General Decimal Arithmetic Testcases

24th March 2009

Mike Cowlishaw

IBM Fellow
IBM UK Laboratories

mfc@uk.ibm.com

Version 2.44

Table of Contents

Introduction

This document describes testcases designed for testing implementations of the general purpose floating-point arithmetic defined in the **General Decimal Arithmetic Specification**.[1]

The testcases are intended to be both language-independent and representation-independent.[2] They comprise individually identified tests, each describing a single operation and its expected results. The tests are grouped into files to make it easier to test a new implementation incrementally, and are available at the General Decimal Arithmetic page, in `dectest.zip` (extended arithmetic) and `dectest0.zip` (subset arithmetic).

The testcase files should be considered experimental (or "beta"), and may contain errors. They are offered on an as-is basis. In particular, even passing all the tests does not guarantee that an implementation complies with any Standard or specification, because the tests are not exhaustive.

Comments on this draft and the testcases are encouraged. Please send any comments, suggestions, and corrections to the author, Mike Cowlishaw (`mfc@uk.ibm.com`).

For further background details, including specifications in various formats and related decimal arithmetic links, please see the material at the General Decimal Arithmetic web site.

Appendix A (see page 21) summarizes the changes to this document since the first public draft.

Acknowledgements

The author is indebted to Aahz, Paul-Georges Crismer, and Tim Peters for their contributions to this document. In addition, many people have contributed directly or indirectly to the testcases themselves; special thanks are due to Brian Marks for his meticulous investigations into different implementations of the base arithmetic.

1 See `http://speleotrove.com/decimal/decarith.html`
2 A mechanism is provided which permits specific representations to be encoded, however.

Testcase file format

This section describes the format of the testcase files. These are distributed as plain text files with a file name that identifies the *group* of tests contained in the file and a file extension (if appropriate) of `.decTest`. File names will not have embedded blanks.

The files are encoded using one byte per character, using 7-bit ASCII encoding. These may be converted to Unicode by treating them as UTF-8-encoded files or by directly converting each 7-bit character to Unicode by prefixing nine 0 bits.

Testcase syntax

Each testcase file consists of one or more lines (the line delimiter mechanism may vary, depending on the operating system). Within each line, control characters (those with encodings in the range 0 through 31) are not used.

Each line is treated as a sequence of *tokens*, delimited by the start of the line, spaces between tokens, or (after the final token) the end of the line. There may be more than one space between tokens, and spaces may also appear before the first token on a line and after the last token on a line. Tokens may also be quoted, to include spaces (see below).

If the first two characters of a token are two hyphens (`--`) the token indicates the start of a comment. The two-hyphen sequence and any characters that follow it, up to the end of the line on which the sequence occurs, are ignored (that is, they are treated as commentary).

The lines in a file may be:

empty Lines with no characters, or only space characters. These lines are treated as commentary and are ignored.

comments Lines whose first token starts a comment. These are also ignored.

directives Lines which are used to control the testcase environment in some way, for example to change the working precision. These lines have two tokens and are of the form:

keyword: *value*

where the case-independent *keyword* describes the purpose of the directive, and the *value* is a parameter associated with the directive. The possible keywords and their values are described below (see page 11).

tests Lines which describe a specific test. These lines have at least five tokens, and are of the form:

id operation operand1 operand2 operand3 `->` *result conditions*

where the named tokens are as follows:

id — A short name which identifies the test. It is case-independent and unique in the file. In the current testcases it comprises three or four alphabetic characters followed by three or four digits (for example, `divx101`).

operation — A case-independent keyword which describes the operation (see page 14) to be carried out for this test (for example, `divide`).

operand1 — The first (or only) operand required for the operation. The token may be quoted (see below). If it is not quoted then its value is the sequence of characters comprising the token, taken exactly as written.

If the value of a token includes an octothorpe character (#, also called the hash or pound sign), the operand is a specific *format-dependent representation*, as described below.

