A simple test qualifying the accuracy of Horner's rule for polynomials

Sylvie Boldo, Marc Daumas

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

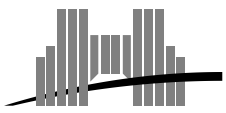
CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

A simple test qualifying the accuracy of Horner's rule for polynomials

Sylvie Boldo & Marc Daumas

Janvier 2003

Research Report N° 2003-01



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

INRIA

A simple test qualifying the accuracy of Horner's rule for polynomials

Sylvie Boldo & Marc Daumas

Janvier 2003

Abstract

Polynomials are used in many applications and hidden in libraries such as `libm`. Whereas the accuracy of the functions used by linear algebra have long been studied, little is available to decide on one scheme to evaluate a polynomial. Common knowledge solely emphasizes that Horner's rule is a good scheme unless the indeterminate is close to one of the polynomial's roots. We propose here a criterion for one step of Horner's scheme to be faithful. A result is defined to be faithful when it was correctly rounded whereas the rounding mode (up, down or to the nearest) cannot be known by the user. Our criterion is checked against the IEEE standard for floating point arithmetic using the Coq automatic proof checker. We then present three programs in Maple, Java and C that check the criterion for a whole polynomial associated with a domain for the indeterminate and a possible truncation error. An example of use is given with the approximation of elementary functions.

Keywords: Floating point, IEEE 754 Standard, Horner's rule, Formal proof, Coq.

Résumé

Les polynômes sont utilisés dans de nombreuses applications et enfouis dans des librairies telles que `libm`. Alors que la précision des fonctions utilisées en algèbre linéaire est étudiée depuis longtemps, on trouve peu d'aide pour décider d'un schéma d'évaluation polynomial. La culture commune reconnaît seulement que l'évaluation de Horner se comporte bien à moins que l'indéterminée ne soit proche d'une des racines du polynôme. Nous proposons ici un critère de fidélité pour une étape du schéma de Horner. Un résultat est défini comme fidèle s'il a été obtenu par un arrondi correct bien que l'utilisateur ne puisse pas savoir quel arrondi a été utilisé (vers le haut, le bas ou au plus proche). Notre critère a été validé vis à vis de la norme IEEE sur l'arithmétique à virgule flottante avec l'outil automatique Coq. Nous présentons ensuite trois programmes en Maple, Java et C qui vérifient notre critère sur un polynôme entier associé à un domaine pour l'indéterminée et une éventuelle erreur de troncature. Un exemple d'utilisation est donné avec l'approximation des fonctions élémentaires.

Mots-clés: Virgule flottante, Norme IEEE 754, Règle de Horner, Preuve formelle, Coq.

A simple test qualifying the accuracy of Horner's polynomial evaluation[1]

Sylvie Boldo & Marc Daumas
E-mail: Sylvie.Boldo@ENS-Lyon.Fr & Marc.Daumas@ENS-Lyon.Fr.

1 Introduction

It is sometimes interesting to open a few books and look at the answers given to one problem. Our problem is to evaluate a polynomial with the floating point arithmetic. The polynomial and the range of the indeterminate are both known in advance and plenty of precomputing can be performed off-line. The question is: Can we find a scheme where the result is an accurate approximation of the polynomial value for all the values of a given domain of the indeterminate?

Neither Higham [11], Epperson [8], Markstein [16], Muller [20], Pan [1] nor Press *et al.* [21] give an alternative method to Horner's rule for a better accuracy of the result. Higham bounds the forward error and proves that Horner's rule has a small relative backward error [11, p. 94-96]. The reader can also use the generic condition number of an univariate real function $|xf'(x)/f(x)|$ to check that the problem is simple unless x is close to one of the polynomial's roots [11, p. 8]. Higham later presents notes and references in a separate subsection [11, p. 102-104]. Knuth [15] presents an alternative scheme but he focuses his work on the number of operations (additions and multiplications).

The problem is not new and Miller proposed alternatives choices [17] but he was unable to present a simple criterion on the choice of one evaluation scheme. It seems that deciding the most accurate or even a very accurate evaluation scheme for an arbitrary polynomial and an arbitrary domain for the indeterminate is a difficult problem. However, authors have presented in the past some functions that use one polynomial evaluation and that are fairly accurate [18, 9, 25].

The common trick of the applications of the later authors is to use Horner's rule with a small indeterminate. As the rule unfolds in equation (1), the ongoing error is scaled down by the multiplication with the indeterminate. If the indeterminate is sufficiently small the final error is only slightly larger than the error of the last addition.

$$P(x) = a_0 + x \times (a_1 + x \times (a_2 + x \times (\ldots (a_{n-1} + x \times a_n) \ldots))) \tag{1}$$

Even with such a small bound on the error of the evaluation scheme, it is not possible to guarantee correct rounding to the nearest, that is:

$$\widetilde{P}(x) = \circ(P(x))$$

where $\widetilde{P}(x)$ is the result of the implemented Horner's rule and $\circ(y)$ is a real function that returns the floating point number nearest to y. One would possibly consider the relative distance between $\widetilde{P}(x)$ and $P(x)$ or more precisely consider the distance between $\widetilde{P}(x)$ and $P(x)$ relative to the weight of one unit in the last place (**ulp**) of $\widetilde{P}(x)$.

As we will see in Section 2 that such distances are unnecessarily loose, we define an implementation to be **faithful** if it returns either the rounded up or the rounded down value of the exact result [7, 3, 4]. We set the floating point unit to round all the basic operations to the nearest number although compound evaluations may return rounded up or down values of the exact results. We prove that requiring faithful rounding is more accurate than bounding the number of ulps of the error and this proof concludes that it is the best rounding scheme if correct rounding cannot be sustained.

We then present a tight sufficient condition to guarantee faithful rounding on one step of Horner's rule. Since the condition is a consequence of the ANSI-IEEE 754 standard for floating point arithmetic [24], it is validated using our specification of the generic floating point arithmetic using the Coq proof assistant [6]. The scripts of the proofs are available on the Internet and can be reviewed at the following address:

[1] This text is also available as a research report of the Institut National de Recherche en Informatique et en Automatique `http://www.inria.fr`.

`http://www.ens-lyon.fr/~sboldo/coq/Axpy.html`.

We indicate the name of the theorem used in the `Axpy` file.

In Section 3, we present three programs in Maple, Java and C that compute an upper bound and an absolute error bound on an arbitrary polynomial and an arbitrary domain for the indeterminate. Both programs in Java and C are used when the target floating point precision is the available machine precision. The C code is much simpler as both floating point correct rounding to the nearest and directed roundings are native. Unfortunately, Java does not implement the directed roundings [14] and these modes are loosely simulated on the Java program whereas the C program dynamically changes the rounding mode to obtain the best result with a simple program.

As the Maple, C and Java programs compute the two bounds on Horner's rule, these bounds are used to check the polynomial against the criterion defined in Section 2. If the implemented polynomial is an approximation to another function such as an elementary function, the programs add the truncation error before testing our criterion. If the criterion is satisfied, the program is able to guarantee the faithful evaluation of the polynomial or faithful evaluation of the approximated function over the whole domain of the indeterminate specified by the user. The programs can be downloaded at

`http://www.ens-lyon.fr/~daumas/SoftArith/index.html.fr#elem`.

We finish this paper (Section 4) by some concluding remarks and two examples where our programs guarantee a faithful evaluation. The first one is used for the approximation to the elementary functions [5]. The second one is due to Fike [9]. We answer questions relative to Horner's rule modified for a pipelined processor and we raise a new question relative to the argument reduction step.

2 Faithful multiple and accumulate operation guaranteed with Coq

Some processors, such as Intel's recent IA64, integrate a fused multiply and accumulate operator to enhance both speed and precision as the result of the atomic floating point operation $ax + y$ only incurs one final rounding [16]. We show that it is feasible to obtain a faithful result under mild useful assumptions using only the IEEE standard addition and multiplication to implement

$$\circ(\circ(a \times x) + y).$$

Either with or without this fused multiply and accumulate, the result may still be faithful when inputs a_0, x_0 and y_0 are not known exactly. We prove in Sections 2.3 and 2.4 that

$$\circ(\circ(a \times x) + y) \quad \text{and} \quad \circ(a \times x + y)$$

are faithful roundings of $a_0 \times x_0 + y_0$ when

$$5\,\frac{2+\mathrm{ulp}}{2-\mathrm{ulp}}\left(|a \times x| + \frac{\lambda \times \mathrm{ulp}}{2}\right) \leq |y|$$

and

$$|y_0 - y| + |a_0 \times x_0 - a \times x| \leq \frac{\mathrm{ulp}}{8}\left((1-\mathrm{ulp}) \times |y| - |a \times x| - 2 \times \lambda\right)$$

where λ is the smallest normal positive number and ulp is the weight of one unit in the last place of the representation of 1. For sake of simplicity, we ask that $\mathrm{ulp} \leq 2^{-3}$. That condition will be met by every reasonable hardware implementation.

We have used the Coq automatic proof checker and our radix and rounding generic library of definitions and properties on floating point arithmetic. Yet, this work uses radix $\beta = 2$ and the rounding to nearest with no specific tie breaking rule.

2.1 Properties of our generic specification of floating point arithmetic in Coq

Numbers are represented with pairs (n, e) that stand for $n\beta^e$ where β is the radix of the floating point number system. In our characterization, we use both an integral signed mantissa n and an integral signed amplitude e for the sake of simplicity. The above definition is not sufficient to identify one unique pair (n, e) for a represented quantity. We add a normalization convention whereby the p-digit magnitude of the mantissa of the **normal** representation of a number is required to start with a non zero digit. The underflow amplitude, a constant, is the lowest amplitude $-e_{\min}$ available.

We define a **bounded** pair (n, e) such that $|n| < \beta^p$ and $e \geq -e_{\min}$. We do not set an upper bound on the amplitude as overflow is easily detected as the result will not be a number by a signed infinity or a . A bounded pair is normal if $\beta \cdot |n| \geq \beta^p$ and it is **subnormal** if $\beta \cdot |n| < \beta^p$ and $e = -e_{\min}$. With this formalism, the smallest bounded positive number is $\lambda_d = \beta^{-e_{\min}}$, one unit in the last place of 1 is $\mathrm{ulp} = \beta^{1-p}$ and the smallest normal positive number is $\lambda = \lambda_d / \mathrm{ulp}$. Each represented number has one unique representation either normal or subnormal. A pair is **canonical** if it is either normal or subnormal.

This formalism was introduced in [6] for our development using the Coq proof environment [12]. Other formalisms of the floating point arithmetic are in use with PVS [19, 13], HOL [2, 10] or ACL2 [22]. Using Curry Howard isomorphism, Coq and HOL rely on a very small inference engine to check the correctness of the proofs. Although Coq and HOL lack many of the automatic techniques implemented in PVS or ACL, they allow the user to safely explore the properties of a system.

We now give three new lemmas that will simplify our proofs in following sections.

Lemma 1 (FulpLeGeneral) *For any bounded pair r,*

$$\mathrm{ulp}(r) \leq (|r| + \lambda) \times \mathrm{ulp}\,.$$

This is easy to prove since $\mathrm{ulp}(r) \leq |r| \times \beta^{1-p}$ if the canonical representation of r is normal and $\mathrm{ulp}(r) = \beta^{-e_{min}}$ if the canonical representation of r is subnormal.

Lemma 2 (RoundLeGeneral and RoundGeGeneral) *Let $v \in \mathbb{R}$ and $r = \circ(v)$,*

$$\frac{2}{2+\mathrm{ulp}} \left(|v| + \frac{\lambda \times \mathrm{ulp}}{2}\right) \leq |r| \leq \frac{2}{2-\mathrm{ulp}} \left(|v| + \frac{\lambda \times \mathrm{ulp}}{2}\right).$$

As v is rounded to the nearest, we know that $|v - r| \leq \mathrm{ulp}(r)/2$. Conclusion follows from this and the previous result on $\mathrm{ulp}(r)$.

