NAME

Math::BigInt - Arbitrary size integer/float math package

SYNOPSIS

```
use Math::BigInt;

# or make it faster with huge numbers: install (optional)
# Math::BigInt::GMP and always use (it will fall back to
# pure Perl if the GMP library is not installed):
# (See also the L<MATH LIBRARY> section!)

# will warn if Math::BigInt::GMP cannot be found
use Math::BigInt lib => 'GMP';

# to suppress the warning use this:
# use Math::BigInt try => 'GMP';

# dies if GMP cannot be loaded:
# use Math::BigInt only => 'GMP';

my $str = '1234567890';
my @values = (64,74,18);
my $n = 1; my $sign = '-';

# Number creation
my $x = Math::BigInt->new($str); # defaults to 0
my $y = $x->copy(); # make a true copy
my $nan = Math::BigInt->bnan(); # create a NotANumber
my $zero = Math::BigInt->bzero(); # create a +0
my $inf = Math::BigInt->binf(); # create a +inf
my $inf = Math::BigInt->binf('-'); # create a -inf
my $one = Math::BigInt->bone(); # create a +1
my $mone = Math::BigInt->bone('-'); # create a -1

my $pi = Math::BigInt->bpi(); # returns '3'
# see Math::BigFloat::bpi()

$h = Math::BigInt->new('0x123'); # from hexadecimal
$b = Math::BigInt->new('0b101'); # from binary
$o = Math::BigInt->from_oct('0101'); # from octal

# Testing (don't modify their arguments)
# (return true if the condition is met, otherwise false)

$x->is_zero(); # if $x is +0
$x->is_nan(); # if $x is NaN
$x->is_one(); # if $x is +1
$x->is_one('-'); # if $x is -1
$x->is_odd(); # if $x is odd
$x->is_even(); # if $x is even
$x->is_pos(); # if $x > 0
$x->is_neg(); # if $x < 0
$x->is_inf($sign); # if $x is +inf, or -inf (sign is default '+')
$x->is_int(); # if $x is an integer (not a float)
```

```
# comparing and digit/sign extraction
$x->bcmp($y); # compare numbers (undef,<0,=0,>0)
$x->bacmp($y); # compare absolutely (undef,<0,=0,>0)
$x->sign(); # return the sign, either +,- or NaN
$x->digit($n); # return the nth digit, counting from right
$x->digit(-$n); # return the nth digit, counting from left

# The following all modify their first argument. If you want to pre-
# serve $x, use $z = $x->copy()->bXXX($y); See under L<CAVEATS> for
# why this is necessary when mixing $a = $b assignments with non-over-
# loaded math.

$x->bzero(); # set $x to 0
$x->bnan(); # set $x to NaN
$x->bone(); # set $x to +1
$x->bone('-'); # set $x to -1
$x->binf(); # set $x to inf
$x->binf('-'); # set $x to -inf

$x->bneg(); # negation
$x->babs(); # absolute value
$x->bsgn(); # sign function (-1, 0, 1, or NaN)
$x->bnorm(); # normalize (no-op in BigInt)
$x->bnot(); # two's complement (bit wise not)
$x->binc(); # increment $x by 1
$x->bdec(); # decrement $x by 1

$x->badd($y); # addition (add $y to $x)
$x->bsub($y); # subtraction (subtract $y from $x)
$x->bmul($y); # multiplication (multiply $x by $y)
$x->bdiv($y); # divide, set $x to quotient
# return (quo,rem) or quo if scalar

$x->bmuladd($y,$z); # $x = $x * $y + $z

$x->bmod($y); # modulus (x % y)
$x->bmodpow($y,$mod); # modular exponentiation (($x ** $y) % $mod)
$x->bmodinv($mod); # modular multiplicative inverse
$x->bpow($y); # power of arguments (x ** y)
$x->blsft($y); # left shift in base 2
$x->brsft($y); # right shift in base 2
# returns (quo,rem) or quo if in sca-
# lar context
$x->blsft($y,$n); # left shift by $y places in base $n
$x->brsft($y,$n); # right shift by $y places in base $n
# returns (quo,rem) or quo if in sca-
# lar context

$x->band($y); # bitwise and
$x->bior($y); # bitwise inclusive or
$x->bxor($y); # bitwise exclusive or
$x->bnot(); # bitwise not (two's complement)

$x->bsqrt(); # calculate square-root
```

```
$x->broot($y); # $y'th root of $x (e.g. $y == 3 => cubic root)
$x->bfac(); # factorial of $x (1*2*3*4*..$x)

$x->bnok($y); # x over y (binomial coefficient n over k)

$x->blog(); # logarithm of $x to base e (Euler's number)
$x->blog($base); # logarithm of $x to base $base (f.i. 2)
$x->bexp(); # calculate e ** $x where e is Euler's number

$x->round($A,$P,$mode); # round to accuracy or precision using
# mode $mode
$x->bround($n); # accuracy: preserve $n digits
$x->bfround($n); # $n > 0: round $nth digits,
# $n < 0: round to the $nth digit after the
# dot, no-op for BigInts

# The following do not modify their arguments in BigInt (are no-ops),
# but do so in BigFloat:

$x->bfloor(); # round towards minus infinity
$x->bceil(); # round towards plus infinity
$x->bint(); # round towards zero

# The following do not modify their arguments:

# greatest common divisor (no OO style)
my $gcd = Math::BigInt::bgcd(@values);
# lowest common multiple (no OO style)
my $lcm = Math::BigInt::blcm(@values);

$x->length(); # return number of digits in number
($xl,$f) = $x->length(); # length of number and length of fraction
# part, latter is always 0 digits long
# for BigInts

$x->exponent(); # return exponent as BigInt
$x->mantissa(); # return (signed) mantissa as BigInt
$x->parts(); # return (mantissa,exponent) as BigInt
$x->copy(); # make a true copy of $x (unlike $y = $x;)
$x->as_int(); # return as BigInt (in BigInt: same as copy())
$x->numify(); # return as scalar (might overflow!)

# conversion to string (do not modify their argument)
$x->bstr(); # normalized string (e.g. '3')
$x->bsstr(); # norm. string in scientific notation (e.g. '3E0')
$x->as_hex(); # as signed hexadecimal string with prefixed 0x
$x->as_bin(); # as signed binary string with prefixed 0b
$x->as_oct(); # as signed octal string with prefixed 0


# precision and accuracy (see section about rounding for more)
$x->precision(); # return P of $x (or global, if P of $x undef)
$x->precision($n); # set P of $x to $n
$x->accuracy(); # return A of $x (or global, if A of $x undef)
```

```
$x->accuracy($n); # set A $x to $n

# Global methods
Math::BigInt->precision(); # get/set global P for all BigInt objects
Math::BigInt->accuracy(); # get/set global A for all BigInt objects
Math::BigInt->round_mode(); # get/set global round mode, one of
# 'even', 'odd', '+inf', '-inf', 'zero',
# 'trunc' or 'common'
Math::BigInt->config(); # return hash containing configuration
```

DESCRIPTION

All operators (including basic math operations) are overloaded if you declare your big integers as

```
$i = new Math::BigInt '123_456_789_123_456_789';
```

Operations with overloaded operators preserve the arguments which is exactly what you expect.

Input

Input values to these routines may be any string, that looks like a number and results in an integer, including hexadecimal and binary numbers.

Scalars holding numbers may also be passed, but note that non-integer numbers may already have lost precision due to the conversion to float. Quote your input if you want BigInt to see all the digits:

```
$x = Math::BigInt->new(12345678890123456789); # bad
$x = Math::BigInt->new('12345678901234567890'); # good
```

You can include one underscore between any two digits.

This means integer values like 1.01E2 or even 1000E-2 are also accepted. Non-integer values result in NaN.

