The Stata Journal (2008)
8, Number 2, pp. 255–268

Mata Matters: Overflow, underflow and the IEEE floating-point format

Jean Marie Linhart
StataCorp
College Station, TX
jlinhart@stata.com

Abstract. Mata is Stata's matrix language. The Mata Matters column shows how Mata can be used interactively to solve problems and as a programming language to add new features to Stata. In this quarter's column, we investigate underflow and overflow and then delve into the details of how floating-point numbers are stored in the IEEE 754 floating-point standard. We show how to test for overflow and underflow. We demonstrate how to use the %21x format to see underflow and the %16H, %16L, %8H, and %8L formats for displaying the byte content of doubles and floats.

Keywords: pr0038, underflow, overflow, denormalized number, normalized number, subnormal number, double precision, missing values, IEEE 754, format, binary, hexadecimal

1 Introduction

The subject of this quarter's column is overflow and underflow.

Overflow occurs when numbers exceed the maximum value that can be represented in the chosen numeric representation, say, double. In Mata (and Stata), when overflow occurs the result returned is a missing value. `maxdouble()` is a function that returns the maximum value that can be stored. Anything larger than `maxdouble()` is missing.

```
: maxdouble()
  8.9885e+307
: maxdouble()*4
  .
```

Overflow also occurs when negative numbers exceed the maximum negative value that can be represented. `mindouble()` returns the largest negative value that can be stored. Anything smaller than `mindouble()` is missing.

```
: mindouble()
  -1.7977e+308
: mindouble()*4
  .
```

Underflow occurs when small nonzero values (positive or negative) become too small:

```
: 2^-1022
  2.2251e-308

: 2^-1023
  0
```

In this case, the underflow caused a zero to be returned. Underflow is more complicated than overflow because it comes in two forms, gradual and abrupt. The example above is abrupt underflow. The two forms will be discussed in section 2.

Along the way, we are also going to discuss how floating-point numbers are represented on modern digital computers, IEEE standard 754, established by IEEE (1985). We will also discuss Stata's and Mata's missing values and discuss the hexadecimal display formats %21x, %16H, %16L, %8H, and %8L.

This may seem arcane, but underflow recently appeared on Statalist in a question from Joseph Coveney, who was translating CHYPER code (Shea 1989) from Fortran-77 (using abrupt underflow) into Mata (using gradual underflow).

2 Overflow and underflow

Overflow occurs when a number is too big to be represented. An IEEE double-precision computer will assign these numbers to infinity (IEEE 1985), and Stata takes these numbers and assigns them to missing. In Stata or Mata, you can test for overflow simply by checking to see if you have a missing value. For example,

```
: if (missing(x)) { ... }
```

`maxdouble()` returns the maximum positive value that can be stored in Stata. Values greater than `maxdouble()` are assigned to missing. Likewise, `mindouble()` is the largest negative value that can be stored in Stata, and values smaller than `mindouble()` are assigned to missing.

It would be natural to assume that `mindouble()` = −`maxdouble()`, but this is not so.

```
: maxdouble()
  8.9885e+307

: mindouble()
  -1.7977e+308

: -2*maxdouble()
  -1.7977e+308
```

This asymmetry is explained in section 6.

Underflow occurs when a number is too small to be represented with full precision. Underflow can result in a number that is lacking full precision, or it can get mapped to zero. *Gradual* underflow stores numbers with less than full precision. *Abrupt* underflow jumps to zero. The example in section 1 is abrupt underflow. It is hard to "see" gradual underflow since the results are just numbers:

```
: 1/maxdouble()
  1.1125e-308
```

1/`maxdouble()` results in a gradual underflow. One way to see this is by comparing 1/`maxdouble()` with `smallestdouble()`. In Mata (and Stata), `smallestdouble()` returns the smallest full-precision double-precision number. Any number smaller in magnitude than `smallestdouble()` does not have full precision.

```
: x = 1/maxdouble()
: smallestdouble()
  2.2251e-308
: if (abs(x) < smallestdouble()) {
> printf("Underflow!\n")
> };
Underflow!
```

This is how to check for underflow. Above we compared `abs(x)` and `smallestdouble()`. This check for underflow will work if `x` is positive or negative and whether the underflow is abrupt or gradual.

