ldecNumber
– a Lua 5.1 wrapper for the Decimal Number library decNumber

Doug Currie
August 7, 2007

Table of Contents

1 Licenses

The ldecNumber distribution includes the C source code of the wrapper itself as a Lua module, a Makefile, examples, test code, and the decNumber source code from IBM. The decNumber code is licensed as follows (see ICU-license.html in the distribution):

1.1 ICU License - ICU 1.8.1 and later

1.2 ldecNumber License

The non-IBM Lua decNumber code and documentation are:

2 Background

The **ldecNumber** package is a Lua module for General Decimal Arithmetic. For the background and rationale for the design of the arithmetic, see *Decimal Floating-Point: Algorism for Computers* in the Proceedings of the 16th IEEE Symposium on Computer Arithmetic (Cowlishaw, M. F., 2003). The **decNumber** package, an arbitrary-precision implementation of the specifications in ANSI C, provides a *reference implementation* for both the arithmetic and the encodings, and is used as the basis for the **ldecNumber** package. See Mike Cowlishaw's decimal page for more information and the latest versions of the **decNumber** package and test code.

The **ldecNumber** package was designed to provide the arithmetic facilities of the **decNumber** package with a Lua flavor. It is a loadable Lua module, designed for Lua 5.1.

2.1 Distribution

The source code from the **decNumber** package and the **decTest** package are included in the **ldecNumber** distribution for a few reasons:

1. A couple of minor modifications were made to decTest to correct syntax errors.
2. The **decNumber** package is fairly small, and including it is a convenience to users as well as a means of documenting exactly the code used to build and test the **ldecNumber** package
3. IBM's liberal ICU License allows me to do so!

The distribution also includes a copy of the **decNumber** C library User's Guide, *decNumber.pdf*. This is the best reference for the specification of most of the functions included in the `decNumber` module.

2.2 Implementation Choices

A few arbitrary choices were made in the implementation of the Lua wrapper.

2.2.1 Precision

The size of structures used in the **decNumber** package determines the maximum number of decimal digits in decimal numbers it can manipulate. The default build of the `decNumber` module is configured for 69 digits. This number was chosen for a couple reasons: the resulting structure size is between 57 and 64 bytes in size (so each decimal number takes this this much space), and this precision is a bit more than twice what is needed for Decimal128 external format. Providing twice the external format's precision seemed like a good practice for mitigation of rounding issues during calculations with these numbers.

*Note: the **ldecNumber** package does not yet support any external binary formats, such as Decimal128, though it may in the future. Presently Lua strings and Lua numbers are the only "external" formats.*

In any case, you may change the value of `DECNUMDIGITS` during a build of the `decNumber` module to accommodate more digits, or reduce memory overhead if you need fewer digits. The `decNumber` module has only been tested with the default setting, and one or two others.

2.2.2 Context

The **decNumber** context provides configuration settings such as working precision and rounding mode, and also holds condition flags. Functions in the **decNumber** package take a context argument. Using this approach in Lua would have made infix operator syntax impossible. This would not have been very Lua-like.

Instead, the `decNumber` module automatically maintains a **decNumber** context per Lua thread. This thread **decNumber** context may be modified and inspected. It may also be retrieved and restored, so threads may maintain multiple contexts if desired. Modifications to the context in one thread do not affect the contexts in other threads, unless, of course, threads explicitly retrieve, exchange, and replace their contexts with a shared context. (If you don't use setcontext you don't need to worry!)

The default **decNumber** context is `DEC_INIT_DECIMAL128` as described in *decNumber.pdf*. This default may be changed during a build of the `decNumber` module by changing the value of `LDN_CONTEXT_DEFAULT`. The context method `setdefault` may be used to initialize a context to one of the well known default configurations.

2.2.3 Naming Convention

Names in the **decNumber** package follow a convention of a "decNumber" or "DEC_" prefix and mixed case, or for some constants, upper case. Lua code, on the other hand, tends toward lower case identifiers. I attempted to unify this by

- using `decNumber` as the module name
- stripping the prefix described above from the function names and constants (since the module name prefix will typically be there anyway)
- using lower case for function names, but otherwise retaining the spelling
- retaining the case for constant names

There are two cases where in following these rules the **decNumber** names clash with Lua reserved words: `or` and `and` – in these cases the **decNumber** functions are called "logical" operations, so I called the functions `lor` and `land`.

Wherever there was a predefined Lua metamethod, e. g., __add, the appropriate function is bound to that name as well as the **decNumber** package function name.

Where it seemed appropriate, functions are provided both as methods on decimal numbers, as well as functions in the `decNumber` module.

2.2.4 Mutability

The decimal numbers created by Lua are not mutable. This decision was based on my judgment that the potential performance benefit, mainly lower memory consumption and less garbage collection, was outweighed by the safety and lack of "surprise" that immutability provides. This makes decimal numbers compatible with Lua numbers and strings, both of which are also not mutable.

2.2.5 Conversion

All the functions in the `decNumber` module automatically convert their arguments from Lua numbers or strings as necessary to perform the operation. This conversion is done with the current settings in the thread **decNumber** context.

3 Example code and verification

3.1 Examples

The distribution contains an examples directory with Lua translations of some of the C examples from the **decNumber** package.

3.2 Verification Tests

The distribution contains a test directory with a few test files.

3.2.1 Unit Tests

File: *ldecNumberUnitTest.lua*

This is a small but expanding set of unit tests for the `decNumber` module. It uses the `lunit` module that can be obtained from Mike Roth's page.

