

Handling Numeric Representation SAS® Errors Caused by Simple Floating-Point Arithmetic Computation

Fuad J. Foty, U.S. Census Bureau, Washington, DC

ABSTRACT

Every SAS® programmer knows that an 8-byte maximum length numeric datatype imposes a machine specific limit on the numbers and their corresponding computations. To be exact, SAS® uses a 64-bit floating-point representation to store those numbers. At the Census Bureau, we collect and process survey specific data from start to finish. The data first is sampled, collected, edited, and then weighted using certain algorithms. One of the more complicated algorithms involves the weighting procedure of survey data where replicate weights and replicate factors go through a great deal of numerical computations to produce the output weights that everyone waits for. The values of those variables and factors result in very large positive or negative numbers due to heavy computations and even just simple calculations. Knowing how those numbers are represented and stored on a particular machine, gives the programmer a better understanding of ways to deal with any given algorithms or specifications. Rounding the values of the stored data is one way to control the numbers and the logic to correct certain obvious errors that are due to bit truncation. In this paper, I will go through a comprehensive explanation of how the floating-point numeric number system is represented on our machines and will shed light on ways to handle numeric datatype errors in general.

DISCLAIMER

Any views expressed are those of the author and not necessarily those of the U.S. Census Bureau.

INTRODUCTION

Hardware limitation on computers is the main cause of why we do not represent numeric numbers exactly as we know them. No matter what hardware configuration machine you are using, there will only be a **finite** amount of numbers that this machine can represent and handle. We all know that the real number system that we learned in math classes growing up is an infinite set. This and other infinite numeric sets cannot be represented fully **on any machine** regardless of its size or scope. Today, we have to use the current hardware configuration available to us and the specific finite numbers that we are allowed to be stored on those machines to represent as many of those infinite real-number systems. As a SAS® programmer, with over twenty-five years of experience programming in the SAS® language, I encountered many real work examples that involved heavy computations that produced gigantic output data sets that one typically does not spend **the needed time** to make sure the computations output results are within the certain acceptable threshold. The paper will start by introducing the basic definitions and assumptions about the floating-point number system. It is followed by a detailed definition of what the real problem is, followed by many examples and diagrams that will explain the scope and magnitude of the problem and how to deal with the numeric representation error that is caused by the floating-point representation on our machines.

Note: because of the differences in hardware limitations and operating systems, the examples that I am using in this paper will work on a Linux/Unix Platform. If you are using my examples on a different platform or computer system, your output results may vary.

BASIC DEFINITIONS AND ASSUMPTIONS

- All SAS® numbers, integer or real, are represented as **double-precision (64 bits)** floating-point of the host computer. Floating-point representation is one method that computers use to store numeric values and the SAS® System uses this method for all numeric variables and all computed mathematical operations.
- Precision, significant digits, loss of significance, and Magnitude, are properties used to define a numeric value that is being stored. **Precision** is used to define the accuracy with which a number can be represented on the given computer. **Significant digits** is referred to a measure of the correctness of the numeric value that is being used where an **upper limit** is determined by the precision of the representation. Because of this upper limit, there will be a **Loss of significance**, which reduces the number of significant digits in a numeric value. The **Magnitude** of this numeric value would be a measure of quantity that is expressed as a multiple of the base number system.
- The Floating-point numbers are usually written using **scientific notations** showing the mantissa and the exponent of those number as shown below:

$$1.2345 = \underbrace{12345}_{\text{mantissa}} \times \overbrace{10^{-4}}^{\text{exponent}}$$

- IEEE is the Institute of Electrical and Electronics Engineers who are responsible for setting the standard of the binary floating-point arithmetic. This standard is called the IEEE 754-2008 that specifies the interchange and arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments. Implementing a floating-point system using this standard will be noticed by users in software, hardware or both. The results are determined by the values of the input data, sequence of operations, and the destination formats, that are entirely under the control of the user.
- **Single-precision** floating-point representation uses **32** bits to store numeric values while the **Double precision** uses **64** bits to store the same numeric values. There are 8 bits in 1 byte, so 32 bits represent 4 bytes while 64 bits equal 8 bytes. Below is a diagram, which shows the single and double precision distribution of bits among the components of any given numeric representation. Those components are the **Sign**, **Exponent** and **Mantissa** of any numeric floating-point representation.

