

Abstract

Numerical representations are the cornerstone of reliable computational mathematics and computer science calculations. This paper explores the theoretical and practical aspects of representing decimal numbers in binary form, digging into the mathematical constraints and their implications. A key focus is a necessary and sufficient condition for exact binary representations: a decimal fraction has an exact binary representation if and only if its denominator (in lowest terms) is a power of two. The discussion extends to floating-point arithmetic, inexact representations, and the propagation of errors in computational tasks. Through analysis, real-world scenarios, and practical tests, this paper offers insights into the limitations of binary representation and strategies to address them.

Introduction

Understanding how numbers are represented in different numeral systems is important in computing and numerical analysis. Decimal-to-binary conversion, a fundamental process, impacts precision and accuracy in computations. Decimal fractions only have an exact binary representation if their fractional part's denominator, when expressed in lowest terms, is a power of two. This paper examines the mathematical principles behind this constraint and highlights its implications for floating-point arithmetic.

For example, consider the decimal fractions between 0 and 1 with one decimal place: 0, 0.1, 0.2, ..., 0.9. In the binary system using three digits, only two numbers—0 and 0.5—can be exactly represented. Adding binary digits increases precision but does not expand the set of exactly representable fractions. This limitation affects numeric analysis and interval arithmetic, where precise representations are important.

Contents

Abstract.....	1
Introduction.....	1
Number Systems Overview	3
Decimal (Base-10) System.....	3
Binary (Base-2) System.....	3
The Necessary and Sufficient Condition	4
Examples	4
Exact Binary Representations	4
Non-Exact Binary Representations	4
Implications in Computing	5
Floating-Point Arithmetic	5
Error Propagation	5
Mitigation Strategies.....	5
Practical test of decimal numbers.	5
AnalyzeFloatDouble().....	6
normalizetoFraction().....	8
toFractionIntMax()	9
Conclusion	11
References.....	11

Number Systems Overview

Understanding the fundamental differences between decimal and binary numeral systems is critical for exploring numerical representations. Decimal, or base-10, is the standard system for everyday arithmetic, characterized by ten distinct digits (0 through 9). Each digit's position in a number increases its value tenfold from right to left. Conversely, the binary system, or base-2, utilizes only two digits (0 and 1). Here, each position increases in value by powers of two. This stark simplicity suits binary for electronic environments where each digit corresponds to an on or off state.

Decimal (Base-10) System

In the decimal system, numbers are constructed using powers of ten. For integers, these powers increase from right to left, starting from 10^0 (ones place), 10^1 (tens place), and so forth. Decimal fractions extend this logic into negative powers of ten, where each decrease in power corresponds to a decimal place moving right, such as 10^{-1} (tenth place) and 10^{-2} (hundredths place).

Integer Part: Each digit represents a power of ten.

Example: The number 345 is calculated as:

$$\begin{aligned} &3 \cdot 10^2 \text{ (three times ten squared) plus} \\ &4 \cdot 10^1 \text{ (four times ten to the first power) plus} \\ &5 \cdot 10^0 \text{ (five times ten to the zero power) or} \\ &3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0 = 345 \end{aligned}$$

Fractional Part: Each digit represents a negative power of ten.

Example: The number 0.678 is calculated as:

$$\begin{aligned} &6 \cdot 10^{-1} \text{ (six times ten to the negative one) plus} \\ &7 \cdot 10^{-2} \text{ (seven times ten to the negative two) plus} \\ &8 \cdot 10^{-3} \text{ (eight times ten to the negative three) or} \\ &6 \cdot 10^{-1} + 7 \cdot 10^{-2} + 8 \cdot 10^{-3} = 0.678 \end{aligned}$$

Binary (Base-2) System

Binary operates under a similar principle but with a base of two. This means each increase in position from right to left represents a power of two. Binary fractions use negative powers of two, markedly simplifying calculations in digital circuits due to only needing to handle zeros and ones, which directly map to off-and-on states in electronic components.

Integer Part: Each digit represents a power of two.