File: *ldecNumberGausstest.lua*

This is a small test for the `randomstate` function and use of random states (see section 4.3.3 Operations on Random States). It defines a Gaussian random number generator, and tests it by graphing the result of many executions. It uses the **Lua DISLIN** library for graphing, see LuaForge for Lua DISLIN.

File: *ldecNumberThreadsTest.lua*

This is a simple test that the decimal context in each thread is independent. It requires visual inspection of the results to verify that thread 1 is rounding ROUND_HALF_DOWN and thread 2 is rounding ROUND_HALF_EVEN.

3.2.2 Performance Tests

File: *ldecNumberPerf.lua*

This is a simple test of the speed of some arbitrary `decNumber` module functions. It was useful to confirm that decimal context caching was worthwhile. It relies on the `lperformance` module, which is included, but has been designed for WindowsXP.

3.2.3 Compliance Test

File: *ldecNumberTestDriver.lua*

This is the big test. It uses **dectest** sources from IBM, and has over 60,000 test cases. The Lua file is a driver to execute the tests specified by **dectest**.

As of version 21 of the Lua `decNumber` module, the results for this test are: `For all 60937 tests, 59951 succeeded, 5 failed, 301 failed(conv), 644 skipped(#), 36 skipped(prec).`

This is as good as possible with the default configuration. What this means is:

59951 succeeded	woot!
5 failed	these 5 tests are know to fail in the decNumber C library; these edge cases are under reconsideration in the Decimal Number Specification
301 failed(conv)	the precision required for the operands is insufficient in the Lua wrapper
644 skipped(#)	the test is for NULL arguments or format conversions not supported
36 skipped(prec)	the test called for a precision larger than provided in the Lua wrapper

4 Reference

Here is a reference to the data types, constants, and functions in the `decNumber` module. Many of these will refer to the **decNumber** C library User's Guide, *decNumber.pdf*, for implementation details.

4.1 Data Types

The `decNumber` module defines three userdata types with their own metatables. These are decimal numbers, decimal contexts, and decimal random states.

4.1.1 Decimal Numbers

Decimal numbers are immutable numeric values. In the default configuration they can have up to 69 digits of precision, and exponents of -999999999 to 999999999.

Decimal numbers are created by the functions in the `decNumber` module. Since any arguments to these functions may be strings, Lua numbers, or decimal numbers, conversion from strings or Lua numbers to decimal numbers can be achieved in may ways. The function `decNumber.tonumber` is the most obvious.

In the descriptions below, arguments or results that may be a string, Lua number, or decimal number are indicated by `<decarg>` whereas arguments or results that must be a decimal number are indicated by `<decnum>`.

The metatable for decimal numbers is available as `decNumber.number_metatable`.

4.1.2 Decimal Contexts

Decimal contexts are mutable records that contain

- flags that control behaviors of the `decNumber` module, such as precision and rounding
- conditions that report the status of sequence of operations, such as overflow

See section 2.2.2 Context for some rational on how the `decNumber` module manages contexts.

In the descriptions below, arguments or results that must be a decimal context are indicated by `<decctx>`.

The metatable for decimal contexts is available as `decNumber.context_metatable`.

4.1.3 Random States

Random states are mutable records that contain opaque data for generation of random numbers.

See 4.3.3 Operations on Random States for the description of functions for creating and using random states.

In the descriptions below, arguments or results that must be a decimal context are indicated by `<decrst>`.

The metatable for decimal contexts is available as `decNumber.drandom_metatable`.

4.2 Constants

Here are the constants exposed in the `decNumber` module from the **decNumber** package.

4.2.1 Rounding

These numeric flags are used with `<decctx>:setround(x)`

`decNumber.ROUND_CEILING`	round towards +infinity
`decNumber.ROUND_UP`	round away from 0
`decNumber.ROUND_HALF_UP`	0.5 rounds up
`decNumber.ROUND_HALF_EVEN`	0.5 rounds to nearest even
`decNumber.ROUND_HALF_DOWN`	0.5 rounds down
`decNumber.ROUND_DOWN`	round towards 0 (truncate)
`decNumber.ROUND_FLOOR`	round towards -infinity
`decNumber.ROUND_05UP`	round for reround

4.2.2 Status Flags

These numeric status flags are used with `<decctx>:getstatus()`

```
decNumber.Conversion_syntax
decNumber.Division_by_zero
decNumber.Division_impossible
decNumber.Division_undefined
decNumber.Insufficient_storage
decNumber.Inexact
decNumber.Invalid_context
decNumber.Invalid_operation
decNumber.Overflow
decNumber.Clamped
decNumber.Rounded
decNumber.Subnormal
decNumber.Underflow
```

These constants are combinations of the above status flags:

```
decNumber.IEEE_854_Division_by_zero
decNumber.IEEE_854_Inexact
decNumber.IEEE_854_Invalid_operation
decNumber.IEEE_854_Overflow
decNumber.IEEE_854_Underflow
```

`decNumber.Errors`	normally errors (results are qNaN, infinite, or 0)
`decNumber.NaNs`	cause a result to become qNaN
`decNumber.Information`	normally for information only (have finite results)

4.2.3 Classifications

These numeric classifications for **decNumber**s are aligned with IEEE 754r and are returned by `<decnum>:class()`. Note that 'normal' and 'subnormal' are meaningful only with a `decContext`.

```
decNumber.CLASS_SNAN
decNumber.CLASS_QNAN
decNumber.CLASS_NEG_INF
decNumber.CLASS_NEG_NORMAL
decNumber.CLASS_NEG_SUBNORMAL
decNumber.CLASS_NEG_ZERO
decNumber.CLASS_POS_ZERO
decNumber.CLASS_POS_SUBNORMAL
decNumber.CLASS_POS_NORMAL
decNumber.CLASS_POS_INF
```

These classifications are also returned as string values from `<decnum>:classasstring()`.

