Abstract

Floating-point numbers are an essential part of modern software, recently gaining particular prominence on the web as the exclusive numeric format of Javascript. To use floating-point numbers, we require a way to convert binary machine representations into human readable decimal outputs. Existing conversion algorithms make trade-offs between completeness and performance. The classic Dragon4 algorithm by Steele and White and its later refinements achieve completeness — i.e. produce correct and optimal outputs on all inputs — by using arbitrary precision integer (bignum) arithmetic which leads to a high performance cost. On the other hand, the recent Grisu3 algorithm by Loitsch shows how to recover performance by using native integer arithmetic but sacrifices optimality for 0.5% of all inputs. We present Errol, a new complete algorithm that is guaranteed to produce correct and optimal results for all inputs by sacrificing a speed penalty of 2.4 times the slowest possible Grisu3 but 5.2× faster than previous complete methods.

Categories and Subject Descriptors I.m [Computing Methodologies]: Miscellaneous

Keywords floating-point printing, dtoa, double-double

1. Introduction

How should we print a floating-point number? Consider the following curious interaction with a recent Python REPL (v2.7.5)

```python
>>> 0.1 + 0.1
0.21000000000000002
```

The same counter-intuitive result is displayed by REPLs for JavaScript (node.js v0.10.30), Haskell (GHCi v7.10.1) and Ocaml (Ocaml v4.01.0). This puzzling behavior could be explained by rounding errors at one of two places: the actual computation or, in the focus of this paper, the printing phase.

The goal of the printing phase is to convert the machine-level binary representation of a floating-point number into a human-readable decimal representation with as few digits as needed to communicate the desired binary value. The conversion process is complicated by the fact that the machine can only represent a finite subset of the real numbers. Each binary machine-representable number corresponds to the set of real numbers in an interval around itself. The “wrong” output 0.21000000000000002 and the “right” output 0.21 may fall inside the same interval, and thus have exactly the same machine representation, making it hard for the output procedure to distinguish between the two. Hence, printing floating-point numbers is a surprisingly difficult problem with a distinguished line of work dating back several decades.

In 1990, Steele and White published the first paper to precisely pin down what it means to correctly and optimally print a floating-point number [14]. Intuitively, a (decimal) output is correct if it belongs inside the interval represented by the corresponding (binary) input. Furthermore, an output is optimal if it has the smallest number of digits among all the numbers in the interval represented by the input. Steele and White [14] show a recipe for designing correct and optimal algorithms and instantiate it in the Dragon4 algorithm for printing floating-point numbers. Unfortunately, Dragon4 relies on large integer (“bignum”) arithmetic, incurring a high performance cost. Several authors including [5] and [1] devised improvements which optimized various steps that required slow bignum computations, making Dragon4 suitable for adoption inside various language run-times where it remained for about two decades. (Consequently, we can rest assured that the quirky behavior in the REPLs above is due to rounding and not conversion errors.)

This state of affairs remained until 2010 when Florian Loitsch presented the Grisu3 algorithm. Grisu3 achieved dramatic (up to 12.5×) improvements in efficiency by mostly replacing bignum computations with native 64-bit arithmetic [10]. Unfortunately, the speedup came at a price. Grisu3 is incomplete in that for about 0.5% of all inputs, the algorithm produces correct but suboptimal outputs, i.e. without the fewest number of digits. Loitsch showed how the sub-optimality could be detected at run-time, and in such cases the algorithm could revert to a slower but optimal Dragon4 conversion. As printing floating-point numbers is a performance critical issue in JavaScript engines, where all numbers are floats, Grisu3 was rapidly adopted by major browser engines including Chrome, Firefox and WebKit.

In this paper, we present Errol, a new algorithm for the output conversion of floating-point numbers that is complete and 5.2× faster than Dragon4. We achieve this via the following contributions.

- First, we generalize Grisu-style approximate conversion algorithms into framework parameterized by an abstract high-precision $\mathbb{HP}$ data-type that is used to compute a narrow (resp. wide) interval which provides over- (resp. under-) approximations of the optimal decimal (\$\|$). [5]

- Second, we instantiate the framework by implementing $\mathbb{HP}$ numbers using Knuth’s double-double representation [9]. The resulting algorithm, Errol1, is efficient due to novel algorithms for the key arithmetic operations. Surprisingly, we show that...
the double-double representation shrinks sub-optimality by an order of magnitude because Erro1 computes the narrow and wide intervals more precisely than Grisu3 (§[3]).

- Third, we show empirically that further precision is useless as the errors are due to pathological values that are guaranteed to fall outside the scope of Grisu-style approximate conversion. Guided by the data, we formally characterize the region containing such pathological values, allowing us to simply fall back on an exact conversion for such inputs. We find the resulting algorithm, Erro2, to be optimal on 99.9999999% of all inputs (§[5]).

- Fourth, even outside the pathological space, Grisu-style approximation may yield sub-optimal results. We develop a novel necessary condition for an input to yield sub-optimal outputs and use it to develop a constraint-based synthesis algorithm that efficiently pre-computes all (141) inputs where Erro3 may yield the sub-optimal conversion. This tabulation yields Erro3 which is guaranteed to return correct and optimal conversion for all inputs ($\S$[5]).

- Finally, we present an empirical evaluation comparing Erro3 against the state of the art. We show that it is 2.5x slower than an incomplete Grisu3, 2.4x slower than Grisu3 made complete via dynamic checking, and 5.2x faster than Dragon4, the previous complete conversion method ($\S$[7]).

2. Preliminaries

We begin with some preliminaries about the properties of floating-point numbers and their representation that are needed to understand our algorithm for output conversion.

2.1 Representation

Machines use floating-point number representations to approximate real number computations.

Floating-Point Representations. A real is a value on the real number line. A floating-point format defines a finite set of representable numbers (or just representations) where each representation covers a wide range (i.e., interval) of the real number line. Variables that name representations will always be adorned with a hat, such as $\hat{v}$, and are denoted with the type $\FP$. Note that unlike integers, which have equal-width gaps between numbers, floating-point representations have a large range of possible gaps with small representations closely packed together and large representations spread far apart.

High-Precision Representations. A high-precision floating-point number refers to a custom format with higher precision than representations supported by the native machine’s architecture. Variables for high-precision numbers are adorned with a tilde, such as $\tilde{v}$, and are denoted with the type $\HP$. High-precision numbers can be implemented in a variety of ways, e.g., big-integers libraries or structures.

IEEE-754. $\FP$ representations consist of a fixed radix (or base), a fixed-width significand, and a fixed-width exponent of the form: $\text{significand} \times \text{radix}^{\text{exponent}}$

The significand consists of a sequence of $N$ digits $d_1 \ldots d_N$ where each digit $d_i$ is in the range defined by the radix: $0 \leq d_i < \text{radix}$. The IEEE-754 standard defines a set of floating-point formats, including the ubiquitous double-precision (double) and single-precision (float) binary formats. For the sake of simplicity, the remainder of the paper will exclusively discuss double numbers unless otherwise specified.

Normal Representations. A single number can be represented by multiple floating-point representations; e.g., $0.12 \times 10^2$ and $1.2 \times 10^3$ represent the same decimal number. A normal representation is a floating-point number where there is a single, non-zero digit left of the radix point. Each floating-point number has a unique normal representation; e.g., $1.2 \times 10^3$ is the unique normal representation of the above number. However, the normal representation does not offer sufficient coverage for very small values close to zero. To expand the range of representable small numbers and enable “gradual underflow”, the IEEE-754 standard defines the notion of a subnormal number where the significand begins with a leading 0 bit followed by several lower order bits.

2.2 Conversion

As the $\FP$ numbers are finite, an arbitrary real number is unlikely to fall exactly upon an $\FP$ value. Next, we describe how reals are rounded to $\FP$ values in the IEEE-754 standard, and the notion of rounding to define correct and optimal conversion from floating-point to decimal format.

Neighbors. For each representation $\hat{v}$ (except at the extremes) there exists a successor representation denoted as $\hat{v}^+$ which is the next (larger) representation, and a predecessor representation denoted as $\hat{v}^-$ which is the previous (smaller) representation. Thus, for each real number there is a pair of adjacent neighbors $\hat{v}$ and $\hat{v}^+$ such that the real is in the interval between the neighbors.

Midpoints. A midpoint $\hat{m}$ is the real value exactly between two adjacent representations. Formally, for a representation $\hat{v}$, we define the succeeding and preceding midpoints respectively as:

$$\hat{m}^+ \triangleq \frac{\hat{v} + \hat{v}^+}{2} \quad \hat{m}^- \triangleq \frac{\hat{v}^- + \hat{v}}{2}$$

The midpoints are not representable using the native representation format but can be represented by most high-precision representations that provide at least a single additional bit.

Rounding: Intervals & Functions. Recall that a given value $\hat{v}$ is surrounded by the midpoints $\hat{m}^-$ and $\hat{m}^+$. Any real $r$ in the range $\hat{m}^- < r < \hat{m}^+$ is rounded to $\hat{v}$. If the real falls exactly on a midpoint between two floating-point numbers, the standard dictates that the value must be rounded to the even floating-point number, which is the number whose last bit is 0 (as opposed to odd numbers whose last bit is 1). Thus, the rounding interval of a binary representation $\hat{v}$ is the range $[\hat{m}^-, \hat{m}^+]$ when $\hat{v}$ is even or $(\hat{m}^-, \hat{m}^+)$ when $\hat{v}$ is odd. The rounding function $\fltr$ takes a real number and rounds it to the nearest floating-point representation. Thus, for any number $r$ in the rounding interval of $\hat{v}$, $\fltr(r) = \hat{v}$.

