Paper 021-2008

Ludicrously Large Numbers - Using Arbitrary Precision Arithmetic in SAS® Applications

Patrick Tan, EOM Data Solutions, The Netherlands
Koen Vyverman, SAS Institute, The Netherlands

ABSTRACT

The SAS® language knows but a single numeric data type, with a maximum length of 8 bytes. This imposes a limit on the maximum precision that a number can have in SAS. Sometimes, more precision is needed to correctly store and handle very large or very tiny numbers, in which case you typically need to resort to seriously arcane coding. In this paper we propose to take some of the tedium out of this, by using the Java arbitrary-precision libraries and calling these via the Data Step Component Interface for Java Objects.

INTRODUCTION

We first discovered that the 8-byte limit for SAS number storage caused problems for data precision when a SAS customer decided to move some of their Oracle data into SAS data sets. Comparing financial reports run against the original data in Oracle with the same reports derived from the SAS data sets, showed differences for numbers comprising more than 16 significant digits. This was caused by the fact that the Oracle data were stored as 16-byte numbers (approximately), which allows for about 38 significant digits. This is a lot more than the 15-16 significant digits which is the best you get from using 8-byte SAS numbers. More information about numerical precision and SAS can be found in the SAS Technical Support Notes 230 and 654.

For the customer this discrepancy seriously affected the reliability of the SAS-based reports, and we set out to find a solution. The solution to this problem is to use arbitrary-precision arithmetic. According to Wikipedia (as of January 2008): “On a computer, arbitrary-precision arithmetic, also called bignum arithmetic, is a technique that allows computer programs to perform calculations on integers or rational numbers (including floating-point numbers) with an arbitrary number of digits of precision, typically limited only by the available memory of the host system.”

This is easier said than done. We ended up tackling the problem by means of Java, but in the present paper we first discuss three other options which we considered briefly: writing SAS code to program arbitrary-precision algorithms as documented in numerical analysis literature; letting the database system do the number crunching; letting Perl do the number crunching. Following this preliminary exploration of alternatives, we describe the usage of the Java BigDecimal Object in the DATA step to achieve arbitrary-precision in SAS. Finally, we offer conclusions and suggestions for future work in this area.

OPTION 1: PROGRAMMING ARBITRARY-PRECISION ALGORITHMS IN SAS

For the basic arithmetical operations — addition, subtraction, multiplication, division — various arbitrary precision algorithms have been published in the literature. The area of multiplication in particular has attracted a lot of interest, and mathematicians keep coming up with algorithms of ever lower complexity, and hence better performance. Knuth (1997) reserves a significant part of the second volume in his Art of Computer Programming to such algorithms, and discusses e.g. the distribution of floating point numbers, the classical arbitrary precision algorithms (the so-called schoolbook algorithms), and the various advanced multiplication algorithms.

Any given algorithm has a certain complexity, which is essentially a measure for how long it takes to make a computation according to the algorithm in question. A Schoolbook Addition e.g. has a complexity which increases in a linear fashion with the number of digits n that the added terms consist of. This is usually expressed with the Big O notation: Schoolbook Addition has O(n) complexity. Another algorithm with order O(n) complexity is e.g. a sequential look-up in an unsorted table.

The Schoolbook Multiplication algorithm has $O(n^2)$ complexity, meaning that the amount of time needed to multiply two numbers of n digits scales quadratically with n. This is bad news because for large (i.e. asymptotic) values of n the computation time basically explodes. Another algorithm of $O(n^2)$ complexity is e.g. the discrete Fourier transform as often used in factoring large integers for primality testing.

Much effort has gone into the development of multiplication algorithms with a complexity behaving better than $O(n^2)$. The Toom-Cook multiplication algorithm e.g. has a complexity of only $O(n^{1.465})$, which is better than $O(n^2)$. Better still, the Schönhage-Strassen Multiplication (Schönhage & Strassen, 1971) has a complexity of O(n (log n) (log log n)). And recently, Fürer (2007) devised a multiplication algorithm of an even lower complexity: $O(n \log n\ 2^{\log^* n})$.

So the good news is that decent algorithms for implementing arbitrary precision arithmetic do exist. The bad news is that these algorithms are not readily available in SAS. And the ugly news is that although it is possible to implement a

Schönhage-Strassen or a Fürer algorithm in SAS, it's going to be a very non-trivial exercise. We therefore preferred to investigate other options.

OPTION 2: LET THE DATABASE SYSTEM DO THE NUMBER CRUNCHING

Using Oracle (or in general: the original RDBMS) as a calculator seemed to be a viable option. Whenever an arbitrary-precision operation needs to be performed, we could conceivably upload the involved data to the RDBMS as a table comprising variables for the numbers to be processed. Or alternatively, two tables with ID columns which can then be joined.

By using PROC SQL Passthrough in combination with a SAS/ACCESS module — in our case SAS/ACCESS to Oracle — the mathematical capabilities and precision of the RDBMS become effectively accessible within SAS code.

Consider the following 2-table example: if you want to calculate the total revenue for very large quantities of products having very small prices, then there is a fair chance that your results are rounded by SAS because the number of significant digits is insufficient to perform the calculation exactly.

Suppose a table SALES contains the number of items sold for a range of products:

```
sales_id  product_id       items_sold
--------  ----------  ----------------
       1           1   123,456,789,012
       2           2    12,345,678,901
       3           3   999,888,777,666
```

Another table, PRICES, contains the product prices:

```
product_id     product_price
----------  ----------------
         1   0.0000123456789
         2   123456.00001991
         3   23.456789012345
```

Calculating the revenue by means of a DATA step produces results which do not meet the required precision. A more extreme example of this effect is shown in the final section of this paper. However, by using the PROC SQL Passthrough facility, it is possible to create a REVENUE table in the RDBMS where the numerical precision is sufficient to generate correct results:

```
proc sql;
    connect to my_rdbms as my_database (user=donkey password=secret);
    execute (insert into revenue (revenue_id, revenue)
               select sales.sales_id
               ,      sales.items_sold * prices.product_price
               from   sales
               ,      prices
               where  sales.product_id = prices.product_id
        ) by my_database;
    disconnect my_database;
quit;
```

We did some testing and this worked like a charm. Results were correct, and to boot the performance was similar to what the customer was used to. The customer however did not like the idea of creating two tables in the database every single time some arithmetic needed to be done. So we abandoned the idea and turned to yet another option.

OPTION 3: LET PERL DO THE NUMBER CRUNCHING

There are a number of software packages around which already offer arbitrary-precision arithmetic, so the next course of investigation was to look for one such package that easily interfaces with SAS. We first considered Perl, a dynamic programming language created by Larry Wall and first released in 1987.

Perl comes with a large number of modules for almost any purpose imaginable. Unsurprisingly, among those there are many modules implementing arbitrary-precision arithmetic: Math::BigFloat, Math::BigInt, Math::BigRat, etc. And no, that's 'rat' as in 'rational numbers'.

This approach looked promising to us at first. A disadvantage which we hit upon fairly quickly however, is the poor integration of SAS and Perl. The only way to access the Math::BigFloat module from SAS seems to be by writing a txt-file, processing it outside SAS via a system command to call a Perl script, and finally reading the results-file back into SAS. All in all a somewhat laborious and inelegant procedure.

For our customer's reporting needs, going down this route would have represented quite a lot of work. Especially when considering the number of steps in the reporting process where arbitrary-precision arithmetic must be invoked.

Doing this the right way might even involve writing a lot of process control code merely to ensure elementary manageability of a system that relies on executing a Perl-script for each and every addition or multiplication in a sizeable stack of SAS code.

This aspect, combined with the following additional considerations led us to abandon the Perl-route:

1. Getting the Perl software working on a standalone computer is not very hard, but the customer's rigid IT change management processes meant that installing Perl on a production server would take months.
2. Very explicit Perl knowledge is required, and ours is rusty at best.

USING THE JAVA BIGDECIMAL OBJECT IN THE DATA STEP

Another language which already has arbitrary-precision libraries (e.g. the BigDecimal class) is Java. And Java Objects can be called from within the DATA step by means of the Data Step Component Interface (DSCI) for Java Objects. The DSCI has been available as an experimental feature in the DATA step since SAS 9.0. SAS Note 30894 states that it will be production as of SAS 9.2.

In this section we first show how to prepare your SAS environment to use the DSCI for Java. Then we introduce a new Java class BigDecimalToSas incorporating the java.math.BigDecimal class. After compiling our BigDecimalToSas class, we show an example of a DATA step calculation with and without the Java arbitrary-precision functionality.

SETTING UP THE SAS ENVIRONMENT FOR JAVA

You might have installed Java already. It is often included in the SAS installation. You can check this by running PROC JAVAINFO:

```
proc javainfo;
run;
```

The SAS log should show something like this:

```
2    proc javainfo;
3    run;

java.version = 1.4.2_09
java.vm.version = 1.4.2_09-b05
java.home = C:\PROGRA~1\SAS\SHARED~1\JRE\1499C1~1.2_0
java.class.path = C:\Program Files\SAS\SAS 9.1\core\sasmisc\sas.launcher.jar
java.ext.dirs = C:\PROGRA~1\SAS\SHARED~1\JRE\1499C1~1.2_0\lib\ext
java.security.policy = C:\Program Files\SAS\SAS 9.1\core\sasmisc\sas.policy
javaplugin.version = <no value>
sas.jre = private
PFS_TEMPLATE = C:\Program Files\SAS\SAS 9.1\core\sasmisc\qrpfstpt.xml
java.vendor = Sun Microsystems Inc.
java.vendor.url = http://java.sun.com/
java.vm.specification.version = 1.0
java.vm.specification.vendor = Sun Microsystems Inc.
java.vm.specification.name = Java Virtual Machine Specification
java.vm.vendor = Sun Microsystems Inc.
java.vm.name = Java HotSpot(TM) Client VM
java.specification.version = 1.4
java.specification.vendor = Sun Microsystems Inc.
java.specification.name = Java Platform API Specification
java.class.version = 48.0
os.name = Windows XP
os.arch = x86
os.version = 5.1
file.separator = \
path.separator = ;
line.separator =
user.name = Patrick
user.home = C:\Documents and Settings\Patrick
user.dir = C:\Documents and Settings\Patrick
NOTE: PROCEDURE JAVAINFO used (Total process time):
      real time           0.35 seconds
      cpu time            0.01 seconds
```

PROC JAVAINFO describes the properties of the Java Runtime Environment (JRE) which is used by SAS and the

DSCI. You can check the Java version, classpath settings, etc. However, since we're going to compile a Java class, having the only the JRE is not enough. We also need a Java Software Development Kit (SDK), preferably the same version as the JRE.

The Java SDK is freely available from http://java.sun.com. Note that the SDK installers also contain the JRE, so if you didn't have the latter, just installing the SDK will suffice. If you are unsure whether you have an SDK already, search for the program "javac.exe". After installing the Java SDK, make sure that the directory path where javac.exe resides is added to the Windows system variable "Path".

CREATING A BIGDECIMALTOSAS JAVA CLASS

The Java library contains the java.math.BigDecimal class which is an implementation of arbitrary-precision arithmetic. The class BigDecimal itself is pretty versatile. It can do the basic mathematical operations like addition, subtraction, multiplication and division, but there is a lot more to it. As a matter of fact, it provides pretty much all the tools you need to build your own Java calculator.

To prove the concept for our customer we only used the add, subtract, multiply and divide methods, as well as all eight rounding mode types for the divide method.

To simplify the use of the Java methods via the DSCI, we created a wrapper class BigDecimalToSas based on the standard BigDecimal class. BigDecimalToSas accepts strings and numerics as parameters. Its methods are described below:

Method name	Description
`BigDecimalToSas(double set_scale)`	Used to create an instance of the class (object).
`getScale()`	Returns the scale of this BigDecimal. The scale is the number of digits to the right of the decimal point.
`add(String s1, String s2)`	Returns the result of the addition of the values s1 and s2. The parameters s1 and s2 should contain the numerical values as a string.
`divide(String s1, String s2, double rm)`	Returns the result of the division of the values s1 and s2 (s1/s2). The parameters s1 and s2 should contain the numerical values as a string. The parameter rm is used to determine the rounding mode. There are eight rounding mode values, 0–7. E.g. rm=2 corresponds to the ROUND_CEILING rounding mode, which rounds towards positive infinity. For a complete overview of all rounding modes, see the Sun Java BigDecimal Class description.
`divide1(String s1, String s2)`	Returns the result of the division of the values s1 and s2 (s1/s2). The parameters s1 and s2 should contain the numerical values as a string. The rounding mode ROUND_DOWN is used.
`divide2(String s1, String s2)`	Returns the result of the division of the values s1 and s2 (s1/s2). The parameters s1 and s2 should contain the numerical values as a string. The rounding mode ROUND_CEILING is used.
`divide3(String s1, String s2)`	Returns the result of the division of the values s1 and s2 (s1/s2). The parameters s1 and s2 should contain the numerical values as a string. The rounding mode ROUND_FLOOR is used.
`divide4(String s1, String s2)`	Returns the result of the division of the values s1 and s2 (s1/s2). The parameters s1 and s2 should contain the numerical values as a string. The rounding mode ROUND_ HALF_UP is used.
`divide5(String s1, String s2)`	Returns the result of the division of the values s1 and s2 (s1/s2). The parameters s1 and s2 should contain the numerical values as a string. The rounding mode ROUND_ HALF_DOWN is used.
`divide6(String s1, String s2)`	Returns the result of the division of the values s1 and s2 (s1/s2). The parameters s1 and s2 should contain the numerical values as a

	string. The rounding mode ROUND_ HALF_EVEN is used.
`divide7(String s1, String s2)`	Returns the result of the division of the values s1 and s2 (s1/s2). The parameters s1 and s2 should contain the numerical values as a string. The rounding mode ROUND_ UNNECESSARY is used.
`subtract(String s1, String s2)`	Returns the result of the subtraction of the values s1 and s2 (s1-s2). The parameters s1 and s2 should contain the numerical values as a string.
`multiply(String s1, String s2)`	Returns the result of the multiplication of the values s1 and s2. The parameters s1 and s2 should contain the numerical values as a string.

The following is the Java source code which we wrote to define the BigDecimalToSas class:

```
import java.math.BigDecimal;

public class BigDecimalToSas
  // This object is used as a layer between SAS and Java object BigDecimal.
  // The purpose of this object is to use the arbitrary-precision capabilities
  // of Java in the SAS DATA step.
  // More info @ http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigDecimal.html
{
  public int scale;
  public BigDecimalToSas(double set_scale) {
  // The scale is used for the divide method.
    scale = (int)set_scale;
  }

  public BigDecimalToSas() {
    scale = 25;
  }

  public int getScale() {
    return scale;
  }

  public String add(String s1, String s2) {
    BigDecimal result_bd, bd1, bd2;
    bd1       = new BigDecimal(s1);
    bd2       = new BigDecimal(s2);
    result_bd = bd1.add(bd2);
    return result_bd.toString();
  }

  public String divide(String s1, String s2, double rm) {
    BigDecimal result_bd, bd1, bd2;
    bd1       = new BigDecimal(s1);
    bd2       = new BigDecimal(s2);
    result_bd = bd1.divide(bd2, scale, (int)rm);
    return result_bd.toString();
  }

  public String divide(String s1, String s2) {
    return divide(s1, s2, 0);
  }

  public String divide1(String s1, String s2) {
    return divide(s1, s2, 1);
  }

  public String divide2(String s1, String s2) {
    return divide(s1, s2, 2);
  }

  public String divide3(String s1, String s2) {
    return divide(s1, s2, 3);
  }

  public String divide4(String s1, String s2) {
    return divide(s1, s2, 4);
  }

  public String divide5(String s1, String s2) {
    return divide(s1, s2, 5);
  }

  public String divide6(String s1, String s2) {
    return divide(s1, s2, 6);
  }

  public String divide7(String s1, String s2) {
    return divide(s1, s2, 7);
  }

  public String subtract(String s1, String s2) {
    BigDecimal result_bd, bd1, bd2;
    bd1       = new BigDecimal(s1);
    bd2       = new BigDecimal(s2);
    result_bd = bd1.subtract(bd2);
    return result_bd.toString();
  }

  public String multiply(String s1, String s2) {
    BigDecimal result_bd, bd1, bd2;
    bd1       = new BigDecimal(s1);
    bd2       = new BigDecimal(s2);
    result_bd = bd1.multiply(bd2);
    return result_bd.toString();
  }
}
```

We included only certain basic operations here, but other operators and functions offered by java.math.BigDecimal may be added to this wrapper class in a rather straightforward manner.

