[5] J. Argyris, G. Faust, and M. Haase. *An exploration of chaos*. Texts on Computational Mechanics, VII. North-Holland Publishing Co., Amsterdam, 1994.

[6] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson. ARPREC: An Arbitrary Precision Computation Package. Technical Report LBNL-53651, Lawrence Berkeley National Laboratory, September 2002.

[7] F.B. Baker and M.R. Harwell. Computing elementary symmetric functions and their derivatives: A didactic. *Appl. Psychol. Meas.*, 20(2):169–192, 1996.

[8] R. P. Brent. Algorithm 524: MP, A Fortran Multiple-Precision Arithmetic Package. *ACM Trans. Math. Softw.*, 4(1):71–81, 1978.

[9] N. Brisebarre and M. Joldeş. Chebyshev interpolation polynomial-based tools for rigorous computing. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 147–154, New York, NY, USA, 2010. ACM.

[10] N. Brisebarre, M. Joldes, P. Kornerup, É. Martin-Dorel, and J.-M. Muller. Augmented precision square roots and 2-d norms, and discussion on correctly rounding $\sqrt{(}x^2+y^2)$. In *IEEE Symposium on Computer Arithmetic*, pages 23–30, 2011.

[11] Daniela Calvetti and Lothar Reichel. On the evaluation of polynomial coefficients. *Numer. Algorithms*, 33(1-4):153–161, 2003.

[12] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

[13] F. Chaitin-Chatelin and V. Frayssé. *Lectures on finite precision computations*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1996.

[14] J.-M. Chesneaux. Study of the computing accuracy by using probabilistic approach. In C. Ullrich, editor, *Contribution to Computer Arithmetic and Self-Validating Numerical Methods*, pages 19–30, IMACS, New Brunswick, New Jersey, USA, 1990.

[15] J.-M. Chesneaux. *L'arithmétique stochastique et le logiciel CADNA*. Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, November 1995.

[16] J.-M. Chesneaux, S. Graillat, and F. Jézéquel. *Encyclopedia of Computer Science and Engineering*, volume 4, chapter Rounding Errors, pages 2480–2494. Wiley, 2009.

[17] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18:224–242, 1971.

[18] J. Demmel and H. D. Nguyen. Fast reproducible floating-point summation. In *Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, TX, USA, April 7-10*, pages 163–172, 2013.

[19] R. L. Devaney. *An introduction to chaotic dynamical systems*. Addison-Wesley Studies in Nonlinearity. Addison-Wesley Publishing Company Advanced Book Program, Redwood City, CA, second edition, 1989.

[20] B. Einarsson and al. *Accuracy and Reliability in Scientific Computing*. Software-Environments-Tools. SIAM, Philadelphia, PA, 2005.

[21] A. Eisinberg and G. Fedele. A property of the elementary symmetric functions. *Calcolo*, 42(1):31–36, 2005.

[22] 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):13, 2007. http://www.mpfr.org.

[23] Z. Galias and W. Tucker. Rigorous study of short periodic orbits for the Lorenz system. In *Proc. IEEE Int. Symposium on Circuits and Systems, ISCAS'08*, pages 764–767, Seattle, May 2008.

[24] E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In *Proceedings of Formal Methods in Industrial Critical Systems, LNCS 4916*. Springer-Verlag, 2007.

[25] S. Graillat. Accurate simple zeros of polynomials in floating point arithmetic. *Comput. Math. Appl.*, 56(4):1114–1120, 2008.

[26] S. Graillat. Accurate floating point product and exponentiation. *IEEE Transactions on Computers*, 58(7):994–1000, 2009.

[27] S. Graillat, F. Jézéquel, S. Wang, and Y. Zhu. Stochastic arithmetic in multiprecision. *Math.comput.sci.*, 5(4):359–375, 2011.

[28] S. Graillat, F. Jézéquel, and Y. Zhu. Stochastic arithmetic in multiprecision. In *NSV3, Third International Workshop on Numerical Software Verification, Edinburgh, UK, July 15th*, 7 pages, 2010.

[29] S. Graillat, Ph. Langlois, and N. Louvet. Algorithms for accurate, validated and fast polynomial evaluation. *Japan J. Indust. Appl. Math.*, 2-3(26):191–214, 2009. Special issue on State of the Art in Self-Validating Numerical Computations.

[30] S. Graillat, N. Louvet, and Ph. Langlois. Compensated Horner scheme. Research Report 04, Équipe de recherche DALI, Laboratoire LP2A, Université de Perpignan Via Domitia, France, 52 avenue Paul Alduy, 66860 Perpignan cedex, France, July 2005.