IEEE Floating-Point Format

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction
---	----------	----------

$$X = (-1)^S \times (1 + \text{Fraction}) \times 2^{\text{Exponent} - \text{Bias}}$$

Exponent: excess representation: actual exponent + Bias

- Ensures exponent is unsigned
- Single precision: Bias=127
- Double precision: Bias=1023

- SAS uses 64 bits double precision floating-point representation. The **Sign** bit indicates whether the number is positive or negative. Zero (0) indicates the number is positive while one (1) indicates that it is negative. The **exponent** is represented as an **unsigned 11-bit binary value**. This representation is called a **biased** representation because, in order to evaluate it, you must subtract 1023. As a result, normal double-precision values have a range of $\approx \pm 1.7976931348623157 \times 10^{308}$. The values closest to zero are $\pm 2^{-1022} \approx \pm 2.2250738585072020 \times 10^{-308}$.
- The **mantissa** is stored in the last fifty-two bits of the 64-bit floating-point number. The mantissa is sometimes referred to as the fraction or significand. The **first bit** of the mantissa is always set to one. This bit is not stored and is called the **hidden bit**. Each successive bit in the mantissa represents a successively smaller fraction of two.
- The IEEE 754 double-precision binary floating-point format used by SAS on Windows, Linux, and UNIX platforms can be represented by the following formula:

$$N = (-1 * \text{sign}) * (2^{\text{exponent} - 1023} (1 + \frac{b_1}{2^1} + \frac{b_2}{2^2} + \frac{b_3}{2^3} + \frac{b_4}{2^4} + \dots + \frac{b_{52}}{2^{52}}))$$

- The power of 2 to the **Exponent** minus 1023 is probably the only confusing number to calculate and comprehend. Let us look at say the number 1023 in base 10 versus 1023 in base 2. As you can see from the below diagram. There are 11 bits that will hold the exponent part of the numeric floating-point representation number. Each of the 11 bit starting from the right most bit is evaluated 2 to the power of zero (0) and added 2 to the power of one (1) and so forth until you reach the eleventh (11th) bit in the exponent. All of that is done to evaluate the exponent followed by a subtraction of 1023 from it and then taking that number as the power of two (2). This calculation is cumbersome and not intuitive. Below shows 1023 base 10 is stored in the 11 bit exponent field of the 64 bit numeric storage double precision system.

$$(01111111111)_2 = (0 + 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0)_2 = (1023)_{10}$$

DEFINING THE PROBLEM

The problem starts with the numeric precision and representation of numbers on our current computers. We know that SAS uses floating-point representation as defined above. When writing a SAS program that compares any two numbers with each other, we compare them as equal numbers and that is where the problem starts.

Consider the decimal representation values of “0.1” and “0.3”. The following two tables represent the SAS program code and its corresponding output listing. The SAS code below is from Clarke Thacher’s elegant **Why .1+.1 Might Not Equal .2** paper modified slightly to print certain numbers in question. It demonstrates the structure of the floating-point representation. It uses SAS BINARY64 format to produce the bit representation of any SAS numeric value.

```
data numbers;
  keep num sign exponent mantissa;
  length bits $ 64 sign $ 1 mantissa $ 52;
  input num @@;

  bits = put(num,binary64.);
  sign = substr(bits,1,1);
  exponent=input(substr(bits,2,11),binary12.)-1023;
  mantissa = substr(bits,13,52);
  format num best19.;
cards;
0.1
0.3
;
proc print noobs ;
  var num sign exponent mantissa;
run;
```

Now let us take a look at the output of the above program:

num	sign	exponent	mantissa
0.1	0	-4	10011001100110011001100110011001100110011010
0.3	0	-2	00110011001100110011001100110011001100110011

Notice that the mantissa binary numbers are repeating. The machine simply does not represent either number equally as we know them. Therefore, if we add up “0.1” ten times, the actual sum of those ten numbers must not equal to 1. Similarly, if we subtract or add a “0.3” to any other number, the resulting calculation is not going to be equal as we know it in our mathematics class.

In fact, the fundamental mathematical laws known as the associative law, “(x + y) + z = x + (y + z)” , is now NOT TRUE when we use floating-point representation. The reason for that is the **Loss of Significance** in the digits associated with storing the three number (x,y,and z).