Example: The binary number 1011 (base 2) equals:

$$\begin{aligned} &1 \cdot 2^3 \text{ (one times two cubed) plus} \\ &0 \cdot 2^2 \text{ (zero times two squared) plus} \\ &1 \cdot 2^1 \text{ (one times two to the first power) plus} \\ &1 \cdot 2^0 \text{ (one times two to the zero power),} \\ &1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11 \text{ in decimal.} \end{aligned}$$

Fractional Part: Each digit represents a negative power of two.

Example: The binary fraction 0.101 equals:

$$\begin{aligned} &1 \cdot 2^{-1} \text{ (one times two to the negative one) plus} \\ &0 \cdot 2^{-2} \text{ (zero times two to the negative two) plus} \\ &1 \cdot 2^{-3} \text{ (one times two to the negative three),} \end{aligned}$$

$$1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.625 \text{ in decimal.}$$

The Necessary and Sufficient Condition

For a decimal number to have an exact binary representation, its denominator must be a power of two when reduced to its simplest form. This section provides an explanation of why this condition is both necessary and sufficient for exact representation.

If the denominator of a fraction is a power of two, the fraction can be exactly represented in binary. This is because the fraction can be expressed as a series of divisions by two, which binary inherently represents without losing information.

On the other hand, if a fraction has an exact binary representation, its denominator must be a power of two. This is derived from the fact that binary fractions expand by multiplying the numerator by increasing powers of two. If these multiplications result in an integer, the denominator's factors must have been entirely in the set of powers of two, thereby proving the necessity.

Examples

Several examples of decimal and binary conversions are provided to illustrate these concepts. They demonstrate both exact and repeating binary forms, showcasing practical instances of how the theoretical principles apply.

Exact Binary Representations

Example 1: $3/4$

Denominator: 4 is 2^2 . Binary Representation: $3/4 = 3 \cdot 2^{-2}$. The binary of 3 is 11. Shifting two places to the right gives 0.11 in binary.

Example 2: 0.125

Fraction: $1/8$. Denominator: 8 is 2^3 . Binary Representation: $1/8 = 1 \cdot 2^{-3}$. This equals 0.001 in binary.

Non-Exact Binary Representations

Example 3: $1/3$

Denominator: 3 is not a power of two. The binary representation is an infinite repeating sequence: 0.010101...

Example 4: 0.1

Fraction: $1/10$. Denominator: 10 has prime factors 2 and 5. The binary Representation 0.1 in decimal cannot be exactly represented in binary. The binary equivalent is an infinite repeating fraction: 0.0001100110011...

Implications in Computing

Floating-Point Arithmetic

Computers use floating-point formats like the IEEE 754 standard to represent real numbers with finite precision. This system is comprised of two key components:

- Mantissa also called the significand. This part of the floating-point format holds the significant digits of the number.
- Exponent. This scales the significand by a power of two, adjusting the size of the number accordingly.

As we have seen, not all decimal fractions are representable within the finite constraints of floating-point formats. For instance, numbers like 0.1, 0.2, 0.3, 0.4, etc., do not have exact binary representations due to these formats' limitations.

All decimal integers will have an exact binary representation if the mantissa is wide enough to handle it. E.g. in a 32bit float the mantissa is 23bit wide which mean that it can accurately represent a number like 16777216, but not 16777217 (which will be rounded to 16777216).

Error Propagation

Minor discrepancies occur when numbers that cannot be precisely represented are stored. These are often small but can become significant in certain contexts. Errors can accumulate through repeated arithmetic operations, potentially leading to significant discrepancies in a long computations sequence.

Mitigation Strategies

There are several ways you can mitigate the error propagation. You can use:

- Interval Arithmetic to Represent numbers as intervals to encapsulate potential conversion errors.
- Use Higher-Precision Formats like quadruple precision or arbitrary-precision libraries like GMP, Boost, or the author's own Arbitrary precision library, which can be found on the www.hvks.com website.
- And lastly, Error Analysis to quantify errors to assess their impact on computations.

Practical test of decimal numbers.

