A Domain Splitting Algorithm for the Mathematical Functions Code Generator

Olga Kupriianova, Christoph Lauter

A Domain Splitting Algorithm for the Mathematical Functions Code Generator

Olga Kupriianova
Sorbonne Universités, UPMC Univ Paris 06,
UMR 7606, LIP6, F-75005, Paris, France
Email: olga.kupriianova@lip6.fr

Christoph Lauter
Sorbonne Universités, UPMC Univ Paris 06,
UMR 7606, LIP6, F-75005, Paris, France
Email: christoph.lauter@lip6.fr

***Abstract*—The general approach to mathematical function implementation consists of three stages: argument reduction, approximation and reconstruction. The argument reduction step is needed to reduce the degree of the approximation polynomial and to simplify the error analysis. For some particular functions (e.g. $\exp$) it is done using its algebraic properties. In the general case the whole domain is split into small subdomains to get low-degree approximation on each of them. Here we present a novel algorithm for the domain splitting that will be integrated soon to Metalibm code generator.**

I. Introduction

A call to a mathematical library (`libm`) is performed each time we evaluate a mathematical function at some point in some programming language. The standard `libms` are limited: they support a limited set of functions with one manually coded implementation for each of them. The functions in `libm` may be too precise: if only 40 bits of accuracy are needed, all 53 bits of double precision have to be computed and then the result is rounded. Handling NaNs and infinities as an input may be a waste of time on large amounts of experimental data when the domains for function evaluation are small and known beforehand. Thus, a modern `libm` ought to contain mathematical functions implementations in different variations (we call them flavors). So, the different variations of function implementation may come from the different result accuracies (correctly rounded, 50 correct bits, 35 correct bits, etc.) or various input domains.

As the quantity of all possible flavors is huge, it is not feasible to manually reimplement a `libm` in a more flexible way. That is why we propose to generate code for each flavor automatically instead of implementing it manually [1], [2]. Besides the flavor implementations our generator (Metalibm) certifies that the error in the produced code is not larger than the target error from flavor specification [3]. The standard `libms` provide implementations of elementary functions and several special functions like Gamma, erf, while sometimes there is a need of more "exotic" functions like Airy function or Voigt profile. As our generator takes a function to implement as a parameter, it can produce the code even for these specific functions, if there is a mean to evaluate this function and its few derivatives over an interval with an arbitrary accuracy.

The mathematical functions are manually implemented within one scheme: argument reduction, approximation and reconstruction. We use polynomial approximations with a bounded degree: we add a parameter maximum degree d_{max} in a flavor's specification. As low-degree polynomials approximate a function well on a small domain only, we have somehow to reduce the implementation domain $I = [a; b]$ to a smaller one $[\alpha; \beta]$. There are methods of argument reduction based on algebraic properties of the function [4], [5], [6], but in general case for functions like $\mathrm{asin}, \mathrm{erf}$ or any other black-box function none of such properties may be applied. In this case to reduce the argument domain splitting is used: we split the domain into non-overlapping parts $I_0, \ldots, I_k$ and build a low-degree polynomial on each of the parts. The reconstruction step is a transition from the small domain to the initial one that allows us to evaluate function on the large domain from the flavor specification. If domain splitting was performed on the first step in the reconstruction procedure we have to determine to which of the subdomains $I_0, \ldots, I_k$ belongs the input $x \in I$ which is usually done with branching.

In this paper we present a novel algorithm for domain splitting that is now implemented in Metalibm code generator. To illustrate our algorithm we use the same function flavor example in the whole paper: we generate the code for asin function on a domain $I = [0; 0.85]$ with target error $\bar{\varepsilon} = 2^{-52}$ and a maximum degree bound $d_{max} = 8$. Other examples can be found in Section V.

This paper is organized as follows: in Section II we observe splitting examples, we notice the need of more sophisticated algorithm that gives a good split. Section III contains useful techniques from numerical analysis that are in a base of new splitting algorithm. Section IV gives an idea and pseudo-code of two algorithms based on the theory from the previous section, Section V contains the results for different splitting methods and different function flavors and Section VI concludes the whole work.