COMPILING THE BIGDECIMALTOSAS CLASS

The next step is to compile the Java code. We save the above Java source code as "c:\ludicrous\BigDecimalToSas.java". Note that the filename should be the same as the name of the class. After opening a command prompt and navigating to the directory c:\ludicrous, executing the command

```
javac BigDecimalToSas.java
```

compiles the source code into a file named "BigDecimalToSas.class". The executable javac.exe is the Java compiler installed with the Java SDK.

What remains to be done is to add the directory path where our brand-new Java class lives to the SAS System's Java application class-path, otherwise SAS won't be able to find it. One way to do this is by editing your SAS configuration file, locating the JREOPTIONS, and add the path to –Dsas.app.class.path, e.g.:

```
-JREOPTIONS=(
   -Dsas.jre.home=C:\PROGRA~1\SAS\SHARED~1\JRE\1499C1~1.2_0
   -Djava.security.policy=!SASROOT\core\sasmisc\sas.policy
   -Dsas.app.class.dirs=!SASROOT\core\sasmisc;C:\PROGRA~1\SAS\SHARED~1\applets\9.1
   -Dsas.jre=private
   -Dsas.ext.config=!SASROOT\core\sasmisc\sas.java.ext.config
   -DPFS_TEMPLATE=!SASROOT\core\sasmisc\qrpfstpt.xml
   -Djava.class.path=!SASROOT\core\sasmisc\sas.launcher.jar
   -Djava.system.class.loader=com.sas.app.AppClassLoader
   -Dsas.app.class.path=d:\woof\classes;c:\ludicr~1
   )
```

USING THE DSCI FOR JAVA OBJECTS

Now we are ready to put this to the test. First however, note that to interface with a Java object in a DATA step we can use only the two data-types available in SAS. Since we started all this work because the standard 8-byte numerical variable was insufficient, we need to use character strings to contain the values which we're going to exchange with Java via the DSCI. For the purpose of this paper, we will create a data set containing some very large numbers stored in character variables. At the customer site the numbers were coming from the RDBMS by means of a database view representing numbers as strings, or via PROC SQL Passthrough, e.g.

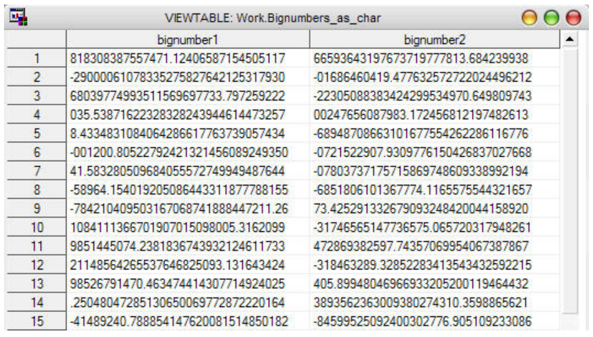
```
proc sql;
    connect to my_relational_database as my_database (user=donkey password=secret);
    create table work.example_nums_in_str as
         select sales_str
         ,      prices_str
         from connection to my_relational_database
            (select tocharacter(sales) as sales_str
             ,      tocharacter (price) as price_str
              from sales);
    disconnect my_database;
quit;
```

The abovementioned conversion function TOCHARACTER() is a placeholder to be replaced by the specific database function to convert numbers into character strings. These functions are different from one RDBMS to another.

