



**Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!**

***Rocky Enterprise Linux 9.2 Manual Pages on command 'Math::BigInt.3perl'***

***\$ man Math::BigInt.3perl***

Math::BigInt(3perl)          Perl Programmers Reference Guide          Math::BigInt(3perl)

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 falls back to
# pure Perl if the GMP library is not installed):
# (See also the L<MATH LIBRARY> section!)
# warns 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 = '-';

# Configuration methods (may be used as class methods and instance methods)

Math::BigInt->accuracy();    # get class accuracy
Math::BigInt->accuracy($n); # set class accuracy
Math::BigInt->precision();    # get class precision
Math::BigInt->precision($n); # set class precision
```

```

Math::BigInt->round_mode(); # get class rounding mode

Math::BigInt->round_mode($m); # set global round mode, must be one of
    # 'even', 'odd', '+inf', '-inf', 'zero',
    # 'trunc', or 'common'

Math::BigInt->config(); # return hash with configuration

# Constructor methods (when the class methods below are used as instance
# methods, the value is assigned the invocand)

$x = Math::BigInt->new($str); # defaults to 0

$x = Math::BigInt->new('0x123'); # from hexadecimal

$x = Math::BigInt->new('0b101'); # from binary

$x = Math::BigInt->from_hex('cafe'); # from hexadecimal

$x = Math::BigInt->from_oct('377'); # from octal

$x = Math::BigInt->from_bin('1101'); # from binary

$x = Math::BigInt->from_base('why', 36); # from any base

$x = Math::BigInt->bzero(); # create a +0

$x = Math::BigInt->bone(); # create a +1

$x = Math::BigInt->bone('-'); # create a -1

$x = Math::BigInt->binf(); # create a +inf

$x = Math::BigInt->binf('-'); # create a -inf

$x = Math::BigInt->bnan(); # create a Not-A-Number

$x = Math::BigInt->bpi(); # returns pi

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

$y = $x->as_int(); # return as a Math::BigInt

# Boolean methods (these don't modify the invocand)

$x->is_zero(); # if $x is 0

$x->is_one(); # if $x is +1

$x->is_one("+"); # ditto

$x->is_one("-"); # if $x is -1

$x->is_inf(); # if $x is +inf or -inf

$x->is_inf("+"); # if $x is +inf

$x->is_inf("-"); # if $x is -inf

$x->is_nan(); # if $x is NaN

$x->is_positive(); # if $x > 0

```

```

$x->is_pos();      # ditto
$x->is_negative(); # if $x < 0
$x->is_neg();      # ditto
$x->is_odd();      # if $x is odd
$x->is_even();     # if $x is even
$x->is_int();      # if $x is an integer

# Comparison methods

$x->bcmp($y);      # compare numbers (undef, < 0, == 0, > 0)
$x->bacmp($y);     # compare absolutely (undef, < 0, == 0, > 0)
$x->beq($y);       # true if and only if $x == $y
$x->bne($y);       # true if and only if $x != $y
$x->blt($y);       # true if and only if $x < $y
$x->ble($y);       # true if and only if $x <= $y
$x->bgt($y);       # true if and only if $x > $y
$x->bge($y);       # true if and only if $x >= $y

# Arithmetic methods

$x->bneg();        # negation
$x->babs();        # absolute value
$x->bsgn();        # sign function (-1, 0, 1, or NaN)
$x->bnorm();       # normalize (no-op)
$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)
$x->bmuladd($y,$z); # $x = $x * $y + $z
$x->bdiv($y);      # division (floored), set $x to quotient
                  # return (quo,rem) or quo if scalar
$x->btdiv($y);     # division (truncated), set $x to quotient
                  # return (quo,rem) or quo if scalar
$x->bmod($y);      # modulus (x % y)
$x->btmod($y);     # modulus (truncated)
$x->bmodinv($mod); # modular multiplicative inverse

```

```

$x->bmodpow($y,$mod); # modular exponentiation (( $x ** y$ ) %  $mod$ )
$x->bpow($y); # power of arguments ( $x ** y$ )
$x->blog(); # logarithm of  $x$  to base  $e$  (Euler's number)
$x->blog($base); # logarithm of  $x$  to base  $base$  (e.g., base 2)
$x->bexp(); # calculate  $e ** x$  where  $e$  is Euler's number
$x->bnok($y); #  $x$  over  $y$  (binomial coefficient  $n$  over  $k$ )
$x->buparrow($n, $y); # Knuth's up-arrow notation
$x->backermann($y); # the Ackermann function
$x->bsin(); # sine
$x->bcos(); # cosine
$x->batan(); # inverse tangent
$x->batan2($y); # two-argument inverse tangent
$x->bsqrt(); # calculate square root
$x->broot($y); #  $y$ 'th root of  $x$  (e.g.  $y == 3 \Rightarrow$  cubic root)
$x->bfac(); # factorial of  $x$  ( $1*2*3*4*..x$ )
$x->blsft($n); # left shift  $n$  places in base 2
$x->blsft($n,$b); # left shift  $n$  places in base  $b$ 
# returns (quo,rem) or quo (scalar context)
$x->brsft($n); # right shift  $n$  places in base 2
$x->brsft($n,$b); # right shift  $n$  places in base  $b$ 
# returns (quo,rem) or quo (scalar context)
# Bitwise methods
$x->band($y); # bitwise and
$x->bior($y); # bitwise inclusive or
$x->bxor($y); # bitwise exclusive or
$x->bnot(); # bitwise not (two's complement)
# Rounding methods
$x->round($A,$P,$mode); # round to accuracy or precision using
# rounding mode  $mode$ 
$x->bround($n); # accuracy: preserve  $n$  digits
$x->bfround($n); #  $n > 0$ : round to  $n$ th digit left of dec. point
#  $n < 0$ : round to  $n$ th digit right of dec. point
$x->bfloor(); # round towards minus infinity

```

