Ruby trunk - Bug #7829

Rounding error in Ruby Time

02/12/2013 05:52 AM - loirotte (Philippe Dosch)

Status:	Closed		
Priority:	Normal		
Assignee:	akr (Akira Tanaka)		
Target version:	2.6		
ruby -v:	ruby 1.9.3p194 (2012-04-20 revision 35410) [i686-linux]	**Backport:**	

Description

Even if I know the precision errors related to the implementation of IEEE 754 floating values, I'm very surprised of:

irb(main):001:0> Time.utc(1970,1,1,0,0,12.860).strftime("%H:%M:%S,%L")
=> "00:00:12,859"

The fact is that I obtain:

irb(main):002:0> Time.utc(1970, 1, 1, 0, 0, 12.860).subsec
=> (60517119992791/70368744177664)
irb(main):003:0> Time.utc(1970, 1, 1, 0, 0, 12.860).subsec.to_f
=> 0.8599999999999994

If I well understand the precision error that is reported for the 12th or 14th digit after the comma, I don't understand why the rounding process gives an unexpected result for this value. In this case, the last significant digit of my value is impacted, and it appears to be a embarrassing behavior. For other values, the obtained result is as expected:

irb(main):001:0> Time.utc(1970,1,1,0,0,12.880).strftime("%H:%M:%S,%L")
=> "00:00:12,880"

Moreover, this is a part of the Time class and I don't know any way to fix it in a program (and I don't know the full list of values reproducing this issue...)

Associated revisions

Revision f674d862 - 04/04/2013 01:37 PM - akr (Akira Tanaka)

- time.c (time_strftime): Describe %L and %N truncates digits under the specified length. [ruby-core:52130] [Bug #7829]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@40109 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 40109 - 04/04/2013 01:37 PM - akr (Akira Tanaka)

- time.c (time_strftime): Describe %L and %N truncates digits under the specified length. [ruby-core:52130] [Bug #7829]

Revision 40109 - 04/04/2013 01:37 PM - akr (Akira Tanaka)

- time.c (time_strftime): Describe %L and %N truncates digits under the specified length. [ruby-core:52130] [Bug #7829]

Revision 40109 - 04/04/2013 01:37 PM - akr (Akira Tanaka)

- time.c (time_strftime): Describe %L and %N truncates digits under the specified length. [ruby-core:52130] [Bug #7829]

Revision 40109 - 04/04/2013 01:37 PM - akr (Akira Tanaka)

- time.c (time_strftime): Describe %L and %N truncates digits under the specified length. [ruby-core:52130] [Bug #7829]

Revision 40109 - 04/04/2013 01:37 PM - akr (Akira Tanaka)

- time.c (time_strftime): Describe %L and %N truncates digits under the specified length. [ruby-core:52130] [Bug #7829]

Revision 40109 - 04/04/2013 01:37 PM - akr (Akira Tanaka)

- time.c (time_strftime): Describe %L and %N truncates digits under the specified length. [ruby-core:52130] [Bug #7829]

History

#1 - 02/12/2013 09:53 AM - akr (Akira Tanaka)

2013/2/12 loirotte (Philippe Dosch) loirotte@gmail.com:

> Bug #7829: Rounding error in Ruby Time
> https://bugs.ruby-lang.org/issues/7829
>
> Even if I know the precision errors related to the implementation of IEEE 754 floating values, I'm very surprised of:
>
> irb(main):001:0> Time.utc(1970,1,1,0,0,12.860).strftime("%H:%M:%S,%L")
> => "00:00:12,859"
>
> The fact is that I obtain:
>
> irb(main):002:0> Time.utc(1970, 1, 1, 0, 0, 12.860).subsec
> => (60517119992791/70368744177664)
> irb(main):003:0> Time.utc(1970, 1, 1, 0, 0, 12.860).subsec.to_f
> => 0.8599999999999994

1. Ruby parser converts 12.860 to 12.859999999999999431565811391919851303100585937 5.

12.860 is converted to IEEE 754 double value at Ruby parser.
The IEEE 754 double value is actually
12.859999999999999431565811391919851303100585937 5.

% ruby -e 'puts "%.100g" % 12.860'
12.85999999999999943156581139191985130310058593 75

Or 0x1.9b851eb851eb80000000p+3, in hexadecimal.

% ruby -e 'puts "%.20a" % 12.860'
0x1.9b851eb851eb80000000p+3

So Time.utc takes the value and Time#subsec returns the value under the point.

% ruby -e 'v = 12.860.to_r - 12; puts v, v.to_f'
60517119992791/70368744177664
0.8599999999999994

The Time object records the value given as is.

A proposal to change (fix) this behavior:
http://www.slideshare.net/mrkn/float-is-legacy

1. Time.strftime("%L") doesn't round, but floor.

%L (and %N) in Time.strftime doesn't round the value but floor the value.

Since 3-digits under the point of
0.859999999999999431565811391919851303100585937 5
is "859", %L shows "859".

rounding is not appropriate here.
It is clearely unexpected that %L for 0.99999 shows "1000".