Gradual underflow results in very small numbers with less than full precision. These numbers are called *subnormal* or *denormalized* numbers, in contrast with full-precision numbers, which are called *normalized* numbers. Subnormal or denormalized numbers fill in a gap on a logarithmic scale between zero and the smallest full-precision number. Otherwise, there is a large gap around zero as seen on a logarithmic scale. The precision of subnormal or denormalized numbers depends on the magnitude of the number, whereas all normalized numbers have precision to 1 part in 2^{52}. Each normalized number has 52 binary (base 2) digits; the precision is derived from this. This precision is approximately 16 decimal (base 10) digits. We will discuss normalized and denormalized numbers in more detail in section 6.

The adoption of gradual underflow caused a controversy largely because of fears it would degrade performance of floating-point processors. Severance (1998) explains this controversy in the story of the creation of the IEEE 754 floating-point standard. Degradation of performance turned out not to be a problem, however; Fortran and C both allow a flush to zero mode to force abrupt underflow.

❑ Technical note

In Stata and Mata, `smallestdouble()` returns the smallest full-precision double-precision number. The `smallestdouble()` function returns a value of approximately 2.2251e−308, or, in Stata's %21x format, exactly +1.0000000000000X−3fe (see section 3 for instructions on reading this number). In addition, the value `smallestdouble` is in Stata's `creturn` list. The smallest full-precision double is 2^{-1022}.

This value was recently corrected in Mata and added to Stata; the previous value was +1.fffffffffffffX−3fe or approximately 4.4501e−308 (twice as large).

For those using Stata 9, here is a work-around for calculating an exact value of 2^{-1022}. Calculating 2^{-1022} directly yields a very good approximation, not the exact value. The best way to calculate the smallest full-precision double in Mata is via repeated division in a loop.

```
: smdoub = 1
: for(i=1; i < 1023; i++){
>         smdoub = smdoub/2
> }
: smdoub
  2.2251e-308
```

Once the smallest full-precision double is obtained, you can test for underflow by testing for values between it and zero, for example,

```
: if (abs(x) < smdoub) { ... }
```

❑

❑ Technical note

Whether you use new or old Stata, it is helpful to compare the value of `smdoub`, calculated in the previous technical note by repeated division, and the value obtained from exponentiation. Results are not the same, but they are close.

```
: smdoub_approx = 2^-1022
: (smdoub_approx - smdoub)/smdoub
  2.75335e-14
```

The value from repeated division is better.

Exponentiation is implemented via an internal function called pow() where $2^3 = \text{pow}(2, 3)$. Calculations with small integer exponents are achieved via direct multiplication, but large integer and noninteger exponents are calculated by use of the exponential function exp() and the logarithm function log(), for example, $2^{-1022} = \text{pow}(2, -1022) = \exp\{-1022 * \log(2)\}$. Both log() and exp() are very accurate, but neither is perfectly accurate.

❑

3 The best way to display underflow: the %21x format

Stata's best tool for visualizing double-precision numbers is the %21x or hexadecimal format, which was discussed in Cox (2006). This format displays the full content of a double-precision number, not an approximation like the %f and %g formats do. The %21x format is similar to scientific notation. The %21x format was created for Stata by William Gould and first appeared in Stata release 8 in January 2003.

The digits in %21x format are base 16 called *hexadecimal*, so the digits are 0–9 plus the letters a–f to represent digits 10–15. Below we use a subscript of x to indicate integer and floating-point numbers in hexadecimal format. Here are a few translations of numbers from hexadecimal to base 10.

$$
\begin{aligned}
10_x &= 16 \\
100_x &= 16^2 = 256 \\
1\mathrm{b}_x &= 1 \times 16 + 11 = 27 \\
21.\mathrm{ae}_x &= 2 \times 16^1 + 1 + 10 \times 16^{-1} + 14 \times 16^{-2} = 33.6796875 \\
1.832_x &= 1 + 8 \times 16^{-1} + 3 \times 16^{-2} + 2 \times 16^{-3} = 1.51220703125
\end{aligned}
$$

The %21x format decodes the bytes of a double-precision number to display its *mantissa* and *exponent*. In the example above, the mantissa is a binary or hexadecimal number like 1.832_x; it has single digit before the point, usually a one. The exponent indicates a power of two, not a power of 10 and not a power of 16.