II. State of the art

The simplest way to perform the domain splitting is to take some large k and split the domain I into k equal parts. For the mentioned example we may take $k = 50$, the diagram of the corresponding degrees may be found on Fig. 1. This approach works but produces too many subdomains and as there is head-room between d_{max} and the real polynomial degrees the splitting can be improved.

Instead of a uniform splitting one can use the splitting algorithm that takes into account the function behavior, for example bisection (Fig. 2) that gives 23 subdomains. Even if we reduce the number of subdomains k in the uniform

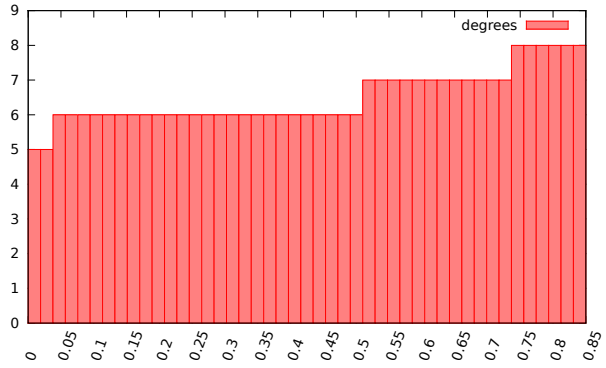

Figure 1. Naive k-equal split for asin function. Polynomial degrees on the domains

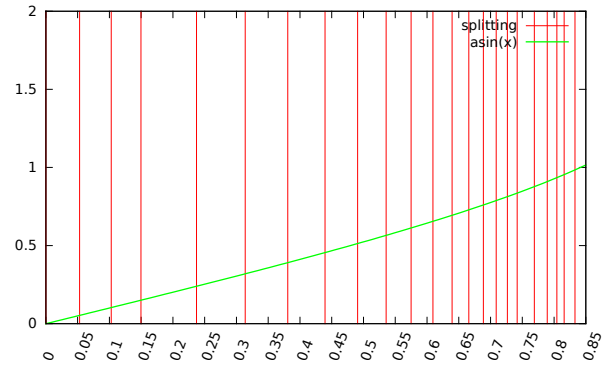

Figure 2. Bisection splitting for asin function.

splitting, the least number in the uniform splitting is 45, so bisection procedure saves memory.

However, a naive bisection procedure may be improved: on some of the subdomains there is still head-room between actual polynomial degree and d_{max} and in some cases the actual degrees on the adjacent subdomains differ too much, so the diagram of the degrees is not regular (see Fig. 3).

We need some theoretical background to perform an optimized splitting that regularizes the diagram of the polynomial degrees on each of the subdomains as much as it can.

III. Theoretical background

It was mentioned that that the function implementations use approximations and our code generator computes polynomial approximations. In this Section we consider the theoretical base to find or to estimate the best polynomial approximation.

A. Minimax polynomials

1) Definition: There are a lot of methods to compute a polynomial approximation, but the most accurate result is obtained by computing a minimax polynomial. The minimax polynomial p for a function f on a given interval I minimizes the approximation error

$$\tilde{\varepsilon} = \|f - p\|_\infty^I = \max_{x \in I} |f - p|$$

among all the polynomials of a given degree d. The same is applicable for relative error. Remez algorithm [7] with a small modification is used to find a minimax polynomial [8]. The classical algorithm produces real coefficients and rounding them to floating-point numbers yields to loss of accuracy. The algorithm proposed in [8] and implemented in Sollya [**?**] finds a minimax polynomial among all the polynomials with floating-point coefficients.

2) Remez algorithm: Remez algorithm has quadratic convergence to a minimax polynomial when the function f is twice differentiable and with additional conditions for approximation points x_i. We do not explain here the whole algorithm, we just give an idea. It is an iterative algorithm and first $n+2$ points $x_0, \ldots, x_{n+1}$ from $[a, b]$ has to be chosen. Then in a loop there are four actions repeated until the needed approximation accuracy is reached. First, an approximation p of f has to be built on the chosen $n+2$ points. Then, to compute the current accuracy, we have to compute $\varepsilon = \max_{i=0,\ldots,n+1} |p(x_i) - f(x_i)|$, we compute or estimate the value of $\|p - f\|_\infty$, take another set of $n+2$ points and repeat the loop. On the first step Chebyshev nodes are often chosen as the set of $n+2$ points:

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{(n+1-i)\pi}{n+1}\right), \; i = 0, \ldots, n+1.$$

It is an expensive algorithm, we have to perform a lot of function evaluations, compute infinite norms and make comparisons. The infinite norm is computed with the algorithm from [9]. For the example of asin Remez approximation procedure makes about 9000 function evaluations for each of the subdomains.