Otherwise, the value of the token is treated as a string, which should be converted to a number using the **to-number** conversion of the arithmetic specification.

operand2 — An optional second operand, if required for the operation. If present, it has the same syntax and follows the same rules as *operand1*.

operand3 — An optional third operand, if required for the operation. If present, it has the same syntax and follows the same rules as *operand1* and there must be an *operand2*.

result — This defines the result of carrying out the *operation* on the operand or operands, and may also be quoted. It will either be the string form of a valid number or a format-dependent representation (see below), or the question mark character (?) which indicates that the result is undefined. The latter is only used in tests for the X3.274 subset of the specification, for error results.

conditions — Zero or more tokens each of which is the case-independent name of a *condition* (see page 16) set by the operation. If no condition is set then no condition tokens will be present.

Quoted tokens

Any operand or result token may be quoted. That is, it may begin with a delimiter which is a single or double quote character and it is then only ended by a matching quote (which must be present and on the same line). A quote which matches the starting quote may be included inside a quoted operand by doubling up the quote; in this case it does not end the token.

The content of the quoted token (after the delimiters have been removed and any doubled delimiter quotes have been reduced to a single instance) forms the value of the token. The value may contain any 7-bit ASCII characters (other than the control characters, whose encodings are in the range 0 through 31), including spaces or the comment start sequence.

If the value of an operand token is not a valid number or format-dependent representation (that is, its syntax is not valid) then the token must be quoted.[3] Result tokens must always have correct syntax.

3 This rule allows future extensions to the syntax of tests.

Processing of a test case line

When the value of an operand token is converted to a number before use, using the **to-number** conversion, it is subject to the values set by directives (see page 11), except that the `precision` directive is only used when the operation (see page 14) is `toSci`, `toEng`, or `apply`. For other operations, sufficient precision is used so that rounding of the operand is avoided.[4] The conversion of operands to numbers may set flags, such as the Rounded or Inexact flags, if the "perfect" value of the operand is outside the bounds set by the directives.

In contrast, the value of a result token is effectively converted to a number without constraints – no flags will be set by this process. The "perfect" result number is compared with the result of the operation on the operand(s) and must match exactly if the test is to succeed.

Note that in all cases the actual processing by a test case interpreter need only act *as though* the steps described here are carried out.

Format-dependent representations

When the value of an operand or result token includes the octothorpe character (#, also called the hash or pound sign), the operand or result is derived from or creates a specific decimal format.

The operand or result token must have one of the following syntaxes:

- The octothorpe alone. For example, `"#"`.

  In this case, the token is a null reference, which can only be used as an operand. Null references are used for testing the behavior of implementations which can be passed numbers by reference, where a null reference would be an error. If this concept is not supported then tests containing null references should be skipped.

- The octothorpe followed by exactly 8, 16, or 32 hexadecimal digits, where a hexadecimal digit is one of the characters `'0'` through `'9'` or `'a'` through `'f'` (in uppercase or lowercase). For example, `"#A23003D0"`.

  In this case, the token is an explicit hexadecimal representation in one of the decimal floating-point formats (decimal16, decimal32, or decimal128, respectively) described in the IEEE 754 standard.

  When used as an operand, these formats are decoded without loss or constraint and are then subject to the values set by directives (see page 11), except that the `precision` directive is only used when the operation (see page 14) is `toSci`, `toEng`, or `apply`. For other operations, sufficient precision is used so that rounding of the operand is avoided (these are the same rules as are applied to operands supplied as numerical strings).

  When used as a result, these formats imply encoding of the result of the operation into the given format. This may modify or clamp the result to fit the format, giving a different result than if the result had been expressed as a string and possibly raising new conditions. Note that the `toSci` and `toEng` operations cannot be used if the result is format-dependent, as these require specific string results (use `apply` instead).

- One of the strings `"32#"`, `"64#"`, or `"128#"`, immediately followed by a numeric string. For example, `"32#-7.50"`.

4 This rule allows the testing of rounding in the `toSci`, `toEng`, and `apply` operations, and also permits the testing of the `lost_digits` condition in the other operations.

In this case, the token forms an alternative method of specifying a number in a particular format, with the characters before the # selecting the target format.

For an operand, the numeric string is first converted to a number and is then encoded in the selected format (this may cause rounding or other conditions, and these conditions will be set as usual). The resulting encoding is then used as though it had been specified in explicit hexadecimal form, as described above.

