Direct Base-to-Base Conversion Algorithm: Integer and Fractional Parts with Array-Based Arithmetic

Darshangouda Patil
Independent Researcher
ORCID: 0009-0002-3638-0220
darshangouda.s.patil@gmail.com

December 2025

Abstract

This paper introduces a direct, precise base-to-base conversion algorithm for numbers in arbitrary bases from 2 to 62, eliminating the need for intermediate decimal conversions. Unlike traditional approaches which convert through base 10 and suffer from floating-point rounding errors, our method employs array-based big integer arithmetic to maintain exact precision for both integer and fractional parts simultaneously. The algorithm processes digits from the source base one by one with inline normalization to the target base, avoiding floating-point operations altogether. We provide formal pseudocode for both integer and fractional parts, complexity analyses, and practical guidance for safe data-type selection. The integer conversion runs in $O(n \cdot m)$ time and fractional conversion in $O(d \cdot p)$, where n, m represent the count of source and target integer digits and d, p the source fractional digits and desired fractional precision. An open-source C++ implementation is available on GitHub. This approach enables exact, efficient conversions suitable for high-precision and arbitrarily large numeric data in modern computational applications.

Keywords: Direct base conversion, fractional base conversion, array-based arithmetic, arbitrary precision, number systems, integer and fractional representation

Subject Areas: Algorithms, Number Theory, Computational Mathematics, Computer Science, Cryptography, Data Structures, Numerical Analysis, Software Engineering, Data Compression, Digital Signal Processing, Embedded Systems, Serialization, Blockchain and Distributed Systems, Financial Computing, Machine Learning, IoT Systems, Web Systems and Information Retrieval

1 Introduction

1.1 Problem Context

Base conversion is a fundamental operation in computing with applications across cryptography, digital signal processing, fixed-point arithmetic, and data representation. Existing algorithms typically follow two approaches:

1. **Convert-through-decimal:** Convert source base → base-10 → target base (introduces floating-point rounding).
2. **Standard division/multiplication:** Direct algorithm using native division/multiplication (limited by machine precision for large numbers).

Both approaches suffer from precision loss when dealing with:

- Large integers exceeding machine word size.
- Fractional numbers requiring high precision.
- Arbitrary base pairs not aligned with machine precision boundaries.

1.2 Novel Contribution

We present a **direct base-to-base conversion algorithm** using array-based arithmetic that:

- Converts directly without intermediate base-10 representation.
- Maintains arbitrary precision through array storage.
- Handles both integer and fractional parts simultaneously.
- Eliminates floating-point rounding errors.
- Achieves exact conversion for rational numbers within precision limits.

1.3 Key Innovation: Digit-by-Digit Processing with Inline Normalization

Rather than accumulating to a single number then converting, our algorithm:

1. **Processes each input digit individually** from the source base representation.
2. **Multiplies the accumulated result** by the source base (shift operation in array form).
3. **Adds the new digit** and immediately **normalizes to target base** through carry propagation.
4. Maintains result in target base throughout computation, eliminating final conversion step.

This approach is fundamentally different from traditional algorithms and enables exact arithmetic without floating-point operations.

2 Problem Statement and Theoretical Foundation

2.1 Base Conversion Problem

Let B_1 and B_2 be integer bases such that $2 \leq B_1, B_2 \leq 62$. A number

$$N = \sum_{k=-m}^{n-1} d_k B_1^k$$

is given by a finite sequence of digits $\{d_{n-1}, \ldots, d_0, d_{-1}, \ldots, d_{-m}\}$ in base B_1. The goal is to compute digits $\{e_{r-1}, \ldots, e_0, e_{-1}, \ldots, e_{-p}\}$ such that

$$N = \sum_{k=-p}^{r-1} e_k B_2^k,$$

with all digits in $\{0, \ldots, B_2 - 1\}$.

2.2 Array-Based Representation

The algorithm stores the target-base integer part as an array

$$N_{\text{int}} = \sum_{i=0}^{M} a_i B_2^i,$$

where $a_i \in \{0, \ldots, B_2 - 1\}$ and index $i = 0$ holds the least significant digit. This little-endian layout matches the carry propagation direction.

Similarly, the fractional part is represented using a working array in base B_1, arranged to support propagation towards increasing indices.

