



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::BigFloat.3perl'

\$ man Math::BigFloat.3perl

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

NAME

Math::BigFloat - Arbitrary size floating point math package

SYNOPSIS

```
use Math::BigFloat;

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

Math::BigFloat->accuracy(); # get class accuracy
Math::BigFloat->accuracy($n); # set class accuracy
Math::BigFloat->precision(); # get class precision
Math::BigFloat->precision($n); # set class precision
Math::BigFloat->round_mode(); # get class rounding mode
Math::BigFloat->round_mode($m); # set global round mode, must be one of
    # 'even', 'odd', '+inf', '-inf', 'zero',
    # 'trunc', or 'common'

Math::BigFloat->config("lib"); # name of backend math library

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

$x = Math::BigFloat->new($str); # defaults to 0
$x = Math::BigFloat->new('0x123'); # from hexadecimal
$x = Math::BigFloat->new('0b101'); # from binary
$x = Math::BigFloat->from_hex('0xc.afep+3'); # from hex
$x = Math::BigFloat->from_hex('cafe'); # ditto
$x = Math::BigFloat->from_oct('1.3267p-4'); # from octal
```

```

$x = Math::BigFloat->from_oct('0377');    # ditto
$x = Math::BigFloat->from_bin('0b1.1001p-4'); # from binary
$x = Math::BigFloat->from_bin('0101');    # ditto
$x = Math::BigFloat->from_ieee754($b, "binary64"); # from IEEE-754 bytes
$x = Math::BigFloat->bzero();              # create a +0
$x = Math::BigFloat->bone();               # create a +1
$x = Math::BigFloat->bone('-');            # create a -1
$x = Math::BigFloat->binf();               # create a +inf
$x = Math::BigFloat->binf('-');            # create a -inf
$x = Math::BigFloat->bnan();               # create a Not-A-Number
$x = Math::BigFloat->bpi();                # returns pi
$y = $x->copy();                           # make a copy (unlike $y = $x)
$y = $x->as_int();                           # return as 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->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 => 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 $nth digit left of dec. point
                  # $n < 0: round to $nth 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 BigInt
$x->exponent();    # return exponent as BigInt
$x->parts();       # return (mantissa,exponent) as 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->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

$x->to_ieee754($format); # to bytes encoded according to IEEE 754-2008

# Other conversion methods

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

```

DESCRIPTION

Math::BigFloat provides support for arbitrary precision floating point. Overloading is also provided for Perl operators.

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

```
$x = Math::BigFloat -> new('12_3.456_789_123_456_789E-2');
```

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

Input

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

? 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.
- ? For hexadecimal and binary numbers, the exponent must be separated from the significand (mantissa) by the letter "p" or "P", not "e" or "E" as with decimal numbers.
- ? One underline is allowed between any two digits, including hexadecimal and binary 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
0x1.921fb5p+1	3.14159262180328369140625e+0
0b1.1001p-4	9.765625e-2

Output

Output values are usually `Math::BigFloat` 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

`Math::BigFloat` supports all methods that `Math::BigInt` supports, except it calculates non-integer results when possible. Please see `Math::BigInt` for a full description of each method. Below are just the most important differences:

Configuration methods

`accuracy()`

```
$x->accuracy(5);    # local for $x
```

```
CLASS->accuracy(5);    # global for all members of CLASS
```

```
    # Note: This also applies to new()!
```

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

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

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

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

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

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

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

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

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

precision()

```
$x->precision(-2);    # local for $x, round at the second
```

```
    # digit right of the dot
```

```
$x->precision(2);    # ditto, round at the second digit
```

```
    # left of the dot
```

```
CLASS->precision(5);    # Global for all members of CLASS
```

```
    # This also applies to new()!
```

```
CLASS->precision(-5);    # ditto
```

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

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

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

Constructor methods

from_hex()

```
$x -> from_hex("0x1.921fb54442d18p+1");
```

```
$x = Math::BigFloat -> from_hex("0x1.921fb54442d18p+1");
```

Interpret input as a hexadecimal string. A prefix ("0x", "x", ignoring case) is optional. A single underscore character ("_") may be placed between any two digits. If the input is invalid, a NaN is returned. The exponent is in base 2 using decimal digits.