For a result, the numeric string is again first converted to a number and is then encoded in the selected format (this may cause rounding or other conditions, but *no* conditions will be set by this process). The resulting encoding is then used as though it had been specified as the result in explicit hexadecimal form, as described above.

Example

Here is an example of a small testcase file, comprising some commentary, directives which set the version and context, and some tests.

```
-- simple.decTest
-- Testcase for some simple operations.
Version: 2.44

Precision:   9
Rounding:    half_up
MaxExponent: 999
MinExponent: -999

simp001  add       1 1 -> 2   -- can we get this right?
simp002  multiply  2 2 -> 4
simp003  divide    1 3 -> 0.333333333  Inexact Rounded
simp004  divide    1 0 -> NaN Division_by_zero
simp005  toSci  '1..2' -> NaN Conversion_syntax
```

Note: Tokens and lines do not have a defined length limit, however the current testcases are limited to a maximum token length of 1050 characters and a maximum line length of 4000 characters.

Directives

Directives are used to control the testcase environment in some way. Each has a *keyword* (which is immediately followed by a colon) and a *value*.

The first four directives are required; no tests can be run without these settings being specified (that is, there is no default value for these settings). Once set, each setting remains in force until a new directive with the same keyword is encountered; the setting is then replaced by the new value.

Keyword	Value
`precision`	An unsigned positive integer. Its value is used to set the *precision* in the context for the following tests. If the setting exceeds the maximum precision that can be handled by an implementation then following tests should be skipped (until the setting is suitably reduced or the end of the file is reached).

`rounding`	A word which describes the *rounding* mode to set in the context for the following tests. It is case-independent and will be one of:
	`ceiling`
	`down`
	`floor`
	`half_down`
	`half_even`
	`half_up`
	`up`
	`05up`
	If an unsupported rounding mode is set, then following tests should be skipped (until the setting is changed to a supported mode or the end of the file is reached).

`maxexponent`	An unsigned integer which may be zero or positive. This value describes the value of the adjusted exponent beyond which overflow will be raised. The following tests will indicate an overflow condition (see page 16) if the adjusted exponent exceeds this setting. Implementations, in general, will have fixed maximum exponent limits, which may not match the setting in the testcase: • If the setting of `maxexponent` is larger than can be handled, then following tests should be skipped (until the setting is suitably reduced or the end of the file is reached). • If the setting of `maxexponent` is smaller than can be enforced, then following tests which indicate an overflow condition should be skipped (until the setting is suitably changed or the end of the file is reached).

Keyword	Value
`minexponent`	An unsigned integer which may be zero or negative. This value describes the value of the adjusted exponent below which underflow will be raised. The following tests will indicate an underflow condition (see page 16) if the adjusted exponent is less than this setting. (Note that if *extended* is set, smaller exponents down to *minexponent* - *precision* + 1 are possible because subnormal values are allowed.) Implementations, in general, will have fixed minimum exponent limits, which may not match the setting in the testcase: • If the setting of `minexponent` is smaller than can be handled, then following tests should be skipped (until the setting is suitably reduced or the end of the file is reached). • If the setting of `minexponent` is smaller than can be enforced, then following tests which indicate an underflow or subnormal condition should be skipped (until the setting is suitably changed or the end of the file is reached).

The next three directives are optional:

Keyword	Value
`version`	A number which describes the version of the testcases which follow. This may be up to five digits, or four digits with an embedded decimal point. For example: `version: 2.40` The meaning of the version number is not defined, except that later versions of testcases should have a larger version number.

`extended`	Either 0 or 1. This directive indicates the level of arithmetic needed for the following tests.
	When set to 1 (the default), numbers whose value is zero may have non-zero sign and exponent, operations may result in subnormal values, extra checking is performed on the length of operands, and the results of operations are defined after errors (they may be 0, infinite, or NaN values). For example:
	`extended: 1 -- enable extended values`
	`div0 divide -1 0 -> -Infinity Division_by_zero`
	When set to 0, only the X3.274 subset of the arithmetic is required, where the sign of a zero value result is always 0, subnormal values raise underflow, and some other differences are expected.
	If an implementation does not support testcases as selected by the *extended* setting then following tests should be skipped (until the extended setting is changed to an acceptable value or the end of the file is reached).