B. Theorem of de la Vallée-Poussin

However, to know error bounds for the best polynomial approximation, it is not always mandatory to compute this approximation itself. We may skip several computation steps estimating the bounds for the approximation error as it will be shown later.

1) Chebyshev nodes and the bounds for approximation error:

Theorem 1 (of de la Vallée-Poussin)**.** *Let be f a continuous function $f \in C_{[a,b]}$, p its approximation polynomial on $n+2$ points $x_0 < x_1 < \cdots < x_{n+1}$ from $[a, b]$ such that the error*

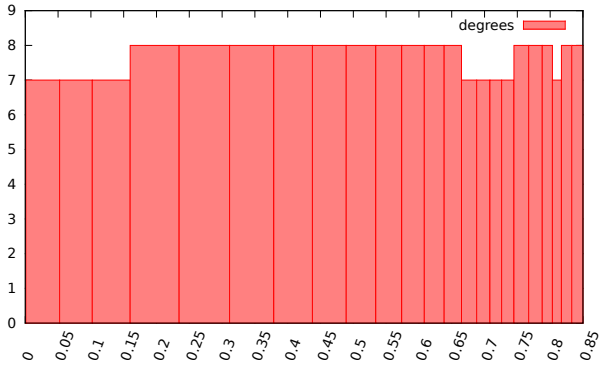

Figure 3. Polynomial degrees for bisection splitting of function asin on domain $[0; 0.75]$

$f - p$ has a local extremum and its sign alternates between two successive points x_i, then the optimal error μ verifies

$$\min_{i=0,1\ldots,n+1} |f(x_i) - p(x_i)| \le \mu \le \max_{i=0,1\ldots,n+1} |f(x_i) - p(x_i)|.$$

The mentioned approximation points x_i for polynomial p may be chosen as Chebyshev's nodes [10]: from the alternation Chebyshev theorem in this case the approximation error oscillates perfectly between its extrema at least $n+2$ times. Theorem of de la Vallée-Poussin takes a polynomial p with an error oscillating $n+2$ times and claims that the quality of the approximation p is related to the quality of the oscillations [11].

This theorem allows us to check the quality of the approximation: polynomial p is considered as the best approximation if ε and $\|f - p\|_\infty$ are sufficiently close.

As Remez is an iterative algorithm, on each step we may check where is the current approximation error relatively to the optimal error. This theorem allows to write a procedure `checkIfSufficientDegree` that checks if it is possible to compute a polynomial of degree d that approximates a function f on an interval I with error ε. You may find a pseudo-code for the mentioned procedure below. It starts with computation of Chebyshev approximation polynomial of degree d_{max} and then it obtains the bounds for optimal approximation (from de la Vallée-Poussin theorem). When the target accuracy is larger than the upper bound for the approximation error, the method returns true, when the target accuracy is lower than the lower bound, it returns false. In the case when the target accuracy is between the bounds, it is not clear and a Remez iteration is needed.

Procedure `checkIfSufficientDegree`(f, I, d, $\overline{\varepsilon}$):
Input : function f, domain $I = [a; b]$, max. degree d, target accuracy $\overline{\varepsilon}$
Output: true in the case of success, false in the case of fail
$chebNodes \leftarrow$ computeChebyshevNodes(I, d_{max});
$p \leftarrow$ computeApproximationOnChebyshevNodes(f, chebNodes, d_{max});
$m \leftarrow \min_{x_i \in chebNodes} |f(x_i) - p(x_i)|$;
$M \leftarrow \max_{x_i \in chebNodes} |f(x_i) - p(x_i)|$;
if $\overline{\varepsilon} \ge M$ **then** $result$ = true;
if $\overline{\varepsilon} \le m$ **then** $result$ = false;
if $\overline{\varepsilon} > m$ *and* $\overline{\varepsilon} < M$ **then**
  $p \leftarrow$Remez$(f, I, d_{max}, \overline{\varepsilon})$;
  $\delta \leftarrow supnorm(f - p, I)$;
  $result \leftarrow \delta \le \overline{\varepsilon}$;
end
return *result*;

This procedure is useful to compute domain splitting based on bisection procedure.