In computational programming, you often have input consisting of decimal numbers that you need to input for data processing. In C++, you can read from the console and store the variable in a floating-point variable. E.g.

```
double d; cin >> d;
```

Above, you read the decimal string from the console and convert it to an internal double number. Since you don't know what the user will input, you don't know if you are getting an actual binary

floating-point representation that matches the decimal input. For example, if the user inputs 0.625, the double `d` variable will represent the exact value of 0.625; however, if the user inputs 0.1, then it will not be accurate, and the `d` variable in the binary form will be the decimal number 0.10000000000000001. This small error doesn't matter for many computational tasks, but when high accuracy or interval arithmetic, you need to account for the inaccuracy. In Interval arithmetic, since the number is an overrepresentation, you need to get the previous number, which is 0.099999999999999992, to be sure the interval does enclose the decimal number 0.1. To implement the test for exact binary representation, we need to convert a number string to a fraction where the numerator and denominator represent the original number without any loss of accuracy. If we keep the fraction as integers, we usually have two 64-bit integers as the fraction. This will work in many situations with smaller decimal numbers; however, if we also have a high exponent (negative or positive), we quickly run into an overflow situation. We need to switch to arbitrary precision integers for a more general approach. We will ignore the potential overflow situation for now and get down to the implementation.

The `analyzeFloatDouble()` relies on two external functions.

- 1) `normalizeToScientific()`
- 2) `toFractionIntMax()`

`normalizeToScientific()` takes a decimal string number and converts it to scientific string notation using only string manipulation functions. Technically, you could avoid it since the next Function (`toFractionIntMax()`) does not require the decimal string number to be in scientific notation. However, the output will be nicer since the formatting is more consistent.

The Function `toFractionIntMax()` converts the decimal string into an integer fraction without loss of accuracy. The output fraction can then be tested for exact presentation, underestimation, and overestimation. If not exact, the reverse under and overestimation is calculated to show the range of valid floating numbers the result is in between (this is the same as the most accurate interval for that decimal representation of the number).

`AnalyzeFloatDouble()`

```
int analyzeFloatDouble(std::string number, int scenario = 2)
{
    std::string typestr[] = { "unknown", "float: mantissa=24bits, exponent=8bits",
    "double: mantissa=53bits, exponent=11bits" };
    // First normalize the number
    std::string normalize = normalizeToScientific(number);

    if (normalize.size() > 0)
    {
        //int_precision numerator, denominator;
        const intmax_t c0(0);
        fraction_precision<intmax_t> orgFrac;
        bool flag;
        int prec(0);

        std::tie(orgFrac, flag) = toFractionIntMax(normalize);
        if (flag)
        {
            // Step 1: Simplify the fraction
            fraction_precision<intmax_t> reducedFrac(orgFrac);
            reducedFrac.normalize();
            fraction_precision<intmax_t> errFrac;
            std::string estimationType;
```

```

double dblValue(0); float fltValue(0);
double next_value(0);
double errorMagnitude(0);
std::string nextstr;

// Step 2: Check if the denominator is a power of 2
bool result = isPowerOfTwo(reducedFrac.denominator());
if (result == false)
{
    // Not an exact binary conversion
    switch (scenario) {
        case 1: prec = std::numeric_limits<float>::max_digits10;
        case 2: prec = std::numeric_limits<double>::max_digits10;
    }
    for (;;) ++prec
    {
        std::string double_str;
        std::ostringstream oss;

        // Step 3: Convert double back to decimal string
        with maximum precision
        switch (scenario) {
            case 1: fltValue = std::stof(normalize);
            case 2: dblValue = std::stod(normalize);
        }
        std::setprecision(std::streamsize(prec)) << fltValue;
        double_str = oss.str();
        break;
        std::setprecision(std::streamsize(prec)) << dblValue;
        double_str = oss.str();
        break;
    }

    // Step 4: Simplify double into fraction
    fraction_precision<intmax_t> dblFrac;
    std::tie(dblFrac, flag) =
toFractionIntMax(double_str);
    dblFrac.normalize();

    // Step 5. Calculate error fraction
    errFrac = reducedFrac - dblFrac;
    if (errFrac.numerator()==0)
        continue;

    // Step 6. Determine overestimation or
    underestimation based on the sign of errNumerator
    switch (scenario) {
        case 1: nextstr = stringnext(fltValue,
errFrac.numerator(<0?-1:1)); break;
        case 2: nextstr = stringnext(dblValue,
errFrac.numerator(<0?-1:1)); break;
    }
    if (errFrac.numerator()==0) {
        nextstr = "Exact representation.";
    }
    errorMagnitude =
std::abs(double(errFrac.numerator()) / double(errFrac.denominator()));
    break;
    }
}

// Output the result.
std::cout << "Analysis type          : " << typestr[scenario] <<
"\n";