4.2.4 Initialization Descriptors

These constants are used with `<decctx>:setdefault(x)`

`decNumber.INIT_BASE`	see note re: traps at 4.3.1.1 below, in `setdefault`
`decNumber.INIT_DECIMAL32`	
`decNumber.INIT_DECIMAL64`	
`decNumber.INIT_DECIMAL128`	

4.2.5 Compile time configuration

These constants provide information about the compile time configuration.

`decNumber.MAX_DIGITS`	constant `DECNUMDIGITS`, the maximum precision
`decNumber.version`	a string with the `decNumber` module version information

4.3 Functions and Methods

4.3.1 Operations on Contexts

The following functions operate on decimal contexts. See the **decNumber** C library User's Guide, *decNumber.pdf* for details about contexts, and constants used for manipulating them.

4.3.1.1 Accessing, Replacing, and Initializing Contexts

`decNumber.getcontext ()`

Returns the thread's current decimal context.

`<decctx>:duplicate()`

Returns a copy of the decimal context argument. This may be used, for example, to save and restore a context around temporary modifications, or to keep multiple decimal contexts on hand for quick wholesale context changes (rather than changing individual fields). You probably don't ever *need* to do this!

`decNumber.setcontext (<decctx>)`

`<decctx>:setcontext ()`

Sets the thread's decimal context to the argument, and returns the previous decimal context, i. e., the one that was just replaced. You probably don't ever *need* to do this!

`<decctx>:setdefault (initconst)`

Initializes the context argument to the settings specified by the `initconst`. See3.2.3 Initialization Descriptors for permitted values for `initconst`. No values are returned.

Note: since traps are not supported, `decNumber.INIT_BASE` differs from the behavior documented in the **decNumber** C library User Guide in that it leaves traps disabled.

Uses the C library function decContextDefault().

4.3.1.2 Accessing and Modifying Context Fields

`<decctx>:getclamp ()`

Returns the integer value of the `clamp` field of the context argument. When 0, a result exponent is limited to `emax` (for example, the exponent of a zero result will be clamped to this value).When 1, a result exponent is limited to `emax`–(`digits`–1).

`<decctx>:getdigits ()`

Returns the integer value of the `digits` field of the context argument. This is the working precision for this decimal context. The results of decimal number operations will be rounded to this length if necessary.

`<decctx>:getemax ()`

Returns the integer value of the `emax` field of the context argument. This is the magnitude of the largest adjusted exponent that is permitted.

`<decctx>:getemin ()`

Returns the integer value of the `emin` field of the context argument. This is the smallest adjusted exponent that is permitted for normal numbers.

`<decctx>:getround ()`

Returns the integer value of the `round` field of the context argument. See 3.2.1 Rounding for possible values.

`<decctx>:getstatus ()`

Returns the integer value of the `status` field of the context argument. See 3.2.2 Status Flags for

possible values. In general, the result will be a bitwise-or of a subset of these values.

`<decctx>:getstatusstring ()`

Returns a string derived from the present value of the context argument's `status` field using the C library function decContextStatusToString().

`<decctx>:gettraps ()`

Returns 0 (we hope!) since traps are not implemented in the Lua wrapper – use the `status` flags instead!

`<decctx>:setclamp (num)`

Sets the integer value of the `clamp` field of the context argument to `num`. When 0, a result exponent is limited to `emax` (for example, the exponent of a zero result will be clamped to this value).When 1, a result exponent is limited to `emax`–(`digits`–1).

Returns the previous value of the `clamp` field.

Note that it is an error if `num` is not one of the values 0 or 1.

`<decctx>:setdigits (num)`

Sets the integer value of the `digits` field of the context argument to `num`. This is the working precision for this decimal context. The results of subsequent decimal number operations will be rounded to this length if necessary.

Returns the previous value of the `digits` field.

Note that it is an error if `num` exceeds `decNumber.MAX_DIGITS`.

`<decctx>:setemax (num)`

Sets the integer value of the `emax` field of the context argument to `num`. This is the magnitude of the largest adjusted exponent that is permitted.

Returns the previous value of the `emax` field.

Note that it is an error if `num` is outside the range 0 though 999,999,999.

`<decctx>:setemin (num)`

Sets the integer value of the `emin` field of the context argument to `num`. This is the smallest adjusted exponent that is permitted for normal numbers. `emin` will usually equal –`emax`, but when a compressed format is used it will be –(`emax`–1).

Returns the previous value of the `emin` field.

Note that it is an error if `num` is outside the range -999,999,999 though 0.

`<decctx>:setround (rounding)`

Sets the integer value of the `round` field of the context argument to `rounding`. This is used to select the rounding algorithm to be used if rounding is necessary during subsequent decimal number operations.

Returns the previous value of the `round` field.

Note that it is an error if `rounding` is not one of the values described in 3.2.1 Rounding

---

`<decctx>:setstatus (flags)`

Sets the integer value of the `status` field of the context argument to `flags`. See 3.2.2 Status Flags for possible values. In general, the `flags` will be a bitwise-or of a subset of these values. Usually the `flags` are 0 to clear all conditions.

Returns the previous value of the `status` field.

Note that it is an error if `flags` is not a bitwise-or of a subset of the values in 3.2.2 Status Flags

---

`<decctx>:setstatusstring (flagname)`

Sets the decimal context's `status` bit corresponding to the name string argument `flagname` using the C library function decContextSetStatusFromString().