We will also need some results about the predecessor r^- of a floating point pair r. The **predecessor** r^- of r is the largest bounded pair smaller than r. Its value can easily be deduced from the mantissa n_r and the amplitude e_r of r if r is canonical. We can also define in the same way the **successor** r^+ of r as the bounded float just greater than r. The inequality $r^- < r < r^+$ holds by construction. We also prove the following lemma.

Lemma 3 (FpredUlpPos, FulpFPredGePos and FulpFPredLe) *If $r > 0$ then*

$$r^- + \mathrm{ulp}\left(r^-\right) = r \quad \textit{and} \quad \mathrm{ulp}\left(r^-\right) \leq \mathrm{ulp}(r) \leq 2\ \mathrm{ulp}\left(r^-\right).$$

2.2 Definition and properties of faithful rounding

Most available general purpose processors have long been compliant with the IEEE 754 standard on floating point arithmetic [23]. It means that they implement precise rounding for the four arithmetic operations (addition, multiplication, division and square root). The result obtained for any of these computer operations is the one given by applying a user-chosen rounding function on the result of the exact mathematical operation. The standard specifies four rounding functions: rounding to the nearest with the even tie breaking rule, rounding up, down or towards 0.

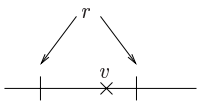


Figure 1: Two possible outputs r of a faithful implementation compared to the exact value v

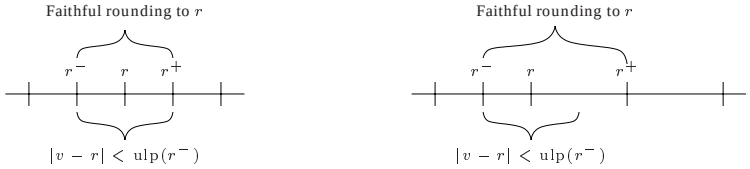


Figure 2: A first condition to conclude on faithful rounding (see Lemma 4)

An operator implements faithful rounding if the result is either the rounded up or the rounded down value of the exact result such as presented in Figure 1. We consider that it is a non deterministic choice and the user has no ability to select the rounding mode *a priori* or to know which rounding mode was used *a posteriori*.

Theorem 1 shows that faithful rounding is the tightest non correct condition for floating point arithmetic. It is more precise than allowing an error strictly less than one unit in the last place as precision wobbles near the exact powers of the floating point radix.

Theorem 1 (MinOrMax_Rlt) *Let $v \in \mathbb{R}$ and r faithful rounding of v,*

$$|v - r| < \mathrm{ulp}(r).$$

The condition of the preceding theorem is not sufficient to ensure faithful rounding. More precisely, when r is far from any end of the binades, faithful rounding is equivalent to $|v - r| < \mathrm{ulp}(r)$ but when r is close to the beginning or to the end of a binade, this is not true anymore.

We present here some lemmas that are used in the proofs of Section 2.3. The first lemma handles most cases while the other one is used for the remaining cases.

Lemma 4 (MinOrMax1) *If $r > 0$ and*

$$|v - r| < \mathrm{ulp}\left(r^{-}\right)$$

then r is a faithful rounding of v (see Figure 2).

Lemma 5 (MinOrMax2) *If $r > 0$ and*

$$|v - r| < \mathrm{ulp}(r) \quad \textit{and} \quad r \leq v$$

then r is a faithful rounding of v (see Figure 3).

To handle negative and null cases, we define r^* as r^- if $r \geq 0$ and r^+ if $r < 0$. It means r^* is the bounded float just near r, towards zero. It can be viewed as the rounded towards zero of the real value $r - \varepsilon$, when ε has the sign of r and it is much smaller than any represented floating point number. With this definition $0^* = \beta^{-e_{\min}}$.

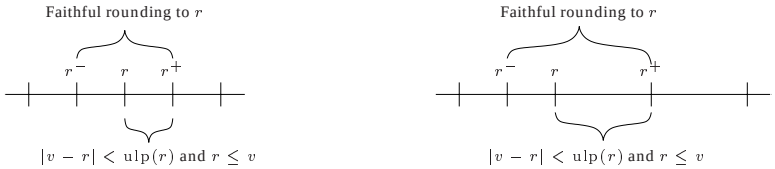


Figure 3: A second condition to conclude on faithful rounding (see Lemma 5)

2.3 Sufficient conditions for a faithful multiply and accumulate

We now give the main theorems of Section 2. Theorem 2, based on Lemma 6, uses the values of the intermediate variable and the value of the result. It yields the tightest condition but the condition cannot be easily tested *a priori*. Theorem 3 is presented later based on the value of the inputs.

Lemma 6 (AxpyPos) *Given real numbers a_0, x_0 and y_0, and bounded floating-point numbers a, x and y, we define bounded floating-point numbers t and u such that*

$$\begin{aligned} t &= \circ(a \times x), \\ u &= \circ(t + y). \end{aligned}$$

For $u > 0$, if no overflow occurs and if

$$4|t| \le |u| \quad \textit{and} \quad |y_0 - y| + |a_0 \times x_0 - a \times x| < \frac{\mathrm{ulp}\,(u^-)}{4}$$

then u is a faithful approximation to $a_0 \times x_0 + y_0$.

We set the bound

$$\begin{aligned} |u - (a_0 \times x_0 + y_0)| &= |u - (t+y) + t - a \times x + y - y_0 + a \times x - a_0 \times x_0| \\ &< |u - (t+y)| + \frac{\mathrm{ulp}(t)}{2} + \frac{\mathrm{ulp}\,(u^-)}{4} \end{aligned}$$

and we separate two cases. If $t + y \le u$, then the rounding of u is upwards, and $t + y$ is nearer to u than to u^-. We conclude that

$$|u - (t+y)| \le \frac{\mathrm{ulp}\,(u^-)}{2}.$$

As $4|t| \le |u|$, in most cases, we have

$$4\,\mathrm{ulp}(t) \le \mathrm{ulp}(u) \le 2\,\mathrm{ulp}\left(u^-\right).$$

The special cases are handled and guaranteed independently with Coq.

From Lemma 3, we deduce that $|u - (a_0 \times x_0 + y_0)| < \mathrm{ulp}\,(u^-)$ and we can conclude by Lemma 4.

Otherwise, $t + y \ge u$ and

$$|u - (a_0 \times x_0 + y_0)| < \frac{\mathrm{ulp}(u)}{2} + \frac{\mathrm{ulp}(u)}{8} + \frac{\mathrm{ulp}(u)}{4} < \mathrm{ulp}(u).$$

We conclude by Lemma 5.

Theorem 2 (Axpy_tFlessu) *Given real numbers a_0, x_0 and y_0, and bounded floating-point numbers a, x and y, we define bounded floating-point numbers t and u such that*

$$\begin{aligned} t &= \circ(a \times x), \\ u &= \circ(t + y). \end{aligned}$$

If no overflow occurs and if

$$4|t| \leq |u| \quad \textit{and} \quad |y_0 - y| + |a_0 \times x_0 - a \times x| < \frac{\mathrm{ulp}\,(u^*)}{4}$$

then u is a faithful approximation to $a_0 \times x_0 + y_0$.

The theorem is exactly Lemma 6 when $u > 0$. When $u < 0$, we use the same lemma but we change the inputs. We use $-a$ in place of a, and $-y$ in place of y. As both rounding to the nearest and faithful rounding are stable through the opposite, we guarantee the correctness of this assertion.

We have to handle $u = 0$ separately. As $4|t| \leq |u|$, this means that $t = 0$. This also means that $t + y = 0$ as t and y are bounded floating point numbers. So $u = t = y = 0$ and

$$|u - (a_0 \times x_0 + y_0)| = |a_0 \times x_0 + y_0| \leq |a \times x - t| + |a \times x - a_0 \times x_0| + |y_0 - y| < \frac{\mathrm{ulp}(t)}{2} + \frac{\mathrm{ulp}\,(u^*)}{4}.$$

As $t = u = 0$, we deduce that $\mathrm{ulp}(t) = \mathrm{ulp}(u^-) = \beta^{-e_{\min}}$. It follows that $|u - (a_0 \times x_0 + y)| < \mathrm{ulp}(u^-)$ and the result follows from a proof analogous to Lemma 4.

The preceding theorem depends on the values of u, u^* and t. We prefer to have a result depending only on the values of the inputs. From previous results and with some long computations we get the Theorem 3.

Theorem 3 (Axpy_opt) *Given real numbers a_0, x_0 and y_0, and bounded floating-point numbers a, x and y, if no overflow occurs and if*

$$5\,\frac{2 + \mathrm{ulp}}{2 - \mathrm{ulp}} \times \left(|a \times x| + \frac{\lambda \times \mathrm{ulp}}{2}\right) \leq |y|$$

and

$$|y_0 - y| + |a_0 \times x_0 - a \times x| \leq \frac{\mathrm{ulp}}{8}\left((1 - \mathrm{ulp}) \times |y| - |a \times x| - 2 \times \lambda\right),$$

then $\circ(\circ(a \times x) + y)$ is a faithful approximation to $a_0 \times x_0 + y_0$.

This theorem is a direct consequence of the previous theorems and of Lemmas 1 and 2 to bound the unknown values from the known ones, namely t, u and u^* from a, x and y.

Since we assumed in the beginning of Section 2 that the field used for the floating point mantissa is at least 4 bit long ($\mathrm{ulp} \leq 2^{-3}$), we can simplify the first condition and get Corollary 1 or we can simplify both conditions to get Corollary 2.

Corollary 1 (Axpy_Simpl1) *Given real numbers a_0, x_0 and y_0, and bounded floating-point numbers a, x and y, if no overflow occurs and if*

$$6\,\left(|a \times x| + \frac{\lambda \times \mathrm{ulp}}{2}\right) \leq |y|$$

and

$$|y_0 - y| + |a_0 \times x_0 - a \times x| \leq \frac{\mathrm{ulp}}{8}\left((1 - \mathrm{ulp}) \times |y| - |a \times x| - 2 \times \lambda\right),$$

then $\circ(\circ(a \times x) + y)$ is a faithful approximation to $a_0 \times x_0 + y_0$.

Corollary 2 (Axpy_Simpl2) *Given real numbers a_0, x_0 and y_0, and bounded floating-point numbers a, x and y, if no overflow occurs and if*

$$6\,\left(|a \times x| + \frac{\lambda \times \mathrm{ulp}}{2}\right) \leq |y|$$

and

$$|y_0 - y| + |a_0 \times x_0 - a \times x| \leq \frac{\mathrm{ulp}}{12}\left(|y| - 3 \times \lambda\right),$$

then $\circ(\circ(a \times x) + y)$ is a faithful approximation to $a_0 \times x_0 + y_0$.

2.4 Using a hardware fused multiply and accumulate operation

The theorems of Section 2.3 can be strengthened if we use a fused multiply and accumulate that only rounds once instead of separately rounding multiplications and additions.

Lemma 7 (Axpy_FLessu_Fmac) *Given real numbers a_0, x_0 and y_0, and bounded floating-point numbers a, x and y, we define the floating point number*

$$u = \circ(a \times x + y).$$

If no overflow occurs and if

$$|y_0 - y| + |a_0 \times x_0 - a \times x| < \frac{\mathrm{ulp}\,(u^*)}{2},$$

then u is a faithful approximation to $a_0 \times x_0 + y_0$.

We deduce Theorem 4 from Lemma 7 with a path similar to the one used in the previous section.

Theorem 4 (Axpy_opt_Fmac) *Given real numbers a_0, x_0 and y_0, and bounded floating-point numbers a, x and y, if no overflow occurs and if*

$$|y_0 - y| + |a_0 \times x_0 - a \times x| \leq \frac{\mathrm{ulp}}{4}\left((1 - \mathrm{ulp}) \times |a \times x + y| - \frac{6 \times \lambda \times \mathrm{ulp}}{4 - \mathrm{ulp}^2}\right),$$

then $\circ(a \times x + y)$ is a faithful approximation to $a_0 \times x_0 + y_0$.

