



- [Preferences](#)

## Manual

- [Overview](#)
- [Tutorials](#)
- [FAQs](#)
- [History / Changes](#)
- [License](#)

## Reference

- [Language](#)
- [Functions](#)
- [Operators](#)
- [Special Variables](#)
- [Pragmas](#)
- [Utilities](#)
- [Internals](#)
- [Platform Specific](#)

## Modules

- [A](#) • [B](#) • [C](#) • [D](#) • [E](#)
- [F](#) • [G](#) • [H](#) • [I](#) • [L](#)
- [M](#) • [N](#) • [O](#) • [P](#) • [S](#)
- [T](#) • [U](#) • [X](#)

# Math::BigInt

Perl 5 version 12.2 documentation

[Go to top](#) • [Download PDF](#)

[Show page index](#) • [Show recent pages](#)

[Home](#) > [Core modules \(M\)](#) > Math::BigInt

Recently read

# Math::BigInt

- [NAME](#)
- [SYNOPSIS](#)
- [DESCRIPTION](#)

- METHODS

- [config\(\)](#)
- [accuracy\(\)](#)
- [precision\(\)](#)
- [brsft\(\)](#)
- [new\(\)](#)
- [from oct\(\)](#)
- [from hex\(\)](#)
- [from bin\(\)](#)
- [bnan\(\)](#)
- [bzero\(\)](#)
- [binf\(\)](#)
- [bone\(\)](#)
- [is\\_one\(\)/is\\_zero\(\)/is\\_nan\(\)/is\\_inf\(\)](#)
- [is\\_pos\(\)/is\\_neg\(\)/is\\_positive\(\)/is\\_negative\(\)](#)
- [is\\_odd\(\)/is\\_even\(\)/is\\_int\(\)](#)
- [bcmp\(\)](#)
- [bacmp\(\)](#)
- [sign\(\)](#)
- [digit\(\)](#)
- [bneg\(\)](#)
- [babs\(\)](#)
- [bnorm\(\)](#)
- [bnot\(\)](#)
- [binc\(\)](#)
- [bdec\(\)](#)
- [badd\(\)](#)
- [bsub\(\)](#)
- [bmul\(\)](#)
- [bmuladd\(\)](#)
- [bdiv\(\)](#)
- [bmod\(\)](#)
- [bmodinv\(\)](#)
- [bmodpow\(\)](#)
- [bpow\(\)](#)
- [blog\(\)](#)
- [bexp\(\)](#)
- [bnok\(\)](#)
- [bpi\(\)](#)
- [bcos\(\)](#)
- [bsin\(\)](#)
- [batan2\(\)](#)
- [batan\(\)](#)
- [blsft\(\)](#)
- [brsft\(\)](#)
- [band\(\)](#)
- [bior\(\)](#)
- [bxor\(\)](#)
- [bnot\(\)](#)
- [bsqrt\(\)](#)
- [broot\(\)](#)
- [bfac\(\)](#)

- [round\(\)](#)
- [bround\(\)](#)
- [bfround\(\)](#)
- [bfloor\(\)](#)
- [bceil\(\)](#)
- [bgcd\(\)](#)
- [blcm\(\)](#)
- [exponent\(\)](#)
- [mantissa\(\)](#)
- [parts\(\)](#)
- [copy\(\)](#)
- [as\\_int\(\)/as\\_number\(\)](#)
- [bstr\(\)](#)
- [bsstr\(\)](#)
- [as\\_hex\(\)](#)
- [as\\_bin\(\)](#)
- [as\\_oct\(\)](#)
- [numify\(\)](#)
- [modify\(\)](#)
- [upgrade\(\)/downgrade\(\)](#)
- [div\\_scale\(\)](#)
- [round\\_mode\(\)](#)
- [ACCURACY and PRECISION](#)
  - [Precision P](#)
  - [Accuracy A](#)
  - [Fallback F](#)
  - [Rounding mode R](#)
- [Infinity and Not a Number](#)
- [INTERNALS](#)
  - [MATH LIBRARY](#)
  - [SIGN](#)
  - [mantissa\(\), exponent\(\) and parts\(\)](#)
- [EXAMPLES](#)
- [Autocreating constants](#)
- [PERFORMANCE](#)
  - [Alternative math libraries](#)
  - [SUBCLASSING](#)
- [Subclassing Math::BigInt](#)
- [UPGRADING](#)
  - [Auto-upgrade](#)
- [EXPORTS](#)
- [CAVEATS](#)
- [LICENSE](#)
- [SEE ALSO](#)
- [AUTHORS](#)

