Easy Accurate Reading and Writing of Floating-Point Numbers

Aubrey Jaffer[1]

August 2018

Abstract

Presented here are algorithms for converting between (decimal) scientific-notation and (binary) IEEE-754 double-precision floating-point numbers. By employing a rounding integer quotient operation these algorithms are much simpler than those previously published. The values are stable under repeated conversions between the formats. Unlike Java-1.8, the scientific representations generated use only the minimum number of mantissa digits needed to convert back to the original binary values.

Introduction

Articles from Steele and White[SW90], Clinger[Cli90], and Burger and Dybvig[BD96] establish that binary floating-point numbers can be converted into and out of decimal representations without losing accuracy while using a minimum number of (decimal) significant digits. Using the minimum number of digits is a property which Java-1.8 does not achieve (10^{23} prints as `9.999999999999999E22`; 8×10^{-323} prints as `7.9E-323`); the `doubleToString` procedure presented here produces only minimal precision mantissas.

The lossless algorithms from these papers all require high-precision integer calculations, although not for every conversion.

In *How to Read Floating-Point Numbers Accurately*[Cli90] Clinger astutely observes that successive rounding operations do not have the same effect as a single rounding operation. This is the crux of the difficulty with both reading and writing floating-point numbers. But instead of constructing his algorithm to do a single rounding operation, Clinger and the other authors follow Matula[Mat68, Mat70] in doing successive roundings while tracking error bands.

The algorithms from *How to Print Floating-point Numbers Accurately*[SW90] and *Printing floating-point numbers quickly and accurately*[BD96] are iterative and complicated. The read and write algorithms presented here do at most 2 and 4 BigInteger divisions, respectively[2]

Over the range of IEEE-754[IEE85] double-precision numbers, the largest intermediate BigInteger used by these power-of-5 algorithms is 242 decimal digits (803 bits). Steele and White[SW90] report that the largest integer used by their algorithm is 1050 bits. These are not large for BigIntegers, being orders of magnitude smaller than the smallest precisions which get speed benefits from FFT multiplication.

Both Steel and White[SW90] and Clinger[Cli90] claim that the input and output problems are fundamentally different from each other because the floating-point format has a fixed precision while the decimal representation does not. Yet, in the algorithms presented here, BigInteger rounding divisions accomplish accurate conversions in both directions.

While the read algorithm tries the division yielding the longer precision quotient first, and retries only if it doesn't fit into the mantissa, the write algorithm tries the shorter precision division first and retries only when the shorter precision fails to read back correctly.

[1] Digilant, 2 Oliver Street Suite 901, Boston, MA 02109. Email: `agj@alum.mit.edu`

[2] Writing exact powers of two takes up to 6 BigInteger divisions, but there are only 2100 of them in the IEEE-754 double-precision range; they could be precomputed.

BigIntegers

Both reading and writing of floating-point numbers can involve division of numbers larger than can be stored in the floating-point registers, causing rounding at unintended steps during the conversion.

BigIntegers (arbitrary precision integers) can perform division of large integers without rounding. What is needed is a BigInteger division-with-rounding operator, called `roundQuotient` here. For positive operands, it can be implementated in Java as follows:

```
public static BigInteger roundQuotient(BigInteger num, BigInteger den) {
   BigInteger quorem[] = num.divideAndRemainder(den);
   int cmpflg = quorem[1].shiftLeft(1).compareTo(den);
   if (quorem[0].and(BigInteger.ONE).equals(BigInteger.ZERO) ?
       1==cmpflg : -1<cmpflg)
       return quorem[0].add(BigInteger.ONE);
   else return quorem[0];
}
```

If the remainder is more than half of the denominator, then it rounds up; if it is less, then it rounds down; if it is equal, then it rounds to even. These are the same rounding rules as the IEEE Standard for Binary Floating-Point Arithmetic[IEE85].

For the algorithms described here the value returned by roundQuotient always fits within a Java long. Having roundQuotient return a Java `long` integer turns out to execute more quickly than when a BigInteger is returned.

```
public static long roundQuotient(BigInteger num, BigInteger den) {
   BigInteger quorem[] = num.divideAndRemainder(den);
   long quo = quorem[0].longValue();
   int cmpflg = quorem[1].shiftLeft(1).compareTo(den);
   if ((quo & 1L) == 0L ? 1==cmpflg : -1<cmpflg) return quo + 1L;
   else return quo;
}
```

In the scaled twos-complement encoding of the mantissa, the representation of 5 and 10 are the same; the exponent differs by one. The same is true of any non-negative integer power of 5 and 10.