```

```

        std::cout << "Original decimal string : " << number << "\n";
        std::cout << "Original fraction      : " << orgFrac.numerator() <<
" / " << orgFrac.denominator() << "\n";
        std::cout << "Reduced fraction      : " << reducedFrac.numerator()
<< " / " << reducedFrac.denominator() << "\n";
        std::cout << "Exact Binary representation : " << (result ? "true"
: "false") << "\n";
        if (result == false)
        {
            // Set the precision to the maximum digits necessary to
uniquely represent a double
            if (scenario == 2)
            {
                std::cout <<
std::setprecision(std::numeric_limits<double>::max_digits10 + 1);
                std::cout << "Double" << " approximation : " <<
dblValue << "\n";
            }
            else
            {
                std::cout <<
std::setprecision(std::numeric_limits<float>::max_digits10 + 1);
                std::cout << "Float" << " approximation : " <<
fltValue << "\n";
            }

            std::cout << "Error fraction      : " <<
errFrac.numerator() << " / " << errFrac.denominator() << "\n";
            if (errFrac.numerator() != 0)
                std::cout << nextstr << "\n";
            std::cout << "Error Magnitude    : " <<
std::setprecision(2) << errorMagnitude << "\n";
        }
        std::cout << "\n";
    }
    return 0;
}
else
    return 1;
}

```

normalizetoFraction()

```

// Function to normalize string number to scientific notation
// e.g. x.yyyy...E[+-]ee...
// only one leading decimal digit, multiple fraction digits and multiple exponent
digits
// note fraction part yyyy... canbe empty
std::string normalizeToScientific(const std::string& number) {
    std::regex pattern("[(-+)?(\\d*)\\.?(\\d*)[Ee]?([+-]?\\d*)");
    std::smatch matches;
    std::string normalized, sign, integerPart, fractionPart, exponentPart;
    long exponent = 0;

    if (std::regex_match(number, matches, pattern)) {
        sign = matches[1];
        integerPart = matches[2];
        fractionPart = matches[3];
        exponentPart = matches[4];

        if (!exponentPart.empty()) {
            exponent = std::stoi(exponentPart);
        }
    }
}

```

```

        if (integerPart.empty() || integerPart == "0") {
            // Handle the case where the integer part is zero or empty
            auto nonZeroPos = fractionPart.find_first_not_of('0');
            if (nonZeroPos != std::string::npos) {
                normalized = fractionPart[nonZeroPos]; // First non-zero
                if (nonZeroPos + 1 < fractionPart.length()) {
                    normalized += '.' + fractionPart.substr(nonZeroPos
+ 1);
                }
                exponent -= (long)(nonZeroPos + 1ull); // Adjust exponent
            }
            else {
                return "0.0E0"; // Return zero if all are zeros
            }
        }
        else {
            // Handle normal or zero integer part with digits
            normalized = integerPart[0]; // First digit of integer
            if (integerPart.length() > 1) {
                normalized += '.' + integerPart.substr(1); // Rest of
integer
            }
            if (!fractionPart.empty()) {
                normalized += '.' + fractionPart; // Append fraction
            }
        }
    }
    else {
        return ""; // Pattern not matched. return empty string
    }

    // Format exponent
    std::string formattedExponent = (exponent >= 0) ? "E" : "E-";
    formattedExponent += std::to_string(std::abs(exponent));

    return sign + normalized + formattedExponent;
}

