Framework for Accuracy and Performance Testing of Mathematical Libraries

Ladislav Horky

Department: PH-CMG-CO

Home Institute: Faculty of Nuclear Science and Physical Engineering,
Czech Technical University, Prague

Supervisor: Danilo Piparo

August 13, 2012

Why?

- When developing a fast math library, you can trade accuracy for speed.
- You need to know, how accurate function are before using them.
- Some free fast libraries which do not respect IEEE accuracy standards do not provide full documentation about it.

→ You need to measure function accuracy and performance across many libraries.

Why?

- When developing a fast math library, you can trade accuracy for speed.
- You need to know, how accurate function are before using them.
- Some free fast libraries which do not respect IEEE accuracy standards do not provide full documentation about it.

→ You need to measure function accuracy and performance across many libraries.

Features

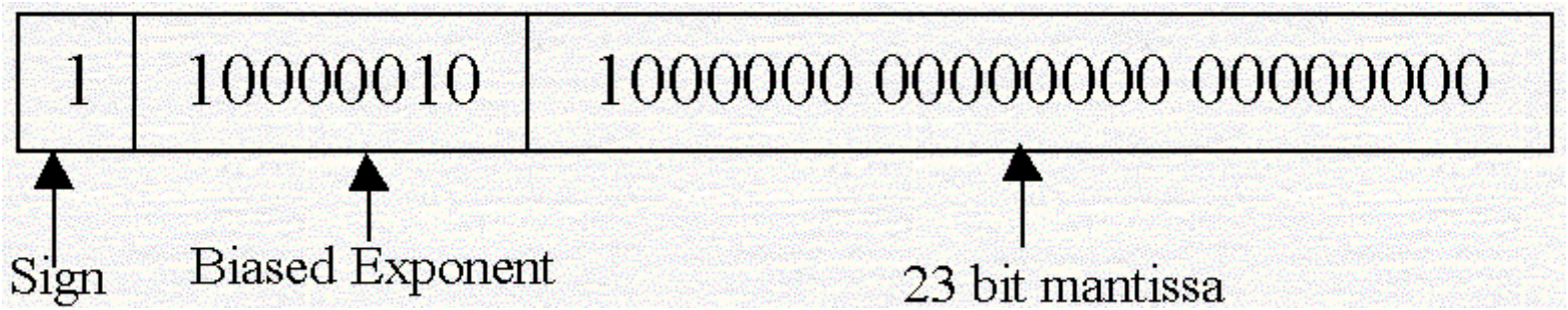
- Arithmetic performance comparison against arbitrary reference implementation (e.g. libm)
- Time performance measurement
- File persistence for all results
- Histograms
- Fully command-line
- Easily extendable to new libraries
- Uses C++11

Reminder: Floating point representation

Arbitrary real number cannot be stored in floating point datatype. Only values of special form specified IEEE 754 standard for base 2 can be represented (showing float case):

$$x = \underbrace{(-1)^s}_{\text{sign}}(1 + \underbrace{\sum_{i=1}^{23} b_i 2^{-i}}_{\text{mantissa}}) \times \underbrace{2^{(e-127)}}_{\text{exponent}} \text{ where } s, b_i \in \{0, 1\}$$

In the memory this data are stored as:



Here, the decimal number -12 is represented (precisely).

Floating point precision II

- Handling floating point numbers in decimal notation may results in imprecise output.

→ Do not rely on standard in/out methods.

→ Treat these numbers as bit (hex) strings instead

→ full information preserved.

In our case, we would represent -12 as
1100 0001 0100 0000 0000 0000 0000 0000
or as 0xC1400000

Floating point precision II

- Handling floating point numbers in decimal notation may results in imprecise output.

→ Do not rely on standard in/out methods.

→ Treat these numbers as bit (hex) strings instead

→ full information preserved.

In our case, we would represent -12 as
1100 0001 0100 0000 0000 0000 0000 0000
or as 0xC1400000

Floating point precision II

- Handling floating point numbers in decimal notation may results in imprecise output.

→ Do not rely on standard in/out methods.

→ Treat these numbers as bit (hex) strings instead
→ full information preserved.

