Metalibm

Olga Kupriianova

Sorbonne Universités, UPMC Univ Paris 06,
UMR 7606, LIP6,
F-75005 Paris, France

Rencontres Arithmétiques de l’Informatique Mathématique
7 April 2015
Outline

1. Introducing libm
2. Problem statement
3. The Metalibm Project
4. General approach to function implementation
Mathematical libraries

The libm

- gives a set of maths functions (exp, log, sin, cos, pow, ...)
- supports important precisions (float, double, long double)

The existing implementations

- glibc libm
- libmcr by Sun\textsuperscript{a}
- crlibm by ENS-Lyon
- libultim by IBM
- Intel’s proprietary mathimf

\textsuperscript{a} All other trademarks and copyrights are the property of their respective owners
The glibc libm

Why glibc libm?
The glibc libm

Why glibc libm?

- Math library running on all linux-powered machines
Why glibc libm?

- Math library running on all linux-powered machines
- Written by different developer teams
- Some codes were written 20 years ago
- Strange naming conventions
Routine halfulp(double x, double y) computes $x^y$ where
result does not need rounding. If the result is closer to 0 than can be represented it returns 0.
In the following cases the function does not compute anything and returns a negative number:
1. if the result needs rounding,
2. if $y$ is outside the interval \([0, 2^{20}−1]\),
3. if $x$ can be represented by $x=2^{\ast\ast}n$ for some integer $n$. 
The restrictions of libm

- Limited set of functions: trigonometric, exponentiation and logarithmic, hyperbolic, special functions
- Limited set of precisions (float, double, long double)
- The input range: “as far as the input and output formats allow”
- The only one implementation, hard-coded long time ago
- No special code for exp(x) on \([-10; 10]\) with 42 bits of accuracy
Common wisdom

The more accurate you compute, the more expensive it gets
Speed vs. accuracy compromise

Common wisdom
The more accurate you compute, the more expensive it gets

Observation
The binary64 format provides about 16 decimal digits, while physical measurements might use only 3!
Speed vs. accuracy compromise

Common wisdom
The more accurate you compute, the more expensive it gets

Observation
The binary64 format provides about 16 decimal digits, while physical measurements might use only 3!

Compiler flag?
Why can’t we have an option like

gcc -c code.c -fp-transcendental=15ulps

to get less accuracy when we don’t need more
Outline

1. Introducing libm
2. Problem statement
3. The Metalibm Project
4. General approach to function implementation
Flavors of functions

**Flavor**

A flavor is a particular specification of a function.

**Absolute error in ulps**

\[ |X - x| \leq \alpha \cdot \text{ulp}(x) \]

**Relative Error**

\[ \left| \frac{X}{x} - 1 \right| \leq \alpha \cdot \beta^{-p+1} \]

**Correct Rounding**

\[ |X - x| \leq 0.5 \cdot \text{ulp}(x) \]
Table Maker’s Dilemma

Sometimes **huge** accuracy is needed
Runtime of function is not bounded or really huge
Table Maker’s Dilemma

Sometimes **huge** accuracy is needed
Runtime of function is not bounded or really huge

**Recommendations**

- Precompute TMD worst cases
- Avoid correct rounding wherever we do not really need it
### Different flavors of the functions

<table>
<thead>
<tr>
<th>Precision</th>
<th>Ulp Error</th>
<th>Decimal Digits</th>
<th>Needed Accuracy</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>1 ulp</td>
<td>7</td>
<td>$2^{-24}$</td>
<td>fast</td>
</tr>
<tr>
<td>Single (CR)</td>
<td>0.5 ulps</td>
<td>7</td>
<td>$2^{-50}$</td>
<td>slow</td>
</tr>
<tr>
<td>Double</td>
<td>1 ulp</td>
<td>16</td>
<td>$2^{-53.5}$</td>
<td>fast</td>
</tr>
<tr>
<td>Double (CR)</td>
<td>0.5 ulps</td>
<td>16</td>
<td>$2^{-150}$</td>
<td>slow</td>
</tr>
</tbody>
</table>
Different flavors of the functions