[31] G.I. Hargreaves. Interval analysis in MATLAB. Numerical Analysis Report No. 416, Manchester Centre for Computational Mathematics, University of Manchester, December 2002. Available at http://www.maths.man.ac.uk/~nareports/narep416.pdf.

[32] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002.

[33] W. Hofschuster and W. Krämer. C-XSC 2.0: A C++ Library for Extended Scientific Computing. In *Numerical Software with Result Verification, Lecture Notes in Computer Science*, volume 2991/2004, pages 15–35. Springer-Verlag, Heidelberg, 2004.

[34] IEEE Computer Society, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.

[35] C.-P. Jeannerod and S. M. Rump. Improved error bounds for inner products in floating-point arithmetic. *SIAM J. Matrix Anal. Appl.*, 34(2):338–344, 2013.

[36] H. Jiang, S. Graillat, and R. Barrio. Accurate and fast evaluation of elementary symmetric functions. In *Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, TX, USA, April 7-10*, pages 183–190, 2013.

[37] H. Jiang, S. Graillat, C. Hu, S. Lia, X. Liao, L. Cheng, and F. Su. Accurate evaluation of the k-th derivative of a polynomial. *J. Comput. Appl. Math.*, 191:28–47, 2013.

[38] F. Jézéquel and J.-M. Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933–955, 2008.

[39] E. Kaltofen, B. Li, Z. Yang, and L. Zhi. Exact certification of global optimality of approximate factorizations via rationalizing sums-of-squares with floating point scalars. In *ISSAC*, pages 155–164, 2008.

[40] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, Reading, MA, USA, third edition, 1998.

[41] U. W. Kulisch. *Advanced Arithmetic for the Digital Computer*. Springer-Verlag, Wien, 2002.

[42] U. W. Kulisch. *Computer arithmetic and validity*, volume 33 of *de Gruyter Studies in Mathematics*. Walter de Gruyter & Co., Berlin, 2008. Theory, implementation, and applications.

[43] Ph. Langlois. Analyse d'erreur en précision finie. In A. Barraud, editor, *Outils d'analyse numérique pour l'Automatique*, Traité IC2, chapter 1, pages 19–52. Hermes Science, 2002.

[44] 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 Transactions on Mathematical Software*, 28(2):152–205, 2002.

[45] M. Mezzarobba. *Autour de l'évaluation numérique des fonctions D-finies*. Phd thesis, École polytechnique, november 2011.

[46] R.E. Moore. *Interval analysis*. Prentice Hall, 1966.

[47] R.E. Moore, R.B. Kearfott, and M.J. Cloud. *Introduction to interval analysis*. 2009.

[48] R. G. Mosier. Root neighborhoods of a polynomial. *Math. Comp.*, 47(175):265–273, 1986.

[49] J.-M. Muller. *Arithmétique des Ordinateurs*. Masson, 1989.

[50] 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 Inc., Boston, MA, 2010.

[51] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.

[52] M. Pichat and J. Vignes. The numerical study of chaotic systems - future and past. In *16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, Lausanne, Switzerland, August 2000.

[53] Rizwana Rehman and Ilse C. F. Ipsen. Computing characteristic polynomials from eigenvalues. *SIAM J. Matrix Anal. Appl.*, 32(1):90–114, 2011.

[54] N. Revol and F. Rouillier. Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library. *Reliable Computing*, 11(4):275–290, 2005.

[55] N. Revol and F. Rouillier. *MPFI (Multiple Precision Floating-point Interval library)*, 2009. Available at http://gforge.inria.fr/projects/mpfi.

[56] Cindy Rubio-Gonzalez, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of SC13 - The International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, November 17-22, 2013*. To appear.

[57] S. M. Rump. *Reliability in Computing. The Role of Interval Methods in Scientific Computing*. Academic Press, 1988.

[58] S. M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999.

[59] S. M. Rump. Computational error bounds for multiple or nearly multiple eigenvalues. *Linear Algebra Appl.*, 324(1-3):209–226, 2001. Special issue on linear algebra in self-validating methods.

[60] S. M. Rump. Ten methods to bound multiple roots of polynomials. *J. Comput. Appl. Math.*, 156(2):403–432, 2003.

[61] S. M. Rump. Computer-assisted proofs and self-validating methods. In B. Einarsson, editor, *Accuracy and Reliability in Scientific Computing*, Software-Environments-Tools, pages 195–240. SIAM, Philadelphia, PA, 2005.

[62] S. M. Rump. Verification methods: rigorous results using floating-point arithmetic. *Acta Numer.*, 19:287–449, 2010.