Keyword	Value
`clamp`	Either 0 or 1. This directive indicates whether explicit exponent clamping is applied. If 0 (the default), the only clamping applies to zero results, which will have maximum and minimum exponents as described under *maxexponent* and *minexponent* above. If 1, a restricted exponent range (as used in certain concrete representations) applies: the maximum exponent is reduced to *maxexponent* - *precision* + 1. This will clamp zeros at a lower value and may cause the coefficient and exponent of certain normal values to be "folded down". If an implementation does not support testcases with the clamp option as selected by the *clamp* setting then following tests should be skipped (until the clamp setting is changed to an acceptable value or the end of the file is reached).

The final directive allows testcase groups (files) to themselves be grouped together in a hierarchy:

Keyword	Value

`dectest`	A word specifying the file name (without extension) of another testcase file to be processed at this point. For example, a testcase which simply runs the testcases for the three division operations might read:
	`-- divides.decTest -- Test divisions`
	`dectest: divide`
	`dectest: divideint`
	`dectest: remainder`
	
	Note that this directive is not an "include"; the current settings are *not* inherited by the file to be processed – that file must be processed in exactly the same way as if it were the only testcase being run.

Operations

Each *test* line identifies an *operation* by means of a case-independent keyword, which is always the second token of the line. The following operations are defined.

Keyword	Definition
`abs`	If the operand is negative, this is `minus`; otherwise it is `plus`.
`add`	The two operands are added together using **add**.
`and`	The two logical operands are anded together using **and**.
`apply`	This operation applies the constraints of the directives to the operand; the result must then match in precision and value.
`canonical`	The operand is converted to an canonical encoding, if necessary (the result should be a format-dependent representation).
`class`	The class of the operand is tested; the result is one of the strings defined for **class**.
`compare`	The operands are compared using **compare**.
`comparesig`	The operands are compared using **compare-signal**.
`comparetotal`	The operands are compared using **compare-total**.
`comparetotalmag`	The operands are compared using **compare-total-magnitude**.
`copy`	The operand is copied to the result.
`copyabs`	The operand is copied to the result using **copy-abs**.
`copynegate`	The operand is copied to the result using **copy-negate**.
`copysign`	The first operand is copied to the result with the sign of the second, using **copy-sign**.
`divide`	The first operand is divided by the second, using **divide**.
`divideint`	The first operand is divided by the second to give an integer result, using **divide-integer**.
`exp`	*e* is raised to the power of the operand, using **exp**.
`fma`	The three operands are combined, using **fma**.
`invert`	The logical operand is inverted using **invert**.
`ln`, `log10`	The logarithm of the operand in base *e* or 10 is computed, using **ln** or **log10**.
`logb`	The exponent of the operand is extracted, using **logb**.
`max`, `min`	The operands are compared using **compare** and the larger or smaller, respectively, is returned.
`maxmag`, `minmag`	The magnitudes of the operands are compared using **compare** and the larger or smaller, respectively, is returned.
`minus`	The operand is subtracted from zero, using **minus**.
`multiply`	The operands are multiplied together using **multiply**.
`nextminus`	The next value less than the operand is computed using **next-minus**.
`nextplus`	The next value greater than the operand is computed using **next-plus**.
`nexttoward`	The next value to the first operand in the direction of the second is computed using **next-toward**.