`<decctx>:settraps (flags)`

Sets the integer value of the `traps` field of the context argument to `flags`.

Note that it is an error if `flags` is not zero since traps are not implemented in the Lua wrapper – use the `status` flags instead!

4.3.2 Operations on Numbers

The following functions operate on decimal numbers. In the descriptions below, arguments or results that may be a string, Lua number, or decimal number are indicated by `<decarg>` whereas arguments or results that must be a decimal number are indicated by `<decnum>`. See 3.1.1 Decimal Numbers

See the **decNumber** C library User's Guide, *decNumber.pdf* for details about limitations of the functions, and behavior in exceptional situations.

4.3.2.1 Conversions

`<decnum>:__concat (<string>)`

Returns a string representing the value of the decimal number argument concatenated with the string argument. See `tostring` below for a description of the conversion operation.

Note that by binding the method `__concat` to this function, the Lua `..` concatenation operator will work with a decimal number and a string, in either order, due to Lua's internal application of this method when one side of the `..` concatenation operator is a decimal number.

Uses the C library function decNumberToString().

`<decnum>:toengstring ()`

`decNumber.toengstring (<decarg>)`

Returns a string representing the value of the argument (converted to a decimal number first if necessary) using engineering notation (where the exponent will be a multiple of three, and there may be up to three digits before any decimal point) if an exponent is needed. It implements the *to–engineering–string* conversion.

Uses the C library function decNumberToEngString().

`decNumber.tonumber (<decarg>)`

Returns a decimal number. If the argument is a decimal number, it is simply returned. If the argument is a Lua number or string, it is converted, using the present decimal context as usual, to a decimal number and this value is returned.

Uses the C library function decNumberFromString().

`<decnum>:tostring ()`

`<decnum>:__tostring ()`

`decNumber.tostring (<decarg>)`

Returns a string representing the value of the argument (converted to a decimal number first if necessary) using scientific notation if an exponent is needed (that is, there will be just one digit before any decimal point). It implements the *to–scientific–string* conversion.

Note that by binding the method `__tostring` to this function, the Lua `print` and `tostring` functions will work with decimal numbers, using this conversion function.

Uses the C library function decNumberToString().

4.3.2.2 Operations

`<decnum>:abs ()`

`decNumber.abs (<decarg>)`

Returns a decimal number that is the absolute value of the argument.

Uses the C library function decNumberAbs()

`<decnum>:add (<decarg>)`

`<decnum>:__add (<decarg>)`

`decNumber.add (<decarg>, <decarg>)`

Returns a decimal number that is the sum of its arguments.

Note that by binding the method `__add` to this function, the Lua addition operator (+) may be used with a <decnum> on the left and a <decarg> on the right.

Uses the C library function decNumberAdd().

`<decnum>:copy ()`

`decNumber.copy (<decarg>)`

Returns a decimal number that is a copy of its argument. This is not too useful since `<decnum>`s are immutable in **ldecNumber**, but it could be used as an alternative to `<decnum>:tonumber()`

No error is possible from this function when its argument is a `<decnum>`.

Uses the C library function decNumberCopy().

`<decnum>:copyabs ()`

`decNumber.copyabs (<decarg>)`

Returns a decimal number that is the absolute value of its argument.

This is the quiet `abs` function described in IEEE 754r.

No error is possible from this function when its argument is a `<decnum>`.

Uses the C library function decNumberCopyAbs().

`<decnum>:copynegate ()`

`decNumber.copynegate (<decarg>)`

Returns a decimal number that is the negation of its argument, in other words it returns a copy of its argument with the sign inverted.

This is the quiet `negate` function described in IEEE 754r.

No error is possible from this function when its argument is a `<decnum>`.

Uses the C library function decNumberCopyNegate().

`<decnum>:copysign (<decarg>)`

`decNumber.copysign (<decarg>, <decarg>)`

Returns a decimal number that is a copy of its first argument but with the sign of its second argument.

This is the quiet `copysign` function described in IEEE 754r.

No error is possible from this function when its arguments are both `<decnum>`s.

Uses the C library function decNumberCopySign().

`<decnum>:divide (<decarg>)`

`<decnum>:__div (<decarg>)`

`decNumber.divide (<decarg>, <decarg>)`

Returns a decimal number that is the left (1st) argument divided by the right (2nd) argument.

Note that by binding the method `__div` to this function, the Lua division operator (`/`) may be used with a <decnum> on the left and a <decarg> on the right.

Uses the C library function decNumberDivide().

`<decnum>:divideinteger (<decarg>)`

`decNumber.divideinteger (<decarg>, <decarg>)`

Returns a decimal number that is the integer part of the result of dividing of its arguments. Note that, per the decNumber specification, this is a convert to integer by truncation. If you want some other rounding mode, use `floor`, or for any rounding mode use `tointegralvalue` or `quantize`.

Uses the C library function decNumberDivideInteger().

`<decnum>:exp ()`

`decNumber.exp (<decarg>)`

Returns a decimal number that is *e* raised to the power of the argument.

Uses the C library function decNumberExp().

`<decnum>:floor (<decarg>)`

`decNumber.floor (<decarg>,<decarg>)`

Returns a decimal number integer that is the floor of the left (1st) argument divided by the right (2nd) argument. Contrast this with `divideinteger` which uses truncation.

The `floor` function is implemented as equal to `divideinteger` if the signs of the arguments are the same or if the remainder is zero, otherwise as equal to the `divideinteger` result minus 1. The current context's rounding mode is used. See `mod`.