```

$x->bceil();      # round towards plus infinity
$x->bint();       # round towards zero
# Other mathematical methods
$x->bgcd($y);     # greatest common divisor
$x->blcm($y);     # least common multiple
# Object property methods (do not modify the invocand)
$x->sign();       # the sign, either +, - or NaN
$x->digit($n);    # the nth digit, counting from the right
$x->digit(-$n);   # the nth digit, counting from the left
$x->length();     # return number of digits in number
($xl,$f) = $x->length(); # length of number and length of fraction
                    # part, latter is always 0 digits long
                    # for Math::BigInt objects
$x->mantissa();   # return (signed) mantissa as a Math::BigInt
$x->exponent();   # return exponent as a Math::BigInt
$x->parts();      # return (mantissa,exponent) as a Math::BigInt
$x->sparts();     # mantissa and exponent (as integers)
$x->nparts();     # mantissa and exponent (normalised)
$x->eparts();     # mantissa and exponent (engineering notation)
$x->dparts();     # integer and fraction part
# Conversion methods (do not modify the invocand)
$x->bstr();       # decimal notation, possibly zero padded
$x->bsstr();     # string in scientific notation with integers
$x->bnstr();     # string in normalized notation
$x->bestr();     # string in engineering notation
$x->bdstr();     # string in decimal notation
$x->to_hex();    # as signed hexadecimal string
$x->to_bin();    # as signed binary string
$x->to_oct();    # as signed octal string
$x->to_bytes();  # as byte string
$x->to_base($b); # as string in any base
$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

# Other conversion methods

$x->numify();    # return as scalar (might overflow or underflow)

```

## DESCRIPTION

Math::BigInt provides support for arbitrary precision integers. Overloading is also provided for Perl operators.

### Input

Input values to these routines may be any scalar number or string that looks like a number and represents an integer.

- ? Leading and trailing whitespace is ignored.
- ? Leading and trailing zeros are ignored.
- ? If the string has a "0x" prefix, it is interpreted as a hexadecimal number.
- ? If the string has a "0b" prefix, it is interpreted as a binary number.
- ? One underline is allowed between any two digits.
- ? If the string can not be interpreted, NaN is returned.

Octal numbers are typically prefixed by "0", but since leading zeros are stripped, these methods can not automatically recognize octal numbers, so use the constructor from\_oct() to interpret octal strings.

Some examples of valid string input

Input string	Resulting value
123	123
1.23e2	123
12300e-2	123
0xcafe	51966
0b1101	13
67_538_754	67538754
-4_5_6.7_8_9e+0_1_0	-4567890000000

Input given as scalar numbers might lose precision. Quote your input to ensure that no digits are lost:

```

$x = Math::BigInt->new( 56789012345678901234 ); # bad

$x = Math::BigInt->new('56789012345678901234'); # good

```

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();
```

## Output

Output values are usually Math::BigInt objects.

Boolean operators "is\_zero()", "is\_one()", "is\_inf()", etc. return true or false.

Comparison operators "bcmp()" and "bacmp()" return -1, 0, 1, or undef.

## METHODS

### Configuration 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.

Setting a class variable effects all object instance that are created afterwards.

#### accuracy()

```
Math::BigInt->accuracy(5);    # set class accuracy
```

```
$x->accuracy(5);             # set instance accuracy
```

```
$A = Math::BigInt->accuracy(); # get class accuracy
```

```
$A = $x->accuracy();         # get instance accuracy
```

Set or get the accuracy, i.e., the number of significant digits. The accuracy must be an integer. If the accuracy is set to "undef", no rounding is done.

Alternatively, one can round the results explicitly using one of "round()", "bround()" or "bfround()" or by passing the desired accuracy to the method as an additional parameter:

```
my $x = Math::BigInt->new(30000);
```

```
my $y = Math::BigInt->new(7);
```

```
print scalar $x->copy()->bdiv($y, 2);    # prints 4300
```

```
print scalar $x->copy()->bdiv($y)->bround(2); # prints 4300
```

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

```
$y = Math::BigInt->new(1234567); # $y is not rounded
```

```
Math::BigInt->accuracy(4);    # set class accuracy to 4
```

```
$x = Math::BigInt->new(1234567); # $x is rounded automatically
```

```
print "$x $y";               # prints "1235000 1234567"
```

```
print $x->accuracy();        # prints "4"
```

```

print $y->accuracy();    # also prints "4", since
                        # class accuracy is 4
Math::BigInt->accuracy(5); # set class accuracy to 5
print $x->accuracy();    # prints "4", since instance
                        # accuracy is 4
print $y->accuracy();    # prints "5", since no instance
                        # accuracy, and class accuracy is 5

```

Note: 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`.

`precision()`

```

Math::BigInt->precision(-2); # set class precision
$x->precision(-2);          # set instance precision
$P = Math::BigInt->precision(); # get class precision
$P = $x->precision();        # get instance precision

```

Set or get the precision, i.e., the place to round relative to the decimal point. The precision must be an integer. Setting the precision to `$P` means that each number is rounded up or down, depending on the rounding mode, to the nearest multiple of  $10^{**$P}$ . If the precision is set to "undef", no rounding is done.

You might 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.

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

```

$y = Math::BigInt->new(1234567); # $y is not rounded
Math::BigInt->precision(4);      # set class precision to 4
$x = Math::BigInt->new(1234567); # $x is rounded automatically
print $x;                        # prints "1230000"

```

Note: 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`.