Keyword	Definition
`or`	The two logical operands are ored together using **or**.
`plus`	The operand is added to zero, using **plus**.
`power`	The first operand is raised to power of the second, using **power**.
`quantize`	The first operand is quantized so that its *exponent* is set to that of the second operand, using **quantize**.
`reduce`	Trailing zeros are removed, using **reduce** (previously named *normalize*).
`remainder`	The first operand (the *dividend*) is divided by the second (the *divisor*) to give a remainder after integer division, using **remainder**.
`remaindernear`	The first operand (the *dividend*) is divided by the second (the *divisor*) to give a remainder after division to the nearest integer, using **remainder-near** (IEEE remainder).
`rescale`	The first operand is rescaled so that its *exponent* is set to the value of the second operand, using **rescale**.
`rotate`	The *coefficient* of the first operand is rotated by the number of digits given by the second operand, using **rotate**.
`samequantum`	The exponents of the operands are compared for equality.
`scaleb`	The *exponent* of the first operand is adjusted by a value given by the second operand, using **scaleb**.
`shift`	The *coefficient* of the first operand is shifted by the number of digits given by the second operand, using **shift**.
`squareroot`	The square root of the operand is computed, using **square-root**.
`subtract`	The second operand is subtracted from the first, using **subtract**.
`toEng`	The operand is converted to a string using **to-engineering-string**.
`tointegral`	The operand removes any fraction, using **round-to-integral-value**.
`tointegralx`	The operand removes any fraction, using **round-to-integral-exact**; it may result in Inexact.
`toSci`	The operand is converted to a string using **to-scientific-string**.
`trim`	Insignificant fractional zeros are removed, using **trim**.
`xor`	The two logical operands are exclusive-ored together using **xor**.

Conditions

Each *test* may cause zero or more *conditions* to be raised. The case-independent names of these conditions (if any) are listed following the *result* token of each test.

Only those conditions occuring during the tested operation are listed unless the operation is `toSci`, `toEng`, or `apply`. For these operations, conditions raised during the conversion of the operand are included (this allows the testing of conversions in both directions).

The following condition names are defined, together with the name used for it in the arithmetic specification and the IEEE 754 exception which would be raised by the condition.

Condition	Specification name	IEEE exception
`clamped`	**Clamped**	*(no equivalent)*
`conversion_syntax`	**Conversion syntax**	Invalid operation
`division_by_zero`	**Division by zero**	Division by zero
`division_impossible`	**Division impossible**	Invalid operation
`division_undefined`	**Division undefined**	Invalid operation
`inexact`	**Inexact**	Inexact
`insufficient_storage`	**Insufficient storage**	Invalid operation
`invalid_context`	**Invalid context**	Invalid operation
`invalid_operation`	**Invalid operation**	Invalid operation
`lost_digits`	**Lost digits**	*(no equivalent)*
`overflow`	**Overflow**	Overflow
`rounded`	**Rounded**	*(no equivalent)*
`subnormal`	**Subnormal**	*(no equivalent)*
`underflow`	**Underflow**	Underflow

Notes:

1. The condition names are simply the names from the arithmetic specification, with spaces changed to underscores so each forms a single token.
2. The `inexact`, `rounded`, and `subnormal` conditions are included in the testcases, even when *extended* is 0, to aid analysis and debugging. (Underflow implies all three.)

   The `rounded` condition indicates that an operand or the result of a test has had one or more zero or non-zero digits removed by rounding. That is, the number of digits in the *coefficient* of the result is fewer than in the *coefficient* of the "ideal" result.

   In contrast, the `inexact` condition indicates only that non-zero trailing digits were removed (that is, the result would compare unequal to the ideal result).
3. Similarly, the `clamped` condition, which can only occur if *extended* is 1, is included. This indicates when a zero is clamped to the maximum or minimum adjusted or representable exponent, or when a normal number is "folded-down" in order to fit in a specific encoding.
4. The `lost_digits` condition can only occur if *extended* is 0.
5. The `insufficient_storage` condition is not a predictable condition and so will not appear in any testcases. It is listed here as a reminder that some implementations could raise this condition for some tests.

Notes

This section describes the testcases included in the testcase package (`dectest.zip`), and their history.

Testcase groups and coverage

The following groups cover the base arithmetic operations of the specification:

Group	Description
`abs`	Tests the `abs` operation.
`add`	Tests the `add` operation, including both positive and negative numbers for the operands.
`compare`, `comparesig`	Test the `compare` and `comparesig` operations.
`divide`	Tests the `divide` operation.
`divideint`	Tests the `divideint` operation.
`fma`	Tests the `fma` (fused multiply-add) operation.
`max`, `maxmag`	Test the `max` and `maxmag` operations.
`min`, `minmag`	Test the `min` and `minmag` operations.
`minus`	Tests the `minus` operation.
`multiply`	Tests the `multiply` operation.
`quantize`	Tests the `quantize` operation.
`reduce`	Tests the `reduce` operation (previously named *normalize*).
`remainder`	Tests the `remainder` operation.
`remaindernear`	Tests the `remaindernear` (IEEE remainder) operation.
`rescale`	Tests the `rescale` operation.
`subtract`	Tests the `subtract` operation.
`tointegral`, `tointegralx`	Test the round-to-integral operations.