Hexadecimal (prefixed with "0x") and binary numbers (prefixed with "0b") are accepted, too. Please note that octal numbers are not recognized by *new()*, so the following will print "123":

```
perl -MMath::BigInt -le 'print Math::BigInt->new("0123")'
```

To convert an octal number, use *from_oct()*;

```
perl -MMath::BigInt -le 'print Math::BigInt->from_oct("0123")'
```

Currently, *Math::BigInt::new()* defaults to 0, while Math::BigInt::new('') results in 'NaN'. This might change in the future, so use always the following explicit forms to get a zero or NaN:

```
$zero = Math::BigInt->bzero();
$nan = Math::BigInt->bnan();
```

`bnorm()` on a BigInt object is now effectively a no-op, since the numbers are always stored in normalized form. If passed a string, creates a BigInt object from the input.

Output

Output values are BigInt objects (normalized), except for the methods which return a string (see "SYNOPSIS").

Some routines (`is_odd()`, `is_even()`, `is_zero()`, `is_one()`, `is_nan()`, etc.) return true or false, while others (`bcmp()`, `bacmp()`) return either undef (if NaN is involved), <0, 0 or >0 and are suited for sort.

METHODS

Each of the methods below (except *config()*, *accuracy()* and *precision()*) accepts three additional parameters. These arguments `$A`, `$P` and `$R` are `accuracy`, `precision` and `round_mode`. Please see the section about "ACCURACY and PRECISION" for more information.

config()

```
use Data::Dumper;

print Dumper ( Math::BigInt->config() );
print Math::BigInt->config()->{lib},"\n";
```

Returns a hash containing the configuration, e.g. the version number, lib loaded etc. The following hash keys are currently filled in with the appropriate information.

```
key Description
Example
============================================================
lib Name of the low-level math library
Math::BigInt::Calc
lib_version Version of low-level math library (see 'lib')
0.30
class The class name of config() you just called
Math::BigInt
upgrade To which class math operations might be
upgraded Math::BigFloat
downgrade To which class math operations might be
downgraded undef
precision Global precision
undef
accuracy Global accuracy
undef
round_mode Global round mode
even
version version number of the class you used
1.61
div_scale Fallback accuracy for div
40
trap_nan If true, traps creation of NaN via croak()
1
trap_inf If true, traps creation of +inf/-inf via croak()
1
```

The following values can be set by passing `config()` a reference to a hash:

```
trap_inf trap_nan
upgrade downgrade precision accuracy round_mode div_scale
```

Example:

```
$new_cfg = Math::BigInt->config(
{ trap_inf => 1, precision => 5 }
);
```

accuracy()

```
$x->accuracy(5); # local for $x
CLASS->accuracy(5); # global for all members of CLASS
# Note: This also applies to new()!

$A = $x->accuracy(); # read out accuracy that affects $x
$A = CLASS->accuracy(); # read out global accuracy
```

Set or get the global or local accuracy, aka how many significant digits the results have. If you set a global accuracy, then this also applies to *new()*!

Warning! The accuracy *sticks*, e.g. once you created a number under the influence of `CLASS->accuracy($A)`, all results from math operations with that number will also be rounded.

In most cases, you should probably round the results explicitly using one of "*round()*", "*bround()*" or "*bfround()*" or by passing the desired accuracy to the math operation as additional parameter:

```
my $x = Math::BigInt->new(30000);
my $y = Math::BigInt->new(7);
print scalar $x->copy()->bdiv($y, 2); # print 4300
print scalar $x->copy()->bdiv($y)->bround(2); # print 4300
```

Please see the section about "ACCURACY and PRECISION" for further details.

Value must be greater than zero. Pass an undef value to disable it:

```
$x->accuracy(undef);
Math::BigInt->accuracy(undef);
```

Returns the current accuracy. For `$x->accuracy()` it will return either the local accuracy, or if not defined, the global. This means the return value represents the accuracy that will be in effect for `$x`:

```
$y = Math::BigInt->new(1234567); # unrounded
print Math::BigInt->accuracy(4),"\n"; # set 4, print 4
$x = Math::BigInt->new(123456); # $x will be automatic-
# ally rounded!
print "$x $y\n"; # '123500 1234567'
print $x->accuracy(),"\n"; # will be 4
print $y->accuracy(),"\n"; # also 4, since
# global is 4
print Math::BigInt->accuracy(5),"\n"; # set to 5, print 5
print $x->accuracy(),"\n"; # still 4
print $y->accuracy(),"\n"; # 5, since global is 5
```

Note: Works also for subclasses like Math::BigFloat. Each class has it's own globals separated from Math::BigInt, but it is possible to subclass Math::BigInt and make the globals of the subclass aliases to the ones from Math::BigInt.

precision()

```
$x->precision(-2); # local for $x, round at the second
# digit right of the dot
$x->precision(2); # ditto, round at the second digit
# left of the dot

CLASS->precision(5); # Global for all members of CLASS
# This also applies to new()!
CLASS->precision(-5); # ditto

$P = CLASS->precision(); # read out global precision
$P = $x->precision(); # read out precision that affects $x
```

Note: You probably want to use "*accuracy()*" instead. With "*accuracy()*" you set the number of digits each result should have, with "*precision()*" you set the place where to round!

`precision()` sets or gets the global or local precision, aka at which digit before or after the dot to round all results. A set global precision also applies to all newly created numbers!

In Math::BigInt, passing a negative number precision has no effect since no numbers have digits after the dot. In Math::BigFloat, it will round all results to P digits after the dot.

Please see the section about ''ACCURACY and PRECISION'' for further details.

Pass an undef value to disable it:

```
$x->precision(undef);
Math::BigInt->precision(undef);
```

Returns the current precision. For `$x->precision()` it will return either the local precision of `$x`, or if not defined, the global. This means the return value represents the prevision that will be in effect for `$x`:

```
$y = Math::BigInt->new(1234567); # unrounded
print Math::BigInt->precision(4),"\n"; # set 4, print 4
$x = Math::BigInt->new(123456); # will be automatically rounded
print $x; # print "120000"!
```

Note: Works also for subclasses like Math::BigFloat. Each class has its own globals separated from Math::BigInt, but it is possible to subclass Math::BigInt and make the globals of the subclass aliases to the ones from Math::BigInt.

brsft()

```
$x->brsft($y,$n);
```

Shifts `$x` right by `$y` in base `$n`. Default is base 2, used are usually 10 and 2, but others work, too.

Right shifting usually amounts to dividing `$x` by `$n` ** `$y` and truncating the result:

```
$x = Math::BigInt->new(10);
$x->brsft(1); # same as $x >> 1: 5
$x = Math::BigInt->new(1234);
$x->brsft(2,10); # result 12
```

There is one exception, and that is base 2 with negative `$x`:

```
$x = Math::BigInt->new(-5);
print $x->brsft(1);
```

This will print -3, not -2 (as it would if you divide -5 by 2 and truncate the result).

new()

```
$x = Math::BigInt->new($str,$A,$P,$R);
```

Creates a new BigInt object from a scalar or another BigInt object. The input is accepted as decimal, hex (with leading '0x') or binary (with leading '0b').

See ''Input'' for more info on accepted input formats.

from_oct()

```
$x = Math::BigInt->from_oct("0775"); # input is octal
```

Interpret the input as an octal string and return the corresponding value. A ''0'' (zero) prefix is optional. A single underscore character may be placed right after the prefix, if present, or between any two digits. If the input is invalid, a NaN is returned.

from_hex()

```
$x = Math::BigInt->from_hex("0xcafe"); # input is hexadecimal
```

Interpret input as a hexadecimal string. A ''0x'' or ''x'' prefix is optional. A single underscore character may be placed right after the prefix, if present, or between any two digits. If the input is invalid, a NaN is returned.

from_bin()

```
$x = Math::BigInt->from_bin("0b10011"); # input is binary
```

Interpret the input as a binary string. A ‘‘0b’’ or ‘‘b’’ prefix is optional. A single underscore character may be placed right after the prefix, if present, or between any two digits. If the input is invalid, a NaN is returned.

bnan()

```
$x = Math::BigInt->bnan();
```

Creates a new BigInt object representing NaN (Not A Number). If used on an object, it will set it to NaN:

```
$x->bnan();
```

bzero()

```
$x = Math::BigInt->bzero();
```

Creates a new BigInt object representing zero. If used on an object, it will set it to zero:

```
$x->bzero();
```

binf()

```
$x = Math::BigInt->binf($sign);
```

Creates a new BigInt object representing infinity. The optional argument is either '-' or '+', indicating whether you want infinity or minus infinity. If used on an object, it will set it to infinity:

```
$x->binf();
$x->binf('-');
```

bone()

```
$x = Math::BigInt->binf($sign);
```

Creates a new BigInt object representing one. The optional argument is either '-' or '+', indicating whether you want one or minus one. If used on an object, it will set it to one:

```
$x->bone(); # +1
$x->bone('-'); # -1
```

is_one()/is_zero()/is_nan()/is_inf()

```
$x->is_zero(); # true if arg is +0
$x->is_nan(); # true if arg is NaN
$x->is_one(); # true if arg is +1
$x->is_one('-'); # true if arg is -1
$x->is_inf(); # true if +inf
$x->is_inf('-'); # true if -inf (sign is default '+')
```

These methods all test the BigInt for being one specific value and return true or false depending on the input. These are faster than doing something like:

```
if ($x == 0)
```

is_pos()/is_neg()/is_positive()/is_negative()

```
$x->is_pos(); # true if > 0
$x->is_neg(); # true if < 0
```

The methods return true if the argument is positive or negative, respectively. `NaN` is neither positive nor negative, while `+inf` counts as positive, and `-inf` is negative. A `zero` is neither positive nor negative.

These methods are only testing the sign, and not the value.

`is_positive()` and `is_negative()` are aliases to `is_pos()` and `is_neg()`, respectively. `is_positive()` and `is_negative()` were introduced in v1.36, while `is_pos()` and `is_neg()` were only introduced in v1.68.

is_odd()/is_even()/is_int()

```
$x->is_odd(); # true if odd, false for even
$x->is_even(); # true if even, false for odd
$x->is_int(); # true if $x is an integer
```

The return true when the argument satisfies the condition. `NaN`, `+inf`, `-inf` are not integers and are neither odd nor even.

In BigInt, all numbers except `NaN`, `+inf` and `-inf` are integers.

bcmp()

```
$x->bcmp($y);
```

Compares `$x` with `$y` and takes the sign into account. Returns -1, 0, 1 or undef.

bacmp()

```
$x->bacmp($y);
```

Compares `$x` with `$y` while ignoring their sign. Returns -1, 0, 1 or undef.

sign()

```
$x->sign();
```

Return the sign, of `$x`, meaning either `+`, `-`, `-inf`, `+inf` or NaN.

If you want `$x` to have a certain sign, use one of the following methods:

```
$x->babs(); # '+'
$x->babs()->bneg(); # '-'
$x->bnan(); # 'NaN'
$x->binf(); # '+inf'
$x->binf('-'); # '-inf'
```

digit()

```
$x->digit($n); # return the nth digit, counting from right
```

If `$n` is negative, returns the digit counting from left.

bneg()

```
$x->bneg();
```

Negate the number, e.g. change the sign between '+' and '-', or between '+inf' and '-inf', respectively. Does nothing for NaN or zero.

babs()

```
$x->babs();
```

Set the number to its absolute value, e.g. change the sign from '-' to '+' and from '-inf' to '+inf', respectively. Does nothing for NaN or positive numbers.

bsgn()

```
$x->bsgn();
```

Signum function. Set the number to -1, 0, or 1, depending on whether the number is negative, zero, or positive, respectively. Does not modify NaNs.

bnorm()

```
$x->bnorm(); # normalize (no-op)
```

bnot()

```
$x->bnot();
```

Two's complement (bitwise not). This is equivalent to

```
$x->binc()->bneg();
```

but faster.

binc()

```
$x->binc(); # increment x by 1
```

bdec()

```
$x->bdec(); # decrement x by 1
```

badd()

```
$x->badd($y); # addition (add $y to $x)
```

bsub()

```
$x->bsub($y); # subtraction (subtract $y from $x)
```

bmul()

```
$x->bmul($y); # multiplication (multiply $x by $y)
```

bmuladd()

```
$x->bmuladd($y,$z);
```

Multiply `$x` by `$y`, and then add `$z` to the result,

This method was added in v1.87 of Math::BigInt (June 2007).

bdiv()

```
$x->bdiv($y); # divide, set $x to quotient
# return (quo,rem) or quo if scalar
```

bmod()

```
$x->bmod($y); # modulus (x % y)
```

bmodinv()

```
$x->bmodinv($mod); # modular multiplicative inverse
```

Returns the multiplicative inverse of `$x` modulo `$mod`. If

```
$y = $x -> copy() -> bmodinv($mod)
```

then `$y` is the number closest to zero, and with the same sign as `$mod`, satisfying

```
($x * $y) % $mod = 1 % $mod
```

If `$x` and `$y` are non-zero, they must be relative primes, i.e., `bgcd($y, $mod)==1`. 'NaN' is returned when no modular multiplicative inverse exists.

bmodpow()

```
$num->bmodpow($exp,$mod); # modular exponentiation
# ($num**$exp % $mod)
```

Returns the value of `$num` taken to the power `$exp` in the modulus `$mod` using binary exponentiation. `bmodpow` is far superior to writing

```
$num ** $exp % $mod
```

because it is much faster - it reduces internal variables into the modulus whenever possible, so it operates on smaller numbers.

`bmodpow` also supports negative exponents.

```
bmodpow($num, -1, $mod)
```

is exactly equivalent to

```
bmodinv($num, $mod)
```

bpow()

```
$x->bpow($y); # power of arguments (x ** y)
```

blog()

```
$x->blog($base, $accuracy); # logarithm of x to the base $base
```

If `$base` is not defined, Euler's number (e) is used:

```
print $x->blog(undef, 100); # log(x) to 100 digits
```

bexp()

```
$x->bexp($accuracy); # calculate e ** X
```

Calculates the expression `e ** $x` where `e` is Euler's number.

This method was added in v1.82 of Math::BigInt (April 2007).

See also "*blog()*".

bnok()

```
$x->bnok($y); # x over y (binomial coefficient n over k)
```

Calculates the binomial coefficient n over k, also called the "choose" function. The result is equivalent to:

```
( n ) n!
| - | = -------
( k ) k!(n-k)!
```

This method was added in v1.84 of Math::BigInt (April 2007).

bpi()

```
print Math::BigInt->bpi(100), "\n"; # 3
```

Returns PI truncated to an integer, with the argument being ignored. This means under BigInt this always returns `3`.

If upgrading is in effect, returns PI, rounded to N digits with the current rounding mode:

```
use Math::BigFloat;
use Math::BigInt upgrade => Math::BigFloat;
print Math::BigInt->bpi(3), "\n"; # 3.14
print Math::BigInt->bpi(100), "\n"; # 3.1415....
```

This method was added in v1.87 of Math::BigInt (June 2007).

bcos()

```
my $x = Math::BigInt->new(1);
print $x->bcos(100), "\n";
```

Calculate the cosinus of `$x`, modifying `$x` in place.

In BigInt, unless upgrading is in effect, the result is truncated to an integer.

This method was added in v1.87 of Math::BigInt (June 2007).

bsin()

```
my $x = Math::BigInt->new(1);
print $x->bsin(100), "\n";
```

Calculate the sinus of `$x`, modifying `$x` in place.

In BigInt, unless upgrading is in effect, the result is truncated to an integer.

This method was added in v1.87 of Math::BigInt (June 2007).

batan2()

```
my $x = Math::BigInt->new(1);
my $y = Math::BigInt->new(1);
print $y->batan2($x), "\n";
```

Calculate the arcus tangens of `$y` divided by `$x`, modifying `$y` in place.

In BigInt, unless upgrading is in effect, the result is truncated to an integer.

This method was added in v1.87 of Math::BigInt (June 2007).

batan()

```
my $x = Math::BigFloat->new(0.5);
print $x->batan(100), "\n";
```

Calculate the arcus tangens of `$x`, modifying `$x` in place.

In BigInt, unless upgrading is in effect, the result is truncated to an integer.

This method was added in v1.87 of Math::BigInt (June 2007).

blsft()

```
$x->blsft($y); # left shift in base 2
$x->blsft($y,$n); # left shift, in base $n (like 10)
```

brsft()

```
$x->brsft($y); # right shift in base 2
$x->brsft($y,$n); # right shift, in base $n (like 10)
```

band()

```
$x->band($y); # bitwise and
```

bior()

```
$x->bior($y); # bitwise inclusive or
```

bxor()

```
$x->bxor($y); # bitwise exclusive or
```

bnot()

```
$x->bnot(); # bitwise not (two's complement)
```

bsqrt()

```
$x->bsqrt(); # calculate square-root
```

broot()

```
$x->broot($N);
```

Calculates the N'th root of `$x`.

bfac()

```
$x->bfac(); # factorial of $x (1*2*3*4*..$x)
```

round()

```
$x->round($A,$P,$round_mode);
```

Round `$x` to accuracy `$A` or precision `$P` using the round mode `$round_mode`.

bround()

```
$x->bround($N); # accuracy: preserve $N digits
```

bfround()

```
$x->bfround($N);
```

If N is > 0, rounds to the Nth digit from the left. If N < 0, rounds to the Nth digit after the dot. Since BigInts are integers, the case N < 0 is a no-op for them.

Examples:

```
Input N Result
================================================
123456.123456 3 123500
123456.123456 2 123450
123456.123456 -2 123456.12
123456.123456 -3 123456.123
```

bfloor()

```
$x->bfloor();
```

Round `$x` towards minus infinity (i.e., set `$x` to the largest integer less than or equal to `$x`). This is a no-op in BigInt, but changes `$x` in BigFloat, if `$x` is not an integer.

bceil()

```
$x->bceil();
```

Round `$x` towards plus infinity (i.e., set `$x` to the smallest integer greater than or equal to `$x`). This is a no-op in BigInt, but changes `$x` in BigFloat, if `$x` is not an integer.

bint()

```
$x->bint();
```

Round `$x` towards zero. This is a no-op in BigInt, but changes `$x` in BigFloat, if `$x` is not an integer.

bgcd()

```
bgcd(@values); # greatest common divisor (no OO style)
```

blcm()

```
blcm(@values); # lowest common multiple (no OO style)
```

length()

```
$x->length();
($xl,$fl) = $x->length();
```

Returns the number of digits in the decimal representation of the number. In list context, returns the length of the integer and fraction part. For BigInt's, the length of the fraction part will always be 0.

exponent()

```
$x->exponent();
```

Return the exponent of `$x` as BigInt.

mantissa()

```
$x->mantissa();
```

Return the signed mantissa of `$x` as BigInt.

parts()

```
$x->parts(); # return (mantissa,exponent) as BigInt
```

copy()

```
$x->copy(); # make a true copy of $x (unlike $y = $x;)
```

as_int()/as_number()

```
$x->as_int();
```

Returns `$x` as a BigInt (truncated towards zero). In BigInt this is the same as `copy()`.

`as_number()` is an alias to this method. `as_number` was introduced in v1.22, while `as_int()` was only introduced in v1.68.

bstr()

```
$x->bstr();
```

Returns a normalized string representation of `$x`.

bsstr()

```
$x->bsstr(); # normalized string in scientific notation
```

as_hex()

```
$x->as_hex(); # as signed hexadecimal string with prefixed 0x
```

as_bin()

```
$x->as_bin(); # as signed binary string with prefixed 0b
```

as_oct()

```
$x->as_oct(); # as signed octal string with prefixed 0
```

numify()

```
print $x->numify();
```

This returns a normal Perl scalar from `$x`. It is used automatically whenever a scalar is needed, for instance in array index operations.

This loses precision, to avoid this use *as_int()* instead.

modify()

```
$x->modify('bpowd');
```

This method returns 0 if the object can be modified with the given operation, or 1 if not.

This is used for instance by Math::BigInt::Constant.

upgrade()/*downgrade()*

Set/get the class for downgrade/upgrade operations. Thuis is used for instance by bignum. The defaults are '', thus the following operation will create a BigInt, not a BigFloat:

```
my $i = Math::BigInt->new(123);
my $f = Math::BigFloat->new('123.1');

print $i + $f,"\n"; # print 246
```

div_scale()

Set/get the number of digits for the default precision in divide operations.

round_mode()

Set/get the current round mode.

ACCURACY and PRECISION

Since version v1.33, Math::BigInt and Math::BigFloat have full support for accuracy and precision based rounding, both automatically after every operation, as well as manually.

This section describes the accuracy/precision handling in Math::Big* as it used to be and as it is now, complete with an explanation of all terms and abbreviations.

Not yet implemented things (but with correct description) are marked with '!', things that need to be answered are marked with '?'.

In the next paragraph follows a short description of terms used here (because these may differ from terms used by others people or documentation).

During the rest of this document, the shortcuts A (for accuracy), P (for precision), F (fallback) and R (rounding mode) will be used.

Precision P

A fixed number of digits before (positive) or after (negative) the decimal point. For example, 123.45 has a precision of -2. 0 means an integer like 123 (or 120). A precision of 2 means two digits to the left of the decimal point are zero, so 123 with P = 1 becomes 120. Note that numbers with zeros before the decimal point may have different precisions, because 1200 can have p = 0, 1 or 2 (depending on what the initial value was). It could also have p < 0, when the digits

after the decimal point are zero.

The string output (of floating point numbers) will be padded with zeros:

```
Initial value P A Result String
------------------------------------------------------------
1234.01 -3 1000 1000
1234 -2 1200 1200
1234.5 -1 1230 1230
1234.001 1 1234 1234.0
1234.01 0 1234 1234
1234.01 2 1234.01 1234.01
1234.01 5 1234.01 1234.01000
```

For BigInts, no padding occurs.

Accuracy A

Number of significant digits. Leading zeros are not counted. A number may have an accuracy greater than the non-zero digits when there are zeros in it or trailing zeros. For example, 123.456 has A of 6, 10203 has 5, 123.0506 has 7, 123.450000 has 8 and 0.000123 has 3.

The string output (of floating point numbers) will be padded with zeros:

```
Initial value P A Result String
------------------------------------------------------------
1234.01 3 1230 1230
1234.01 6 1234.01 1234.01
1234.1 8 1234.1 1234.1000
```

For BigInts, no padding occurs.

Fallback F

When both A and P are undefined, this is used as a fallback accuracy when dividing numbers.

Rounding mode R

When rounding a number, different 'styles' or 'kinds' of rounding are possible. (Note that random rounding, as in Math::Round, is not implemented.)

'trunc'

truncation invariably removes all digits following the rounding place, replacing them with zeros. Thus, 987.65 rounded to tens (P=1) becomes 980, and rounded to the fourth sigdig becomes 987.6 (A=4). 123.456 rounded to the second place after the decimal point (P=-2) becomes 123.46.

All other implemented styles of rounding attempt to round to the ''nearest digit.'' If the digit D immediately to the right of the rounding place (skipping the decimal point) is greater than 5, the number is incremented at the rounding place (possibly causing a cascade of incrementation): e.g. when rounding to units, 0.9 rounds to 1, and -19.9 rounds to -20. If D < 5, the number is similarly truncated at the rounding place: e.g. when rounding to units, 0.4 rounds to 0, and -19.4 rounds to -19.

However the results of other styles of rounding differ if the digit immediately to the right of the rounding place (skipping the decimal point) is 5 and if there are no digits, or no digits other than 0, after that 5. In such cases:

'even'

rounds the digit at the rounding place to 0, 2, 4, 6, or 8 if it is not already. E.g., when rounding to the first sigdig, 0.45 becomes 0.4, -0.55 becomes -0.6, but 0.4501 becomes 0.5.

'odd'

rounds the digit at the rounding place to 1, 3, 5, 7, or 9 if it is not already. E.g., when rounding to the first sigdig, 0.45 becomes 0.5, -0.55 becomes -0.5, but 0.5501 becomes 0.6.

'+inf'
: round to plus infinity, i.e. always round up. E.g., when rounding to the first sigdig, 0.45 becomes 0.5, -0.55 becomes -0.5, and 0.4501 also becomes 0.5.

'-inf'
: round to minus infinity, i.e. always round down. E.g., when rounding to the first sigdig, 0.45 becomes 0.4, -0.55 becomes -0.6, but 0.4501 becomes 0.5.

'zero'
: round to zero, i.e. positive numbers down, negative ones up. E.g., when rounding to the first sigdig, 0.45 becomes 0.4, -0.55 becomes -0.5, but 0.4501 becomes 0.5.

'common'
: round up if the digit immediately to the right of the rounding place is 5 or greater, otherwise round down. E.g., 0.15 becomes 0.2 and 0.149 becomes 0.1.

The handling of A & P in MBI/MBF (the old core code shipped with Perl versions <= 5.7.2) is like this:

Precision
: * ffround($p) is able to round to $p number of digits after the decimal point
* otherwise P is unused

Accuracy (significant digits)
: * fround($a) rounds to $a significant digits
* only fdiv() and fsqrt() take A as (optional) parameter
+ other operations simply create the same number (fneg etc), or more (fmul) of digits
+ rounding/truncating is only done when explicitly calling one of fround or ffround, and never for BigInt (not implemented)
* fsqrt() simply hands its accuracy argument over to fdiv.
* the documentation and the comment in the code indicate two different ways on how fdiv() determines the maximum number of digits it should calculate, and the actual code does yet another thing
POD:
max($Math::BigFloat::div_scale,length(dividend)+length(divisor))
Comment:
result has at most max(scale, length(dividend), length(divisor)) digits
Actual code:
scale = max(scale, length(dividend)-1,length(divisor)-1);
scale += length(divisor) - length(dividend);
So for lx = 3, ly = 9, scale = 10, scale will actually be 16 (10
So for lx = 3, ly = 9, scale = 10, scale will actually be 16 (10+9-3). Actually, the 'difference' added to the scale is calculated from the number of "significant digits" in dividend and divisor, which is derived by looking at the length of the mantissa. Which is wrong, since it includes the + sign (oops) and actually gets 2 for '+100' and 4 for '+101'. Oops again. Thus 124/3 with div_scale=1 will get you '41.3' based on the strange assumption that 124 has 3 significant digits, while 120/7 will get you '17', not '17.1' since 120 is thought to have 2 significant digits. The rounding after the division then uses the remainder and $y to determine whether it must round up or down.
? I have no idea which is the right way. That's why I used a slightly more
? simple scheme and tweaked the few failing testcases to match it.

This is how it works now:

Setting/Accessing

```
* You can set the A global via Math::BigInt->accuracy() or
Math::BigFloat->accuracy() or whatever class you are using.
* You can also set P globally by using Math::SomeClass->precision()
likewise.
* Globals are classwide, and not inherited by subclasses.
* to undefine A, use Math::SomeCLass->accuracy(undef);
* to undefine P, use Math::SomeClass->precision(undef);
* Setting Math::SomeClass->accuracy() clears automatically
Math::SomeClass->precision(), and vice versa.
* To be valid, A must be > 0, P can have any value.
* If P is negative, this means round to the P'th place to the right of the
decimal point; positive values mean to the left of the decimal point.
P of 0 means round to integer.
* to find out the current global A, use Math::SomeClass->accuracy()
* to find out the current global P, use Math::SomeClass->precision()
* use $x->accuracy() respective $x->precision() for the local
setting of $x.
* Please note that $x->accuracy() respective $x->precision()
return eventually defined global A or P, when $x's A or P is not
set.
```

Creating numbers

```
* When you create a number, you can give the desired A or P via:
$x = Math::BigInt->new($number,$A,$P);
* Only one of A or P can be defined, otherwise the result is NaN
* If no A or P is give ($x = Math::BigInt->new($number) form), then the
globals (if set) will be used. Thus changing the global defaults later on
will not change the A or P of previously created numbers (i.e., A and P of
$x will be what was in effect when $x was created)
* If given undef for A and P, NO rounding will occur, and the globals will
NOT be used. This is used by subclasses to create numbers without
suffering rounding in the parent. Thus a subclass is able to have its own
globals enforced upon creation of a number by using
$x = Math::BigInt->new($number,undef,undef):

use Math::BigInt::SomeSubclass;
use Math::BigInt;

Math::BigInt->accuracy(2);
Math::BigInt::SomeSubClass->accuracy(3);
$x = Math::BigInt::SomeSubClass->new(1234);

$x is now 1230, and not 1200. A subclass might choose to implement
this otherwise, e.g. falling back to the parent's A and P.
```

Usage

```
* If A or P are enabled/defined, they are used to round the result of each
operation according to the rules below
* Negative P is ignored in Math::BigInt, since BigInts never have digits
after the decimal point
* Math::BigFloat uses Math::BigInt internally, but setting A or P inside
Math::BigInt as globals does not tamper with the parts of a BigFloat.
A flag is used to mark all Math::BigFloat numbers as 'never round'.
```

Precedence

```
* It only makes sense that a number has only one of A or P at a time.
If you set either A or P on one object, or globally, the other one will
be automatically cleared.
* If two objects are involved in an operation, and one of them has A in
effect, and the other P, this results in an error (NaN).
* A takes precedence over P (Hint: A comes before P).
If neither of them is defined, nothing is used, i.e. the result will have
as many digits as it can (with an exception for fdiv/fsqrt) and will not
be rounded.
* There is another setting for fdiv() (and thus for fsqrt()). If neither of
A or P is defined, fdiv() will use a fallback (F) of $div_scale digits.
If either the dividend's or the divisor's mantissa has more digits than
the value of F, the higher value will be used instead of F.
This is to limit the digits (A) of the result (just consider what would
happen with unlimited A and P in the case of 1/3 :-)
* fdiv will calculate (at least) 4 more digits than required (determined by
A, P or F), and, if F is not used, round the result
(this will still fail in the case of a result like 0.12345000000001 with A
or P of 5, but this can not be helped - or can it?)
* Thus you can have the math done by on Math::Big* class in two modi:
+ never round (this is the default):
This is done by setting A and P to undef. No math operation
will round the result, with fdiv() and fsqrt() as exceptions to guard
against overflows. You must explicitly call bround(), bfround() or
round() (the latter with parameters).
Note: Once you have rounded a number, the settings will 'stick' on it
and 'infect' all other numbers engaged in math operations with it, since
local settings have the highest precedence. So, to get SaferRound[tm],
use a copy() before rounding like this:

$x = Math::BigFloat->new(12.34);
$y = Math::BigFloat->new(98.76);
$z = $x * $y; # 1218.6984
print $x->copy()->fround(3); # 12.3 (but A is now 3!)
$z = $x * $y; # still 1218.6984, without
# copy would have been 1210!

+ round after each op:
After each single operation (except for testing like is_zero()), the
method round() is called and the result is rounded appropriately. By
setting proper values for A and P, you can have all-the-same-A or
all-the-same-P modes. For example, Math::Currency might set A to undef,
and P to -2, globally.

?Maybe an extra option that forbids local A & P settings would be in order,
?so that intermediate rounding does not 'poison' further math?
```

Overriding globals

```
* you will be able to give A, P and R as an argument to all the calculation
routines; the second parameter is A, the third one is P, and the fourth is
R (shift right by one for binary operations like badd). P is used only if
the first parameter (A) is undefined. These three parameters override the
globals in the order detailed as follows, i.e. the first defined value
wins:
(local: per object, global: global default, parameter: argument to sub)
+ parameter A
+ parameter P
+ local A (if defined on both of the operands: smaller one is taken)
+ local P (if defined on both of the operands: bigger one is taken)
+ global A
+ global P
+ global F
* fsqrt() will hand its arguments to fdiv(), as it used to, only now for two
arguments (A and P) instead of one
```

Local settings

```
* You can set A or P locally by using $x->accuracy() or
$x->precision()
and thus force different A and P for different objects/numbers.
* Setting A or P this way immediately rounds $x to the new value.
* $x->accuracy() clears $x->precision(), and vice versa.
```

Rounding

```
* the rounding routines will use the respective global or local settings.
fround()/bround() is for accuracy rounding, while ffround()/bfround()
is for precision
* the two rounding functions take as the second parameter one of the
following rounding modes (R):
'even', 'odd', '+inf', '-inf', 'zero', 'trunc', 'common'
* you can set/get the global R by using Math::SomeClass->round_mode()
or by setting $Math::SomeClass::round_mode
* after each operation, $result->round() is called, and the result may
eventually be rounded (that is, if A or P were set either locally,
globally or as parameter to the operation)
* to manually round a number, call $x->round($A,$P,$round_mode);
this will round the number by using the appropriate rounding function
and then normalize it.
* rounding modifies the local settings of the number:

$x = Math::BigFloat->new(123.456);
$x->accuracy(5);
$x->bround(4);

Here 4 takes precedence over 5, so 123.5 is the result and $x->accuracy()
will be 4 from now on.
```

Default values

```
* R: 'even'
* F: 40
* A: undef
* P: undef
```

Remarks

```
* The defaults are set up so that the new code gives the same results as
the old code (except in a few cases on fdiv):
+ Both A and P are undefined and thus will not be used for rounding
after each operation.
+ round() is thus a no-op, unless given extra parameters A and P
```

Infinity and Not a Number

While BigInt has extensive handling of inf and NaN, certain quirks remain.

oct()/hex()

These perl routines currently (as of Perl v.5.8.6) cannot handle passed inf.

```
te@linux:> perl -wle 'print 2 ** 3333'
inf
te@linux:> perl -wle 'print 2 ** 3333 == 2 ** 3333'
1
te@linux:> perl -wle 'print oct(2 ** 3333)'
0
te@linux:> perl -wle 'print hex(2 ** 3333)'
Illegal hexadecimal digit 'i' ignored at -e line 1.
0
```

The same problems occur if you pass them Math::BigInt->*binf()* objects. Since overloading these routines is not possible, this cannot be fixed from BigInt.

==, !=, <, >, <=, >= with NaNs

BigInt's *bcmp()* routine currently returns undef to signal that a NaN was involved in a comparison. However, the overload code turns that into either 1 or '' and thus operations like `NaN != NaN`migh t return wrong values.

log(-inf)

`log(-inf)` is highly weird. Since log(-x)=pi*i+log(x), then log(-inf)=pi*i+inf. However, since the imaginary part is finite, the real infinity "overshadows" it, so the number might as well just be infinity. However, the result is a complex number, and since BigInt/BigFloat can only have real numbers as results, the result is NaN.

exp(), *cos()*, *sin()*, *atan2()*

These all might have problems handling infinity right.

INTERNALS

The actual numbers are stored as unsigned big integers (with separate sign).

You should neither care about nor depend on the internal representation; it might change without notice. Use **ONLY** method calls like `$x->sign();` instead relying on the internal representation.

MATH LIBRARY

Math with the numbers is done (by default) by a module called `Math::BigInt::Calc` This is equivalent to saying:

```
use Math::BigInt try => 'Calc';
```

You can change this backend library by using:

```
use Math::BigInt try => 'GMP';
```

Note: General purpose packages should not be explicit about the library to use; let the script author decide which is best.

If your script works with huge numbers and Calc is too slow for them, you can also for the loading of one of these libraries and if none of them can be used, the code will die:

```
use Math::BigInt only => 'GMP,Pari';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and when this

also fails, revert to Math::BigInt::Calc:

```
use Math::BigInt try => 'Foo,Math::BigInt::Bar';
```

The library that is loaded last will be used. Note that this can be overwritten at any time by loading a different library, and numbers constructed with different libraries cannot be used in math operations together.

What library to use?

Note: General purpose packages should not be explicit about the library to use; let the script author decide which is best.

Math::BigInt::GMP and Math::BigInt::Pari are in cases involving big numbers much faster than Calc, however it is slower when dealing with very small numbers (less than about 20 digits) and when converting very large numbers to decimal (for instance for printing, rounding, calculating their length in decimal etc).

So please select carefully what library you want to use.

Different low-level libraries use different formats to store the numbers. However, you should **NOT** depend on the number having a specific format internally.

See the respective math library module documentation for further details.

SIGN

The sign is either '+', '-', 'NaN', '+inf' or '-inf'.

A sign of 'NaN' is used to represent the result when input arguments are not numbers or as a result of 0/0. '+inf' and '-inf' represent plus respectively minus infinity. You will get '+inf' when dividing a positive number by 0, and '-inf' when dividing any negative number by 0.

mantissa(), *exponent()* **and** *parts()*

`mantissa()` and `exponent()` return the said parts of the BigInt such that:

```
$m = $x->mantissa();
$e = $x->exponent();
$y = $m * ( 10 ** $e );
print "ok\n" if $x == $y;
```

`($m,$e) = $x->parts()`is just a shortcut that giv es you both of them in one go. Both the returned mantissa and exponent have a sign.

Currently, for BigInts `$e` is always 0, except +inf and -inf, where it is `+inf`; and for NaN, where it is `NaN`; and for `$x == 0`, where it is `1` (to be compatible with Math::BigFloat's internal representation of a zero as `0E1`).

`$m` is currently just a copy of the original number. The relation between `$e` and `$m` will stay always the same, though their real values might change.

EXAMPLES

```
use Math::BigInt;

sub bigint { Math::BigInt->new(shift); }

$x = Math::BigInt->bstr("1234") # string "1234"
$x = "$x"; # same as bstr()
$x = Math::BigInt->bneg("1234"); # BigInt "-1234"
$x = Math::BigInt->babs("-12345"); # BigInt "12345"
$x = Math::BigInt->bnorm("-0.00"); # BigInt "0"
$x = bigint(1) + bigint(2) # BigInt "3"
$x = bigint(1) + "2"; # ditto (auto-BigIntify of "2")
$x = bigint(1) # BigInt "1"
$x = $x + 5 / 2; # BigInt "3"
```

```
$x = $x ** 3; # BigInt "27"
$x *= 2; # BigInt "54"
$x = Math::BigInt->new(0); # BigInt "0"
$x--; # BigInt "-1"
$x = Math::BigInt->badd(4,5) # BigInt "9"
print $x->bsstr(); # 9e+0
```

Examples for rounding:

```
use Math::BigFloat;
use Test;

$x = Math::BigFloat->new(123.4567);
$y = Math::BigFloat->new(123.456789);
Math::BigFloat->accuracy(4); # no more A than 4

ok ($x->copy()->fround(),123.4); # even rounding
print $x->copy()->fround(),"\n"; # 123.4
Math::BigFloat->round_mode('odd'); # round to odd
print $x->copy()->fround(),"\n"; # 123.5
Math::BigFloat->accuracy(5); # no more A than 5
Math::BigFloat->round_mode('odd'); # round to odd
print $x->copy()->fround(),"\n"; # 123.46
$y = $x->copy()->fround(4),"\n"; # A = 4: 123.4
print "$y, ",$y->accuracy(),"\n"; # 123.4, 4

Math::BigFloat->accuracy(undef); # A not important now
Math::BigFloat->precision(2); # P important
print $x->copy()->bnorm(),"\n"; # 123.46
print $x->copy()->fround(),"\n"; # 123.46
```

Examples for converting:

```
my $x = Math::BigInt->new('0b1'.'01' x 123);
print "bin: ",$x->as_bin()," hex:",$x->as_hex()," dec: ",$x,"\n";
```

Autocreating constants

After `use Math::BigInt ':constant'`all the **in teger** decimal, hexadecimal and binary constants in the given scope are converted to `Math::BigInt` This conversion happens at compile time.

In particular,

```
perl -MMath::BigInt=:constant -e 'print 2**100,"\n"'
```

prints the integer value of `2**100`. Note that without conversion of constants the expression 2**100 will be calculated as perl scalar.

Please note that strings and floating point constants are not affected, so that

```
use Math::BigInt qw/:constant/;

$x = 12345678901234567890123456789012345678901234567890
+ 123456789123456789;
$y = '12345678901234567890123456789012345678901234567890'
+ '123456789123456789';
```

do not work. You need an explicit Math::BigInt->*new()* around one of the operands. You should also quote large constants to protect loss of precision:

```
use Math::BigInt;
```

```
$x = Math::BigInt->new('123456788912345678912345678912345678');
```

Without the quotes Perl would convert the large number to a floating point constant at compile time and then hand the result to BigInt, which results in an truncated result or a NaN.

This also applies to integers that look like floating point constants:

```
use Math::BigInt ':constant';

print ref(123e2),"\n";
print ref(123.2e2),"\n";
```

will print nothing but newlines. Use either bignum or Math::BigFloat to get this to work.

PERFORMANCE

Using the form `$x += $y`; etc over `$x = $x + $y` is faster, since a copy of `$x` must be made in the second case. For long numbers, the copy can eat up to 20% of the work (in the case of addition/subtraction, less for multiplication/division). If `$y` is very small compared to `$x`, the form `$x += $y` is MUCH faster than `$x = $x + $y` since making the copy of `$x` takes more time then the actual addition.

With a technique called copy-on-write, the cost of copying with overload could be minimized or even completely avoided. A test implementation of COW did show performance gains for overloaded math, but introduced a performance loss due to a constant overhead for all other operations. So Math::BigInt does currently not COW.

The rewritten version of this module (vs. v0.01) is slower on certain operations, like `new()`, `bstr()` and `numify()`. The reason are that it does now more work and handles much more cases. The time spent in these operations is usually gained in the other math operations so that code on the average should get (much) faster. If they don't, please contact the author.

Some operations may be slower for small numbers, but are significantly faster for big numbers. Other operations are now constant (O(1), like `bneg()`, `babs()` etc), instead of O(N) and thus nearly always take much less time. These optimizations were done on purpose.

If you find the Calc module to slow, try to install any of the replacement modules and see if they help you.

Alternative math libraries

You can use an alternative library to drive Math::BigInt. See the section "MATH LIBRARY" for more information.

For more benchmark results see <http://bloodgate.com/perl/benchmarks.html>.

SUBCLASSING

Subclassing Math::BigInt

The basic design of Math::BigInt allows simple subclasses with very little work, as long as a few simple rules are followed:

- The public API must remain consistent, i.e. if a sub-class is overloading addition, the sub-class must use the same name, in this case *badd()*. The reason for this is that Math::BigInt is optimized to call the object methods directly.
- The private object hash keys like `$x->{sign}` may not be changed, but additional keys can be added, like `$x->{_custom}`.
- Accessor functions are available for all existing object hash keys and should be used instead of directly accessing the internal hash keys. The reason for this is that Math::BigInt itself has a pluggable interface which permits it to support different storage methods.

More complex sub-classes may have to replicate more of the logic internal of Math::BigInt if they need to change more basic behaviors. A subclass that needs to merely change the output only needs to overload `bstr()`.

All other object methods and overloaded functions can be directly inherited from the parent class.

At the very minimum, any subclass will need to provide its own `new()` and can store additional hash keys in the object. There are also some package globals that must be defined, e.g.:

```
# Globals
$accuracy = undef;
$precision = -2; # round to 2 decimal places
$round_mode = 'even';
$div_scale = 40;
```

Additionally, you might want to provide the following two globals to allow auto-upgrading and auto-downgrading to work correctly:

```
$upgrade = undef;
$downgrade = undef;
```

This allows Math::BigInt to correctly retrieve package globals from the subclass, like `$SubClass::precision`. See t/Math/BigInt/Subclass.pm or t/Math/BigFloat/SubClass.pm completely functional subclass examples.

Don't forget to

```
use overload;
```

in your subclass to automatically inherit the overloading from the parent. If you like, you can change part of the overloading, look at Math::String for an example.

UPGRADING

When used like this:

```
use Math::BigInt upgrade => 'Foo::Bar';
```

certain operations will 'upgrade' their calculation and thus the result to the class Foo::Bar. Usually this is used in conjunction with Math::BigFloat:

```
use Math::BigInt upgrade => 'Math::BigFloat';
```

As a shortcut, you can use the module bignum:

```
use bignum;
```

Also good for one-liners:

```
perl -Mbignum -le 'print 2 ** 255'
```

This makes it possible to mix arguments of different classes (as in 2.5 + 2) as well es preserve accuracy (as in *sqrt(3)*).

Beware: This feature is not fully implemented yet.

Auto-upgrade

The following methods upgrade themselves unconditionally; that is if upgrade is in effect, they will always hand up their work:

bsqrt()
div()
blog()
bexp()

Beware: This list is not complete.

All other methods upgrade themselves only when one (or all) of their arguments are of the class mentioned in `$upgrade` (This might change in later versions to a more sophisticated scheme):

EXPORTS

`Math::BigInt` exports nothing by default, but can export the following methods:

```
bgcd
blcm
```

CAVEATS

Some things might not work as you expect them. Below is documented what is known to be troublesome:

bstr(), *bsstr()* and 'cmp'

Both `bstr()` and `bsstr()` as well as automated stringify via overload now drop the leading '+'. The old code would return '+3', the new returns '3'. This is to be consistent with Perl and to make `cmp` (especially with overloading) to work as you expect. It also solves problems with `Test.pm`, because its `ok()` uses 'eq' internally.

Mark Biggar said, when asked about to drop the '+' altogether, or make only `cmp` work:

```
I agree (with the first alternative), don't add the '+' on positive
numbers. It's not as important anymore with the new internal
form for numbers. It made doing things like abs and neg easier,
but those have to be done differently now anyway.
```

So, the following examples will now work all as expected:

```
use Test;
BEGIN { plan tests => 1 }
use Math::BigInt;

my $x = new Math::BigInt 3*3;
my $y = new Math::BigInt 3*3;

ok ($x,3*3);
print "$x eq 9" if $x eq $y;
print "$x eq 9" if $x eq '9';
print "$x eq 9" if $x eq 3*3;
```

Additionally, the following still works:

```
print "$x == 9" if $x == $y;
print "$x == 9" if $x == 9;
print "$x == 9" if $x == 3*3;
```

There is now a `bsstr()` method to get the string in scientific notation aka `1e+2` instead of `100`. Be advised that overloaded 'eq' always uses *bstr()* for comparison, but Perl will represent some numbers as 100 and others as 1e+308. If in doubt, convert both arguments to Math::BigInt before comparing them as strings:

```
use Test;
BEGIN { plan tests => 3 }
use Math::BigInt;

$x = Math::BigInt->new('1e56'); $y = 1e56;
ok ($x,$y); # will fail
ok ($x->bsstr(),$y); # okay
$y = Math::BigInt->new($y);
ok ($x,$y); # okay
```

Alternatively, simply use `<=>` for comparisons, this will get it always right. There is not yet a way to get a number automatically represented as a string that matches exactly the way Perl represents it.

See also the section about ''Infinity and Not a Number'' for problems in comparing NaNs.

int()

`int()` will return (at least for Perl v5.7.1 and up) another BigInt, not a Perl scalar:

```
$x = Math::BigInt->new(123);
$y = int($x); # BigInt 123
$x = Math::BigFloat->new(123.45);
$y = int($x); # BigInt 123
```

In all Perl versions you can use `as_number()` or `as_int` for the same effect:

```
$x = Math::BigFloat->new(123.45);
$y = $x->as_number(); # BigInt 123
$y = $x->as_int(); # ditto
```

This also works for other subclasses, like Math::String.

If you want a real Perl scalar, use `numify()`:

```
$y = $x->numify(); # 123 as scalar
```

This is seldom necessary, though, because this is done automatically, like when you access an array:

```
$z = $array[$x]; # does work automatically
```

length()

The following will probably not do what you expect:

```
$c = Math::BigInt->new(123);
print $c->length(),"\n"; # prints 30
```

It prints both the number of digits in the number and in the fraction part since print calls `length()` in list context. Use something like:

```
print scalar $c->length(),"\n"; # prints 3
```

bdiv()

The following will probably not do what you expect:

```
print $c->bdiv(10000),"\n";
```

It prints both quotient and remainder since print calls `bdiv()` in list context. Also, `bdiv()` will modify `$c`, so be careful. You probably want to use

```
print $c / 10000,"\n";
```

or, if you want to modify `$c` instead,

```
print scalar $c->bdiv(10000),"\n";
```

The quotient is always the greatest integer less than or equal to the real-valued quotient of the two operands, and the remainder (when it is non-zero) always has the same sign as the second operand; so, for example,