In the algorithms below, `bipows5` is an array of 326 BigInteger successive integer powers of 5. Constant `dblMantDig` is the number of bits in the mantissa of the normalized floating-point format (53 for IEEE-754 double-precision numbers). Constant `llog2` is the base 10 logarithm of 2.

Reading

The `MantExpToDouble` algorithm computes the closest (binary) floating-point number to a given number in scientific notation by finding the power-of-2 scaling factor which, when combined with the power-of-10 scaling specified in the input, yields a rounded-quotient integer which just fits in the binary mantissa (having `dblMantDig` bits).

The first argument, `lmant`, is the integer representing the string of mantissa digits with the decimal point removed. The second argument, `point`, is the (decimal) exponent less the number of digits of mantissa to the left of the decimal point. If there was no decimal point it is treated as though it appears to the right of the least significant digit. Thus `point` will be zero when the floating-point value equals the integer `lmant`.

When `point` is non-negative, the mantissa is multiplied by $5^{\texttt{point}}$ and held in variable `num`. If `num` fits within the binary mantissa, `num.doubleValue()` converts to the correct double-precision mantissa value and `Math.scalb` scales by `point` bits. Otherwise `MantExpToDouble` calls `roundQuotient` to divide and round to `dblMantDig` bits. Because the divisor is a power-of-2, the number of bits in the quotient is one more than the difference of the number of bits of dividend and divisor.

With a negative `point`, the mantissa will be multiplied by a power of 2, then divided by $\texttt{scl} = 5^{-\texttt{point}}$. To scale by $2^{-\texttt{point}}$, `point` is added to `bex` to form the binary exponent for the returned floating-point number.

The integer quotient of a n-bit positive integer and a smaller m-bit positive integer ($0 < m < n$) will always be between $n - m$ and $1 + n - m$ bits in length. Because rounding can cause a carry to propagate through the quotient, the longest integer returned by the `roundQuotient` of a n-bit positive integer and a smaller m-bit positive integer is $2 + n - m$ bits in length, for example $\texttt{roundQuotient}(7, 2) \to 4$. If this happens for some power-of-five divisor (which is close to a power of 2) then it must happen when the dividend is the largest possible n-bit integer, $2^n - 1$.

Over the double-precision floating-point range (including denormalized numbers) there are only 2100 distinct positive numbers with mantissa values which are all (binary) ones ($2^n - 1$); testing all of them finds that in doing double-precision floating-point conversions, there is no integer power-of-5 close enough to an integer power-of-2 which, as divisor, causes the quotient to be $2 + n - m$ bits in length.

Thus the longest a rounded-quotient of a n bit integer and a m bit power-of-5 can be is $1 + n - m$ bits; the shortest is $n - m$ bits. This means that no more than 2 rounded-quotients need be computed in order to yield a mantissa which is `mantlen` bits in length.

```
public static double MantExpToDouble(long lmant, int point) {
    BigInteger mant = BigInteger.valueOf(lmant);
    if (point >= 0) {
        BigInteger num = mant.multiply(bipows5[point]);
        int bex = num.bitLength() - dblMantDig;
        if (bex <= 0) return Math.scalb(num.doubleValue(), point);
        long quo = roundQuotient(num, BigInteger.ONE.shiftLeft(bex));
        return Math.scalb((double)quo, bex + point);
    }
    BigInteger scl = bipows5[-point];
    int mantlen = dblMantDig;
    int bex = mant.bitLength() - scl.bitLength() - mantlen;
    int tmp = bex + point + 1021 + mantlen;
    if (tmp < 0) {bex -= tmp + 1; mantlen += tmp;}
    BigInteger num = mant.shiftLeft(-bex);
    long quo = roundQuotient(num, scl);
    if (64 - Long.numberOfLeadingZeros(quo) > mantlen)
        {bex++; quo = roundQuotient(num, scl.shiftLeft(1));}
    return Math.scalb((double)quo, bex + point);
}
```

The lines involving `tmp` reduce `mantlen` for denormalized floating-point representation when the number is too small for the floating-point exponent. `bex` and `mantlen` are offset by different amounts because the normalized mantissa has an implied most significant 1 digit, while it is explicit in denormalized mantissas.

