gmp

Ruby bindings to the GMP library
Edition 0.4.7
25 March 2010

written by Sam Rawlins
with extensive quoting from the GMP Manual

This manual describes how to use the gmp Ruby gem, which provides bindings to the GNU multiple precision arithmetic library, version 4.3.x or 5.0.x.

Copyright 2009, 2010 Sam Rawlins.
No license yet.

Contents

1 Introduction to GNU MP

This entire page is copied verbatim from the GMP Manual.

GNU MP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types.

Many applications use just a few hundred bits of precision; but some applications may need thousands or even millions of bits. GMP is designed to give good performance for both, by choosing algorithms based on the sizes of the operands, and by carefully keeping the overhead at a minimum.

The speed of GMP is achieved by using fullwords as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code for the most common inner loops for many different CPUs, and by a general emphasis on speed (as opposed to simplicity or elegance).

There is assembly code for these CPUs: ARM, DEC Alpha 21064, 21164, and 21264, AMD 29000, AMD K6, K6-2, Athlon, and Athlon64, Hitachi SuperH and SH-2, HPPA 1.0, 1.1, and 2.0, Intel Pentium, Pentium Pro/II/III, Pentium 4, generic x86, Intel IA-64, i960, Motorola MC68000, MC68020, MC88100, and MC88110, Motorola/IBM PowerPC 32 and 64, National NS32000, IBM POWER, MIPS R3000, R4000, SPARCv7, SuperSPARC, generic SPARCv8, UltraSPARC, DEC VAX, and Zilog Z8000. Some optimizations also for Cray vector systems, Clipper, IBM ROMP (RT), and Pyramid AP/XP.

For up-to-date information on GMP, please see the GMP web pages at
`http://gmplib.org/`

The latest version of the library is available at
`ftp://ftp.gnu.org/gnu/gmp/`

Many sites around the world mirror '`ftp.gnu.org`', please use a mirror near you, see `http://www.gnu.org/ord`
for a full list.

There are three public mailing lists of interest. One for release announcements, one for general questions and discussions about usage of the GMP library, and one for bug reports. For more information, see

`http://gmplib.org/mailman/listinfo/`.

The proper place for bug reports is gmp-bugs@gmplib.org. See Chapter 4 [Reporting Bugs], page 28 for information about reporting bugs.

2 Introduction to the gmp gem

The gmp Ruby gem is a Ruby library that provides bindings to GMP. The gem is incomplete, and will likely only include a subset of the GMP functions. It is built as a C extension for ruby, interacting with gmp.h. The gmp gem is not endorsed or supported by GNU or the GMP team. The gmp gem also does not ship with GMP, so GMP must be compiled separately.

3 Installing the gmp gem

3.1 Prerequisites

OK. First, we've got a few requirements. To install the gmp gem, you need one of the following versions of Ruby:

- (MRI) Ruby 1.8.6 - tested lightly.
- (MRI) Ruby 1.8.7 - tested seriously.
- (MRI) Ruby 1.9.1 - tested seriously.

As you can see only Matz's Ruby Interpreter (MRI) is supported. I haven't even put a thought into trying other interpreters/VMs. I intend to look into FFI, which supposedly will allow me to load this extension into JRuby and Rubinius, not sure about others...

Next is the platform, the combination of the architecture (processor) and OS. As far as I can tell, if you can compile GMP and Ruby on a given platform, you can use the gmp gem there too. Please report problems with that hypothesis.

Lastly, GMP. GMP must be compiled and working. "and working" means you ran "make check" while installing GMP. The following versions of GMP have been tested:

- GMP 4.3.1
- GMP 4.3.2
- GMP 5.0.0
- GMP 5.0.1

That's all. I don't intend to test any older versions, maybe 4.3.0 for completeness.