Conversion: Correctness & Optimality. Let $\hat{v}$ be a binary input representation and let $r$ be the decimal number produced as the conversion output. The conversion from $\hat{v}$ to $r$ is correct if $r$ falls in rounding interval of $\hat{v}$ (i.e., $\fltr(r) = \hat{v}$). The length of a real $r$ is the number of decimal digits required to write the significand as a string. For example, the length of $1.24 \times 10^3$ is 3. The conversion from $\hat{v}$ to $r$ is optimal if $r$ is the value in the rounding interval of $\hat{v}$ with the smallest length, i.e., if every value in the rounding interval of $\hat{v}$ has length at least as large as $r$. Note that there may exist multiple, unique values of $r$ of optimal length.

3. A Generic Conversion Framework

Our work builds on the Grisu3 algorithm of Loitsch [10]. In this section, we distill the insights from Grisu3 into a general conversion framework parameterized by an abstract high-precision representation $\HP$ and discuss the requirements on $\HP$ that ensure cor-

---

1 IEEE-754 defines a set of five rounding modes; however, we focus only on round to nearest, ties to even.
3.1 Generic Conversion Algorithm

Recall that a correct and optimal decimal conversion of a number \( \hat{v} \) is the shortest decimal in the rounding interval of \( \hat{v} \). The key insight in Grisu3, illustrated in Figure 1, is to:

- **Step 1:** Compute narrow boundaries \( \hat{n}^-, \hat{n}^+ \) such that:

  \[ \hat{m}^- < \hat{n}^- < \hat{v} < \hat{n}^+ < \hat{m}^+ \]

- **Step 2:** Compute shortest decimal in the interval \([\hat{n}^-, \hat{n}^+]\).

By relaxing the constraints of using exact midpoints \( \hat{m}^-, \hat{m}^+ \), Grisu3 can use efficient operations over limited-precision numbers (instead of Dragon4’s bignum) to yield provably correct albeit possibly *suboptimal* conversions.

**Scaled Narrow Intervals.** A triple \((e, \hat{n}^-, \hat{n}^+)\) is a scaled narrow interval for \( \hat{v} \) if there exists \( \hat{n}^-, \hat{n}^+ \) such that:

1. \( \hat{n}^+ \in (1, 10] \)
2. \( \hat{n}^- \approx 10^{-e} \times \hat{n}^- \)
3. \( \hat{n}^+ \approx 10^{-e} \times \hat{n}^+ \)
4. \( \hat{m}^- < \hat{n}^- < \hat{v} < \hat{n}^+ < \hat{m}^+ \)

Intuitively, a scaled narrow interval for \( \hat{v} \) corresponds to a narrow interval for \( v \) where the boundaries are scaled by \( 10^{-e} \) to ensure that the upper bound \( \hat{n}^+ \) is in \((1, 10] \). Note that the last requirement allows us to compute the (scaled) narrow intervals approximately, e.g. using IEEE numbers, as long as the (unscaled) boundaries reside within the exact midpoints \( \hat{m}^- \) and \( \hat{m}^+ \). Finally, only the upper boundary \( \hat{n}^+ \) must be in the interval \((1, 10] \), hence we observe that the exponent is \( e \approx \lfloor \log_{10} \hat{n}^+ \rfloor \).

**Algorithm.** Figure 2 formalizes the above intuition in a generic algorithm to convert an input FP value \( \hat{v} \) into decimal form comprising a pair of a sequence of digits \( d_1, \ldots, d_N \) and an exponent \( e \) denoting the decimal value \( 0.d_1 \ldots d_N \times 10^e \). (Although this differs slightly from the normal format with a leading non-zero digit – we can normalize by shifting the decimal point to the right.) The convert algorithm is split into two procedures, narrow_interval and digits, corresponding to the steps described above. The first phase narrow_interval begins with the input \( \hat{v} \) and computes a scaled narrow interval \((e, \hat{n}^-, \hat{n}^+)\) for \( \hat{v} \). The second phase digits uses the scaled narrow interval to compute the final output digits corresponding to the shortest decimal value within the scaled narrow interval \([\hat{n}^-, \hat{n}^+]\). Next, we describe the two steps in detail.

```python
def convert(\(v\)):
    \((e, \hat{n}^-, \hat{n}^+)\) = narrow_interval(\(v\))
    digits = digits(\(\hat{n}^-, \hat{n}^+\))
    return (digits, \(e\))

def narrow_interval(\(v\)):
    \((\hat{n}^-, \hat{n}^+)\) = boundary(\(v\))
    \(e\) = floor(\(\log_{10}(\hat{n}^+)\))
    \(\hat{n}^-\) = multiply(\(\hat{n}^-\), \(10^{-e}\))
    \(\hat{n}^+\) = multiply(\(\hat{n}^+\), \(10^{-e}\))
    return (\(e, \hat{n}^-, \hat{n}^+\))

def digits(\(\hat{n}^-, \hat{n}^+\)):
    digits = []
    repeat:
        \((\hat{n}^-, d^-)\) = next_digit(\(\hat{n}^-\))
        \((\hat{n}^+, d^+)\) = next_digit(\(\hat{n}^+\))
        digits.append(d^+)
    until \(d^- \neq d^+\)
    return digits

def next_digit(\(n\)):
    \(d = \text{truncate}(\(n\))\)
    \(\hat{n} = \text{multiply}(\(n - d\), 10)\)
    return (\(\hat{n}, d\))
```

Figure 2: A Generic Conversion Algorithm

3.2 Step 1: Compute Narrow Interval

The first phase computes the scaled narrow interval \((e, \hat{n}^-, \hat{n}^+)\) for the input \( \hat{v} \). First, the (unscaled) boundary \( \hat{n}^-, \hat{n}^+ \) is computed directly from the input \( \hat{v} \) by the function boundary. Second, the exponent \( e \) is directly computed from the upper boundary \( \hat{n}^+ \) by scaling it to ensure that its significand lies in the range \([1, 10]\). That is, the exponent \( e \) is computed as the value \( \lfloor \log_{10} \hat{n}^+ \rfloor \). Finally, using the exponent, the scaled narrow boundaries are computed by multiplying the narrow boundaries by \( 10^{-e} \).

The functions boundary and multiply are deliberately left abstract. However, any concrete implementations must take care to ensure that despite errors introduced by rounding and propagation, the overall output is indeed a valid narrow interval for \( \hat{v} \). Consequently, the narrow boundaries computed by boundary are chosen in a conservative fashion – as shown in Figure 3 – so that despite any rounding and propagation errors, the results remain within the actual midpoints, and hence form a valid narrow interval.

3.3 Step 2: Compute Digits

Once we have calculated the scaled narrow interval and corresponding exponent \( e \), we can extract the digits using Steele & White’s method [12].

One approach would simply generate digits of the upper bound in the following manner. Recall that the upper bound is scaled to be of the form \( d_1 d_2 \ldots d_N \) – a result of falling in the range \([1, 10]\). Thus, the leading digit is retrieved by simply truncating the value to an integer, leaving a remainder bound comprising the digits \(0.d_2 \ldots d_N\). The remainder is multiplied by 10 to return it to the decimal format \(d_2 d_3 \ldots d_N\). The process can be iteratively performed \(N\) times, until all decimal digits are exhausted and the remainder is zero.

While this process yields a correct result that is in the rounding interval of the input \( \hat{v} \), it does not yield the optimal value with the
shortest digit sequence. To recover optimality, Steele & White [14] make clever use of the lower boundary \( \hat{n}^- \): they simultaneously perform the above extraction process on both boundaries, generating two sequences of digits: \( d_1^-, d_2^- \ldots \) and \( d_1^+, d_2^+ \ldots \), and repeating the process until it finds the first pair of digits that differ i.e. the first \( k \) such that \( d_k^- \neq d_k^+ \). The sequence of digits that is then output is the upper sequence \( 0.d_1^+ d_2^+ \ldots \) which is identical to the lower sequence except at the very last digit.

Looking at it another way, the algorithm iteratively generates the digits of the upper bound \( \hat{n}^+ \), forming the sequence \( (d_1^+, d_2^+, d_3^+, \ldots) \). The process terminates once the generated number falls within the rounding interval. Figure 4 visually shows this process for the sample boundaries \( \hat{n}^- = 1.212 \) and \( \hat{n}^+ = 1.234 \). In this example, the algorithm produces the correct and optimal number 1.23.

**Theorem 1.** The function digits \( (\hat{n}^-, \hat{n}^+) \) returns the optimal (shortest) decimal value in the interval \([\hat{n}^-, \hat{n}^+]).

By construction, the output digits are guaranteed to be correct: the sequence of digits is guaranteed to be less than \( \hat{n}^- \) and the algorithm terminates once digits is greater than \( \hat{n}^- \). To show that the generated digits are optimal, i.e. they are the shortest sequence between \( \hat{n}^- \) and \( \hat{n}^+ \), we refer to Theorem 6.2 in [10].

### 3.4 Optimality Verification

The above process computes a decimal output that is both correct and optimal within narrow boundaries; however, the output is not necessarily optimal within the larger rounding interval. In particular, the uncovered interval depicted in Figure 1 may contain a smaller decimal than any in the narrow interval considered by digits, and hence convert may fail to generate the shortest possible decimal output in the rounding interval.