USING REAL-LIFE SURVEY REPLICATE WEIGHTING FACTORS EXAMPLE

The survey weighting operation at the U.S. Census Bureau is an extensive process that requires enormous computation applied to various variables and factors. The algorithm is a theoretical approach that uses computers to get as close as possible to the real weight that we are seeking. As we discussed earlier, we are dealing with a finite machine and a finite number of bits to store all of the variable values and the factor values to do the computation that is needed to produce the approximate result.

In the following table, I took ten (10) real random weighting factors and examined their mantissa values as I tried to apply different SAS format options to influence the input reading and storing of those numbers to see how we lose significant digits during the computation. The variable of interest is WGT_F which is a weighting factor that has up to 14 significant digits. I am using the same program that was used above to read and print the BINARY64 values of those numbers.

```
data weights;
  keep WGT_F sign exponent mantissa;
  length bits $ 64 sign $ 1 mantissa $ 52;
  input WGT_F @@;

  bits = put(WGT_F,binary64.);
  sign = substr(bits,1,1);
  exponent=input(substr(bits,2,11),binary12.)-1023;
  mantissa = substr(bits,13,52);
  format WGT_F best19.;
cards;
52.4733398932871
99.3425472645333
86.734833475419
91.314286832256
43.3674167377095
154.232730026116
73.7246084541062
344.053471017549
35.3422365038066
51.1794663184596
;

proc print noobs ;
  var WGT_F sign exponent mantissa;
run;
```

Here are the results of the above program which shows the 52-bit mantissa binary display :

WGT_F	sign	exponent	mantissa
52.4733398932871	0	5	1010001111001001011001100110110100001100011110110101
99.3425472645333	0	6	1000110101011110110001001011010111001010000001001110
86.734833475419	0	6	010110101111000001111000001011111000011101110001111
91.314286832256	0	6	01101101010000001110101000110100001001000011010010111
43.3674167377095	0	5	010110101111000001111000001011111000011101110001111
154.232730026116	0	7	0011010001110111001010000110001111010101111011100101
73.7246084541062	0	6	0010011011100101111111111000010001100110010010100100
344.053471017549	0	8	010110000000110110110000100011011001111101010000010
35.3422365038066	0	5	000110101011110011100110011111011111010110001100010
51.1794663184596	0	5	100110010110111110001100000010011000010000100011110

Notice how the mantissa values differ as you go through the eight and a half bytes that are used to store mantissa of the variable WGT_F. Now, let's modify the above program by inserting the following round statement after the input of WGT_F from cards.

```
"WGT_F = round(WGT_F,0.001);"
```

The above statement will round WGT_F to 3 decimal places and produces the following output:

WGT_F	sign	exponent	mantissa
52.473 Orig	0	5	1010001111001000101101000011100101011000000100000110 1010001111001001011001100110110100001100011110110101
99.343	0	6	1000110101011111001110110110010001011010000111001011
86.735	0	6	0101101011110000101000111101011100001010001111010111
91.314	0	6	0110110101000001100010010011011101001011110001101010
43.367	0	5	0101101011101111100111011011001000101101000011100101
154.233	0	7	0011010001110111010010111100011010100111111011111010
73.725	0	6	00100110111001100110011001100110011001100110011001110
344.053	0	8	0101100000001101100100010110100001110010101100000010
35.342	0	5	0001101010111100011010100111111011111001110110110010
51.179	0	5	10011001011011110100101111000110101001111110111110100

If we increase the significant digits of the same number using the ROUND function. Here is what we get:

WGT_F	sign	exponent	mantissa
52.4733398932871	0	5	1010001111001001011001100110110100001100011110110101
52.473	0	5	1010001111001000101101000011100101011000000100000110
52.4733	0	5	1010001111001001010100011000001010101001100100110001
52.47334	0	5	101000111100100101100110011110110101111000110111111
52.47334	0	5	101000111100100101100110011110110101111000110111111
52.4733399	0	5	10100011110010010110011001101101111110011001000101010

As you can see from the above table, we loose quite a bit of significant bits from the mantissa. You might want to reconsider rounding to less than six (6) significant digits. At least, I do not feel comfortable doing during heavy computations.