C. The base for a new splitting algorithm

The upper-mentioned theory gives a base for a domain splitting algorithm. We know how to solve a non-standard approximation problem: for a given function F on an interval I compute a polynomial of an unknown degree d that approximates f on I with an error $\varepsilon, \varepsilon \le \tilde{\varepsilon}$. We may use la Vallée-Poussin theorem to compute the splitting. We will need to compute a polynomial approximation p in Chebyshev nodes and check the infinite norm of p against target error. This can be done with `checkIfSufficientDegree` procedure.

IV. A new splitting algorithm

A. Bisection

The first improvement of the linear split on k equal subdomains is bisection. There is a set of parameters for the algorithm: a function f, needed approximation error $\tilde{\varepsilon}$, minimal width of the subdomain w_{min}, maximum bound for the polynomial degree d_{max} and the domain I. The algorithm returns a list of split points.

So, we start to check if it is possible to approximate the function f on the whole domain by a minimax polynomial of degree d_{max} with the error bounded by $\tilde{\varepsilon}$. If `checkIfSufficientDegree` returns true than the splitting is computed and the empty list has to be returned. If the checking procedure returned false, we have to split the interval I into two equal non-overlapping parts I_{left} and I_{right} and to repeat it recursively for the left part. We continue to bisect the current interval until `checkIfSufficientDegree` returns true for all the parameters and current interval I_{left}. In this case we append $m = \sup(I_{left})$ to the list of split points and repeat the procedure for the rest of the initial interval, i.e. $[m, b]$. The algorithm returns error if size of currently checked interval is less than w_{min}.

Here is the pseudo-code for the bisection splitting. It uses previously explained procedure `checkIfSufficientDegree` for an interval, if it returns false, it splits interval into two equal parts. The procedure is repeated recursively.

Procedure `computeOptimizedSplitting`(f, I, d, $\overline{\varepsilon}$):
Input : function f, domain $I = [a; b]$, max. degree d, target accuracy $\overline{\varepsilon}$
Output: list ℓ of points in I where domain needs to be split
if `checkIfSufficientDegree`*(f, I, d, $\overline{\varepsilon}$)* **then return**
$\ell = [\,]$;
$m \leftarrow b$;
while *not* `checkIfSufficientDegree`*(f, $[a; m]$, d, $\overline{\varepsilon}$)*
do $m \leftarrow (a + m)/2$;
$J \leftarrow [m; b]$;
$\ell \leftarrow$**prepend**(m, `computeOptimizedSplitting`(f, J, d, $\overline{\varepsilon}$));
return ℓ;

Thus, the algorithm returns only splitting points inside the initial interval I, the resulting list does not contain its borders a, b.

For the asin example bisection splits the domain into 23 subdomains and the degrees diagram is on the Fig. 3. The other examples can be found in Section V.

B. Improved bisection

Bisection produces less intervals than the naive linear approach, but it is still not optimal: some intervals may be merged together to reduce the headroom between d_{max} and actual polynomial degree. The improved version of splitting is

based on the bisection, but then, as soon as we find a suitable interval on the left, we try to move its right border on some δ. And then we repeat for the rest of the initial interval. This algorithm contains of two procedures: bisection and enlarging. The only difference with the previously described classical bisection is in enlarging procedure: as soon as we find a leftmost suitable interval, we try to move its right border. Then this moved right border is added into a list of split points.