# NAME

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

# SYNOPSIS

```
1.  use Math::BigInt;
2.
3.  # or make it faster with huge numbers: install (optional)
4.  # Math::BigInt::GMP and always use (it will fall back to
5.  # pure Perl if the GMP library is not installed):
6.  # (See also the L<MATH LIBRARY> section!)
7.
8.  # will warn if Math::BigInt::GMP cannot be found
9.  use Math::BigInt lib => 'GMP';
10.
11. # to suppress the warning use this:
12. # use Math::BigInt try => 'GMP';
13.
14. # dies if GMP cannot be loaded:
15. # use Math::BigInt only => 'GMP';
16.
17. my $str = '1234567890';
18. my @values = (64,74,18);
19. my $n = 1; my $sign = '-';
20.
21. # Number creation
22. my $x = Math::BigInt->new($str);          # defaults to 0
23. my $y = $x->copy();                      # make a true copy
24. my $nan = Math::BigInt->bnan();           # create a NotANumber
25. my $zero = Math::BigInt->bzero();         # create a +0
26. my $inf = Math::BigInt->binf();           # create a +inf
27. my $inf = Math::BigInt->binf('-');        # create a -inf
28. my $one = Math::BigInt->bone();           # create a +1
29. my $mone = Math::BigInt->bone('-');       # create a -1
30.
31. my $pi = Math::BigInt->bpi();             # returns '3'
32.                                         # see Math::BigFloat::bpi()
33.
34. $h = Math::BigInt->new('0x123');          # from hexadecimal
35. $b = Math::BigInt->new('0b101');         # from binary
36. $o = Math::BigInt->from_oct('0101');     # from octal
37.
38. # Testing (don't modify their arguments)
39. # (return true if the condition is met, otherwise false)
40.
41. $x->is_zero();                          # if $x is +0
42. $x->is_nan();                           # if $x is NaN
43. $x->is_one();                           # if $x is +1
44. $x->is_one('-');                        # if $x is -1
45. $x->is_odd();                           # if $x is odd
46. $x->is_even();                          # if $x is even
47. $x->is_pos();                           # if $x >= 0
48. $x->is_neg();                           # if $x < 0
49. $x->is_inf($sign);                     # if $x is +inf, or -inf (sign is default '+')
50. $x->is_int();                           # if $x is an integer (not a float)
51.
52. # comparing and digit/sign extraction
53. $x->bcmp($y);                           # compare numbers (undef,<0,=0,>0)
54. $x->bacmp($y);                          # compare absolutely (undef,<0,=0,>0)
55. $x->sign();                             # return the sign, either +, - or NaN
56. $x->digit($n);                          # return the nth digit, counting from right
57. $x->digit(-$n);                         # return the nth digit, counting from left
58.
59. # The following all modify their first argument. If you want to preserve
60. # $x, use $z = $x->copy()->bXXX($y); See under L<CAVEATS> for why this is
61. # necessary when mixing $a = $b assignments with non-overloaded math.
62.
```

```

63. $x->bzero();           # set $x to 0
64. $x->bnan();           # set $x to NaN
65. $x->bone();           # set $x to +1
66. $x->bone('-');        # set $x to -1
67. $x->binf();           # set $x to inf
68. $x->binf('-');        # set $x to -inf
69.
70. $x->bneg();           # negation
71. $x->babs();           # absolute value
72. $x->bnorm();          # normalize (no-op in BigInt)
73. $x->bnot();           # two's complement (bit wise not)
74. $x->binc();           # increment $x by 1
75. $x->bdec();           # decrement $x by 1
76.
77. $x->badd($y);          # addition (add $y to $x)
78. $x->bsub($y);          # subtraction (subtract $y from $x)
79. $x->bmul($y);          # multiplication (multiply $x by $y)
80. $x->bdiv($y);          # divide, set $x to quotient
81.                       # return (quo,rem) or quo if scalar
82.
83. $x->bmuladd($y,$z);    # $x = $x * $y + $z
84.
85. $x->bmod($y);          # modulus (x % y)
86. $x->bmodpow($exp,$mod); # modular exponentiation (($num**$exp) % $mod))
87. $x->bmodinv($mod);     # the inverse of $x in the given modulus $mod
88.
89. $x->bpow($y);          # power of arguments (x ** y)
90. $x->blsft($y);         # left shift in base 2
91. $x->brsft($y);         # right shift in base 2
92.                       # returns (quo,rem) or quo if in scalar context
93. $x->blsft($y,$n);      # left shift by $y places in base $n
94. $x->brsft($y,$n);      # right shift by $y places in base $n
95.                       # returns (quo,rem) or quo if in scalar context
96.
97. $x->band($y);          # bitwise and
98. $x->bior($y);          # bitwise inclusive or
99. $x->bxor($y);          # bitwise exclusive or
100. $x->bnot();           # bitwise not (two's complement)
101.
102. $x->bsqrt();           # calculate square-root
103. $x->broot($y);         # $y'th root of $x (e.g. $y == 3 => cubic root)
104. $x->bfac();            # factorial of $x (1*2*3*4*..$x)
105.
106. $x->bnok($y);          # x over y (binomial coefficient n over k)
107.
108. $x->blog();            # logarithm of $x to base e (Euler's number)
109. $x->blog($base);       # logarithm of $x to base $base (f.i. 2)
110. $x->bexp();            # calculate e ** $x where e is Euler's number
111.
112. $x->round($A,$P,$mode); # round to accuracy or precision using mode $mode
113. $x->bround($n);         # accuracy: preserve $n digits
114. $x->bround($n);         # $n > 0: round $nth digits,
115.                       # $n < 0: round to the $nth digit after the
116.                       # dot, no-op for BigInts
117.
118. # The following do not modify their arguments in BigInt (are no-ops),
119. # but do so in BigFloat:
120.
121. $x->bffloor();          # return integer less or equal than $x
122. $x->bceil();            # return integer greater or equal than $x
123.
124. # The following do not modify their arguments:
125.
126. # greatest common divisor (no 00 style)
127. my $gcd = Math::BigInt::bgcd(@values);
128. # lowest common multiplicator (no 00 style)