If called as an instance method, the value is assigned to the invocand.

from_oct()

```
$x -> from_oct("1.3267p-4");  
$x = Math::BigFloat -> from_oct("1.3267p-4");
```

Interpret input as an octal string. A single underscore character ("_") may be placed between any two digits. If the input is invalid, a NaN is returned. The exponent is in base 2 using decimal digits.

If called as an instance method, the value is assigned to the invocand.

from_bin()

```
$x -> from_bin("0b1.1001p-4");  
$x = Math::BigFloat -> from_bin("0b1.1001p-4");
```

Interpret input as a hexadecimal string. A prefix ("0b" or "b", ignoring case) is optional. A single underscore character ("_") may be placed between any two digits. If the input is invalid, a NaN is returned. The exponent is in base 2 using decimal digits.

If called as an instance method, the value is assigned to the invocand.

from_ieee754()

Interpret the input as a value encoded as described in IEEE754-2008. The input can be given as a byte string, hex string or binary string. The input is assumed to be in big-endian byte-order.

```
# both $dbl and $mbf are 3.141592...  
$bytes = "\x40\x09\x21\xfb\x54\x44\x2d\x18";  
$dbl = unpack "d>", $bytes;  
$mbf = Math::BigFloat -> from_ieee754($bytes, "binary64");
```

bpi()

```
print Math::BigFloat->bpi(100), "\n";
```

Calculate PI to N digits (including the 3 before the dot). The result is rounded according to the current rounding mode, which defaults to "even".

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

Arithmetic methods

`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()`

```
$q = $x->bdiv($y);
```

```
($q, $r) = $x->bdiv($y);
```

In scalar context, divides $\$x$ by $\$y$ and returns the result to the given or default accuracy/precision. In list context, does floored division (F-division), returning an integer $\$q$ and a remainder $\$r$ so that $\$x = \$q * \$y + \r . The remainder (modulo) is equal to what is returned by "`$x->bmod($y)`".

`bmod()`

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

Returns $\$x$ modulo $\$y$. When $\$x$ is finite, and $\$y$ is finite and non-zero, the result is identical to the remainder after floored division (F-division). If, in addition, both $\$x$ and $\$y$ are integers, the result is identical to the result from Perl's `%` operator.

`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).

`bnok()`

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

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

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

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

`bsin()`

```
my $x = Math::BigFloat->new(1);
```

```
print $x->bsin(100), "\n";
```

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

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

bcos()

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

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

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

batan()

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

Calculate the arcus tanges of \$x, modifying \$x in place. See also "batan2()".

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

batan2()

```
my $y = Math::BigFloat->new(2);  
my $x = Math::BigFloat->new(3);  
print $y->batan2($x), "\n";
```

Calculate the arcus tanges of \$y divided by \$x, modifying \$y in place. See also "batan()".

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

as_float()

This method is called when Math::BigFloat encounters an object it doesn't know how to handle. For instance, assume \$x is a Math::BigFloat, or subclass thereof, and \$y is defined, but not a Math::BigFloat, 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_float()".

The method "as_float()" 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.

In Math::BigFloat, "as_float()" has the same effect as "copy()".

to_ieee754()

Encodes the invocand as a byte string in the given format as specified in IEEE

754-2008. Note that the encoded value is the nearest possible representation of the

value. This value might not be exactly the same as the value in the invocand.

```
# $x = 3.1415926535897932385
$x = Math::BigFloat -> bpi(30);
$b = $x -> to_ieee754("binary64"); # encode as 8 bytes
$h = unpack "H*", $b;           # "400921fb54442d18"
# 3.141592653589793115997963...
$y = Math::BigFloat -> from_ieee754($h, "binary64");
```

All binary formats in IEEE 754-2008 are accepted. For convenience, some aliases are recognized: "half" for "binary16", "single" for "binary32", "double" for "binary64", "quadruple" for "binary128", "octuple" for "binary256", and "sexdecuple" for "binary512".

See also <https://en.wikipedia.org/wiki/IEEE_754>.

ACCURACY AND PRECISION

See also: Rounding.

Math::BigFloat supports both precision (rounding to a certain place before or after the dot) and accuracy (rounding to a certain number of digits). For a full documentation, examples and tips on these topics please see the large section about rounding in Math::BigInt.