There are two ways to use Theorem 4. One first mean is to recognize that the condition of Theorem 4 is a consequence of the conditions of Theorem 3. Thereafter, we can check only the later conditions and obtain a function that will faithfully run on both type of implementations with or without a fused multiply and accumulate operation. We can also check the easier condition of Theorem 4 to validate an implementation for hardware that use a fused multiply and accumulate operation.

3 Qualifying Horner's rule on an approximated polynomial

The criterions defined in the theorems of Section 2.3 can be used to test automatically the faithfulness of Horner's rule on a polynomial or an approximated polynomial implementation of a function. We have written Maple, Java and C programs that do so. They are available through the Internet under the GNU Lesser General Public License version 2.1 as published by the Free Software Foundation.

3.1 Supporting functions in Maple

We define functions to mimic the IEEE standard floating point arithmetic using the exact multiple precision rational arithmetic of Maple. Most of them use two parameters: `lgfr`, `lgex` that are the length of the fraction field and the length of the exponent field in the IEEE standard representation. These lengths are $(23, 8)$ for single precision, $(52, 11)$ for double precision and $(63, 15)$ for PC double extended precision.

The functions `MinExpIEEE(lgex)` and `MaxExpIEEE(lgex)` return the smallest and the highest allowed exponents. Function `UlpCstIEEE(lgfr)` computes the value of one unit in the last place of the representation of the number 1. Function `LambdaIEEE(lgex)` yields the smallest normalised positive number.

Function `ExpIEEE(x, lgex)` computes the unbiased exponent associated with the number x. It uses the approximated floating point logarithm available in Maple and two tests have been added to ensure a correct behavior in all cases. Function `UlpIEEE(x, lgfr, lgex)` computes the value of one unit in the last place of x. Function `BiasedIEEE(x, lgfr, lgex)` rounds x to the nearest floating point number. Biased rounding is used in place of the even tie breaking rule to yield a simple and safe

overestimation. This function can be applied to one number or to an array or a polynomial expression. Function `UpIEEE(x, lgfr, lgex)` rounds x up. The result is the signed infinity if x overflows the maximum exponent associated with `lgex`.

For all the following functions the polynomial coefficients are supposed to be exact. The user may use `BiasedIEEE(P, lgfr, lgex)` where `P` is a polynomial expression to get a rounded polynomial.

3.2 Usual bounds for Horner's rule in Maple

The `HornerBounds(P, XMax, relround)` and `HornerIEEE(P, XMax, lgfr, lgex)` recursive functions were implemented as a reference. They can also be used in qualifying the accuracy of Horner's rule for polynomials although we do not provide a example in Maple. The input is a number `XMax` defining the range `[-XMax..XMax]` for the indeterminate. The outputs are a bound `PMax` on the polynomial value and a bound `AbsError` on the absolute error of the polynomial evaluation.

Function `HornerBounds` uses the relative error bound of the so-called standard model [11, p. 40]. Function `HornerIEEE` is a more precise program that mimics closely the IEEE standard behavior.

Given the polynomial

$$P(x) = a_0 + x \times (a_1 + x \times (a_2 + x \times (\ldots (a_{n-1} + x \times a_n) \ldots)))$$

we use any of the two functions `HornerBounds` or `HornerIEEE` to bound both the evaluation of

$$P_1(x) = a_1 + x \times (a_2 + x \times (\ldots (a_{n-1} + x \times a_n) \ldots))$$

and the absolute error in the evaluation of $P_1(x)$ with Horner's rule. The criterion is checked with `PMax` and `AbsError` to decide whether the evaluation of $P(x)$ is faithful.

3.3 Guaranteed faithful result of Horner's rule in Maple

The `HornerAXPY(P, XMax, Err0, Err1, ErrX)` function, whose code is given below, is the main function of our Maple library. It tests if Horner's rule applied to the polynomial `P` with the indeterminate in a subrange S of `[-XMax..XMax]` yields a faithful evaluation of the function f provided $f(0) - P(0)$ is bounded by `Err0`, $(f(x) - P(x) - f(0) + P(0))/x$ is bounded by `Err1` on S and a possible error on x is bounded by `ErrX`.

As these quantities are available, the function produces a bound $PMax$ on the polynomial value and a bound $AbsError$ on the absolute error of the polynomial evaluation. The last output is a binary encoded certificate. If the certificate is strictly less than 1, the final result is faithful. If the certificate is 0, all the recurrent function calls were faithful.

The function uses predicates `AXPYCond1(a, x, y)` and `AXPYCond2(a, err_a, x, err_x, y, err_y)` defined from Section 2.3. The predicate `AXPYFCond2(a, err_a, x, err_x, y, err_y)` is a substitute condition defined in Theorem 4. It can be used when the hardware provides a fused multiply and accumulate operation. Some parameters have been omitted in the function prototypes to get a more readable code.

```
HornerAXPY := proc(P, XMax, Err0, Err1, ErrX)
  local rec_max, erreur, rec_erreur, check, rec_check, PMax, SMax;
  if degree(P) = 0 then abs(P), Err0, 0;
  else
    rec_max, rec_erreur, rec_check :=
      HornerAXPY (HornerStep(P), XMax, Err1, 0, ErrX);
    PMax := BiasedIEEE (rec_max * XMax);
    SMax := BiasedIEEE (PMax + abs(eval(P, x = 0)));
    if  AXPYCond1
      (rec_max,             XMax,       abs(eval(P, x = 0))      )
    and AXPYCond2
```

```
    (rec_max, rec_erreur, XMax, ErrX, abs(eval(P, x = 0)), Err0)
  then check := 0; else check := 1; fi;
  SMax,   rec_erreur * XMax + rec_max * ErrX + Err0
        + (UlpIEEE(PMax) + UlpIEEE(SMax)) / 2,
  check + rec_check / 2 ;
fi; end:
```

3.4 Guaranteed faithful result of Horner's rule in C and Java

The C and Java programs are not presented in this text. Some care is needed to handle a polynomial as a list of coefficients stored in the global `Coefficients` array and the `rec_max`, `rec_erreur` and `rec_check` global variables are used to avoid returning three results to a function call.

Compared to the Maple code, the user defined `BiasedIEEE` function is nicely replaced by the hardware implementation on the native floating point type. On the other end, the exact rational computations on the error bounds are now performed using the floating point unit. For the C implementation we use the `fpgetround` and `fpsetround` functions to dynamically set the active rounding mode to positive infinity or to restore the rounding mode to its default value. For the Java implementation, we have changed the formulas by adding ulps in many places to guarantee a sufficient condition.

4 Example of use and concluding remarks

We have just presented fairly tight conditions to get a faithful implementation of a polynomial evaluation with Horner's rule. That means that the result is guaranteed to be almost correctly rounded by just running a little Maple, Java or C program. The question remaining is to know whether or not these conditions do occur in applications. We will see two examples.

4.1 Example 1: Polynomial approximation to the exponential

In this example, we approximate $exp(x)$ to sufficient accuracy using Chebyshev's orthonormal basis and the least squares projection with x in the interval $[-2^{-4}, 2^{-4}]$. The polynomial is forced to begin with $1 + x + x^2/2$. The result is rounded and the parameters are defined to use IEEE double precision arithmetic.

$$P(x) = 1 + x + \frac{1}{2}\,x^2 + \frac{6004799503158175}{36028797018963968}\,x^3 + \frac{6004799503152767}{144115188075855872}\,x^4$$

$$+ \frac{4803839629518891}{576460752303423488}\,x^5 + \frac{1601279914265145}{1152921504606846976}\,x^6$$

$$+ \frac{3660108472028231}{18446744073709551616}\,x^7 + \frac{7318367436494265}{29514790517935282 5856}\,x^8$$

The truncation error $(exp(x) - P(x))/x$ is bounded by

$$\frac{5509901405496691}{81129638414606681695789005144064}.$$

Examples of function calls to `HornerBounds` and `HornerIEEE` are presented in Maple and various quantities are computed. Finally the function `HornerAXPY` is called. This same example is also tested in the Java and the C programs. We conclude from the tests that Horner's rule applied to this polynomial yields a faithful approximation to the exponential over the range $[-2^{-4}, 2^{-4}]$.

As it is easy to modify the program in Maple, we have checked that this method yields a faithful polynomial over the range $[-2^{-3}, 2^{-3}]$. The criterion fails over the range $[-2^{-2}, 2^{-2}]$. This failure is not

surprising as the first condition of Theorem 3 roughly forces $5|a \times x| \leq |y|$ and the later indeterminate range yields roughly to $4|a \times x| \approx |y|$ at the interval bounds. We have also tested that, using a fused multiply and accumulate operator, the method yields a faithful approximation over the range $[-2^{-2}, 2^{-2}]$. Since our criterion is very simple we are able to explore many different solutions for hardware and software.

The Coq theorems proved in Section 2 can be applied to build faithful approximations to the elementary functions over the full range of the floating point input. We have tested in the remaining of the specific Maple subsection available on the Internet the exponential function with the full range of the input. We used a technique similar to the one first used by Wong and Goto [26] and assumed range reduction targeting to an error less than half an ulp of the maximum reduced argument with a total of 7 polynomials. Stronger range reduction consistently provides faithfull approximations with smaller polynomials although it uses tables with more entries.

4.2 Example 2: A polynomial from Fike

This example is due to Fike. The following polynomial appeared in [9] and is used as a single precision approximation to 2^x over the range $[-1/16, 0]$.

$$1 + .6931471805599346\,x + .2402265069563678\,x^2 + .05550410840231345\,x^3 \\ + .009618117095313700\,x^4 + .001333073417706260\,x^5 + .0001507368551403575\,x^6$$

Actually the implemented polynomial follows after rounding the coefficients to IEEE single precision.

$$P(x) = 1 + \frac{1453635}{2097152}\,x + \frac{1007583}{4194304}\,x^2 + \frac{14899271}{268435456}\,x^3 + \frac{10327375}{1073741824}\,x^4$$

$$+ \frac{11451013}{8589934592}\,x^5 + \frac{5179279}{34359738368}\,x^6$$

The truncation error $(2^x - P(x))/x$ is bounded by

$$\frac{8577801}{4503599627370496}$$

and a function call in Maple to `HornerAXPY` guarantees that the approximation is faithful over $[-1/16, 0]$.

4.3 Concluding remarks and perspectives

The presented criterion on a faithful multiply and accumulate operation is a tool to deduce a relative error bound from an absolute bound. It is powerful for two reasons. First, obtaining an absolute error bound on Horner's rule is simple and methods are presented in textbooks. Second, this technique works as soon as the indeterminate is small enough, a situation that occurs in many implementations.

Another example of use lies in an extended Horner's rule where the loop has been unrolled once. The polynomial P is uniquely replaced by the two polynomials Q and R such that

$$P(x) = P(0) + x \times Q\left(x^2\right) + x^2 \times R\left(x^2\right).$$

On a pipelined or superscalar processor, polynomials Q and R can be evaluated with Horner's rule in the same time needed to evaluate Q alone. This technique almost divides by two the time to evaluate P compared usual Horner's rule.

We have tested unrolled Horner's rule with the polynomial of the first example. The criterion is tested in `DoubleAXPY(P, XMax, Err0, Err1, ErrX)` function to conclude that this Horner's pipeline-oriented rule yields a faithful approximation to the exponential over $[-2^{-4}, 2^{-4}]$.

Although we have restricted to $\beta = 2$ for this work, careful reading of the proofs allows us to believe that this work is true whatever the actual radix value. Similar theorems (with slightly different bounds) could be proved whatever the radix.

Finally, this work is another attempt to enhance dependability through automatic proof checking. On one hand, we have proved and checked difficult results related to floating point arithmetic. On the other hand, we have trustfully used pen and paper proofs on common mathematical objects. We did also trust our ability to write small pieces of correct software manipulating simple data. On contrary, a full check from scratch of the correction of the C procedure used to implement the exponential function would be long and tedious and it would never let us explore trade-offs as we did with this work.