1. Use Time#round.

There is a method to rounding Time: Time#round.

If you needs a Time value rouinding 3-digits under the second,
use time.round(3).

% ruby -e 'p Time.utc(1970,1,1,0,0,12.860).round(3).strftime("%H:%M:%S,%L")'
"00:00:12,860"
--
Tanaka Akira

#2 - 02/12/2013 10:48 AM - drbrain (Eric Hodel)

- Category changed from core to doc

- Target version changed from 1.9.3 to 2.6

Seems like %L uses floor, not rounding should be documented so I'll switch this to a DOC ticket.

#3 - 02/12/2013 05:51 PM - loirotte (Philippe Dosch)

drbrain (Eric Hodel) wrote:

> Seems like %L uses floor, not rounding should be documented so I'll switch this to a DOC ticket.

Improve the documentation makes sense, but is there really a good reason to floor this value instead of rounding it? I'm searching for examples where floor could be interesting, but I don't see any.

#4 - 02/12/2013 05:58 PM - loirotte (Philippe Dosch)

akr (Akira Tanaka) wrote:

> 1. Time.strftime("%L") doesn't round, but floor.
>
> %L (and %N) in Time.strftime doesn't round the value but floor the value.
>
> Since 3-digits under the point of
> 0.859999999999999943156581139191985130310058593 75
> is "859", %L shows "859".
>
> rounding is not appropriate here.
> It is clearely unexpected that %L for 0.99999 shows "1000".

Understood, except for your sentence "rounding is not appropriate here". The example clearly shows the opposite! Just surprised that floor is used instead of round in this situation...

> 1. Use Time#round.
>
> There is a method to rounding Time: Time#round.
>
> If you needs a Time value rouinding 3-digits under the second,
> use time.round(3).
>
> % ruby -e 'p Time.utc(1970,1,1,0,0,12.860).round(3).strftime("%H:%M:%S,%L")'
> "00:00:12,860"

It fixes my personal issue in a first time, thanks!

Philippe

#5 - 02/13/2013 02:59 AM - david_macmahon (David MacMahon)

On Feb 12, 2013, at 12:51 AM, loirotte (Philippe Dosch) wrote:

> Improve the documentation makes sense, but is there really a good reason to floor this value instead of rounding it? I'm searching for examples where floor could be interesting, but I don't see any.

IMHO, using floor when reducing the precision of a time representation is the appropriate thing to do to avoid "time travel" into the future. One real world reason for this has to do with timestamps of files. Consider a file stored on a filesystem that supports lower than native timestamp precision. If the filesystem were to round the time rather than truncate it then it would be possible for files to be timestamped with future times which can defeat/confuse tools that compare timestamps.

Dave

#6 - 02/18/2013 08:57 AM - ko1 (Koichi Sasada)

- Assignee set to akr (Akira Tanaka)

#7 - 02/21/2013 12:46 AM - loirotte (Philippe Dosch)

david_macmahon (David MacMahon) wrote:

> On Feb 12, 2013, at 12:51 AM, loirotte (Philippe Dosch) wrote:
>
> > Improve the documentation makes sense, but is there really a good reason to floor this value instead of rounding it? I'm searching for examples where floor could be interesting, but I don't see any.
>
> IMHO, using floor when reducing the precision of a time representation is the appropriate thing to do to avoid "time travel" into the future. One real world reason for this has to do with timestamps of files. Consider a file stored on a filesystem that supports lower than native timestamp precision. If the filesystem were to round the time rather than truncate it then it would be possible for files to be timestamped with future times which can defeat/confuse tools that compare timestamps.

Once again, more documentation about this behavior is a good thing, but I'm really not sure that this is the best solution for the original issue. Typing this instruction:

```
irb(main):001:0> Time.utc(1970,1,1,0,0,12.860).strftime("%H:%M:%S,%L")
=> "00:00:12,859"
```

gives an unexpected intuitive result. If I do understand some time travel side-effects for some uses, I remain convinced that these uses are not representative of all kinds of uses of Time class in Ruby. One of the original wishes of Matz was that the language is simple, clear, emphasizing human needs more than computers. With this single instruction, I think we get the inverse of this philosophy. I'm an assistant professor, teaching in university at master level. I confess being in trouble to explain this contradiction to my students. The time travel issue results intrinsically of external problems to Ruby. I see no relevant reason why Ruby should natively solve these problems, that are not directly related, to the detriment of other more general purposes.

#8 - 02/21/2013 04:23 AM - david_macmahon (David MacMahon)

On Feb 20, 2013, at 7:46 AM, loirotte (Philippe Dosch) wrote:

> Typing this instruction:
>
> ```
> irb(main):001:0> Time.utc(1970,1,1,0,0,12.860).strftime("%H:%M:%S,%L")
> => "00:00:12,859"
> ```
>
> gives an unexpected intuitive result.

