Why the increasing emphasis on decimal arithmetic?

Most people in the world use decimal (base 10) arithmetic. When large or small values are needed, exponents which are powers of ten are used. However, most computers have only binary (base two) arithmetic, and when exponents are used (in floating-point numbers) they are powers of two.

Binary floating-point numbers can only approximate common decimal numbers. The value **0.1**, for example, would need an infinitely recurring binary fraction. In contrast, a decimal number system can represent 0.1 exactly, as one tenth (that is, 10^{-1}). Consequently, binary floating-point cannot be used for financial calculations, or indeed for any calculations where the results achieved are required to match those which might be calculated by hand. See below for examples.

For these reasons, most commercial data are held in a decimal form. Calculations on decimal data are carried out using decimal arithmetic, almost invariably operating on numbers held as an integer and a scaling power of ten.

Until recently, the IBM z900 (mainframe) computer range was the only widely-used computer architecture with built-in decimal arithmetic instructions. However, those early instructions work with decimal integers only, which then require manually applied scaling. This is error-prone, difficult to use, and hard to maintain, and requires unnecessarily large precisions when both large and small values are used in a calculation.

The same problems existed in the original Java decimal libraries; the problems were so severe that IBM raised a Java Specification Request (JSR), formally endorsed by a wide range of other companies, detailing the problems. This has now been implemented (see the Final Draft for details) and was shipped with Java 1.5 in 2004.

Both problems (the artifacts of binary floating-point, and the limitations of decimal fixed-point) can be solved by using a decimal floating-point arithmetic. This is now available in both hardware and software, as described on the **General Decimal Arithmetic** page, and is standardized in the IEEE Standard for Floating-Point Arithmetic (IEEE 754).

What problems are caused by using binary floating-point?

Binary floating-point cannot exactly represent decimal fractions, so if binary floating-point is used it is not possible to guarantee that results will be the same as those using decimal arithmetic. This makes it extremely difficult to develop and test applications that use exact real-world data, such as commercial and financial values.

Here are some specific examples:

1. Taking the number 9 and repeatedly dividing by ten yields the following results:

Decimal	Binary
0.9	0.9
0.09	0.089999996
0.009	0.0090
0.0009	9.0E-4
0.00009	9.0E-5
0.000009	9.0E-6
9E-7	9.0000003E-7
9E-8	9.0E-8
9E-9	9.0E-9
9E-10	8.9999996E-10

   Here, the left hand column shows the results delivered by decimal floating-point arithmetic (such as the BigDecimal class for Java or the decNumber C package), and the right hand column shows the results obtained by using the Java `float` data type. The results from using the `double` data type are similar to the latter (with more repeated 9s or 0s).

   Some problems like this can be partly hidden by rounding (as in the C printf function), but this confuses users. Errors accumulate unseen and then surface after repeated operations.

   For example, using the Java or C `double` datatype, 0.1 × 8 (a binary multiple) gives the result 0.800000000000000044408920985006261616945266723632812 5 but 0.1 added to itself 8 times gives the different answer 0.799999999999999933386618522490607574582099914550781 25. The two results would not compare equal, and further, if these values are multiplied by ten and rounded to the nearest integer below (the 'floor' function), the result will be 8 in one case and 7 in the other.

Similarly, the Java or C expression `(0.1+0.2==0.3)` evaluates to *false*.

2. Even a *single* operation can give very unexpected results. For example:
   - Consider the calculation of a 5% sales tax on an item (such as a $0.70 telephone call), which is then rounded to the nearest cent.

     Using double binary floating-point, the result of 0.70 x 1.05 is 0.73499999999999998667732370449812151491641998291015625; the result should have been 0.735 (which would be rounded up to $0.74) but instead the rounded result would be $0.73.

     Similarly, the result of 1.30 x 1.05 using binary is 1.365000000000000213162820728030055761337280273437 5; this would be rounded up to $1.37. However, the result should have been 1.365 – which would be rounded *down* to $1.36 (using "banker's rounding").

     Taken over a million transactions of this kind, as in the 'telco' benchmark, these systematic errors add up to an overcharge of more than $20. For a large company, the million calls might be two-minutes-worth; over a whole year the error then exceeds $5 million.