If we display 2^{-1022} in %21x format, we get +1.0000000000000X−3fe, which is $1 \times 2^{-3\mathrm{fe}_x} = 2^{-(3\times256+15\times16+14)} = 2^{-1022}$. A capital X is used in %21x format to indicate that digits are in hexadecimal.

Translating from %21x format to a base-10 number is straightforward. One translates the hexadecimal floating-point number, then multiplies it by the appropriate power of two; for example, −1.4400000000000X+003 is

$$-(1 + 4 \times 16^{-1} + 4 \times 16^{-2}) \times 2^3 = -1.265625 \times 8 = -10.125$$

The %21x format makes it easy to tell which numbers have full precision. All full-precision numbers start with 1 in front of the floating point. Numbers with less than full precision have a 0 in front of the floating point. Full-precision numbers or normalized numbers have 13 hexadecimal digits after the floating point, which is a precision of one part in 16^{13} or, equivalently, 1 part in 2^{52}. Subnormal numbers have less than full precision, start with zero, and their precision depends on the first nonzero digit of the number. Earlier we noted that the result of 1/`maxdouble()` was an underflow; the result of the calculation is a subnormal or denormalized number. Let's take a look:

```
: printf("%21x", 1/maxdouble())
+0.8000000000000X-3ff
```

The leading zero tells us this is a subnormal number. The exponent is $-3\mathrm{ff}_x = -1023$; all numbers with this exponent are either zero or denormalized, as we will see in section 6.

Let's look at the smallest denormalized number. We can see that it is the smallest by the leading zeros.

```
: smallestdouble()/(2^52)
  4.9407e-324
: printf("%21x\n", smallestdouble()/(2^52))
+0.0000000000001X-3ff
```

We can see the difference between a double and a float easily in %21x format as well. Here the constant π is displayed in %21x format, and the constant π rounded to the nearest single-precision floating-point value.

```
: printf("%21x\n", pi())
+1.921fb54442d18X+001
: printf("%21x\n", floatround(pi()))
+1.921fb60000000X+001
```

Single-precision numbers have 0s for the final 7 hexadecimal digits of their mantissa.

Stata and Mata have four other formats for displaying double- and single-precision numbers in the form the computer stores them; %16H and %16L display double-precision numbers and %8H and %8L display single-precision numbers. These four formats display the raw byte content of the number; %21x has the same information but it first translates the bytes into a form similar to scientific notation. The H in %16H and %8H format indicates the bytes that make up the number that will be displayed in *hilo* ordering (highest order or most significant byte to lowest order or least significant byte), and the L in %16L and %8L format indicates the bytes will be displayed in *lohi* ordering (lowest order byte to highest order byte). Each byte is represented by two hexadecimal digits. Unlike %21x format, in these formats there is no simple observational test to tell whether the number is positive or negative, much less to tell whether it is normalized or denormalized. To better understand these formats and the internal representation of numbers, read sections 5 and 6 on binary floating point and IEEE floating-point numbers as well as section 7 on the formats.

❑ Technical note

In the technical notes in section 2, we saw how to compute the smallest full-precision double by repeated division. We named our result `smdoub`. We also calculated 2^{-1022} and called this result `smdoub_approx`. We asserted that repeated division got the better result. Using the %21x format, we can see this:

```
: smdoub
  2.2251e-308
: smdoub_approx
  2.2251e-308
: printf("%21x\n", smdoub)
+1.0000000000000X-3fe
: printf("%21x\n", smdoub_approx)
+1.000000000007cX-3fe
```

Note the noise 7c before the X in the %21x display of `smdoub_approx`.