`div_scale()`

Set/get the fallback accuracy. This is the accuracy used when neither accuracy nor precision is set explicitly. It is used when a computation might otherwise attempt to return an infinite number of digits.

round\_mode()

Set/get the rounding mode.

upgrade()

Set/get the class for upgrading. When a computation might result in a non-integer, the operands are upgraded to this class. This is used for instance by bignum. The default is "undef", thus the following operation creates a Math::BigInt, not a Math::BigFloat:

```
my $i = Math::BigInt->new(123);
my $f = Math::BigFloat->new('123.1');
print $i + $f, "\n";          # prints 246
```

downgrade()

Set/get the class for downgrading. The default is "undef". Downgrading is not done by Math::BigInt.

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.

config()

```
Math::BigInt->config("trap_nan" => 1); # set
$accu = Math::BigInt->config("accuracy"); # get
```

Set or get class variables. Read-only parameters are marked as RO. Read-write parameters are marked as RW. The following parameters are supported.

Parameter	RO/RW	Description
Example		
=====		
lib	RO	Name of the math backend library Math::BigInt::Calc
lib_version	RO	Version of the math backend library 0.30
class	RO	The class of config you just called Math::BigRat
version	RO	version number of the class you used 0.10

upgrade	RW	To which class numbers are upgraded
		undef
downgrade	RW	To which class numbers are downgraded
		undef
precision	RW	Global precision
		undef
accuracy	RW	Global accuracy
		undef
round_mode	RW	Global round mode
		even
div_scale	RW	Fallback accuracy for division etc.
		40
trap_nan	RW	Trap NaNs
		undef
trap_inf	RW	Trap +inf/-inf
		undef

#### Constructor methods

`new()`

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

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

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

`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_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\_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.

from\_bytes()

```
$x = Math::BigInt->from_bytes("\xf3\x6b"); # $x = 62315
```

Interpret the input as a byte string, assuming big endian byte order. The output is always a non-negative, finite integer.

In some special cases, from\_bytes() matches the conversion done by unpack():

```
$b = "\x4e";           # one char byte string
$x = Math::BigInt->from_bytes($b);   # = 78
$y = unpack "C", $b;     # ditto, but scalar
$b = "\xf3\x6b";       # two char byte string
$x = Math::BigInt->from_bytes($b);   # = 62315
$y = unpack "S>", $b;    # ditto, but scalar
$b = "\x2d\xe0\x49\xad"; # four char byte string
$x = Math::BigInt->from_bytes($b);   # = 769673645
$y = unpack "L>", $b;    # ditto, but scalar
$b = "\x2d\xe0\x49\xad\x2d\xe0\x49\xad"; # eight char byte string
$x = Math::BigInt->from_bytes($b);   # = 3305723134637787565
$y = unpack "Q>", $b;    # ditto, but scalar
```

from\_base()

Given a string, a base, and an optional collation sequence, interpret the string as a number in the given base. The collation sequence describes the value of each character in the string.

If a collation sequence is not given, a default collation sequence is used. If the base is less than or equal to 36, the collation sequence is the string consisting of the 36 characters "0" to "9" and "A" to "Z". In this case, the letter case in the input is ignored. If the base is greater than 36, and smaller than or equal to 62, the collation sequence is the string consisting of the 62 characters "0" to "9", "A" to "Z", and "a" to "z". A base larger than 62 requires the collation sequence to be

specified explicitly.

These examples show standard binary, octal, and hexadecimal conversion. All cases return 250.

```
$x = Math::BigInt->from_base("11111010", 2);
```

```
$x = Math::BigInt->from_base("372", 8);
```

```
$x = Math::BigInt->from_base("fa", 16);
```

When the base is less than or equal to 36, and no collation sequence is given, the letter case is ignored, so both of these also return 250:

```
$x = Math::BigInt->from_base("6Y", 16);
```

```
$x = Math::BigInt->from_base("6y", 16);
```

When the base greater than 36, and no collation sequence is given, the default collation sequence contains both uppercase and lowercase letters, so the letter case in the input is not ignored:

```
$x = Math::BigInt->from_base("6S", 37);    # $x is 250
```

```
$x = Math::BigInt->from_base("6s", 37);    # $x is 276
```

```
$x = Math::BigInt->from_base("121", 3);    # $x is 16
```

```
$x = Math::BigInt->from_base("XYZ", 36);   # $x is 44027
```

```
$x = Math::BigInt->from_base("Why", 42);   # $x is 58314
```

The collation sequence can be any set of unique characters. These two cases are equivalent

```
$x = Math::BigInt->from_base("100", 2, "01"); # $x is 4
```

```
$x = Math::BigInt->from_base("|--", 2, "-|"); # $x is 4
```

bzero()

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

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

Returns a new Math::BigInt object representing zero. If used as an instance method, assigns the value to the invocand.

bone()

```
$x = Math::BigInt->bone();    # +1
```

```
$x = Math::BigInt->bone("+"); # +1
```

```
$x = Math::BigInt->bone("-"); # -1
```

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

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

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

Creates a new `Math::BigInt` object representing one. The optional argument is either '-' or '+', indicating whether you want plus one or minus one. If used as an instance method, assigns the value to the invocand.

`binf()`

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

Creates a new `Math::BigInt` object representing infinity. The optional argument is either '-' or '+', indicating whether you want infinity or minus infinity. If used as an instance method, assigns the value to the invocand.

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

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

`bnan()`

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

Creates a new `Math::BigInt` object representing NaN (Not A Number). If used as an instance method, assigns the value to the invocand.