Since things like $\sqrt{2}$ or "1 / 3" must be presented with a limited accuracy lest an operation consumes all resources, each operation produces no more than the requested number of digits.

If there is no global precision or accuracy set, and the operation in question was not called with a requested precision or accuracy, and the input \$x has no accuracy or precision set, then a fallback parameter will be used. For historical reasons, it is called "div_scale" and can be accessed via:

```
$d = Math::BigFloat->div_scale();    # query
Math::BigFloat->div_scale($n);      # set to $n digits
```

The default value for "div_scale" is 40.

In case the result of one operation has more digits than specified, it is rounded. The rounding mode taken is either the default mode, or the one supplied to the operation after the scale:

```
$x = Math::BigFloat->new(2);
Math::BigFloat->accuracy(5);        # 5 digits max
```

```

$y = $x->copy()->bdiv(3);          # gives 0.66667
$y = $x->copy()->bdiv(3,6);        # gives 0.666667
$y = $x->copy()->bdiv(3,6,undef,'odd'); # gives 0.666667
Math::BigFloat->round_mode('zero');
$y = $x->copy()->bdiv(3,6);        # will also give 0.666667

```

Note that "Math::BigFloat->accuracy()" and "Math::BigFloat->precision()" set the global variables, and thus any newly created number will be subject to the global rounding immediately. This means that in the examples above, the 3 as argument to "bdiv()" will also get an accuracy of 5.

It is less confusing to either calculate the result fully, and afterwards round it explicitly, or use the additional parameters to the math functions like so:

```

use Math::BigFloat;
$x = Math::BigFloat->new(2);
$y = $x->copy()->bdiv(3);
print $y->bround(5),"\\n";        # gives 0.66667
or
use Math::BigFloat;
$x = Math::BigFloat->new(2);
$y = $x->copy()->bdiv(3,5);        # gives 0.66667
print "$y\\n";

```

Rounding

`bround (+$scale)`

Rounds to the \$scale'th place left from the '.', counting from the dot. The first digit is numbered 1.

`bround (-$scale)`

Rounds to the \$scale'th place right from the '.', counting from the dot.

`bround (0)`

Rounds to an integer.

`round (+$scale)`

Preserves accuracy to \$scale digits from the left (aka significant digits) and pads the rest with zeros. If the number is between 1 and -1, the significant digits count from the first non-zero after the '.'

`round (-$scale)` and `round (0)`

These are effectively no-ops.

All rounding functions take as a second parameter a rounding mode from one of the following: 'even', 'odd', '+inf', '-inf', 'zero', 'trunc' or 'common'.

The default rounding mode is 'even'. By using "Math::BigFloat->round_mode(\$round_mode);" you can get and set the default mode for subsequent rounding. The usage of "\$Math::BigFloat::\$round_mode" is no longer supported. The second parameter to the round functions then overrides the default temporarily.

The "as_number()" function returns a BigInt from a Math::BigFloat. It uses 'trunc' as rounding mode to make it equivalent to:

```
$x = 2.5;  
$y = int($x) + 2;
```

You can override this by passing the desired rounding mode as parameter to "as_number()":

```
$x = Math::BigFloat->new(2.5);  
$y = $x->as_number('odd');    # $y = 3
```

Autocreating constants

After "use Math::BigFloat ':constant'" all the floating point constants in the given scope are converted to "Math::BigFloat". This conversion happens at compile time.

In particular

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

prints the value of "2E-100". Note that without conversion of constants the expression 2E-100 will be calculated as normal floating point number.

Please note that ':constant' does not affect integer constants, nor binary nor hexadecimal constants. Use bignum or Math::BigInt to get this to work.

Math library

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

```
use Math::BigFloat lib => 'Calc';
```

You can change this by using:

```
use Math::BigFloat lib => 'GMP';
```

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

Note: The keyword 'lib' will warn when the requested library could not be loaded. To suppress the warning use 'try' instead:

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

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

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

See the respective low-level library documentation for further details.