Here is a table of the exact environments on which I have tested the gmp gem:

Platform	Ruby	GMP
Cygwin on x86	(MRI) Ruby 1.8.7	GMP 4.3.1
Linux (LinuxMint 7) on x86	(MRI) Ruby 1.8.7	GMP 4.3.1
Mac OS X 10.5.7 on x86 (32-bit)	(MRI) Ruby 1.8.6	GMP 4.3.1
Mac OS X 10.5.7 on x86 (32-bit)	(MRI) Ruby 1.9.1	GMP 4.3.1
Windows XP on x86 (32-bit)	(MRI) Ruby 1.9.1	GMP 5.0.1

3.2 Installing

You may clone the gmp gem's git repository with:

```
git clone git://github.com/srawlins/gmp.git
```

Or you may install the gem from gemcutter:

```
gem install gmp
```

At this time, the gem does self-compiles. If required libraries cannot be found, you may compile the C extensions manually with:

```
cd <gmp gem directory>/ext
ruby extconf.rb
make
```

There shouldn't be any errors, or warnings.

4 Testing the gmp gem

Testing the gmp gem is quite simple. The test/unit_tests.rb suite uses Unit::Test. You can run this test suite with:

```
cd <gmp gem directory>/test
ruby unit_tests.rb
```

All tests should pass. If you don't have the test-unit gem installed, then you may run into one error.

5 GMP and gmp gem basics

5.1 Classes

The gmp gem includes the namespace `GMP` and four classes within `GMP`:

- `GMP::Z` - Methods for signed integer arithmetic. There are about 64 methods here.
- `GMP::Q` - Methods for rational number arithmetic. There are at least 11 methods here (still accounting).
- `GMP::F` - Methods for floating-point arithmetic. There are at least 6 methods here (still accounting).
- `GMP::RandState` - Methods for random number generation. There are 3 methods here.

In addition to the above four classes, there are also four constants within `GMP`:

- `GMP::GMP_VERSION` - The version of GMP linked into the gmp gem
- `GMP::GMP_CC` - The compiler that compiled GMP linked into the gmp gem

- `GMP::GMP_CFLAGS` - The compiler flags used to compile GMP linked into the gmp gem
- `GMP::GMP_BITS_PER_LIMB` - The number of bits per limb

6 MPFR basics

The gmp gem can optionally link to MPFR, the Multiple Precision Floating-Point Reliable Library. The x86-mswin32 version of the gmp gem comes with MPFR. This library uses the floating-point type from GMP, and thus the MPFR functions mapped in the gmp gem become methods in `GMP::F`.
There are additional constants within `GMP` when MPFR is linked:

- `GMP::MPFR_VERSION` - The version of MPFR linked into the gmp gem.
- `GMP::GMP_RNDN` - Rounding mode representing "round to nearest."
- `GMP::GMP_RNDZ` - Rounding mode representing "round toward zero."
- `GMP::GMP_RNDU` - Rounding mode representing "round toward positive infinity."
- `GMP::GMP_RNDD` - Rounding mode representing "round toward negative infinity."

7 Integer Functions

7.1 Initializing, Assigning Integers

new

GMP::Z.new → *integer*
GMP::Z.new(*numeric = 0*) → *integer*
GMP::Z.new(*str*) → *integer*

This method creates a new *GMP::Z* integer. It takes one optional argument for the value of the integer. This argument can be one of several classes. Here are some examples:

```
GMP::Z.new                   #=> 0 (default)
GMP::Z.new(1)                #=> 1 (Ruby Fixnum)
GMP::Z.new("127")            #=> 127 (Ruby String)
GMP::Z.new(4294967296)       #=> 4294967296 (Ruby Bignum)
GMP::Z.new(GMP::Z.new(31)) #=> 31 (GMP Integer)
```

There is also a convenience method available, `GMP::Z()`.

7.2 Converting Integers

to_d

integer.to_d → *float*

Returns *integer* as an Float if *integer* fits in a Float.
Otherwise returns the least significant part of `integer`, with the same sign as *integer*.
If *integer* is too big to fit in a Float, the returned result is probably not very useful. To find out if the value will fit, use the function *mpz_fits_slong_p* (**Unimplemented**).

to_i

integer.to_i → *fixnum*

Returns *integer* as a Fixnum if *integer* fits in a Fixnum.