Loitsch’s Grisu3 algorithm introduces an a posteriori optimality verification step. Grisu3 computes a second decimal output over the wide interval that is guaranteed optimal (or shorter) but not necessarily correct, as it includes points outside the rounding interval. Nevertheless, if the length of the original and second outputs are equal, then the original output must be optimal [10]. Of course, there may be shorter decimals inside the wide interval but not inside the rounding interval. In this case, the verification would produce a false negative, errantly claiming the output is sub-optimal.

### 4. Errol1: Fast & More Optimal

Our first contribution is the Errol1 conversion algorithm, which instantiates the generic convert (Figure 2) with a novel HP implementation that simultaneously improves the accuracy and performance of conversion. The performance benefits come from implementing HP numbers using Knuth’s double-double representation [9] and by developing novel algorithms for efficiently performing the key operations over double-doubles that are needed for conversion. Surprisingly, the double-double representation also improves the accuracy of conversion as it lets Errol1 compute the narrow and wide intervals more accurately, thereby shrinking the uncovered intervals and increasing the likelihood that the shortest number in the narrow interval is indeed the optimal result.

Next, we describe double-double based HP values (§4.1) and how Errol1 uses them to implement the key narrow_interval (§4.2) and digits (§4.3) procedures of Figure 2. These procedures require fast and accurate implementations of specific arithmetic operations over HP numbers (§4.4), and finally we show how we ensure accuracy in the presence of rounding error (§4.5). For the remainder of the paper, the HP type refers to double-double floating-point numbers.

#### 4.1 Double-Double Representation

We represent double-double numbers (of type HP) as a pair of native floating-point values (of type FP). That is, type HP = \( \langle FP_r, FP_l \rangle \)

HP values are written as \( (\hat{v}_b, \hat{v}_d) \) where the first element \( \hat{v}_b \) is a base value corresponding to the nearest approximation of the target HP value, and the second element \( \hat{v}_d \) is an offset value corresponding to the difference between the target value and the base. Thus, the HP value \( \hat{v} \) represented by \( (\hat{v}_b, \hat{v}_d) \) is the sum:

\[
\hat{v} \doteq \hat{v}_b + \hat{v}_d
\]

Our pair-based representation doubles the precision of the native representation (e.g. if the native FP has 53 bits of precision, then our HP has 106 bits of precision). Figure 5 shows how the real 0.2 is represented as a native 8-bit FP and as a 16-bit HP value; the latter approximates the real more faithfully.

#### Non-Overlapping Invariant

Let \( \hat{v}_b \) and \( \hat{v}_d \) be two \( p \)-bit FP values. We say that \( (\hat{v}_b, \hat{v}_d) \) is non-overlapping if \( e_b \geq e_d + p \) where \( e_b \) and \( e_d \) are the exponents of \( \hat{v}_b \) and \( \hat{v}_d \), respectively. This implies the weaker statement \( |\hat{v}_b| > |\hat{v}_d|^{2p-1} \) that relates the magnitudes of \( \hat{v}_b \) and \( \hat{v}_d \). Our algorithms maintain the invariant that in any HP value \( (\hat{v}_b, \hat{v}_d) \), the components \( \hat{v}_b \) and \( \hat{v}_d \) are non-overlapping to ensure the most faithful representation of the target real number.
def narrow_interval_hp(v):
    # Phase 1: Exponent Estimation
    (_e2) = frexp(v) # e2 ≈ log2 v
    e = floor(e2*0.30103) # e ≈ log10 v
    l = pow10lookup(e) # l ≈ 10^-e
    ṽ = multiply(v, l)
    while(10 <= ṽ):
        ṽ = ṽ / 10
        e = e + 1
        l̃ = l / 10
    while(ṽ < 1):
        ṽ = ṽ * 10
        e = e - 1
        l̃ = l / 10
    # Phase 2: Boundary Computation
    ñ = (ṽ, ṽ + (ṽ - ṽ) * l̃ / (2 * e))
    n̂ = (ṽ, ṽ + (ṽ - ṽ) * l̃ / (2 * e))
    # Phase 3: Exponent Rectification
    while(10 <= n̂):
        n̂ = n̂ / 10
        ñ = ñ / 10
    n̂ = n̂ / 10
    e = e + 1
    while(n̂ < 1):
        n̂ = n̂ * 10
        e = e - 1
        ñ = ñ * 10
    return (e, ñ, n̂)

Figure 6: Erroll1: Computing the Scaled Narrow Interval.

### 4.2 Step 1: Compute Narrow Interval

The narrow interval computation, summarized in Figure 6, is split into three distinct phases: exponent estimation, boundary computation, and exponent rectification. Each phase uses various HP arithmetic operations. We defer the implementation of these operations and the errors incurred to §4.4.

**Phase 1: Exponent Estimation.** Unlike the description in Figure 2, Erroll1 first estimates the exponent directly from the input ṽ before computing the boundaries. The estimate optimizes performance by allowing Erroll1 to use a fast lookup table to avoid several slow multiplications (by 10). However, as the estimated exponent may be incorrect, it is later rectified in the third phase.

The initial exponent estimation is shown on lines 1-14. The exponent is directly estimated from the input ṽ: the frexp function returns the binary exponent and is scaled by log10 2 ≈ 0.30103 to provide a rough estimation of log10 v. The value of 10^-e is stored as an HP value in a lookup table and retrieved on line 5. The values ṽ (an FP) and 10^-e (an HP) are multiplied to produce a scaled input ṽ (an HP) on line 6. At the end of line 6, the value ṽ has been scaled by 10^-e (as stored in l̃).

The lookup table for 10^-e can only contain values in the range [10^-291, 10^-608] in order to prevent overflow or underflow of an HP value. Consequently, input values below 10^-308 or above 10^291 are not successfully scaled into the range [1, 10]. Instead, the two loops on lines 6-13 check and incrementally multiply or divide by ten in order to completely scale ṽ, by adjusting the exponent e and the base component of l̂; the offset is not used later.

**Phase 2: Boundary Computation.** The boundaries ñ^- and ñ^+ are computed from the original input ṽ and scaled input ṽ on lines 17 and 18. Using the definition of the scaled upper boundary n̂^+, its computation directly follows from the calculation

\[
\frac{n^+ - \hat{v} + \hat{v}^+}{2} = \frac{\hat{v} \times 10^{-e} + \hat{v}^+}{2} = \frac{\hat{v} \times 10^{-e} + \hat{v}^+}{2} = \frac{\hat{v} + \hat{v}^+ - \hat{v}}{2} \times l̃
\]

where the last equality arises as ṽ and l̃ are equal to ṽ × 10^-e and 10^-e, respectively. Typically, summing the first term (ṽ) and the second term would require a full addition between two HP numbers. However, the second term is extremely small compared to the first due to the small difference between ṽ and ṽ^+ and so the second term can be directly added to the offset component l̃.

The same process applies to computing the lower boundary ñ^- by substituting the successor ṽ^+ with the predecessor ṽ^-.

**Accounting for Error with ε.** The equality above corresponds to the exact boundaries. To compute the narrow (or wide boundaries), the divisor of 2 is adjusted by a factor ε in order to narrow or widen the interval [ñ^-, ñ^+]. As ε determines the width of the narrow and wide intervals, its exact value depends on the worst-case error incurred when computing the scaled boundaries ñ^- and ñ^+ (as illustrated informally in figure 3). In §4.4, we analyze the errors, and in §5.3, we show they yield a suitable ε.

**Phase 3: Exponent Rectification.** Although the scaled input ṽ is guaranteed to be within the interval [1, 10], the scaled upper boundary n̂^+ is not guaranteed to fall within the interval [1, 10]. The code on lines 21-28 inspects the value of n̂^+, multiplying or dividing by ten in order to scale it to the desired range. The lower bound ñ^- and exponent are adjusted accordingly. Consequently, we can show that:

After the exponent and narrow interval are fully adjusted, the scaling invariants from section 3.3 are satisfied: the scaled upper boundary n^+ falls uniquely in the range [1, 10]; the exponent satisfies the equation n^- = 10^-e n^+; and the lower bound is scaled in the form ñ^- = 10^-e n̂^-.

**Theorem 2.** The function narrow_interval_hp(ṽ) returns a scaled narrow interval (e, ñ^-, n̂^+) for ṽ.

### 4.3 Step 2: Compute Digits

We extract the digits from the scaled narrow interval by using the method of Steele & White [13], specialized to our double-double HP representation, as summarized in Figure 7.

Truncation in lines 4 and 5 is performed directly on the base component of the boundary. Care must be taken in order to guarantee the accuracy of truncation. Recall that by construction, an HP value consists of a base component that best approximates the target value (as an FP) and a smaller offset component that accounts for the remainder (as a non-overlapping FP). In order for the offset to affect truncation, the base component must be an integer: any non-integer value indicates the HP value is too far from an integer for the offset component to affect truncation (otherwise the base component is not the best approximation). If the base is an integer, the offset can only affect the truncation if it is negative; the code on lines 8-13 checks and accounts for this case. The extracted digit is removed from the boundary (lines 6, 7, 10, and 13), and the boundary is multiplied by ten (lines 15 and 16) in order to prepare the next digit for extraction. Subtraction is performed on the base component n_b; the offset is too small to be affected.
def get_digits_hp(ñ, n̂):
    digits = []
    repeat:
        d̅ = trunc(ñ)
        d̂ = trunc(n̂)
        ñ = ñ - d̅
        n̂ = n̂ - d̂
        if (ñ ≤ 0) & (n̂ < 0)
            d̅ = d̅ - 1
        ñ = ñ + 1
        if (ñ ≥ 0) & (n̂ < 0)
            d̂ = d̂ + 1
        n̂ = n̂ + 1
        digits.append(d̂)
        n̂ = n̂ * 10
        ñ = ñ * 10
        until (d̃ != d̂)
    return digits

Figure 7: Erroll1 algorithm for generating digits based on the boundaries ñ and n̂.

