Ruby trunk - Bug #2121

mathn/rational destroys Fixnum#/, Fixnum#quo and Bignum#/, Bignum#quo

09/19/2009 02:07 PM - headius (Charles Nutter)

Status:	Closed	
Priority:	Normal	
Assignee:	keiju (Keiju Ishitsuka)	
Target version:		
ruby -v:	Any 1.8.6 or higher	**Backport:**

Description

=begin
I've known this for a while, but only now realized this is actually a terrible bug.

The mathn library replaces Fixnum#/ and Bignum#/ causing them to return a different value. When the result of a division is not an integral value, the default versions will return 0. I can think of many algorithms that would use this expectation, and most other languages will not upconvert integral numeric types to floating-point or fractional types without explicit consent by the programmer.

When requiring 'mathn', Fixnum#/ and Bignum#/ are replaced with versions that return a fractional value ('quo') causing a core math operator to return not just a different type, but *a different value*.

No core library should be allowed to modify the return value of core numeric operators, or else those operators are worthless; you can't rely on them to return a specific value *ever* since someone else could require 'mathn' or 'rational'.

Note also that 'rational' destroys Fixnum#quo and Bignum#quo. This is also a bug that should be fixed, though it is less dangerous because they're not commonly-used operators.

The following code should not fail; Fixnum#/ should never return a value of a different magnitude based on which libraries are loaded:

{{{
require 'test/unit'

class TestFixnumMath < Test::Unit::TestCase
0 to ensure it runs first, for illustration purposes
def test_0_without_mathn
assert_equal 0, 1/3
end

def test_with_mathn
require 'mathn'
assert_equal 0, 1/3
end
end
}}}

But it does fail:

{{{
~/projects/jruby ruby test_fixnum_math.rb
Loaded suite test_fixnum_math
Started
.F
Finished in 0.016003 seconds.

1) Failure:
test_with_mathn(TestFixnumMath) [test_fixnum_math.rb:11]:
expected but was
.

2 tests, 2 assertions, 1 failures, 0 errors
}}}
=end

Related issues:

Related to Ruby trunk - Feature #10169: It might be better to make Mathn clas...	**Closed**

History

#1 - 09/19/2009 02:08 PM - headius (Charles Nutter)

=begin
One correction; 'quo' is broken in that it returns 0.333333333... in the default case and "1/3" in the mathn/rational case. These values are also of different magnitude, since 1/3 cannot be represented exactly in a floating point number.
=end

#2 - 09/19/2009 05:04 PM - headius (Charles Nutter)

=begin
On Sat, Sep 19, 2009 at 12:28 AM, Joel VanderWerf
vjoel@path.berkeley.edu wrote:

> Perhaps it should be the responsibility of users of numeric operators to
> #floor explicitly when that is the intent, rather than rely on the (mostly
> standard, sometimes convenient, but questionable) 1/2==0 behavior. Doing so
> would make it easier to adapt the code to float, rational, or other numeric
> types.
>
> In your proposal, would Rational(1,3) be the preferred notation, since
> 1/3==0? Or would there be something else, 1//3 or ...?
>
> I've always thought of mathn as a kind of alternate ruby, not just another
> core library, hence to be used with caution...

I think Brian Ford expressed what I feel best...there should always be another method or operator. Using another operator or method is an explicit "buy-in" by the user--rather than a potential (at some undetermined time in the future) that everything you know about integral division in your program changes wildly. It should not be possible for any library to undermine the basic mathematical expectations of my program. Doing so, or expecting the user to do extra work to guarantee the *common case*, is a recipe for serious failure.

- Charlie

=end

#3 - 09/19/2009 05:07 PM - headius (Charles Nutter)

=begin
On Sat, Sep 19, 2009 at 1:01 AM, Charles Oliver Nutter
headius@headius.com wrote:

> integral division in your program changes wildly. It should not be
> possible for any library to undermine the basic mathematical
> expectations of my program. Doing so, or expecting the user to do
> extra work to guarantee the *common case*, is a recipe for serious
> failure.

I will revise this slightly; it should not be possible for any *core library* to undermine the basic mathematical expectations of my program. There's a well-accepted assumption that third-party libraries are not subject to the stricter requirements of a core set, and I don't mean to say that it should not be possible to make these sorts of intriguing changes. But it should not be standard practice for any language runtime to violate key, core expectations of a programming language like the results of integral division. So I understand the utility and the intrigue, but I don't think it should be allowed in the core libraries to go that far.

- Charlie

=end

#4 - 09/20/2009 04:29 PM - brixen (Brian Shirai)

=begin
Hi,

On Sat, Sep 19, 2009 at 1:04 AM, Charles Oliver Nutter
headius@headius.com wrote:

> On Sat, Sep 19, 2009 at 12:28 AM, Joel VanderWerf
> vjoel@path.berkeley.edu wrote:
>
> > Perhaps it should be the responsibility of users of numeric operators to
> > #floor explicitly when that is the intent, rather than rely on the (mostly
> > standard, sometimes convenient, but questionable) 1/2==0 behavior. Doing so
> > would make it easier to adapt the code to float, rational, or other numeric
> > types.
> >
> > In your proposal, would Rational(1,3) be the preferred notation, since
> > 1/3==0? Or would there be something else, 1//3 or ...?
> >
> > I've always thought of mathn as a kind of alternate ruby, not just another
> > core library, hence to be used with caution...
>
> I think Brian Ford expressed what I feel best...there should always be
> another method or operator. Using another operator or method is an
> explicit "buy-in" by the user--rather than a potential (at some
> undetermined time in the future) that everything you know about
> integral division in your program changes wildly. It should not be
> possible for any library to undermine the basic mathematical
> expectations of my program. Doing so, or expecting the user to do
> extra work to guarantee the *common case*, is a recipe for serious
> failure.

There are a number of issues combined here, but I think they generally
reduce to these:

1. How do you model the abstractions that are number systems in the abstractions that are classes and methods.
2. Should the behavior of mathn be acceptable in the core language.

We seem to think of the basic mathematical operations +, -, *, / as
being roughly equal. But of these four, division on the integers is
distinct. The set of integers is closed under addition, subtraction,
and multiplication. Given any two integers, you can add, subtract, or
multiply them and get an integer. But the result of dividing one
integer by another is not always an integer. The integers are not
closed under division. In mathematics, whether a set is closed under
an operation is a significant property.

As such, there is nothing at all questionable about defining division
on the integers to be essentially floor(real(a)/real(b)) (where
real(x) returns the (mathematical) real number corresponding to the
value x, because the integers are embedded in the reals and the reals
are closed under division). You basically have five choices:

1. floor(real(a)/real(b))
2. ceil(real(a)/real(b))
3. round(real(a)/real(b)) where round may use floor or ceil
4. real(a)/real(b)
5. raise an exception

In computer programming, there are a number of reasons for choosing 1,
2 or 3 but basically it is because that's the only way to get the
"closest" integer (i.e. define division in a way that the integers are
closed under the operation) . Convention has selected option 1.
Numerous algorithms are implemented with the assumption that integral
division is implement as option 1. It's not right or wrong, but the
convention has certain advantages. In programming, we are typically
implementing algorithms, not just "doing math" in some approximations
of these abstractions called number systems. Any system for doing math
takes serious steps to implement the real number system in as
mathematically correct form as possible.

My contention that we should always have two operators for integral
division is a compromise between the need to implement algorithms and
the desire to have concise "operator" notation for doing more

math-oriented computation. Given that programming in Ruby is more about algorithms than it is about doing math, it's unreasonable to expect (a/b).floor instead of a / b. At the same time, math-oriented programs are not going to be happy with a.quo b. The reasons for options 1 and 4 above are not mutually exclusive nor can one override the other.