```

toFractionIntMax()

```

// Function to determine if the float or double number has an exact binary
// representation
// number is in normalized form and dont use arbitrary precision. This mean it
// canoverflow
// if either the numerator or denominator exceed 2^64
std::tuple<fraction_precision<intmax_t>, bool> toFractionIntMax(const std::string&
number) {
    // Regular expression to split the number into sign, integer, fractional, and
    // exponent parts
    std::regex pattern("[+-]?((\\d*)\\.?(\\d*)[Ee]?([+-]?\\d*)");
    std::smatch matches;
    // Determine if number is a power of two
    /*auto isPowerOfTwo = [](intmax_t x) -> bool {
        return (x>0) && ((x & (x - 1))==0);
    };*/

    // Function to remove leading zeros from the fractional part and return the
    // count of removed zeros
    auto removeLeadingZeros = [](std::string& fraction) -> std::tuple<std::string,
int> {
        size_t firstNonZero = fraction.find_first_not_of('0');
        if (firstNonZero != std::string::npos) {
            return std::make_tuple(fraction.substr(firstNonZero),
int(firstNonZero));
        }
    };
}

```

```

    }
    return std::make_tuple("0", fraction.length() > 0 ?
int(fraction.length() - 1) : 0);
};

if (std::regex_match(number, matches, pattern)) {
    // Extracting parts of the number
    std::string signPart = matches[1].str();
    std::string integerPart = matches[2].str();
    std::string fractionalPart = matches[3].str();
    std::string exponentPart = matches[4].str();

    // Determine the sign
    bool isNegative = (signPart == "-");

    // Convert string parts to integer values
    intmax_t integer = integerPart.empty() ? 0 : std::stoll(integerPart);
    std::string cleanedFraction;
    size_t zeroCount = 0;
    // we need to remove leading zero from the fraction to avoid it for
being interpreted as an octal number
    std::tie(cleanedFraction, zeroCount) =
removeLeadingZeros(fractionalPart.empty() ? "0" : fractionalPart);
    intmax_t fractional = std::stoll(cleanedFraction);
    intmax_t exponent = exponentPart.empty() ? 0 :
std::stoll(exponentPart);
    exponent -= zeroCount;

    // Calculate the denominator based on the fractional part
    intmax_t denominator = ipow(10, fractionalPart.length());

    // Calculate the numerator (before applying sign)
    intmax_t numerator = integer * denominator + fractional;

    // Adjust numerator and denominator based on the exponent
    if (exponent != 0) {
        if (exponent > 0) {
            numerator *= intmax_t(pow(10, exponent));
        }
        else {
            denominator *= intmax_t(pow(10, -exponent));
        }
    }

    // Apply the sign to the numerator
    if (isNegative)
        numerator = -numerator;

    // Ensure denominator is positive
    if (denominator < 0) {
        numerator = -numerator;
        denominator = -denominator;
    }

    return std::make_tuple(fraction_precision(numerator, denominator),
true);
}

return std::make_tuple(fraction_precision(0), false); // Return false if the
number does not match the pattern
}

```

Conclusion

By understanding the mathematical foundation of number representations, especially the limitations of binary floating-point formats, professionals can make informed decisions in algorithm design and data processing, leading to more accurate and reliable software systems. The function `analyseFloatDouble()` is to demonstrate the exact/ inexact representation for any given decimal input. The key takeaway is that a decimal fraction has an exact binary representation if its fraction representation has a denominator (in lowest terms) that is a power of two.

Awareness of this rule helps predict and mitigate rounding errors in numerical algorithms. It is crucial for developers and scientists working with floating-point arithmetic.

These techniques mentioned in this paper have been added to a web page you can use to explore the IEEE754 numbers for a deep analysis of the IEEE754 format. See <https://www.lvks.com/Numerical/webIEEE754.html>

References

1. IEEE Standard for Floating-Point Arithmetic (IEEE 754)
2. David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic"