The Ruby library *long-decimal*

Karl Brodowsky
IT Sky Consulting GmbH
Switzerland

2018-12-15

Abstract

The goal of the Ruby-library *long-decimal* is to provide a new numerical type *LongDecimal* and arithmetic operations to deal with this type.

1 Introduction

Ruby supports non-integral numbers with the built-in types *Float* and *Rational* and *BigDecimal*. While these are very useful for many purposes, the development of finance application requires the availability of a type like *LongDecimal*, that allows explicit control of the rounding and uses a decimal representation for the fractional part. An instance of *LongDecimal* can be represented as a pair of integral numbers (n, d), where the value of the number is $\frac{n}{10^d}$. The ring-operations $+$, $-$ and $*$ can be trivially defined in a way that does not loose any information (and thus does not require any rounding).
Division requires some additional thought, because the exact quotient of two *LongDecimal*s can very well be expressed as Rational, but not always as *LongDecimal*. A supplementary numeric type *LongDecimalQuot* has been introduced to support storing pairs (r, d), where r is a rational number and d is an estimation about the significant digits after the decimal point. The represented value is the rational number r.
Calculation of roots and trancendental functions require additional information as input parameters to control the number of digits after the decimal point and the rounding rules for achieving the result.
In the remainder of this document we will write $\mathrm{l}(n, d)$ for the *LongDecimal*-number represented by the pair (n, d) in the manner described above and $\mathrm{q}(r, d)$ for the *LongDecimalQuot*-number represented by the pair (r, n). It needs to be observed that the part d in both cases carries information about the precision of the number in terms of significant digits after the decimal point.

2 Ring Operations

The ring operations $+$, $-$ and $*$ and obvious derived operations like negation are defined like this:

$$\bigwedge_{e,d\in\mathbb{N}_0}\bigwedge_{m,n\in\mathbb{Z}} \mathrm{l}(m,d)+\mathrm{l}(n,e)=\mathrm{l}(m\cdot 10^{\max(d,e)-d}+n\cdot 10^{\max(d,e)-e},\max(d,e))$$

$$\bigwedge_{e,d\in\mathbb{N}_0}\bigwedge_{m,n\in\mathbb{Z}} \mathrm{l}(m,d)-\mathrm{l}(n,e)=\mathrm{l}(m\cdot 10^{\max(d,e)-d}-n\cdot 10^{\max(d,e)-e},\max(d,e))$$

$$\bigwedge_{e,d\in\mathbb{N}_0}\bigwedge_{m,n\in\mathbb{Z}} \mathrm{l}(m,d)\cdot\mathrm{l}(n,e)=\mathrm{l}(mn,d+e)$$

3 Rounding Operations

The library supports two rounding operations that can be applied both to *LongDecimal* and *LongDecimalQuot*. In the latter case it implies a conversion to *LongDecimal*. The normal rounding function *round_to_scale* changes the precision to the given value, while keeping the value expressed by the number approximately the same.
The optional second parameter describes how rounding will be done, if the process looses information. It defaults to *ROUND_UNNECESSARY*, in which case an exception is raised whenever a value changing rounding process would be required.
The other rounding operations can be grouped into two groups: ROUND_UP, ROUND_DOWN, ROUND_CEILING and ROUND_FLOOR already answer the question of how to round completely because they use the values that result from rounding as boundaries and obviously round these to themselves.
The other rounding operations put a boundary somewhere in the middle between allowed rounded values, but they leave the question of how to round the boundary itself to be answered as well. This is expressed by having a major rounding mode that defines where the boundaries lie and a minor rounding mode that is applicable for values that lie exactly on the boundary. In some cases this cannot happen, because boundaries are irrational and *LongDecimal* (and *LongDecimalQuot*) can only express (some) rational numbers, but for the sake of completeness and applicability to other representations, that might allow some irrational values, this is done in a uniform way. Since rounding operations can also apply to internal intermediate results, it is a good idea not to constrain their definition to rational numbers.
Major rounding modes are

MAJOR_UP	always round in such a way that the absolute value does not decrease.No minor rounding mode needed. In short: round away from zero.
MAJOR_DOWN	always round in such a way that the absolute value does not increase. No minor rounding mode needed. In short: round towards zero.
MAJOR_CEILING	always round in such a way that the rational value does not decrease. No minor rounding mode needed. In short: round towards infinity.
MAJOR_FLOOR	always round in such a way that the rational value does not increase. No minor rounding mode needed. In short: round towards negative infinity.
MAJOR_HALF	round to the nearest available value. The boundary is the arithmetic mean of the two adjacent available values. In short: what we usually think of when talking about rounding.
MAJOR_GEOMETRIC	round to one of the two adjacent available values x, y and use their geometric mean $\sqrt{xy}$ as boundary. For negative x, y use the negated square root instead.
MAJOR_HARMONIC	round to one of the two adjacent available values x, y and use their harmonic mean $\frac{2xy}{x+y}$ as boundary.
MAJOR_QUADRATIC	round to one of the two adjacent available values x, y and use their quadratic mean $\sqrt{\frac{x^2+y^2}{2}}$ as boundary. If $x, y \leq 0$ use the negated square root instead.
MAJOR_CUBIC	round to one of the two adjacent available values x, y and use their cubic mean $\sqrt[3]{\frac{x^3+y^3}{2}}$ as boundary.
MAJOR_UNNECESSARY	raise an exception if value changing rounding would be needed.

Minor rounding modes are

rounding mode	**description**
MINOR_UNUSED	no minor rounding mode applies, can only be combined with ROUND_UP, ROUND_DOWN, ROUND_CEILING and ROUND_FLOOR.
MINOR_UP	round values exactly on the boundary by increasing the absolute value (away from zero).
MINOR_DOWN	round values exactly on the boundary by decreasing the absolute value (towards zero).
MINOR_CEILING	round values exactly on the boundary by increasing the rational value (towards ∞).
MINOR_FLOOR	round values exactly on the boundary by decreasing the rational value (towards $-\infty$).
MINOR_EVEN	round values exactly on the boundary by using the choice resulting in the rounded last digit to be even.
MINOR_ODD	round values exactly on the boundary by using the choice resulting in the rounded last digit to be odd.

These combine to rounding modes:

rounding mode	**description**
ROUND_UP	always round in such a way that the absolute value does not decrease.
ROUND_DOWN	always round in such a way that the absolute value does not increase.
ROUND_CEILING	always round in such a way that the rational value does not decrease.
ROUND_FLOOR	always round in such a way that the rational value does not increase.
ROUND_HALF_UP	round to the nearest available value, prefer increasing the absolute value if the last digit is 5
ROUND_HALF_DOWN	round to the nearest available value, prefer decreasing the absolute value if the last digit is 5
ROUND_HALF_CEILING	round to the nearest available value, prefer increasing the rational value if the last digit is 5
ROUND_HALF_FLOOR	round to the nearest available value, prefer decreasing the rational value if the last digit is 5
ROUND_HALF_EVEN	round to the nearest available value, prefer the resulting last digit to be even if the last digit prior to rounding is 5
ROUND_HALF_ODD	round to the nearest available value, prefer the resulting last digit to be odd if the last digit prior to rounding is 5
ROUND_GEOMETRIC_UP	round to the available value using the geometric mean as boundary, prefer increasing the absolute value if the unrounded value is exactly on the boundary
ROUND_GEOMETRIC_DOWN	round to the available value, using the geometric mean as boundary, prefer decreasing the absolute value if the unrounded value is exactly on the boundary
...	...
ROUND_HARMONIC_UP	round to the available value using the harmonic mean as boundary, prefer increasing the absolute value if the unrounded value is exactly on the boundary
ROUND_HARMONIC_DOWN	round to the available value, using the harmonic mean as boundary, prefer decreasing the absolute value if the unrounded value is exactly on the boundary
...	...
ROUND_QUADRATIC_UP	round to the available value using the quadratic mean as boundary, prefer increasing the absolute value if the unrounded value is exactly on the boundary
ROUND_QUADRATIC_DOWN	round to the available value, using the quadratic mean as boundary, prefer decreasing the absolute value if the unrounded value is exactly on the boundary
...	...
ROUND_CUBIC_UP	round to the available value using the cubic mean as boundary, prefer increasing the absolute value if the unrounded value is exactly on the boundary
ROUND_CUBIC_DOWN	round to the available value, using the cubic mean as boundary, prefer decreasing the absolute value if the unrounded value is exactly on the boundary
...	...
ROUND_CUBIC_ODD	round to the available value using the cubic mean as boundary, prefer the resulting last digit to be odd if the unrounded value is exactly on the boundary
ROUND_UNNECESSARY	raise an exception if value changing rounding would be needed