The following groups cover the mathematical functions of the specification:

Group	Description
`exp`	Tests the `exp` function.
`ln`	Tests the `ln` function.
`log10`	Tests the `log10` function.
`power`, `powersqrt`	Test the `power` operation.
`squareroot`	Tests the `squareroot` function.

The following groups cover the logical and shifting functions of the specification:

Group	Description
`and`	Tests the `and` (digit-wise logical and) operation.
`invert`	Tests the `invert` (digit-wise logical invert) operation.
`or`	Tests the `or` (digit-wise logical or) operation.
`rotate`	Tests the `rotate` (coefficient rotation) operation.
`shift`	Tests the `shift` (coefficient shifts) operation.
`xor`	Tests the `xor` (digit-wise logical xor) operation.

The following groups cover miscellaneous functions of the specification:

Group	Description
`class`	Tests the `class` (classification) operation.
`comparetotal`, `comparetotmag`	Test the total-ordering operations.
`copy`, `copyabs`, `copynegate`, `copysign`	Test the quiet copy and sign-manipulation operations.
`logb`	Tests the `logb` (exponent extract) operation.
`nextminus`, `nextplus`, `nexttoward`	Test the select-next-value operations.
`samequantum`	Tests the `samequantum` operation.
`scaleb`	Tests the `scaleb` (exponent manipulation) operation.

The following groups cover more general aspects of the operations:

Group	Description
`base`	Tests the base string conversions (`toSci` and `toEng` operations), including strings which are not valid numbers.
`inexact`	Tests edge cases for the `inexact` and `rounded` conditions, using a selection of operations.
`randoms`	4000 randomly-generated tests, using the `add`, `compare`, `divide`, `divideint`, `multiply`, `power`, `remainder`, and `subtract` operations.
`rounding`	Tests the different *rounding* modes. Each rounding mode is tested for each of the major operations.
`trim`	Tests the `trim` operation.

Three testcase groups collect together a number of testcase groups for each of the three main decimal encodings for decimal arithmetic:[5]

Group	Description
`decSingle`	Includes testcase groups for the "decimal32" decimal data type (7 digits, maximum exponent +96). The groups included in `decSingle` all have names starting with the letters `ds`, followed by a word corresponding to the operations they test.
`decDouble`	Includes testcase groups for the "decimal64" decimal data type (16 digits, maximum exponent +384). The groups included in `decDouble` all have names starting with the letters `dd`, followed by a word corresponding to the operations they test.
`decQuad`	Includes testcase groups for the "decimal128" decimal data type (34 digits, maximum exponent +6144). The groups included in `decQuad` all have names starting with the letters `dq`, followed by a word corresponding to the operations they test.

Four testcase groups are designed for testing the boundary conditions and encodings associated with the concrete representations for decimal arithmetic and a further group includes random tests for boundary conditions around 32 digits:

Group	Description
`clamp`	Tests clamped operations, independent of format.
`dsEncode`	Tests for the "decimal32" decimal data type (7 digits, maximum exponent +96); included in `decSingle`.
`ddEncode`	Tests for the "decimal64" decimal data type (16 digits, maximum exponent +384); included in `decDouble`.
`dqEncode`	Tests for the "decimal128" decimal data type (34 digits, maximum exponent +6144); included in `decQuad`.
`randombound32`	2400 tests, as in the `randoms` group, with precisions 31 through 33 and maximum exponent +9999.

All the above groups appear with the name as shown above – in these groups testcases are run with the *extended* directive set to 1; these testcases are in the file `dectest.zip`

5 See `http://speleotrove.com/decimal/decbits.html`

A subset of these groups also appears with the name as shown above with the suffix 0 – in these groups the *extended* directive is set to 0; these testcases are in the file `dectest0.zip`

This separation makes it easier to test the full or subset arithmetics separately.