```

```

129. my $lcm = Math::BigInt::blcm(@values);
130.
131. $x->length();           # return number of digits in number
132. ($xl,$f) = $x->length(); # length of number and length of fraction part,
133.                          # latter is always 0 digits long for BigInts
134.
135. $x->exponent();          # return exponent as BigInt
136. $x->mantissa();          # return (signed) mantissa as BigInt
137. $x->parts();             # return (mantissa,exponent) as BigInt
138. $x->copy();              # make a true copy of $x (unlike $y = $x;)
139. $x->as_int();            # return as BigInt (in BigInt: same as copy())
140. $x->numify();            # return as scalar (might overflow!)
141.
142. # conversation to string (do not modify their argument)
143. $x->bstr();              # normalized string (e.g. '3')
144. $x->bsstr();            # norm. string in scientific notation (e.g. '3E0')
145. $x->as_hex();           # as signed hexadecimal string with prefixed 0x
146. $x->as_bin();           # as signed binary string with prefixed 0b
147. $x->as_oct();           # as signed octal string with prefixed 0
148.
149.
150. # precision and accuracy (see section about rounding for more)
151. $x->precision();         # return P of $x (or global, if P of $x undef)
152. $x->precision($n);       # set P of $x to $n
153. $x->accuracy();          # return A of $x (or global, if A of $x undef)
154. $x->accuracy($n);        # set A $x to $n
155.
156. # Global methods
157. Math::BigInt->precision(); # get/set global P for all BigInt objects
158. Math::BigInt->accuracy();  # get/set global A for all BigInt objects
159. Math::BigInt->round_mode(); # get/set global round mode, one of
160.                          # 'even', 'odd', '+inf', '-inf', 'zero', 'trunc' or 'common'
161. Math::BigInt->config();    # return hash containing configuration

```

## DESCRIPTION

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

```
1. $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 conversation to float. Quote your input if you want BigInt to see all the digits:

```

1. $x = Math::BigInt->new(12345678890123456789); # bad
2. $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":

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

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

```
1.          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:

```
1.          $zero = Math::BigInt->bzero();
2.          $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()

```
1.          use Data::Dumper;
2.
3.          print Dumper ( Math::BigInt->config() );
4.          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.

1.	key	Description
2.		Example
3.	=====	=====
4.	lib	Name of the low-level math library
5.		Math::BigInt::Calc
6.	lib_version	Version of low-level math library (see 'lib')
7.		0.30
8.	class	The class name of config() you just called
9.		Math::BigInt
10.	upgrade	To which class math operations might be upgraded
11.		Math::BigFloat
12.	downgrade	To which class math operations might be downgraded
13.		undef
14.	precision	Global precision
15.		undef
16.	accuracy	Global accuracy
17.		undef
18.	round_mode	Global round mode
19.		even
20.	version	version number of the class you used

21.		1.61
22.	div_scale	Fallback accuracy for div
23.		40
24.	trap_nan	If true, traps creation of NaN via croak()
25.		1
26.	trap_inf	If true, traps creation of +inf/-inf via croak()
27.		1

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

1.	trap_inf trap_nan
2.	upgrade downgrade precision accuracy round_mode div_scale

Example:

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

## accuracy()

1.	<code>\$x-&gt;accuracy(5);</code>	<code># local for \$x</code>
2.	<code>CLASS-&gt;accuracy(5);</code>	<code># global for all members of CLASS</code>
3.		<code># Note: This also applies to new()!</code>
4.		
5.	<code>\$A = \$x-&gt;accuracy();</code>	<code># read out accuracy that affects \$x</code>
6.	<code>\$A = CLASS-&gt;accuracy();</code>	<code># read out global accuracy</code>

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 `bround()` or by passing the desired accuracy to the math operation as additional parameter:

```
1.      my $x = Math::BigInt->new(30000);
2.      my $y = Math::BigInt->new(7);
3.      print scalar $x->copy()->bdiv($y, 2);      # print 4300
4.      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:

1.	<code>\$x-&gt;accuracy(undef);</code>
2.	<code>Math::BigInt-&gt;accuracy(undef);</code>

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`:

1.	<code>\$y = Math::BigInt-&gt;new(1234567);</code>	<code># unrounded</code>
2.	<code>print Math::BigInt-&gt;accuracy(4), "\n";</code>	<code># set 4, print 4</code>
3.	<code>\$x = Math::BigInt-&gt;new(123456);</code>	<code># \$x will be automatically rounded!</code>
4.	<code>print "\$x \$y\n";</code>	<code># '123500 1234567'</code>
5.	<code>print \$x-&gt;accuracy(), "\n";</code>	<code># will be 4</code>
6.	<code>print \$y-&gt;accuracy(), "\n";</code>	<code># also 4, since global is 4</code>
7.	<code>print Math::BigInt-&gt;accuracy(5), "\n";</code>	<code># set to 5, print 5</code>
8.	<code>print \$x-&gt;accuracy(), "\n";</code>	<code># still 4</code>
9.	<code>print \$y-&gt;accuracy(), "\n";</code>	<code># 5, since global is 5</code>



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`.

## precision()

```
1.      $x->precision(-2);      # local for $x, round at the second digit right of the dot
2.      $x->precision(2);       # ditto, round at the second digit left of the dot
3.
4.      CLASS->precision(5);     # Global for all members of CLASS
5.      # This also applies to new()!
6.      CLASS->precision(-5);    # ditto
7.
8.      $P = CLASS->precision();  # read out global precision
9.      $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:

```
1.      $x->precision(undef);
2.      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 precision that will be in effect for `$x`:

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

```
1.      $x->brsft($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:

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

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

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

```
2.          print $x->brsft(1);
```

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

## new()

```
1.          $x = Math::BigInt->new($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()

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

## from\_hex()

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

## from\_bin()

```
1.          $x = Math::BigInt->from_bin("0x10011"); # input is binary
```

## bnan()

```
1.          $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:

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

## bzero()

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

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

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

## binf()

```
1.          $x = Math::BigInt->binf($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:

```
1.          $x->binf();
2.          $x->binf('-');
```