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

`bpi()`

```
$x = Math::BigInt->bpi(100);    # 3
```

```
$x->bpi(100);                  # 3
```

Creates a new `Math::BigInt` object representing PI. If used as an instance method, assigns the value to the invocand. With `Math::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....
```

`copy()`

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

`as_int()`

`as_number()`

These methods are called when `Math::BigInt` encounters an object it doesn't know how to handle. For instance, assume `$x` is a `Math::BigInt`, or subclass thereof, and `$y` is

defined, but not a `Math::BigInt`, or subclass thereof. If you do

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

`$y` needs to be converted into an object that `$x` can deal with. This is done by first checking if `$y` is something that `$x` might be upgraded to. If that is the case, no further attempts are made. The next is to see if `$y` supports the method `"as_int()`". If it does, `"as_int()`" is called, but if it doesn't, the next thing is to see if `$y` supports the method `"as_number()`". If it does, `"as_number()`" is called. The method `"as_int()`" (and `"as_number()`") is expected to return either an object that has the same class as `$x`, a subclass thereof, or a string that `"ref($x)->new()`" can parse to create an object.

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

In `Math::BigInt`, `"as_int()`" has the same effect as `"copy()`".

## Boolean methods

None of these methods modify the invocand object.

`is_zero()`

```
$x->is_zero();      # true if $x is 0
```

Returns true if the invocand is zero and false otherwise.

`is_one( [ SIGN ] )`

```
$x->is_one();       # true if $x is +1
```

```
$x->is_one("+");    # ditto
```

```
$x->is_one("-");    # true if $x is -1
```

Returns true if the invocand is one and false otherwise.

`is_finite()`

```
$x->is_finite();    # true if $x is not +inf, -inf or NaN
```

Returns true if the invocand is a finite number, i.e., it is neither `+inf`, `-inf`, nor

`NaN`.

`is_inf( [ SIGN ] )`

```
$x->is_inf();       # true if $x is +inf
```

```
$x->is_inf("+");    # ditto
```

```
$x->is_inf("-");    # true if $x is -inf
```

Returns true if the invocand is infinite and false otherwise.

`is_nan()`

```
$x->is_nan();      # true if $x is NaN
```

is\_positive()

is\_pos()

```
$x->is_positive();  # true if > 0
```

```
$x->is_pos();      # ditto
```

Returns true if the invocand is positive and false otherwise. A "NaN" is neither positive nor negative.

is\_negative()

is\_neg()

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

```
$x->is_neg();      # ditto
```

Returns true if the invocand is negative and false otherwise. A "NaN" is neither positive nor negative.

is\_non\_positive()

```
$x->is_non_positive();  # true if <= 0
```

Returns true if the invocand is negative or zero.

is\_non\_negative()

```
$x->is_non_negative();  # true if >= 0
```

Returns true if the invocand is positive or zero.

is\_odd()

```
$x->is_odd();       # true if odd, false for even
```

Returns true if the invocand is odd and false otherwise. "NaN", "+inf", and "-inf" are neither odd nor even.

is\_even()

```
$x->is_even();      # true if $x is even
```

Returns true if the invocand is even and false otherwise. "NaN", "+inf", "-inf" are not integers and are neither odd nor even.

is\_int()

```
$x->is_int();       # true if $x is an integer
```

Returns true if the invocand is an integer and false otherwise. "NaN", "+inf", "-inf" are not integers.

## Comparison methods

None of these methods modify the invocand object. Note that a "NaN" is neither less than,

greater than, or equal to anything else, even a "NaN".

`bcmp()`

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

Returns -1, 0, 1 depending on whether \$x is less than, equal to, or greater than \$y.

Returns undef if any operand is a NaN.

`bacmp()`

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

Returns -1, 0, 1 depending on whether the absolute value of \$x is less than, equal to, or greater than the absolute value of \$y. Returns undef if any operand is a NaN.

`beq()`

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

Returns true if and only if \$x is equal to \$y, and false otherwise.

`bne()`

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

Returns true if and only if \$x is not equal to \$y, and false otherwise.

`blt()`

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

Returns true if and only if \$x is less than \$y, and false otherwise.

`ble()`

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

Returns true if and only if \$x is less than or equal to \$y, and false otherwise.

`bgt()`

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

Returns true if and only if \$x is greater than \$y, and false otherwise.

`bge()`

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

Returns true if and only if \$x is greater than or equal to \$y, and false otherwise.

#### Arithmetic methods

These methods modify the invocand object and returns it.

`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)
```

Normalize the number. This is a no-op and is provided only for backwards compatibility.

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
```

Divides \$x by \$y by doing floored division (F-division), where the quotient is the floored (rounded towards negative infinity) quotient of the two operands. In list context, returns the quotient and the remainder. The remainder is either zero or has the same sign as the second operand. In scalar context, only the quotient is returned.

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,

$1 / 4 \Rightarrow (0, 1)$

$1 / -4 \Rightarrow (-1, -3)$

$-3 / 4 \Rightarrow (-1, 1)$

$-3 / -4 \Rightarrow (0, -3)$

$-11 / 2 \Rightarrow (-5, 1)$

$11 / -2 \Rightarrow (-5, -1)$

The behavior of the overloaded operator % agrees with the behavior of Perl's built-in % operator (as documented in the perlop manpage), and the equation

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

holds true for any finite \$x and finite, non-zero \$y.

Perl's "use integer" might change the behaviour of % and / for scalars. This is because under 'use integer' Perl does what the underlying C library thinks is right, and this varies. However, "use integer" does not change the way things are done with Math::BigInt objects.

btdiv()