In addition to these commonly available rounding operations *long-decimal* provides rounding to remainder sets. This is motivated by the practice in some currencies to prefer using certain multiples of the smallest unit. For example in CHF you commonly use two digits after the decimal point, but the last digit is required to be 0 or 5. This is achieved as a special case of a more general concept *round_to_allowed_remainders*. A modulus $M \geq 2$ and a set $R \subset \mathbb{N}_0$ are given and a number x is rounded to $\mathrm{l}(n, d)$ such that

$$\bigvee_{r \in R} n \equiv r \mod M$$

Typically M is ten or a power of ten, but this is not required. Neither is it required that zero is a member of R. In that case an additional parameter is needed in order to define to which direction a potential last digit of zero would be rounded. In the CHF case we would have $M = 10$ and $R = \{0, 5\}$. This kind of rounding can be applied to integers as well as to *LongDecimal* and *LongDecimalQuot*, in which case the integral numerator n of $\mathrm{l}(n, d) = \frac{n}{10^d}$ must fullfill the additional constraint. In order to avoid complications, this is not supported in conjunction with minor rounding modes MINOR_EVEN and MINOR_ODD, because all matching rounded values might be even or odd and we cannot rely on the alternation of even and odd values. It is possible to exclude 0 from the set R. In this case a ZERO-rounding-mode needs to be provided that tells in which way a zero, should it occur, shoud be rounded.

4 Division

Regular division using / yields an instance of *LongDecimalQuot*. The approximate number of significant digits is estimated by using the partial derivatives of $f(x,y) = \frac{x}{y}$. Assume $x = \mathrm{l}(m,s)$ and $y = \mathrm{l}(n,t)$. So we get for $\mathrm{l}(m,s)/\mathrm{l}(n,t)$ an result

$$\frac{x}{y} = \mathrm{q}(10^{t-s}\frac{m}{n}, r)$$

with

$$r \approx -\log_{10}\left(10^{-s}\frac{1}{|y|} + 10^{-t}\frac{|x|}{y^2}\right) = 2\log_{10}(y) + s + t - \log_{10}\left(|m| + |n|\right))$$

In order to avoid expensive logarithmic operation for such basic operations as division this is approximated by

$$r = \max(0, 2v + s + t - \max(u + s, v + t) - 3)$$

with $u = \lfloor \log|m| \rfloor - s + 1$ and $v = \lfloor \log|n| \rfloor - t + 1$ for m and n not zero and $u = -s$ for $m = 0$ and $v = -t$ for $n = 0$. Using $\max(a,b)$ instead of $log_{10}(10^a + 10^b)$ is a good approximation, when a and b are far apart.

It is recommended to use explicit rounding after having performed division rather than to rely on this somewhat arbitrary estimation on the number of significant digits. It is possible to retain intermediate results with *LongDecimalQuot*, because the full arithmetic is available for this type as well, but using rational numbers internally it will blow up numerator and denominator to an extent that diminishes performance by quite a margin in longer calculations.

5 Roots

Square root and cube root of *LongDecimal* can be calculated quite efficiently using an algorithm that is somewhat similar to the algorithm for long integer division. This has been preferred over the more commonly known Newton algorithm. Since square and cube roots are usually irrational, it is mandatory to provide rounding information concerning the number of desired digits and the rounding mode.
Square roots and cube roots can also calculated of integers, in which case a variant is available that also calculates a remainder r in addition to the approximated square root s, such that $a = s^2 + r$.

