NAME

bignum - transparent big number support for Perl

SYNOPSIS

```
use bignum;

$x = 2 + 4.5;                  # Math::BigFloat 6.5
print 2 ** 512 * 0.1;            # Math::BigFloat 134...09.6
print 2 ** 512;                 # Math::BigInt 134...096
print inf + 42;                # Math::BigInt inf
print NaN * 7;                  # Math::BigInt NaN
print hex("0x1234567890123490");    # Perl v5.10.0 or later

{
   no bignum;
   print 2 ** 256;              # a normal Perl scalar now
}

# for older Perls, import into current package:
use bignum qw/hex oct/;
print hex("0x1234567890123490");
print oct("01234567890123490");
```

DESCRIPTION

Literal numeric constants

By default, every literal integer becomes a Math::BigInt object, and literal non-integer becomes a Math::BigFloat object. Whether a numeric literal is considered an integer or non-integers depends only on the value of the constant, not on how it is represented. For instance, the constants 3.14e2 and 0x1.3ap8 become Math::BigInt objects, because they both represent the integer value decimal 314.

The default "use bignum;" is equivalent to

```
use bignum downgrade => "Math::BigInt", upgrade => "Math::BigFloat";
```

The classes used for integers and non-integers can be set at compile time with the "downgrade" and "upgrade" options, for example

```
# use Math::BigInt for integers and Math::BigRat for non-integers
use bignum upgrade => "Math::BigRat";
```

Note that disabling downgrading and upgrading does not affect how numeric literals are converted to objects

```
# disable both downgrading and upgrading
use bignum downgrade => undef, upgrade => undef;
$x = 2.4;       # becomes 2.4 as a Math::BigFloat
$y = 2;         # becomes 2 as a Math::BigInt
```

Upgrading and downgrading

By default, when the result of a computation is an integer, an Inf, or a NaN, the result is downgraded even when all the operands are instances of the upgrade class.

```
use bignum;
$x = 2.4;       # becomes 2.4 as a Math::BigFloat
$y = 1.2;       # becomes 1.2 as a Math::BigFloat
$z = $x / $y;   # becomes 2 as a Math::BigInt due to downgrading
```

Equivalently, by default, when the result of a computation is a finite non-integer, the result is upgraded even when all the operands are instances of the downgrade class.

```
use bignum;
$x = 7;         # becomes 7 as a Math::BigInt
$y = 2;         # becomes 2 as a Math::BigInt
$z = $x / $y;   # becomes 3.5 as a Math::BigFloat due to upgrading
```

The classes used for downgrading and upgrading can be set at runtime with the "**downgrade()**" and "**upgrade()**" methods, but see "CAVEATS" below.

The upgrade and downgrade classes don’t have to be Math::BigInt and Math::BigFloat. For example, to use Math::BigRat as the upgrade class, use

```
use bignum upgrade => "Math::BigRat";
$x = 2;         # becomes 2 as a Math::BigInt
$y = 3.6;       # becomes 18/5 as a Math::BigRat
```

The upgrade and downgrade classes can be modified at runtime

```
use bignum;
$x = 3;         # becomes 3 as a Math::BigInt
$y = 2;         # becomes 2 as a Math::BigInt
```

```
$z = $x / $y;   # becomes 1.5 as a Math::BigFlaot

bignum -> upgrade("Math::BigRat");
$w = $x / $y;   # becomes 3/2 as a Math::BigRat
```

Disabling downgrading doesn't change the fact that literal constant integers are converted to the downgrade class, it only prevents downgrading as a result of a computation. E.g.,

```
use bignum downgrade => undef;
$x = 2;         # becomes 2 as a Math::BigInt
$y = 2.4;       # becomes 2.4 as a Math::BigFloat
$z = 1.2;       # becomes 1.2 as a Math::BigFloat
$w = $x / $y;   # becomes 2 as a Math::BigFloat due to no downgrading
```

If you want all numeric literals, both integers and non-integers, to become Math::BigFloat objects, use the bigfloat pragma.

Equivalently, disabling upgrading doesn't change the fact that literal constant non-integers are converted to the upgrade class, it only prevents upgrading as a result of a computation. E.g.,

```
use bignum upgrade => undef;
$x = 2.5;       # becomes 2.5 as a Math::BigFloat
$y = 7;         # becomes 7 as a Math::BigInt
$z = 2;         # becomes 2 as a Math::BigInt
$w = $x / $y;   # becomes 3 as a Math::BigInt due to no upgrading
```

If you want all numeric literals, both integers and non-integers, to become Math::BigInt objects, use the bigint pragma.

You can even do

```
use bignum upgrade => "Math::BigRat", upgrade => undef;
```

which converts all integer literals to Math::BigInt objects and all non-integer literals to Math::BigRat objects. However, when the result of a computation involving two Math::BigInt objects results in a non-integer (e.g., 7/2), the result will be truncted to a Math::BigInt rather than being upgraded to a Math::BigRat, since upgrading is disabled.