Otherwise returns the least significant part of *integer*, with the same sign as *integer*.

If *integer* is too big to fit in a Fixnum, the returned result is probably not very useful. To find out if the value will fit, use the function *mpz_fits_slong_p* (**Unimplemented**).

to_s *integer*.to_s(*base = 10*) → *str*

Converts *integer* to a string of digits in base *base*. The *base* argument may vary from 2 to 62 or from -2 to -36, or be a symbol, one of *:bin*, *:oct*, *:dec*, or *:hex*.

For *base* in the range 2..36, digits and lower-case letters are used; for -2..-36 (and *:bin*, *:oct*, *:dec*, and *:hex*), digits and upper-case letters are used; for 37..62, digits, upper-case letters, and lower-case letters (in that significance order) are used. Here are some examples:

```
GMP::Z(1).to_s         #=> "1"
GMP::Z(32).to_s(2)     #=> "100000"
GMP::Z(32).to_s(4)     #=> "200"
GMP::Z(10).to_s(16)    #=> "a"
GMP::Z(10).to_s(-16)   #=> "A"
GMP::Z(255).to_s(:bin) #=> "11111111"
GMP::Z(255).to_s(:oct) #=> "377"
GMP::Z(255).to_s(:dec) #=> "255"
GMP::Z(255).to_s(:hex) #=> "ff"
```

7.3 Integer Arithmetic

+ *integer* + *numeric* → *numeric*

Returns the sum of *integer* and *numeric*. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *GMP::Q*, *GMP::F*, or *Bignum*.

add! *integer*.add!(*numeric*) → *numeric*

Sums *integer* and *numeric*, in place. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *GMP::Q*, *GMP::F*, or *Bignum*.

- *integer* - *numeric* → *numeric*
integer.sub!(*numeric*) → *numeric*

Returns the difference of *integer* and *numeric*. The destructive method calculates the difference in place. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *GMP::Q*, *GMP::F*, or *Bignum*. Here are some examples:

```
seven = GMP::Z(7)
nine  = GMP::Z(9)
half  = GMP::Q(1,2)
pi    = GMP::F("3.14")
nine - 5       #=> 4 (GMP Integer)
nine - seven   #=> 2 (GMP Integer)
nine - (2**32) #=> -4294967287 (GMP Integer)
nine - nine    #=> 0 (GMP Integer)
nine - half    #=> 8.5 (GMP Rational)
nine - pi      #=> 5.86 (GMP Float)
```

***** *integer* * *numeric* → *numeric*
integer.mul(*numeric*) → *numeric*
integer.mul!(*numeric*) → *numeric*

Returns the product of *integer* and *numeric*. The destructive method calculates the product in place. *numeric* can be an instance of *GMP::Z*, *Fixnum*, *GMP::Q*, *GMP::F*, or *Bignum*.

<< *integer* <<*numeric* → *integer*

Returns *integer* times 2 to the *numeric* power. This can also be defined as a left shift by *numeric* bits.

-@ -*integer*
integer.neg
integer.neg!

Returns the negation, the additive inverse, of *integer*. The destructive method negates in place.

abs *integer*.abs
integer.abs!

Returns the absolute value of *integer*. The destructive method calculates the absolute value in place.

7.4 Integer Division

tdiv *integer*.tdiv(*numeric*) → *integer*

Returns the division of *integer* by *numeric*, truncated. *numeric* can be an instance of $GMP :: Z$, *Fixnum*, *Bignum*. The return object's class is always $GMP :: Z$.

fdiv *integer*.fdiv(*numeric*) → *integer*

Returns the division of *integer* by *numeric*, floored. *numeric* can be an instance of $GMP :: Z$, *Fixnum*, *Bignum*. The return object's class is always $GMP :: Z$.

cdiv *integer*.cdiv(*numeric*) → *integer*

Returns the ceiling division of *integer* by *numeric*. *numeric* can be an instance of $GMP :: Z$, *Fixnum*, *Bignum*. The return object's class is always $GMP :: Z$.

tmod *integer*.tmod(*numeric*) → *integer*

Returns the remainder after truncated division of *integer* by *numeric*. *numeric* can be an instance of $GMP :: Z$, *Fixnum*, or *Bignum*. The return object's class is always $GMP :: Z$.

fmod *integer*.fmod(*numeric*) → *integer*

Returns the remainder after floored division of *integer* by *numeric*. *numeric* can be an instance of $GMP :: Z$, *Fixnum*, or *Bignum*. The return object's class is always $GMP :: Z$.

cmod *integer*.cmod(*numeric*) → *integer*

Returns the remainder after ceilinged division of *integer* by *numeric*. *numeric* can be an instance of $GMP :: Z$, *Fixnum*, or *Bignum*. The return object's class is always $GMP :: Z$.

% *integer* % *numeric* → *integer*

Returns *integer* modulo *numeric*. *numeric* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*. The return object's class is always *GMP::Z*.

7.5 Integer Exponentiation

****** *integer* ** *numeric* → *numeric*
integer.pow(*numeric*) → *numeric*

Returns *integer* raised to the *numeric* power.

powmod *integer*.powmod(*exp*, *mod*) $\rightarrow$ *integer*

Returns *integer* raised to the *exp* power, modulo *mod*. Negative *exp* is supported if an inverse, $integer^{-1}$ modulo *mod*, exists. If an inverse doesn't exist then a divide by zero exception is raised.

7.6 Integer Roots

root *integer*.root(*numeric*) $\rightarrow$ *numeric*

Returns the integer part of the *numeric*'th root of *integer*.

sqrt *integer*.sqrt $\rightarrow$ *numeric*
integer.sqrt! $\rightarrow$ *numeric*

Returns the truncated integer part of the square root of *integer*.

sqrtrem *integer*.sqrtrem $\rightarrow$ *sqrt*, *rem*

Returns the truncated integer part of the square root of *integer* as *sqrt* and the remainder, $integer - sqrt * sqrt$, as *rem*, which will be zero if *integer* is a perfect square.

power? *integer*.power? $\rightarrow$ *true* | *false*

Returns true if *integer* is a perfect power, i.e., if there exist integers a and b, with $b > 1$, such that *integer* equals a raised to the power b.

Under this definition both 0 and 1 are considered to be perfect powers. Negative values of integers are accepted, but of course can only be odd perfect powers.

square? *integer*.square? $\rightarrow$ *true* | *false*

Returns true if *integer* is a perfect square, i.e., if the square root of *integer* is an integer. Under this definition both 0 and 1 are considered to be perfect squares.

7.7 Number Theoretic Functions

is_probab_prime? *integer*.is_probab_prime?(*reps* = 5) → 0, 1, or 2

Determine whether *integer* is prime. Returns 2 if *integer* is definitely prime, returns 1 if *integer* is probably prime (without being certain), or returns 0 if *integer* is definitely composite.

This function does some trial divisions, then some Miller-Rabin probabilistic primality tests. *reps* controls how many such tests are done, 5 to 10 is a reasonable number, more will reduce the chances of a composite being returned as probably prime.

Miller-Rabin and similar tests can be more properly called compositeness tests. Numbers which fail are known to be composite but those which pass might be prime or might be composite. Only a few composites pass, hence those which pass are considered probably prime.

next_prime

integer.next_prime → *prime*
integer.nextprime → *prime*
integer.next_prime! → *prime*
integer.nextprime! → *prime*

Returns the next prime greater than *integer*. The destructive method sets *integer* to the next prime greater than *integer*.