In our case, we would represent -12 as
1100 0001 0100 0000 0000 0000 0000 0000
or as 0xC1400000

Floating point precision II

- Handling floating point numbers in decimal notation may results in imprecise output.

→ Do not rely on standard in/out methods.

→ Treat these numbers as bit (hex) strings instead
→ full information preserved.

In our case, we would represent -12 as
1100 0001 0100 0000 0000 0000 0000 0000
or as 0xC1400000

Precision comparing

Finding the most significant different bit:

1001110010001001110 ↓

1001110001111010010 ←

0000000000011111100

- Measuring precision in terms of different bits.
- Applying this to the whole type instead of mantissa only is logically valid.

Precision comparing

Finding the most significant different bit:

$1001110010001001110 \downarrow$

$1001110001111010010 -$

0000000000011111100

- Measuring precision in terms of different bits.
- Applying this to the whole type instead of mantissa only is logically valid.

Precision comparing

Finding the most significant different bit:

$$\begin{array}{l} 10011100100\color{red}{1}001110 \downarrow \\ 10011100011\color{red}{1}010010 - \\ 00000000000\color{red}{1}111100 \end{array}$$

- Measuring precision in terms of different bits.
- Applying this to the whole type instead of mantissa only is logically valid.

Sample plots

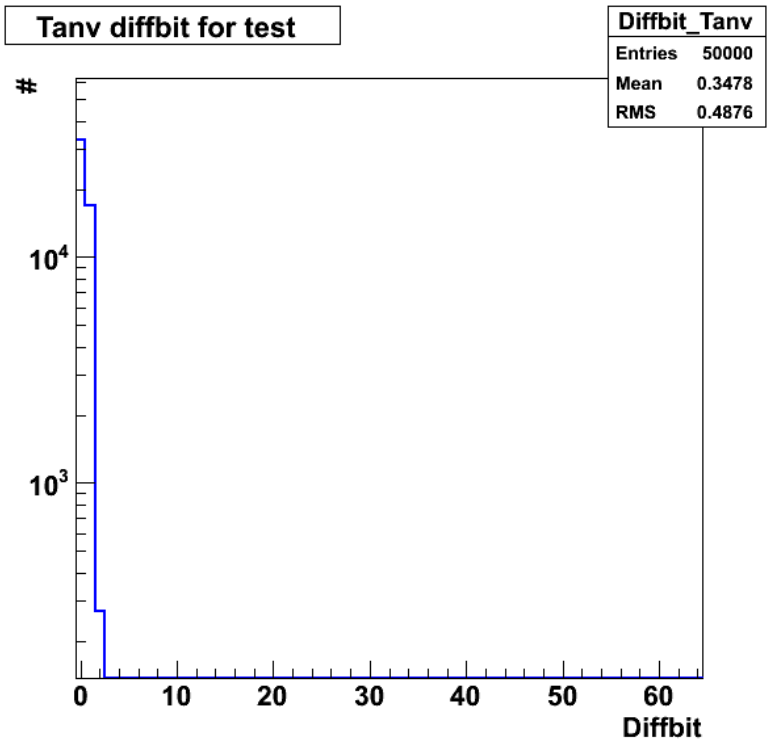


Figure : Acceptable different bit distribution

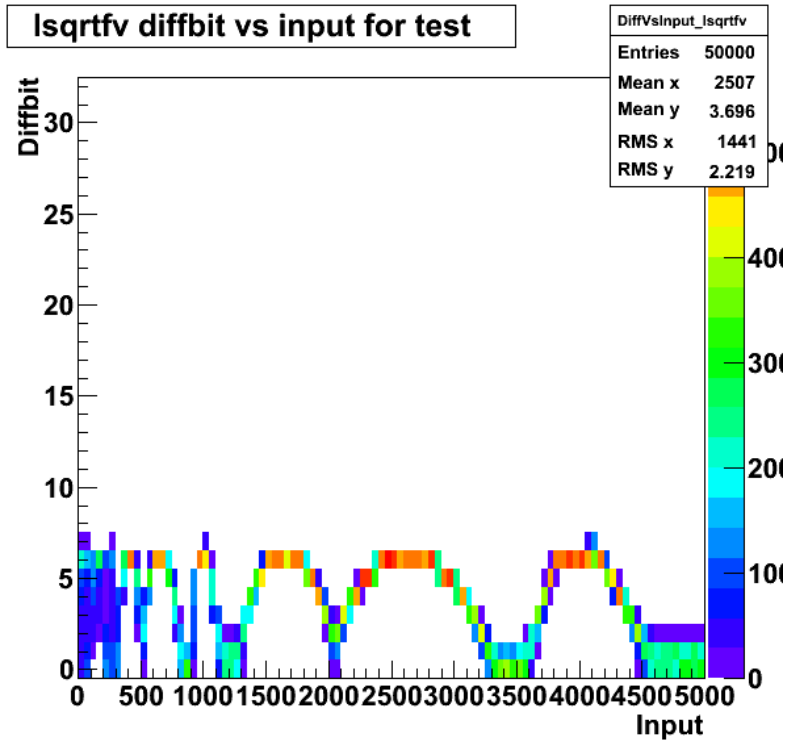


Figure : Fast inverse sqrt behavior (Quake III)

Performance measurement

- Harder than it looks.
- Single function execution $\rightarrow$ nanoseconds $\rightarrow$ impossible $\rightarrow$ measure the loop across several (thousand) iterations.
- In many libraries you even cannot measure single function due to vector signatures.
- Measuring large number of iterations (> 50000) in several trials to remove statistical fluctuations proved sufficient.