Overloading

Since all numeric literals become objects, you can call all the usual methods from Math::BigInt and

Math::BigFloat on them. This even works to some extent on expressions:

```
perl -Mbignum -le '$x = 1234; print $x->bdec()'
perl -Mbignum -le 'print 1234->copy()->binc();'
perl -Mbignum -le 'print 1234->copy()->binc()->badd(6);'
```

Options

"bignum" recognizes some options that can be passed while loading it via via "use". The following options exist:

a or accuracy

This sets the accuracy for all math operations. The argument must be greater than or equal to zero. See Math::BigInt's **bround()** method for details.

```
perl -Mbignum=a,50 -le 'print sqrt(20)'
```

Note that setting precision and accuracy at the same time is not possible.

p or precision

This sets the precision for all math operations. The argument can be any integer. Negative values mean a fixed number of digits after the dot, while a positive value rounds to this digit left from the dot. 0 means round to integer. See Math::BigInt's **bfround()** method for details.

```
perl -Mbignum=p,-50 -le 'print sqrt(20)'
```

Note that setting precision and accuracy at the same time is not possible.

l, lib, try, or only

Load a different math lib, see "Math Library".

```
perl -Mbignum=l,GMP -e 'print 2 ** 512'
perl -Mbignum=lib,GMP -e 'print 2 ** 512'
perl -Mbignum=try,GMP -e 'print 2 ** 512'
perl -Mbignum=only,GMP -e 'print 2 ** 512'
```

hex Override the built-in **hex()** method with a version that can handle big numbers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as **hex()** is lexically overridden in the current scope whenever the "bignum" pragma is active.

oct Override the built-in **oct()** method with a version that can handle big numbers. This overrides it

by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as **oct()** is lexically overridden in the current scope whenever the "bignum" pragma is active.

v or version
: this prints out the name and version of the modules and then exits.

```
perl -Mbignum=v
```

Math Library

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

```
use bignum lib => 'Calc';
```

you can change this by using:

```
use bignum lib => 'GMP';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and if this also fails, revert to Math::BigInt::Calc:

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

Using c<lib> warns if none of the specified libraries can be found and Math::BigInt and Math::BigFloat fell back to one of the default libraries. To suppress this warning, use "try" instead:

```
use bignum try => 'GMP';
```

If you want the code to die instead of falling back, use "only" instead:

```
use bignum only => 'GMP';
```

Please see respective module documentation for further details.

Method calls

Since all numbers are now objects, you can use the methods that are part of the Math::BigInt and Math::BigFloat API.

But a warning is in order. When using the following to make a copy of a number, only a shallow copy will be made.

```
$x = 9; $y = $x;
$x = $y = 7;
```

Using the copy or the original with overloaded math is okay, e.g., the following work:

```
$x = 9; $y = $x;
print $x + 1, " ", $y,"\n";     # prints 10 9
```

but calling any method that modifies the number directly will result in **both** the original and the copy being destroyed:

```
$x = 9; $y = $x;
print $x->badd(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->binc(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->bmul(2), " ", $y,"\n";        # prints 18 18
```

Using methods that do not modify, but test that the contents works:

```
$x = 9; $y = $x;
$z = 9 if $x->is_zero();                # works fine
```

See the documentation about the copy constructor and "=" in overload, as well as the documentation in Math::BigFloat for further details.

Methods

inf()

A shortcut to return "inf" as an object. Useful because Perl does not always handle bareword "inf" properly.

NaN()

A shortcut to return "NaN" as an object. Useful because Perl does not always handle bareword "NaN" properly.

e

```
# perl -Mbignum=e -wle 'print e'
```

Returns Euler’s number "e", aka **exp**(1) (= 2.7182818284...).

PI

perl -Mbignum=PI -wle ’print PI’

Returns PI (= 3.1415926532..).

bexp()

bexp($power, $accuracy);

Returns Euler’s number "e" raised to the appropriate power, to the wanted accuracy.

Example:

perl -Mbignum=bexp -wle ’print bexp(1,80)’

bpi()

bpi($accuracy);

Returns PI to the wanted accuracy.

Example:

perl -Mbignum=bpi -wle ’print bpi(80)’

accuracy()

Set or get the accuracy.

precision()

Set or get the precision.

round_mode()

Set or get the rounding mode.

div_scale()

Set or get the division scale.

upgrade()

Set or get the class that the downgrade class upgrades to, if any. Set the upgrade class to "undef" to disable upgrading. See "/CAVEATS" below.

downgrade()

Set or get the class that the upgrade class downgrades to, if any. Set the downgrade class to "undef" to disable upgrading. See "CAVEATS" below.

in_effect()