[63] S. M. Rump. Error estimation of floating-point summation and dot product. *BIT*, 52(1):201–220, 2012.

[64] S. M. Rump and S. Graillat. Verified error bounds for multiple roots of systems of nonlinear equations. *Numer. Algorithms*, 54(3):359–377, 2010.

[65] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM J. Sci. Comput.*, 31(1):189–224, 2008.

[66] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part II: Sign, k-fold faithful and rounding to nearest. *SIAM J. Sci. Comput.*, 31(2):1269–1302, 2008.

[67] S. R. Rump. Solving Algebraic Problems with High Accuracy. In Ulrich W. Kulisch and Willard L. Miranker, editors, *A new approach to scientific computation*, volume 7 of *Notes and Reports in Computer Science and Applied Mathematics*, pages 51–120, New York, 1983. Academic Press Inc.

[68] Pat H. Sterbenz. *Floating-point computation*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1974. Prentice-Hall Series in Automatic Computation.

[69] F. Tisseur. Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 22(4):1038–1057 (electronic), 2001.

[70] K.-C. Toh and L. N. Trefethen. Pseudozeros of polynomials and pseudospectra of companion matrices. *Numer. Math.*, 68(3):403–425, 1994.

[71] W. Tucker. The Lorenz attractor exists. *C. R. Acad. Sci. Paris Sér. I Math.*, 328(12):1197–1202, 1999.

[72] W. Tucker. Fundamentals of chaos. Kocarev, Ljupco (ed.) et al., Intelligent computing based on chaos. Berlin: Springer. Studies in Computational Intelligence 184, 1-23 (2009)., 2009.

[73] W. Tucker. *Validated numerics*. Princeton University Press, Princeton, NJ, 2011. A short introduction to rigorous computations.

[74] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Math. Comput. Simulation*, 35:233–261, 1993.

[75] J. Vignes. Discrete stochastic arithmetic for validating results of numerical software. *Num. Algo.*, 37(1–4):377–390, dec 2004.

[76] J. H. Wilkinson. Rounding errors in algebraic processes. (32), 1963. Also published by Prentice-Hall, Englewood Cliffs, New Jersey, USA. Reprinted by Dover, New York, 1994.

[77] J. H. Wilkinson. *Rounding errors in algebraic processes*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1963.

[78] L.-S. Yao. Computed chaos or numerical errors. *Nonlinear Analysis: Modelling and Control*, 15(1):109–126, 2010.

Résumé :

Dans ce manuscrit, nous présentons les recherches que nous avons effectuées depuis notre thèse. Nous commençons par faire un bref résumé de nos récentes contributions. Nous proposons ensuite un rapide état de l'art sur l'arithmétique des ordinateurs et sur les méthodes d'analyse des erreurs d'arrondi (méthode directe et inverse, analyse par intervalle et méthode stochastique). Nous présentons ensuite le coeur de notre recherche qui concerne l'amélioration de la précision des algorithmes. Pour cela, nous utilisons principalement les transformations exactes (algorithmes qui permettent d'estimer les erreurs d'arrondi) pour corriger ensuite le résultat final. Nous étudions aussi des méthodes permettant de prouver des résultats mathématiques sur ordinateur en précision finie. Pour cela nous combinons l'utilisation de résultats mathématiques d'existence (théorème de point fixe) et la fiabilité de l'arithmétique d'intervalle. Nous décrivons ensuite un outil qui permet la validation numérique de grands codes sans grandement modifier celui-ci. Finalement, nous esquissons un programme de recherche pour les années à venir.

Mots-clés :

Arithmétique des ordinateurs, arithmétique flottante, analyse d'erreur, arithmétique d'intervalle, arithmétique stochastique, méthodes auto-validantes, multiprécision.

Abstract:

This manuscript presents some of our work since the PhD thesis. We first present a brief description of our research and our recent results. We then provide an overview on computer arithmetic and on methods used to perform rounding error analysis (forward and backward error analysis, interval analysis and stochastic analysis). After that, we present the core of our research that is the increase of the accuracy of numerical algorithms. For that we use the error-free transformations (which make it possible to estimate the rounding errors) to correct the computed result. We then show how to prove mathematical results on computers in finite precision. For that, we combine mathematical theorems (fixed point theorems) and the reliability of interval computations. We also describe a tool that enable one to validate huge numerical codes with only few modifications of the program. Finally we sketch a research program for the next few years.

Keywords:

Computer arithmetic, rounding error analysis, interval arithmetic, floating-point arithmetic, stochastic arithmetic, self-validating methods, multiprecision.