NAME

bignum - Transparent BigNumber support for Perl

SYNOPSIS

```
use bignum;

$x = 2 + 4.5,"\n";   # BigFloat 6.5
print 2 ** 512 * 0.1,"\n";  # really is what you think it is
print inf * inf,"\n";   # prints inf
print NaN * 3,"\n";   # prints NaN

{
  no bignum;
  print 2 ** 256,"\n";  # a normal Perl scalar now
}

# for older Perls, note that this will be global:
use bignum qw/hex oct/;
print hex("0x1234567890123490"),"\n";
print oct("01234567890123490"),"\n";
```

DESCRIPTION

All operators (including basic math operations) are overloaded. Integer and floating-point constants are created as proper BigInts or BigFloats, respectively.

If you do

```
use bignum;
```

at the top of your script, Math::BigFloat and Math::BigInt will be loaded and any constant number will be converted to an object (Math::BigFloat for floats like 3.1415 and Math::BigInt for integers like 1234).

So, the following line:

```
$x = 1234;
```

creates actually a Math::BigInt and stores a reference to in $x. This happens transparently and behind your back, so to speak.

You can see this with the following:

```
perl -Mbignum -le 'print ref(1234)'
```

Don't worry if it says Math::BigInt::Lite, bignum and friends will use Lite if it is installed since it is faster for some operations. It will be automatically upgraded to BigInt whenever necessary:

```
perl -Mbignum -le 'print ref(2**255)'
```

This also means it is a bad idea to check for some specific package, since the actual contents of $x might be something unexpected. Due to the transparent way of bignum `ref()` should not be necessary, anyway.

Since Math::BigInt and BigFloat also overload the normal math operations, the following line will still work:

```
perl -Mbignum -le 'print ref(1234+1234)'
```

Since numbers are actually objects, you can call all the usual methods from BigInt/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);'
perl -Mbignum -le 'print +(1234)->copy()->binc()'
```

(Note that print doesn't do what you expect if the expression starts with '(' hence the +)

You can even chain the operations together as usual:

```
perl -Mbignum -le 'print 1234->copy()->binc->badd(6);'
1241
```

Under bignum (or bigint or bigrat), Perl will "upgrade" the numbers appropriately. This means that:

```
perl -Mbignum -le 'print 1234+4.5'
1238.5
```

will work correctly. These mixed cases don't do always work when using Math::BigInt or Math::BigFloat alone, or at least not in the way normal Perl scalars work.

If you do want to work with large integers like under `use integer;`, try `use bigint;`:

```
perl -Mbigint -le 'print 1234.5+4.5'
1238
```

There is also `use bigrat;` which gives you big rationals:

```
perl -Mbigrat -le 'print 1234+4.1'
12381/10
```

The entire upgrading/downgrading is still experimental and might not work as you expect or may even have bugs. You might get errors like this:

```
Can't use an undefined value as an ARRAY reference at
/usr/local/lib/perl5/5.8.0/Math/BigInt/Calc.pm line 864
```

This means somewhere a routine got a BigFloat/Lite but expected a BigInt (or vice versa) and the upgrade/downgrad path was missing. This is a bug, please report it so that we can fix it.

You might consider using just Math::BigInt or Math::BigFloat, since they allow you finer control over what get's done in which module/space. For instance, simple loop counters will be Math::BigInts under `use bignum;` and this is slower than keeping them as Perl scalars:

```
perl -Mbignum -le 'for ($i = 0; $i < 10; $i++) { print ref($i); }'
```

Please note the following does not work as expected (prints nothing), since overloading of '..' is not yet possible in Perl (as of v5.8.0):

```
perl -Mbignum -le 'for (1..2) { print ref($_); }'
```

Options

bignum recognizes some options that can be passed while loading it via use. The options can (currently) be either a single letter form, or the long form. 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() function 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 or 1 mean round to integer. See Math::BigInt's bfround() function for details.

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

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

t or trace

This enables a trace mode and is primarily for debugging bignum or Math::BigInt/Math::BigFloat.

l or lib

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

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

Currently there is no way to specify more than one library on the command line. This means the following does not work:

```
perl -Mbignum=l,GMP,Pari -e 'print 2 ** 512'
```

This will be hopefully fixed soon ;)

hex

Override the built-in hex() method with a version that can handle big integers. Note that under Perl older than v5.9.4, this will be global and cannot be disabled with "no bigint;".

oct

Override the built-in oct() method with a version that can handle big integers. Note that under Perl older than v5.9.4, this will be global and cannot be disabled with "no bigint;".

v or version

This prints out the name and version of all modules used and then exits.

```
perl -Mbignum=v
```

Methods

Beside import() and AUTOLOAD() there are only a few other methods.

Since all numbers are now objects, you can use all functions that are part of the BigInt or BigFloat API. It is wise to use only the bxxx() notation, and not the fxxx() notation, though. This makes it possible that the underlying object might morph into a different class than BigFloat.

Caveats

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;
```

If you want to make a real copy, use the following:

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

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 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 BigInt for further details.

inf()

A shortcut to return Math::BigInt->binf(). Useful because Perl does not always handle bareword `inf` properly.

NaN()

A shortcut to return Math::BigInt->bnan(). 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).

PI()

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

Returns PI.

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)'
```

upgrade()

Return the class that numbers are upgraded to, is in fact returning `$Math::BigInt::upgrade`.

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.

Math Library

Math with the numbers is done (by default) by a module called Math::BigInt::Calc. This 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 when this also fails, revert to Math::BigInt::Calc:

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

Please see respective module documentation for further details.

Using `lib` warns if none of the specified libraries can be found and *Math::BigInt* did fall 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';
```

INTERNAL FORMAT

The numbers are stored as objects, and their internals might change at anytime, especially between math operations. The objects also might belong to different classes, like Math::BigInt, or Math::BigFLoat. Mixing them together, even with normal scalars is not extraordinary, but normal and

expected.You should not depend on the internal format, all accesses must go through accessor methods. E.g. looking at $x->{sign} is not a bright idea since there is no guaranty that the object in question has such a hashkey, nor is a hash underneath at all.

SIGN

The sign is either '+', '-', 'NaN', '+inf' or '-inf' and stored separately. You can access it with the sign() method.

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.

CAVEATS

in_effect()

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

hex()/oct()

`bigint` 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 bigint":

```
use bigint qw/hex oct/;

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

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

Compare this to:

```
use bigint;

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

MODULES USED

`bignum` is just a thin wrapper around various modules of the Math::BigInt family. Think of it as the head of the family, who runs the shop, and orders the others to do the work.

The following modules are currently used by bignum:

```
Math::BigInt::Lite (for speed, and only if it is loadable)
Math::BigInt
Math::BigFloat
```

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=a,65,l,GMP -le 'print 7 ** 7777'
```

LICENSE

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

SEE ALSO

Especially *bigrat* as in `perl -Mbigrat -le 'print 1/3+1/4'`.

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

AUTHORS

(C) by Tels *http://bloodgate.com/* in early 2002 - 2007.