Metalibm: A Mathematical Functions Code Generator

Olga Kupriianova, Christoph Lauter

Metalibm: a Mathematical Functions Code Generator

Olga Kupriianova and Christoph Lauter

Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005 Paris, France
{olga.kupriianova, christoph.lauter}@lip6.fr,
http://www.kupriianova.info/, http://www.christoph-lauter.org/

Abstract. There are several different libraries with code for mathematical functions such as exp, log, sin, cos, etc. They provide only one implementation for each function. As there is a link between accuracy and performance, that approach is not optimal. Sometimes there is a need to rewrite a function's implementation with the respect to a particular specification.
In this paper we present a code generator for parametrized implementations of mathematical functions. We discuss the benefits of code generation for mathematical libraries and present how to implement mathematical functions. We also explain how the mathematical functions are usually implemented and generalize this idea for the case of arbitrary function with implementation parameters.
Our code generator produces C code for parametrized functions within a known scheme: range reduction (domain splitting), polynomial approximation and reconstruction. This approach can be expanded to generate code for black-box functions, e.g. defined only by differential equations.

Keywords: code generation, elementary functions, mathematical libraries

1 Introduction

Each time we evaluate mathematical functions in some programming language a corresponding function from a mathematical library (libm) is called. There are several examples of existing libms: glibc libm [1], crlibm by ENS-Lyon [2], libmcr by Sun[1], libultim by IBM[2], etc. They differ not only by the developer company or supported language but also by final accuracy. The common fact for all the versions is that they provide only one manually coded implementation of each supported function and precision. As the codes for elementary functions are used in various applications, the implementations are done for the widest possible domain, for maximum possible accuracy, etc. However, such generalization complicates the algorithms and leads to poor performance: for some particular tasks there is no need to compute a precise result. For example, most physical measurements have only small number of digits after the decimal point, so

[1] http://www.math.utah.edu/cgi-bin/man2html.cgi?/usr/local/man/man3/libmcr.3
[2] http://www.math.utah.edu/cgi-bin/man2html.cgi?/usr/local/man/man3/libultim.3

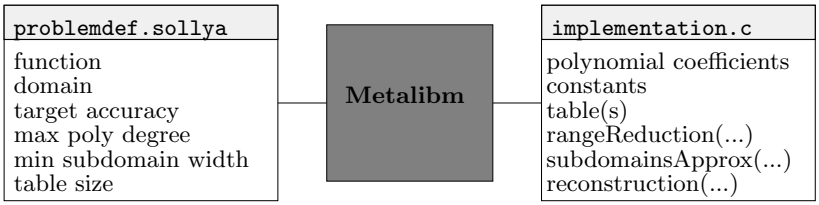

Fig. 1. Metalibm scheme

computation of the result with 53 mantissa bits (about 15 decimal digits) is a waste of time. Another example is a small domain known beforehand, which means that that the result is a finite number (no overflows/underflows are possible). In this case, handling the special values like NaNs (Not-A-Number [3]) or infinities in the beginning of the implementation can be skipped, hence the implementation gets faster.

There is also a link between speed and accuracy, so when we process big amounts of data and the needed accuracy is only about several bits, there is no reason to compute a precise result [4].

So, there is a growing need to provide several different versions (flavors) for each libms function. Manual implementation of different function flavors is almost impossible due to the quantity of all the possible parameters and coding time. Thus, we propose to write a code generator that produces parametrized implementations. The name of this prototype is Metalibm[3]. Besides producing different versions of standard libm functions, Metalibm generates implementations for composite functions as well.

The paper is organized as following: in Section 2 we explain in general the generation of "black-box" parametrized functions, Section 3 explains the workflow of the generator, in Section 4 we show how to detect a known type of function, and finally Section 5 shows the importance of the Metalibm, future work on the project and its possible application.