```
$x->btdiv($y);      # divide, set $x to quotient
```

Divides \$x by \$y by doing truncated division (T-division), where quotient is the truncated (rounded towards zero) quotient of the two operands. In list context, returns the quotient and the remainder. The remainder is either zero or has the same sign as the first operand. In scalar context, only the quotient is returned.

bmod()

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

Returns \$x modulo \$y, i.e., the remainder after floored division (F-division). This method is like Perl's % operator. See "bdiv()".

btmod()

```
$x->btmod($y);     # modulus
```

Returns the remainder after truncated division (T-division). See "btdiv()".

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  $\text{mod}$ , satisfying

$$(x * y) \% \text{mod} = 1 \% \text{mod}$$

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  $\text{num}$  taken to the power  $\text{exp}$  in the modulus  $\text{mod}$  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)
```

"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 modifies  $x$ , 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.

`blog()`

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

If  $\text{base}$  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, which is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

when n and k are non-negative. This method implements the full Kronenburg extension (Kronenburg, M.J. "The Binomial Coefficient for Negative Arguments." 18 May 2011.

<http://arxiv.org/abs/1105.3689/>) illustrated by the following pseudo-code:

```
if n >= 0 and k >= 0:
    return binomial(n, k)
if k >= 0:
    return (-1)^k*binomial(-n+k-1, k)
if k <= n:
    return (-1)^(n-k)*binomial(-k-1, n-k)
else
    return 0
```

The behaviour is identical to the behaviour of the Maple and Mathematica function for negative integers n, k.

buparrow()

uparrow()

```
$a -> buparrow($n, $b);      # modifies $a
$x = $a -> uparrow($n, $b);  # does not modify $a
```

This method implements Knuth's up-arrow notation, where \$n is a non-negative integer representing the number of up-arrows. \$n = 0 gives multiplication, \$n = 1 gives exponentiation, \$n = 2 gives tetration, \$n = 3 gives hexation etc. The following illustrates the relation between the first values of \$n.

See <[https://en.wikipedia.org/wiki/Knuth%27s\\_up-arrow\\_notation](https://en.wikipedia.org/wiki/Knuth%27s_up-arrow_notation)>.

backermann()

ackermann()

```
$m -> backermann($n);    # modifies $a
$x = $m -> ackermann($n); # does not modify $a
```

This method implements the Ackermann function:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Its value grows rapidly, even for small inputs. For example,  $A(4, 2)$  is an integer of 19729 decimal digits.

See [https://en.wikipedia.org/wiki/Ackermann\\_function](https://en.wikipedia.org/wiki/Ackermann_function)

`bsin()`

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

Calculate the sine of  $\$x$ , modifying  $\$x$  in place.

In `Math::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).

`bcos()`

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

Calculate the cosine of  $\$x$ , modifying  $\$x$  in place.

In `Math::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 `Math::BigFloat`, unless upgrading is in effect, the result is truncated to an integer.

This method was added in v1.87 of `Math::BigFloat` (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 `Math::BigFloat`, unless upgrading is in effect, the result is truncated to an integer.

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

bsqrt()

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

"bsqrt()" returns the square root 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
```

broot()

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

Calculates the N'th root of \$x.

bfac()

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

Returns the factorial of \$x, i.e., the product of all positive integers up to and including \$x.

bdfac()

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

Returns the double factorial of \$x. If \$x is an even integer, returns the product of all positive, even integers up to and including \$x, i.e.,  $2*4*6*...*$x$ . If \$x is an odd integer, returns the product of all positive, odd integers, i.e.,  $1*3*5*...*$x$ .

bfib()

```
$F = $n->bfib();      # a single Fibonacci number
```

```
@F = $n->bfib();      # a list of Fibonacci numbers
```

In scalar context, returns a single Fibonacci number. In list context, returns a list of Fibonacci numbers. The invocand is the last element in the output.

The Fibonacci sequence is defined by

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2)$$

In list context,  $F(0)$  and  $F(n)$  is the first and last number in the output, respectively. For example, if  $\$n$  is 12, then " $@F = \$n->bfib()$ " returns the following values,  $F(0)$  to  $F(12)$ :

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

The sequence can also be extended to negative index  $n$  using the re-arranged recurrence relation

$$F(n-2) = F(n) - F(n-1)$$

giving the bidirectional sequence

n -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7  
F(n) 13 -8 5 -3 2 -1 1 0 1 1 2 3 5 8 13

If  $\$n$  is -12, the following values,  $F(0)$  to  $F(12)$ , are returned:

0, 1, -1, 2, -3, 5, -8, 13, -21, 34, -55, 89, -144

`blucas()`

$\$F = \$n->blucas();$  # a single Lucas number  
 $@F = \$n->blucas();$  # a list of Lucas numbers

In scalar context, returns a single Lucas number. In list context, returns a list of Lucas numbers. The invocand is the last element in the output.

The Lucas sequence is defined by

$$L(0) = 2$$
$$L(1) = 1$$
$$L(n) = L(n-1) + L(n-2)$$

In list context,  $L(0)$  and  $L(n)$  is the first and last number in the output, respectively. For example, if  $\$n$  is 12, then " $@L = \$n->blucas()$ " returns the following values,  $L(0)$  to  $L(12)$ :

2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322

The sequence can also be extended to negative index  $n$  using the re-arranged recurrence relation

$$L(n-2) = L(n) - L(n-1)$$

giving the bidirectional sequence

n -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7  
L(n) 29 -18 11 -7 4 -3 1 2 1 3 4 7 11 18 29

If  $\$n$  is -12, the following values,  $L(0)$  to  $L(-12)$ , are returned:

2, 1, -3, 4, -7, 11, -18, 29, -47, 76, -123, 199, -322

brsft()

`$x->brsft($n);` # right shift \$n places in base 2

`$x->brsft($n, $b);` # right shift \$n places in base \$b

The latter is equivalent to

`$x -> bdiv($b -> copy() -> bpow($n))`

blsft()

`$x->blsft($n);` # left shift \$n places in base 2

`$x->blsft($n, $b);` # left shift \$n places in base \$b

The latter is equivalent to

`$x -> bmul($b -> copy() -> bpow($n))`

#### Bitwise methods

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)

Two's complement (bitwise not). This is equivalent to, but faster than,

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

#### Rounding methods

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

Rounds \$x to an accuracy of \$N digits.

bfround()

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

Rounds to a multiple of  $10^{**}N$ . 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  $x$ .

`bceil()`

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

Round  $x$  towards plus infinity, i.e., set  $x$  to the smallest integer greater than or equal to  $x$ .

`bint()`

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

Round  $x$  towards zero.

Other mathematical methods

`bgcd()`