   - Using binary floating-point, calculating the remainder when 1.00 is divided by 0.10 will give a result of exactly 0.09999999999999995003996389186795568093657493591308593 75 (here's a test program showing this). Even if rounded this will still give a result of 0.1, instead of 0, the result obtained if decimal encoding and arithmetic are used.
3. Binary calculations can make apparently predictable decisions unsafe. For example, the Java loop:

```
for (double d = 0.1; d <= 0.5; d += 0.1) System.out.println(d);
```

   displays five numbers, whereas the similar loop:

```
for (double d = 1.1; d <= 1.5; d += 0.1) System.out.println(d);
```

displays only four numbers. (If *d* had a decimal type then five numbers would be displayed in both cases.)

4. Rounding can take place at unexpected places (power-of-two boundaries rather than power-of-ten boundaries). For example, the C program:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  double x = 0.49999999999999994;
  int    i = (int)(x+0.5);
  printf("%0.17f  %d\n", x, i);
  return 0;
  }
```

uses a decades-old idiom for rounding a floating-point number to an integer. Unfortunately in this particular case the value of *x* is rounded up during the addition, so *i* ends up with the value 1 instead of the expected zero. (Thanks to Steve Witham for this example.)

5. In engineering, exact measurements are often kept in a decimal form; processing such values in binary can lead to inaccuracies. This was the cause of the Patriot missile failure in 1991, when a missile failed to track and intercept an incoming Scud missile. The error was caused by multiplying a time (measured in tenths of a second) by 0.1 (approximated in binary floating-point) to calculate seconds. See Douglas Arnold's page for more details.
6. Many programming languages do not specify whether literal constants are represented by binary or decimal floating-point values (even when written using decimal notation). Hence simple constant expressions can give different results depending on the implementation. For example, the C test program:

```
#include
int main(int argc, char *argv[]) {
  if (argc>0) printf("Running: %s\n", argv[0]);