2 A Black-box Function Generator

The Metalibm code is a collection of scripts written in Sollya[4], a software tool for safe floating-point (FP) development with plenty of rigorous numerical algorithms [5].

Among the existing versions of mathematical libraries we are mostly interested in improvement of the current `glibc libm`. It runs on all *nix-powered machines (from supercomputers to mobile phones), so tends to be the most used one. Thus, Metalibm generates implementations in C. For each input set of parameters Metalibm generates the C code within the same scheme. On Fig. 1 there is an illustration of Metalibm routine. Metalibm computes and stores the

[3] http://lipforge.ens-lyon.fr/www/metalibm/
[4] http://sollya.gforge.inria.fr/

constants with the needed accuracy, and as we use table-driven methods [4], it computes and stores the tables. Then Metalibm generates the code for range reduction (domain splitting), polynomial approximation on each of the small domains, and the final reconstruction procedure. The purpose of these procedures is explained later in the paper.

As all the computations for the code generation are done in Sollya, a function to be implemented can be considered as black-box. On the step of range reduction we need to evaluate the function in some values. Sollya provides elementary functions and theirs combinations. The generation of code for some "exotic functions" like functions purely defined by differential equations (e.g. Dickman's) gets possible as soon as we have a corresponding Sollya implementation, even if it is bound to Sollya only dynamically [6].

Precision of all the constants and accuracy of the interim computations are specially selected in order to obtain the final result of the specified accuracy (target accuracy parameter). Besides the generated code, Metalibm verifies the final results accuracy and generates a Gappa proof [7].

A simple example of the parametrization file is provided in Listing 1.1. The sense of all the parameters will be explained later with the implementation details of Metalibm.

```
1  f = exp(x); // we want to get the code for exp(x)
2  dom = [−70, 70]; // on domain [−70, 70]
3  target = 2^−42; // the final error has to be not more that 2^−42
4  maxDegree = 5; // degree of approximating polynomials is not more than 5
5  minWidth = (sup(dom) − inf(dom)) ∗ 1/4096; // minimal size of the subdomain
6  tableIndexWidth = 5; // the size of table index is 5 bits, so 32 entries in table
```

Listing 1.1. Example of the parametrization file

3 Function Generation Workflow

In order to implement a mathematical function on a given domain we have to care first about special cases (NaNs and infinities). So, the first or even precomputing step is always filtering the special inputs, handling exceptions, too large or too small inputs unless the domain is so small that these special cases cannot occur.

When designing algorithms for mathematical functions evaluation we usually start with writing the inputs and output in a form of floating-point (FP) numbers $2^E m$ and trying to separate all the factors into two groups: a power of two $n = 2^E$ that represents the results exponential part and one non-integer number with values in a small range to represent significand. This technique of emphasizing a non-integer part with a small range (results significand) is usually called range reduction [8]. The next step is building an approximation. There are different approximation techniques, but we only consider polynomial ones. The larger the range of argument is, the higher polynomial degrees are required. This means that the computation time will be high and the error analysis becomes more

difficult with the growing number of FP operations. This is the main reason why we need argument reduction. In Metalibm, we use a parameter `maxDegree` to limit the maximal degree of constructed approximations. These polynomials are computed with Remez algorithm that is implemented in Sollya [6], [9].

All the transformations on the first step fully depend on mathematical properties of the implemented function, i.e. $e^{a+b} = e^a e^b$ [10]. It can happen that it is impossible to reduce the range using only mathematical properties. In this situation the required domain for the implementation is divided into subdomains. On each of the subdomains the argument range is small and the degree of the approximating polynomial gets lower. In this case, one has to build approximations for each of the subdomains and then the reconstruction step gets more complex. In Metalibm we bound minimal size of subdomain by a parameter `minWidth`.