References

[1] Dario Bini and Victor Y. Pan. *Polynomial and Matrix Computations: Fundamental Algorithms*. Birkhäuser, 1994.

[2] Victor A. Carreño. Interpretation of IEEE-854 floating-point standard and definition in the HOL system. Technical Report Technical Memorandum 110189, NASA Langley Research Center, 1995.

[3] Marc Daumas. Basis for the implementation of a reliable dot product. Master's thesis, Southern Methodist University, Dallas, Texas, 1992.

[4] Marc Daumas and David W. Matula. Validated roundings of dot products by sticky accumulation. *IEEE Transactions on Computers*, 46(5):623–629, 1997.

[5] Marc Daumas and Claire Moreau-Finot. Exponential: implementation trade-offs for hundred bit precision. In *Real Numbers and Computers*, pages 61–74, Dagstuhl, Germany, 2000.

[6] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001.

[7] Theodorus J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[8] James F. Epperson. *An Introduction to Numerical Methods and Analysis*. John Wiley & Sons, 2001.

[9] C. T. Fike. Methods of evaluating polynomial approximations in function evaluation routines. *Communications of the ACM*, 10(3):175–178, 1967.

[10] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *12th International Conference on Theorem Proving in Higher Order Logics*, pages 113–130, Nice, France, 1999.

[11] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002. Second edition.

[12] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant: a tutorial: version 7.2. Technical Report 256, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France, 2002.

[13] Christian Jacobi. Formal verification of a theory of IEEE rounding. In *14th International Conference on Theorem Proving in Higher Order Logics*, pages 239–254, Edinburgh, Scotland, 2001. Supplemental Proceedings.

[14] William Kahan and Joseph D. Darcy. How java's floating-point hurts everyone everywhere. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, page 81, Palo Alto, California, 1998.

[15] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1997. Third edition.

[16] Peter Markstein. *IA-64 and elementary functions: speed and precision*. Prentice Hall, 2000.

[17] Webb Miller. Computational complexity and numerical stability. In *Proceedings of the 6th ACM Symposium of the Theory Of Computation*, pages 317–322, Seattle, Washington, 1974.

[18] Webb Miller. Software for roundoff analysis. *ACM Transactions on Mathematical Software*, 1(2):108–128, 1975.

[19] Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Report Technical Memorandum 110167, NASA Langley Research Center, 1995.

[20] Jean-Michel Muller. *Elementary functions, algorithms and implementation*. Birkhauser, 1997.

[21] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes*. Cambridge University Press, 1989.

[22] David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[23] N. L. Schryer. A test of computer's floating-point arithmetic unit. Technical report 89, AT&T Bell Laboratories, 1981.

[24] David Stevenson et al. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.

[25] Ping Tak Peter Tang. Table-lookup algorithms for elementary functions and their error analysis. In Peter Kornerup and David Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, 1991.

[26] Weng Fai Wong and Eiichi Goto. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, 1994.

Module Axpy

```
Require Export ClosestProp.
Require Export ClosestPlus.
Require Export FminOp.
Section AxpyMisc.

Local radix := (POS (xO xH)).

Local FtoRradix := (FtoR radix).
Coercion FtoRradix : float >→ R.
Variable b : Fbound.
Variable precision : nat.
Hypothesis precisionGreaterThanOne : (lt (S O) precision).
Hypothesis pGivesBound : (POS (vNum b)) = (Zpower_nat radix precision).

Theorem TwoMoreThanOne: (Zlt (POS xH) radix).
Hints Resolve TwoMoreThanOne .

Theorem TwoMoreThanOneR: (Rlt R1 radix).
Hints Resolve TwoMoreThanOneR .

Theorem FulpLeGeneral:
 (p : float)
 (Fbounded b p) →
  (Rle
    (Fulp b radix precision p)
    (Rplus
      (Rmult (Rabsolu (FtoRradix p)) (powerRZ radix (Zs (Zopp precision))))
      (powerRZ radix (Zopp (dExp b))))).

Theorem RoundLeGeneral:
 (p : float)
 (z : R)
 (Fbounded b p) →
 (Closest b radix z p) →
  (Rle
    (Rabsolu p)
    (Rplus
      (Rmult (Rabsolu z) (Rinv (Rminus R1 (powerRZ radix (Zopp precision)))))
      (Rmult
        (powerRZ radix (Zpred (Zopp (dExp b))))
        (Rinv (Rminus R1 (powerRZ radix (Zopp precision))))))).

Theorem RoundGeGeneral:
 (p : float)
 (z : R)
 (Fbounded b p) →
 (Closest b radix z p) →
  (Rle
    (Rminus
      (Rmult (Rabsolu z) (Rinv (Rplus R1 (powerRZ radix (Zopp precision)))))
      (Rmult
```

```
    (powerRZ radix (Zpred (Zopp (dExp b))))
    (Rinv (Rplus R1 (powerRZ radix (Zopp precision)))))) (Rabsolu p)).

Theorem FpredUlpPos:
 (x : float)
 (Fcanonic radix b x) →
 (Rlt R0 x) →
  (Rplus
   (FPred b radix precision x)
   (Fulp b radix precision (FPred b radix precision x))) == x.

Theorem LeFulpPos:
 (x, y : float)
 (Fbounded b x) →
 (Fbounded b y) →
 (Rle R0 x) →
 (Rle x y) → (Rle (Fulp b radix precision x) (Fulp b radix precision y)).

Theorem FulpFPredLe:
 (f : float)
 (Fbounded b f) →
 (Fcanonic radix b f) →
  (Rle
   (Fulp b radix precision f)
   (Rmult (S (S O)) (Fulp b radix precision (FPred b radix precision f)))).

Theorem FulpFPredGePos:
 (f : float)
 (Fbounded b f) →
 (Fcanonic radix b f) →
 (Rlt R0 f) →
  (Rle
   (Fulp b radix precision (FPred b radix precision f))
   (Fulp b radix precision f)).

Theorem FulpFabs:
 (f : float)
  <R> (Fulp b radix precision f) == (Fulp b radix precision (Fabs f)).

Theorem ExactSum_Near:
 (p, q, f : float)
 (Fbounded b p) →
 (Fbounded b q) →
 (Fbounded b f) →
 (Closest b radix (Rplus p q) f) →
 (Fexp p) = (Zopp (dExp b)) →
 (Rlt (Rabsolu (Rminus (Rplus p q) f)) (powerRZ radix (Zopp (dExp b)))) →
  <R> (Rplus p q) == f.
End AxpyMisc.
Section AxpyAux.

Local radix := (POS (xO xH)).
```

```
Local FtoRradix := (FtoR radix).
Coercion FtoRradix : float >→ R.
Variable b : Fbound.
Variable precision : nat.
Hypothesis precisionGreaterThanOne : (lt (S O) precision).
Hypothesis pGivesBound : (POS (vNum b)) = (Zpower_nat radix precision).
Variables a1, x1, y1 : R.
Variables a, x, y, t, u, r : float.
Hypothesis Fa : (Fbounded b a).
Hypothesis Fx : (Fbounded b x).
Hypothesis Fy : (Fbounded b y).
Hypothesis Ft : (Fbounded b t).
Hypothesis Fu : (Fbounded b u).
Hypothesis tDef : (Closest b radix (Rmult a x) t).
Hypothesis uDef : (Closest b radix (Rplus t y) u).
Hypothesis rDef : (isMin b radix (Rplus (Rmult a1 x1) y1) r).

Definition MinOrMax :=
    [z : R] [p : float] (isMin b radix z p) ∨ (isMax b radix z p).

Theorem MinOrMax_Rlt:
 (z : R)
 (p : float)
 (MinOrMax z p) → (Rlt (Rabsolu (Rminus z p)) (Fulp b radix precision p)).

Theorem MinOrMax1:
 (z : R)
 (p : float)
 (Fbounded b p) →
 (Fcanonic radix b p) →
 (Rlt R0 p) →
 (Rlt
  (Rabsolu (Rminus z p)) (Fulp b radix precision (FPred b radix precision p))) →
  (MinOrMax z p).

Theorem MinOrMax2:
 (z : R)
 (p : float)
 (Fbounded b p) →
 (Fcanonic radix b p) →
 (Rlt R0 p) →
 (Rlt (Rabsolu (Rminus z p)) (Fulp b radix precision p)) →
 (Rle p z) → (MinOrMax z p).

Theorem Axpy_aux1:
 (Fcanonic radix b u) →
 (Rle
  (Rabsolu (Rminus (Rmult (FtoRradix a) (FtoRradix x)) (FtoRradix t)))
  (Rmult
   (Rinv (S (S (S (S O))))) (Fulp b radix precision (FPred b radix precision u)))) →
 (Rlt R0 u) →
```

```
 (Rle (Rmult (S (S (S (S O)))) (Rabsolu t)) (Rabsolu u)) →
 (Rlt
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rmult
   (Rinv (S (S (S (S O))))) (Fulp b radix precision (FPred b radix precision u)))) →
  (MinOrMax (Rplus (Rmult a1 x1) y1) u).
```

Theorem *Axpy_aux1_aux1*:

```
 (Fnormal radix b t) →
 (Fcanonic radix b u) →
 (Rlt R0 u) →
 (Rle (Rmult (S (S (S (S O)))) (Rabsolu t)) (Rabsolu u)) →
  (Rle
   (Rabsolu (Rminus (Rmult (FtoRradix a) (FtoRradix x)) (FtoRradix t)))
   (Rmult
    (Rinv (S (S (S (S O)))))
    (Fulp b radix precision (FPred b radix precision u)))).
```

Theorem *Axpy_aux2*:

```
 (Fcanonic radix b u) →
 (Fsubnormal radix b t) →
 (Rlt R0 u) →
 <R> u == (Rplus t y) →
 (Rlt
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rmult
   (Rinv (S (S (S (S O))))) (Fulp b radix precision (FPred b radix precision u)))) →
  (MinOrMax (Rplus (Rmult a1 x1) y1) u).
```

Theorem *Axpy_aux1_aux2*:

```
 (Fsubnormal radix b t) →
 (Fcanonic radix b u) →
 (Rlt R0 u) →
 (Zle (Zs (Zopp (dExp b))) (Fexp (FPred b radix precision u))) →
  (Rle
   (Rabsolu (Rminus (Rmult (FtoRradix a) (FtoRradix x)) (FtoRradix t)))
   (Rmult
    (Rinv (S (S (S (S O)))))
    (Fulp b radix precision (FPred b radix precision u)))).
```

Theorem *Axpy_aux1_aux3*:

```
 (Fsubnormal radix b t) →
 (Fcanonic radix b u) →
 (Rlt R0 u) →
 (Zle (Zs (Zopp (dExp b))) (Fexp (FPred b radix precision u))) →
  (Rle
   (Rabsolu (Rminus (Rmult (FtoRradix a) (FtoRradix x)) (FtoRradix t)))
   (Rmult
    (Rinv (S (S (S (S O)))))
    (Fulp b radix precision (FPred b radix precision u)))).
```

Theorem *Axpy_aux3*:

```
(Fcanonic radix b u) →
(Fsubnormal radix b t) →
(Rlt R0 u) →
(Fexp (FPred b radix precision u)) = (Zopp (dExp b)) →
(Zle (Zs (Zopp (dExp b))) (Fexp u)) →
(Rlt
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rmult
    (Rinv (S (S (S (S (S O)))))) (Fulp b radix precision (FPred b radix precision u)))) →
  (MinOrMax (Rplus (Rmult a1 x1) y1) u).
```

Theorem *AxpyPos*:

```
(Fcanonic radix b u) →
(Fcanonic radix b t) →
(Rlt R0 u) →
(Rle (Rmult (S (S (S (S O)))) (Rabsolu t)) (Rabsolu u)) →
(Rlt
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rmult
    (Rinv (S (S (S (S (S O)))))) (Fulp b radix precision (FPred b radix precision u)))) →
  (MinOrMax (Rplus (Rmult a1 x1) y1) u).
```