```
$x -> bgcd($y);      # GCD of $x and $y
```

```
$x -> bgcd($y, $z, ...); # GCD of $x, $y, $z, ...
```

Returns the greatest common divisor (GCD).

`blcm()`

```
$x -> blcm($y);      # LCM of $x and $y
```

```
$x -> blcm($y, $z, ...); # LCM of $x, $y, $z, ...
```

Returns the least common multiple (LCM).

Object property methods

`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.

digitsum()

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

Computes the sum of the base 10 digits and returns it.

bdigitsum()

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

Computes the sum of the base 10 digits and assigns the result to the invocand.

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 Math::BigInt objects, the length of the fraction part is always 0.

The following probably doesn't 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
```

mantissa()

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

Return the signed mantissa of \$x as a Math::BigInt.

exponent()

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

Return the exponent of \$x as a Math::BigInt.

parts()

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

Returns the significand (mantissa) and the exponent as integers. In Math::BigFloat, both are returned as Math::BigInt objects.

sparts()

Returns the significand (mantissa) and the exponent as integers. In scalar context,

only the significand is returned. The significand is the integer with the smallest absolute value. The output of "sparts()" corresponds to the output from "bsstr()".

In `Math::BigInt`, this method is identical to "parts()".

`nparts()`

Returns the significand (mantissa) and exponent corresponding to normalized notation.

In scalar context, only the significand is returned. For finite non-zero numbers, the significand's absolute value is greater than or equal to 1 and less than 10. The output of "nparts()" corresponds to the output from "bnstr()". In `Math::BigInt`, if the significand can not be represented as an integer, upgrading is performed or NaN is returned.

`eparts()`

Returns the significand (mantissa) and exponent corresponding to engineering notation.

In scalar context, only the significand is returned. For finite non-zero numbers, the significand's absolute value is greater than or equal to 1 and less than 1000, and the exponent is a multiple of 3. The output of "eparts()" corresponds to the output from "bestr()". In `Math::BigInt`, if the significand can not be represented as an integer, upgrading is performed or NaN is returned.

`dparts()`

Returns the integer part and the fraction part. If the fraction part can not be represented as an integer, upgrading is performed or NaN is returned. The output of "dparts()" corresponds to the output from "bdstr()".

## String conversion methods

`bstr()`

Returns a string representing the number using decimal notation. In `Math::BigFloat`, the output is zero padded according to the current accuracy or precision, if any of those are defined.

`bsstr()`

Returns a string representing the number using scientific notation where both the significand (mantissa) and the exponent are integers. The output corresponds to the output from "sparts()".

123 is returned as "123e+0"

1230 is returned as "123e+1"

12300 is returned as "123e+2"

12000 is returned as "12e+3"

10000 is returned as "1e+4"

#### bnstr()

Returns a string representing the number using normalized notation, the most common variant of scientific notation. For finite non-zero numbers, the absolute value of the significand is greater than or equal to 1 and less than 10. The output corresponds to the output from "nparts()".

123 is returned as "1.23e+2"

1230 is returned as "1.23e+3"

12300 is returned as "1.23e+4"

12000 is returned as "1.2e+4"

10000 is returned as "1e+4"

#### bestr()

Returns a string representing the number using engineering notation. For finite non-zero numbers, the absolute value of the significand is greater than or equal to 1 and less than 1000, and the exponent is a multiple of 3. The output corresponds to the output from "eparts()".

123 is returned as "123e+0"

1230 is returned as "1.23e+3"

12300 is returned as "12.3e+3"

12000 is returned as "12e+3"

10000 is returned as "10e+3"

#### bdstr()

Returns a string representing the number using decimal notation. The output corresponds to the output from "dparts()".

123 is returned as "123"

1230 is returned as "1230"

12300 is returned as "12300"

12000 is returned as "12000"

10000 is returned as "10000"

#### to\_hex()

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

Returns a hexadecimal string representation of the number. See also from\_hex().

to\_bin()

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

Returns a binary string representation of the number. See also from\_bin().

to\_oct()

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

Returns an octal string representation of the number. See also from\_oct().

to\_bytes()

```
$x = Math::BigInt->new("1667327589");
```

```
$s = $x->to_bytes();          # $s = "cafe"
```

Returns a byte string representation of the number using big endian byte order. The invocand must be a non-negative, finite integer. See also from\_bytes().

to\_base()

```
$x = Math::BigInt->new("250");
```

```
$x->to_base(2);  # returns "11111010"
```

```
$x->to_base(8);  # returns "372"
```

```
$x->to_base(16); # returns "fa"
```

Returns a string representation of the number in the given base. If a collation sequence is given, the collation sequence determines which characters are used in the output.