  if (0.1+0.2==0.3) printf("Literals are decimal\n");
               else printf("Literals are binary\n");
  return 0;
  }
```

will display a different result depending on the implementation (usually

'Literals are binary').

7. Finally, there are legal and other requirements (for example, in Euro regulations) which dictate the working precision (in decimal digits) and rounding method (to decimal digits) to be used for calculations. These requirements can only be met by working in base 10, using an arithmetic which preserves precision.

Do applications actually use decimal data?

Yes. Data collected for a survey of commercial databases (the survey reported in IBM Technical Report TR 03.413 by A. Tsang & M. Olschanowsky) analyzed the column datatypes of databases owned by 51 major organizations. These databases covered a wide range of applications, including Airline systems, Banking, Financial Analysis, Insurance, Inventory control, Management reporting, Marketing services, Order entry, Order processing, Pharmaceutical applications, and Retail sales.

In all, 1,091,916 columns were analysed. Of these columns, 41.8% contained identifiably numeric data (53.7% contained 'char' data, with an average length of 8.58 characters, some of which will have contained numeric data in character form).

Of the numeric columns, the breakdown by datatype was:

Type	Columns	percent
Decimal	251,038	55.0
SmallInt	120,464	26.4
Integer	78,842	17.3
Float	6,180	1.4

These figures indicate that almost all (98.6%) of the numbers in commercial databases have a decimal or integer representation, and the majority are decimal (scaled by a power of ten). The integer types are often held as decimal numbers, and in this case almost all numeric data are decimal.

Why are trailing fractional zeros important?

Trailing fractional zeros (for example, in the number 2.40) are ignored in binary floating-point arithmetic, because the common (IEEE 754) representation of such

numbers cannot distinguish numbers of equal value.

However, decimal numbers can preserve the zero in the example because they are almost always represented by an integer which is multiplied by an exponent (sometimes called a scale). The number 2.40 is represented as 240 times 10^{-2} (`240E-2`), which does encode the final zero.

Similarly, the decimal addition 1.23 + 1.27 (for example) typically gives the result 2.50, as most people would expect.

In some programing languages and decimal libraries, however, trailing fractional zeros are either removed or hidden. While sometimes harmless, serious and material problems are caused by this:

- End users expect trailing zeros to be preserved in addition, subtraction, and multiplication (as in the 1.23 + 1.27 example). This is especially true for financial and commercial applications. If the zeros are not preserved, users are surprised, frustrated, and lose confidence in the application.
- Currency calculations are often defined in terms of a given precision (for instance, regulations dictate that Euro exchange rates must be quoted to 6 digits). All the digits must be present, even if some trailing fractional digits are zero. For example, 1 Euro = 340.750 Greek drachmas.
- The original unit of a measurement is often indicated by means of the number of digits recorded. If trailing fractional zeros are removed, measurements and specifications may appear to be more vague (less precise) than intended, and information (the datatype of the number) is lost. For example, the length of a steel beam might be specified as 1.200 meters; if this value is altered to 1.2 meters then a contractor might be entitled to provide a beam that is within 5 centimeters of that length, rather than measured to the nearest millimeter.

  As another example, consider driving directions. “Turn left after 7 miles” and “Turn left after 7.0 miles” lead to different driver behavior.

- Similarly, geographical survey and mapping records indicate the precision of measurements using fractional trailing zeros as necessary. Loop closure software makes use of this information for distributing errors. If fractional zeros are lost then precisely measured segments will appear imprecise; they

will be over-adjusted and the final result will be corrupt.

- Most numeric data in databases have a decimal data type (see above), which is characterized by a given scale (number of fractional digits). This scale needs to be preserved when data are moved in or out of the database so that 'round trips' are possible without loss of information.
- In medical practice, trailing zeros are used where required to demonstrate the level of precision of a value being reported, such as for laboratory results, imaging studies that report size of lesions, or catheter/tube sizes (however, a single trailing zero immediately following a decimal point must not be used in medication orders or other medication-related documentation).
- When decimal numbers are being used simply as labels, and are not expected to be subject to arithmetic, programmers often (perhaps incorrectly) use a decimal datatype to store them. Here, trailing fractional zeros are often significant – section 3.20 of a book, for example, is distinct from section 3.2.
- Requiring that fractional trailing zeros be removed can cause significant extra processing, especially if the coefficient of a number is represented by a binary integer. In this case an division by some power of ten, or the equivalent, is required to remove the trailing zeros. Similarly, this may cause numbers to become unaligned (for example, prices in dollars and cents) which slows addition or subtraction considerably.

For these reasons, decimal libraries must not assume that only the numerical value of a number is significant; they must also take care to preserve the full precision of the number where appropriate.

See also "Why are the encodings unnormalized?" and "Why is the arithmetic unnormalized?".

How much precision and range is needed for decimal arithmetic?

This depends, of course, on the application. Let's consider financial applications. For these applications, values similar to the Gross National Product (GNP) of large countries give an idea of the largest values needed to represent a good range of currency values exactly.

Two examples of these are the USA (in cents) and Japan (in Yen). Both of these need values up to about 10^{15} (a recent GDP number for the USA was

$8,848,200,000,000.00, and for Japan ¥554,820,000,000,000). If stored in a fixed-point representation, these would require 15 digits.