To simulate big numbers coming in from some RDBMS the following DATA step generates a data set of random numbers of 32 significant digits stored in 200-byte character variables.

```
data bignumbers_as_char;
    keep bignumber:;
    array bignumbers $200 bignumber1-bignumber2;
    scale = 32;
    number_of_records = 100;
    do j= 1 to number_of_records;
        do over bignumbers;
            set_decimalpoint = 0;
            /* Initialize bignumber and randomly choose a sign. */
            if 2*ranuni(0) gt 1 then bignumbers = "-";
                                else bignumbers = "";
            do i = 1 to scale;
                /* Fill bignumber with random digits and one decimal point. */
                if scale*ranuni(0) gt 2 or set_decimalpoint eq 1 then do;
                    bignumbers = cats(bignumbers, put(int(10*ranuni(0)), 1.));
                end;
                else do;
                    set_decimalpoint = 1;
                    bignumbers = cats(bignumbers,'.', put(int(10*ranuni(0)), 1.));
                end;
            end;
        end;
        output;
    end;
run;
```

The first few records of the data set bignumbers_as_char look like this:

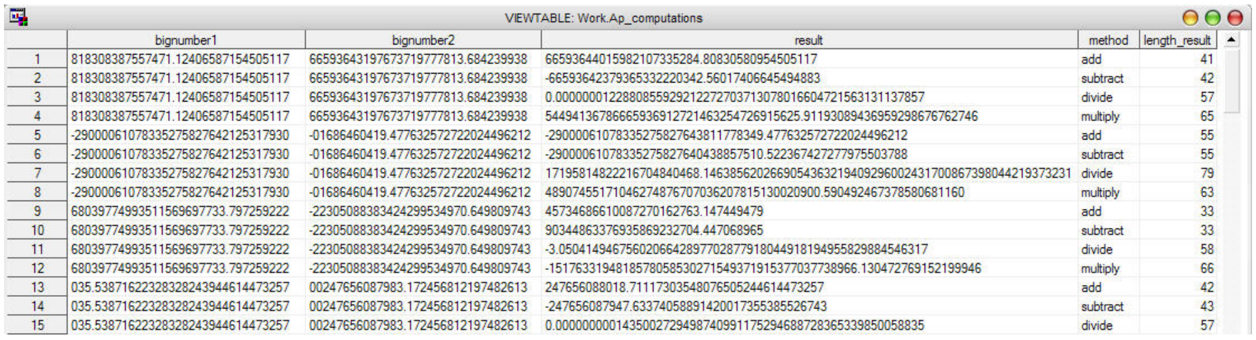

VIEWTABLE: Work.Bignumbers_as_char

	bignumber1	bignumber2
1	818308387557471.12406587154505117	665936431976737197778 13.684239938
2	-29000061078335275827642125317930	-01686460419.4776325727220244962 12
3	680397749935115696977 33.797259222	-22305088383424299534970.649809743
4	035.53871622328328243944614473257	00247656087983.17245681219748261 3
5	8.4334831084064286617763739057434	-68948708663101677554262286116776
6	-001200.80522792421321456089249350	-0721522907.93097761504268370276 68
7	41.583280509684055572749949487644	-07803737175715869748609338992194
8	-58964.15401920508644331187778815 5	-6851806101367774.1165575544321657
9	-7842104095031670687418884472 11.26	73.425291332679093248420044158920
10	10841113667019070150980 05.3162099	-3174656514773657 5.06572031794826 1
11	9851445074.238183674393212461 1733	472869382597.74357069954067387867
12	2114856426553764682509 3.131643424	-318463289.32852283413543432592215
13	98526791470.46347441430771492402 5	405.89948046966933205200119464432
14	.25048047285130650069772872220164	38935623630093802743 10.3598865621
15	-41489240.78885414762008151485018 2	-845995250924003027 76.905109233086

Then we can use the DSCI for Java objects by calling the methods from our BigDecimalToSas class. The data step below performs an arbitrary-precision addition, subtraction, division, and multiplication for each pair of big number strings in the bignumbers_as_char data set:

```
data ap_computations;
    set bignumbers_as_char;
    drop i;
    if _n_ eq 1 then do;
       declare JavaObj bd1 ('BigDecimalToSas', 55);
    end;
    length result $200 method $8;
    array methods(4) $ _temporary_ ("add" "subtract" "divide" "multiply");
    do i = 1 to  dim(methods);
        method = methods(i);
        result='';
        bd1.callStringMethod(strip(method), strip(bignumber1), strip(bignumber2),
                             result);
        length_result = length(strip(result));
        output;
    end;
run;
```