2.3 Comparison with Existing Approaches and Algorithms

Aspect	**Traditional Division/Multiply**	**Convert-via-Decimal**	**Our Approach**
Integer Precision	Limited by machine word size	Machine precision (∼15 digits)	Arbitrary via arrays
Fractional Precision	Floating-point errors	Floating-point errors (∼15 digits)	Arbitrary via arrays
Speed	Fast (native operations)	Fast (native operations)	Moderate (array ops)
Large Numbers	Limited	Limited	✓ Unlimited
Exact Fractions	Approximation	Approximation	✓ Exact
Direct Base-to-Base	✓ Yes (limited precision)	× Goes through base-10	✓ Yes (exact)

Table 1: Comparison of base conversion approaches by precision and capabilities [1, 2, 5]

3 Proposed Algorithm

3.1 High-Level Structure

Given an input string in base B_1, such as `"3A.F4"`, the algorithm proceeds in three phases:

1. Parse the string, split at the radix point, and map characters to digit values in base B_1.
2. Convert the integer part using the direct base-to-base integer algorithm.
3. Convert the fractional part using the direct base-to-base fractional algorithm, up to a configurable precision limit.

3.2 Integer Part Conversion

Let `integerDigits[0..n-1]` be the source integer digits in base B_1, in left-to-right order (most significant first). The algorithm maintains `resultArray[]` as the accumulated value in base B_2, little-endian.

Algorithm 1 IntegerBaseConvert

Require: `integerDigits[0..n-1]`, `base1`, `base2`
Ensure: `resultArray[0..highestPos]` in base2

Initialize `resultArray[]` to all zeros
`resultLength` $\leftarrow 1$
`carryIndex` $\leftarrow 1$
`maxDigitBase2` $\leftarrow$ `base2` $- 1$
for $k = 0$ to $n - 1$ **do**
  Multiply current result by `base1`
  for $p = 0$ to `resultLength` - 1 **do**
    `resultArray[p]` $\leftarrow$ `resultArray[p]` $\times$ `base1`
  end for
  Add new input digit: `resultArray[0]` $\leftarrow$ `resultArray[0]` $+$ `integerDigits[k]`
  Normalize to base2 (carry propagation)
  for $i = 0$ to `resultLength` - 1 **do**
    `idx` $\leftarrow i$
    while `resultArray[idx]` $>$ `maxDigitBase2` **do**
      `resultArray[idx + 1]` $\leftarrow$ `resultArray[idx + 1]` $+ \lfloor \frac{\texttt{resultArray[idx]}}{\texttt{base2}} \rfloor$
      `resultArray[idx]` $\leftarrow$ `resultArray[idx]` mod `base2`
      `idx` $\leftarrow$ `idx` $+ 1$
    end while
    if `carryIndex` $<$ `idx` **then**
      `carryIndex` $\leftarrow$ `idx`
    end if
  end for
  `resultLength` $\leftarrow$ `carryIndex`
end for
`highestPos` $\leftarrow 0$
for $p = 0$ to MAX_SIZE - 1 **do**
  if `resultArray[p]` > 0 **then**
    `highestPos` $\leftarrow p$
  end if
end for
return `resultArray`, `highestPos`

3.3 Fractional Part Conversion

Let `fractionDigits[0..d-1]` be the source fractional digits in base B_1, stored in reverse order (least significant digit first). A working array `fractionWork[]` is initialized with these values. The algorithm produces at most `fractionLimit` digits in base B_2.

Algorithm 2 FractionalBaseConvert

Require: `fractionDigits[0..d-1]`, `base1`, `base2`, `fractionLimit`
Ensure: `fractionOut[0..resultLength]` in base2, or -1 if no fractional part

if $d = 0$ **then**
  return empty output, -1
end if
Copy input fraction to working array
for $i = 0$ to $d - 1$ **do**
  `fractionWork[i]` $\leftarrow$ `fractionDigits[i]`
end for
`fractionResultIndex` $\leftarrow -1$
repeat
  Multiply all working digits by `base2`
  for $p = 0$ to $d - 1$ **do**
    `fractionWork[p]` $\leftarrow$ `fractionWork[p]` $\times$ `base2`
  end for
  Propagate carries via division by `base1`
  for $i = 0$ to $d - 1$ **do**
    `fractionWork[i + 1]` $\leftarrow$ `fractionWork[i + 1]` $+ \lfloor \frac{\texttt{fractionWork[i]}}{\texttt{base1}} \rfloor$
    `fractionWork[i]` $\leftarrow$ `fractionWork[i]` mod `base1`
  end for
  Extract output digit
  `fractionResultIndex` $\leftarrow$ `fractionResultIndex` $+ 1$
  `fractionOut[fractionResultIndex]` $\leftarrow$ `fractionWork[d]`
  Clear extracted position: `fractionWork[d]` $\leftarrow 0$
  Check if fractional part exhausted
  `fractionNotZero` $\leftarrow$ **false**
  for $p = 0$ to $d - 1$ **do**
    if `fractionWork[p]` $\neq 0$ **then**
      `fractionNotZero` $\leftarrow$ **true**
      break
    end if
  end for
until `fractionNotZero` = **false** OR `fractionResultIndex` $\geq$ `fractionLimit` - 1
return `fractionOut`, `fractionResultIndex`

4 Data Type Selection for Storage

Each cell in the working arrays must accommodate the maximum intermediate value from a single update step to avoid overflow while using memory efficiently.