Thus, as before, we can show that digits_hp returns a correct and optimal decimal in the given narrow interval:

**Theorem 3.** The function digits_hp(ñ, n̂) returns the optimal (shortest) decimal value in the interval [ñ, n̂].

### 4.4 Double-Double Arithmetic

The eagle-eyed reader will have noticed that Erroll1 (i.e. the code in Figures 6 and 7) requires only the following arithmetic operations: (1) add-FP-to-FP (2) multiply-FP-to-FP (3) multiply-FP-by-10 (4) divide-FP-by-10 Next, we describe novel algorithms to implement these arithmetic operations, and provide a detailed error analysis by bounding the maximum error using the standard numerical analysis notion of machine epsilon \( \varepsilon \) (also known as "macheps" or "unit roundoff") [6]. In \$4.5 \$ we will use the per-operation error bounds to derive a value for \( \varepsilon \) that yields Theorem 2. As our HP format has twice the precision of FP numbers, our error analysis will be measured in terms of \( \varepsilon^2 \) which is equivalent to the maximum round-off error of an HP number. In the sequel, for all operations, we write \( \hat{x} \) and \( \hat{z} \) to denote the HP input and output respectively.

1. **Add-FP-to-FP.** In Erroll1, we only sum an HP number \( \hat{x} \) with an FP number \( \hat{y} \) that is smaller than \( \hat{x} \). While not used directly in figures 6 and 7, this procedure serves as a subroutine used in subsequent operations. The output \( \hat{z} \) is computed component-wise, using a compensation \( c \) inspired by the Kahan summation algorithm [8]:

\[
\hat{z}_b = \text{flt}(\hat{x}_b + \hat{y}) - \hat{y} = \text{flt}(\hat{x}_b - c)
\]

The base component \( \hat{z}_b \) is the best approximation of the sum between the two FP numbers \( \hat{x}_b \) and \( \hat{y} \). The value \( c \) is a backwards compensation that restores the digits that are "lost" when rounding \( \hat{z}_b \). Most importantly, the compensation \( c \) is computed exactly without incurring any rounding error. Then, the compensation is added into the offset \( \hat{x}_b \) and rounded to the nearest FP. Figure 8 illustrates how the truncated digits are recovered by the compensation.

![Figure 8: Example of computing the compensation \( c \) when summing the FP numbers \( \hat{x}_b \) and \( \hat{y} \). The grey numbers correspond to digits that are discarded due to truncation. The right example shows two inputs of very different size causing truncation of three bits, all of which are recovered.](image)

**Error Analysis.** As the compensation \( c \) recovers all lost bits from the initial summation, the error of the addition operation lies only in the final rounding of \( \hat{z}_b - c \). Thus, due to the non-overlapping invariant, the resulting error is at most \( \varepsilon^2 \) for the entire operation.

2. **Multiply-FP-by-FP.** In Erroll1 we need to multiply an HP number \( \hat{x} \) with an FP number \( \hat{y} \), e.g. at line 6 of figure 6. By expanding the definition for HP, we can write the output \( \hat{z} \) as:

\[
\hat{z} = \hat{x} \times \hat{y} = (\hat{x}_b + \hat{x}_c) \times \hat{y} = \hat{x}_b \times \hat{y} + \hat{x}_c \times \hat{y}
\]

(1)

The first multiplication is done using Knuth’s method of splitting each \( p \)-bit FP number into two \( \frac{p}{2} \)-bit FP numbers and performing long-form multiplication, yielding an HP result without error [9].

\[
\hat{v} = \hat{x}_b \times \hat{y}
\]

(2)

Though the second term \( \hat{x}_c \times \hat{y} \) can be also computed as an HP number, we can safely ignore the least-significant FP (i.e. the offset component), as this portion would almost entirely be lost when rounding the final \( \hat{z} \). Finally, the left term (an HP) and the right term (an FP) are added together using the previously described addition algorithm. The entire process is illustrated in Figure 9.

**Error Analysis.** There are three possible sources of error: the multiplication of the first term, the multiplication of the second term, and the final summation. The first term \( \hat{x}_b \times \hat{y} \) is computed without error, as explained above. The second term \( \hat{x}_c \times \hat{y} \) is computed with an error 

![Figure 9: Example: Multiplying a 8-bit HP number \( \hat{x} \) with a 4-bit FP number \( \hat{y} \). The base \( \hat{x}_b \) is multiplied by \( \hat{y} \) without error to produce the HP value \( \hat{v} \). The greyed values are omitted from the computation because they minimally affect the output \( \hat{z} \).](image)
\( \hat{y} \) in \( \epsilon \) is computed using native FP multiplication which incurs an error of \( \epsilon \). Due to non-overlapping, the components \( \hat{x}_b \) and \( \hat{x}_\delta \) must be related by |\( \hat{x}_b | > | \hat{x}_\delta | 2^{p-1} |, and so:

\[
|\hat{x}_b \hat{y}| > |\hat{x}_\delta \hat{y}| 2^{p-1}
\]

Hence, |\( \hat{x}_b \hat{y} | \) is smaller than the final output \( \hat{z} \) by a factor of \( 2^{-p+1} \) and so the total error caused by the second term is \( \epsilon \times 2^{-p+1} \) or \( 2\epsilon^2 \). Finally, the third source of error, the summation of the HP value \( \hat{l} \) to the unrounded FP \( \hat{x}_b \times \hat{y} \), incurs an error of \( \epsilon^2 \) as described above. Thus, in total, the multiply-HP-by-FP has relative error of \( 3\epsilon^2 \).

(c) Multiply by 10. The procedure get_digits_hp (Figure 7) frequently uses multiply by 10 to shift the decimal point to the right. Each FP component (base and offset) is multiplied by 10:

\[
\hat{h} \doteq \text{flt}(10 \times \hat{x}_b) \quad \hat{l} \doteq \text{flt}(10 \times \hat{x}_\delta)
\]

Rounding occurs for both operations; though it is tolerable for the lower product \( \hat{l} \), rounding of \( \hat{h} \) incurs a significant amount of error. Fortunately, the product is an addition of some bitshifts, the latter being multiplication by a power of two:

\[
\hat{h} \doteq 10 \times \hat{x}_b = 8 \times \hat{x}_b + 2 \times \hat{x}_b
\]

As in add-HP-to-FP, a compensated value \( c \) is backwards computed to recover the bits lost in the computation of \( h \) as an FP number:

\[
c \doteq (\hat{h} - 8 \times \hat{x}_b) - 2 \times \hat{x}_b
\]

Note that multiplication by 2 and 8 incurs no error so that the compensated value itself is computed without error. The final output \( \hat{z} \) integrates the compensation into the offset component:

\[
\hat{z}_b \doteq \hat{h} \quad \hat{z}_\delta \doteq \hat{l} - c
\]

Error Analysis. Multiply-by-10 has three sources of error: computing \( \hat{h} \), computing \( \hat{l} \), and performing the final addition. First, computing \( \hat{h} \) may incur error, but the lost bits are exactly recovered using compensation. Second, the error in \( \hat{l} \) is at most \( \epsilon \); when accounting for the size of \( \hat{l} \) compared to \( \hat{z} \), we have at most \( 2\epsilon^2 \) of error. Third, the final addition incurs an additional \( \epsilon^2 \) of error. Combined, multiply-by-10 incurs a maximum relative error of \( 3\epsilon^2 \).

(d) Divide-by-10. Division follows a similar pattern to multiplication. First, both components are divided using native FP numbers:

\[
\hat{h} \doteq \text{flt}(\hat{x}_b/10) \quad \hat{l} \doteq \text{flt}(\hat{x}_\delta/10)
\]

For multiplication, the FP value \( \hat{h} \) was approximately ten times larger than \( \hat{x}_b \); however, for division, the base component \( \hat{x}_b \) is ten times larger. Consequently, we compute the compensated value with respect to the input \( \hat{x}_b \) (instead of the output):

\[
c \doteq (\hat{x}_b - 8 \times \hat{h}) - 2 \times \hat{h}
\]

As with multiplication, the compensation is computed without error. The compensated value \( c \) corresponds to the backwards difference between the actual and exact results multiplied by ten (i.e. between \( 10 \times \hat{h} \) and \( 10 \times h \)). Notice that the exact error on the output satisfies \( h - \hat{h} = c/10 \). Unfortunately, division often produces numbers of infinite length that cannot be exactly represented by FP numbers. Consequently, the compensated value is rounded to the nearest FP value before being integrated into the result \( \hat{z} \):

\[
\hat{z}_b \doteq \hat{h} \quad \hat{z}_\delta \doteq \hat{l} - \text{flt}(c/10)
\]

Error Analysis. Error occurs during three steps in the division operation: rounding of \( l \), rounding of \( c/10 \), and error during the final addition. As with the previous operations, rounding \( \hat{l} \) incurs at most \( 2\epsilon^2 \) error, and the final addition incurs \( \epsilon^2 \) error. The rounding of the \( \hat{c}/10 \) term produces an additional error on the order of \( \epsilon^2 \). Combining all sources or error, divide-by-10 has a maximum relative error of \( 4\epsilon^2 \).