This function uses a probabilistic algorithm to identify primes. For practical purposes it's adequate, the chance of a composite passing will be extremely small.

gcd *a*.gcd(*b*) → *g*

Computes the greatest common divisor of *a* and *b*. *g* will always be positive, even if *a* or *b* is negative. *b* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.

```
GMP::Z(24).gcd(GMP::Z(8))   #=> GMP::Z(8)
GMP::Z(24).gcd(8)   #=> GMP::Z(8)
GMP::Z(24).gcd(2**32)   #=> GMP::Z(8)
```

invert *a*.invert(*m*) → *integer*

Computes the inverse of *a* mod *m*. *m* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.

```
GMP::Z(2).invert(GMP::Z(11))   #=> GMP::Z(6)
GMP::Z(3).invert(11)   #=> GMP::Z(4)
GMP::Z(5).invert(11)   #=> GMP::Z(9)
```

jacobi

a.jacobi(b) → *integer*
GMP::Z.jacobi(a, b) → *integer*

Returns the Jacobi symbol (a/b). This is defined only for b odd. If b is even, a range exception will be raised.

GMP::Z.jacobi (the instance method) requires b to be an instance of *GMP::Z*. *GMP::Z#jacobi* (the class method) requires a and b each to be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.

legendre

a.legendre(b) → *integer*

Returns the Legendre symbol (a/b). This is defined only for p an odd positive prime. If p is even, negative, or composite, a range exception will be raised.

remove

n.remove($factor$) → (*integer*, *times*)

Remove all occurrences of the factor $factor$ from n. $factor$ can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*. *integer* is the resulting integer, an instance of *GMP::Z*. *times* is how many times $factor$ was removed, a *Fixnum*.

fac

GMP::Z.fac(n) → *integer*

Returns $n!$, or, n factorial.

fib

GMP::Z.fib(n) → *integer*

Returns $F[n]$, or, the nth Fibonacci number.

7.8 Integer Comparisons

<=>

a <=> b → *fixnum*

Returns a negative Fixnum if a is less than b.
Returns 0 if a is equal to b.
Returns a positive Fixnum if a is greater than b.

<

$a < b$ → *boolean*

Returns true if a is less than b.

<=

a <= b → *boolean*

Returns true if a is less than or equal to b.

==

$a == b$ → *boolean*

Returns true if a is equal to b.

>= $a >= b \to boolean$

Returns true if a is greater than or equal to b.

> $a > b \to boolean$

Returns true if a is greater than b.

cmpabs a.cmpabs(b) $\to$ *fixnum*

Returns a negative Fixnum if abs(a) is less than abs(b).
Returns 0 if abs(a) is equal to abs(b).
Returns a positive Fixnum if abs(a) is greater than abs(b).

sgn a.sgn $\to -1, 0$, or 1

Returns -1 if a is less than b.
Returns 0 if a is equal to b.
Returns 1 if a is greater than b.

eql? a.eql?(b) $\to$ *boolean*

Used when comparing objects as Hash keys.

hash a.hash $\to$ *string*

Used when comparing objects as Hash keys.

7.9 Integer Logic and Bit Fiddling

and $a \,\&\, b \to integer$

Returns *integer*, the bitwise and of a and b.

ior $a \mid b \to integer$

Returns *integer*, the bitwise inclusive or of a and b.

xor $a \,\hat{}\, b \to integer$

Returns *integer*, the bitwise exclusive or of a and b.

com *integer*.com $\to$ *complement*
integer.com! $\to$ *complement*

Returns the one's complement of *integer*. The destructive method sets *integer* to the one's complement of *integer*.

popcount n.popcount → *fixnum*

If $n >= 0$, return the population count of n, which is the number of 1 bits in the binary representation. If $n < 0$, the number of 1s is infinite, and the return value is the largest possible *mp_bitcnt_t*.

scan0 n.scan0(i) → *integer*

Scans n, starting from bit i, towards more significant bits, until the first 0 bit is found. Return the index of the found bit.

If the bit at i is already what's sought, then i is returned.