Procedure `computeOptimizedSplitting`(f, I, d, $\overline{\varepsilon}$):
Input : function f, domain $I = [a; b]$, max. degree d, target accuracy $\overline{\varepsilon}$
Output: list ℓ of points in I where domain needs to be split
if `checkIfSufficientDegree`(f, I, d, $\overline{\varepsilon}$) **then return** $\ell = [\,]$;
$m \leftarrow b$;
while *not* `checkIfSufficientDegree`(f, $[a; m]$, d, $\overline{\varepsilon}$) **do** $m \leftarrow (a + m)/2$;
$s \leftarrow$`enlargeDomain`(f, $[a; m]$, $[m; b]$, $\overline{\varepsilon}$, d) ;
$\ell \leftarrow$**prepend**(s, `computeOptimizedSplitting`(f, $[s; b]$, d, $\overline{\varepsilon}$));
return ℓ;

Procedure `enlargeDomain`(f, I, J, $\overline{\varepsilon}$, d):
Input : function f, domain $I = [a; b]$, remaining domain $J = [b; c]$, $\overline{\varepsilon}$, d
Output: optimal split point location $s \in J$
$\delta \leftarrow (b - a)/3$;
while $\delta > \overline{\delta}$, $\overline{\delta}$ *a constant, and* $b < c$ **do**
  $s \leftarrow b + \delta$;
  while `checkIfSufficientDegree`(f, $[a; s]$, d, $\overline{\varepsilon}$) **do** $s \leftarrow s + \delta$;
  $s \leftarrow b - \delta$;
  $\delta \leftarrow \delta/2$;
end
return s;

For the asin example improved bisection method produces 21 subdomains, Fig. 4 shows the corresponding polynomial degrees diagram. The degrees on 20 of the intervals are equal to 8, and only on the last small interval the obtained degree is 6. Other examples can be found in Section V.

1) Left-to-right and right-to-left directions: As on each step of the algorithm we try to enlarge the leftmost suitable interval, we may have a situation when the degrees on the first intervals are close to d_{max}, but on the last one (or even several last intervals) corresponding polynomial degree is small. A similar algorithm may be obtained, when instead of the leftmost suitable intervals we take the rightmost suitable intervals. In the first case we compute the split points from left to right, in the second case from right to left. For right-to-left direction we enlarge the intervals from bisection procedure by moving their left borders down. In this case we may have low degrees on several first intervals, while on the other intervals the degree is close to d_{max}.

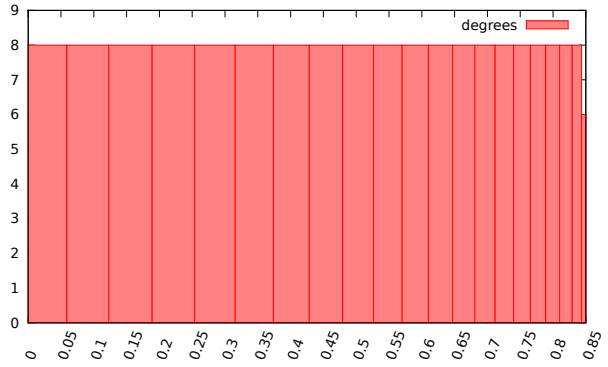

Figure 4. Polynomial degrees for improved bisection splitting for asin example.

V. Results

For the results we provide here a table with different measurements (in rows) for several different flavors (in columns). We do not provide here performance measurements: we gain in memory consumption with the improved bisection method while performance stays the same. We do not give the results for the uniform splitting here neither: simple bisection procedure splits better. In Table I there are the flavor specifications and then in Table II there are results of the domain splitting procedures for these flavors.

VI. Conclusions

As it was mentioned, domain splitting is connected with reconstruction procedure. When we try to evaluate function at some point $x \in [a, b]$, we have to take the right polynomial. To do that the corresponding interval I_k that contains the input x has to be determined first. It is done with several `if-else` statements. However, one of the goals for Metalibm was generation of vectorizable code, which means that the branching has to be avoided.

In [12] a method to avoid branching in reconstruction is proposed. The main idea is in computing an interpolation polynomial on the split points. However, the polynomial computed by technique from [12] does not always allow us to avoid branching. However, there may be no need to compute an optimal splitting if it makes vectorizable reconstruction impossible. A new step in development of splitting algorithm is searching for this connection with reconstruction and implementing a compromise between splitting and reconstruction.