```
use bignum;

print "in effect\n" if bignum::in_effect;       # true
{
   no bignum;
   print "in effect\n" if bignum::in_effect;   # false
}
```

Returns true or false if "bignum" is in effect in the current scope.

This method only works on Perl v5.9.4 or later.

CAVEATS

The **upgrade()** and **downgrade()** methods

Note that setting both the upgrade and downgrade classes at runtime with the "**upgrade()**" and "**downgrade()**" methods, might not do what you expect:

```
# Assuming that downgrading and upgrading hasn’t been modified so far, so
# the downgrade and upgrade classes are Math::BigInt and Math::BigFloat,
# respectively, the following sets the upgrade class to Math::BigRat, i.e.,
# makes Math::BigInt upgrade to Math::BigRat:

bignum -> upgrade("Math::BigRat");

# The following sets the downgrade class to Math::BigInt::Lite, i.e., makes
# the new upgrade class Math::BigRat downgrade to Math::BigInt::Lite

bignum -> downgrade("Math::BigInt::Lite");

# Note that at this point, it is still Math::BigInt, not Math::BigInt::Lite,
# that upgrades to Math::BigRat, so to get Math::BigInt::Lite to upgrade to
# Math::BigRat, we need to do the following (again):

bignum -> upgrade("Math::BigRat");
```

A simpler way to do this at runtime is to use **import()**,

```
bignum -> import(upgrade => "Math::BigRat",
                 downgrade => "Math::BigInt::Lite");
```

Hexadecimal, octal, and binary floating point literals

Perl (and this module) accepts hexadecimal, octal, and binary floating point literals, but use them with care with Perl versions before v5.32.0, because some versions of Perl silently give the wrong result.

Operator vs literal overloading

"bigrat" works by overloading handling of integer and floating point literals, converting them to Math::BigRat objects.

This means that arithmetic involving only string values or string literals are performed using Perl's built-in operators.

For example:

```
use bigrat;
my $x = "900000000000000009";
my $y = "900000000000000007";
print $x - $y;
```

outputs 0 on default 32-bit builds, since "bignum" never sees the string literals. To ensure the expression is all treated as "Math::BigFloat" objects, use a literal number in the expression:

```
print +(0+$x) - $y;
```

Ranges

Perl does not allow overloading of ranges, so you can neither safely use ranges with "bignum" endpoints, nor is the iterator variable a "Math::BigFloat".

```
use 5.010;
for my $i (12..13) {
  for my $j (20..21) {
    say $i ** $j;  # produces a floating-point number,
                   # not an object
  }
}
```

in_effect()

This method only works on Perl v5.9.4 or later.

hex()/oct()

"bignum" overrides these routines with versions that can also handle big integer values. Under Perl prior to version v5.9.4, however, this will not happen unless you specifically ask for it with the two import tags "hex" and "oct" - and then it will be global and cannot be disabled inside a scope with "no bignum":

```
use bignum qw/hex oct/;

print hex("0x1234567890123456");
{
   no bignum;
   print hex("0x1234567890123456");
}
```

The second call to **hex()** will warn about a non-portable constant.

Compare this to:

```
use bignum;

# will warn only under Perl older than v5.9.4
print hex("0x1234567890123456");
```

EXAMPLES

Some cool command line examples to impress the Python crowd ;)

```
perl -Mbignum -le 'print sqrt(33)'
perl -Mbignum -le 'print 2**255'
perl -Mbignum -le 'print 4.5+2**255'
perl -Mbignum -le 'print 3/7 + 5/7 + 8/3'
perl -Mbignum -le 'print 123->is_odd()'
perl -Mbignum -le 'print log(2)'
perl -Mbignum -le 'print exp(1)'
perl -Mbignum -le 'print 2 ** 0.5'
perl -Mbignum=a,65 -le 'print 2 ** 0.2'
perl -Mbignum=l,GMP -le 'print 7 ** 7777'
```

BUGS

Please report any bugs or feature requests to "bug-bignum at rt.cpan.org", or through the web interface at <https://rt.cpan.org/Ticket/Create.html?Queue=bignum> (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 bignum
```

You can also look for information at:

- GitHub

  <https://github.com/pjacklam/p5-bignum>

- RT: CPAN's request tracker

  <https://rt.cpan.org/Dist/Display.html?Name=bignum>

- MetaCPAN

  <https://metacpan.org/release/bignum>

- CPAN Testers Matrix

  <http://matrix.cpantesters.org/?dist=bignum>

- CPAN Ratings

  <https://cpanratings.perl.org/dist/bignum>

LICENSE

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

SEE ALSO

bigint and bigrat.

Math::BigInt, Math::BigFloat, Math::BigRat and Math::Big as well as Math::BigInt::FastCalc,

Math::BigInt::Pari and Math::BigInt::GMP.

AUTHORS

- (C) by Tels <http://bloodgate.com/> in early 2002 - 2007.
- Maintained by Peter John Acklam <pjacklam@gmail.com>, 2014-.