If there's no bit found, then $INT2FIX(ULONG\_MAX)$ is returned. This will happen in scan0 past the end of a negative number.

scan1 n.scan1(i) → *integer*

Scans n, starting from bit i, towards more significant bits, until the first 1 bit is found. Return the index of the found bit.

If the bit at i is already what's sought, then i is returned.

If there's no bit found, then $INT2FIX(ULONG\_MAX)$ is returned. This will happen in scan1 past the end of a negative number.

[] $n[bit\_index]$ → 0 or 1

Tests bit *bit_index* in n and return 0 or 1 accordingly.

[]= $n[bit\_index]=i$ → *nil*

Sets bit *bit_index* in n to i.

7.10 Miscellaneous Integer Functions

odd? n.odd? → *boolean*

Returns whether n is odd.

even? n.even? → *boolean*

Returns whether n is even.

sizeinbase n.sizeinbase(b) → *digits*

Returns the number of digits in base b. b can vary between 2 and 62.

size_in_bin n.size_in_bin $\rightarrow$ *digits*

Returns the number of digits in n's binary representation.

8 Rational Functions

8.1 Initializing, Assigning Rationals

new

GMP::Q.new → *rational*
GMP::Q.new(*numerator = 0*, *denominator = 1*) → *rational*
GMP::Q.new(*str*) → *rational*

This method creates a new *GMP::Q* rational number. It takes two optional arguments for the value of the numerator and denominator. These arguments can each be an instance of several classes. Here are some examples:

```
GMP::Q.new                  #=> 0 (default)
GMP::Q.new(1)               #=> 1 (Ruby Fixnum)
GMP::Q.new(1,3)             #=> 1/3 (Ruby Fixnums)
GMP::Q.new("127")           #=> 127 (Ruby String)
GMP::Q.new(4294967296)      #=> 4294967296 (Ruby Bignum)
GMP::Q.new(GMP::Z.new(31)) #=> 31 (GMP Integer)
```

There is also a convenience method available, `GMP::Q()`.

9 Random Number Functions

9.1 Random State Initialization

new

GMP::RandState.new → *mersenne twister state*
GMP::RandState.new(:default) → *mersenne twister state*
GMP::RandState(:mt) → *mersenne twister random state*
GMP::RandState.new(:lc_2exp, a, c, m2exp) → *linear congruential state*
GMP::RandState.new(:lc_2exp_size, size) → *linear congruential state*

This method creates a new *GMP::RandState* instance. The first argument defaults to *:default* (also *:mt*), which initializes the *GMP::RandState* for a Mersenne Twister algorithm. No other arguments should be given if *:default* or *:mt* is specified.

If the first argument given is *:lc_2exp*, then the *GMP::RandState* is initialized for a linear congruential algorithm. *:lc_2exp* must be followed with a, c, and $m2exp$. The algorithm can then proceed as $(X = (a * X + c) \mod 2^{m2exp})$.

GMP::RandState can also be initialized for a linear congruential algorithm with *:lc_2exp_size*. This initializer instead takes just one argument, *size*. a, c, and $m2exp$ are then chosen from a table, with $m2exp/2 > size$. The maximum size currently supported is 128.

```
GMP::RandState.new
GMP::RandState.new(:mt)
GMP::RandState.new(:lc_2exp, 1103515245, 12345, 15)   #=> Perl's
old rand()
GMP::RandState.new(:lc_2exp, 25_214_903_917, 11, 48)   #=> drand48
```

9.2 Random State Seeding

seed

state.seed(*integer*) → *integer*

Set an initial seed value into *state*. *integer* can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.

9.3 Integer Random Numbers

urandomb

state.urandomb(n) → *integer*

Generates a uniformly distributed random integer in the range 0 to $2^n - 1$, inclusive.

urandomm

state.urandomm(n) → *integer*

Generates a uniformly distributed random integer in the range 0 to $n-1$, inclusive. n can be an instance of *GMP::Z*, *Fixnum*, or *Bignum*.