The mathn library is clearly exploiting an implementation detail. Were Ruby implemented like Smalltalk (or Rubinius), mathn would have never been written as it is. The fact that it is even possible to load mathn results from the fact that countless critical algorithms in MRI are in the walled garden of C code. That's not true for your Ruby programs. Any algorithm you implement that relies on the very reasonable assumption of integral division will be broken by mathn.

You can say, "but mathn is in the standard library, you have to require it to use it". But that ignores the fact that requiring the library fundamentally changes assumptions that are at the very core of writing algorithms. Essential computer programming and mathn can never coexist without jumping through hoops.

This is the point were some grandiose scheme like selector namespaces are suggested. But I think the simple solution of two distinct operators handily solves the problem of the messy facts of mathematical number systems implemented in very untheoretic (i.e. really real) silicon.

As for which symbol to select, what about '/.' for real(a)/real(b).

Cheers,
Brian

=end

#5 - 09/21/2009 04:22 AM - runpaint (Run Paint Run Run)

=begin
I agree that mathn constitutes a problem, but due to the means it employs rather than the end it achieves. In Ruby 1.8 the expectation is that '/' performs integer division, and having such a fundamental be subverted is untenable.

However, under Ruby 1.9, where Rational is a native class, it is altogether more reasonable that, by default, the '/' method coerce to Rational (as Lisp), preferably, or Float (as Perl and Python 3) when necessary.

GvR rationalizes (pun unintentional) his decision to have Python 3 perform mathematical division by default, in contrast to Python 2.0 which behaves as Ruby sans mathn, as follows:

"When you write a function implementing a numeric algorithm (for example, calculating the phase of the moon) you typically expect the arguments to be specified as floating point numbers. However, since Python doesn't have type declarations, nothing is there to stop a caller from providing you with integer arguments. In a statically typed language, like C, the compiler will coerce the arguments to floats, but Python does no such thing – the algorithm is run with integer values until the wonders of mixed-mode arithmetic produce intermediate results that are floats."

"For everything except division, integers behave the same as the corresponding floating point numbers. For example, 1+1 equals 2 just as 1.0+1.0 equals 2.0, and so on. Therefore one can easily be misled to expect that numeric algorithms will behave regardless of whether they execute with integer or floating point arguments. However, when division is involved, and the possibility exists that both operands are integers, the numeric result is silently truncated, essentially inserting a potentially large error into the computation. Although one can write defensive code that coerces all arguments to floats upon entry, this is tedious, and it doesn't enhance the readability or maintainability of the code. Plus, it prevents the same algorithm from being used with complex arguments (although that may be highly special cases)."

http://python-history.blogspot.com/2009/03/problem-with-integer-division.html

http://research.microsoft.com/en-us/um/people/daan/download/papers/divmodnote.pdf and http://python.org/dev/peps/pep-0238/ are also relevant.
=end

#6 - 09/21/2009 04:52 AM - brixen (Brian Shirai)

=begin
Hi,

On Sun, Sep 20, 2009 at 9:19 AM, Rick DeNatale rick.denatale@gmail.com wrote:

> Actually in most languages which I've encountered, and that's quite a few. Mixed mode arithmetic has been implemented by having some kind of rules on how to 'overload' arithmetic operators based on the arguments, not by having different operator syntax.

> And those rules are usually based on doing conversions only when
> necessary so as to preserve what information can be preserved given
> the arguments,
>
> So, for example
>
>   integer op integer - normally produces an integer for all of the
> 'big four' + - * /
>   integer op float   - normally produces a float, as does float op integer
>
> As new numeric types are added, in languages which either include them
> inherently or allow them to be added, this pattern is usually
> followed.

This is a distinctly different issue. Mixed-type arithmetic in Ruby is handled by the #coerce protocol.

> > As for which symbol to select, what about '/.' for real(a)/real(b).
>
> Well, first the problem we are talking about is Rationals, not Floats,
> and second, what happens as more numeric classes are introduced.

The mathn library aliases Fixnum and Bignum #quo to #/. By default #quo returns a Float. Rational redefines #quo to produce a Rational rather than a Float.

But what class of object is not the point. It could be Complex. The point is that integers are not closed under division so you *must* choose one of the options if you expect a value when dividing any two integers.

The division operation is so fundamental that assumptions about it should not change under your feet. Having a separate operator that returns a different type when integral division would be undefined allows both normal algorithms and mathy stuff to coexist nicely. In my algorithms, I *never* want my integral division to suddenly return something non-integer. In my math codes, I almost never want my quotient truncated or rounded.

Cheers,
Brian

=end

#7 - 09/21/2009 05:18 AM - RickDeNatale (Rick DeNatale)

=begin
On Sun, Sep 20, 2009 at 3:51 PM, brian ford brixen@gmail.com wrote:

> Hi,
>
> On Sun, Sep 20, 2009 at 9:19 AM, Rick DeNatale rick.denatale@gmail.com wrote:
>
> > Actually in most languages which I've encountered, and that's quite a
> > few. Mixed mode arithmetic has been implemented by having some kind of
> > rules on how to 'overload' arithmetic operators based on the
> > arguments, not by having different operator syntax.
> >
> > And those rules are usually based on doing conversions only when
> > necessary so as to preserve what information can be preserved given
> > the arguments,
> >
> > So, for example
> >
> >   integer op integer - normally produces an integer for all of the
> > 'big four' + - * /
> >   integer op float   - normally produces a float, as does float op integer
> >
> > As new numeric types are added, in languages which either include them
> > inherently or allow them to be added, this pattern is usually
> > followed.

> This is a distinctly different issue. Mixed-type arithmetic in Ruby is handled by the #coerce protocol.

Not sure why it's distinctly different, what happens when a new numeric class is introduced, e.g. Rational, is what we seem to be talking about.

And #coerce is just an implemention detail whose motivation seems to be in line with what I'm saying.

> > As for which symbol to select, what about '/.' for real(a)/real(b).
>
> Well, first the problem we are talking about is Rationals, not Floats, and second, what happens as more numeric classes are introduced.

> The mathn library aliases Fixnum and Bignum #quo to #/. By default #quo returns a Float. Rational redefines #quo to produce a Rational rather than a Float.
>
> But what class of object is not the point. It could be Complex. The point is that integers are not closed under division so you *must* choose one of the options if you expect a value when dividing any two integers.

Right, and Ruby like most other languages made a choice to use integer division, rather than converting.

Smalltalk made another choice, to return a Fraction when dividing two integers.

In both cases, the / operator is effectively overloaded, and can return other kinds of numbers given different pairs of arguments.

> The division operation is so fundamental that assumptions about it should not change under your feet. Having a separate operator that returns a different type when integral division would be undefined allows both normal algorithms and mathy stuff to coexist nicely. In my algorithms, I *never* want my integral division to suddenly return something non-integer. In my math codes, I almost never want my quotient truncated or rounded.

Yes, I agree that I don't want the rules to change under my feet. I want a / b to give me the same integer as Ruby 1.8 sans mathn gives me when a and b are integers, and I expect 1 / 1.2 to give me the same float etc. I'm not sure I see the need for additional operators, but that's a side issue.

Run Paint Run suggested that 1.9 SHOULD produce a Rational or maybe a float as the result of dividing two integers, because "that what Guido would do."

The brutal facts are that there's is lots of code written in Ruby, and lots of that code uses integer divide, and would be broken if this change were made, it would be the same as silently including mathn in every existing ruby program, which seems like a bad idea.

Guess what! I did some experimentation with irb1.9 and was pleasantly surprised to find that 1.9 seems to be doing quite the opposite, it acts just like the "thought experiment" proposal I suggested here.

```
$ irb1.9
irb(main):001:0> Rational
=> Rational
irb(main):002:0> 1/2
=> 0
irb(main):003:0> Rational(1)
=> Rational(1, 1)
irb(main):004:0> 1.to_r
=> Rational(1, 1)
```

Which I guess indicates that "that's what Matz would do."

--
Rick DeNatale

Blog: http://talklikeaduck.denhaven2.com/
Twitter: http://twitter.com/RickDeNatale
WWR: http://www.workingwithrails.com/person/9021-rick-denatale
LinkedIn: http://www.linkedin.com/in/rickdenatale

=end

#8 - 09/21/2009 07:56 AM - runpaint (Run Paint Run Run)

=begin

> Run Paint Run suggested that 1.9 SHOULD produce a Rational or maybe a
> float as the result of dividing two integers, because "that what Guido
> would do."

He said, of course, no such thing. I suggest that when your strawman necessitates the sensationalist mis-characterization of another's position that amateur rhetoric may not be your calling. In fact, your pastiche is not even internally consistent because GvR did not advocate rational results.

I was simply illustrating that other, similar languages have faced this issue, and so providing a justification for, and the results of, their decisions.

> The brutal facts are that there's is lots of code written in Ruby, and
> lots of that code uses integer divide, and would be broken if this
> change were made, it would be the same as silently including mathn in
> every existing ruby program, which seems like a bad idea.

The same argument can always be rallied to support inertia. As we progress toward Ruby 2.0 it behooves us to revisit our prior decisions and consider whether they remain defensible in hindsight. Further, your analogy is flawed: it would not at all "be the same as silently including mathn in every existing ruby program", because neither would such a change in language semantics be ushered in "silently", nor does mathn perform only this function.

> Which I guess indicates that "that's what Matz would do."

The existence of an "infallible designer" would obviate this very bug tracker, as every aspect of the language could be reasoned so. It indicates what the current behavior is, nothing more.

As to the matter at hand, Brian's solution seems eminently reasonable for 1.8 at least; the desired behavior of mathn and '/' under 1.9 is perhaps a separate issue.
=end

#9 - 09/25/2009 07:23 PM - tadf (tadayoshi funaba)

- Assignee set to keiju (Keiju Ishitsuka)

=begin

=end

#10 - 10/12/2009 01:22 PM - RickDeNatale (Rick DeNatale)

=begin
On Sun, Sep 20, 2009 at 3:29 AM, brian ford brixen@gmail.com wrote:

> Hi,
>
> On Sat, Sep 19, 2009 at 1:04 AM, Charles Oliver Nutter
> headius@headius.com wrote:
>
> > On Sat, Sep 19, 2009 at 12:28 AM, Joel VanderWerf
> > vjoel@path.berkeley.edu wrote:
> >
> > > Perhaps it should be the responsibility of users of numeric operators to
> > > #floor explicitly when that is the intent, rather than rely on the (mostly
> > > standard, sometimes convenient, but questionable) 1/2==0 behavior. Doing so
> > > would make it easier to adapt the code to float, rational, or other numeric
> > > types.
> > >
> > > In your proposal, would Rational(1,3) be the preferred notation, since

> > 1/3==0? Or would there be something else, 1//3 or ...?
> >
> > I've always thought of mathn as a kind of alternate ruby, not just another
> > core library, hence to be used with caution...
>
> I think Brian Ford expressed what I feel best...there should always be
> another method or operator. Using another operator or method is an
> explicit "buy-in" by the user--rather than a potential (at some
> undetermined time in the future) that everything you know about
> integral division in your program changes wildly. It should not be
> possible for any library to undermine the basic mathematical
> expectations of my program. Doing so, or expecting the user to do
> extra work to guarantee the *common case*, is a recipe for serious
> failure.

There are a number of issues combined here, but I think they generally
reduce to these:

1. How do you model the abstractions that are number systems in the abstractions that are classes and methods.
2. Should the behavior of mathn be acceptable in the core language.