6 π

The number π can be calculated to a given number of digits, which works sufficiently fast for a few thousend digits.
Please use dedicated programs if you seriously want to calculate π to millions of digits, Ruby is not fast enough for this kind of number crunching to compete with the best C-programs.

7 Transcendental Functions

It is the goal of this library to support the most common transcendental functions in the long run. Currently $\exp$ and $\log$ are supported. These are calculated using the Taylor series, but the calculation has been improved. Most important it is to improve the convergence, which is the case with a series of the form

$$\sum_{n=0}^{\infty} \frac{x^n}{n!}$$

for $x < 0.5$.
For the exponential function this can always be achieved by using the following transformations: $x = 2^n x_0$ with $x_0 < 0.5$ implies

$$\exp(x) = \exp(x_0)^{2^n}$$

which can be easily calculated using successive squaring operations.
For the logarithm we first use $x = e^n x_0$ with $n \in \mathbb{N}_0$ and $x_0 < e$ with

$$\log(x) = n + \log(x_0).$$

From here we make use of the efficient square root calculation facility and use

$$\bigwedge_{m=0}^{\infty} x_{m+1} = \sqrt{x_m}$$

and

$$\log(x) = n + 2^m \log x_m$$

Using this x_m can be brought sufficiently close to 1 to make the Taylor series of $\log$ converge sufficiently fast to be useful for the calculation:

$$\log(x_m) = \sum_{k=1}^{\infty} \frac{(x_m - 1)^k (-1)^{k+1}}{k}$$

Another minor optimization uses some integer j and adds

$$\bigwedge_{l=0}^{j-1} s_k = \sum_{l=0}^{n} \frac{x^{lj}}{(lj+k)!}$$

from which we finally multiply the partial sums with appropriate low powers of x. This saves on the number of multiplications. Similar patterns will be applied to the calculation of other transcendental functions.

8 Powers

Calculation of powers with any positive *LongDecimal* as base and any *LongDecimal* as exponent is quite challenging to do.
It is quite trivial that $x^y = \exp(y \log x)$, but the challenges are to achieve the result in acceptable time and with the required precision. The builtin class *BigDecimal* does not meet these goals, because it becomes incredibly slow for certain combinations of x and y. The power function of *LongDecimal* has been optimized to handle a broad range of cases with different approaches to provide precision and speed. This should work fine for reasonable practical use, but it will still be possible to construct corner cases with bases very close to 1 and large exponents which fail in an attempt to do an accurate calculation in their last digits.

9 Means

In the class *LongMath* there are methods for calculating the following means: arithmetic_mean, geometric_mean, harmonic_mean, quadratic_mean, cubic_mean, arithmetic_geometric_mean, harmonic_geometric_mean. See Wikipedia for their definitions.

10 Rounding with sum constraint

Experimental support for rounding of several numbers simultanously in such a way that their rounded sum is the sum of the rounded numbers is included with the methods *LongMath.round_sum_hm* which uses the Haare-Niemeyer-approach and *LongMath.round_sum_divisor* which uses one of several divisor based approaches, like D'Hondt. The approach is chosen by the rounding mode. These are not yet implemented efficiently nor are they tested well, so use at your own risk (like the whole library).

11 Limitations

Rounding with sum constraint is not yet production stable.
Powers with bases that are off 1 by 10^{-20} or less and exponents in the order of magnitude of 10^20 or more are not always calculated precisely.
Some interesting transcendental functions are missing.
Many operations (like power, exp, log and other transcendental functions) fail to work with numbers whose magnitude cannot be expressed as double.
Using transcendental functions that have a rounding mode and precision as part of their parameter set has not been tested with the newer rounding modes, ..._ODD, ROUND_GEOMETRIC_..., ROUND_HARMONIC_..., ROUND_QUADRATIC_... and ROUND_CUBIC_.... These combinations will be tested and improved in future versions.

12 Tests

Unit tests have been added to test much of the implemented functionality.
More unit tests are desirable.