❑

4 Precision protection

In Mata, numbers are stored and calculations are performed in double precision. On the other hand, data in Stata are usually stored in single precision, but calculations are done in double precision. Single-precision floating-point values have precision to 1 part in 2^{23} or approximately 7 decimal digits, which is more than enough for recording most statistical data. Calculations are done in double precision to provide a buffer against precision problems of all varieties, including overflow and underflow and mundane precision problems that may occur in addition or subtraction. Double precision has precision to 1 part in 2^{52} or approximately 16 decimal digits.

Single-precision numbers are stored in 4 bytes (32 bits). Double-precision numbers are stored in 8 bytes (64 bits).

Computer hardware offers another layer of protection against precision problems. Registers on computer chips are at a higher precision than even double precision, 80 bits versus 64 bits, called extended double precision. Calculation takes place at this higher precision and periodically results are rounded and saved in double precision. Different hardware and software can get different results if the rounding due to saving to double precision occurs at different times in the calculation.

Rarely is single precision insufficient for data storage. One case occurs when a 9-digit U.S. Social Security number is used as an identifier. Nine decimal digits cannot be exactly recorded in single-precision floating point. Social Security numbers must be stored as longs, doubles, or strings.

Denormalized numbers have less than full precision but may have plenty of precision for a given problem. You can calculate the amount of precision from the %21x representation, for example,

```
: x = 2.0e-312
: printf("%21x",x)
+0.0005e403a93f7X-3ff
```

has 9 full digits of hexadecimal precision, with a 5 in the lead digit. Each hexadecimal digit has precision to 1 part in 16. This is precision of 1 part in 5×16^9, about 11 decimal digits.

5 Binary floating point

All numbers on the computer are ultimately stored in binary digits, which can be translated into a binary floating-point number. This means that every number is stored as a set of zeros and ones, and each type of number such as single precision and double precision has a format associated with it for translating those zeros and ones to a binary floating point.

Let's look at some simple examples of binary floating-point numbers. Here the subscript 2 indicates we are looking at binary (base 2) numbers.

$$\begin{aligned}
10_2 &= 2\\
100_2 &= 4\\
11_2 &= 3\\
1.1_2 &= 1 + 1 \times 2^{-1} = 1.5\\
101.101_2 &= 1 \times 2^2 + 1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 5.625
\end{aligned}$$

Each binary digit is a bit of information. We generally speak of bytes of information and each byte contains 8 bits of information. A byte is represented by two hexadecimal digits. Four binary digits give numbers from $0000_2 = 0$ to $1111_2 = 15 = \mathrm{f}_x$ so each set of 4 binary digits determines a hexadecimal digit. To translate 8 binary digits to a byte separate them into two groups of four and translate each group of 4 binary digits to hexadecimal digit shoving them together at the end, see, for example, Gould (2006).

$$11010011_2 \quad \rightarrow \quad 1101_2, \quad 0011_2 \quad \rightarrow \quad \mathrm{d}_x, \quad 3_x \quad \rightarrow \quad \mathrm{d}3_x$$

The magic of translating by position and putting the digits together at the end works when one base is a power of another; hexadecimal is base 16 and $16 = 2^4$.

Raw data stored on a computer is generally displayed in bytes rather than bits.

Before we delve into the details of how a computer stores an 8-byte double, let's look at a manufactured format for a 1-byte number. This made-up format has 1 bit devoted to the sign, 3 bits to the exponent, and 4 bits for the fraction.

The format is

$$\overbrace{0}^{\text{sign}} \quad \underbrace{100}_{\text{exponent}} \quad \overbrace{1001}^{\text{fraction}}$$

Our example number is 01001001_2.

We read from left to right, and we write the most significant bit as the leftmost bit, similar to decimal numbers, where the leftmost digit is the most significant digit.

The sign bit is the first bit, and it is the easiest to understand. A zero indicates a positive number and a one indicates a negative number. Our number is positive.

The power of two is determined by the exponent. Exponents are usually offset or biased so that we can represent either positive or negative exponents. Let's assume the bias is 3. A 3 bit exponent is a number that ranges from 0 to 7 in base 10. We subtract the bias, making the exponent range from -3 to 4. Our number is $100_2 = 1 \times 2^2 = 4$. With the bias removed, this exponent is 1, and we multiply by $2^1 = 2$.