ABSOLUTE AND RELATIVE ERROR FUZZING MACROS

By now, you should know that using the EQUAL sign to compare two values is not the right thing to do because those numbers are affected by representation errors. In our case, ALMOST EQUAL is really what we are looking for. We can easily achieve that by using a macro that compares two numbers that would be less than some small values called the FUZZ.

There are two macros to consider. One using absolute error called EQFUZZ_AE and another utilizing relative error EQFUZZ_RE. Below is the code to those two macros:

Absolute Error Fuzz Macro	Relative Error Fuzz Macro
<pre>%macro eqfuzz ae(v1, v2, fuzz=1e-12); abs(&v1 - &v2) < &fuzz %mend;</pre>	<pre>%macro eqfuzz re(v1, v2, fuzz=1e-12); abs((&v1 - &v2)/&v1) < &fuzz %mend;</pre>

Here is the storage representation of fuzz:

fuzz	sign	exponent	mantissa
1E-12	0	-40	0001100101111001100110000001001011011110101000010001

As you can see, fuzz limit is a very small value. So, when we compare the Absolute or Relative error to that fuzz value, we are not equal but almost equal in comparison.

Let us now sum up the values of "0.3" ten (10) times. Theoretically, the sum should add up to three (3). Consider the following SAS code:

```
%macro eqfuzz re(v1, v2, fuzz=1e-12);
  abs((&v1 - &v2)/&v1) < &fuzz
%mend;

data null_;
  Sum=0;
  do i = 1 to 10;
    Sum = Sum + 0.3;
  end;
  Tot=3;
  if Sum eq Tot then put 'Sum EQUAL Tot';
  else if %eqfuzz_re(Sum,Tot) then put 'Sum CLOSE TO Tot';
  else put 'Sum NOT CLOSE TO Tot';
run;
```

The above code uses the relative error FUZZ macro **%eqfuzz_re** to compare **Sum** to **Tot**. Executing the above program, produces a log with the message “**Sum CLOSE TO Tot**” printed as shown in the following log output:

```

216 %macro eqfuzz_re(v1, v2, fuzz=1e-12);
217     abs((&v1 - &v2)/&v1) < &fuzz
218 %mend;
219
220 data _null_;
221     Sum=0;
222     do i = 1 to 10;
223         Sum = Sum + 0.3;
224     end;
225     Tot=3;
226     if Sum eq Tot then put 'Sum EQUAL Tot';
227     else if %eqfuzz_re(Sum,Tot) then put 'Sum CLOSE TO
Tot';
228     else put 'Sum NOT CLOSE TO Tot';
229 run;

Sum CLOSE TO Tot
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

```

You can also use macro **%eqfuzz_ae** to compare an absolute error fuzz value of 1E-12. All you have to do is replace the macro in the above code with the absolute error fuzz macro. Also, you may want to change the fuzz value from 1E-12 to a smaller value depending on the machine you are working with.

WHAT ABOUT INEGERS AND SAS LENGTH STATEMENT

Consider the LENGTH 4 statement which allows 4 bytes for storage of those integers. The maximum number that can be stored in 4 BYTES is **2,097,152** using a PC or a Unix/Linux machine. Below an example code that shows a 4 bytes storage versus 5 bytes. The difference between **2097152** and **2097153** is in byte 5. If we limit it to only 4 bytes then both numbers would equal each other on the machine. To show this, take a look at the following SAS program and it's output log.

```

data ONE;                                data two;
  length A1 A2 4;                          set ONE;
  length B1 B2 5;                          put A1=22.16;
  A1 = 2097152;                             out A1=binarv64.;
  A2 = 2097153;                             put A2=22.16;
  B1 = 2097152;                             put A2=binary64.;
  B2 = 2097153;
run;                                       put B1=22.16;
                                          out B1=binarv64.;
                                          put B2=22.16;
                                          out B2=binary64.;
                                          run;

```



```

%macro create data(Set,n);
data &set bin_nums;
  pow&n = 1;
  do i = 1 to 100;
    pow&n = pow&n * &n;
    val = pow&n; rand = uniform(0);output;
    val = -pow&n; rand = uniform(0);output;
  end;
  file print notitles;
  put ' ';
  Put " ";
  Put "POWERS OF &n";
  Put 'RANGE: ' val e16. ' to ' pow&n e16.;
  put ' ' val = binary64.;
  put pow&n = binary64.;
run;
%mend;

```

Below is the output SAS data set of those positive and negative powers of **n=3**. Notice, **val** is alternating between positive and negative values of the powers of 3. That is intentional because we want to add them up to get zero sum. The variable **pow3** is the absolute value of the variable **val**, and **rand** is just a random number between 0 and 1 that is calculated by the SAS **uniform** function. If we sort the dataset by rand, then it is in **Random order**. If we sort the dataset by val, then we are summing the **smallest val** numbers to the **largest val** numbers. We will sort the bottom data set four different ways and see which way works best.

DATASET: POSITIVE AND NEGATIVE POWERS OF 3

Note: Only the first 10 and last 10 positive and negative powers of 3 are displayed below to show the range of values from smallest to largest.

VIEWTABLE: Work.Power3					VIEWTABLE: Work.Power3				
	pow3	i	val	rand		pow3	i	val	rand
1	3	1	3	0.3450169164	179	8.7279636E42	90	8.7279636E42	0.1154560126
2	3	1	-3	0.7104695438	180	8.7279636E42	90	-8.727964E42	0.88046552
3	9	2	9	0.4592930998	181	2.6183891E43	91	2.6183891E43	0.179576893
4	9	2	-9	0.5864293881	182	2.6183891E43	91	-2.618389E43	0.069538334
5	27	3	27	0.783218529	183	7.8551672E43	92	7.8551672E43	0.935387854
6	27	3	-27	0.2053041571	184	7.8551672E43	92	-7.855167E43	0.0742034451
7	81	4	81	0.7091671618	185	2.3565502E44	93	2.3565502E44	0.2005078486
8	81	4	-81	0.0056616506	186	2.3565502E44	93	-2.35655E44	0.3497752167
9	243	5	243	0.7825928641	187	7.0696505E44	94	7.0696505E44	0.0549874716
10	243	5	-243	0.5716783691	188	7.0696505E44	94	-7.06965E44	0.8288184036
11	729	6	729	0.660374747	189	2.1208951E45	95	2.1208951E45	0.0806922238
12	729	6	-729	0.0688784705	190	2.1208951E45	95	-2.120895E45	0.6379717531
13	2187	7	2187	0.465671763	191	6.3626854E45	96	6.3626854E45	0.1820965829
14	2187	7	-2187	0.7400448968	192	6.3626854E45	96	-6.362685E45	0.2401075886
15	6561	8	6561	0.7668611606	193	1.9088056E46	97	1.9088056E46	0.1851550216
16	6561	8	-6561	0.5369390196	194	1.9088056E46	97	-1.908806E46	0.5956827712
17	19683	9	19683	0.8084329519	195	5.7264169E46	98	5.7264169E46	0.454849608
18	19683	9	-19683	0.2374616779	196	5.7264169E46	98	-5.726417E46	0.4473893943
19	59049	10	59049	0.615297505	197	1.7179251E47	99	1.7179251E47	0.0095542981
20	59049	10	-59049	0.0089173596	198	1.7179251E47	99	-1.717925E47	0.3338758016
					199	5.1537752E47	100	5.1537752E47	0.2710261435
					200	5.1537752E47	100	-5.153775E47	0.7652251347

To do that, we need the following code:

```
%macro sum_set(Set,val,key,msg);
proc sort data=&set;by &key;run;
data null;
  set &set end=done;
  format sum best16.;
  retain sum;
  if ( n = 1) then sum =0;
  sum = sum + &val;

  file print notitles;

  if (done) then do;
    put "SORTED by &key" @30 "SUM=" SUM e16. " <--- &msg" ;
    put sum = binary64.;
  end;
run;
%mend;

%macro main;
%do i = 3 %to 3;
  %let pow=power&i;
  %create_data (&pow,&i);
  /* Random order */
  %sum_set(&pow,val,rand,Random order);
  /* order by val */
  %sum_set(&pow,val,val,VAL Order);
  /* order by reverse val */
  %sum_set(&pow,val,descending val,Reverse VAL Order);
  /* order by absolute value */
  %sum_set(&pow,val,POW&i,Absolute value);
%end;
%mend;

%main;
```

Here is the output listing for the above %create_data macro code using powers of 3 is below:

POWERS OF 3

RANGE: -5.153775207E+47 to 5.153775207E+47

val=110010011101011010010001100101001111001010011001110011011011010
pow3=010010011101011010010001100101001111001010011001110011011011010

Notice that the difference goes from negative to positive. A print of BINARY64 function of the range reveals the **sign bit** which indicates whether it is positive or negative. We are talking about very small numbers here.