<table>
<thead>
<tr>
<th>Precision</th>
<th>Ulp Error</th>
<th>Decimal Digits</th>
<th>Needed Accuracy</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>1 ulp</td>
<td>7</td>
<td>$2^{-24}$</td>
<td>fast</td>
</tr>
<tr>
<td>Single (CR)</td>
<td>0.5 ulps</td>
<td>7</td>
<td>$2^{-50}$</td>
<td>slow</td>
</tr>
<tr>
<td>Double</td>
<td>1 ulp</td>
<td>16</td>
<td>$2^{-53.5}$</td>
<td>fast</td>
</tr>
<tr>
<td>Double (CR)</td>
<td>0.5 ulps</td>
<td>16</td>
<td>$2^{-150}$</td>
<td>slow</td>
</tr>
</tbody>
</table>

Not fast enough? Still more flavors
Different flavors of the functions

<table>
<thead>
<tr>
<th>Precision</th>
<th>Ulp Error</th>
<th>Decimal Digits</th>
<th>Needed Accuracy</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>1 ulp</td>
<td>7</td>
<td>$2^{-24}$</td>
<td>fast</td>
</tr>
<tr>
<td>Single (CR)</td>
<td>0.5 ulps</td>
<td>7</td>
<td>$2^{-50}$</td>
<td>slow</td>
</tr>
<tr>
<td>Double</td>
<td>1 ulp</td>
<td>16</td>
<td>$2^{-53.5}$</td>
<td>fast</td>
</tr>
<tr>
<td>Double (CR)</td>
<td>0.5 ulps</td>
<td>16</td>
<td>$2^{-150}$</td>
<td>slow</td>
</tr>
</tbody>
</table>

Not fast enough? Still more flavors

<table>
<thead>
<tr>
<th>Precision</th>
<th>Ulp Error</th>
<th>Decimal Digits</th>
<th>Needed Accuracy</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>$2^5$ ulps</td>
<td>5</td>
<td>$2^{-17}$</td>
<td>faster</td>
</tr>
<tr>
<td>Single</td>
<td>$2^{11}$ ulps</td>
<td>3</td>
<td>$2^{-11}$</td>
<td>the fastest</td>
</tr>
<tr>
<td>Double</td>
<td>$2^{11}$ ulps</td>
<td>12</td>
<td>$2^{-40}$</td>
<td>faster</td>
</tr>
<tr>
<td>Double</td>
<td>$2^{42}$ ulps</td>
<td>3</td>
<td>$2^{-10}$</td>
<td>the fastest</td>
</tr>
</tbody>
</table>
We should get even more flavors!

- set floating-point flags or not: 2 possibilities
- support all rounding modes or not: 4 possibilities
  - save mode, perform computations, restore mode
  - perform computations to support all the rounding modes
  - perform integer-based computations

⇒ Eight more flavors
Task

Rough Computations

- 3 precisions
- \( \sim 50 \) functions in libm
- \( \sim 10 \) flavors in terms of accuracy
- 8 additional flavors

"I have to implement 12000 functions...

Each function implementation takes \( \sim 1 \) man-month

It will take me 1000 years to rewrite the libm"
Task

Rough Computations

- 3 precisions
- \(\sim 50\) functions in libm
- \(\sim 10\) flavors in terms of accuracy
- 8 additional flavors

I have to implement **12000** functions...
Task

Rough Computations

- 3 precisions
- \( \sim 50 \) functions in libm
- \( \sim 10 \) flavors in terms of accuracy
- 8 additional flavors

I have to implement 12000 functions...

Each function implementation takes \( \sim 1 \) man-month
Task

Rough Computations

- 3 precisions
- ~ 50 functions in libm
- ~ 10 flavors in terms of accuracy
- 8 additional flavors

I have to implement 12000 functions...

Each function implementation takes ~ 1 man-month
It will take me 1000 years to rewrite the libm
Outline

1. Introducing libm
2. Problem statement
3. The Metalibm Project
4. General approach to function implementation
Metalibm

Write a tool that produces code for math functions

Analogy

assembly → compilers
code → code generators

```assembly
.cfi_startproc
subq $8, %rsp
.cfi_def_cfa_offset 16
movl $52, %r8d
movl $37, %ecx
movl $15, %edx
movl $.LC0, %esi
movl $1, %edi
xorl %eax, %eax
call __printf_chk
xorl %eax, %eax
addq $8, %rsp
.cfi_def_cfa_offset 8
ret
.cfi_endproc
```

```c
#include <stdio.h>

int main() {
    int a, b, sum;
    a = 15;
    b = 37;
    sum = a + b;
    printf("%d + %d = %d\n", a, b, sum);
    return 0;
}
```
Metalibm prototype