The fraction is 1001_2. There is a leading 1 that is assumed on the fraction to yield a *mantissa*. This mantissa is 1.1001_2 or, leaving out the zeros, $1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-4} = 1.5625$. We multiply by 2 and get 3.125 as our final result.

We can also look at this number in hexadecimal; 01001001_2 is 49_x, but this does not mean $49_x = 73$ like it does for a regular translation from hexadecimal to base 10. We have to use the hexadecimal equivalent of the binary format above to translate and understand that in this format this number represents 3.125. The first hexadecimal digit 4 represents the sign and the exponent. If the digit is greater than 8, we have a negative number and we must subtract 8 to give the biased exponent. This number is not negative, and no modification to the first digit is required; our biased exponent is 4. Remove the bias, and we see we will multiply by 2^1. The final hexadecimal digit represents the fraction, so our mantissa is $1.9_x = 1 + 9 \times 16^{-1} = 1.5625$, and we multiply this by 2 to obtain 3.125 just like we calculated from binary.

6 IEEE Floating point formats and Stata's missing values

The IEEE 754 floating-point standard, established by IEEE (1985), defines how floating-point numbers are stored on modern computers and defines some ground rules for performing calculations. There is a format for double precision, 8-byte floats, familiar as Stata's double type, and single precision, 4-byte floats, known as Stata's float type.

There are also standards for both single- and double-extended precision, which we will not cover here. We discuss the formats for single precision and double precision.

IEEE single-precision numbers consist of 32 bits (4 bytes). The first bit gives the sign, the next 8 bits give the exponent, and the last 23 bits give the fraction. Likewise, IEEE double-precision numbers consist of 64 bits (or 8 bytes), where the first bit gives the sign, the next 11 bits give the exponent, and the remaining 52 bits give the fraction (IEEE 1985), for example,

Single precision:

$$\overbrace{1}^{\text{sign}} \qquad \underbrace{10000000}_{\text{8 bits for the exponent}} \qquad \overbrace{11001000000000000000000}^{\text{23 bits for the fraction}}$$

Double precision:

$$\overbrace{1}^{\text{sign}} \qquad \underbrace{10000000000}_{\text{11 bits for the exponent}} \qquad \overbrace{1100100000000000000000000000000000000000\ldots0}^{\text{52 bits for the fraction}}$$

The exponent is biased in both single and double precision. In single precision, the biased exponent ranges from 0 to 127 with a bias of 63. In double precision, the biased exponent ranges from 0 to 2047 with a bias of 1023. In double precision, the range of unbiased exponents is -1023 to 1024. We will say exponent to indicate the unbiased exponent; i.e., the exponent after the bias has been subtracted off. In the IEEE definition, two exponents, the smallest and largest, are reserved for special cases. Exponents in the middle range (−1022 to 1023 for double) indicate normalized or full-precision numbers. In the normalized numbers, the mantissa is calculated from the fraction by placing a 1 before the floating point.

The smallest exponent, −1023 for double or −63 for single, indicates either a zero or a denormalized number. This was chosen in part for the nice property that zeros have all zeros in the biased exponent and in the fraction, but they can have positive or negative sign. Denormalized or subnormal numbers have the smallest exponent but a nonzero fraction. These numbers have less precision than normalized numbers because the mantissa is assumed to have a leading zero instead of a leading one. The first nonzero digit determines the precision for the number, which is at most 1 part in 2^{51} because there are at most 51 full binary digits.

The largest exponent (64 or 1024 for single and double precision, respectively) holds two different types of results, either what IEEE calls an infinity or what IEEE calls a "not-a-number" (NaN). Infinities have the largest exponent and zeros in the fraction. They can have a positive or a negative sign. Taking the logarithm of zero would result in a value of negative infinity. NaNs also have the highest exponent, but they have a nonzero fraction. A NaN is produced, for example, when one tries to take the logarithm of a negative number.