4.5 Ensuring Correctness under Rounding Errors

As shown in Figure 3, the maximum amount of error from the algorithm directly affects the selection of the narrow and wide bounds. The worst-case error for the entire Errol1 algorithm is found by summing the maximum error of every operation based on the worst-case run of the algorithm. We use this error to set \( \epsilon \) to a value that ensures that the computed boundaries correctly lie within the actual midpoints.

**Theorem 4 (Maximum Error).** Given an FP format with a maximum round-off error of \( \epsilon \), the maximum relative error of for Errol1 using HP numbers is at most \( 79\epsilon^2 \).

**Proof.** From figure 6 there are four loops that are executed a variable number of times. The worst case for small numbers occurs for inputs below \( 10^{-323} \) where the lookup table can only perform an initial multiply of \( 10^{308} \). The remaining factor \( 10^{15} \) requires 15 executions of the loop on lines 10-13 incurring a total error of \( 45\epsilon^2 \). The worst case for large numbers occurs for values above \( 10^{1008} \) where the lookup table can only perform an initial multiply of \( 10^{-293} \). The remaining factor \( 10^{-17} \) requires 17 executions of the loop on lines 6-9 incurring a total error of \( 68\epsilon^2 \). Because only a single loop is executed for a given input, the worse case error is the maximum of the two branches \( 68\epsilon^2 \).

The remaining errors occur in three places: the operations used in exponent computation, a single \( \epsilon^2 \) when computing the powers of ten lookup table, and a loss during digit generation. The digit generation portion of the algorithm loses at most 2 bits of precision (equivalent to \( 2\epsilon^2 \)). Future truncation and multiplications do not incur any errors: as digits are extracted, the number of bits shrink such that there is no further rounding. This process, graphically shown in figure 10 demonstrates a worst-case example where an input of all 1 bits is truncated and multiplied by ten to generate a number with two extra bits. Summing all sources of error, the maximum possible error from the entire algorithm is \( 79\epsilon^2 \).  

**Correctness.** Next, we must set the \( \epsilon \) to ensure that the bounds on lines 14 and 15 of Figure 6 are indeed correct (i.e. narrow). Setting \( \epsilon \) to \( n\epsilon \) accounts for \( n\epsilon^2 \) of error. Thus, for double-precision numbers with \( \epsilon = 2^{-53} \), we define \( \epsilon \doteq 8.78 \times 10^{-15} \) which lets us ensure correctness, or formally, Theorem 2.
Optimality. Although Errol1 is not guaranteed to produce the shortest output, the narrow intervals provide a close enough approximation to the rounding interval that Errol1 is guaranteed to produce a decimal output with 17 digits or less.

Theorem 5. (Maximum Length) The function convert_hp(ṽ) returns a decimal value with at most 17 digits.

We prove this result using the analysis of Matula that provides an upper bound on the length required to uniquely print a number \[ D \]. In particular, given a floating-point format with a radix \( b \) and \( p \) bits of precision, every number can be uniquely printed in decimal with \[ D = [n \log_{10} p] + 2 \] digits. We can extend Matula’s analysis to our setting (with narrow boundaries) to show that for double-precision floating-point numbers, every floating-point number can be correctly converted with no more than \( D = 17 \) digits.

Empirical Evaluation. By empirically running Errol1 on one billion inputs, we observed that all conversions were correct and 99.973\% were optimal as compared to Grisu3’s 99.5\%. Thus, Errol1 is sub-optimal for an order of magnitude fewer inputs than Grisu3.

5. Errol2: Almost Optimal

Recall (from Figure [1]) that Grisu3 and Errol1 return sub-optimal conversions if there is a number in the uncovered intervals that is shorter than the shortest number in the narrow interval. As Errol uses a more accurate HP format than Grisu3 (106-bits vs 64-bits), Errol1 is able to expand the size of the narrow interval and shrink the size of the uncovered intervals, thereby lowering the sub-optimality rate by an order of magnitude as smaller uncovered intervals are less likely to contain a shorter number. Surprisingly, we found that further expanding the narrow interval, e.g., by using even higher precision, does not improve optimality. Next, we describe how we empirically establish the above phenomenon ([5]), analytically characterize the inputs where sub-optimal outputs are possible ([5]), and how insights from the above yield Errol2, which is fast, correct and optimal on 99.9999999\% of all inputs ([5]).

5.1 Empirically Locating Optimality Failures

To empirically investigate the source of optimality failures, we randomly generated and converted one billion double-precision floating-point values and checked the results for optimality.

The Bad News: Pathological Midpoints. From the data, we observed the pattern that every observed failure was caused by a pathological midpoint (\( \hat{m} \) or \( \hat{n} \)) whose length was shorter than every number inside the exclusive interval (\( \hat{m} \) or \( \hat{n} \)). Because the midpoints are the most extreme points of the rounding interval, any narrowing – regardless of how precise – can never generate such numbers and hence, will not be optimal.

The Good News: Pathology is Contained. Fortunately, we discovered that the distribution of optimality failures forms a very unexpected and useful pattern. We split the space of inputs into 2098 bins where each bin had approximately 500,000 values drawn from the interval \( (2^{p+1}, 2^{p+2}) \). For every bin, we computed the percentage of optimality failures and show the results in Figure [1]. Curiously, we found that pathological midpoints spike around \( e = 58 \) and exponentially decay as \( e \) goes to 0. Outside the range \( 2^{58} \) to \( 2^{62} \), we observed no optimality failures.

5.2 Analytically Characterizing Optimality Failures

Next, we present a theoretical analysis that explains the curious spike, i.e., both the cause and location of pathological midpoints. First, we show that pathological midpoints must be integers, i.e., they have no bits right of the radix point. Second, we demonstrate that they become more rare as the values get larger; and after a certain point, the midpoints become so large that pathological cases become extinct. Consequently, for double-precision numbers, pathological midpoints must be integers in \([2^{54}, 2^{131}]\).

Theorem 6. (Pathological Midpoints) If \( \hat{n} \) is a pathological midpoint then: (i) \( \hat{n} \) must be an integer, and (ii) \( 2^{54} \leq \hat{n} \leq 2^{131} \).

The proof of of the above requires a few basic definitions and facts about floating point numbers.

Index Decomposition. We say that \( d_i \) is the digit at index \( i \) in \( r \) when \( r \) is expressed in position notation. For example, when \( r = 3.1415 \), we have \( d_0 = 3 \) and \( d_4 = 5 \), i.e. 3 and 5 are the digits at index 0 and \( -4 \) respectively. Thus, every real \( r = \sum_{k=-\infty}^{\infty} d_k 10^k \) where \( d_k \) is the digit at index \( k \) in \( r \).

Leftmost & Rightmost Index. For finite-length numbers, let:

\[ N(r) = \max \{ k \mid d_k = 0 \} \quad M(r) = \min \{ k \mid d_k \neq 0 \} \]

We call \( N(r) \) (resp. \( M(r) \)) the left-most (resp. right-most) index of \( r \). The left-most index is exactly computable as: \( N(r) = \lceil \log_{10} r \rceil \). The right-most index is trickier in general but for integers is simply the number of trailing zeros, i.e., the maximum number of times the integer is evenly divisible by 10: \( M(z) = f_{10}(z) \) where \( f_{10}(z) \) is the multiplicities of the factor 10 of \( z \). For example, \( M(12300) = f_{10}(12300) = 2 \).

Length of a Number. The length of a real number \( r \in \mathbb{R} \), written \( L(r) \), is defined as: \( L(r) = N(r) - M(r) + 1 \). Thus, \( L(r) \) is the minimum number of digits necessary to write out the significand of \( r \) in decimal. Note that multiplication by (powers of) 10 does not effect a number’s length; i.e. for any integer \( n \), we have \( L(r) = L(r \times 10^n) \). For example, \( L(1.23) = L(1.23 \times 10^{-6}) = 3 \).

Lemma 1. (Shifting) In a floating-point format with \( p \) bits of precision, if \( \hat{v} \) is a floating point number, and \( \hat{m} \) is a midpoint adjacent to \( \hat{v} \) then there exists: (1) an integer \( e \), (2) a natural number \( z_e < 2^{p+1} \) and (3) an odd natural number \( 2^{p+1} \leq z_n \leq 2^{p+2} \), such that: \( \hat{v} = z_e 2^e \) and \( \hat{m} = z_n 2^e \).

To see that \( \hat{v} = z_e 2^e \), we need only shift the binary significand of \( \hat{v} \) at most \( p \) digits to the right until it is a natural. The fact that \( z_n \) is odd follows from averaging two adjacent floating-point numbers \( \hat{v} \) and \( \hat{v}^{'} \) after writing them in the above shifted format; the average of two adjacent integers is an irreducible rational of the form \( z_n \) and the factor two is incorporated into the \( 2^e \) term.