## bone()

```
1.          $x = Math::BigInt->binf($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:

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

## **is\_one()/is\_zero()/is\_nan()/is\_inf()**

```
1.          $x->is_zero();        # true if arg is +0
2.          $x->is_nan();         # true if arg is NaN
3.          $x->is_one();         # true if arg is +1
4.          $x->is_one('-');      # true if arg is -1
5.          $x->is_inf();         # true if +inf
6.          $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:

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

## **is\_pos()/is\_neg()/is\_positive()/is\_negative()**

```
1.          $x->is_pos();         # true if > 0
2.          $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()**

```
1.          $x->is_odd();         # true if odd, false for even
2.          $x->is_even();        # true if even, false for odd
3.          $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()**

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

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

## **bacmp()**

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

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

## sign()

```
1.          $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:

```
1.          $x->babs();           # '+'
2.          $x->babs()->bneg();    # '-'
3.          $x->bnan();           # 'NaN'
4.          $x->binf();           # '+inf'
5.          $x->binf('-');        # '-inf'
```

## digit()

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

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

## bneg()

```
1.          $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()

```
1.          $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.

## bnorm()

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

## bnot()

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

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

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

but faster.

## binc()

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

## **bdec()**

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

## **badd()**

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

## 

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

## **bmul()**

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

## **bmuladd()**

```
1.          $x->bmuladd($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()**

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

## **bmod()**

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

## **bmodinv()**

```
1.          num->bmodinv($mod);   # modular inverse
```

Returns the inverse of \$num in the given modulus \$mod . 'NaN ' is returned unless \$num is relatively prime to \$mod ,  
i.e. unless `bgcd($num, $mod)==1` .

## **bmodpow()**

```
1.          $num->bmodpow($exp,$mod);   # modular exponentation
2.          # ($num**$exp % $mod)
```

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

```
1.          $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.

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

is exactly equivalent to

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

## bpow()

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

## blog()

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

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

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

## bexp()

```
1.          $x->bexp($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()

```
1.          $x->bnok($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:

```
1.          ( n )      n!  
2.          | - |  =  -----  
3.          ( k )      k!(n-k)!
```

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

## bpi()

```
1.          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:

```

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

```

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

## bcos()

```

1.      my $x = Math::BigInt->new(1);
2.      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()

```

1.      my $x = Math::BigInt->new(1);
2.      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()

```

1.      my $x = Math::BigInt->new(1);
2.      my $y = Math::BigInt->new(1);
3.      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()

```

1.      my $x = Math::BigFloat->new(0.5);
2.      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()

```

1.      $x->blsft($y);          # left shift in base 2

```

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

## **brsft()**

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

## **band()**

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

## **bior()**

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

## **bxor()**

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

## **bnot()**

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

## **bsqrt()**

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

## **broot()**

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

Calculates the N'th root of \$x .

## **bfac()**

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

## **round()**

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

Round \$x to accuracy \$A or precision \$P using the round mode \$round\_mode .

## **bround()**

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

## **bfround()**



```
1.          $x->bround($N);
```

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

Examples:

	Input	N	Result
1.	123456.123456	3	123500
2.	123456.123456	2	123450
3.	123456.123456	-2	123456.12
4.	123456.123456	-3	123456.123

## bfloor()

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

Set  $x$  to the integer less or equal than  $x$ . This is a no-op in BigInt, but does change  $x$  in BigFloat.

## bceil()

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

Set  $x$  to the integer greater or equal than  $x$ . This is a no-op in BigInt, but does change  $x$  in BigFloat.

## bgcd()

```
1.          bgcd(@values);          # greatest common divisor (no 00 style)
```

## blcm()

```
1.          blcm(@values);          # lowest common multiplicator (no 00 style)
```

head2 length()

```
1.          $x->length();
2.          ($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()

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

Return the exponent of  $x$  as BigInt.

## mantissa()

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

Return the signed mantissa of  $x$  as BigInt.

## parts()

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

## copy()

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

## as\_int()/as\_number()

```
1.          $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()

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

Returns a normalized string representation of \$x .