Definition *FLess* :=

```
  [p : float] Cases (case_Rabsolu p) of
                      (leftT _) ⇒ (FSucc b radix precision p)
                    | (rightT _) ⇒ (FPred b radix precision p)
                  end.
```

Theorem *UlpFlessuGe_aux*:

```
(p : float)
(Fbounded b p) →
(Fcanonic radix b p) →
  (Rle (Rminus (Rabsolu p) (Fulp b radix precision p)) (Rabsolu (FLess p))).
```

Theorem *UlpFlessuGe*:

```
(Fcanonic radix b u) →
  (Rle
    (Rplus
      (Rplus
        (Rmult
          (Rinv
            (Rmult
              (Rmult (S (S (S (S O)))) (Rminus (powerRZ radix precision) R1))
              (Rplus R1 (powerRZ radix (Zopp precision)))))
          (Rmult (Rminus R1 (powerRZ radix (Zs (Zopp precision)))) (Rabsolu y)))
        (Ropp
          (Rmult
            (Rinv
              (Rmult
                (Rmult
                  (Rmult (S (S (S (S O)))) (Rminus (powerRZ radix precision) R1))
                  (Rplus R1 (powerRZ radix (Zopp precision))))
```

```
          (Rminus R1 (powerRZ radix (Zopp precision)))))
        (Rmult
         (Rminus R1 (powerRZ radix (Zs (Zopp precision)))) (Rabsolu (Rmult a x))))))
    (Ropp
     (Rmult
      (powerRZ radix (Zpred (Zopp (dExp b))))
      (Rplus
       (Rinv (Rmult (S (S O)) (Rminus (powerRZ radix precision) R1)))
       (Rmult
        (Rinv
         (Rmult
          (Rmult
           (Rmult (S (S (S (S O)))) (Rminus (powerRZ radix precision) R1))
           (Rplus R1 (powerRZ radix (Zopp precision))))
          (Rminus R1 (powerRZ radix (Zopp precision)))))
        (Rminus R1 (powerRZ radix (Zs (Zopp precision))))))))))
  (Rmult (Rinv (S (S (S (S O))))) (Fulp b radix precision (FLess u)))).

Theorem UlpFlessuGe2:
 (Fcanonic radix b u) →
  (Rlt
   (Rplus
    (Rplus
     (Rmult
      (Rmult
       (powerRZ radix (Zpred (Zpred (Zopp precision))))
       (Rminus R1 (powerRZ radix (Zs (Zopp precision))))) (Rabsolu y))
     (Ropp
      (Rmult
       (powerRZ radix (Zpred (Zpred (Zopp precision)))) (Rabsolu (Rmult a x)))))
    (Ropp (powerRZ radix (Zpred (Zpred (Zopp (dExp b)))))))
   (Rmult (Rinv (S (S (S (S O))))) (Fulp b radix precision (FLess u)))).
End AxpyAux.
Section Axpy.

Local radix := (POS (xO xH)).

Local FtoRradix := (FtoR radix).
Coercion FtoRradix : float >→ R.
Variable b : Fbound.
Variable precision : nat.
Hypothesis precisionGreaterThanOne : (lt (S O) precision).
Hypothesis pGivesBound : (POS (vNum b)) = (Zpower_nat radix precision).

Theorem MinOrMax_Fopp:
 (x : R) (f : float) (MinOrMax b (Ropp x) (Fopp f)) → (MinOrMax b x f).

Theorem MinOrMax3_aux:
 (z : R)
 (p : float)
 (Fbounded b p) →
 (Fcanonic radix b p) →
```

```
R0 == p →
(Rle z R0) →
(Rlt (Ropp z) (Fulp b radix precision (FPred b radix precision p))) →
  (MinOrMax b z p).
```

Theorem *MinOrMax3*:

```
(z : R)
(p : float)
(Fbounded b p) →
(Fcanonic radix b p) →
R0 == p →
(Rlt
  (Rabsolu (Rminus z p)) (Fulp b radix precision (FPred b radix precision p))) →
  (MinOrMax b z p).
```

Theorem *Axpy_tFlessu*:

```
(a1, x1, y1 : R)
(a, x, y, t, u : float)
(Fbounded b a) →
(Fbounded b x) →
(Fbounded b y) →
(Fbounded b t) →
(Fbounded b u) →
(Closest b radix (Rmult a x) t) →
(Closest b radix (Rplus t y) u) →
(Fcanonic radix b u) →
(Fcanonic radix b t) →
(Rle (Rmult (S (S (S (S O)))) (Rabsolu t)) (Rabsolu u)) →
(Rlt
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rmult
    (Rinv (S (S (S (S O))))) (Fulp b radix precision (FLess b precision u)))) →
  (MinOrMax b (Rplus (Rmult a1 x1) y1) u).
```

Theorem *Axpy_opt*:

```
(a1, x1, y1 : R)
(a, x, y, t, u : float)
(Fbounded b a) →
(Fbounded b x) →
(Fbounded b y) →
(Fbounded b t) →
(Fbounded b u) →
(Closest b radix (Rmult a x) t) →
(Closest b radix (Rplus t y) u) →
(Fcanonic radix b u) →
(Fcanonic radix b t) →
(Rle
  (Rmult
    (Rmult
      (Rplus
        (S (S (S (S O))))
        (Rmult (S (S (S (S O)))) (powerRZ radix (Zopp precision))))
```

```
    (Rinv (Rminus R1 (powerRZ radix (Zopp precision)))))
   (Rplus (Rabsolu (Rmult a x)) (powerRZ radix (Zpred (Zopp (dExp b))))))
  (Rabsolu y)) →
 (Rle
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rplus
   (Rplus
    (Rmult
     (Rmult
      (powerRZ radix (Zpred (Zpred (Zopp precision))))
      (Rminus R1 (powerRZ radix (Zs (Zopp precision)))))) (Rabsolu y))
    (Ropp
     (Rmult
      (powerRZ radix (Zpred (Zpred (Zopp precision)))) (Rabsolu (Rmult a x)))))
   (Ropp (powerRZ radix (Zpred (Zpred (Zopp (dExp b)))))))) →
  (MinOrMax b (Rplus (Rmult a1 x1) y1) u).

Theorem Axpy_Simpl1:
 (a1, x1, y1 : R)
 (a, x, y, t, u : float)
 (le (S (S (S (S O)))) precision) →
 (Fbounded b a) →
 (Fbounded b x) →
 (Fbounded b y) →
 (Fbounded b t) →
 (Fbounded b u) →
 (Closest b radix (Rmult a x) t) →
 (Closest b radix (Rplus t y) u) →
 (Fcanonic radix b u) →
 (Fcanonic radix b t) →
 (Rle
  (Rmult
   (S (S (S (S (S (S O))))))
   (Rplus (Rabsolu (Rmult a x)) (powerRZ radix (Zpred (Zopp (dExp b))))))
  (Rabsolu y)) →
 (Rle
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rplus
   (Rplus
    (Rmult
     (Rmult
      (powerRZ radix (Zpred (Zpred (Zopp precision))))
      (Rminus R1 (powerRZ radix (Zs (Zopp precision)))))) (Rabsolu y))
    (Ropp
     (Rmult
      (powerRZ radix (Zpred (Zpred (Zopp precision)))) (Rabsolu (Rmult a x)))))
   (Ropp (powerRZ radix (Zpred (Zpred (Zopp (dExp b)))))))) →
  (MinOrMax b (Rplus (Rmult a1 x1) y1) u).

Theorem Axpy_Simpl1bis:
 (a1, x1, y1 : R)
 (a, x, y, t, u : float)
```

```
(le (S (S (S (S O)))) precision) →
(Fbounded b a) →
(Fbounded b x) →
(Fbounded b y) →
(Fbounded b t) →
(Fbounded b u) →
(Closest b radix (Rmult a x) t) →
(Closest b radix (Rplus t y) u) →
(Fcanonic radix b u) →
(Fcanonic radix b t) →
(Rle
  (Rmult
    (S (S (S (S (S O))))))
    (Rplus (Rabsolu (Rmult a x)) (powerRZ radix (Zpred (Zopp (dExp b))))))
  (Rabsolu y)) →
(Rle
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rplus
    (Rmult
      (Rmult
        (powerRZ radix (Zpred (Zpred (Zopp precision))))
        (Rminus
          (Rmult (S (S (S (S (S O))))) (Rinv (S (S (S (S (S O)))))))
          (powerRZ radix (Zs (Zopp precision))))) (Rabsolu y))
    (Ropp (powerRZ radix (Zpred (Zpred (Zopp (dExp b)))))))) →
  (MinOrMax b (Rplus (Rmult a1 x1) y1) u).
```

Theorem *Axpy_Simpl2*:

```
(a1, x1, y1 : R)
(a, x, y, t, u : float)
(le (S (S (S (S O)))) precision) →
(Fbounded b a) →
(Fbounded b x) →
(Fbounded b y) →
(Fbounded b t) →
(Fbounded b u) →
(Closest b radix (Rmult a x) t) →
(Closest b radix (Rplus t y) u) →
(Fcanonic radix b u) →
(Fcanonic radix b t) →
(Rle
  (Rmult
    (S (S (S (S (S O))))))
    (Rplus (Rabsolu (Rmult a x)) (powerRZ radix (Zpred (Zopp (dExp b))))))
  (Rabsolu y)) →
(Rle
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rplus
    (Rmult
      (Rmult
        (powerRZ radix (Zpred (Zpred (Zopp precision))))
        (Rmult (S (S O)) (Rinv (S (S (S O)))))) (Rabsolu y))
```

```
    (Ropp (powerRZ radix (Zpred (Zpred (Zopp (dExp b)))))))) →
  (MinOrMax b (Rplus (Rmult a1 x1) y1) u).
End Axpy.
Section AxpyFmac.

Local radix := (POS (xO xH)).

Local FtoRradix := (FtoR radix).
Coercion FtoRradix : float >→ R.
Variable b : Fbound.
Variable precision : nat.
Hypothesis precisionGreaterThanOne : (lt (S O) precision).
Hypothesis pGivesBound : (POS (vNum b)) = (Zpower_nat radix precision).

Theorem AxpyPos_Fmac:
 (a1, x1, y1 : R)
 (a, x, y, u : float)
 (Fbounded b a) →
 (Fbounded b x) →
 (Fbounded b y) →
 (Fbounded b u) →
 (Closest b radix (Rplus (Rmult a x) y) u) →
 (Fcanonic radix b u) →
 (Rlt R0 u) →
 (Rlt
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rmult (Rinv (S (S O))) (Fulp b radix precision (FPred b radix precision u)))) →
  (MinOrMax b (Rplus (Rmult a1 x1) y1) u).

Theorem AxpyFLessu_Fmac:
 (a1, x1, y1 : R)
 (a, x, y, u : float)
 (Fbounded b a) →
 (Fbounded b x) →
 (Fbounded b y) →
 (Fbounded b u) →
 (Closest b radix (Rplus (Rmult a x) y) u) →
 (Fcanonic radix b u) →
 (Rlt
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rmult (Rinv (S (S O))) (Fulp b radix precision (FLess b precision u)))) →
  (MinOrMax b (Rplus (Rmult a1 x1) y1) u).

Theorem Axpy_opt_Fmac:
 (a1, x1, y1 : R)
 (a, x, y, u : float)
 (Fbounded b a) →
 (Fbounded b x) →
 (Fbounded b y) →
 (Fbounded b u) →
 (Closest b radix (Rplus (Rmult a x) y) u) →
 (Fcanonic radix b u) →
```

```
 (Rlt
  (Rplus (Rabsolu (Rminus y1 y)) (Rabsolu (Rminus (Rmult a1 x1) (Rmult a x))))
  (Rminus
   (Rmult
    (Rabsolu (Rplus (Rmult a x) y))
    (Rmult
     (powerRZ radix (Zpred (Zopp precision)))
     (Rminus R1 (powerRZ radix (Zs (Zopp precision))))))
   (Rmult
    (powerRZ radix (Zpred (Zpred (Zopp (dExp b)))))
    (Rmult
     (Rplus R1 (Rplus R1 R1))
     (Rinv (Rminus (powerRZ radix precision) (powerRZ radix (Zopp precision)))))))) →
  (MinOrMax b (Rplus (Rmult a1 x1) y1) u).
End AxpyFmac.
```

A simple Maple test qualifying the accuracy of Horner's rule for polynomials

Formulas of Section 3 are part of the graduate work of **Sylvie BOLDO**, 2001-2003.
Some methods were first developed during the Ph.D. of **Claire MOREAU-FINOT**, 1998-2001.