Uses the C library function decNumberDivideInteger(), and then decNumberMultiply() followed by decNumberCompare() decNumberIsZero() to check if the remainder is zero, and decNumberSubtract().

`<decnum>:fma (<decarg>, <decarg>)`

`decNumber.fma (<decarg>, <decarg>, <decarg>)`

Returns a decimal number that is the result of multiplying the first argument by the second argument and then adding the third argument to that intermediate result. It is equivalent to a multiplication followed by an addition except that the intermediate result is not rounded and will not cause overflow or underflow. That is, only the final result is rounded and checked.

Uses the C library function decNumberFMA().

`<decnum>:invert ()`

`decNumber.invert (<decarg>)`

Returns a decimal number that is the result of the digit-wise logical inversion of the argument (a 0 digit becomes 1 and vice versa).

Uses the C library function decNumberInvert().

`<decnum>:land (<decarg>)`

`decNumber.land (<decarg>, <decarg>)`

Returns a decimal number that is the digit-wise logical and of the arguments. Note that all digits of the arguments must be 0 or 1 or else this operation returns NaN,

Uses the C library function decNumberAnd().

`<decnum>:ln ()`

`decNumber.ln (<decarg>)`

Returns a decimal number that is the natural logarithm (logarithm in base *e*) of the argument.

Uses the C library function decNumberLn().

`<decnum>:log10 ()`

`decNumber.log10 (<decarg>)`

Returns a decimal number that is the logarithm in base ten of the argument.

Uses the C library function decNumberLog10().

`<decnum>:logb ()`

`decNumber.logb (<decarg>)`

Returns a decimal number that is the adjusted exponent of the argument, according to the rules for the `logB` operation of the IEEE 754r proposal. This returns the exponent of the argument as though its decimal point had been moved to follow the first digit while keeping the same value. The result is not limited by `emin` or `emax`.

Uses the C library function decNumberLogB().

`<decnum>:lor (<decarg>)`

`decNumber.lor (<decarg>, <decarg>)`

Returns a decimal number that is the digit-wise logical inclusive or of the arguments. Note that all

digits of the arguments must be 0 or 1 or else this operation returns NaN,

Uses the C library function decNumberOr().

`<decnum>:max (<decarg>)`

`decNumber.max (<decarg>, <decarg>)`

Returns a decimal number that is the maximum of its arguments.

Uses the C library function decNumberMax().

`decnum:maxmag (decarg)`

`decNumber.maxmag (decarg, decarg)`

Returns a decimal number that is the one of its arguments that has the maximum magnitude. It is identical to `max` except that the signs of the operands are ignored and taken to be 0 (non-negative).

Uses the C library function decNumberMaxMag().

`<decnum>:min (<decarg>)`

`decNumber.min (<decarg>, <decarg>)`

Returns a decimal number that is the minimum of its arguments.

Uses the C library function decNumberMin().

`decnum:minmag (decarg)`

`decNumber.minmag (decarg, decarg)`

Returns a decimal number that is the one of its arguments that has the minimum magnitude. It is identical to `min` except that the signs of the operands are ignored and taken to be 0 (non-negative).

Uses the C library function decNumberMinMag().

`<decnum>:minus ()`

`<decnum>:__unm ()`

`decNumber.minus (<decarg>)`

Returns a decimal number that is the result of subtracting the argument from 0.

Note that by binding the method `__unm` to this function, the Lua unary minus operator (−) may be used with decimal numbers.

Uses the C library function decNumberMinus().

`<decnum>:mod (<decarg>)`

`<decnum>:__mod (<decarg>)`

`decNumber.mod (<decarg>, <decarg>)`

Returns a decimal number that is remainder of the left (1st) argument divided by the right (2nd) argument based on Lua rules for the mod operator (%). Lua defines

```
a % b == a - floor(a/b)*b
```

whereas the General Decimal Arithmetic Specification defines remainder using truncation.

The mod function is implemented as equal to remainder if the signs of the arguments are the same or the remainder is zero, otherwise as equal to the remainder plus the divisor. The current context's rounding mode is used.

Note that by binding the method __mod to this function, the Lua modulo operator (%) may be used with a <decnum> on the left and a <decarg> on the right.

Uses the C library functions decNumberRemainder(), decNumberIsNegative(), decNumberIsZero(), and decNumberAdd().

`<decnum>:multiply (<decarg>)`

`<decnum>:__mul (<decarg>)`

`decNumber.multiply (<decarg>, <decarg>)`

Returns a decimal number that is the product of its arguments.

Note that by binding the method __mul to this function, the Lua multiplication operator (*) may be used with a <decnum> on the left and a <decarg> on the right.

Uses the C library function decNumberMultiply().

`<decnum>:nextminus ()`

`decNumber.nextminus (<decarg>)`

Returns a decimal number that is the closest value to the argument in the direction of -Infinity. This is computed as though by subtracting an infinitesimal amount from the argument using ROUND_FLOOR, except that no flags are set as long as the argument is a <decnum> (unless the argument is a signaling NaN).

This function is a generalization of the IEEE 754 nextDown operation.

Uses the C library function decNumberNextMinus().

`<decnum>:nextplus ()`

`decNumber.nextplus (<decarg>)`

Returns a decimal number that is the closest value to the argument in the direction of +Infinity. This is computed as though by adding an infinitesimal amount from the argument using ROUND_CEILING, except that no flags are set as long as the argument is a <decnum> (unless the argument is a signaling NaN).

This function is a generalization of the IEEE 754 nextUp operation.

Uses the C library function decNumberNextPlus().

