

Generating function approximations at compile time

Jean-Michel Muller

CNRS - Laboratoire LIP (CNRS, ENS Lyon, INRIA, Université Claude Bernard)

Ecole Normale Supérieure de Lyon

46 alle d'Italie

69364 Lyon cédex 07

FRANCE

Abstract—Usually, the mathematical functions used in a numerical programs are decomposed into elementary functions (such as sine, cosine, exponential, logarithm...), and for each of these functions, we use a program from a library. This may have some drawbacks: first in frequent cases, it is a compound function (e.g. $\log(1 + \exp(-x))$) that is needed, so that directly building a polynomial or rational approximation for that function (instead of decomposing it) would result in a faster and/or more accurate calculation. Also, at compile-time, we might have some information (e.g., on the range of the input value) that could help to simplify the program. We investigate the possibility of directly building accurate approximations at compile-time.

I. INTRODUCTION

There are several solutions for implementing the elementary and special functions in software as well as in hardware: polynomial or rational approximations, table look-up, shift-and-add CORDIC-like algorithms, etc.

The solution that is most used is the use of polynomial approximations, because of its versatility. In this paper, we will focus on this solution only. The basic algorithm for computing “minimax” polynomial approximations to functions in a given interval is the Remez algorithm [10], [11].

In the 70’s and the beginning of the 80’s, computing such approximations was a difficult and rather long task. Hence, the approximations used to be designed using big machines, then published in textbooks (famous references were Hart et al [8], or Cody and Waite [3]). The elementary function libraries would then be carefully written using these approximations.

Now, thanks to some algorithmic progress (especially the availability of fast and reasonably reliable multiple-precision arithmetic) and, even more, thanks to the huge increase in terms of available computing power (speed as well as memory) in the last 3 decades, such approximations can be computed in a few seconds on any PC, for instance using tools such as Maple or Mathematica.

And yet, this drastic change does not show much for the end user: even if the libraries can be designed more efficiently and quickly, even if they are significantly more accurate, we still basically do the same thing: we decompose the functions

we wish to evaluate into elementary functions, that are called from libraries.

The goal of this paper is to investigate an alternate solution: could we design the approximations at *compile time*, so that we can use possible contextual information (on range, rounding mode, desired accuracy. . .) and directly approximate a compound function ? This is a long-term project, and this paper just presents the initial ideas.

Of course, minimax approximations are not the whole thing, and there is much human expertise in the libraries. We will try to see how we can – at least partially – replace it.

The usual steps when designing a function are:

- range reduction: we must find an adequate compromise between the domain of the reduced argument and degree of the approximation;
- we start from the minimax polynomial given by the Remez algorithm;
- we tune the coefficients (so that they are exactly representable in single or double precision, so that the very first ones coincide with those of the Taylor series...);
- we find an evaluation scheme (Horner, Estrin, in-between) for the generated polynomial. This depends much on target architecture (depth of the pipelines, etc.);
- we evaluate a bound (as tight as possible) on the roundoff error.

... and we restart from scratch if the global error (approximation+roundoff) is too large.

There are several reasons for trying to generate approximations at compile-time:

- what we frequently need is *compound functions*: for instance, we could directly generate an approximation to $\log(1 + e^{-x})$ instead of calling `exp` and `log` (hence using 2 consecutive approximations). This might result in faster and smaller code, and might also sometimes improve the accuracy;
- we can try to *specialize programs*: at compile time, some special cases (infinities, NaNs) may be known not to happen, the input domain of the function (or

a bound on that domain) may be known (which is important, the smaller the domain in which we need a polynomial approximation, the lower the degree of that approximation for a given accuracy). The rounding mode and the desired final accuracy may also sometimes be known at compile time.

II. CLASSICAL RESULTS ON POLYNOMIAL APPROXIMATION

The basic result on minimax approximation is the following theorem, due to Chebyshev.

Theorem 1 (Chebyshev): p^* is the minimax degree- n approximation to f on $[a, b]$ if and only if there exist at least $n + 2$ values

$$a \leq x_0 < x_1 < \dots < x_{n+1} \leq b$$

such that:

$$\begin{aligned} & p^*(x_i) - f(x_i) \\ &= (-1)^i [p^*(x_0) - f(x_0)] \\ &= \pm \|f - p^*\|_\infty. \end{aligned}$$

Remez's algorithm [11] consists in iteratively building the set of points x_0, x_1, \dots, x_{n+1} of Chebyshev's theorem. Although making sure that it almost always work is complicated, the rough sketch is simple:

- 1) Start from an initial set x_0, x_1, \dots, x_{n+1} in $[a, b]$.
- 2) Consider the linear system of equations

$$\begin{cases} p_0 + p_1x_0 + p_2x_0^2 + \dots + p_nx_0^n - f(x_0) &= +\epsilon \\ p_0 + p_1x_1 + p_2x_1^2 + \dots + p_nx_1^n - f(x_1) &= -\epsilon \\ p_0 + p_1x_2 + p_2x_2^2 + \dots + p_nx_2^n - f(x_2) &= +\epsilon \\ \dots & \dots \\ p_0 + \dots + p_nx_{n+1}^n - f(x_{n+1}) &= (-1)^{n+1}\epsilon. \end{cases}$$

we have $n + 2$ equations, with $n + 2$ unknowns ($p_0, \dots, p_n, \epsilon$) thus, in non-degenerated cases, we have one solution ($p_0, p_1, \dots, p_n, \epsilon$) only. Solving the linear system gives a polynomial $P(x) = p_0 + p_1x + \dots + p_nx^n$.

