Easy Accurate Reading and Writing of Floating-Point Numbers

Aubrey Jaffer

January 2015

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-1.6, the scientific representations generated use only the minimum number of mantissa digits needed to convert back to the original binary values.

Implemented in Java these algorithms execute as fast or faster than Java’s native conversions over nearly all of the IEEE-754 double-precision range.

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 constraint which Java-1.6 does not support (5E-324 prints as 4.9E-324); 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. This simplicity is responsible for the speed of the algorithms.

Over the range of IEEE-754[IEE85] double-precision numbers, the largest intermediate BigInteger used by the power-of-10 algorithms is 339 decimal digits (1126 bits); for the power-of-5 algorithms, it is 242 decimal digits (803 bits). According to Steele and White[SW90], 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 fast accurate conversions in both directions.

1 Digilant, 100 North Washington Street Suite 502, Boston, MA 02114. Email: agj@alum.mit.edu
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 implemented in Java as follows:

```java
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.

```java
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;
}
```

For its floating-point conversions Java uses a small special-purpose big-integer implementation named FDBigInt. The implementations of the new algorithms presented here use the Java BigInteger package, which comes with the Java distribution.

In the algorithms below, `bipows10` is an array of 326 BigInteger successive integer powers of 10 and `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.

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 $10^{\text{point}}$ and held in variable `num`. The expression `num.doubleValue()` will convert to the correct double-precision floating-point value. But `BigInteger.doubleValue()` as implemented in Java-1.6 is slow for large BigIntegers; so `MantExpToDouble` calls `roundQuotient` if the `num.bitLength()` is greater than the number of bits in the binary mantissa (dblMantDig).

With a negative `point`, the mantissa will be multiplied by a power of 2, then divided by $scl = 10^{-\text{point}}$. The variable `bex` is 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 will always be between $1 + n - m$ and $n - m$ 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 can be $2 + n - m$ bits in length, for example `roundQuotient(3, 2) → 2`. If this happens for some power-of-ten divisor (which is close to a power of 2) then it must happen when the dividend is $2^n - 1$.

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-10 close enough to an integer power-of-2 which, as divisor, causes the quotient to be $2 + n - m$ bits in length. This is also true for power-of-5 divisors.

Thus the longest a rounded-quotient of a $n$ bit integer and a $m$ bit power-of-2 (or 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 must be computed in order to yield a mantissa which is `MantExpToDouble` bits in length.

When `point` is negative, the initial value of `bex` corresponds to this $n - m$ case.

If the number returned by the call to `roundQuotient` is more than `dblMantDig` 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`.

```java
public static double MantExpToDouble(long lmant, int point) {
    BigInteger mant = BigInteger.valueOf(lmant);
    if (point >= 0) {
        BigInteger num = mant.multiply(bipows10[point]);
        int bex = num.bitLength() - dblMantDig;
        if (bex <= 0) return num.doubleValue();
        long quo = roundQuotient(num, BigInteger.ONE.shiftLeft(bex));
        return Math.scalb((double)quo, bex);
    }
    BigInteger scl = bipows10[-point];
    int bex = mant.bitLength() - scl.bitLength() - dblMantDig;
    BigInteger num = mant.shiftLeft(-bex);
    long quo = roundQuotient(num, scl);
    if (64 - Long.numberOfLeadingZeros(quo) > dblMantDig) {
        bex++;
        quo = roundQuotient(num, scl.shiftLeft(1));
    }
    return Math.scalb((double)quo, bex);
}
```
Factoring powers of 2 from the powers of 10 in the MantExpToDouble algorithm enables a 29% reduction in the length of intermediate BigIntegers.

The constant maxpow is used to handle point values less than −325.

```java
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);
    }
    int maxpow = bipows5.length - 1;
    BigInteger scl = (-point <= maxpow) ? bipows5[-point] :
        bipows5[maxpow].multiply(bipows5[-point-maxpow]);
    int bex = mant.bitLength() - scl.bitLength() - dblMantDig;
    BigInteger num = mant.shiftLeft(-bex);
    long quo = roundQuotient(num, scl);
    if (64 - Long.numberOfLeadingZeros(quo) > dblMantDig) {
        bex++;
        quo = roundQuotient(num, scl.shiftLeft(1));
    }
    return Math.scalb((double)quo, bex + point);
}
```

Writing

The algorithm for writing a floating-point number is more complicated in order to output only the shortest decimal mantissa which reads back as the original floating-point input. First, the positive integer mantissa mant and integer exponent (of 2) $e_2$ are extracted from floating-point input $f$. The variable point is set to the upper-bound of the decimal approximation of $e_2$ and would be the output decimal exponent if the decimal point were to the right of the mantissa least significant digit. Constant $\llog2$ is the base 10 logarithm of 2.

When $e_2$ 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 $e_2$ bits into num. The roundQuotient of num and 10$^{\text{point}}$ 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 $e_2$ is negative, den is set to $2^{-e_2}$ 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 10$^{\text{point}}$. 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.
public static String doubleToString(double f) {
    long lbits = Double.doubleToLongBits(f);
    if (f != f) return "NaN";
    if (f+f==f) return (f==0.0) ? "0.0" : (f > 0) ? "Infinity" : "-Infinity";
    int ue2 = (int)(lbits >>> 52 & 0x7ff);
    int e2 = ue2 - 1023 - 52 + (ue2==0 ? 1 : 0);
    int point = (int)Math.ceil(e2*llog2);
    long lquo, lmant = (lbits & ((1L << 52) - 1)) + (ue2==0 ? 0L : 1L << 52);
    BigInteger mant = BigInteger.valueOf(lmant);
    if (e2 > 0) {
        BigInteger num = mant.shiftLeft(e2 - point);
        lquo = roundQuotient(num, bipows5[point]);
        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--;
            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 (f < 0) str.append('-');
    str.append(sman, 0, 1);
    str.append('.');
    str.append(sman, 1, lent);
    str.append('E');
    str.append(point + len - 1);
    return str.toString();
}

Performance

IEEE-754 floating-point numbers have a finite range. And the bulk of floating-point usage tends to have magnitudes within the range $1 \times 10^{-30}$ to $1 \times 10^{30}$. 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 Java version 1.6.0_33 running on a 2.30GHz Intel Core i7-3610QM CPU with 16 GB of RAM hosting Ubuntu GNU/Linux 3.5.0-49.

A Java program was written which generates 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 1.6.0.33.

The power-of-5 algorithms are as fast or faster than Java native conversions except for writes in the exponent range $-5 < n < 25$. When the first (starved precision) conversion is skipped, `doubleToString` is about as fast as Java native conversion in that exponent range as well; but, like Java-1.6 native conversions, some of the scientific-notation mantissas have one digit more than is necessary in order to read back the original floating-point value.

Better performance than Figure 4 in the exponent range $10 < n < 20$ (with minimal precision mantissa) is achieved by doing the calculations in Java `long` integers when they fit; the times are shown in Figures 5 and 6.
There are four regions of the write and read curves of Figures 3 and 5. In the range $0 \leq n \leq 30$, the intermediate integers are small, fitting in a few cache lines. For $n < -300$ the mantissa is unnormalized and requires smaller BigIntegers than for $n = -300$. In the remaining regions the running time grows with the length of the intermediate BigIntegers, although the BigInteger computations take less than half of the overall conversion times.

**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, these conversion algorithms executed as fast or faster than Java’s native conversions over nearly all of the IEEE-754 double-precision range. In addition, the `doubleToString` procedure is superior to Java native conversion in that it always produces the minimum length mantissa which converts back to the original number.

**References**