The worst-case maximum for the integer part is given by:

$$\text{INTEGER\_MAX} = (B_2 - 1) \times B_1 + (B_1 - 1) \tag{1}$$

For the fractional part, the initial bound is:

$$\text{FRACTION\_MAX} = (B_1 - 1) + \left\lfloor \frac{B_1 \times (B_1 - 1)}{B_2} \right\rfloor \tag{2}$$

However, due to carry propagation in subsequent iterations of the algorithm, the fractional bound grows to match the integer bound. Therefore, both integer and fractional parts effectively

share the same maximum bound:

$$\text{MAX} = (B_2 - 1) \times B_1 + (B_1 - 1)$$

This unified bound MAX should be used to select the smallest unsigned integer type that can safely store intermediate values without overflow.

For typical bases $2 \leq B_1, B_2 \leq 62$, this bound never exceeds 3843, so a `uint16_t` data type is always sufficient [1, 2, 5].

4.1 Storage Type Selection

Select the smallest unsigned integer type exceeding `MAX`:

- MAX < 256: `uint8_t`
- MAX < 65536: `uint16_t`
- MAX < 4294967296: `uint32_t`

For bases $2 \leq B_1, B_2 \leq 36$:

$$\text{MAX} \leq (36 - 1) \times 36 + (36 - 1) = 1427$$

Thus `uint16_t` is always sufficient.

5 Complexity Analysis

Let:

- n: number of integer digits in base B_1,
- m: number of integer digits in base B_2,
- d: number of fractional digits in base B_1,
- p: number of fractional digits produced in base B_2.

The integer part performs an $O(m)$ multiply and an $O(m)$ carry sweep for each input digit, resulting in

$$T_{\text{int}} = O(n \cdot m).$$

The fractional part requires p output digits, each produced by multiplying d digits and propagating carries:

$$T_{\text{frac}} = O(d \cdot p).$$

Space usage is dominated by integer result, fractional working array, and output:

$$O(m + d + p).$$

These time and space complexities follow standard big-O analysis [3].

6 Exploration of Base Conversion Techniques

6.1 Input and Output Chunking for Efficient Storage and Processing

Input chunking divides source digits into groups based on powers of base B_1:

- Decimal: groups of 2–N digits
- Hex: groups of 2–N hex digits
- Base36/62: multi-character groups

The following table illustrates chunking examples:

Base1	**Chunk size 2 (M, B)**	**Chunk size 3 (M, B)**
2	M='11', B=$2^2 = 4$	M='111', B=$2^3 = 8$
3	M='22', B=$3^2 = 9$	M='222', B=$3^3 = 27$
4	M='33', B=$4^2 = 16$	M='333', B=$4^3 = 64$

Table 2: Chunking examples showing Max string (M) and resulting Base (B)

Note 1: Chunk input and output is limited to the maximum digit allowed by the base. For example, base 4 with chunk size 3 corresponds to base $4^3 = 64$, but digits are only from $\{0, 1, 2, 3\}$, excluding characters 4–9, A–Z, a–z.

Note 2: Input and output chunk sizes are limited by the capacity of the chosen data type (e.g., `uint16_t` limits base-1000 chunks to 999; `uint32_t` supports base-10000 chunks up to 9999).

Implementation Note: Although not currently enforced in the reference implementation, these digit and data type constraints represent essential robustness considerations for production use. Chunk sizes correspond to powers B^M for arbitrary natural numbers M, enabling flexible grouping across any base system tailored to application requirements and storage constraints.

6.2 Demonstration Examples

A. Base 10 to Base 16 via Base 256 Conversion This example illustrates how the input decimal number is chunked into base-100 digits for internal storage, converted precisely into base-256 chunks, and displayed as base 16 with consistent padding for clarity.