Once the generation process is launched, Metalibm checks whether it is possible to build a polynomial for the specified function, domain and other parameters. If it is not possible, it tries to reduce argument for the set of known functions (exponential, logarithm, periodic). Then there is symmetry detection and expression decomposition for composite functions. When it is still not possible to build a polynomial approximation for the reduced domain, domain splitting is performed. Metalibm adapts computational precision [11] in order to get the needed accuracy, so in some cases it uses double-double or triple-double arithmetic [12], [13].

4 Some Technical Details

As it was already mentioned, Metalibm tries to detect some known properties of the function to perform range reduction. For example, let us have a look on how Metalibm tests hypothesis that the function is exponential and finds its base.

In order to detect the exponential function and to perform the appropriate argument reduction we accept the hypothesis that the function has a form of $f(x) = b^x$ on the implementation domain $\mathcal{I}$ with the unknown base b. It means that we can determine the base from the following:

$$b = \exp\left(\frac{\ln f(x)}{x}\right) = const \ \forall x \in \mathcal{I}.$$

The base b can be computed in Sollya without any information about the function $f(x)$. Sollya will provide the needed value for the function. This is why we were talking about "black-box" functions: we do not know what code are we generating, but we have a mean to evaluate some function values.

As we have accepted the function type as $f(x) = b^x$, it means that for another argument x_1 from the implementation domain the function value is $f(x_1) = b^{x_1}$. Of course the base b stays the same if the hypothesis was correct and we know that we can perform the exponential argument reduction now. If the value $\tilde{\varepsilon} = \left\| \frac{b^x}{f(x)} - 1 \right\|_\infty^{\mathcal{I}}$ is sufficiently small we accept the initial hypothesis.

The detection of other types of functions is done in the analogous way.

5 Conclusions

The Metalibm code generator is still under development, but it already produces code for basic functions. The next goals are an optimized domain splitting procedure, producing vectorizable implementations and addition of range reduction for other functions.

As it was told, the generated code tends to improve and expand current `glibc libm`. So, we will try to provide generated code on the whole code generator to the GNU community for integration with the `glibc libm`.

References

1. S. Loosemore, R. M. Stallman et al., *The GNU C Library Reference Manual* for version 2.19, Free Software Foundation, Inc.
2. C. Daramy-Loirat, D. Defour, F. de Dinechin et al. *CR-LIBM. A library of correctly rounded elementary functions in double-precision*, user's manual, 2013.
3. IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008, August 2008.
4. J.-M. Muller, *Elementary Functions: Algorithms and Implementation*, Birkhauser Boston, Inc., 1997, Secaucus, NJ, USA.
5. S. Chevillard and M. Joldeş and C. Lauter, *Sollya: An Environment for the Development of Numerical Codes*, Mathematical Software - ICMS 2010.
6. S. Chevillard, Ch. Lauter, M, Joldeş, *Users manual for the `Sollya` tool*, Release 4.0, May 2013.
7. F. de Dinechin, Ch. Lauter and G. Melquiond, *Certifying the Floating-Point Implementation of an Elementary Function Using Gappa*, IEEE Transactions on Computers, IEEE Computer Society, 2011, 60, 242-253.
8. J.-M. Muller, N. Brisebarre, F. de Dinechin et al., *Handbook of Floating-Point Arithmetic*, Birkhäuser, 2010.
9. S. Chevillard, *Évaluation efficace de fonctions numériques. Outils et exemples*, PhD Thesis, ENS de Lyon, 2009.
10. P. T. P. Tang, *Table-driven implementation of the exponential function in IEEE floating-point arithmetic*, ACM Transactions on Mathematical Software, 18(2):211222, june 1989.
11. Ch. Lauter, *Arrondi correct de fonctions mathématiques*, PhD Thesis, ENS de Lyon, 2008.
12. J. R. Schewchuk, *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*, Discrete & Computational Geometry, 1997.
13. F. de Dinechin, D. Defour, Ch. Lauter, *Fast correct rounding of elementary functions in double precision using double-extended arithmetic*, Research report, ENS de Lyon, 2004.