The final testcase simply runs all the testcases described above:

Group	Description
`testall`	Runs all the testcases described above (over 64,000 in all).

(Again, a `testall0` group runs the tests where the *extended* directive is set to 0. This adds a further 16,300 testcases.)

Coverage of these testcases is (of course) not exhaustive. Instead, the testcases assure the basic operations of the arithmetic and concentrate on "difficult cases"; those tests where the result may not be immediately obvious, or where some implementation in the past has shown a problem.

Testcase history

The tests in the testcase groups are derived from a number of sources, and are intended to cover the paths and edge cases found in:

- Testcases and examples used by the X3 (now NCITS) J18 committee (1991+) which developed the ANSI standard X3.274-1996.[6]
- IBM VM/CMS S/370 Rexx implementation testcases (1981+)
- IBM Vienna Laboratory Rexx compiler testcases (1988+)
- NetRexx testcases (1996+)
- DiagBigDecimal – the open source testcases for the `com.ibm.math.BigDecimal` Java class (1997+)
- The decNumber reference implementation testcase library (2000+)
- New testcases, *e.g.*, for the typical concrete representations' edge cases, for extended values, operations, and logical and mathematical functions, and the random tests.

The authoritative sources for how the underlying operations should work are:

- for the subset decimal arithmetic: **ANSI X3.274-1996** (plus errata, 1997–2001)[7]
- for conversions, conditions, and rounding modes, and the precise definition of result coefficients: the **General Decimal Arithmetic Specification**.[8]
- for floating-point arithmetic, including subnormal and special values (but excluding the deviations noted in the General Decimal Arithmetic Specification): **IEEE standard 754-2008**,[9]

Please send suggestions for improvements to the testcases to the author, Mike Cowlishaw.

---

6 *American National Standard for Information Technology – Programming Language REXX, X3.274-1996*, American National Standards Institute, New York, 1996.

7 *ibid.*

8 See `http://speleotrove.com/decimal/decarith.html`

9 IEEE Std 754-2008 – *IEEE Standard for Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 2008.

Appendix A – Changes

This appendix lists changes since the first public draft of this document.

Changes in Draft 0.18 (13 July 2001)

- Two new testcase groups, `randomDouble` and `randomSingle`, have been added.
- Minor corrections and clarifications have been made.

Changes in Draft 0.20 (25 September 2001)

- The `rounding` testcase group now includes tests for the **ceiling** and **floor** modes.

Changes in Version 1.00 (21 November 2001)

The specification has been enhanced to include tests for extended value arithmetic (including subnormal values and values such as `NaN` and `Infinity`). In particular:

- A new `version` directive has been added to allow a formal version number to be indicated.
- A new `extended` directive controls the *extended-values* setting.
- Operands and results with extended values are now possible.
- A new operation, `integer` (round-to-integer) has been added.
- Three new testcase groups, `extend`, `integer`, and `randomBound32`, have been added.

Changes in Version 1.02 (30 November 2001)

- A new operation, `remaindernear` (IEEE remainder) has been added, together with a testcase group of the same name.
- Some underflow and overflow exception results and conditions have been corrected.

Changes in Version 1.03 (20 March 2002)

- Three new operations and testcase groups (`abs`, `max`, and `min`) have been added.
- Corrected the description of the `maxexponent` directive; a value of zero is permitted.

Changes in Version 2.01 (3 July 2002)

This version marks a major update of the testcases to match the new combined arithmetic specification. The underlying syntax, *etc.*, of the testcases is unchanged, but specific changes include:

- Each testcase group has been split into two groups, one with the *extended* directive set to 1 and the other with it set to 0.
- One new operation and (pair of) testcase groups (`trim`) has been added.
- The `extended` testcase group has been removed and its testcases incorporated into other groups.

Changes in Version 2.06 (1 September 2002)

This version incorporates updates to testcases for subnormal numbers (previous testcases included subnormals which were more precise than allowed by IEEE 754), and also adds testcases for rounding **to-number** conversions. The `subnormal` condition has been added, and subnormal numbers may be rounded and/or inexact (in the latter case, underflow is raised).