- 3) Compute the points y_i in $[a, b]$ where $P - f$ has its extremes, and start again (step 2), replacing the x_i 's by the y_i 's.

This algorithm exhibits a *quadratic convergence* [12], hence in general a few iterations only are necessary (but one iteration requires solving a linear system and computing the extremes of $P - f$). It is now available on PCs in tools such as Maple. For instance, on a DELL810 laptop with a 1.86GHz processor and 2 Gbytes of RAM, the degree-5 minimax approximation to $\log(1 + e^{-x})$ in $[0, 1]$ is found in less than 1.2 seconds.

And yet, the obtained approximations are not fully satisfying:

- the coefficients are “exact” coefficients, that are not exactly representable in floating-point (FP) arithmetic. One can round them to the nearest FP number, but it is very unlikely that by doing that we will get the best approximation among the ones with FP coefficients;
- the approximation error given by most tools is only an estimate. They sometimes overestimate the actual error and sometimes underestimate it.

When designing a library dedicated to a given function, these problems are easily overcome by human expertise. When trying to automatically generate the approximations this is more difficult. Also, how can we certify that the obtained result will be satisfactory?

III. VARIOUS TOOLS DESIGNED BY THE ARENAIRE TEAM

The Aenaire group research of LIP laboratory et ENS Lyon <http://www.ens-lyon.fr/LIP/Aenaire/> has designed several tools that may help to solve these problems:

- a *Remez* algorithm and a *validated infnorm* in \mathbb{C} (Chevillard, Lauter) that uses multiple precision interval arithmetic and returns certain and tight bounds on

$$\max_{[a,b]} |p(x) - f(x)|$$

- a tool that computes the best or the nearly best polynomial among the ones that satisfy constraints on the size of the coefficients (Brisebarre, Chevillard, Muller, Tisserand, Torres). To do that, we use two approaches, based on the reduction of our initial problem to
 - enumerating integer points in a polytope [2];
 - using the LLL algorithm (lattice reduction) [?].
- *Gappa* (Melquiond): a tool that computes error bounds on FP calculations, and generates formal proofs of these bounds [9];

These tools have been heavily used for building our CRLIBM library of correctly-rounded elementary functions [4], [6], [7].

A. Polynomial generation

Let us illustrate what can the polynomial generation tool do with an example used in CRLIBM, and obtained by Sylvain Chevillard and Christoph Lauter. To evaluate function *arcsin* near 1 with correct rounding, after a change of variables we actually have to compute

$$g(z) = \frac{\arcsin(1 - (z + m)) - \frac{\pi}{2}}{\sqrt{2 \cdot (z + m)}}$$

where

$0x\text{BFBC28F800009107} \leq z \leq 0x\text{3FBC28F7FFFF6EF1}$ (roughly speaking $-0.110 \leq z \leq 0.110$) and $m = 0x\text{3FBC28F80000910F} \simeq 0.110$. We want to generate a

degree-21 polynomial approximation. If we round to nearest the coefficients of the Remez polynomial, we get the error curve given in Figure 1. If we use our tool (with the LLL algorithm), we get the error curve given in Figure 2.

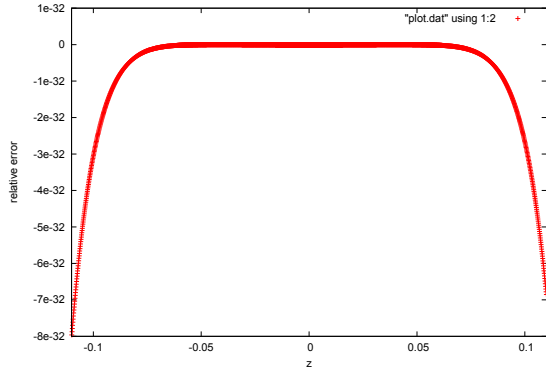


Fig. 1. Difference between $g(z)$ and the polynomial obtained by rounding to the nearest the coefficients of the Remez polynomial.

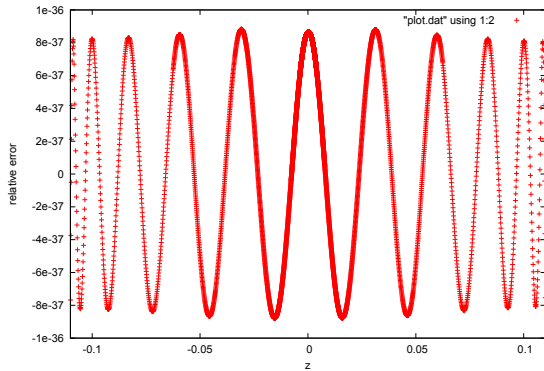


Fig. 2. Difference between $g(z)$ and the polynomial obtained by our tool.