When carrying out calculations, more precision and range is needed because currency values are often multiplied by very precise numbers. For example, if a daily interest rate multiplier, *R*, is (say) 1.000171 (0.0171%, or roughly 6.4% per annum) then the exact calculation of the yearly rate in a non-leap year is *R* raised to the power of 365. To calculate this to give an exact (unrounded) result needs 2191 digits (both the Java BigDecimal class and the C decNumber package easily handle this kind of length).

It's useful to be able to do this kind of calculation exactly when checking algorithms, generating testcases, and comparing the theoretical exact result with the result obtained with daily rounding.

In production algorithms, it is rarely, if ever, necessary to calculate to thousands of digits. Typically a precision of 25-30 digits is used, though the ISO COBOL 2002 standard requires 32-digit decimal floating-point for intermediate results.

As a more concrete example of why 20 digits or more are needed, consider a telephone company's billing program. For this, calls are priced and taxed individually (that is, tax is calculated on each call). Taxes (state, federal, local, excise, *etc.*) are often apportioned and are typically calculated to six places (*e.g.,* $0.000347) before rounding. Data are typically stored with 18 digits of precision with 6 digits after the decimal point (but even with this precision, a nickel call immediately has a rounding error).

(See elsewhere for a definition of *precision*.)

What rounding modes are needed for decimal arithmetic?

The three most important are

1. *round-half-even*, where a number is rounded to the nearest digit, and if it is exactly half-way between two values then it is rounded to the nearest even digit. This method ensures that rounding errors cancel out (on average), and is sometimes called "banker's rounding". For example, 12.345 rounded to four digits is 12.34 and 12.355 rounded to four digits is 12.36. This is also called

round to nearest, ties to even.

2. *round-half-up*, where a number is rounded to the nearest digit, and if it is exactly half-way between two values then it is rounded to the digit above. Here, 12.345 rounded to four digits is 12.35 and 12.355 rounded to four digits is 12.36. This is also called *round to nearest, ties away from zero*.
3. *round-down*, where a number is truncated (rounded towards zero).

The first of these is commonly used for mathematical applications and for financial applications (other than tax calculations) in most states of the USA. The second is used for general arithmetic, for financial applications in the UK and Europe, and for tax applications in the USA. The third is used for both tax and other calculations, world-wide.

Decimal arithmetic packages often provide more rounding modes than these. See, for example, the decNumber context settings, which include eight rounding modes.

Which programming languages support decimal arithmetic?

Decimal data are extremely common in commerce (55% of numeric data columns in databases have the decimal datatype, see above), so almost all major programming languages used for commercial applications support decimal arithmetic either directly or through libraries. (A notable and regrettable exception is JavaScript/JScript.)

Almost all major IBM language products support decimal arithmetic, including C (as a native data type, on System z machines), C++ (via libraries), COBOL, Java (through the Sun or IBM BigDecimal class), OS/400 CL, PL/I, PL/X (on AS/400), NetRexx, PSM (in DB2), Rexx, Object Rexx, RPG, and VisualAge Generator. Many other languages also support decimal arithmetic directly, including Microsoft's Visual Basic and C# languages (both of which provide a floating-point decimal arithmetic using the .Net runtime Decimal class). Decimal arithmetic packages are available for most other languages, including Eiffel and Python.

It is certain that in the future more languages will add decimal datatypes; the ISO JTC1/SC22/WG14 C and WG21 C++ committees are preparing Technical Reports which add the decimal types to those languages, and this will make it much simpler for other languages to add similar support. The GCC and IBM XL C compilers have

already implemented the support described in the C technical report.

For a list of implementations of the new decimal types, see:
http://speleotrove.com/decimal/#links

What disadvantages are there in using decimal arithmetic?

Decimal numbers are traditionally held in a binary coded decimal form which uses about 20% more storage than a purely binary representation.

More compact representations can be used (such as the Densely Packed Decimal and Chen-Ho encodings) but these do complicate conversions slightly (in hardware, compressing or uncompressing costs about two gate delays). (Pure binary integers can also be used, as in the Java BigDecimal class, but this makes rounding, scaling, conversions, and many other decimal operations very expensive.) Even with these more efficient representations, decimal arithmetic requires a few extra bits (an extra digit) to achieve the same accuracy as binary arithmetic.

Calculations in decimal can therefore require about 15% more circuitry than pure binary calculations, and will typically be a little slower. However, if conversions would be needed to use a binary representation because the data are in a decimal base then it can be considerably more efficient to do the calculations in decimal.

Some properties that hold for binary do not hold for any other base. For example, (*d* ÷ 2) × 2 gives *d* in binary (unless there is an underflow), but with base 10 it might not if *d* is full precision and *d* ÷ 2 is Inexact.

Currently, binary floating-point is usually implemented by the hardware in a computer, whereas decimal floating-point is implemented in software. This means that decimal computations are slower than binary operations (typically between 100 and 1000 times slower).

Even using the BCD decimal integer support in the IBM System z, the simulation of scaled or floating-point arithmetic has a significant overhead (mostly in keeping track of scale, and also partly because z900 decimal arithmetic uses Store-to-Store, not Register-to-Register, instructions).

The new hardware support in the Power6 processor (and in the expected z6

processor) is much faster than software, but is still somewhat slower than binary floating-point (for details of the hardware implementations, see the General Decimal Arithmetic page). Binary floating-point is therefore widely used for 'numerically intensive' work where performance is the main concern. This is likely to continue to be the case for some time.

For more details, see the papers in the Choice of base section of the Bibliography.