In this document:

- The `subnormal` condition has been added.
- The `conversion underflow` and `conversion overflow` conditions have been removed (these conditions now raise `underflow` or `overflow`, respectively).
- The operand to the `toSci` and `toEng` operations is now subject to rounding.
- Minor clarifications have been added.

Changes in Version 2.09 (8 October 2002)

- The `normalize` and `squareroot` operations and testcase groups have been added.
- Extended tests now admit unrounded long operands without input rounding; the Lost_digits condition can therefore only occur when *extended* is 0.
- Three cases in `remaindernear.decTest` which should have produced Division_impossible, but did not, have been corrected (and extra tests have been added).

Changes in Version 2.19 (21 February 2003)

- The testcase groups are now separated into two `.zip` files, one for extended operations and the other for subset arithmetic.
- The default setting for the *extended* directive is now 1.
- A new required directive, *minexponent*, has been added.
- A new condition, *clamped*, has been added.
- Hexadecimal representation notations have been added, in preparation for the new decimal32, decimal64, and decimal128 testcase groups which have replaced the old single and double format groups.

Changes in Version 2.21 (3 March 2003)

- A new directive, *clamp*, allows testing of clamped operations without requiring a specific concrete format.
- A new testcase group, `clamp`, has been added.
- A new operator, `apply` has been added. This works like `toSci` and `toEng` except that the result does not have to be in one of the strict canonical formats.

Changes in Version 2.25 (13 June 2003)

- The `quantize` operation and testcase group have been added.

Changes in Version 2.27 (23 July 2003)

- The `integer` (round-to-integer) operation has been replaced by the `tointegral` (round-to-integral-value) operation. The latter implements the IEEE 754-2008 operation (no Inexact flag, *etc.*).

Changes in Version 2.31 (29 August 2003)

- The `samequantum` operation and testcase group have been added.

Changes in Version 2.35 (27 November 2005)

- The `exp`, `ln`, and `log10` operations and testcase groups have been added.
- The `power` testcase group has been greatly extended, to cover non-integer second operands, and the `powersqrt` testcase group has been added.

Changes in Version 2.36 (28 April 2006)

- The `comparetotal` operation and testcase group has been added; this operation compares two numbers using IEEE 754-2008 total ordering.

Changes in Version 2.38 (16 March 2007)

- Testcases may now have up to three operands (for `fma`).
- The following operations and testcase groups have been added: `and`, `canonical`, `class`, `comparesig`, `comparetotmag`, `copy`, `copyabs`, `copynegate`, `copysign`, `fma`, `invert`, `logb`, `maxmag`, `minmag`, `nextminus`, `nextplus`, `nexttoward`, `or`, `rotate`, `scaleb`, `shift`, `tointegralx`, `xor`.

Changes in Version 2.40 (22 May 2007)

- Testcase groups that test only using the precisions and ranges of the three decimal formats in the IEEE 754-2008 standard have been added. These are grouped under the names `decSingle`, `decDouble`, and `decQuad`, where each includes further groups whose names are prefixed with `ds`, `dd`, and `dq` respectively. There are nearly 30,000 testcases in these new groups.

- The `decimal32`, `decimal64`, and `decimal128` groups have been renamed to `dsEncode`, `ddEncode` and `dqEncode` respectively, to match the naming scheme for the other fixed-size testcases.
- A new rounding mode has been added: `05up` ("round for re-reound", where only 0 and 5 might be rounded up; other last-place digits round down).
- The `normalize` operation and testcase group has been renamed `reduce` to avoid confusion.

Changes in Version 2.43 (29 Jul 2008)

References to IEEE 854 and the old IEEE 754 standard have been removed and/or changed to refer to IEEE 754-2008, and specific URLs for the General Decimal Arithmetic site have been removed.

Also, all references to the General Decimal Arithmetic website have been updated to `http://speleotrove.com/decimal` (its new location).

Changes in Version 2.44 (24 Mar 2009)

The document is now formatted using OpenOffice (generated from GML), for improved PDF files with bookmarks, hot links, *etc.* There are no technical changes.

Index