We seem to think of the basic mathematical operations +, -, *, / as
being roughly equal. But of these four, division on the integers is
distinct. The set of integers is closed under addition, subtraction,
and multiplication. Given any two integers, you can add, subtract, or
multiply them and get an integer. But the result of dividing one
integer by another is not always an integer. The integers are not
closed under division. In mathematics, whether a set is closed under
an operation is a significant property.

As such, there is nothing at all questionable about defining division
on the integers to be essentially floor(real(a)/real(b)) (where
real(x) returns the (mathematical) real number corresponding to the
value x, because the integers are embedded in the reals and the reals
are closed under division). You basically have five choices:

1. floor(real(a)/real(b))
2. ceil(real(a)/real(b))
3. round(real(a)/real(b)) where round may use floor or ceil
4. real(a)/real(b)
5. raise an exception

In computer programming, there are a number of reasons for choosing 1,
2 or 3 but basically it is because that's the only way to get the
"closest" integer (i.e. define division in a way that the integers are
closed under the operation) . Convention has selected option 1.
Numerous algorithms are implemented with the assumption that integral
division is implement as option 1. It's not right or wrong, but the
convention has certain advantages. In programming, we are typically
implementing algorithms, not just "doing math" in some approximations
of these abstractions called number systems. Any system for doing math
takes serious steps to implement the real number system in as
mathematically correct form as possible.

My contention that we should always have two operators for integral
division is a compromise between the need to implement algorithms and
the desire to have concise "operator" notation for doing more
math-oriented computation. Given that programming in Ruby is more
about algorithms than it is about doing math, it's unreasonable to
expect (a/b).floor instead of a / b. At the same time, math-oriented
programs are not going to be happy with a.quo b. The reasons for
options 1 and 4 above are not mutually exclusive nor can one override
the other.

Actually in most languages which I've encountered, and that's quite a
few. Mixed mode arithmetic has been implemented by having some kind of
rules on how to 'overload' arithmetic operators based on the
arguments, not by having different operator syntax.

And those rules are usually based on doing conversions only when
necessary so as to preserve what information can be preserved given
the arguments,

So, for example

```
 integer op integer - normally produces an integer for all of the
```

'big four' + - * /
integer op float - normally produces a float, as does float op integer

As new numeric types are added, in languages which either include them inherently or allow them to be added, this pattern is usually followed.

Smalltalk has the concept of generality of a number class. More general classes can represent more numbers, albeit with some potential for adding 'fuzziness' in the standard image Floats are the most general, then Fractions, then equally LargePositiveIntegers and LargeNegativeIntegers (which together serve the same role as Bignum in Ruby), then SmallInteger (Ruby's Fixnum).

> The mathn library is clearly exploiting an implementation detail. Were Ruby implemented like Smalltalk (or Rubinius), mathn would have never been written as it is. The fact that it is even possible to load mathn results from the fact that countless critical algorithms in MRI are in the walled garden of C code. That's not true for your Ruby programs. Any algorithm you implement that relies on the very reasonable assumption of integral division will be broken by mathn.

The problem with mathn is that it introduces new numeric types, and also changes the behavior of the existing types, particularly integer, so that when mathn is included

```
integer / integer produces a rational if the result can't be
```

reduced to an integer.

This is at odds with most languages and, as Charles points out, it effectively changes the 'rules of physics' for other code which is likely unaware that mathn has been introduced.

In Smalltalk, there is a standard Fraction class and integer division does in fact return a Fraction rather than an Integer. But that's known and expected by Smalltalk programmers.

> You can say, "but mathn is in the standard library, you have to require it to use it". But that ignores the fact that requiring the library fundamentally changes assumptions that are at the very core of writing algorithms. Essential computer programming and mathn can never coexist without jumping through hoops.

Yes this is a problem IMHO. The difference between Ruby and Smalltalk here is that one language starts out including Rationals/Fractions, and the other treats them as an optional add on which, when added, changes the rules.

> This is the point were some grandiose scheme like selector namespaces are suggested. But I think the simple solution of two distinct operators handily solves the problem of the messy facts of mathematical number systems implemented in very untheoretic (i.e. really real) silicon.
>
> As for which symbol to select, what about '/.' for real(a)/real(b).