## bsstr()

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

## as\_hex()

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

## as\_bin()

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

## as\_oct()

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

## numify()

```
1.          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()

```
1.          $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. This is used for instance by [bignum](#). The defaults are "", thus the following operation will create a BigInt, not a BigFloat:

```
1.          my $i = Math::BigInt->new(123);
2.          my $f = Math::BigFloat->new('123.1');
3.
4.          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:

1.	Initial value	P	A	Result	String
----	---------------	---	---	--------	--------

2.	-----			
3.	1234.01	-3	1000	1000
4.	1234	-2	1200	1200
5.	1234.5	-1	1230	1230
6.	1234.001	1	1234	1234.0
7.	1234.01	0	1234	1234
8.	1234.01	2	1234.01	1234.01
9.	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:

1.	Initial value	P	A	Result	String
2.	-----				-----
3.	1234.01		3	1230	1230
4.	1234.01		6	1234.01	1234.01
5.	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**

1. \* `ffround($p)` is able to round to \$p number of digits after the decimal
2. point
3. \* otherwise P is unused

- **Accuracy (significant digits)**

1. \* `fround($a)` rounds to \$a significant digits
2. \* only `fdiv()` and `fsqrt()` take A as (optional) parameter
3. + other operations simply create the same number (`fneg` etc), or more (`fmul`)
4. of digits
5. + rounding/truncating is only done when explicitly calling one of `fround`
6. or `ffround`, and never for `BigInt` (not implemented)
7. \* `fsqrt()` simply hands its accuracy argument over to `fdiv`.
8. \* the documentation and the comment in the code indicate two different ways
9. on how `fdiv()` determines the maximum number of digits it should calculate,
10. and the actual code does yet another thing
11. POD:
12. `max($Math::BigFloat::div_scale, length(dividend)+length(divisor))`
13. Comment:
14. result has at most `max(scale, length(dividend), length(divisor))` digits
15. Actual code:
16. `scale = max(scale, length(dividend)-1, length(divisor)-1);`
17. `scale += length(divisor) - length(dividend);`
18. So for `lx = 3, ly = 9, scale = 10`, scale will actually be 16 (`10+9-3`).
19. Actually, the 'difference' added to the scale is calculated from the
20. number of "significant digits" in dividend and divisor, which is derived
21. by looking at the length of the mantissa. Which is wrong, since it includes
22. the + sign (oops) and actually gets 2 for `'+100'` and 4 for `'+101'`. Oops
23. again. Thus `124/3` with `div_scale=1` will get you `'41.3'` based on the strange
24. assumption that 124 has 3 significant digits, while `120/7` will get you
25. `'17'`, not `'17.1'` since 120 is thought to have 2 significant digits.

26. The rounding after the division then uses the remainder and \$y to determine
27. whether it must round up or down.
28. ? I have no idea which is the right way. That's why I used a slightly more
29. ? simple scheme and tweaked the few failing testcases to match it.

This is how it works now:

### • Setting/Accessing

1. \* You can set the A global via `C<< Math::BigInt->accuracy() >>` or
2. `C<< Math::BigFloat->accuracy() >>` or whatever class you are using.
3. \* You can also set P globally by using `C<< Math::SomeClass->precision() >>`
4. likewise.
5. \* Globals are classwide, and not inherited by subclasses.
6. \* to undefine A, use `C<< Math::SomeClass->accuracy(undef); >>`
7. \* to undefine P, use `C<< Math::SomeClass->precision(undef); >>`
8. \* Setting `C<< Math::SomeClass->accuracy() >>` clears automatically
9. `C<< Math::SomeClass->precision() >>`, and vice versa.
10. \* To be valid, A must be  $> 0$ , P can have any value.
11. \* If P is negative, this means round to the P'th place to the right of the
12. decimal point; positive values mean to the left of the decimal point.
13. P of 0 means round to integer.
14. \* to find out the current global A, use `C<< Math::SomeClass->accuracy() >>`
15. \* to find out the current global P, use `C<< Math::SomeClass->precision() >>`
16. \* use `C<< $x->accuracy() >>` respective `C<< $x->precision() >>` for the local
17. setting of `C<< $x >>`.
18. \* Please note that `C<< $x->accuracy() >>` respective `C<< $x->precision() >>`
19. return eventually defined global A or P, when `C<< $x >>`'s A or P is not
20. set.

### • Creating numbers

1. \* When you create a number, you can give the desired A or P via:
2. `$x = Math::BigInt->new($number,$A,$P);`
3. \* Only one of A or P can be defined, otherwise the result is NaN
4. \* If no A or P is give (`$x = Math::BigInt->new($number)` form), then the
5. globals (if set) will be used. Thus changing the global defaults later on
6. will not change the A or P of previously created numbers (i.e., A and P of
7. \$x will be what was in effect when \$x was created)
8. \* If given undef for A and P, B<no> rounding will occur, and the globals will
9. B<not> be used. This is used by subclasses to create numbers without
10. suffering rounding in the parent. Thus a subclass is able to have its own
11. globals enforced upon creation of a number by using
12. `C<< $x = Math::BigInt->new($number,undef,undef) >>`:
- 13.
14. `use Math::BigInt::SomeSubclass;`
15. `use Math::BigInt;`
- 16.
17. `Math::BigInt->accuracy(2);`
18. `Math::BigInt::SomeSubClass->accuracy(3);`
19. `$x = Math::BigInt::SomeSubClass->new(1234);`
- 20.
21. \$x is now 1230, and not 1200. A subclass might choose to implement
22. this otherwise, e.g. falling back to the parent's A and P.

### • Usage

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

### • Precedence

```

1.  * It only makes sense that a number has only one of A or P at a time.
2.  If you set either A or P on one object, or globally, the other one will
3.  be automatically cleared.
4.  * If two objects are involved in an operation, and one of them has A in
5.  effect, and the other P, this results in an error (NaN).
6.  * A takes precedence over P (Hint: A comes before P).
7.  If neither of them is defined, nothing is used, i.e. the result will have
8.  as many digits as it can (with an exception for fdiv/fsqrt) and will not
9.  be rounded.
10. * There is another setting for fdiv() (and thus for fsqrt()). If neither of
11. A or P is defined, fdiv() will use a fallback (F) of $div_scale digits.
12. If either the dividend's or the divisor's mantissa has more digits than
13. the value of F, the higher value will be used instead of F.
14. This is to limit the digits (A) of the result (just consider what would
15. happen with unlimited A and P in the case of 1/3 :-))
16. * fdiv will calculate (at least) 4 more digits than required (determined by
17. A, P or F), and, if F is not used, round the result
18. (this will still fail in the case of a result like 0.12345000000001 with A
19. or P of 5, but this can not be helped - or can it?)
20. * Thus you can have the math done by on Math::Big* class in two modi:
21. + never round (this is the default):
22.   This is done by setting A and P to undef. No math operation
23.   will round the result, with fdiv() and fsqrt() as exceptions to guard
24.   against overflows. You must explicitly call bround(), bround() or
25.   round() (the latter with parameters).
26.   Note: Once you have rounded a number, the settings will 'stick' on it
27.   and 'infect' all other numbers engaged in math operations with it, since
28.   local settings have the highest precedence. So, to get SaferRound[tm],
29.   use a copy() before rounding like this:
30.
31.     $x = Math::BigFloat->new(12.34);
32.     $y = Math::BigFloat->new(98.76);
33.     $z = $x * $y;                                     # 1218.6984
34.     print $x->copy()->fround(3);                       # 12.3 (but A is now 3!)
35.     $z = $x * $y;                                     # still 1218.6984, without
36.                                                         # copy would have been 1210!
37.
38. + round after each op:
39.   After each single operation (except for testing like is_zero()), the
40.   method round() is called and the result is rounded appropriately. By
41.   setting proper values for A and P, you can have all-the-same-A or
42.   all-the-same-P modes. For example, Math::Currency might set A to undef,
43.   and P to -2, globally.
44.
45. ?Maybe an extra option that forbids local A & P settings would be in order,
46. ?so that intermediate rounding does not 'poison' further math?

```

## • Overriding globals