Selected references

S. Boldo and M. Daumas, Faithful rounding without fused multiply and accumulate, in IMACS-GAMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics, (Paris, France), 2002.

M. Daumas and C. Moreau-Finot, Exponential: implementation trade-offs for hundred bit precision, in Real Numbers and Computers, (Dagstuhl, Germany), pp. 61-74, 2000.

Selected links

http://www.ens-lyon.fr/~daumas/SoftArith
http://www.ens-lyon.fr/~sboldo/coq/

A Treatment of polynomials

```
> HornerStep := proc (P)
> simplify((P - eval (P, x = 0)) / x);         end:
> EvenPoly   := proc (P)
> simplify((P + eval (P, x = -x)) / 2);
> subs(x = sqrt(x), %);                        end:
```

B Exponent, unit in the last place and rounding

```
> MinExpIEEE := proc (lgex) -2^(lgex-1) + 2;     end:
> MaxExpIEEE := proc (lgex)  2^(lgex-1) - 1;     end:
> UlpCstIEEE := proc (lgfr)  2^(-lgfr);          end:
> LambdaIEEE := proc (lgex)  2^MinExpIEEE(lgex); end:
> ExpIEEE    := proc(x, lgex) local elem, exposant;
> if x = 0 then exposant := MinExpIEEE(lgex);
> else
> exposant := floor(log[2](abs(evalf(x))));
> if abs(x) >= 2^(exposant+1)
```

```
> then exposant := exposant + 1; fi;
> if abs(x) <  2^ exposant
> then exposant := exposant - 1; fi;
> fi;
> if exposant < MinExpIEEE(lgex)
> then exposant := MinExpIEEE(lgex); fi;
> if exposant > MaxExpIEEE(lgex)
> then infinity else exposant;
> fi;
> end:
> UlpIEEE    := proc(x, lgfr, lgex) local elem;
> UlpCstIEEE(lgfr) * 2^ExpIEEE (x, lgex); end:
> BiasedIEEE := proc(x, lgfr, lgex) local elem, ulp;
> if type(x, numeric) then
> ulp := UlpIEEE(x, lgfr, lgex);
> if ulp < infinity
> then round(x / ulp)*ulp; else x * infinity fi;
> else if type(x, name) then
> x;
> else
> applyop(BiasedIEEE, {seq(i, i=1..nops(x))}, x,
> lgfr, lgex);
> fi; fi; end:
> UpIEEE     := proc(x, lgfr, lgex) local elem, ulp;
> ulp := UlpIEEE(x, lgfr, lgex);
> if ulp < infinity
> then ceil(x / ulp)*ulp; else x * infinity fi;
> end:
```

C Qualifying the accuracy of Horner's rule

The polynomials are supposed to be exact arrays of coefficients.

Possibly use BiasedIEEE(P, lgfr, lgex) where P is a polynomial expression to get a rounded polynomial.

XMax is the maximum value of X defining the range [-XMax..XMax].

lgfr, lgex are the length of the fraction and the exponent:

(23,8) for single precision;
(52, 11) for double precision;
(63,15) for PC double extended.

C.1 Usual bounds for Horner's rule using the standard model

Input

relround: A bound on the relative error using the so called "Standard Model" see Higham, 2002, eq. 2.4, p 40.

Output:

PMax: A bound on the polynomial value;
AbsError: A bound on the absolute error of the polynomial evaluation.

```
> HornerBounds := proc(P, XMax, relround)
> local rec_max, rec_erreur, PMax, SMax;
> if degree(P) = 0 then abs(P), 0;
> else
> rec_max, rec_erreur :=
> HornerBounds (HornerStep(P), XMax, relround);
> PMax := rec_max * XMax * (1 + relround);
> SMax := (PMax + abs(eval(P, x = 0))) * (1 + relround);
> SMax, rec_erreur * XMax + (PMax + SMax) * relround;
> fi; end:
```

C.2 Precise bounds for Horner's rule with the IEEE 754 standard

Biased rounding is used in place of the even tie breaking rule to yield a safe overestimation.

Input

lgfr, lgex: Length of the fraction and the exponent. (23,8) for single precision, (52, 11) for double precision and (63,15) for PC double extended.

Output:

PMax: A bound on the polynomial value;
AbsError: A bound on the absolute error of the polynomial evaluation.

```
> HornerIEEE   := proc(P, XMax, lgfr, lgex)
> local rec_max, rec_erreur, PMax, SMax;
> if degree(P) = 0 then abs(P), 0;
> else
> rec_max, rec_erreur :=
> HornerIEEE   (HornerStep(P), XMax, lgfr, lgex);
> PMax := BiasedIEEE (rec_max * XMax,  lgfr, lgex);
> SMax := BiasedIEEE ((PMax + abs(eval(P, x = 0))),
> lgfr, lgex);
> SMax,   rec_erreur * XMax
> + (  UlpIEEE(PMax, lgfr, lgex)
> + UlpIEEE(SMax, lgfr, lgex)) / 2;
> fi; end:
```

C.3 Guaranteed faithful result of Horner's rule

Biased rounding is used in place of the even tie breaking rule to yield a safe overestimation.

Input:

Res_Err: The approximation error of function f by polynomial P Res_Err = max | P(x) - f(x) | / x.

Output:

PMax: A bound on the polynomial value;
AbsError: A bound on the absolute error of the polynomial evaluation;
Guarantee: A binary encoded guarantee. If strictly less than 1, the final result is faithful. If 0, all the recurent function calls were faithful.

C.3.1 Faithful multiply and accumulate operation guaranteed with Coq

See Boldo and Daumas, 2002.

Coq proof script are available and they can be reviewed and checked at

http://www.ens-lyon.fr/~sboldo/coq

All the inputs are supposed to be positive.

```
> AXPYCond1 := proc (a, x, y, lgfr, lgex)
> 5 * (2 + UlpCstIEEE(lgfr))
> / (2 - UlpCstIEEE(lgfr))
> * (  a * x
> + LambdaIEEE(lgex) * UlpCstIEEE(lgfr) / 2)
> <= y; end:
> AXPYCond2 := proc (a, err_a, x, err_x, y, err_y,
> lgfr, lgex)
> err_y + err_a * x + a * err_x + err_x * err_a
> <= UlpCstIEEE(lgfr) / 8
> * (  (1 - UlpCstIEEE(lgfr)) * y
> - a *x - 2 * LambdaIEEE(lgex));
> end:
> AXPYFCond := proc (a, err_a, x, err_x, y, err_y,
> lgfr, lgex)
> err_y + err_a * x + a * err_x + err_x * err_a
> <= UlpCstIEEE(lgfr) / 4
> * (  (1 - UlpCstIEEE(lgfr)) * (y - a * x)
> - 6 * LambdaIEEE(lgex) * UlpCstIEEE(lgfr)
> / (4 - UlpCstIEEE(lgfr)^2));
> end:
> HornerAXPY := proc(P, XMax, Err0, Err1, ErrX, lgfr, lgex)
> local rec_max, erreur, rec_erreur, check, rec_check,
> PMax, SMax;
> if degree(P) = 0 then abs(P), Err0, 0;
```

```
> else
> rec_max, rec_erreur, rec_check :=
> HornerAXPY (HornerStep(P), XMax, Err1, 0, ErrX,
> lgfr, lgex);
> PMax := BiasedIEEE (rec_max * XMax, lgfr, lgex);
> SMax := BiasedIEEE (PMax + abs(eval(P, x = 0)),
> lgfr, lgex);
> if  AXPYCond1
> (rec_max,             XMax,
> abs(eval (P, x = 0)),       lgfr, lgex)
> and AXPYCond2
> (rec_max, rec_erreur, XMax, ErrX,
> abs(eval (P, x = 0)), Err0, lgfr, lgex)
> then check := 0; else check := 1; fi;
> SMax,   rec_erreur * XMax + rec_max * ErrX + Err0
> + (  UlpIEEE(PMax, lgfr, lgex)
> + UlpIEEE(SMax, lgfr, lgex)) / 2,
> check + rec_check / 2 ;
> fi; end:
> HornerFAXPY := proc(P, XMax, Err0, Err1, ErrX, lgfr, lgex)
> local rec_max, erreur, rec_erreur, check, rec_check,
> SMax;
> if degree(P) = 0 then abs(P), Err_0, 0;
> else
> rec_max, rec_erreur, rec_check :=
> HornerAXPY (HornerStep(P), XMax, Err1, 0, ErrX,
> lgfr, lgex);
> SMax := BiasedIEEE
> (rec_max * XMax + abs(eval(P, x = 0)), lgfr, lgex);
> if AXPYFCond
> (rec_max, rec_erreur, XMax, ErrX,
> abs(eval (P, x = 0)), Err0, lgfr, lgex)
> then check := 0; else check := 1; fi;
> SMax,   rec_erreur * XMax + rec_max * ErrX + Err0
> + UlpIEEE(SMax, lgfr, lgex) / 2,
> check + rec_check / 2 ;
> fi; end:
> DoubleAXPY := proc(P, XMax, Err0, Err1, ErrX, lgfr, lgex)
> local Even, odd_max, odd_err, odd_check,
> Odd,  eve_max, eve_err, eve_check,
> check, PMax, SMax, EMax, XMax2, ErrX2;
> Even := HornerStep(EvenPoly(P));
> Odd  := EvenPoly(HornerStep(P));
> XMax2 := UpIEEE (XMax^2, lgfr, lgex);
> ErrX2 :=   2 * XMax * ErrX + ErrX^2
```

```
> + UlpIEEE(XMax2,  lgfr, lgex) / 2;
> odd_max, odd_err, odd_check :=
> HornerAXPY (Odd,  XMax2, Err1, 0, ErrX2, lgfr, lgex);
> eve_max, eve_err, eve_check :=
> HornerAXPY (Even, XMax2, 0,    0, ErrX2, lgfr, lgex);
> PMax := BiasedIEEE (eve_max * XMax,   lgfr, lgex);
> EMax := BiasedIEEE (PMax + odd_max,   lgfr, lgex);
> odd_err := odd_err + eve_err * XMax + ErrX * eve_max
> + (  UlpIEEE(PMax, lgfr, lgex)
> + UlpIEEE(EMax, lgfr, lgex)) / 2;
> if  AXPYCond1
> (EMax,             XMax,
> abs(eval(P, x = 0)), lgfr, lgex)
> and AXPYCond2
> (EMax, odd_err, XMax, ErrX,
> abs(eval(P, x = 0)), Err0, lgfr, lgex)
> then check := 0; else check := 1; fi;
> PMax := BiasedIEEE (EMax * XMax,      lgfr, lgex);
> SMax := BiasedIEEE (PMax + abs(eval(P, x = 0)),
> lgfr, lgex);
> SMax,   odd_err * XMax + ErrX * EMax + Err0
> + (  UlpIEEE(PMax, lgfr, lgex)
> + UlpIEEE(SMax, lgfr, lgex)) / 2,
> check;
> end:
```