`<decnum>:nexttoward (<decarg>)`

`decNumber.nexttoward (<decarg>, <decarg>)`

Returns a decimal number that is the closest value to the first argument in the direction of the second argument. This is computed as though by adding or subtracting an infinitesimal amount to the first argument using either ROUND_CEILING or ROUND_FLOOR depending on whether the second argument is larger or smaller than the first argument. If the arguments are numerically equal, then the result is a copy of the first argument with the sign of the second argument. Flags are set as usual for an

addition or subtraction (no flags are set in the equals case).

This function is a generalization of the IEEE 754 `nextAfter` operation.

Uses the C library function decNumberNextToward().

`<decnum>:normalize ()`

`decNumber.normalize (<decarg>)`

Returns a decimal number that is the result of adding the argument to 0, and putting the result in its simplest form. That is, a non-zero number which has any trailing zeros in the coefficient has those zeros removed by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly, and a zero has its exponent set to 0.

Uses the C library function decNumberNormalize()

`<decnum>:plus ()`

`decNumber.plus (<decarg>)`

Returns a decimal number that is the result of adding the argument to 0. This takes place according to the settings given in the decimal context, following the usual arithmetic rules. This may therefore be used for rounding.

Uses the C library function decNumberPlus().

`<decnum>:power (<decarg>)`

`<decnum>:__pow (<decarg>)`

`decNumber.power (<decarg>, <decarg>)`

Returns a decimal number that is the left (1st) argument raised to the power of the right (2nd) argument.

Note that by binding the method `__pow` to this function, the Lua power operator (^) may be used with a <decnum> on the left and a <decarg> on the right.

Uses the C library function decNumberPower().

`<decnum>:quantize (<decarg>)`

`decNumber.quantize (<decarg>, <decarg>)`

Returns a decimal number that is numerically equal (except for any rounding) to the left (1st) argument but modified so its exponent has a specific value, equal to that of the right (2nd) argument. To achieve this, the coefficient of the result is adjusted (by rounding or shifting) so that its exponent has the requested value. For example, if the left (1st) argument had the value 123.4567, and the right (2nd) argument had the value 0.12, the result would be 123.46 (that is, 12346 with an exponent of –2, matching the right (2nd) argument).

Uses the C library function decNumberQuantize().

`<decnum>:remainder (<decarg>)`

`decNumber.remainder (<decarg>, <decarg>)`

Returns a decimal number that is the remainder of the left (1st) argument divided by the right (2nd) argument. The identity

```
lhs == (lhs:divideinteger(rhs) * rhs) + lhs:remainder(rhs)
```

holds.

Uses the C library function decNumberRemainder().

```
<decnum>:remaindernear (<decarg>)
```

```
decNumber.remaindernear (<decarg>, <decarg>)
```

Returns a decimal number that is the remainder of the left (1st) argument divided by the right (2nd) argument using the rules defined in IEEE 854.This follows the same definition as `decNumber.remainder`, except that the nearest integer quotient (or the nearest even integer if the remainder is equidistant from two) is used instead of the result from `decNumber.divideinteger`.

For example, `decNumber.remaindernear(10, 6)` has the result -2 (instead of 4) because the multiple of 6 nearest to 10 is 12 (rather than 6).

Uses the C library function decNumberRemainderNear().

```
<decnum>:rescale (<decarg>)
```

```
decNumber.rescale (<decarg>, <decarg>)
```

Returns a decimal number that is numerically equal (except for any rounding) to the left (1st) argument but modified so its exponent has the value of the right (2nd) argument. (See `decNumber.quantize`). The right (2nd) argument must be a whole number (before any rounding); that is, any digits in the fractional part of the number must be zero.

Uses the C library function decNumberRescale().

```
<decnum>:rotate (<decarg>)
```

```
decNumber.rotate (<decarg>, <decarg>)
```

Returns a decimal number that is the first argument with the digits of its coefficient rotated to the left (if the second argument is positive) or to the right (if the second argument is negative) without adjusting the exponent or the sign.

If the first argument has fewer digits than context `digits` the coefficient is padded with zeros on the left before the rotate. Any leading zeros in the result are ignored, as usual.

The second argument is the count of digits to rotate; it must be an integer (that is, it must have an exponent of 0) and must be in the range `-digits` through `+digits` in the current context.

Uses the C library function decNumberRotate().

```
<decnum>:samequantum (<decarg>)
```

```
decNumber.samequantum (<decarg>, <decarg>)
```

Returns the decimal number 1 when the exponents of the arguments are equal, or if they are both Infinities or they are both NaNs; in all other cases returns the decimal number 0. This function is used to test whether the exponents of two numbers are equal. The coefficients and signs of the arguments are ignored.

Uses the C library function decNumberSameQuantum().

`<decnum>:scaleb (<decarg>)`

`decNumber.scaleb (<decarg>, <decarg>)`

This function returns the result of multiplying the first argument by ten raised to the power of the second argument. It is used to adjust (scale) the exponent of a number, using the rules of the `scaleB` operation in the IEEE 754r proposal. The second argument must be an integer (that is, it must have an exponent of 0) and it must also be in the range `-n` through `+n`, where `n` is `2 *(emax + digits)` in the present context.

Uses the C library function decNumberScaleB().

`<decnum>:shift (<decarg>)`

`decNumber.shift (<decarg>, <decarg>)`

Returns a decimal number that is the first argument with the digits of its coefficient shifted to the left (if the second argument is positive) or to the right (if the second argument is negative) without adjusting the exponent or the sign.

The coefficient is padded with zeros on the left or right, as necessary. Any leading zeros in the result are ignored, as usual.

The second argument is the count of digits to shift; it must be an integer (that is, it must have an exponent of 0) and must be in the range `-digits` through `+digits` in the current context.