Proof of Theorem 0 (i): \( \hat{m} \) must be an Integer. We prove this case by contradiction. Suppose that midpoint \( \hat{m} \) is a non-integer rational i.e., has a fractional component (\( e < 0 \)). By Lemma 1, we have \( \hat{m} = z_n 2^{-e} \) where \( e = -e \) is nonnegative. Hence:

\[ L(\hat{m}) = L(z_n 2^{-e}) = L(z_n) - \log_{10} 2^e = L(z_n) 2^e \]

where, by Lemma 1, \( z_n \) is an odd integer. Because \( 5^e \) is also odd, \( z_n 5^e \) must be odd, and therefore not divisible by 10. This implies that the right-most index \( M(\hat{m}) = 0 \). Via the logarithmic form of \( N(r) \) we have:

\[ L(\hat{m}) = N(\hat{m}) - M(\hat{m}) + 1 = \lceil \log_{10} z_n 5^e \rceil + 1 \]

By Lemma 1 the natural \( z_n \geq 2^{p+1} \), so:

\[ L(\hat{m}) \geq [(p + 1) \log_{10} 2 + e \log_{10} 5] + 1 \]

For double-precision floating-point numbers \( p = 53 \), so every non-integer midpoint has at least 17 digits. By the Maximum Length...
Theorem 5. Errol1 will produce an output of 17 digits or less, and so the non-integer midpoint $\tilde{m}$ cannot be pathological.

Proof of Theorem 5 (ii): $2^{54} \leq \tilde{m} \leq 2^{131}$. The lower bound $2^{54} \leq \tilde{m}$ follows as $\tilde{m}$ must be an integer. Recall that the right-most index $M(\tilde{m})$ is determined by multiplicities of factors ten, which further split into prime factors five and two. Hence, we have:

$$L(\tilde{m}) = \left\lfloor \log_{10} \tilde{m} \right\rfloor - \min(f_5, f_2) + 1$$

where $f_5$ and $f_2$ are the multiplicities of prime factors five and two (of $\tilde{m}$). By the properties of $\min$, the above implies:

$$L(\tilde{m}) \geq \left\lfloor \log_{10} \tilde{m} \right\rfloor - f_5 + 1$$

From Lemma 1, only the $z_{\tilde{m}}$ term can contain factors of five, and it can have at most $f_5 \leq \lfloor \log_5 2^{p+2} \rfloor$ factors, and so:

$$L(\tilde{m}) \geq \left\lfloor \log_{10} \tilde{m} \right\rfloor - \lfloor \log_5 2^{p+2} \rfloor + 1$$

That is, the length $L(\tilde{m})$ grows with the size of the midpoint $\tilde{m}$. By the Maximum Length Theorem 5, Errol1 always prints decimals with 17 digits or less for double-precision numbers, so midpoints $\tilde{m}$ cannot be pathological if:

$$L(\tilde{m}) \geq \left\lfloor \log_{10} \tilde{m} \right\rfloor - \lfloor \log_5 2^{54} \rfloor + 1 \geq 17$$

By solving the minimum $\tilde{m}$ for which the above equation holds, we conclude that midpoints larger than $2^{131}$ cannot be pathological.

5.3 Handling Pathological Midpoints

The Pathological Midpoint Theorem 6 guarantees that midpoints are pathological only if they fall within a pathological range corresponding to midpoints that are small whole numbers between $(2^{54}, 2^{131})$ (in our double-precision floating-point setting).

Errol2. Based on these properties, we designed the second iteration of our algorithm Errol2 which behaves as follows. If the input $\tilde{v}$ is outside the pathological range, Errol2 converts $\tilde{v}$ using the Errol1 algorithm. When the input $\tilde{v}$ is in the pathological range $(2^{54}, 2^{131})$, it is possible that one of the adjacent midpoints is pathological. Consequently, Errol2 computes the midpoints $\tilde{m}^{-}$ and $\tilde{m}^{+}$ exactly as an integer, using well known techniques for binary to decimal conversion for integers [9]. Errol2 then computes the output digits by invoking `digits_hp` (Figure 7) on the exact midpoints. As there is no narrowing, Theorem 5 ensures that the output digits are correct and optimal.

Empirical Evaluation. Note that Errol2 does not guarantee optimal conversion; narrowing outside the pathological range may still yield a sub-optimal output. However, we ran Errol2 on the sample set of one billion random inputs, and we observed zero optimality failures outside the pathological range (and of course, zero inside the range, thanks to the exact midpoints). Therefore, we have empirically tested Errol2 to have an accuracy of approximately 99.999999999% or better.

6. Errol3: Always Optimal

A 99.9999999% optimality rate is not bad, but why leave anything to chance? Next, we present the final refinement, Errol3, which guarantees correct and optimal conversion for all inputs.

Pathological Inputs and Outputs. A pathological input is an FP value for which the optimal (shortest) decimal is found extremely close to the midpoint (in the uncovered interval) as shown in Figure 1. The decimal output corresponding to a pathological input is called a pathological output. Note that the Maximum Length Theorem 5 implies that for double-precision floating-point arithmetic, pathological outputs have fewer than 17 digits.

Optimality via Enumeration. Errol3 is founded upon two key insights. First, we identify necessary conditions, a set of modular arithmetic constraints, that characterize the midpoints whose neighborhoods contain pathological inputs and outputs (§6.2). Second, we provide an efficient algorithm to efficiently enumerate all the solutions to the modular arithmetic constraints, thereby tabulating all the possible pathological inputs and their corresponding outputs (§6.3). Thus, given an arbitrary $\tilde{v}$, Errol3 simply checks if it is one of the pathological inputs and if so, returns its tabulated output. Otherwise, it computes using a modified version of Errol2 (§6.4).

6.1 Preliminaries

Our analysis partitions the input space by the (binary) exponent $e$, i.e. into sub-ranges comprising the intervals $(2^e, 2^{e+1})$, which we call the input range of $e$. That is, for a given input range, the exponent $e$ is a fixed constant. For double-precision floating-point numbers, there are 4096 possible exponents $e$, including subnormal...
numbers. Our characterization and enumeration algorithm finds all pathological inputs by iteratively searching all possible exponents. In the sequel, we assume a fixed exponent $e$ and its input range, and recall that $p$ denotes the number of bits of precision of the given format.

**Enumerating Inputs & Midpoints.** The first number (resp. midpoint) in the input range is named $\hat{v}_0 = 2^e$ (resp. $\hat{m}_0 = 2^e + 2^{e-p+1}$). The spacing between floating-point numbers (resp. midpoints) is exactly $2^{e-p}$. Therefore, the $k^{th}$ number (resp. midpoint) in the input range, where $k$ is a natural number less than $2^p$, is $\hat{v}_k = 2^e + k \times 2^{e-p}$ (resp. $\hat{m}_k = 2^e + 2^{e-p+1} + k2^{e-p}$).

**Pathological Range.** The pathological range denotes an area around the midpoint $\hat{m}_k$ that may contain pathological inputs as shown in Figure 12. The size of the pathological range is defined by the error factor, and is:

$$\left(\hat{m}_k - 2^{e-p} \varepsilon, \hat{m}_k + 2^{e-p} \varepsilon\right)$$

Recall that by the Maximum Error Theorem, Erroll1 has a relative error of at most $79\varepsilon$. Thus, by setting $\varepsilon \geq 79\varepsilon$, the pathological range is guaranteed to cover the amount of error incurred by Erroll1 (a factor of $\epsilon$ is lost from the factor $2^{-e}$).

**Pathological Outputs.** A pathological output $r$ has the form:

$$r = \hat{m}_k + \sigma 2^{e-p} = 2^e + 2^{e-p} + k2^{e-p} + \sigma 2^{e-p}$$

$$= 2^{e-p-1}(2^{e+1} + 1 + 2k + 2\sigma)$$

where $|\sigma| < \varepsilon \leq 79\varepsilon$ (3)

A value of $\sigma = 0$ indicates that $r$ is exactly a midpoint. By Theorem 8 outside of the pathological midpoint range, $r$ can only be pathological if $\sigma \neq 0$.

**Minimal Congruence.** The congruence class $z \pmod{n}$ consists of all values $v_k = z + kn$ generated by the integers $k$. The minimal congruence, written $M_z(z, n)$, is the smallest value from the congruence class $z \pmod{n}$. We write $M_z^+ (z, n)$ (resp. $M_z^- (z, n)$) for the smallest non-negative (resp. largest non-positive) value from the congruence class $z \pmod{n}$.

6.2 **Characterizing Pathologies**

To characterize pathological inputs and outputs, we partition the input space into those with (i) integer midpoints, i.e. inputs greater than $2^{p+1}$, and (ii) non-integer midpoints, i.e. inputs less than $2^{p+1}$.

**Integer Midpoints.** The following theorem provides a necessary condition that must hold for an integer midpoint $\hat{m}_k$ to have a pathological output $r$ in its neighborhood.

**Theorem 7 (Integer Midpoints).** A midpoint $\hat{m}_k > 2^{p+1}$, has a pathological output $r$ in its neighborhood only if

$$|M_z^+(\hat{m}_k, 5^n)| \leq 2^{e-p-n}$$

where $n = \lceil \log_{10} r \rceil - D + 2$ and $\hat{m}_k = \hat{m}_k 2^{-n}$.

**Proof.** First, we show that a pathological output $r$ must be an integer. Suppose not, i.e. that $r$ is not an integer. Then its right-most index $M(r) < 0$ and its left-most index $N(r) \geq \lceil \log_{10} 2^{p+1} \rceil$. Combined, the length $L(r) > \lceil \log_{10} 2^{p+1} \rceil + 1$. The Maximum Length Theorem implies that such an $r$ is too long to be optimal. Therefore, a pathological $r$ must be an integer.