When `point`< 0, if the number returned by the call to `roundQuotient` is more than `mantlen` bits long, then call `roundQuotient` with double the denominator `scl`. In either case, the final step is to convert to floating-point and scale it using `Math.scalb`.

Because the quotient which gets used is rounded by a single operation to the correct number of bits, it is the closest to the decimal value possible in binary floating-point representation.

Separating powers of 2 from powers of 5 in the `MantExpToDouble` algorithm enables a 29% reduction in the length of intermediate BigIntegers.

In *Fast Path Decimal to Floating-Point Conversion*[Reg11] Regan describes using floating-point multiplication and division when the mantissa and power-of-ten scale fit within floating-point mantissas. This second version of `MantExpToDouble` uses floating-point for small magnitude expoenents, separating the powers of 5 and powers of 2 and postponing the binary scaling until after the multiplication or division by power of 5, which extends the range for which Regan's method applies.

```
 public static double MantExpToDouble(long lmant, int point) {
    long quo; int bex;
    if (point >= 0) {
        if (point < dpows5.length &&
            64 - Long.numberOfLeadingZeros(lmant) <= dblMantDig)
            return Math.scalb(((double)lmant) * dpows5[point], point);
        BigInteger mant = BigInteger.valueOf(lmant);
        BigInteger num = mant.multiply(bipows5[point]);
        bex = num.bitLength() - dblMantDig;
        quo = roundQuotient(num, BigInteger.ONE.shiftLeft(bex));
        return Math.scalb((double)quo, bex + point);
    }
    if (-point < dpows5.length &&
        64 - Long.numberOfLeadingZeros(lmant) <= dblMantDig)
        return Math.scalb(((double)lmant) / dpows5[-point], point);
    BigInteger mant = BigInteger.valueOf(lmant);
    BigInteger scl = bipows5[-point];
    int mantlen = dblMantDig;
    bex = mant.bitLength() - scl.bitLength() - mantlen;
    int tmp = bex + point + 1021 + mantlen;
    if (tmp < 0) {bex -= tmp + 1; mantlen += tmp;}
    BigInteger num = mant.shiftLeft(-bex);
    quo = roundQuotient(num, scl);
    if (64 - Long.numberOfLeadingZeros(quo) > mantlen)
        {bex++; quo = roundQuotient(num, scl.shiftLeft(1));}
    return Math.scalb((double)quo, bex + point);
}
```

Writing

The goal for writing a floating-point number is to output the shortest decimal mantissa which reads back as the original floating-point input. But there are subtleties to this simple sounding idea.

Consider reading back a power-of-two; the number of bits read back can depend on which way the decimal output was rounded. If it was rounded up, then the binary mantissa will have the correct number of bits. If it was rounded down, then the binary mantissa will be one bit short; if that decimal representation doesn't read back correctly, then the binary-to-decimal algorithm recomputes with more precision.

In some cases, the rounded up number reads back correctly, even though the rounded down number is more accurate (but with one more decimal digit). This is not the best idea when reading a number into a higher precision floating-point format than the format it was written from. The number read may not be the closest to the original value. Burger and Dybvig[BD96] use a different criteria: the decimal number written should be the shortest mantissa correctly rounded decimal number. Both cases are treated below.

The integer quotient of a n-digit positive integer and a smaller m-digit positive integer ($0 < m < n$) will always be between $n - m$ and $1 + n - m$ decimal digits in length. Because rounding can cause a carry to propagate through the quotient, the longest integer returned by the `roundQuotient` of a n-digit positive integer and a smaller m-digit positive integer is $2 + n - m$ digits in length, for example `roundQuotient`$(995, 10) \to 100$.

A starved precision is tried first; if that does not read back correctly, then the written precision is increased by one decimal digit; if that does not read back correctly, then the written precision is increased by a second decimal digit. The only cases where starved precision correctly rounded numbers are longer than necessary are when the trailing digits are zero. Code at the end of the algorithm truncates strings of least-significant 0 digits.