Uses the C library function decNumberShift().

`<decnum>:squareroot ()`

`decNumber.squareroot (<decarg>)`

Returns a decimal number that is the square root of its argument, rounded if necessary using the digits setting in the decimal context and using the round–half–even rounding algorithm.

Uses the C library function decNumberSquareRoot().

`<decnum>:subtract (<decarg>)`

`<decnum>:__sub (<decarg>)`

`decNumber.subtract (<decarg>, <decarg>)`

Returns a decimal number that is the left (1st) argument minus the right (2nd) argument.

Note that by binding the method `__sub` to this function, the Lua subtraction operator (`-`) may be used with a <decnum> on the left and a <decarg> on the right.

Uses the C library function decNumberSubtract().

`<decnum>:tointegralexact ()`

`decNumber.tointegralexact (<decarg>)`

Returns a decimal number that is the argument with any fractional part removed, if necessary, using the rounding mode in the decimal context.

The `Inexact` flag is set if the result is numerically different from the argument. Other than that, no flags are set as long as the argument is a <decnum> (unless the argument is a signaling NaN).

The result may have a positive exponent.

Uses the C library function decNumberToIntegralExact().

`<decnum>:tointegralvalue ()`

`decNumber.tointegralvalue (<decarg>)`

Returns a decimal number that is the argument with any fractional part removed, if necessary, using the rounding mode in the decimal context.

No flags, not even `Inexact`, are set as long as the argument is a <decnum> (unless the argument is a signaling NaN).

The result may have a positive exponent.

Uses the C library function decNumberToIntegralValue().

`<decnum>:trim ()`

`decNumber.trim (<decarg>)`

Returns a decimal number that is the argument with any insignificant trailing zeros removed. That is, if the number has any fractional trailing zeros they are removed by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly.

Uses the C library function decNumberTrim().

`<decnum>:xor (<decarg>)`

`decNumber.xor (<decarg>, <decarg>)`

Returns a decimal number that is the digit-wise logical exclusive or of the arguments. Note that all digits of the arguments must be 0 or 1 or else this operation returns NaN,

Uses the C library function decNumberXor().

4.3.2.3 Comparisons and Predicates

`<decnum>:class ()`

`decNumber.class (<decarg>)`

Returns the class of a decNumber. No error is possible. The class is one of the decNumber Classifications. See: 4.2.3 Classifications.

Uses the C library function decNumberClass().

`decnum:classasstring ()`

`decNumber.classasstring (decarg)`

Returns the class of a decNumber as a string. No error is possible. The class is one of "-Infinity", "-Normal", "-Subnormal", "-Zero", "+Zero", "+Subnormal","+Normal", "+Infinity", "NaN", "sNaN", or "Invalid"

Uses the C library functions decNumberClass() and decNumberClassToString().

`decNumber.classtostring (<enum>)`

Converts the Classifications (see: 4.2.3 Classifications) of a decNumber to a string. No error is possible. The class is one of "-Infinity", "-Normal", "-Subnormal", "-Zero", "+Zero", "+Subnormal", "+Normal", "+Infinity", "NaN", "sNaN", or "Invalid".

Uses the C library function decNumberClassToString().

`<decnum>:compare (<decarg>)`

`decNumber.compare (<decarg>, <decarg>)`

Returns a decimal number that is the comparison of its arguments numerically. If the left (1st) argument is less than the right (2nd) argument then the result is –1. If they are equal (that is, when subtracted the result would be 0), then the result is 0. If the left (1st) argument is greater than the right (2nd) argument then the result is 1.If the operands are not comparable (that is, one or both is a NaN) then the result is NaN.

Uses the C library function decNumberCompare().

`<decnum>:comparetotal (<decarg>)`

`decNumber.comparetotal (<decarg>, <decarg>)`

Returns a decimal number that is the comparison of its arguments numerically using the IEEE 754r proposed ordering. The result is the similar to `decNumber.compare` above, except that NaN is never returned. The total order differs from the numerical comparison in that:

–NaN < –sNaN < –Infinity < –finites < –0 < +0 < +finites < +Infinity < +sNaN < +NaN.

Also, 1.000 < 1.0 (etc.) and NaNs are ordered by payload.

Uses the C library function decNumberCompareTotal().

`<decnum>:comparetotalmag (<decarg>)`

`decNumber.comparetotalmag (<decarg>, <decarg>)`

Returns a decimal number that is the comparison of the magnitude of its arguments using the IEEE 754r proposed ordering. It is identical to `comparetotal` above except that the signs of the operands are ignored and taken to be 0 (non-negative).

Uses the C library function decNumberCompareTotalMag().

`<decnum>:eq (<decarg>)`

`<decnum>:__eq (<decarg>)`

`decNumber.eq (<decarg>, <decarg>)`

Returns a boolean that is true when the arguments are equal, false otherwise.

Note that by binding the method `__eq` to this function, the Lua equality operators (== and ~=) may be used with a <decnum> on the left and a <decnum> on the right.

Uses the C library functions decNumberCompare() and decNumberIsZero().

`<decnum>:iscanonical ()`

`decNumber.iscanonical (<decarg>)`

Returns true always, because decNumbers always have canonical encodings (the function is provided for compatibility with the IEEE 754r operation `isCanonical`). No error is possible.

Uses the C library function decNumberIsCanonical().

`<decnum>:isfinite ()`

`decNumber.isfinite (<decarg>)`

Returns a boolean that is true if the argument is finite, false otherwise (that is, the argument is an infinity or a NaN). No error is possible.