In this paper we have used numerical analysis results (theorems) to solve a non-standard approximation problem.

name	function f	target accuracy	domain I	degree bound
f_1	asin	$\overline{\varepsilon} = 2^{-52}$	$I = [0, 0.75]$	$d_{max} = 8$
f_2	asin	$\overline{\varepsilon} = 2^{-45}$	$I = [-0.75, 0.75]$	$d_{max} = 8$
f_3	erf	$\overline{\varepsilon} = 2^{-51}$	$I = [-0.75, 0.75]$	$d_{max} = 9$
f_4	erf	$\overline{\varepsilon} = 2^{-45}$	$I = [-0.75, 0.75]$	$d_{max} = 7$
f_5	erf	$\overline{\varepsilon} = 2^{-431}$	$I = [-0.75, 0.75]$	$d_{max} = 6$

Table I. Flavor specifications

measure	f_1	f_2	f_3	f_4	f_5
subdomain qty in bisection	24	15	9	12	39
subdomain qty in improved bisection	18	10	5	8	25
subdomains saved	25%	30%	44%	30%	36%
coefficients saved	42	31	27	24	79
memory saved (bytes)	336	248	216	192	632

Table II. Table of measurements for several function flavors

Theorem of de la Vallée-Poussin is the base of the implemented domain splitting algorithms. It allows us to compute a bisection splitting, that produces less subdomains than the simplest uniform splitting. However, it is possible to even improve bisection splitting. The new bisection-based splitting saves us about 30% subdomains, so reduces the memory consumption as well. The new algorithm is now integrated to Metalibm. As the splitting and reconstruction steps are connected, this connection has to be identified.

REFERENCES

[1] O. Kupriianova and C. Q. Lauter, "Metalibm: A mathematical functions code generator," in *Mathematical Software - ICMS 2014 - 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings*, 2014, pp. 713–717. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44199-2_106

[2] N. Brunie, F. De Dinechin, O. Kupriianova, and C. Lauter, "Code generators for mathematical functions," Tech. Rep., Nov. 2014. [Online]. Available: http://hal.upmc.fr/hal-01084726

[3] F. de Dinechin, C. Q. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using gappa," *IEEE Trans. Computers*, vol. 60, no. 2, pp. 242–253, 2011. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TC.2010.128

[4] P. T. P. Tang, "Table-driven implementation of the exponential function in ieee floating-point arithmetic," *ACM Trans. Math. Softw.*, vol. 15, no. 2, pp. 144–157, Jun. 1989. [Online]. Available: http://doi.acm.org/10.1145/63522.214389

[5] ——, "Table-driven implementation of the logarithm function in IEEE floating-point arithmetic," *ACM Trans. Math. Softw.*, vol. 16, no. 4, pp. 378–400, 1990. [Online]. Available: http://doi.acm.org/10.1145/98267.98294

[6] ——, "Table-driven implementation of the expm1 function in IEEE floating-point arithmetic," *ACM Trans. Math. Softw.*, vol. 18, no. 2, pp. 211–222, 1992. [Online]. Available: http://doi.acm.org/10.1145/146847.146928

[7] E. Remez, *Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation.* Académie des Sciences, Paris, 1934.

[8] N. Brisebarre and S. Chevillard, "Efficient polynomial l-approximations," in *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007), 25-27 June 2007, Montpellier, France*, 2007, pp. 169–176. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ARITH.2007.17

[9] S. Chevillard, M. Joldes, and C. Q. Lauter, "Certified and fast computation of supremum norms of approximation errors," in *19th IEEE Symposium on Computer Arithmetic, ARITH 2009, Portland, Oregon, USA, 9-10 June 2009*, 2009, pp. 169–176. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ARITH.2009.18

[10] E. W. Cheney, *Introduction to approximation theory*. New York, N.Y. Chelsea, 1982. [Online]. Available: http://opac.inria.fr/record=b1091261

[11] S. Chevillard, "Évaluation efficace de fonctions numériques – Outils et exemples," Ph.D. dissertation, École Normale Supérieure de Lyon, Lyon, France, 2009.

[12] O. Kupriianova and C. Lauter, "Replacing branches by polynomials in vectorizable elementary functions," in *Book of abstracts for 16th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, 2014.