Description	**Value**
Input decimal number	1,200,406
Input decimal array (base 100)	`[1, 20, 4, 6]`
Input expansion in base 100	$1 \times 100^3 + 20 \times 100^2 + 4 \times 100 + 6 = 1,200,406$
Output decimal array (base 256)	`[18, 81, 22]`
Output expansion	$18 \times 256^2 + 81 \times 256 + 22 = 1,200,406$
Displayed output	`12|51|16|` (2-digit padded)
Note	Padding ensures consistent and readable formatting

Table 3: Base conversion example demonstrating input chunking and output display formatting

B. Base 10 to Base 36 via Base 1296 Conversion This example demonstrates conversion from decimal chunking at base 100, to base 1296 super-digits, then displaying the result as base-36 digits with fixed-width formatting.

Description	Value
Input decimal number	123,456
Input decimal array (base 100)	`[12, 34, 56]`
Input expansion (base 100)	$12 \times 100^2 + 34 \times 100 + 56 = 123456$
Output decimal array (base 1296)	`[95, 336]`
Output expansion (base-1296)	$95 \times 1296 + 336 = 123456$
Output expansion (base 1296)	$2N \mid 9C$
Conversion to base 36 digits	$95 = 2 \times 36 + 23 = 2N$
	$336 = 9 \times 36 + 12 = 9C$

Table 4: Base conversion example from decimal to base-1296 chunking and display in base-36

6.3 Strengths and Practical Considerations

Grouping digits enables the algorithm to:

- Handle arbitrarily large bases by representing digits as multi-character groups.
- Use data types efficiently by selecting element types based on super-digit maximum values.
- Maintain exactness throughout conversion without introducing floating-point rounding errors.
- Display results consistently and unambiguously using fixed-width padded digit groups.

6.4 Potential Applications and Future Extensions

The direct base-to-base conversion algorithm is ideally suited for digital systems and computational domains requiring exact and flexible numeric processing, including:

- **High-precision cryptographic applications:** Exact arithmetic with predictable digit representations critical for security and correctness [1].
- **Blockchain:** Efficient crypto-fiat conversions and compact transaction encoding for secure distributed ledgers.
- **Financial Trading:** Exact fixed-point arithmetic ensuring no rounding errors in high-speed quantitative computations [5].
- **IoT Sensors:** Multi-sensor data packing via arbitrary base grouping for reduced transmission and power consumption.
- **Machine Learning Models:** Lossless weight compression and exact reconstruction for efficient storage and deployment.
- **Fixed-point DSP Systems:** Precise base conversion eliminating floating-point errors in signal filtering and transformations.
- **Embedded Systems:** Memory-efficient, integer-only base conversion suited for constrained hardware with accuracy requirements.
- **Scientific Computing:** Exact integer and fractional conversions enabling stable, reliable numerical simulations and symbolic computations.
- **Data Compression and Storage:** High-base grouping dramatically reduces digit count for compact storage and faster transmission.

- **Web ID Encoding:** Base-62 (0–9, A–Z, a–z) encoding for ultra-compact URL shorteners, database identifiers, version control commit hashes, and barcode/QR code numeric encoding (e.g., Twitter, YouTube, GitHub).
- **Serialization:** Lossless numeric encoding and decoding for platform-independent compact data interchange formats.
- **Numeric Validation and Verification:** Rigorous, reversible base transformations to verify correctness of numerical software implementations.

7 Conclusion

We have presented a direct multi-base conversion algorithm supporting arbitrary bases up to at least 62 with extensions beyond via digit grouping. The approach uses integer-only array operations for exact, arbitrary precision conversion of both integer and fractional parts, avoiding floating-point rounding errors.

Appendix A: Pseudocode Summary

Algorithm Flow:

1. **Input parsing:** Extract integer and fractional digits, convert ASCII to numeric values
2. **Integer conversion:** Left-to-right multiplication with inline base-B_2 normalization
3. **Fractional conversion:** Repeated multiplication by B_2 with carry propagation via division by B_1
4. **Output formatting:** Convert numeric digits back to ASCII characters, combine integer and fractional parts

Appendix B: GitHub Repository

Repository: https://github.com/darshangouda/BaseConversionAlgorithm

Language: C++

License: Apache 2.0

Key Files

- `base_converter.cpp` – Core algorithm implementation (integer and fractional parts).
- `README.md` – Usage documentation and examples.
- `LICENSE` – Apache 2.0 license text.
- `CITATION.cff` – Citation metadata file.

References

[1] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Addison–Wesley, 1997.

[2] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Addison–Wesley, 1994.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[4] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019, 2019.

[5] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*. Cambridge University Press, 2010.

Recommended Citation

Darshangouda Patil. (2025). Direct Base-to-Base Conversion Algorithm: Integer and Fractional Parts with Array-Based Arithmetic. https://doi.org/10.5281/zenodo.17735748.