Stata does not report infinities or NaNs. Instead, Stata maps these values to missing. Stata takes up the largest positive power of two for its missing values. This means that all positive values that are stored in IEEE format with the exponent 1023 are recognized by Stata as missing.

For an in-depth general discussion of computers, how they do calculations, and precision, see Goldberg (1991), Knuth (1998), and (with detailed reference to Mata) Gould (2006),

❑ Technical note

The asymmetry in `maxdouble()` and `mindouble()` is explained by the definition of Stata's missing values. We can see this when we display `maxdouble()` and `mindouble()` and `.` in %21x format.

```
: printf("%21x", maxdouble())
+1.fffffffffffffX+3fe
: printf("%21x", mindouble())
-1.fffffffffffffX+3ff
: printf("%21x", .)
+1.0000000000000X+3ff
```

`mindouble()` has the same mantissa as `maxdouble()`, but the power of two is one larger—$3\text{fe}_x = 1022$ and $3\text{ff}_x = 1023$. Stata allows negative values with the exponent 1023, but missing values are positive with the exponent 1023.

❑

7 %21x, %16H, %16L, %8H, and %8L formats

It is difficult for us humans to keep track of the large number of binary digits in a double-precision number; it is easier to translate to a hexadecimal representation. The %21x format is by far the easiest exact representation of a floating-point number to understand. Understanding and translating the %21x format was covered in section 3. Of the remaining formats, only the %16H format is not too difficult to translate to decimal. The %16H, %16L, %8H, and %8L formats are mostly used for byte comparisons; this most commonly occurs when you are looking at a hexadecimal encoding of a binary file called a hexdump.

We will start by learning %16H format, which displays the 16 hexadecimal digits or 64 bits of information used to store a double-precision number. The 16 in %16H indicates the number of hexadecimal digits that will be displayed. The H in %16H indicates that the number is displayed in hilo format. This means the highest order byte comes first. This is the normal way we understand numbers, e.g., if we write 125 (base 10) the first digit, 1, is in the hundreds place, the highest order or most significant digit, and the last digit, 5, is in the ones place, the lowest order or least significant digit.

Let's translate the binary double-precision example given in the previous section from %16H format to normal decimal format. In binary, this number was

$$11000000000011001000000000000000000000000000000000000\ldots0_2$$

The %16H representation is more manageable, c00c800000000000$_x$ with the format

$$\underbrace{\text{c00}}_{\text{sign and exponent}} \qquad \underbrace{\text{c800000000000}}_{\text{fraction}}$$

The sign bit and 11 bits of the exponent make up the first 3 hexadecimal digits c00$_x$. The remainder give the fraction, c800000000000$_x$.

The sign and part of the exponent are all mixed together in the first hexadecimal digit. We can either translate from hexadecimal to binary (c$_x$ = 1100$_2$) and see that this is negative because the first digit is a one, or note that the number is negative if its first hexadecimal digit is greater than 8. Removing the first digit if it is a one is equivalent to subtracting 8 if the hexadecimal digit is larger than 8. If the first hexadecimal digit is less than 8, no subtraction is needed. c_x is larger than 8; our number is negative. Subtracting 8 from the first digit yields 4 so our biased exponent is $400_x = 4 \times 16^2 = 1024$. After removing the bias, we see the power of two we multiply by is 2^1.

Translating the fraction c800000000000$_x$ we get $12 \times 16^{-1} + 8 \times 16^{-2}$ (omitting the zeros), which results in a mantissa of 1.78125. Our number is $-1.78125 \times 2 = -3.5625$.

It is much easier to see what is going on from the %21x format than it is from the %16H format. In %21x format, -3.5625 is $-1.\text{c}800000000000\text{X} + 001 = -(1 + 12 \times 16^{-1} + 8 \times 16^{-2}) \times 2^1 = -1.78125 \times 2$.