Please note that Math::BigFloat does not use the denoted library itself, but it merely passes the lib argument to Math::BigInt. So, instead of the need to do:

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

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

you can roll it all into one line:

```
use Math::BigFloat lib => 'GMP';
```

It is also possible to just require Math::BigFloat:

```
require Math::BigFloat;
```

This will load the necessary things (like BigInt) when they are needed, and automatically.

See Math::BigInt for more details than you ever wanted to know about using a different low-level library.

Using Math::BigInt::Lite

For backwards compatibility reasons it is still possible to request a different storage class for use with Math::BigFloat:

```
use Math::BigFloat with => 'Math::BigInt::Lite';
```

However, this request is ignored, as the current code now uses the low-level math library for directly storing the number parts.

EXPORTS

"Math::BigFloat" exports nothing by default, but can export the "bpi()" method:

```
use Math::BigFloat qw/bpi/;
```

```
print bpi(10), "\n";
```

CAVEATS

Do not try to be clever to insert some operations in between switching libraries:

```
require Math::BigFloat;
```

```
my $matter = Math::BigFloat->bone() + 4; # load BigInt and Calc
```

```
Math::BigFloat->import( lib => 'Pari' ); # load Pari, too
```

```
my $anti_matter = Math::BigFloat->bone()+4; # now use Pari
```

This will create objects with numbers stored in two different backend libraries, and VERY BAD THINGS will happen when you use these together:

```
my $flash_and_bang = $matter + $anti_matter; # Don't do this!
```

stringify, bstr()

Both stringify and bstr() now drop the leading '+'. The old code would return '+1.23', the new returns '1.23'. See the documentation in Math::BigInt for reasoning and details.

brsft()

The following will probably not print what you expect:

```
my $c = Math::BigFloat->new('3.14159');  
print $c->brsft(3,10),"\n"; # prints 0.00314153.1415
```

It prints both quotient and remainder, since print calls "brsft()" in list context.

Also, "\$c->brsft()" will modify \$c, so be careful. You probably want to use

```
print scalar $c->copy()->brsft(3,10),"\n";  
# or if you really want to modify $c  
print scalar $c->brsft(3,10),"\n";
```

instead.

Modifying and =

Beware of:

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

It will not do what you think, e.g. making a copy of \$x. Instead it just makes a second reference to the same object and stores it in \$y. Thus anything that modifies \$x will modify \$y (except overloaded math operators), and vice versa. See Math::BigInt for details and how to avoid that.

precision() vs. accuracy()

A common pitfall is to use "precision()" when you want to round a result to a certain number of digits:

```
use Math::BigFloat;  
Math::BigFloat->precision(4); # does not do what you
```

think it does

```

my $x = Math::BigFloat->new(12345); # rounds $x to "12000"!
print "$x\n";                       # print "12000"

my $y = Math::BigFloat->new(3);      # rounds $y to "0"!
print "$y\n";                       # print "0"

$z = $x / $y;                       # 12000 / 0 => NaN!
print "$z\n";

print $z->precision(),"\n";         # 4

```

Replacing "precision()" with "accuracy()" is probably not what you want, either:

```

use Math::BigFloat;

Math::BigFloat->accuracy(4);        # enables global rounding:

my $x = Math::BigFloat->new(123456); # rounded immediately
                                     # to "12350"

print "$x\n";                       # print "123500"

my $y = Math::BigFloat->new(3);      # rounded to "3"
print "$y\n";                       # print "3"

print $z = $x->copy()->bdiv($y)," \n"; # 41170

print $z->accuracy(),"\n";          # 4

```

What you want to use instead is:

```

use Math::BigFloat;

my $x = Math::BigFloat->new(123456); # no rounding
print "$x\n";                       # print "123456"

my $y = Math::BigFloat->new(3);      # no rounding
print "$y\n";                       # print "3"

print $z = $x->copy()->bdiv($y,4)," \n"; # 41150

print $z->accuracy(),"\n";          # undef

```

In addition to computing what you expected, the last example also does not "taint" the result with an accuracy or precision setting, which would influence any further operation.

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::BigFloat
```

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::BigInt 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>> in 2001-2008.

? Florian Ragwitz <flora@cpan.org>, 2010.

? Peter John Acklam <pjacklam@online.no>, 2011-.

perl v5.34.0

2023-11-23

Math::BigFloat(3perl)