```

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

```

- **Local settings**

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

- **Rounding**

1. \* the rounding routines will use the respective global or local settings.
2. `fround()/bround()` is for accuracy rounding, while `ffround()/bfround()`
3. is for precision
4. \* the two rounding functions take as the second parameter one of the
5. following rounding modes (R):
6. 'even', 'odd', '+inf', '-inf', 'zero', 'trunc', 'common'
7. \* you can set/get the global R by using `C<< Math::SomeClass->round_mode() >>`
8. or by setting `C<< $Math::SomeClass::round_mode >>`
9. \* after each operation, `C<< $result->round() >>` is called, and the result may
10. eventually be rounded (that is, if A or P were set either locally,
11. globally or as parameter to the operation)
12. \* to manually round a number, call `C<< $x->round($A,$P,$round_mode); >>`
13. this will round the number by using the appropriate rounding function
14. and then normalize it.
15. \* rounding modifies the local settings of the number:
- 16.
17. `$x = Math::BigFloat->new(123.456);`
18. `$x->accuracy(5);`
19. `$x->bround(4);`
- 20.
21. Here 4 takes precedence over 5, so 123.5 is the result and `$x->accuracy()`
22. will be 4 from now on.

- **Default values**

1. \* R: 'even'
2. \* F: 40
3. \* A: undef
4. \* P: undef

- **Remarks**

1. \* The defaults are set up so that the new code gives the same results as
2. the old code (except in a few cases on `fdiv`):
3. + Both A and P are undefined and thus will not be used for rounding
4. after each operation.
5. + `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`.

1. `te@linux:~> perl -wle 'print 2 ** 3333'`
2. `inf`
3. `te@linux:~> perl -wle 'print 2 ** 3333 == 2 ** 3333'`
4. `1`
5. `te@linux:~> perl -wle 'print oct(2 ** 3333)'`
6. `0`
7. `te@linux:~> perl -wle 'print hex(2 ** 3333)'`



```
8.      Illegal hexadecimal digit 'i' ignored at -e line 1.
9.      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` might return wrong values.

- **`log(-inf)`**

[`log\(-inf\)`](#) is highly weird. Since  $\log(-x) = \pi i + \log(x)$ , then  $\log(-\infty) = \pi i + \infty$ . 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:

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

You can change this backend library by using:

```
1.      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:

```
1.      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`:

```
1.      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:

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

`($m,$e) = $x->parts()` is just a shortcut that gives 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

```
1.      use Math::BigInt;
2.
3.      sub bint { Math::BigInt->new(shift); }
4.
5.      $x = Math::BigInt->bstr("1234")           # string "1234"
6.      $x = "$x";                               # same as bstr()
7.      $x = Math::BigInt->bneg("1234");          # BigInt "-1234"
8.      $x = Math::BigInt->babs("-12345");         # BigInt "12345"
9.      $x = Math::BigInt->bnorm("-0.00");        # BigInt "0"
10.     $x = bint(1) + bint(2);                  # BigInt "3"
```

```

11.    $x = bint(1) + "2";           # ditto (auto-BigIntify of "2")
12.    $x = bint(1);                 # BigInt "1"
13.    $x = $x + 5 / 2;               # BigInt "3"
14.    $x = $x ** 3;                 # BigInt "27"
15.    $x *= 2;                       # BigInt "54"
16.    $x = Math::BigInt->new(0);     # BigInt "0"
17.    $x--;                          # BigInt "-1"
18.    $x = Math::BigInt->badd(4,5)    # BigInt "9"
19.    print $x->bsstr();              # 9e+0

```

Examples for rounding:

```

1.    use Math::BigFloat;
2.    use Test;
3.
4.    $x = Math::BigFloat->new(123.4567);
5.    $y = Math::BigFloat->new(123.456789);
6.    Math::BigFloat->accuracy(4);     # no more A than 4
7.
8.    ok ($x->copy()->fround(),123.4);  # even rounding
9.    print $x->copy()->fround(),"\n";  # 123.4
10.   Math::BigFloat->round_mode('odd'); # round to odd
11.   print $x->copy()->fround(),"\n";  # 123.5
12.   Math::BigFloat->accuracy(5);     # no more A than 5
13.   Math::BigFloat->round_mode('odd'); # round to odd
14.   print $x->copy()->fround(),"\n";  # 123.46
15.   $y = $x->copy()->fround(4)," \n"; # A = 4: 123.4
16.   print "$y, ", $y->accuracy()," \n"; # 123.4, 4
17.
18.   Math::BigFloat->accuracy(undef);   # A not important now
19.   Math::BigFloat->precision(2);      # P important
20.   print $x->copy()->bnorm(),"\n";    # 123.46
21.   print $x->copy()->fround(),"\n";   # 123.46

```

Examples for converting:

```

1.    my $x = Math::BigInt->new('0b1'. '01' x 123);
2.    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,

```

1.    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

```

1.    use Math::BigInt qw/:constant/;
2.
3.    $x = 1234567890123456789012345678901234567890
4.        + 123456789123456789;
5.    $y = '1234567890123456789012345678901234567890'
6.        + '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:

```
1.      use Math::BigInt;  
2.  
3.      $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 BigInt, which results in an truncated result or a NaN.

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

```
1.      use Math::BigInt ':constant';  
2.  
3.      print ref(123e2), "\n";  
4.      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.:

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

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

```
1.  $upgrade = undef;
2.  $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

```
1.      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:

```
1.      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`:

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

As a shortcut, you can use the module `bignum`:

```
1.      use bignum;
```

Also good for oneliners:

```
1.          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 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:

```
1.          bgcd
2.          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:

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

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

```
1.          use Test;
2.          BEGIN { plan tests => 1 }
```

```

3.      use Math::BigInt;
4.
5.      my $x = new Math::BigInt 3*3;
6.      my $y = new Math::BigInt 3*3;
7.
8.      ok ($x,3*3);
9.      print "$x eq 9" if $x eq $y;
10.     print "$x eq 9" if $x eq '9';
11.     print "$x eq 9" if $x eq 3*3;

```

Additionally, the following still works:

```

1.      print "$x == 9" if $x == $y;
2.      print "$x == 9" if $x == 9;
3.      print "$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:

```

1.      use Test;
2.      BEGIN { plan tests => 3 }
3.      use Math::BigInt;
4.
5.      $x = Math::BigInt->new('1e56'); $y = 1e56;
6.      ok ($x,$y);                      # will fail
7.      ok ($x->bsstr(),$y);              # okay
8.      $y = Math::BigInt->new($y);
9.      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:

```

1.      $x = Math::BigInt->new(123);
2.      $y = int($x);                      # BigInt 123
3.      $x = Math::BigFloat->new(123.45);
4.      $y = int($x);                      # BigInt 123

```

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

```

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

```

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

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

```

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

```

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

```

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

```

- **length**

The following will probably not do what you expect:

```
1.      $c = Math::BigInt->new(123);
2.      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:

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

- **bdiv**

The following will probably not do what you expect:

```
1.      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

```
1.      print $c / 10000,"\\n";
2.      print scalar $c->bdiv(10000),"\\n"; # or if you want to modify $c
```

instead.

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 nonzero) always has the same sign as the second operand; so, for example,

```
1.      1 / 4  => ( 0, 1)
2.      1 / -4 => (-1, -3)
3.     -3 / 4  => (-1, 1)
4.     -3 / -4 => ( 0, -3)
5.    -11 / 2  => (-5, 1)
6.     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 perl op manpage), and the equation

```
1.      $x == ($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:

1.	A / B = C,	R so that C *	B +	R =	A
2.	=====				=====
3.	5 / 8 = 0,	5	0 * 8 +	5 =	5
4.	0 / 8 = 0,	0	0 * 8 +	0 =	0
5.	0 / inf = 0,	0	0 * inf +	0 =	0



6.	0 / -inf = 0,	0	0 * -inf + 0 = 0
7.	5 / inf = 0,	5	0 * inf + 5 = 5
8.	5 / -inf = 0,	5	0 * -inf + 5 = 5
9.	-5 / inf = 0,	-5	0 * inf + -5 = -5
10.	-5 / -inf = 0,	-5	0 * -inf + -5 = -5
11.	inf / 5 = inf,	0	inf * 5 + 0 = inf
12.	-inf / 5 = -inf,	0	-inf * 5 + 0 = -inf
13.	inf / -5 = -inf,	0	-inf * -5 + 0 = inf
14.	-inf / -5 = inf,	0	inf * -5 + 0 = -inf
15.	5 / 5 = 1,	0	1 * 5 + 0 = 5
16.	-5 / -5 = 1,	0	1 * -5 + 0 = -5
17.	inf / inf = 1,	0	1 * inf + 0 = inf
18.	-inf / -inf = 1,	0	1 * -inf + 0 = -inf
19.	inf / -inf = -1,	0	-1 * -inf + 0 = inf
20.	-inf / inf = -1,	0	1 * -inf + 0 = -inf
21.	8 / 0 = inf,	8	inf * 0 + 8 = 8
22.	inf / 0 = inf,	inf	inf * 0 + inf = inf
23.	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
1.	=====	=====
3.	-inf / 0 = -inf,	-inf * 0 + inf = -inf
4.	-8 / 0 = -inf,	-inf * 0 + 8 = -8

## • Modifying and =

Beware of:

```
1. $x = Math::BigFloat->new(5);
2. $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:

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

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

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

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

```
1. $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:

```
1. print bpow($x,$i),"\n";      # modify $x
2. print $x->bpow($i),"\n";     # ditto
3. print $x **= $i,"\n";       # the same
```

```
4.          print $x ** $i, "\n";          # leave $x alone
```

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

- **Overloading -`$x`**

The following:

```
1.          $x = -$x;
```

is slower than

```
1.          $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:

```
1.          $float = 5.0 + 2;
2.          $float = 2 + 5.0;
3.          $float = 5 / 2;
```