The %16L format displays the same information but in lohi format. The lowest order byte comes first. Here is −3.5625 in both %16H and %16L format:

```
: printf("%16H\n", -3.5625)
c00c800000000000
: printf("%16L\n", -3.5625)
0000000000800cc0
```

One might expect %16L to give the reverse of the hexadecimal digits in the %16H format, but each byte is two hexadecimal digits. It is the pairs that are reversed. Our most significant byte is c0$_x$, which appears at the end, the next is 0c$_x$, 80$_x$, etc. Once this is figured out, we could translate as we did with the %16H example. The sign and exponent make up the three most significant hexadecimal digits (c00$_x$), the remainder is the fraction. Getting the order correct is the biggest new trick here. %16L format does not look very neat visually since it does not break evenly on byte boundaries.

$$\overbrace{\text{000000000080}}^{\text{part of fraction}} \qquad \underbrace{\text{0c}}_{\substack{\text{first digit is part of exponent}\\ \text{last digit is part of fraction}}} \qquad \overbrace{\text{c0}}^{\text{sign and first part of exponent}}$$

For this reason, it is easier to use %16H instead.

The %16L format has the advantage over the %8H and %8L formats in that it does break evenly on all but the first hexadecimal digit.

The %8H and %8L formats both display the 8 hexadecimal digits or 32 bits of information that are used to store a single-precision number. These formats are messy; %8H format has the disadvantage that the first hexadecimal digit contains information from the sign and exponent, and the third hexadecimal digit contains information from both the exponent and the fraction. For this reason, for single-precision numbers, it is often just as easy to go from a binary representation, rather than using the confusing intermediary of %8H (or, even worse, %8L) format.

```
: printf("%8H", -3.5625)
c0640000
: printf("%16H", -3.5625)
c00c800000000000
```

For %8H we have

$$\underbrace{\text{c0}}_{\text{sign and exponent}} \qquad \underbrace{\text{640000}}_{\text{exponent and fraction}}$$

Unless you need to compare bytes, avoid all these complications by displaying in %21x format.

8 Conclusion

Stata does calculations in double precision to protect users from precision problems. Programmers may still need to know how to test for overflow and underflow and may benefit from numerical formats to show the exact contents of numbers, rather than a base-10 approximation.

You may test for overflow with

```
    : if (missing(x)) { ... }
```

and underflow with

```
    : if (abs(x) < smallestdouble()) { ... }
```

Here `smallestdouble()` $= 2^{-1022}$ is the smallest full-precision double-precision.

Underflow can result in either a zero or a number with less than full-precision double-precision. Numbers with less than full-precision double-precision are called subnormal or denormalized numbers.

The %21x format can be used to see whether a number is denormalized. A full-precision number starts with a 1 before the floating point; a denormalized number starts with a 0 before the floating point.

Numbers are stored in the computer in a standard format given by the IEEE 754 floating-point standard (IEEE 1985). Stata has several formats for visualizing the binary contents of numbers. The %21x format is most useful and intuitive. Exact bytes in the order stored may be displayed with the %16H and %16L formats for doubles or the %8H and %8L formats for single-precision floats. The %8H and %8L formats are particularly nonintuitive because the parts of the number do not end on even hexadecimal digit boundaries. %16H, %16L, %8H, and %8L are mostly used for examining a hexadecimal encoding of a binary file called a hexdump; for most applications, %21x is preferred.

9 References

Cox, N. J. 2006. Stata tip 33: Sweet sixteen: Hexadecimal formats and precision problems. *Stata Journal* 6: 282–283.

Goldberg, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys* 23: 5–48. Available at http://docs.sun.com/source/806-3568/ncg_goldberg.html.

Gould, W. 2006. Mata Matters: Precision. *Stata Journal* 6: 550–560.

IEEE. 1985. *IEEE Standard 754-1985 for binary floating-point arithmetic*. New York: Institute of Electrical and Electronics Engineers.

Knuth, D. E. 1998. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. 3rd ed. Reading, MA: Addison–Wesley.

Severance, C. 1998. An interview with the old man of floating-point. *IEEE Computer* 114–115. Available at http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html.

Shea, B. L. 1989. A remark on Algorithm AS152: Cumulative hypergeometric probabilities. *Applied Statistics* 38: 199–204. Code available at http://lib.stat.cmu.edu/apstat/152.

About the author

Jean Marie Linhart is a senior mathematician at StataCorp.