Here are some more examples

```
$x = Math::BigInt->new("16")->to_base(3);  # returns "121"
```

```
$x = Math::BigInt->new("44027")->to_base(36); # returns "XYZ"
```

```
$x = Math::BigInt->new("58314")->to_base(42); # returns "Why"
```

```
$x = Math::BigInt->new("4")->to_base(2, "-|"); # returns "|--"
```

See from\_base() for information and examples.

as\_hex()

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

As, "to\_hex()", but with a "0x" prefix.

as\_bin()

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

As, "to\_bin()", but with a "0b" prefix.

as\_oct()

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

As, "to\_oct()", but with a "0" prefix.

as\_bytes()

This is just an alias for "to\_bytes()".

#### Other conversion methods

numify()

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

Returns a Perl scalar from \$x. It is used automatically whenever a scalar is needed, for instance in array index operations.

#### ACCURACY and PRECISION

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::BigInt and Math::BigFloat 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) are be used.

#### Precision P

Precision is 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 at least 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) is 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 Math::BigInt objects, 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, 123.45000 has 8 and 0.000123 has 3.

The string output (of floating point numbers) is 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 Math::BigInt objects, 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.)

##### Directed rounding

These round modes always round in the same direction.

##### 'trunc'

Round towards zero. Remove all digits following the rounding place, i.e., replace them with zeros. Thus, 987.65 rounded to tens (P=1) becomes 980, and rounded to the fourth significant digit becomes 987.6 (A=4). 123.456 rounded to the second place after the decimal point (P=-2) becomes 123.46. This corresponds to the IEEE 754 rounding mode 'roundTowardZero'.

##### Rounding to nearest

These rounding modes round to the nearest digit. They differ in how they determine which

way to round in the ambiguous case when there is a tie.

'even'

Round towards the nearest even digit, e.g., when rounding to nearest integer, -5.5 becomes -6, 4.5 becomes 4, but 4.501 becomes 5. This corresponds to the IEEE 754 rounding mode 'roundTiesToEven'.

'odd'

Round towards the nearest odd digit, e.g., when rounding to nearest integer, 4.5 becomes 5, -5.5 becomes -5, but 5.501 becomes 6. This corresponds to the IEEE 754 rounding mode 'roundTiesToOdd'.

'+inf'

Round towards plus infinity, i.e., always round up. E.g., when rounding to the nearest integer, 4.5 becomes 5, -5.5 becomes -5, and 4.501 also becomes 5. This corresponds to the IEEE 754 rounding mode 'roundTiesToPositive'.

'-inf'

Round towards minus infinity, i.e., always round down. E.g., when rounding to the nearest integer, 4.5 becomes 4, -5.5 becomes -6, but 4.501 becomes 5. This corresponds to the IEEE 754 rounding mode 'roundTiesToNegative'.

'zero'

Round towards zero, i.e., round positive numbers down and negative numbers up. E.g., when rounding to the nearest integer, 4.5 becomes 4, -5.5 becomes -5, but 4.501 becomes 5. This corresponds to the IEEE 754 rounding mode 'roundTiesToZero'.

'common'

Round away from zero, i.e., round to the number with the largest absolute value. E.g., when rounding to the nearest integer, -1.5 becomes -2, 1.5 becomes 2 and 1.49 becomes 1. This corresponds to the IEEE 754 rounding mode 'roundTiesToAway'.

The handling of A & P in MBI/MBF (the old core code shipped with Perl versions  $\leq$  5.7.2)

is like this:

Precision

- \* `bround($p)` is able to round to \$p number of digits after the decimal point

- \* otherwise P is unused

Accuracy (significant digits)

- \* `bround($a)` rounds to \$a significant digits

- \* only `bdiv()` and `bsqrt()` take `A` as (optional) parameter
- + other operations simply create the same number (`bneg` etc), or more (`bmul`) of digits
- + rounding/truncating is only done when explicitly calling one of `bround` or `bfround`, and never for `Math::BigInt` (not implemented)
- \* `bsqrt()` simply hands its accuracy argument over to `bdiv`.
- \* the documentation and the comment in the code indicate two different ways on how `bdiv()` 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:

- \* 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 Math::BigInt objects 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 Math::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 bdiv/bsqrt) and will not be rounded.
- \* There is another setting for bdiv() (and thus for bsqrt()). If neither of A or P is defined, bdiv() 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 :-)
- \* bdiv 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 bdiv() and bsqrt() 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()->bround(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

\* `bsqrt()` will hand its arguments to `bdiv()`, 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.

`round()` is for accuracy rounding, while `bround()` 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 `bdiv`):

+ 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 `Math::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 `Math::BigInt`.