It turns out that this second extra digit is needed only for powers of two and, for IEEE-754 double precision format, only normalized powers of two. Thus the test for this condition simply compares the binary mantissa with the largest power of two possible for the mantissa, 2^{53}. Before trying the second extra digit, the quotient plus 1 is checked whether it reads back correctly:

```
if (MantExpToDouble(++lquo, point) != f) {
```

In order to implement the Burger and Dybvig criteria, replace the two occurrences of that line by:

```
{
```

In the algorithm, the positive integer mantissa `mant` and integer exponent (of 2) `e2` are extracted from floating-point input `f`. Constant `llog2` is the base 10 logarithm of 2. The variable `point` is set to the upper-bound of the decimal approximation of `e2`, and would be the output decimal exponent if the decimal point were to the right of the mantissa least significant digit.

When `e2` is positive, `point` is the upper-bound of the number of decimal digits of `mant` in excess of the floating-point mantissa's precision. `mant` is left shifted by `e2` bits into `num`. The `roundQuotient` of `num` and $5^{\texttt{point}}$ yields the integer decimal mantissa `lquo`. If `mantExpToDouble(lquo,point)` is not equal to the original floating-point value `f`, then the `roundQuotient` is recomputed with the divisor effectively divided by 10, yielding one more digit of precision.

When `e2` is negative, `den` is set to $2^{-\texttt{e2}}$ and `point` is the negation of the lower-bound of the number of decimal digits in `den`. `num` is bound to the product of `mant` and $5^{\texttt{point}}$. The `roundQuotient` of `num` and `den` produces the integer `lquo`. If `mantExpToDouble(lquo,point)` is not equal to the original floating-point value `f`, then the `roundQuotient` is computed again with `num` multiplied by 10, yielding one more digit of precision.

The last part of `doubleToString` constructs the output using Java `StringBuilder`. The mantissa trailing zeros are eliminated by scanning the `sman` string in reverse for non-zero digits and the decimal point is shifted to the most significant digit.

The Java code for `doubleToString` shown below uses powers of 5 instead of 10 for speed. The arguments to `BigInteger.leftShift` are adjusted accordingly to be differences of `e2` and `point`.

```java
public static String doubleToString(double f) {
    if (f != f) return "NaN";
    if (f+f==f)
        return 1/f<0?"-0.0":(f==0.0)?"0.0":((f > 0) ? "Infinity" : "-Infinity");
    boolean mns = f < 0; if (mns) f = -f;
    long lbits = Double.doubleToLongBits(f);
    int ue2 = (int)(lbits >>> 52 & 0x7ff);
    int e2 = ue2 - 1023 - 52 + (ue2==0 ? 1 : 0);
    long lquo, lmant = (lbits & ((1L << 52) - 1)) + (ue2==0 ? 0L : 1L << 52);
    int point = (int)Math.ceil(e2*llog2);
    BigInteger mant = BigInteger.valueOf(lmant);
    if (e2 > 0) {
        BigInteger num = mant.shiftLeft(e2 - point);
        lquo = roundQuotient(num, bipows5[point]);
        if (MantExpToDouble(lquo, point) != f) {
            num = num.shiftLeft(1);
            lquo = roundQuotient(num, bipows5[--point]);
            if (lmant==1L<<52 && MantExpToDouble(lquo, point) != f) {
                if (MantExpToDouble(++lquo, point) != f)
                    lquo = roundQuotient(num.shiftLeft(1), bipows5[--point]);
            }
        }
    } else {
        BigInteger num = mant.multiply(bipows5[-point]);
        BigInteger den = BigInteger.ONE.shiftLeft(point - e2);
        lquo = roundQuotient(num, den);
        if (MantExpToDouble(lquo, point) != f) {
            point--;
            num = num.multiply(BigInteger.TEN);
            lquo = roundQuotient(num, den);
            if (lmant==1L<<52 && MantExpToDouble(lquo, point) != f) {
                if (MantExpToDouble(++lquo, point) != f) {
                    point--;
                    lquo = roundQuotient(num.multiply(BigInteger.TEN), den);
                }
            }
        }
    }
    String sman = ""+lquo; int len = sman.length(), lent = len;
    while (sman.charAt(lent-1)=='0') {lent--;}
    StringBuilder str = new StringBuilder(23);
    if (mns) str.append('-');
    if (lent < 8 && point+len <= 0 && point+len > -3) {
        int zs = point+len; str.append("0."); while (zs++ < 0) str.append("0");
        return str.append(sman, 0, lent).toString();
    }
    if (lent < 8 && lent==point+len)
        return str.append(sman, 0, lent).append(".0").toString();
    str.append(sman, 0, 1).append('.').append(sman, 1, lent);
    if (lent==1) str.append('0');
    return str.append('E').append(point + len - 1).toString();
}
```