As done in the analysis of pathological midpoints, we write the length of $r$ in terms of the left-most index and multiplicities of factors

$$L(r) = \lceil \log_{10} r \rceil - \min(f_2, f_3) + 1 \leq D - 1$$

where the latter inequality arises from Theorem 9. Thus, we have a lower bound on the number of prime factors two and five:

$$\min(f_2, f_3) \geq \lceil \log_{10} r \rceil - D + 2$$

Let $n$ abbreviate $\min(f_2, f_3)$. Then $r \equiv 0 \pmod{2^n5^n}$ and so

$$r2^{-n} \equiv 0 \pmod{5^n}$$

Based on the definition of a pathological output, $\sigma$ is an extremely small, non-zero value. As $\sigma < 1$, $\sigma$ is not an integer, which implies that $r$ cannot divide $2^{e-p}$. Thus, $r$ divided $2$ at most $e - p + 1$ times, or $n \leq e - p + 1$. By the midpoint definition, $\hat{m}_k$ divides $2^{e-p} - 1$, and hence, $2^e$. Hence, we create the normalized output $\hat{r} = r2^{-e}$ and the normalized midpoint $\hat{m}_k = \hat{m}_k 2^{-e}$ related by $\hat{r} = \hat{m}_k + \sigma 2^{e-p}$. This lets us write $\hat{m}_k$ as a congruence class $\hat{m}_k \equiv -\sigma 2^{e-p-n}$ (mod $5^n$). As $\sigma$ is bounded by $\varepsilon$, there must be a minimal congruence $|M_z(\hat{m}_k, 5^n)| \leq 2^{e-p-n}$.

**Non-Integer Midpoints.** The following theorem provides a necessary condition that must hold for an non-integer midpoint $\hat{m}_k \leq 2^{p+1}$ to have a pathological output $r$ in its neighborhood.

**Theorem 8 (Non-Integer Range).** A midpoint $\hat{m}_k < 2^{p+1}$ has a pathological output $r$ in its neighborhood only if

$$|M_z(\hat{m}_k, 5^n)| \leq 5^{e-p-n}$$

where $n = \lceil \log_{10} r \rceil - D + 2$ and $\hat{m}_k = \hat{m}_k 5^{-n}10^{-e-p+1}$.

**Proof.** In order to apply the previous integer techniques, the midpoints $\hat{m}_k$ are scaled by powers of ten to produce integers

$$\hat{m}_k = \hat{m}_k 10^{-e+p+1} = 5^{-e-p+1}2^{e+1} + 5^{-e+p+1} + 2k5^{-e+p+1}$$

such that $\hat{m}_k' \in \mathbb{Z}$ (shown by applying the Midpoint Format Definition). By construction, their lengths are equal, i.e. $L(\hat{m}_k') = L(\hat{m}_k)$. A pathological output $r$ must have the form

$$r = 2^{e-p} + k2^{e-p} + \sigma 2^{e-p}$$

In order to relate $r$ to $\hat{m}_k'$, we construct a modified output $r'$ as

$$r' = r10^{-e+p+1}$$

$$= \hat{m}_k' + 2\sigma 5^{-e+p}$$

$$= 5^{-e+p+1}2^{e+1} + 5^{-e+p+1} + 2k5^{-e+p+1} + 2\sigma 5^{-e+p}$$

so that $L(r') = L(r)$. Using the same reasoning as the Integer Range Theorem, we conclude that $r'$ must be an integer.

Following the same logic as the previous section by substituting the modified output $r'$ for the original $r$, we obtain the relation

$$\min(f_2, f_3) \geq \lceil \log_{10} r' \rceil - D + 2$$

Again, we use the name $n$ for the minimum number of factors of two and five, and obtain the congruence relations

$$r' \equiv 0 \pmod{2^n5^n} \Rightarrow r'2^{-n} \equiv 0 \pmod{5^n}$$
By construction, $5^n$ must divide $r'$. Because $\sigma$ is neither zero nor an integer, $r'$ cannot divide $5^{e+p+1}$ implying $n < -e + p + 1$. Because $\tilde{m}_k$ divides $5^{e+p+1}$, it must also divide $5^n$. This fact serves as the basis for creating the normalized output $\tilde{r} = r'5^{-n}$ and the normalized midpoint $\tilde{m}_k = \tilde{m}_k 5^{-n}$ which are related by:

$$\tilde{r} = \tilde{m}_k + r 5^{-e-p-n}$$

This previous equation allows us to write $\tilde{m}_k$ as a congruence class

$$\tilde{m}_k \equiv \sigma 5^{-e-p-n} \pmod{2^n}$$

As the value of $\sigma$ is bounded in terms of $e$, there must be a minimal congruence such that: $|M_k.(\tilde{m}_k, 2^n)| \leq 5^{-e-p-n} \square$

6.3 Enumerating Pathologies

Theorems 7 and 8 provide necessary conditions for a midpoint $\tilde{m}$ to have a pathological output in its interval. Consequently we can phrase the problem of enumerating pathological inputs as computing the solutions of a system of pathological constraints.

The Pathological Constraint Problem. Given (1) an arithmetic sequence $(m_0, m_1, \ldots)$ whose $k^{th}$ element defined by an initial (normalized midpoint) $m_0$ and spacing factor $\alpha$ such that $m_k = m_0 + k \times \alpha$; (2) a modulus $\tau$; (3) and a threshold $\Delta$ the pathological constraint problem is to compute the set of points $m_k$ such that $|M_k.(m_k, \tau)| \leq \Delta$. Exhaustive testing of all midpoints is computationally infeasible for many floating-point formats including double-precision numbers. Instead, we developed an algorithm that finds the (maximal) subsequence of midpoints such that every successive $M_k.(m_k, \tau)$ is smaller than the last. In this manner, the algorithm quickly converges on the midpoints that satisfy the pathological constraint.

Offsets. An offset is the component of a midpoint that takes the form $x_k = k \alpha$. The offset component forms a linear relationship where $x_i + x_j = x_{i+j}$. An offset $x_j$ can be added to a midpoint $m_i$ to form subsequent midpoints $m_i + x_j = m_{i+j}$. There are two elements of the congruence class $m_{i+j} \mod \tau$ of importance: the first real number above or equal to $z^+ = M_k.(m_i, \tau)$ and the first real number below or equal to $z^- = M_k.(m_i, \tau)$. Based on this interpretation, adding $x_j$ can be seen as a shift upward to larger real $z^+$ or a shift downward to the smaller real $z^-$. For a given offset $x_j$, the downward shift and upward shift are respectively defined as:

$$x_j^+ = M_k^+(x_j, \tau) \quad x_j^+ = M_k^+(x_j, \tau)$$

Using the idea of shifts, the overall goal is restated as: starting at an initial $m_0$, find an optimal sequence of shifts that generate successive midpoints closer to zero.

Optimal Shift Sequences. The optimal sequence of upward shifts $X^+$ is defined as the lexicographically smallest subsequence of $(x_0^+, \ldots, x_N^+)$ that is decreasing in magnitude; i.e., for any two adjacent elements $x_i^+, x_j^+ \in X^+$, there is no $x_k^+$ such that $i + k < j$ and $x_k^+ < x_j^+ < x_j^+$. The optimal sequence of downward shifts $X^-$ is similarly defined with respect to sequence $(x_0^-, \ldots, x_N^-)$.

Optimal Sequence Construction. We begin with the initial sequences $X_0^+ = (x_0^+, \ldots, x_N^+)$ and $X_0^- = (x_0^-, \ldots, x_N^-)$, and by inductively extending them. Without loss of generality, assume $j \leq k$. Given the two optimal sequences $X_j^+$ and $X_j^-$, the next element is constructed in the following manner: select the last element $x_i^+ \in X^+$ and select first element $x_i^- \in X^- \in X^+$ such that $x_j^+ + x_i^- > x_i^-$; then, the generated element is the sum $x_k = x_i^+ + x_i^-$. This element $x_k$ represents either a negative shift ($x_k \leq 0$) or a positive shift ($x_k \geq 0$) and is added to the appropriate sequence to create either $X_k^+$ or $X_k^-$. 

Lemma 2. Given two upward shifts $x_i^+$ and $x_j^+$, the difference $x_i = x_i^+ - x_j^+$ must be either an upward shift if $x_i \in (0, \tau)$ and a downward shift if $x_i \in (-\tau, 0)$. If $x_i$ is zero, it is both an upward and downward shift. Symmetrically for two downward shifts $x_i^-$ and $x_j^-$, their difference must also be either an upward or downward shift or both.

Proof. The upward shifts must lie in the range $x_i^+, x_j^+ \in (0, \tau)$, therefore their difference $x_i$ must lie in the range $(-\tau, \tau)$. Symmetrically, the difference between two downward shifts lie in the same range $(-\tau, \tau)$. The difference $x_i$ falling in the range $(-\tau, \tau)$, is either an upward or downward shift or both.

Lemma 3. Given natural numbers $j$ and $c$ such that $j < c$ and shifts $x_j^+$ and $x_j^+$ such that $x_j^+ < x_j^+$, then there exists a downward shift $x_{c-j}^+ = x_j^+ - x_j^+$. Symmetrically, there exists an equivalent upward shift $x_{c-j}^- = x_j^+ - x_j^-$.

Proof. By Lemma 2 the resulting shift $x_{c-j}$ must exist and be either a downward or upward shift. Because $x_j^+ < x_j^+$, the shift $x_{c-j}$ must be negative and therefore is a downward shift. The same argument shows the difference of equivalently defined downward shifts is an upward shift.