Framework benefits

- Easily controllable and extendable code.
- Modular, decoupled stages - function response saving, arithmetic comparison, plots, performance testing.
- No hardcoded things (cmd options).
- Histograms serves as ultimate tool for possible function debugging - you can see, where the error can be.

C++11 features used

- template typedefs
- STL template metaprogramming (`std::function`, `std::tuple`)
- constexpr - way to get rid of #define constants
- auto - automatic type deduction

Which helps in general to:

- move things from runtime to compiletime
- ease work with large amount of templates
- enhance program logic and structure without any overhead at runtime (!)

Generated file example

Contents of file generated by an arithmetic comparison stage (blah__Asinfv__comparison.txt):

```
Arithmetics performance comparison file (the first 5 lines are the heac
Single Precision
Comparison specs/function name = blah
Format: input out1 out2 diffbit (decimal)input
First line are stats: Max Min 0xMean 0xRMS
3 0 0x3fe36848beb5b2d5 0x3fdc20f1e7dcb457
0xbf3b7485 0xbf52555d 0xbf52555e 1 -0.732247
0xbf3a28db 0xbf50704b 0xbf50704c 1 -0.727186
0xbdc7d2e2 0xbdc82466 0xbdc82466 0 -0.097570
0xbf753c50 0xbfa3cfb1 0xbfa3cfb2 1 -0.957952
0xbe98ae2a 0xbe9b09f2 0xbe9b09f1 1 -0.298204
0x3f529d86 0x3f775718 0x3f775719 1 0.822716
0xbd6f9939 0xbd6fbc42 0xbd6fbc41 1 -0.058496
0xbf59e4f6 0xbf825376 0xbf825377 1 -0.851150
0x3e0f0c06 0x3e0f842f 0x3e0f8430 1 0.139694
0x3e8a7a0d 0x3e8c3920 0x3e8c3920 0 0.270462
0xbf523331 0xbf769c6d 0xbf769c6e 1 -0.821094
0x3de61bd8 0x3de69880 0x3de6987f 1 0.112358
0x3f144d43 0x3f1e2d06 0x3f1e2d08 2 0.579304
0xbf0e8608 0xbf17278e 0xbf17278f 1 -0.556733
0xbe26911d 0xbe274f6e 0xbe274f6e 0 -0.162663
0xbf001d1c 0xbf062c30 0xbf062c32 2 -0.500444
0xbed5216e 0xbedbd26f 0xbedbd26f 0 -0.416271
0x3f1b41cb 0x3f26d05b 0x3f26d05c 1 0.606473
```

Thank You.
Questions.