Well, first the problem we are talking about is Rationals, not Floats, and second, what happens as more numeric classes are introduced.

Another alternative would be to change mathn (or maybe make a new alternative mathn for compatibility for programs already using mathn) which

```
1. Left 1 / 2 as producing the Integer 1
2. Allowed explicit instantiation of Rationals
       * Rational.new(1,2)  # i.e. make the new method public.
       * Change Object#Rational to always return a Rational for an
```

integer argument, with a denominator of 1.
* Integer#to_rational which could be implemented as:
class Integer
def to_rational
Rational(self)
end
end

Then rational arithmetic could be implemented so that

```
  5 / 3  => 1
  5.to_rational / 3 => 5 / 3
  5 / 3.to_rational => 5 / 3

  (5.to_r / 3).to_i => 1
```

Which would be in-line with standard arithmetic in Ruby IMHO.

Note: it might be more ruby-like to name the coercion method to_r
instead of to_rational, but that might be confused by some as meaning
something else like to real, although I don't really thing that that
would be that much of an issue.

--
Rick DeNatale

Blog: http://talklikeaduck.denhaven2.com/
Twitter: http://twitter.com/RickDeNatale
WWR: http://www.workingwithrails.com/person/9021-rick-denatale
LinkedIn: http://www.linkedin.com/in/rickdenatale

=end

#11 - 09/14/2010 04:14 PM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Assigned

=begin

=end

#12 - 06/26/2011 01:53 PM - akr (Akira Tanaka)

- Project changed from Ruby to Ruby trunk

#13 - 11/17/2012 01:29 AM - headius (Charles Nutter)

This is still an issue. Requiring a standard library should never change the result of a Fixnum operation. The current behavior is a terrible bug.

#14 - 11/17/2012 02:39 AM - marcandre (Marc-Andre Lafortune)

- Description updated

- Category set to lib

It does create some problems in real apps (e.g. https://github.com/rails/rails/pull/8222)

The problem is compatibility. I always thought that it was the intent of mathn to change the semantics of / so that the same operations would have different meanings. For example Matrix[[1]] / 2 returns different results depending on the presence of mathn or not.

Avoiding the problem is trivial: when you mean "division with truncation to integer", use Integer#div instead of Integer#/.

div has the advantage of being crystal clear, since foo.div(bar) always truncates down to an integer, while foo / bar can return any of Integer, Rational, Float, BigDecimal, Complex, Matrix, Vector, etc., depending on the classes of foo, bar (and if mathn is loaded or not)

What I'm trying to say is that, given the choices made in the past, using / instead of div can be seen as a user mistake, although it clearly is an easy one to make. Ideally, library/framework authors would be aware of this and reserve Integer#/ for mathematical libraries and use div where appropriate.

I don't see how we can change the current behavior, but adding a note in the doc for Integer#/ about this could not hurt.

#15 - 11/17/2012 07:16 PM - headius (Charles Nutter)

I sympathize with the desire to avoid breaking backward compatibility, but the idea that "10 / 2" is the wrong way and you should instead use "10.div(2)" is pretty anti-Ruby. This is terribly surprising and certainly not what any user would expect. I would suggest that a new method be added to Integer for the variable behavior...something like #ratio or #fraction.

Another example of why this sucks, from a Ruby implementer perspective...

In JRuby, the basic math operations are recognize and optimized for better performance...all except #/ because of the mathn problem. Silly, isn't it?

#16 - 08/27/2014 02:44 AM - hsbt (Hiroshi SHIBATA)

- Related to Feature #10169: It might be better to make Mathn class deprecated added

#17 - 08/27/2014 02:45 AM - hsbt (Hiroshi SHIBATA)

- Status changed from Assigned to Closed

mathn library is deprecated on trunk [Feature #10169]

#18 - 08/27/2014 04:25 AM - headius (Charles Nutter)

Bravo!