Lemma 4. Given an arbitrary shift $x_c$ that is not found within the optimal sequence of shifts $X$, then there exists an optimal shift $x_d$ such that $d < c$ and $|x_d| < |x_c|$.

Proof. Let $x_d$ be the first element in the optimal sequence $X$ that precedes $x_c$. By construction, $d < c$. Assume $|x_d| \geq |x_c|$, then $x_c$ must be an optimal shift and found in $X$, thus reaching a contradiction. Thus, $x_d$ satisfies the properties of the optimal shift.

Theorem 9. If $X_k^-$ and $X_k^+$ are optimal subsequences then the inductive extensions $X_k^+$ or $X_k^-$ are also optimal.

Proof. Without loss of generality, assume $x_i$ is an upward shift. Assume $X_k^+$ is suboptimal, i.e., there is a $x_i^+$ such that $j < c < z$ and $x_j^+ < x_j^+$. By Lemma 2 there exists a downward shift $x_{c-j}^+ = x_j^+ - x_j^+$. Since $c < z$, the $x_{c-j}$ shift occurs earlier than the selected $x_{c-j} = x_n$ shift. If $x_{c-j}$ is in $X^-$, then the algorithm failed to select the first available downward shift, thus reaching a contradiction. If $x_{c-j}$ is not in $X^-$, then by Lemma 2 there exists an optimal shift $x_{c-j}$ in $X^-$ that the algorithm failed to select, thus reaching a contradiction.

Midpoint Search. The midpoint search begins at the initial midpoint $m_0$ and is performed inductively. Given a midpoint $m_0$ that is positive, we select the first offset $x_{c} \in X^-$ such that the next midpoint $m_{k+1} = m_k + x_c$ is closer to zero $|M_k.(m_{k+1}, \tau)| < |M_k.(m_k, \tau)|$.

Theorem 10. After the midpoint $m_k$, the generated midpoint $m_{k+1}$ is the first midpoint whose minimal congruence is closer to zero.

Proof. By construction, any different shift from $X^-$ would create a midpoint with too large or small a minimal congruence. Suppose there exists a larger shift $x_0 \notin X^-$ that creates a midpoint satisfying the condition $|M_k.(m_{k+1}, \tau)| < |M_k.(m_k, \tau)|$. Based on this construction, $c$ must be larger than $i$ (in order to preserve the optimal sequence definition), which indicates that $m_{k+c}$ is not the first midpoint closer to zero. By contradiction, the midpoint $m_{k+c}$ is the first such midpoint.
Symmetrically, if \( m_k \) is negative, we select the first offset \( x_k^+ \in X^+ \) such that the next midpoint is nonpositive. In this manner, we generate a sequence of midpoints that “ping-pong” back and forth across zero, always decreasing in magnitude. In the event that a midpoint satisfying \( |M(m_k, \tau)| \leq \varepsilon \) is found, the above process no longer applies; instead, midpoints are more exhaustively searched by performing all shifts that land within the range \([-\varepsilon, \varepsilon]\). This ensures that all midpoints satisfying midpoints are found.

Handling Subnormals. Because the above has assumed an arbitrary floating-point format with \( p \) bits of precision, subnormal numbers are easily encoded by modifying the value of \( p \) based on the exponent \( e \). In this manner, the enumeration algorithm searches the subnormal number ranges for pathological inputs.

6.4 Guaranteeing Optimal Conversion

Before running the enumeration algorithm, the underlying algorithm of Errol2 was modified to remove narrowing and widening. Remember, the purpose of narrowing (and widening) is to guarantee the algorithm always generates a correct (or optimal) result. This step is no longer necessary for Errol3 since all suboptimal or incorrect results are enumerated a priori. Hence, Errol3 obtains a performance benefit because verifying outputs at runtime is no longer necessary.

After removing the narrowing and widening steps, we used the enumeration algorithm to generate a list of possibly incorrect or suboptimal inputs. Each input was run using Errol2 to enumerate a complete list of inputs that do not generate correct and optimal outputs. In total, we found that only 45 inputs (of the nearly 2^{64} inputs) generate incorrect or suboptimal results; all other inputs are guaranteed to generate correct and optimal output. In order to correctly and optimally handle the failing inputs, they are hard-coded into a lookup table that maps the failing inputs to correct and optimal outputs. Combining the special handling of integers and this lookup table, Errol3 is guaranteed to be correct and optimal without runtime checks.

7. Performance Evaluation

Next, we evaluate the performance of Errol3 with the goal of comparing it against previous state of the art algorithms.

Methodology. Our experiment compares Errol3, Grisu3, and an updated version of Dragon4. For each algorithm, we measure performance by recording the time in clock cycles taken to convert a floating-point representation to a decimal string. Inputs are randomly generated IEEE-754 double-precision floating-point numbers. Outputs consist of a decimal string significand and an integer exponent. All experiments were performed on an Intel(R) Core(TM) i7-3667U CPU at 2.00GHz running Xubuntu Linux 14.04 and compiled with gcc 4.8.2. Each algorithm is tested using 20,000 random inputs with each input converted 100 times, of which the median 80 values were averaged to compute the final result. The source code can be downloaded at https://github.com/marcandrysco/Errol. The performance numbers are shown in Figure 15 which we look at part by part.

Errol3 across all inputs. Figure 15(a) shows the performance of Errol3 in isolation. We can see that its performance across the input space is fairly constant, except for three anomalies. First, Errol3 incurs a slowdown for values larger than \( 10^{291} \). This is caused by the fact that the lookup table for powers-of-ten can only store numbers down to \( 10^{-291} \), and so inputs above that point must be normalized by iteratively dividing by ten, incurring a performance penalty. Second, Errol3 incurs two slowdowns for values smaller than \( 10^{-292} \). This is caused by significant processor slowdowns when operating on subnormal numbers, and the powers-of-ten lookup table can only store numbers down to \( 10^{-308} \). Finally, there is a sizable performance anomaly for numbers in the range \( 2^{53} \) to \( 2^{127} \). These are the numbers converted using the integer algorithm which only applies to numbers in that range.

Errol3 vs. Grisu3. Figure 15(b) compares the performance of Grisu3 (in black) and Errol3 (in grey). Across the entire input space, Errol3 is, on average, 2.4× slower than Grisu3.

Errol3 vs. Dragon4. Figure 15(c) shows Dragon4 (in black) and Errol3 (in grey). Furthermore, we can also see that the performance of Dragon4 varies significantly across the input space, forming a “V” shape that increases linearly as numbers get further away from 0. In contrast, both Errol3 and Grisu3 have much less variation in performance across the input space. Across the entire input space, Errol3 is, on average, 5.2× faster than Dragon4.

Errol3 vs Grisu3-with-fallback. Finally, because Grisu3 fails to generate optimal outputs for approximately 0.5% of inputs, we consider a version of Grisu3 which falls back to Dragon4 when Grisu3 fails to generate an optimal output. Figure 15(d) shows this “Grisu3 with fallback” (in black) and Errol3 (in grey). On average, Errol3 is 2.4× slower than Grisu3 with fallback.

Performance across Architectures. As Errol3 uses floating-point operations instead of the integer operations used by Grisu3 and Dragon4, its performance compared to those algorithms is dependent on the relative speed of the floating-point operations versus integer operations on a given architecture. Thus, testing Errol3 on a variety of architectures produced an appreciable variation in performance numbers. For example, relative to Dragon4, we observed results ranging from the 5.2× speedup on an Intel(R) Core(TM) i7-3667U to a 5.8× on an Intel Xeon X5660.

8. Related Work

Coonen published an implementation guide for IEEE-754 floating-point arithmetic detailing an early binary-to-decimal conversion algorithm, and later expanded on conversion algorithms, covering both correctly rounded and “imperfect” conversions.

Steele and White published a paper for printing-floating point numbers that detailed their Dragon4 algorithm. Dragon4 provided strong guarantees that the output is both correct and optimal, a process that utilized large integer arithmetic. Later work by Gay and Burger provided performance improvements over the vanilla Dragon4 algorithm.

Loitsch published the Grisu3 algorithm, introducing a fast conversion method that guarantees correct but possibly suboptimal output. Unlike previous work, he established a method for verifying the output, and in the 0.05% of cases when the result is suboptimal, the algorithm returns a flag indicating failure. Although Grisu3 significantly outperforms Dragon4 and its successors, Grisu3 still relies on predecessor algorithms as a fallback when it fails to generate optimal outputs.

The use of high-precision floating-point data types consisting of multiple, non-overlapping components can be found in previous literature by Knuth, Dekker, Priest, and Shewchuk. Previous research by Hida has demonstrated algorithms for supporting a large range of arithmetic operations on floating-point numbers consisting of up to four double-precision numbers.

Acknowledgements

We thank the anonymous reviewers for this paper and for a previous version, for their invaluable feedback and suggestions, in particular, for providing a reference to David Matula’s In-and-out Conversions which greatly simplified the proof of the Maximum
Figure 13: Performance results, shown in black, for (a) Errol3, (b) Grisu3, (c) Dragon4 and (d) Grisu3 with fallback to Dragon4. Plots (b), (c), and (d) also show Errol3 in grey for reference. Each point plots a single value tested against all algorithms; the x-axis provides the magnitude of the value and the y-axis gives the number of clock cycles required to complete the conversion.

Length Theorem. This work was supported by NSF grant CNS-1514435 and a generous gift from Microsoft Research.

References