Academic prototype
- Produces flexible math functions implementations
- Gives a function on a specified domain
- It’s free and already available!
  Try it on http://metalibm.org/

State of the art
- Supports almost all libm functions
- The documentation has to be finished
- We are working on the vectorizable implementations
- We are working on support of specific functions (e.g. dickman’s, erf)
Demo
Timings for exponential function

precision=30

O. Kupriianova (LIP6, Paris)
Timings for exponential function

precision=31

time

precision = 31
Timings for exponential function

precision=32

O. Kupriianova (LIP6, Paris)
Timings for exponential function

- Precision = 33

![3D graph showing time, degree, and table with color scale indicating precision.](image)
Timings for exponential function

precision=36
Timings for exponential function

precision=37

O. Kupriianova (LIP6, Paris)
Timings for exponential function

precision=38

O. Kupriianova (LIP6, Paris)  Metalibm  RAIM, 7 April 2015  18 / 25
Timings for exponential function

precision=40

O. Kupriianova (LIP6, Paris)
Timings for exponential function

precision=41

O. Kupriianova (LIP6, Paris)
Timings for exponential function

precision=42

O. Kupriianova (LIP6, Paris)
Timings for exponential function

precision=43
Timings for exponential function

![3D plot of exponential function timings](image)

- **Table**: The table shows the timing data for different degrees and time values.
- **Graph**: The graph illustrates the relationship between the degree, table, and time, with a color gradient indicating the precision level of 44.

*O. Kupriianova (LIP6, Paris)*
Timings for exponential function

precision=45
Timings for exponential function

precision=46
Timings for exponential function

precision=47
Timings for exponential function

![3D graph showing a grid with time and degree axes, and a color bar indicating precision=48]
Timings for exponential function

precision=49
Timings for exponential function

precision=50
Timings for exponential function

 precision=51

 O. Kupriianova (LIP6, Paris)
Timings for exponential function

precision=52

O. Kupriianova (LIP6, Paris)
Timings for exponential function

precision=54

O. Kupriianova (LIP6, Paris)
Timings for exponential function
Timings for exponential function

precision=56

time
precision=56
table
degree

time
0
0.2
0.4
0.6
0.8
1

table
4 5 6 7 8 9 10 11 12
2 4 6 8 10 12 14 16

degree
3 4 5 6 7 8 9 10 11 12 14 16

0 0.2 0.4 0.6 0.8 1

O. Kupriianova  (LIP6, Paris)
Metalibm
RAIM, 7 April 2015
Timings for exponential function

precision=57

O. Kupriianova (LIP6, Paris)
Timings for exponential function
Timings for exponential function

precision=59

degree
time

0 0.2 0.4 0.6 0.8 1

0 2 4 6 8 10 12 14 16

precision=59
Timings for exponential function

precision=60

degree
time

0
0.2
0.4
0.6
0.8
1

time

0
0.2
0.4
0.6
0.8
1
Timings for exponential function

precision=61
Timings for exponential function

precision=62

time

Table:

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>0.2</th>
<th>0.4</th>
<th>0.6</th>
<th>0.8</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
</tr>
</tbody>
</table>

O. Kupriianova (LIP6, Paris)  Metalibm  RAIM, 7 April 2015
Timings for exponential function

precision=63
Timings for exponential function

precision=64

O. Kupriianova (LIP6, Paris)
Timings for exponential function

precision=65

O. Kupriianova (LIP6, Paris)
Timings for exponential function

precision=66

time

degree

time
Timings for exponential function

![3D plot of exponential function timings]

- time
- precision=67
- table
degree
time
- 0
- 0.2
- 0.4
- 0.6
- 0.8
- 1

O. Kupriianova (LIP6, Paris)
Timings for exponential function

![3D plot of exponential function timings with precision=68]

- Time
- Precision
- Table
- Degree

O. Kupriianova (LIP6, Paris)
Timings for exponential function

precision=69
Timings for exponential function

precision=70

time
0
0.2
0.4
0.6
0.8
1
table
0
0.2
0.4
0.6
0.8
1
degree
0
0.2
0.4
0.6
0.8
1
1.0
0
1
2
3
Timings for exponential function

Making this movie meant implementing 6150 functions
Outline

1. Introducing libm
2. Problem statement
3. The Metalibm Project
4. General approach to function implementation
Steps in function implementation