```
1 / 4 => ( 0, 1)
1 / -4 => (-1,-3)
-3 / 4 => (-1, 1)
-3 / -4 => ( 0,-3)
-11 / 2 => (-5,1)
11 /-2 => (-5,-1)
```

As a consequence, the behavior of the operator % agrees with the behavior of Perl's built-in % operator (as documented in the perlop(1) manpage), and the equation

```
$x == ($x / $y) * $y + ($x % $y)
```

holds true for any `$x` and `$y`, which justifies calling the two return values of *bdiv()* the quotient and remainder. The only exception to this rule are when `$y` == 0 and `$x` is negative, then the remainder will also be negative. See below under ''infinity handling'' for the reasoning behind this.

Perl's 'use integer;' changes the behaviour of % and / for scalars, but will not change BigInt's way to do things. This is because under 'use integer' Perl will do what the underlying C thinks is right and this is different for each system. If you need BigInt's behaving exactly like Perl's 'use integer', bug the author to implement it ;)

infinity handling

Here are some examples that explain the reasons why certain results occur while handling infinity:

The following table shows the result of the division and the remainder, so that the equation above holds true. Some ''ordinary'' cases are strewn in to show more clearly the reasoning:

```
A / B = C, R so that C * B + R = A
=========================================================
5 / 8 = 0, 5 0 * 8 + 5 = 5
0 / 8 = 0, 0 0 * 8 + 0 = 0
0 / inf = 0, 0 0 * inf + 0 = 0
0 /-inf = 0, 0 0 * -inf + 0 = 0
5 / inf = 0, 5 0 * inf + 5 = 5
5 /-inf = 0, 5 0 * -inf + 5 = 5
-5/ inf = 0, -5 0 * inf + -5 = -5
-5/-inf = 0, -5 0 * -inf + -5 = -5
inf/ 5 = inf, 0 inf * 5 + 0 = inf
-inf/ 5 = -inf, 0 -inf * 5 + 0 = -inf
inf/ -5 = -inf, 0 -inf * -5 + 0 = inf
-inf/ -5 = inf, 0 inf * -5 + 0 = -inf
5/ 5 = 1, 0 1 * 5 + 0 = 5
-5/ -5 = 1, 0 1 * -5 + 0 = -5
inf/ inf = 1, 0 1 * inf + 0 = inf
-inf/-inf = 1, 0 1 * -inf + 0 = -inf
inf/-inf = -1, 0 -1 * -inf + 0 = inf
-inf/ inf = -1, 0 1 * -inf + 0 = -inf
8/ 0 = inf, 8 inf * 0 + 8 = 8
inf/ 0 = inf, inf inf * 0 + inf = inf
0/ 0 = NaN
```

These cases below violate the ''remainder has the sign of the second of the two arguments'', since they wouldn't match up otherwise.

```
A / B = C, R so that C * B + R = A
=========================================================
-inf/ 0 = -inf, -inf -inf * 0 + inf = -inf
-8/ 0 = -inf, -8 -inf * 0 + 8 = -8
```

Modifying and =

Beware of:

```
$x = Math::BigFloat->new(5);
$y = $x;
```

It will not do what you think, e.g. making a copy of `$x`. Instead it just makes a second reference to the **same** object and stores it in `$y`. Thus anything that modifies `$x` (except overloaded operators) will modify `$y`, and vice versa. Or in other words, `=` is only safe if you modify your BigInts only via overloaded math. As soon as you use a method call it breaks:

```
$x->bmul(2);
print "$x, $y\n"; # prints '10, 10'
```

If you want a true copy of `$x`, use:

```
$y = $x->copy();
```

You can also chain the calls like this, this will make first a copy and then multiply it by 2:

```
$y = $x->copy()->bmul(2);
```

See also the documentation for overload.pm regarding `=`.

bpow

`bpow()` (and the rounding functions) now modifies the first argument and returns it, unlike the old code which left it alone and only returned the result. This is to be consistent with `badd()` etc. The first three will modify `$x`, the last one won't:

```
print bpow($x,$i),"\n"; # modify $x
print $x->bpow($i),"\n"; # ditto
print $x **= $i,"\n"; # the same
print $x ** $i,"\n"; # leave $x alone
```

The form `$x **= $y` is faster than `$x = $x ** $y;`, though.

Overloading -$x

The following:

```
$x = -$x;
```

is slower than

```
$x->bneg();
```

since overload calls `sub($x,0,1);` instead of `neg($x)`. The first variant needs to preserve `$x` since it does not know that it later will get overwritten. This makes a copy of `$x` and takes O(N), but `$x->`*bneg()* is O(1)

Mixing different object types

In Perl you will get a floating point value if you do one of the following:

```
$float = 5.0 + 2;
$float = 2 + 5.0;
$float = 5 / 2;
```

With overloaded math, only the first two variants will result in a BigFloat:

```
use Math::BigInt;
use Math::BigFloat;

$mbf = Math::BigFloat->new(5);
$mbi2 = Math::BigInteger->new(5);
$mbi = Math::BigInteger->new(2);

# what actually gets called:
$float = $mbf + $mbi; # $mbf->badd()
$float = $mbf / $mbi; # $mbf->bdiv()
$integer = $mbi + $mbf; # $mbi->badd()
$integer = $mbi2 / $mbi; # $mbi2->bdiv()
$integer = $mbi2 / $mbf; # $mbi2->bdiv()
```

This is because math with overloaded operators follows the first (dominating) operand, and the operation of that is called and returns thus the result. So, *Math::BigInt::bdiv()* will always return a Math::BigInt, regardless whether the result should be a Math::BigFloat or the

second operant is one.

To get a Math::BigFloat you either need to call the operation manually, make sure the operands are already of the proper type or casted to that type via Math::BigFloat->*new()*:

```
$float = Math::BigFloat->new($mbi2) / $mbi; # = 2.5
```

Beware of simple ''casting'' the entire expression, this would only convert the already computed result:

```
$float = Math::BigFloat->new($mbi2 / $mbi); # = 2.0 thus wrong!
```

Beware also of the order of more complicated expressions like:

```
$integer = ($mbi2 + $mbi) / $mbf; # int / float => int
$integer = $mbi2 / Math::BigFloat->new($mbi); # ditto
```

If in doubt, break the expression into simpler terms, or cast all operands to the desired resulting type.

Scalar values are a bit different, since:

```
$float = 2 + $mbf;
$float = $mbf + 2;
```

will both result in the proper type due to the way the overloaded math works.

This section also applies to other overloaded math packages, like Math::String.

One solution to you problem might be autoupgrading|upgrading. See the pragmas bignum, bigint and bigrat for an easy way to do this.

bsqrt()

`bsqrt()` works only good if the result is a big integer, e.g. the square root of 144 is 12, but from 12 the square root is 3, regardless of rounding mode. The reason is that the result is always truncated to an integer.

If you want a better approximation of the square root, then use:

```
$x = Math::BigFloat->new(12);
Math::BigFloat->precision(0);
Math::BigFloat->round_mode('even');
print $x->copy->bsqrt(),"\n"; # 4

Math::BigFloat->precision(2);
print $x->bsqrt(),"\n"; # 3.46
print $x->bsqrt(3),"\n"; # 3.464
```

brsft()

For negative numbers in base see also brsft.

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Math::BigFloat, Math::BigRat and Math::Big as well as Math::BigInt::Pari and Math::BigInt::GMP.

The pragmas bignum, bigint and bigrat also might be of interest because they solve the autoupgrading/downgrading issue, at least partly.

The package at <http://search.cpan.org/search?mode=module&query=Math%3A%3ABigInt> contains more documentation including a full version history, testcases, empty subclass files and benchmarks.

AUTHORS

Original code by Mark Biggar, overloaded interface by Ilya Zakharevich. Completely rewritten by Tels http://bloodgate.com in late 2000, 2001 - 2006 and still at it in 2007.

Many people contributed in one or more ways to the final beast, see the file CREDITS for an (incomplete) list. If you miss your name, please drop me a mail. Thank you!