Uses the C library function decNumberIsFinite().

`<decnum>:isinfinite ()`

`decNumber.isinfinite (<decarg>)`

Returns a boolean that is true if the argument is infinite, false otherwise. No error is possible.

Uses the C library function decNumberIsInfinite().

`<decnum>:isnan ()`

`decNumber.isnan (<decarg>)`

Returns a boolean that is true if the argument is a NaN (quiet or signaling), false otherwise. No error is possible.

Uses the C library function decNumberIsNaN().

`<decnum>:isnegative ()`

`decNumber.isnegative (<decarg>)`

Returns a boolean that is true if the argument is negative, false otherwise. No error is possible.

Uses the C library function decNumberIsNegative().

`<decnum>:isnormal ()`

`decNumber.isnormal (<decarg>)`

Returns a boolean that is true if the argument is negative, false otherwise. No error is possible.

Uses the C library function C<decNumberIsNormal()>.

`<decnum>:isqnan ()`

`decNumber.isqnan (<decarg>)`

Returns a boolean that is true if the argument is a quiet NaN, false otherwise. No error is possible.

Uses the C library function decNumberIsQNaN().

`<decnum>:issnan ()`

`decNumber.issnan (<decarg>)`

Returns a boolean that is true if the argument is a signaling NaN, false otherwise. No error is possible.

Uses the C library function decNumberIsSNaN().

`<decnum>:isspecial ()`

`decNumber.isspecial (<decarg>)`

Returns a boolean that is true if the argument has a special value (Infinity or NaN), false otherwise; it is the inversion of `isfinite`. No error is possible.

Uses the C library function decNumberIsSpecial().

`<decnum>:issubnormal ()`

`decNumber.issubnormal (<decarg>)`

Returns a boolean that is true if the argument is subnormal (that is, finite, non-zero, and not in the range of normal values), false otherwise. No error is possible.

Uses the C library function decNumberIsSubnormal().

`<decnum>:iszero`

`decNumber.iszero (<decarg>)`

Returns a boolean that is true if the argument is zero, false otherwise. No error is possible.

Uses the C library function decNumberIsZero().

`<decnum>:le (<decarg>)`

`<decnum>:__le (<decarg>)`

`decNumber.le (<decarg>, <decarg>)`

Returns a boolean that is true when left (1st) argument is less than or equal to the right (2nd) argument, false otherwise.

Note that by binding the method `__le` to this function, the Lua comparison operators (<= and >=) may be used with a <decnum> on the left and a <decnum> on the right.

Uses the C library functions decNumberCompare() decNumberIsNegative() and decNumberIsZero().

`<decnum>:lt (<decarg>)`

`<decnum>:__lt (<decarg>)`

`decNumber.lt (<decarg>, <decarg>)`

Returns a boolean that is true when left (1st) argument is less than the right (2nd) argument, false otherwise.

Note that by binding the method `__lt` to this function, the Lua comparison operators (< and >) may be used with a <decnum> on the left and a <decnum> on the right. Lua also assumes that `a <= b` is equivalent to `not (b < a)` which in the presence of NaNs may or may not be what you want – if not, use `<decnum>:compare` directly.

Uses the C library functions decNumberCompare() and decNumberIsNegative().

`<decnum>:radix ()`

`decNumber.radix (<decarg>)`

Returns the radix (number base) used by the decNumber package. This always returns 10. No error is possible.

Uses the C library function decNumberRadix().

4.3.3 Operations on Random States

The following functions operate on random states.

The random number generator in the **ldecNumber** package is based on a lagged Fibonacci generator ("LFIB4"). George Marsaglia has this to say about LFIB4:

> *LFIB4 is an extension of what I have previously defined as a lagged Fibonacci generator [...] I have developed the 4-lag generator LFIB4 using addition [...] Its period is 2^31*(2^256-1), about 2^287, and it seems to pass all tests --- in particular, those of the kind for which 2-lag generators using +,-, xor seem to fail.*

The **ldecNumber** package uses LFIB4 to produce a stream of bits in 10-bit chunks. This is convenient for making decimal numbers in multiples of three decimal digits, and fits with the default setting of the **decNumber** compile time parameter `DECDPUN`. If you change `DECDPUN` then you may not be able to use the **ldecNumber** package's random states without modification to the C code. There is a compile time setting `LDN_ENABLE_RANDOM` that should be defined to 0 to disable random state features.

The **ldecNumber** package random state code has been tested with L'Ecuyer and Simard's TestU01 Crush (see http://www.iro.umontreal.ca/~simardr/testu01/tu01.html)

```
========= Summary results of Crush =========

 Version:          TestU01-1.1

 Generator:        Generator dec12

 Number of statistics:  144

 Total CPU time:   02:55:11.06

 All tests were passed
```

While this confers a great deal of confidence in the quality of the generator, two caveats are in order:

1. Crush uses IEEE doubles, and so the quality of the generator with more than a dozen or so digits is untested, though believed to be good
2. This generator was not designed for cryptographic use, and so is probably only useful for its intended applications: simulation and testing

---

`decNumber.randomstate ([a [, b [, c [, d]]]])`

Creates and returns a new random state `<decrst>`. If any arguments are not supplied, defaults are provided. The arguments `a`, `b`, `c`, and `d` are used as inputs to the random number generator KISS to initialize the random state.

`<decrst>([digits [, exponent]])`

Returns a new random decimal number. If supplied, `digits` is the number of decimal digits in the new random decimal number; the default is 12. If supplied, `exponent` is the exponent of the new random decimal number; the default is `-digits` so the new random decimal number is between zero (inclusive) and one (exclusive).