1. Eliminating special cases and values: zeros, infinities, NaNs, etc.
2. Argument reduction: $\bar{x} \in [\alpha, \beta]$ is a small interval
3. Polynomial approximation: Remez algorithm for minimax approximations, polynomial of low degree
4. Reconstruction

Example

implement $f(x) = e^x$

\[
e^x = 2^{\text{int}(x \log 2)} \cdot 2^{x \log 2 - \text{int}(x \log 2)} = 2^E \cdot e^{x-E \log 2} = 2^E \cdot e^r
\]
Range reduction

Step is based on mathematical properties:
\( n^{a+b} = n^a \cdot n^b \), \( \sin(x + 2\pi) = \sin(x) \), \( \log(a \cdot b) = \log(a) + \log(b) \), ...
Range reduction

Step is based on mathematical properties:
\[ n^{a+b} = n^a \cdot n^b, \sin(x + 2\pi) = \sin(x), \log(a \cdot b) = \log(a) + \log(b), \ldots \]

What properties do we know for \( \text{erf}(x) \), or a function defined by ODE?
Range reduction

Step is based on mathematical properties:
\[ n^{a+b} = n^a \cdot n^b, \sin(x + 2\pi) = \sin(x), \log(a \cdot b) = \log(a) + \log(b), \ldots \]

What properties do we know for \( \text{erf}(x) \), or a function defined by ODE?

When range reduction does not work

Piecewise-polynomial implementation
Range reduction

Step is based on mathematical properties:
\[ n^{a+b} = n^a \cdot n^b, \quad \sin(x + 2\pi) = \sin(x), \quad \log(a \cdot b) = \log(a) + \log(b), \ldots \]

What properties do we know for \( \text{erf}(x) \), or a function defined by ODE?

When range reduction does not work

Piecewise-polynomial implementation
- Algorithm to split the domain
- Reconstruction becomes an execution of if-statements
Domain Splitting. Different approaches

\[ f = \arcsin(x), \; l = [0, 0.85], \; d \leq 8, \; \bar{\varepsilon} = 2^{-52} \]

Simple and naive: split domain into \( k \) equal parts, \( k \) is large.
Domain Splitting. Different approaches

\[ f = \arcsin(x), \quad I = [0, 0.85], \quad d \leq 8, \quad \bar{\varepsilon} = 2^{-52} \]

Bisection (use of a theorem of de la Vallée Poussin)
$f = \sin^{-1}(x)$, $I = [0, 0.85]$, $d \leq 8$, $\bar{\varepsilon} = 2^{-52}$

Our optimized method (use of a theorem of de la Vallée Poussin)
### Results

<table>
<thead>
<tr>
<th>name</th>
<th>function $f$</th>
<th>target accuracy $\bar{\varepsilon} = 2^{-52}$</th>
<th>domain $I$ $I = [0, 0.85]$</th>
<th>degree bound $d_{max} = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>asin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_2$</td>
<td>asin</td>
<td>$\bar{\varepsilon} = 2^{-45}$</td>
<td>$I = [-0.75, 0.75]$</td>
<td>$d_{max} = 8$</td>
</tr>
<tr>
<td>$f_3$</td>
<td>erf</td>
<td>$\bar{\varepsilon} = 2^{-51}$</td>
<td>$I = [-0.75, 0.75]$</td>
<td>$d_{max} = 9$</td>
</tr>
<tr>
<td>$f_4$</td>
<td>erf</td>
<td>$\bar{\varepsilon} = 2^{-45}$</td>
<td>$I = [-0.75, 0.75]$</td>
<td>$d_{max} = 7$</td>
</tr>
<tr>
<td>$f_5$</td>
<td>erf</td>
<td>$\bar{\varepsilon} = 2^{-50}$</td>
<td>$I = [-0.75, 0.75]$</td>
<td>$d_{max} = 6$</td>
</tr>
</tbody>
</table>

**Table:** Flavor specifications

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

**Table:** Table of measurements for several function flavors
• writing code $\rightarrow$ writing code generator
• gives functions for the specified domain, specified accuracies, etc.
• parametrized function implementation in several minutes
• new domain splitting procedure
• adding argument reduction procedures
• available at http://metalibm.org
• work on vectorizable implementations and domain splitting
• work on documentation
Thank you for your attention!
Questions?