High-Performance Decimal Floating-Point Arithmetic: Algorithms and Implementation in C++

MATTHEW BORLAND[*] and CHRISTOPHER KORMANYOS[*]

This article presents a comprehensive, portable C++ implementation of decimal floating-point arithmetic conforming to IEEE 754-2019 standards[4]. Our system offers three IEEE 754-compliant types, and three additional types that prioritize performance over strict standard adherence, providing flexible options for various computational needs. We describe in detail the Decimal system architecture, its standard library, and usage guidelines. The implementation incorporates novel algorithms for key operations, significantly improving performance in common use cases. Rigorous testing results demonstrate the system's correctness and IEEE 754 compliance. Performance benchmarks show competitive or superior results compared to existing implementations. This work addresses the growing demand for precise decimal arithmetic in financial, scientific, and engineering applications, offering a robust, efficient solution for C++ developers.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Additional Key Words and Phrases: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
Matthew Borland and Christopher Kormanyos. 2018. High-Performance Decimal Floating-Point Arithmetic: Algorithms and Implementation in C++. *ACM Trans. Math. Softw.* 37, 4, Article 111 (August 2018), 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Decimal floating-point arithmetic is crucial for many applications[2], particularly in financial and scientific computing. While several C++ decimal floating-point packages exist, they often lack IEEE 754[4] conformance, interoperability with the C++ Standard Template Library (STL), or both, and have limited portability. This paper presents a novel decimal system that addresses these limitations and advances decimal floating-point technology. Our system is standalone, relies on a minimal subset of the C++ STL, and has been tested on a wide range of devices, from S390X mainframes to AVR boards. It provides seamless interoperability with existing C++ standard types and full IEEE 754 conformance, features not collectively offered by any known package. This work significantly enhances the toolset available for high-precision decimal computations across diverse computing environments.

[*]Both authors contributed equally to this research.

Authors' address: Matthew Borland, matt@mattborland.com; Christopher Kormanyos, e_float@yahoo.com.

ACM 0098-3500/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

2 THE DECIMAL SYSTEM

2.1 Background

The major difference between the layout of binary and decimal floating-point numbers lies in their internal structure and how they represent the same numerical value. Figure 1 illustrates this difference by showing the bit layout of the same number in both formats.

Binary floating-point numbers consist of three parts:

- The sign bit (shown in blue)
- The exponent (shown in green)
- The significand (shown in red)

Decimal floating-point numbers, on the other hand, are composed of four distinct parts:

- The sign bit (shown in blue)
- The combination field (shown in orange)
- The exponent continuation (shown in green)
- The coefficient continuation (shown in red)

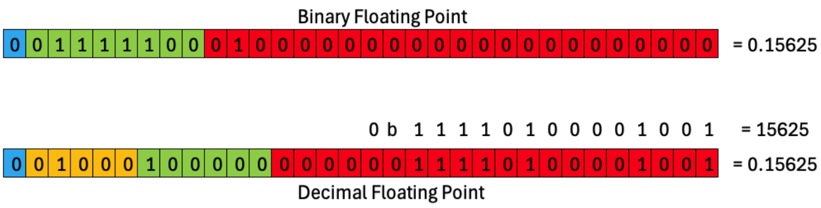

Fig. 1. Bit layouts of Binary and Decimal Floating Point Numbers

As depicted in Figure 1, these different structures allow the same numerical value to be represented in two fundamentally different ways. The binary format uses a base-2 system, while the decimal format uses a base-10 system, which affects how precisely certain decimal values can be represented and how rounding errors propagate in calculations. Due to these differences decimal can represent numbers exactly that binary can not, for example the number 0.3 which with a C++ `double` would actually be: 0.30000000000000004.

2.2 Provided Types

The decimal system provides 6 total types: `decimal32`, `decimal64`, `decimal128`, `decimal32_fast`, `decimal64_fast`, and `decimal128_fast`. The first three types are conformant to IEEE-754 standards while the latter three provided identical numerical results without the constraints of IEEE-754.

The three conformant types is the design specifications provided in IEEE-754 namely their properties:

Parameter	decimal32	decimal64	decimal128
Storage Width	32	64	128
Precision (decimal digits)	7	16	34
Max Exponent	96	384	6144
Exponent Bias	101	398	6176
Sign Width	1	1	1
Combination Field Width	11	13	17
Significand Continuation Width	20	50	110

Table 1. Decimal Floating-Point Format Parameters

The three so-called fast types have the same values of precision, range, and exponent as their analogous conformant type, but require more space.

2.3 System Architecture

The decimal system architecture is robust and flexible. Each type is implemented completely independently of each other, but they share many of the same function implementations from the STL.