#### INTERNALS

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 dies:

```
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 is 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 get

'+inf' when dividing a positive number by 0, and '-inf' when dividing any negative number

by 0.

## 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");   # Math::BigInt "-1234"

$x = Math::BigInt->babs("-12345");  # Math::BigInt "12345"

$x = Math::BigInt->bnorm("-0.00");  # Math::BigInt "0"

$x = bigint(1) + bigint(2);       # Math::BigInt "3"

$x = bigint(1) + "2";             # ditto (auto-Math::BigIntify of "2")

$x = bigint(1);                   # Math::BigInt "1"

$x = $x + 5 / 2;                  # Math::BigInt "3"

$x = $x ** 3;                     # Math::BigInt "27"

$x *= 2;                           # Math::BigInt "54"

$x = Math::BigInt->new(0);          # Math::BigInt "0"

$x--;                              # Math::BigInt "-1"

$x = Math::BigInt->badd(4,5)       # Math::BigInt "9"

print $x->bsstr();                 # 9e+0
```

Examples for rounding:

```
use Math::BigFloat;

use Test::More;

$x = Math::BigFloat->new(123.4567);

$y = Math::BigFloat->new(123.456789);

Math::BigFloat->accuracy(4);       # no more A than 4

is ($x->copy()->bround(),123.4);    # even rounding

print $x->copy()->bround(),"\n";    # 123.4

Math::BigFloat->round_mode('odd'); # round to odd

print $x->copy()->bround(),"\n";    # 123.5

Math::BigFloat->accuracy(5);       # no more A than 5

Math::BigFloat->round_mode('odd'); # round to odd

print $x->copy()->bround(),"\n";    # 123.46

$y = $x->copy()->bround(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()->bround(),"\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 integer 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 is calculated using Perl scalars.

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

```

use Math::BigInt qw/:constant/;

$x = 1234567890123456789012345678901234567890
    + 123456789123456789;

$y = '1234567890123456789012345678901234567890'
    + '123456789123456789';

```

does not give you what you expect. 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('1234567889123456789123456789123456789');

```

Without the quotes Perl would convert the large number to a floating point constant at compile time and then hand the result to Math::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";

```

prints 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 than 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 needs 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 '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 as 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 always hands up their work:

```
div bsqrt blog bexp bpi bsin bcos batan batan2
```

All other methods upgrade themselves only when one (or all) of their arguments are of the class mentioned in `$upgrade`.

### 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:

#### Comparing numbers as strings

Both `"bstr()"` and `"bsstr()"` as well as `stringify` via overload drop the leading `'+'`.

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"` and `Test::More`, which `stringify` arguments before comparing them.

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 now works as expected:

```
use Test::More tests => 1;
use Math::BigInt;
my $x = Math::BigInt -> new(3*3);
my $y = Math::BigInt -> new(3*3);
is($x,3*3, 'multiplication');
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 represents 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::More tests => 3;
use Math::BigInt;
$x = Math::BigInt->new('1e56'); $y = 1e56;
is($x,$y);          # fails
is($x->bsstr(),$y); # okay
$y = Math::BigInt->new($y);
is($x,$y);          # okay
```

Alternatively, simply use "<=>" for comparisons, this always gets it 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()" returns (at least for Perl v5.7.1 and up) another Math::BigInt, not a Perl scalar:

```
$x = Math::BigInt->new(123);
```

```
$y = int($x);          # 123 as a Math::BigInt
```

```
$x = Math::BigFloat->new(123.45);
```

```
$y = int($x);          # 123 as a Math::BigFloat
```

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

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

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

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

Modifying and =

Beware of:

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

```
$y = $x;
```

This makes a second reference to the same object and stores it in \$y. Thus anything that modifies \$x (except overloaded operators) also modifies \$y, and vice versa. Or in other words, "=" is only safe if you modify your Math::BigInt objects 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 first makes a copy and then multiply it by 2:

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

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

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 gets overwritten. This makes a copy of \$x and takes O(N), but \$x->bneg() is O(1).

With overloaded operators, it is the first (dominating) operand that determines which method is called. Here are some examples showing what actually gets called in various cases.

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

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

For instance, `Math::BigInt->bdiv()` always returns a `Math::BigInt`, regardless of whether the second operand is a `Math::BigFloat`. To get a `Math::BigFloat` you either need to call the operation manually, make sure each operand already is a `Math::BigFloat`, or cast to that type via `Math::BigFloat->new()`:

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

Beware of casting the entire expression, as this would cast the result, at which point it is too late:

```
$float = Math::BigFloat->new($mbi2 / $mbi); # = 2
```

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;
```

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 your problem might be `autoupgrading|upgrading`. See the pragmas `bignum`,

bigint and bigrat for an easy way to do this.

## BUGS

Please report any bugs or feature requests to "bug-math-bigint at rt.cpan.org", or through the web interface at <https://rt.cpan.org/Ticket/Create.html?Queue=Math-BigInt> (requires login). We will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

## SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc Math::BigInt
```

You can also look for information at:

? RT: CPAN's request tracker

<https://rt.cpan.org/Public/Dist/Display.html?Name=Math-BigInt>

? AnnoCPAN: Annotated CPAN documentation

<http://annocpan.org/dist/Math-BigInt>

? CPAN Ratings

<https://cpanratings.perl.org/dist/Math-BigInt>

? MetaCPAN

<https://metacpan.org/release/Math-BigInt>

? CPAN Testers Matrix

<http://matrix.cpan testers.org/?dist=Math-BigInt>

? The Bignum mailing list

? Post to mailing list

"bignum at lists.scsys.co.uk"

? View mailing list

<http://lists.scsys.co.uk/pipermail/bignum/>

? Subscribe/Unsubscribe

<http://lists.scsys.co.uk/cgi-bin/mailman/listinfo/bignum>

## 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 and Math::BigRat as well as the backends Math::BigInt::FastCalc, Math::BigInt::GMP, and Math::BigInt::Pari.

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

#### AUTHORS

? Mark Biggar, overloaded interface by Ilya Zakharevich, 1996-2001.

? Completely rewritten by Tels <<http://bloodgate.com>>, 2001-2008.

? Florian Ragwitz <[flora@cpan.org](mailto:flora@cpan.org)>, 2010.

? Peter John Acklam <[pjacklam@online.no](mailto:pjacklam@online.no)>, 2011-.

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!

perl v5.34.0

2023-11-23

Math::BigInt(3perl)