For more information about the DSCI for Java syntax used in the above, we refer to DeVenezia (2005) where all of this is explained in the most illuminating fashion. The first few records of the output data set ap_computations are as follows:

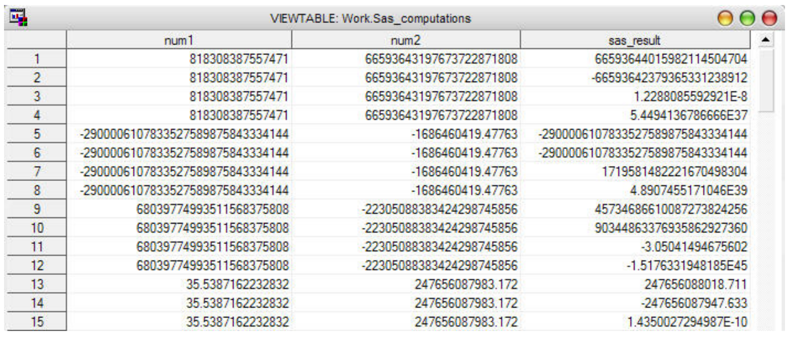

VIEWTABLE: Work.Ap_computations

	bignumber1	bignumber2	result	method	length_result
1	818308387557471.12406587154505117	665936431976737197777813.684239938	665936440159821073352848.08305809545051 17	add	41
2	818308387557471.12406587154505117	665936431976737197777813.684239938	-665936423793653322203425.6017406645494883	subtract	42
3	818308387557471.12406587154505117	665936431976737197777813.684239938	0.0000000122880855929212272703713078016604721563131137857	divide	57
4	818308387557471.12406587154505117	665936431976737197777813.684239938	544941367866659369127214632547269156 25.911930894369592986767627 46	multiply	65
5	-29000061078335275827642125317930	-01686460419.47763257272202449 6212	-2900006107833527582764381177834 9.477632572722024496212	add	55
6	-29000061078335275827642125317930	-01686460419.477632572722024496212	-29000061078335275827640438857510.522367427277975503788	subtract	55
7	-29000061078335275827642125317930	-01686460419.477632572722024496212	17195814822216704840468.1463856202669054363219409296002431700867398044219373231	divide	79
8	-29000061078335275827642125317930	-01686460419.477632572722024496212	48907455171046274876707036207815130020900.590492467378580681160	multiply	63
9	680397749935115696977 33.797259222	-223050883834242995349 70.649809743	457346866100872701627 63.147449479	add	33
10	68039774993511569697733.797259222	-22305088383424299534970.649809743	90344863376935869232704.447068965	subtract	33
11	68039774993511569697733.797259222	-22305088383424299534970.649809743	-3.050414946756020664289770287791804491819495582988 4546317	divide	58
12	68039774993511569697733.797259222	-22305088383424299534970.649809743	-1517633194818578058530271549371915377037738966.13047276915 2199946	multiply	66
13	035.53871622328328243944614473257	0024765608798 3.172456812197482613	24765608801 8.71117303548076505244614473257	add	42
14	035.53871622328328243944614473257	00247656087983.172456812197482613	-247656087947.633740588914200173553855 26743	subtract	43
15	035.53871622328328243944614473257	00247656087983.172456812197482613	0.0000000001435002729498740991175294688728365339850058835	divide	57

Observe the variable length (in terms of number of digits) of the result. As many digits are used as is necessary to perform the operation with full precision, which is exactly what our customer wanted to achieve. If we would use plain old SAS 8-byte arithmetic, the differences are plain to see:

```
data sas_computations(drop = method length_result);
    set ap_computations;
    length num1 num2 sas_result 8;
    format num1 num2 sas_result best32.;
    num1 = input(bignumber1, best32.);
    num2 = input(bignumber2, best32.);
    select (method);
        when ('add')      sas_result = num1+num2;
        when ('subtract') sas_result = num1-num2;
        when ('divide')   sas_result = num1/num2;
        when ('multiply') sas_result = num1*num2;
        otherwise;
    end;
run;
```