Performance

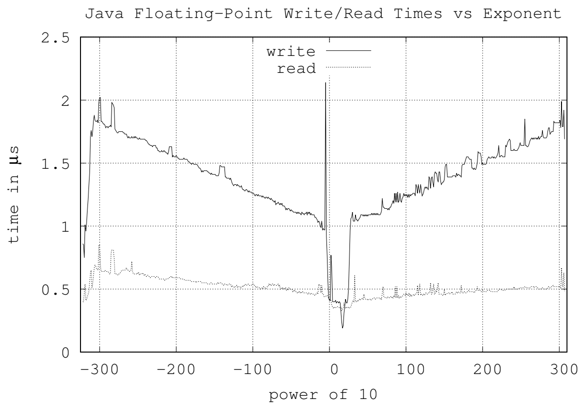

Figure 1

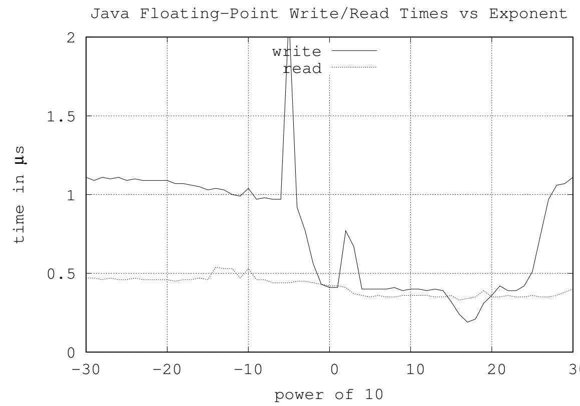

Figure 2

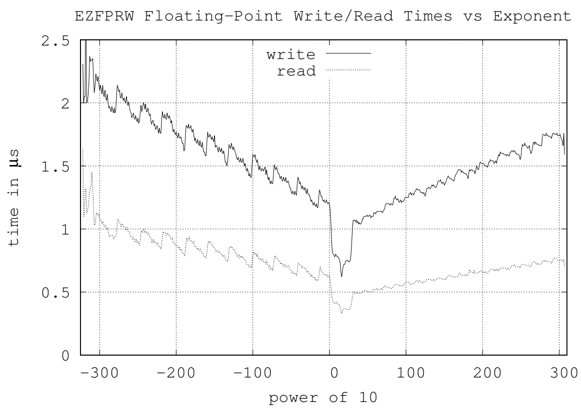

Figure 3

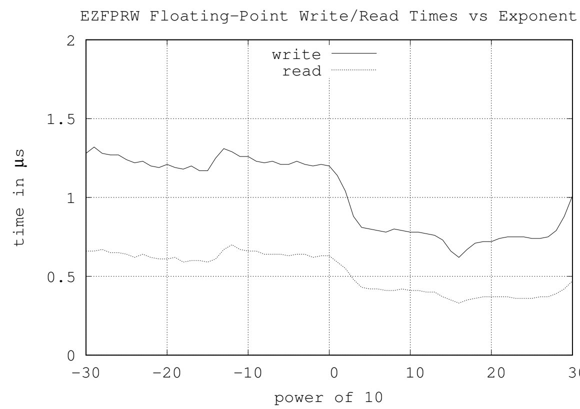

Figure 4

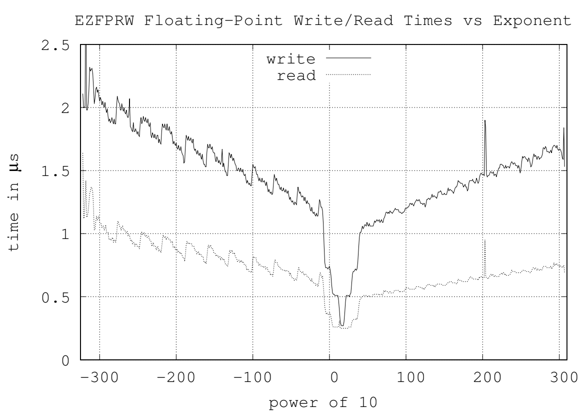

Figure 5

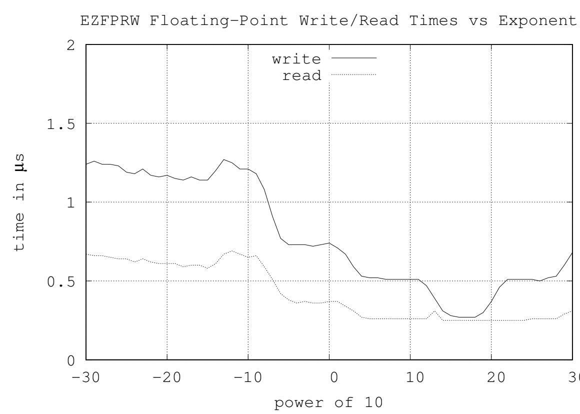

Figure 6

IEEE-754 floating-point numbers have a finite range. And the bulk of floating-point usage tends to have magnitudes within the range 1×10^{-30} to 1×10^{30}. Thus the asymptotic running time of floating-point conversion operations is of limited practical interest. Instead, this article looks at measured running times of Java native conversions and conversions by these new algorithms over the full floating-point range. These measurements were performed on Openjdk version "1.8.0_171" running on a 2.40GHz Intel Core i7-5500U CPU with 16 GB of RAM hosting Ubuntu 16.04 GNU/Linux kernel 4.4.0-127.

A program was written which generated a vector of 100,000 numbers, 10^X where X is a normally distributed random variable. Then for each integer $-322 \leq n \leq 307$, the vector of numbers is scaled by 10^n, written to a file, read back in, and checked against the scaled vector. The CPU time for writing and the time for reading were measured and plotted in Figure 1. An expanded view of Figure 1 for $-30 \leq n \leq 30$ is plotted in Figure 2.

Figures 1 and 2 show the performance of native conversions in Java version 1.8.0_171.

Figures 3 and 4 show the results for the power-of-5 `BigInteger` algorithms implemented in Java.

Figures 5 and 6 show the results for the power-of-5 `BigInteger` algorithms enhanced to use `double`s and `long`s instead of `BigInteger` when the precision allows.

The enhanced power-of-5 read algorithm is faster than Java native conversion in the exponent range $-5 < n < 30$. Both write algorithms are at parity with Java native conversion for positive exponents; the read algorithm is roughly 50% slower. Denormalized conversions ($n < -309$) are less than half of the speed of Java native operations. Over the rest of the negative exponent range the algorithms are roughly 50% slower than Java native conversions.