Now, take a look at the listing of the macro `%sum_set` using the different **Sort Orders**:

```
SORTED by rand          SUM= 5.316703994E+31 <--- Random order
sum=0100011010000100111110000111111111000101010011001100111101110101
```

```
SORTED by val          SUM=-8.112963841E+31 <--- VAL order
sum=1100011010010000000000000000000000000000000000000000000000000000
```

```
SORTED by descending val SUM= 8.112963841E+31 <--- Reverse VAL Order
sum=0100011010010000000000000000000000000000000000000000000000000000
```

```
SORTED by POW3          SUM= 0.000000000E+00 <--- Absolute Value
sum=0000000000000000000000000000000000000000000000000000000000000000
```

The above output listing shows that if you take the absolute value of all of the numbers, you will get closer to the true value. The sum must equal **zero** because we are taking the first 100 positive and negative powers of three and adding them together. Theoretically, we should get zero. But look at what we got with the different sort orders.

Precision was compromised by the addition process in all of the above sort orders except **POW3**. In many additions, the difference in the exponents was greater the number of bits in our mantissa, and significant bits were shifted away. We got the expected answer only when the observations were ordered in a way that every positive observation was followed by a negative observation, resulting in an intermediate value of 0. Strange as it might seem, all of the results that we got should be considered within the range of acceptable answers. Because we had 200 observations in the range of $\pm 3^{100}$, any variation whose absolute value is less than $200 * 3^{100-52} (5.6E16)$ should not be considered significant.

CONCLUSION

As SAS programmers, it is important to know and understand the difference between the numbers that you see on your display monitor as opposed to those numbers that are stored on your machine. The stored values are the main reason why we have the floating-point representation errors. In addition to those numeric representation error, we also have SAS rounding errors and other errors that are not mentioned in this paper that are caused by computations and other processes and procedures. This is something that needs an open eye on. Understanding the problem and knowing your data, helps one keep the issue of numeric representation errors under control. The error can occur at any step of your computation process or storage. Using certain techniques that you learned in this paper can give the programmer the vision of what to do in order to contain those errors so that they are manageable and fall within certain acceptable boundaries.

REFERENCES

- SAS TS-230 “Dealing with Numeric Representation Error in SAS Applications” Available at: <http://support.sas.com/techsup/technote/ts230.html>
- SAS TS-654 “Numeric Precision 101” Available at: <http://support.sas.com/techsup/technote/ts654.pdf>
- Quality Control With SAS Numeric Data, Paul Gorrell, Social & Scientific Systems, Inc., Silver Spring, MD, Paper AT001, <http://www.albany.edu/~msz03/epi697/papers/at001.pdf>, NESUG 16.
- *Why .1 + .1 Might Not Equal .2 and Other Pitfalls of Floating-Point Arithmetic*, Clarke Thacher, SAS Institute Inc., Cary, NC, SAS Global Forum 2011, *Programming: Foundations and Fundamentals*, Paper 275-2011, <http://support.sas.com/resources/papers/proceedings11/275-2011.pdf>, 4/2011.
- Rounding in SAS®: Preventing Numeric Representation Problems, Imelda C. Go, South Carolina Department of Education, Columbia, SC, Paper PO-082, <http://analytics.ncsu.edu/sesug/2008/PO-082.pdf>, SESUG 2008.
- 754-2008 – IEEE Standard for Floating-Point Arithmetic, <http://ieeexplore.ieee.org/document/4610935/>, ISBN:978-0-7381-5752-8, 8/2008.

ACKNOWLEDGMENTS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.
Other brand and product names are trademarks of their respective companies.

CONTACT INFORMATION

Your comments and questions are welcomed and encouraged. Contact the author at:

Fuad J Foty
U.S. Census Bureau
4600 Silver Hill Road
Washington, DC 20233
301-763-5476
fuad.j.foty@census.gov