That is, the generated polynomial is almost 10000 times more accurate than the naive rounded Remez polynomial.

B. Evaluation and validation

Once we have generated a polynomial, we have to design an evaluation scheme (using Horner’s scheme, Estrin’s scheme or something in-between). This will depend much on the target architecture (depth of the pipeline, availability of one or several FPU, availability of a fused multiply-add instruction...).

Once the evaluation scheme is known, we must compute a tight bound on the evaluation error. We will start from Melquiond’s Gappa tool [5], [9] (available at <http://lipforge.ens-lyon.fr/www/gappa/>). Gappa bounds values and rounding errors of a straight-line program, and generates a formal proof of these bounds that

can be checked using tools such as Coq or PVS. Gappa is an interactive tool (it needs hints from the user). And yet, in the very particular case of our applications (we only evaluate polynomials), we should be able to automate it.

IV. AN EXAMPLE

Consider function $\log_2(1+2^{-x})$, for $x \in [0, 1]$. We look for a polynomial approximation of degree 6, with single precision coefficients.

- the approximation error of the minimax polynomial (with real coefficients) is 8.34×10^{-10} ;
- if we round to the nearest single-precision number the coefficients of the minimax polynomial, we get an approximation error equal to 1.19×10^{-8} ;
- using our method, we get the following polynomial, whose coefficients are single-precision numbers:

$$P(x) = \frac{5750871}{137438953472} \times x^6 + \frac{13462391}{549755813888} \times x^5 - \frac{7528339}{4294967296} \times x^4 + \frac{14577171}{219902325552} \times x^3 + \frac{5814467}{67108864} \times x^2 - \frac{8388607}{16777216} \times x + 1;$$

the approximation error is less than 1.024×10^{-9} .

- assuming \log_2 and 2^x are correctly-rounded functions, the maximum error obtained by evaluating $\log_2(1+2^{-x})$ usually on 10000 samples is 8.6×10^{-8} .
- using our polynomial with Horner’s scheme, the maximum error is 7.3×10^{-8} ;

In that case, we get a slightly more accurate, and certainly much faster and smaller code.

V. CONCLUSION, AND GENERAL METHOD

In cooperation with a compiler group of ST Microelectronics, we are going to investigate the possibility of generating function approximations at compile-time, for some specific signal processing applications. The general sketch of the method will be:

- 1) determinate which functions might be interesting to directly approximate;
- 2) determinate, as sharply as possible, the input domains, using:
 - interval arithmetic;
 - “annotations” by the programmer;
 - possibly, questions to the programmer;
- 3) run the Remez algorithm to find the minimax approximations. If they are not interesting, give up, otherwise;
- 4) run our algorithms to find “real world” approximations;
- 5) choose the polynomial evaluation algorithm (Horner, Estrin...). The compiler is “aware” that it evaluates a polynomial \rightarrow must be used to find a good Estrin scheme;

- 6) run Gappa to evaluate a bound on the error (approximation+rounding errors). Gappa needs interaction, but if we restrict to polynomials, should be automated.

ACKNOWLEDGEMENTS

Nicolas Brisebarre, Sylvain Chevillard and Christoph Lauter provided some of the examples presented here, and did a great work on polynomial approximation generation. I owe them many thanks.

REFERENCES

- [1] M. Abramowitz and I. A. Stegun. *Handbook of mathematical functions with formulas, graphs and mathematical tables*. Applied Math. Series 55. National Bureau of Standards, Washington, D.C., 1964.
- [2] Nicolas Brisebarre, Jean-Michel Muller, and Arnaud Tisserand. Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software*, 32(2):236–256, June 2006.
- [3] W. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.
- [4] Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. CR-LIBM, a correctly rounded elementary function library. In *SPIE 48th Annual Meeting International Symposium on Optical Science and Technology*, August 2003.
- [5] M. Dumas and G. Melquiond. Generating formally certified bounds on values and round-off errors. In *6th Conference on Real Numbers and Computers*, pages 55–70, Schloss Dagstuhl, Germany, November 2004.
- [6] F. de Dinechin, D. Defour, and C. Lauter. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Research report 5137, INRIA, March 2004. Submitted to TOMS.
- [7] Florent de Dinechin, Christof Lauter, and Jean-Michel Muller. Fast and correctly rounded logarithms in double-precision. *Theoretical Informatics and Applications (to appear)*, 2006.
- [8] J. F. Hart, E. W. Cheney, C. L. Lawson, H. J. Maehly, C. K. Mesztenyi, J. R. Rice, H. G. Thacher, and C. Witzgall. *Computer Approximations*. Wiley, New York, 1968.
- [9] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, Ecole Normale Supérieure de Lyon, November 2006. Available at <http://www.ens-lyon.fr/LIP/Pub/PhD2006.php>.
- [10] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser Boston, 2nd edition, 2006.
- [11] E. Remez. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Académie des Sciences, Paris*, 198:2063–2065, 1934.
- [12] L. Veidinger. On the numerical determination of the best approximations in the Chebyshev sense. *Numerische Mathematik*, 2:99–105, 1960.