D Example 1: Polynomial approximation to the exponential

```
> with(numapprox);
```

$$[chebdeg,\ chebmult,\ chebpade,\ chebsort,\ chebyshev,\ confracform,\ hermite\_pade,\ hornerform,\ infnorm,\ laurent,\ minimax,\ pade,\ remez]$$

```
> with (orthopoly);
```

$$[G,\ H,\ L,\ P,\ T,\ U]$$

```
> Digits := 40:  XMax := 2^(-4):
> lgfr := 52:
> lgex := 11:
> chexp := chebyshev((exp(x) - 1 - x - x^2/2)/x^3, x=-XMax..XMax,
> UlpCstIEEE(lgfr)/XMax/XMax/XMax/2);
```

$$\begin{aligned} chexp := {} & .16668294384369481742265329197656682861 03\,\mathrm{T}(0,\ 16\,x) \\ & + .00260442099460103589760785418619932708 6753\,\mathrm{T}(1,\ 16\,x) \\ & + .00001627755554349751995437496401565024040209\,\mathrm{T}(2,\ 16\,x) \\ & + .84778442485114635340557955360768624470 67\,10^{-7}\,\mathrm{T}(3,\ 16\,x) \\ & + .37844634598785479855225155775309262980 45\,10^{-9}\,\mathrm{T}(4,\ 16\,x) \\ & + .14779325339373674399041344484505726508 93\,10^{-11}\,\mathrm{T}(5,\ 16\,x) \end{aligned}$$

```
> approx := simplify(1 + x/2 + x^2 * convert(chexp, polynom));
```

$$\begin{aligned} approx := {} & 1. + .50000000000000000000000000000000000000000\, x \\ & + .16666666666659766589055371556480273612 30\, x^2 \\ & + .04166666666661189157421871538966267237013\, x^3 \\ & + .00833333338015414709001254656482264475216 4\, x^4 \\ & + .00138888892944293803627902459393681607241 5\, x^5 \\ & + .00019841487784528041662336286471125342869 49\, x^6 \\ & + .00002479559335529454401063868293469612268772\, x^7 \end{aligned}$$

```
> infnorm(approx, x=-XMax..XMax);
```

$$1.03191134268575080513467438682686286273 0$$

```
> infnorm(exp(x) - 1 - x * approx, x=-XMax..XMax);
```

$$.42424734454662107441800936973368733193673\, 10^{-17}$$

```
> HornerBounds(approx, XMax, UlpCstIEEE(lgfr)/2):
> evalf(%); log[2] (%[2]);
```

$$\begin{aligned} & 1.03191134268575092693475275995913402355 4, \\ & .12180007837313228468334922288399807281 00\, 10^{-15} \\ & -52.86633445664080253931041944960545911165 \end{aligned}$$

```
> approxIEEE := BiasedIEEE(approx, lgfr, lgex); evalf(%);
```

$$\begin{aligned} approxIEEE := {} & 1 + \frac{1}{2}\, x + \frac{6004799503158175}{36028797018963968}\, x^2 + \frac{6004799503152767}{144115188075855872}\, x^3 \\ & + \frac{4803839629518891}{576460752303423488}\, x^4 + \frac{1601279914265145}{1152921504606846976}\, x^5 \\ & + \frac{3660108472028231}{18446744073709551616}\, x^6 + \frac{7318367436494265}{29514790517935282585 6}\, x^7 \end{aligned}$$

$$\begin{aligned} & 1. + .50000000000000000000000000000000000000000\, x \\ & + .16666666666659765705382767464470816776 16\, x^2 \\ & + .04166666666661188872522458837011072319001\, x^3 \\ & + .00833333338015414780419742868389221257530 2\, x^4 \\ & + .00138888892944293799308619341559278836939 5\, x^5 \\ & + .00019841487784528040938178405916403335140 79\, x^6 \\ & + .00002479559335529454380830406690083833609606\, x^7 \end{aligned}$$

```
> infnorm ((exp(x) - 1 - x * approxIEEE)/x, x=-XMax..XMax):
> erreurIEEE := UpIEEE(%, lgfr, lgex); evalf(%);
```

$$erreurIEEE := \frac{5509901405496691}{81129638414606681695789005144064}$$

$$.67914778287791326469293740624562733468 90\, 10^{-16}$$

```
> HornerIEEE(approxIEEE, XMax, lgfr, lgex):
> evalf(%); log[2] (%[2]);
```

$$\begin{aligned} & 1.03191134268575090970898600062355399131 8, \\ & .11807133732776259432036666153459484484 47\, 10^{-15} \\ & -52.91119073562933945484960582793333724361 \end{aligned}$$

```
> HornerAXPY(1 + x * approxIEEE, XMax, 0, erreurIEEE, UlpIEEE(XMax,
> lgfr, lgex)/2, lgfr, lgex);
> evalf(%); log[2] (%[2]);
```

$$\frac{2397028424260215}{2251799813685248}, \frac{15639408299229932414632750904 89}{11417981541647679048466287755595961091061972992}, \frac{1}{2}$$

$$1.064494458917859542879114087554626166821,$$
$$.136971742704123146688655118634278273752\,10^{-15},$$
$$.500000000000000000000000000000000000000$$
$$-52.696971222569850624960855622231948001 13$$

We conclude that Horner's rule applied to the polynomial yields a faithful approximation to the exponential function for the given range.

```
> DoubleAXPY(1 + x * approxIEEE, XMax, 0, erreurIEEE, UlpIEEE(XMax,
> lgfr, lgex)/2, lgfr, lgex);
> evalf(%); log[2] (%[2]);
```

$$\frac{4794056848520429}{4503599627370496},$$
$$\frac{1480082256570454858170837914938771553684159608 9}{1028440348325753776346855739098344065614209916020987414592880 64}, 0$$
$$1.064494458917859320834509162523318082094,$$
$$.143915226486392730787073636448375840800310^{-15},\ 0.$$
$$-52.625630278348323034487643529490816827 53$$

We conclude that the modified Horner's rule for a pipelined processor applied to the polynomial yields a faithful approximation to the exponential function for the given range.

```
> Tabule := proc (fct, ifct, orig, sttb, lgfr, lgtb, lgex)
> local imag, mid, cheb, pol, app, err0, err1, errX;
> imag := BiasedIEEE(evalf(fct(orig)), lgfr, lgex);
> mid  := BiasedIEEE(evalf(ifct(imag)), lgfr + lgtb, lgex);
> cheb := chebyshev(fct(x + mid), x=-sttb..sttb,
> UlpCstIEEE(lgfr+lgtb)/4);
> pol  := eval(cheb);
> app  := BiasedIEEE(pol, lgfr, lgex);
> err0 := fct(mid) - eval(app, x=0);
> err1 := infnorm((fct(x + mid) - app - err0) / x,
> x=-sttb..sttb);
> errX := UlpIEEE(sttb, lgfr, lgex)/2;
> HornerAXPY(app, sttb, evalf(err0), err1, errX, lgfr, lgex);
> end:
> lgtb := 3; sttb := 2^(-lgtb);
> sttr := (sttb + UlpIEEE(sttb, lgfr, lgex)) / 2;
```

$$lgtb := 3$$
$$sttb := \frac{1}{8}$$
$$sttr := \frac{4503599627370497}{72057594037927936}$$

```
> Tabule(x->exp(x), x->ln(x), 1/4, sttr, lgfr, lgtb, lgex);evalf(%,
> 5);
```

$$\frac{6155690842546165}{4503599627370496},\ .13761496279316402404129775561175999147901 0^{-15},\ \frac{1}{2}$$
$$1.3668,\ .13761\,10^{-15},\ .50000$$

```
> [seq([i * sttb, Tabule(x->exp(x), x->ln(x), i * sttb, sttr, lgfr,
> lgtb, lgex)[3]],
```

```
> i = -ceil(ln(2) / 2 / sttb)..ceil(ln(2) / 2 /sttb))];
```

$$[[\frac{-3}{8}, \frac{1}{2}], [\frac{-1}{4}, \frac{1}{2}], [\frac{-1}{8}, \frac{1}{2}], [0, \frac{1}{2}], [\frac{1}{8}, \frac{1}{2}], [\frac{1}{4}, \frac{1}{2}], [\frac{3}{8}, \frac{1}{2}]]$$

We conclude that Horner's rule applied to all the distinct polynomials yields a faithful approximation to the exponential function for the full range of the function.

```
> lgtb := 4; sttb := 2^(-lgtb);
> sttr := (sttb + UlpIEEE(sttb, lgfr, lgex)) / 2;
```

$$lgtb := 4$$

$$sttb := \frac{1}{16}$$

$$sttr := \frac{4503599627370497}{144115188075855872}$$

```
> Tabule(x->cos(x), x->arccos(x), 1/16, sttr, lgfr, lgtb,
> lgex);evalf(%, 5);
```

$$\frac{2252896551948915}{2251799813685248}, .11293474720127131449981040624016240940921\,10^{-15}, \frac{5}{8}$$

$$1.0005, .11293\,10^{-15}, .62500$$

```
> [seq([i * sttb, Tabule(x->cos(x), x->arccos(x), i * sttb, sttr, lgfr,
> lgtb, lgex)[3]],
> i = 0..ceil(Pi / 4 / sttb))];
```

$$[[0, \frac{21}{32}], [\frac{1}{16}, \frac{5}{8}], [\frac{1}{8}, \frac{1}{2}], [\frac{3}{16}, \frac{1}{2}], [\frac{1}{4}, \frac{1}{2}], [\frac{5}{16}, \frac{1}{2}], [\frac{3}{8}, \frac{1}{2}], [\frac{7}{16}, \frac{1}{2}], [\frac{1}{2}, \frac{1}{2}], [\frac{9}{16}, \frac{1}{2}], [\frac{5}{8}, \frac{1}{2}], [\frac{11}{16}, \frac{1}{2}], [\frac{3}{4}, \frac{1}{2}], [\frac{13}{16}, \frac{1}{2}]]$$

We conclude that Horner's rule applied to all the distinct polynomials yields a faithful approximation to the cosine function for the full given range.

```
> [seq([i * sttb, Tabule(x->sin(x), x->arcsin(x), i * sttb, sttr, lgfr,
> lgtb, lgex)[3]],
> i = 0..ceil(Pi / 4 / sttb))];
```

$$[[0, \frac{85}{64}], [\frac{1}{16}, \frac{7}{4}], [\frac{1}{8}, \frac{3}{2}], [\frac{3}{16}, \frac{3}{2}], [\frac{1}{4}, \frac{3}{2}], [\frac{5}{16}, \frac{1}{2}], [\frac{3}{8}, 0], [\frac{7}{16}, \frac{1}{2}], [\frac{1}{2}, \frac{1}{2}], [\frac{9}{16}, \frac{1}{2}], [\frac{5}{8}, \frac{1}{2}], [\frac{11}{16}, \frac{1}{2}], [\frac{3}{4}, \frac{1}{2}], [\frac{13}{16}, \frac{1}{2}]]$$

We conclude that Horner's rule applied to all the distinct polynomials fails to yield a faithful approximation to the sine function for the given range.