Because `doubleToString` calls `MantExpToDouble`, improvements to the speed of `MantExpToDouble` will benefit both.

Acknowledgments

Thanks to Tim Peters for finding a class of corner-cases which failed the first version of the `doubleToString`.

Conclusion

The introduction of an integer `roundQuotient` procedure facilitates algorithms for lossless (and minimal) conversions between (decimal) scientific-notation and (binary) IEEE-754 double-precision floating-point numbers which are much simpler than algorithms previously published.

Measurements of conversion times were conducted. Implemented in Java, the optimized conversion algorithms executed faster than Java's native conversions over the range 10^{-5} to 10^{30}, was comparable for writes of numbers greater than 10^{30}, and 50% slower over the rest of the IEEE-754 double-precision range. The `doubleToString` procedure is superior to Java native conversion in that it produces the minimum length mantissa which converts back to the original number.

References

[BD96] Robert G. Burger and R. Kent Dybvig. Printing Floating-point Numbers Quickly and Accurately. *SIGPLAN Not.*, 31(5):108–116, May 1996.

[Cli90] William D. Clinger. How to Read Floating Point Numbers Accurately. *SIGPLAN Not.*, 25(6):92–101, June 1990.

[IEE85] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 1985.

[Mat68] David W. Matula. In-and-out Conversions. *Commun. ACM*, 11(1):47–50, January 1968.

[Mat70] D.W. Matula. A Formalization of Floating-Point Numeric Base Conversion. *Computers, IEEE Transactions on*, C-19(8):681–692, Aug 1970.

[Reg11] Rick Regan. Fast path decimal to floating-point conversion, 2011. [Online; accessed 30-May-2018].

[SW90] Guy L. Steele, Jr. and Jon L. White. How to Print Floating-point Numbers Accurately. *SIGPLAN Not.*, 25(6):112–126, June 1990.