Looking at the output data set sas_computations and comparing the values with those in the ap_computations data set, it is apparent that SAS is unable to store the big numbers at full precision and rounds them off so that any subsequent computations on these numbers will also deviate to some degree from the real result:


VIEWTABLE: Work.Sas_computations

	num1	num2	sas_result
1	818308387557471	6659364319767372287 1808	66593644015982114504704
2	818308387557471	66593643197673722871808	-66593642379365331238912
3	818308387557471	66593643197673722871808	1.2288085592921E-8
4	818308387557471	66593643197673722871808	5.4494136786666E37
5	-29000061078335275898758433341 44	-1686460419.47763	-2900006107833527589875843334144
6	-2900006107833527589875843334144	-1686460419.47763	-2900006107833527589875843334144
7	-2900006107833527589875843334144	-1686460419.47763	171958148222167049830 4
8	-2900006107833527589875843334144	-1686460419.47763	4.8907455171046E39
9	680397749935115683758 08	-223050883834242987458 56	457346866100872738242 56
10	68039774993511568375808	-22305088383424298745856	90344863376935862927360
11	68039774993511568375808	-22305088383424298745856	-3.05041494675602
12	68039774993511568375808	-22305088383424298745856	-1.5176331948185E45
13	35.5387162232832	247656087983.172	247656088018.711
14	35.5387162232832	247656087983.172	-247656087947.633
15	35.5387162232832	247656087983.172	1.4350027294987E-10

The arbitrary-precision achieved by using the DSCI for Java comes at a cost of course; there's no such thing as a free lunch… On a 100,000 record test data set we observed an eightfold increase in processing time when using the DSCI method as compared to the less accurate simple SAS method. Yet, it has to be borne in mind that this was merely a simple test-run on a UNIX platform with the big numbers residing in SAS data sets akin to the above example. It would be interesting to measure the performance against a database but we leave that as an exercise to the reader.

CONCLUSION

Using the Data Step Component Interface for Java Objects in conjunction with a custom Java class provides a relatively simple means to achieve arbitrary-precision arithmetic in SAS. The performance cost of this mechanism remains to be rigorously determined though. The main advantage of using this method over the others which we examined is that it is nicely integrated in the SAS DATA step and requires very little additional coding. The method's main limitation is of course that it works only within the confines of the DATA step, and all the PROCs which significantly add to the power of the SAS language are still off-limits for arbitrary-precision computation.

REFERENCES

DeVenezia, Richard A., "Java in SAS®: JavaObj, a DATA Step Component Object", *Proceedings of the Thirtieth Annual SAS Users Group International Conference*, paper 241, 2005.
Available at http://www.devenezia.com/papers/sugi-30/241-30%20(JavaObj).pdf

Fürer, M., "Faster Integer Multiplication", *STOC 2007 Proceedings*, pp. 57-66.
Available at http://www.cse.psu.edu/~furer/Papers/mult.pdf

Knuth, Donald E., "The Art of Computer Programming", Volume 2: Seminumerical Algorithms (3rd Edition), 1997. Addison-Wesley, Chapter 4: Arithmetic, pp. 194-318.

Schönhage, A. and Strassen, V., "Schnelle Multiplikation Großer Zahlen", *Computing 7* (1971), pp. 281-292.

RECOMMENDED READING

Perl BigFloat module: http://perldoc.perl.org/Math/BigFloat.html

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

SAS Usage Note 30894 "Documentation for JavaObj in the DATA Step"
Available at: http://support.sas.com/kb/30/894.html

Sun Java 2 API specification: http://java.sun.com/j2se/1.4.2/docs/api/constant-values.html

Sun Java BigDecimal Class description: http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigDecimal.html

Wikipedia on Arbitrary-precision Arithmetic: http://en.wikipedia.org/wiki/Bignum

Wikipedia on Perl: http://en.wikipedia.org/wiki/Perl

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Patrick Tan
EOM Data Solutions BV
WTC Alnovum
P.J. Oudweg 11
1314 CH Almere
The Netherlands
Tel.: +31 (36) 548 39 50
E-mail: ptan@eom.nl
Web: www.eom.nl

Koen Vyverman
SAS Institute B.V.
Flevolaan 69
1272PC Huizen
The Netherlands
Email: support@snl.sas.com