E Example 2: Polynomial used by Fike, 1967

```
> lgfr := 23;
> lgex := 8;
```

$$lgfr := 23$$

$$lgex := 8$$

```
> Fike := 1 + 0.6931471805599346*x + 0.2402265069563678*x*x +
> 0.05550410840231345*x*x*x + 0.009618117095313700*x*x*x*x +
> 0.001333073417706260*x*x*x*x*x + 0.0001507368551403575*x*x*x*x*x*x;
```

$$Fike := 1 + .6931471805599346\,x + .2402265069563678\,x^2 + .05550410840231345\,x^3 + .009618117095313700\,x^4 + .001333073417706260\,x^5 + .0001507368551403575\,x^6$$

```
> FikeIEEE := BiasedIEEE(Fike, lgfr, lgex);
```

$$FikeIEEE := 1 + \frac{1453635}{2097152}\,x + \frac{1007583}{4194304}\,x^2 + \frac{14899271}{268435456}\,x^3 + \frac{10327375}{1073741824}\,x^4 + \frac{11451013}{8589934592}\,x^5 + \frac{5179279}{34359738368}\,x^6$$

```
> Fike - FikeIEEE;
```

$$\begin{aligned} &-.19046650093750000000000000000000\,10^{-8}\,x \\ &-.23052184843750000000000000000000\,10^{-9}\,x^2 \\ &-.13081414907958984375000000000000000\,10^{-8}\,x^3 \\ &-.37870582743530273437500000000000\,10^{-9}\,x^4 \\ &+.54066845689544677734375000000000\,10^{-10}\,x^5 \\ &-.27637637797164916992187500000000\,10^{-11}\,x^6 \end{aligned}$$

```
> HornerAXPY(FikeIEEE, 1/16, 0, 0, 0, lgfr, lgex);
```

$$\frac{8760003}{8388608},\ \frac{146165235969}{2305843009213693952},\ 0$$

We conclude that Horner's rule applied to the polynomial yields a faithful result **for the given range**.

```
> infnorm ((2^x - FikeIEEE)/x, x=-1/16..0):
> FikeErreur := UpIEEE(%, lgfr, lgex);
```

$$FikeErreur := \frac{8577801}{4503599627370496}$$

```
> HornerAXPY(FikeIEEE, 1/16, 0, FikeErreur, 0, lgfr, lgex);
```

$$\frac{8760003}{8388608},\ \frac{146439725601}{2305843009213693952},\ 0$$

We conclude that Horner's rule applied to the polynomial yields a faithful approximation to 2^x for the given range.

```
//  A simple C++ test qualifying the accuracy of Horner's rule for polynomials

//  Copyright (C) 2003, Marc DAUMAS, Marc.Daumas@ENS-Lyon.Fr

//  Formulas are part of the graduate work of Sylvie BOLDO, 2001-2002.

//  Selected reference

//    S. Boldo and M. Daumas, Faithful rounding without fused multiply and
//    accumulate, in IMACS-GAMM International Symposium on Scientific
//    Computing, Computer Arithmetic and Validated Numerics, (Paris,
//    France), 2002.

//  Selected links

//    http://www.ens-lyon.fr/~daumas/SoftArith
//    http://www.ens-lyon.fr/~sboldo/coq/

//  This library is free software; you can redistribute it and/or modify
//  it under the terms of the GNU Lesser General Public License version
//  2.1 as published by the Free Software Foundation.

//  This library is distributed in the hope that it will be useful, but
//  WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
//  Lesser General Public License for more details.

//  You should have received a copy of the GNU Lesser General Public
//  License along with this library; if not, write to the Free Software
//  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
//  USA.

#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <ieeefp.h>
#include <float.h>

#include <nan.h>

double *Coefficients;
int Degres;

double UlpIEEE(double x) {
  if (IsNANorINF(x)) return x;
  ((dnan *)&(x))->inf_parts.fraction_low   = 0;
  ((dnan *)&(x))->inf_parts.bits           = 0;
  ((dnan *)&(x))->inf_parts.sign           = 0;
  if (x == 0) // if x was a denormal, otherwise its fraction is 0
    ((dnan *)&(x))->inf_parts.fraction_low = 1;
  else
    x = x * DBL_EPSILON;
  return x;
}

int AXPYCond1(double a, double x, double y) {
  double petit;
```

```
  fp_rnd old;
  old = fpgetround(); fpsetround(FP_RP);
  petit  = 5 * (2 + DBL_EPSILON) / (2 - DBL_EPSILON) * (a * x + DBL_MIN / 2) ;
  fpsetround(old);
  return petit  <= y;
}

int AXPYCond2(double a, double err_a, double x, double err_x, double y, double err_y) {
  double petit, grand, ax;
  fp_rnd old;
  old = fpgetround(); fpsetround(FP_RP);
  petit  = err_y + err_a*(x + err_x) + x*err_a;
  grand  = a * x + 2 * DBL_MIN / DBL_EPSILON;
  grand  = DBL_EPSILON / 8 * ((DBL_EPSILON - 1) * y + grand);
  fpsetround(old);
  return petit <= -grand;
}

double rec_max;
double rec_erreur;
double rec_check;

void HornerAXPY (int position, double XMax, double Err0, double Err1, double ErrX) {
  double check, PMax, SMax;
  fp_rnd old;
  if (position == Degres) {
    rec_max    = fabs(Coefficients[position]);
    rec_erreur = Err0;
    rec_check  = 0.0;
  } else {
    HornerAXPY (position + 1, XMax, Err1, 0, ErrX);
    PMax = rec_max * XMax; SMax = PMax + fabs(Coefficients[position]);
    if (AXPYCond1 (rec_max,             XMax,       fabs(Coefficients[position]))
        &&
        AXPYCond2 (rec_max, rec_erreur, XMax, ErrX, fabs(Coefficients[position]), Err0))
      check = 0; else check = 1;
    old = fpgetround(); fpsetround(FP_RP);
    rec_erreur *= XMax;
    rec_erreur += rec_max * ErrX + Err0 + (PMax + SMax) * DBL_EPSILON / 2.0;
    // ulp of PMax and SMax would be tighter
    fpsetround(old);
    rec_max    = SMax;
    rec_check = check + rec_check / 2;
  }
}

int main(void) {
  int lgfr;
  double Err0, Err1, ErrX, XMax;

  cout << ... Omitted messages ...;

  // Enter the polynomial
  Degres = 9;
  Coefficients = new double[Degres + 1];
  Coefficients[0] = 1.0;
  Coefficients[1] = 1.0;
```

```
  Coefficients[2] = 0.5;
  Coefficients[3] = 0.16666666666659765705382767464470816776l6    ;
  Coefficients[4] = 0.04166666666661188872522458837011072319001   ;
  Coefficients[5] = 0.00833333338015414780419742868389221257530 2  ;
  Coefficients[6] = 0.00138888892944293799308619341559278836939 5  ;
  Coefficients[7] = 0.00019841487784528040938178405916403335140 79 ;
  Coefficients[8] = 0.00002479559335529454380830406690083833609606;

  // Set the residual error
  Err0 = 0;
  Err1 = 0.67914778287791326469293740624562733468 90e-16;

  // Set the range for the indeterminate
  XMax = 0.0625;
  ErrX = UlpIEEE(XMax);

  // Test the criterion
  HornerAXPY (0, XMax, Err0, Err1, ErrX);

  cout << "Result : " << rec_max
       <<         " " << rec_erreur
       <<         " " << rec_check << endl;

  if (rec_check == 0.0)
    cout << "All the steps of Horner's rule were faithful\n";
  else if (rec_check < 1.0)
    cout << "The final step of Horner's rule was faithful\n";
  else if (rec_check >= 1.0)
    cout << "Error too large to guarantee that Horner's rule was faithful\n";
}
```

```
//  A simple JAVA test qualifying the accuracy of Horner's rule for polynomials

//  Copyright (C) 2003, Marc DAUMAS, Marc.Daumas@ENS-Lyon.Fr

//  Formulas are part of the graduate work of Sylvie BOLDO, 2001-2002.

//  Selected reference

//    S. Boldo and M. Daumas, Faithful rounding without fused multiply and
//    accumulate, in IMACS-GAMM International Symposium on Scientific
//    Computing, Computer Arithmetic and Validated Numerics, (Paris,
//    France), 2002.

//  Selected links

//    http://www.ens-lyon.fr/~daumas/SoftArith
//    http://www.ens-lyon.fr/~sboldo/coq/

//  This library is free software; you can redistribute it and/or modify
//  it under the terms of the GNU Lesser General Public License version
//  2.1 as published by the Free Software Foundation.

//  This library is distributed in the hope that it will be useful, but
//  WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
//  Lesser General Public License for more details.

//  You should have received a copy of the GNU Lesser General Public
//  License along with this library; if not, write to the Free Software
//  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
//  USA.

import java.io.*;
import java.lang.Double;
import java.lang.Math;

class Horner {
    public static double[] Coefficients;

    public static double UlpCstIEEE;

    public static void main(String[] args) {
        int lgfr;
        double Err0, Err1, XMax;
        System.out.println(... Omitted messages ...);

        // Initialize Ulp
        for (UlpCstIEEE = 1.0, lgfr = 52; lgfr > 0; lgfr--) UlpCstIEEE /= 2.0;

        // Enter the polynomial
        Coefficients = new double[9];
        Coefficients[0] = 1.0;
        Coefficients[1] = 1.0;
        Coefficients[2] = 0.5;
        Coefficients[3] = 0.16666666666659765705382767464470816776 16    ;
        Coefficients[4] = 0.04166666666661188872522458837011072319001   ;
        Coefficients[5] = 0.00833333338015414780419742868389221257530 2  ;
```

```
        Coefficients[6] = 0.0013888889294429379930861934155927883693 95  ;
        Coefficients[7] = 0.0001984148778452804093817840591640333514079 ;
        Coefficients[8] = 0.0000247955933552945438083040669008383360 9606;

        // Set the residual error
        Err0 = 0;
        Err1 = 0.679147782877913264692937406245627334689 0e-16;

        // Set the range for the indeterminate
        XMax = 0.0625;

        // Test the criterion
        HornerAXPY (0, XMax, Err0, Err1);

        if (rec_check == 0.0)
            System.out.println("All the steps of Horner's rule were faithful");
        else if (rec_check < 1.0)
            System.out.println("The final step of Horner's rule was faithful");
        else if (rec_check >= 1.0)
            System.out.println("Error too large to guarantee that Horner's rule was faithful");
    }

    public static double UlpIEEE(double x) {
        if (Double.isNaN(x) || Double.isInfinite(x)) return x;
        x = Math.abs(x);
        x = Double.longBitsToDouble((Double.doubleToLongBits(x)>>>52)<<52);
        if (x == 0) // if x was a denormal, otherwise its fraction is 0
            x = Double.longBitsToDouble((Double.doubleToLongBits(x)|1));
        else
            x = x * UlpCstIEEE;
        return x;
    }

    public static boolean AXPYCond1 (double a, double x, double y) {
        double petit;
        petit  = a * x;                  petit += UlpIEEE(petit);
        petit *= (5 + 8 * UlpCstIEEE);   petit += UlpIEEE(petit);
        // Overestimations, rounding up would be sufficient
        return petit  <= y;
    }

    public static boolean AXPYCond2 (double a, double err_a, double x, double y, double err_y) {
        double petit, grand, ax;
        petit  = err_a*x;                petit += UlpIEEE(petit);
        petit += err_y;                  petit += UlpIEEE(petit);
        ax     = a * x;                  ax    += UlpIEEE(ax);
        grand  = (UlpCstIEEE - 1) * y;   grand += UlpIEEE(grand);
        grand += ax;                     grand += UlpIEEE(grand);
        grand *= UlpCstIEEE / 8;         grand += UlpIEEE(grand);
        // Overestimations
        return petit <= -grand;
    }

    public static double rec_max;
    public static double rec_erreur;
    public static double rec_check;
```

```
    public static void HornerAXPY (int position, double XMax, double Err0, double Err1) {
        double check, PMax, SMax, fin;
        if (position == Coefficients.length - 1) {
            rec_max    = Math.abs(Coefficients[position]);
            rec_erreur = Err0;
            rec_check  = 0.0;
        } else {
            HornerAXPY (position + 1, XMax, Err1, 0);
            PMax = rec_max * XMax; SMax = PMax + Math.abs(Coefficients[position]);
            // Native floating point arithmetic
            if (AXPYCond1 (rec_max,             XMax, Math.abs(Coefficients[position]))
                &&
                AXPYCond2 (rec_max, rec_erreur, XMax, Math.abs(Coefficients[position]), Err0))
                check = 0; else check = 1;
            rec_erreur *= XMax;                          rec_erreur += UlpIEEE(rec_erreur);
            rec_erreur += Err0;                          rec_erreur += UlpIEEE(rec_erreur);
            // Overestimation, rounding up would be sufficient
            fin         =  UlpIEEE(PMax) + UlpIEEE(SMax); fin        += UlpIEEE(fin);
            rec_erreur += fin/2.0;                       rec_erreur += UlpIEEE(rec_erreur);
            rec_max     = SMax;
            rec_check   = check + rec_check / 2;
        }
    }
}
```