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

```
1.          use Math::BigInt;
2.          use Math::BigFloat;
3.
4.          $mbf = Math::BigFloat->new(5);
5.          $mbi2 = Math::BigInteger->new(5);
6.          $mbi = Math::BigInteger->new(2);
7.
8.                                     # what actually gets called:
9.          $float = $mbf + $mbi;      # $mbf->badd()
10.         $float = $mbf / $mbi;      # $mbf->bdiv()
11.         $integer = $mbi + $mbf;    # $mbi->badd()
12.         $integer = $mbi2 / $mbi;   # $mbi2->bdiv()
13.         $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 operand 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()`:

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

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

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

Beware also of the order of more complicated expressions like:

```
1.          $integer = ($mbi2 + $mbi) / $mbf;                # int / float => int
2.          $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:

```
1.      $float = 2 + $mbf;
2.      $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 your 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:

```
1.      $x = Math::BigFloat->new(12);
2.      Math::BigFloat->precision(0);
3.      Math::BigFloat->round_mode('even');
4.      print $x->copy->bsqrt(),"\n";          # 4
5.
6.      Math::BigFloat->precision(2);
7.      print $x->bsqrt(),"\n";              # 3.46
8.      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::BitVect](#), [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!

Page index

- [NAME](#)
- [SYNOPSIS](#)
- [DESCRIPTION](#)
- [METHODS](#)
  - [config\(\)](#)
  - [accuracy\(\)](#)
  - [precision\(\)](#)
  - [brsft\(\)](#)
  - [new\(\)](#)
  - [from\\_oct\(\)](#)
  - [from\\_hex\(\)](#)
  - [from\\_bin\(\)](#)
  - [bnan\(\)](#)
  - [bzero\(\)](#)
  - [binf\(\)](#)
  - [bone\(\)](#)
  - [is\\_one\(\)/is\\_zero\(\)/is\\_nan\(\)/is\\_inf\(\)](#)
  - [is\\_pos\(\)/is\\_neg\(\)/is\\_positive\(\)/is\\_negative\(\)](#)
  - [is\\_odd\(\)/is\\_even\(\)/is\\_int\(\)](#)
  - [bcmp\(\)](#)
  - [bacmp\(\)](#)
  - [sign\(\)](#)
  - [digit\(\)](#)
  - [bneg\(\)](#)
  - [babs\(\)](#)
  - [bnorm\(\)](#)
  - [bnot\(\)](#)
  - [binc\(\)](#)
  - [bdec\(\)](#)
  - [badd\(\)](#)
  - [bsub\(\)](#)
  - [bmul\(\)](#)
  - [bmuladd\(\)](#)
  - [bdiv\(\)](#)
  - [bmod\(\)](#)
  - [bmodinv\(\)](#)
  - [bmodpow\(\)](#)
  - [bpow\(\)](#)
  - [blog\(\)](#)
  - [bexp\(\)](#)
  - [bnok\(\)](#)
  - [bpi\(\)](#)
  - [bcos\(\)](#)
  - [bsin\(\)](#)
  - [batan2\(\)](#)
  - [batan\(\)](#)
  - [blsft\(\)](#)
  - [brsft\(\)](#)
  - [band\(\)](#)

- [bior\(\)](#)
- [bxor\(\)](#)
- [bnot\(\)](#)
- [bsqrt\(\)](#)
- [broot\(\)](#)
- [bfac\(\)](#)
- [round\(\)](#)
- [bround\(\)](#)
- [bfround\(\)](#)
- [bfloor\(\)](#)
- [bceil\(\)](#)
- [bgcd\(\)](#)
- [blcm\(\)](#)
- [exponent\(\)](#)
- [mantissa\(\)](#)
- [parts\(\)](#)
- [copy\(\)](#)
- [as\\_int\(\)/as\\_number\(\)](#)
- [bstr\(\)](#)
- [bsstr\(\)](#)
- [as\\_hex\(\)](#)
- [as\\_bin\(\)](#)
- [as\\_oct\(\)](#)
- [numify\(\)](#)
- [modify\(\)](#)
- [upgrade\(\)/downgrade\(\)](#)
- [div\\_scale\(\)](#)
- [round\\_mode\(\)](#)
- [ACCURACY and PRECISION](#)
  - [Precision P](#)
  - [Accuracy A](#)
  - [Fallback F](#)
  - [Rounding mode R](#)
- [Infinity and Not a Number](#)
- [INTERNALS](#)
  - [MATH LIBRARY](#)
  - [SIGN](#)
  - [mantissa\(\), exponent\(\) and parts\(\)](#)
- [EXAMPLES](#)
- [Autocreating constants](#)
- [PERFORMANCE](#)
  - [Alternative math libraries](#)
  - [SUBCLASSING](#)
- [Subclassing Math::BigInt](#)
- [UPGRADING](#)
  - [Auto-upgrade](#)
- [EXPORTS](#)
- [CAVEATS](#)
- [LICENSE](#)
- [SEE ALSO](#)
- [AUTHORS](#)

perldoc.perl.org - Official documentation for the Perl programming language

Contact details

Site maintained by [Jon Allen \(JJ\)](#)

See the [project page](#) for more details

Documentation maintained by the [Perl 5 Porters](#)

- Manual
  - [Overview](#)
  - [Tutorials](#)
  - [FAQs](#)
  - [Changes](#)
- Reference
  - [Language](#)
  - [Functions](#)
  - [Operators](#)
  - [Variables](#)
- Modules
  - [Modules](#)
  - [Pragmas](#)
  - [Utilities](#)
- Misc
  - [License](#)
  - [Internals](#)
  - [Platforms](#)