2.4 Using the System

Every effort was made during design and implementation to ensure that using the decimal system was straightforward and intuitive. The library contains zero dependencies, and uses only a small subsection of the C++ STL. The required language standard for the library is C++14.

```
#include <boost/decimal.hpp>
#include <iostream>
#include <iomanip>

int main()
{
    using namespace boost::decimal;

    constexpr decimal32 val_1 {100};         // Construction from an
    integer
    constexpr decimal32 val_2 {10, 1};       // Construction from an
    integer and exponent
    constexpr decimal32 val_3 {1, 2, false}; // Construction from an
    integer, exponent, and sign

    std::cout << "Val_1: " << val_1 << '\n'
              << "Val_2: " << val_2 << '\n'
              << "Val_3: " << val_3 << '\n';

    if (val_1 == val_2 && val_2 == val_3 && val_1 == val_3)
    {
        std::cout << "All equal values" << std::endl;
    }

    constexpr decimal64 val_4 {decimal64{2, -1} + decimal64{1, -1}};
    constexpr double float_val_4 {0.2 + 0.1};
    const decimal64 val_5 { float_val_4 }; // Explicit Conversion from
    double

    std::cout << std::setprecision(17) << "Val_4: " << val_4 << '\n'
              <<  "Float: " << float_val_4 << '\n'
              <<  "Val_5: " << val_5 << '\n';

    if (val_4 == val_5)
    {
        std::cout << "Floats are equal" << std::endl;
    }
    else
    {
        std::cout << "Floats are not equal" << std::endl;
    }

    return 0;
}
```

Listing 1. Basic Usage

In Listing 1 we show how to construct the type and that it works just like a built-in flowing point type. The following is an example that leverages one of decimal-floating points main strengths and one that we expect many people will use.

```
#include <boost/decimal.hpp>
#include <iostream>
#include <cassert>

int main()
{
    using namespace boost::decimal;

    decimal64 val {0.25}; // Construction from a double (not recommended
    but explicit construction is allowed)

    char buffer[256];
    auto r_to = to_chars(buffer, buffer + sizeof(buffer) - 1, val);
    assert(r_to); // checks std::errc()
    *r_to.ptr = '\0';

    decimal64 return_value;
    auto r_from = from_chars(buffer, buffer + std::strlen(buffer),
    return_value);
    assert(r_from);

    assert(val == return_value);

    std::cout << " Initial Value: " << val << '\n'
              << "Returned Value: " << return_value << std::endl;

    return 0;
}
```

Listing 2. Basic Usage

In Listing 2 we show how to serialize and parse numbers. These techniques and functions are an extension of Boost.Charconv[1].

3 IMPLEMENTATION DETAILS

3.1 IEEE 754 Conformant Decimal Types

The decimal system provides three IEEE-754 compliant types: `decimal32`, `decimal64`, and `decimal128`.

3.2 IEEE 754 Fast Types

Now that we have discussed the three IEEE-754 conformant types we will turn our attention to much more performant, but non-compliant types. The library offers: `decimal32_fast`, `decimal64_fast`, and `decimal128_fast`.

4 RESULTS

4.1 Testing and Precision

4.2 Performance

5 CONCLUSION AND OUTLOOK

The portable C++ decimal system has been presented. The system allows for users to easily employ decimal-floating point numbers in their existing code bases. For the first time a complete and interoperable system has been fully provided.

During the course of our research and implementation we have limited ourselves to 32, 64, and 128-bit types. Further research can be conducted into making higher precision types as the properties of such numbers are specified in IEEE 754. We could also expand our range of compatibility to allow the types to be massively parallelized such as providing support for CUDA devices.

ACKNOWLEDGMENTS

To the C++ Alliance for sponsoring the development of this library.

REFERENCES

[1] Matt Borland, Peter Dimov, Junekey Jeon, Alexander Grund, Andrzej Krzemieński, Dmitry, Vinnie Falco, and Sam Darwin. 2024. *boostorg/charconv: Boost 1.86.0.* https://doi.org/10.5281/zenodo.13323694

[2] Michael F. Cowlishaw. 2003. Decimal Floating-Point: Algorism for Computers. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic.* IEEE, 104–111. https://doi.org/10.1109/ARITH.2003.1207670

[3] John F. Hart, E. W. Cheney, Charles L. Lawson, Hans J. Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thacher, and Christoph Witzgall. 1968. *Computer Approximations.* John Wiley & Sons, New York.

[4] IEEE. 2019. *IEEE Standard for Floating-Point Arithmetic.* Standard IEEE 754-2019. IEEE Computer Society, New York, NY, USA. https://doi.org/10.1109/IEEESTD.2019.8766229

[5] Donald E. Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. 1-4B. Addison-Wesley, Reading, MA.

[6] Christopher Kormanyos. 2011. Algorithm 910: A Portable C++ Multiple-Precision System for Special-Function Calculations. *ACM Transactions on Mathematical Software (TOMS)* 37, 4, Article 45 (feb 2011), 27 pages. https://doi.org/10.1145/1916461.1916469

[7] Jean-Michel Muller. 2016. *Elementary Functions: Algorithms and Implementation* (3 ed.). Birkhäuser, Boston. https://doi.org/10.1007/978-1-4899-7983-4

[8] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic* (2 ed.). Birkhäuser, Cham, Switzerland. https://doi.org/10.1007/978-3-319-76526-6

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009