I totally agree with you that this is an unexpected, unintuitive result. The "problem" arises from the fact that you are passing in a Float for the number of seconds yet I suspect that Time uses Rational to support arbitrary precision. The conversion from the 12.860 literal to double precision floating point is limited in precision. The nearest representable value in this case is less than the "true" value. Converting Float to Rational is "perfect" in that the conversion back to Float results in the same (limited precision) value. The storing of 12.86 also "wastes" some bits of precision on the integer portion of the value:

```
12.86-12
=> 0.8599999999999994
```

You can avoid this "problem" by passing in a Rational instead of a Float for the seconds:

```
irb(main):001:0> Time.utc(1970,1,1,0,0,Rational(12860,1000)).strftime("%H:%M:%S,%L")
=> "00:00:12,860"
```

The DateTime class, which I think also uses Rational internally, does not seem to suffer the same problem:

```
irb(main):001:0> DateTime.civil(1970,1,1,0,0,12.86).strftime('%H:%M:%S,%L')
=> "00:00:12,860"
```

If DateTime also got it wrong I'd say it's just a limitation of floating point representation. The fact that DateTime behaves as expected leads me to believe that maybe Time's implementation could be altered to match. My guess is that Time uses Float.to_r on the seconds parameter directly thereby getting a power-of-2 denominator in the Rational whereas DateTime defaults to nanosecond precision thereby getting a denominator of 86,400,000,000,000 (or a factor thereof) which is the number of nanoseconds per day.

Perhaps it would be a nice feature to allow user specified precision on instances of these classes. That can already be done by passing in a Rational, but it could be convenient to have a separate parameter for this purpose. For example, to limit an instance to millisecond precision:

```
Time.utc(1970, 1, 1, 0, 0, 12.86, precision: 1000)
```

I know that millisecond precision would normally be specified as 1e-3, but that gets into floating point issues so I think it's cleaner to specify precision using the inverse.

Not using precision (or precision=1)

```
12.86.to_r-12
=> (60517119992791/70368744177664)
```

Using precision=1000

```
Rational(12.86*1000).to_r/1000-12
=> (43/50)
```

This is not perfect since it still breaks when the integer portion in large, but it would work well for values representing seconds which are typically 60.0 (for leap seconds) or less. Maybe it would even be useful to add an optional precision parameter to Float#to_r, i.e. Float#to_r(precision=1), which would then return the equivalent of "Rational(self*precision, precision)".

Interestingly, Ruby 1.9 has String#to_r which leads to this::

```
Time.utc(1970,1,1,0,0,12.86.to_s.to_r).strftime("%H:%M:%S,%L")
=> "00:00:12,860"
```

Please let me know if this would be more appropriate for ruby-talk.

Thanks,
Dave

#9 - 02/21/2013 03:59 PM - akr (Akira Tanaka)

2013/2/21 loirotte (Philippe Dosch) loirotte@gmail.com:

> irb(main):001:0> Time.utc(1970,1,1,0,0,12.860).strftime("%H:%M:%S,%L")
> => "00:00:12,859"
>
> gives an unexpected intuitive result. If I do understand some time travel side-effects for some uses, I remain convinced that these uses are not representative of all kinds of uses of Time class in Ruby. One of the original wishes of Matz was that the language is simple, clear, emphasizing human needs more than computers. With this single instruction, I think we get the inverse of this philosophy. I'm an assistant professor, teaching in university at master level. I confess being in trouble to explain this contradiction to my students. The time travel issue results intrinsically of external problems to Ruby. I see no relevant reason why Ruby should natively solve these problems, that are not directly related, to the detriment of other more general purposes.

I hope people supports mrkn's proposal:
http://www.slideshare.net/mrkn/float-is-legacy

The proposal fixes this issue and abolish unintuitiveness of float literal.
--
Tanaka Akira

#10 - 02/22/2013 09:23 AM - david_macmahon (David MacMahon)

On Feb 20, 2013, at 10:57 PM, Tanaka Akira wrote:

> I hope people supports mrkn's proposal:
> http://www.slideshare.net/mrkn/float-is-legacy
>
> The proposal fixes this issue and abolish unintuitiveness of float literal.

It is an interesting idea. I like the concept, but when dealing with large amounts of floating point data (even using NArray and/or GSL) it seems like there could be a lot of conversion between Rational and Float. I also wonder/worry about the performance of Rational when dealing with very large or very small numbers especially for addition/subtraction when the LCM must be computed on very large denominator values. Then again, if I really care about computational performance I'll write a C extension. More often I just want things to work correctly and performance is a secondary concern. I guess I'm not opposed to the idea, but not a proponent either. It would be very interesting to see it in action.

Dave

#11 - 02/22/2013 09:29 AM - david_macmahon (David MacMahon)

On Feb 20, 2013, at 11:17 AM, David MacMahon wrote:

> Interestingly, Ruby 1.9 has String#to_r which leads to this::
>
> ```
> Time.utc(1970,1,1,0,0,12.86.to_s.to_r).strftime("%H:%M:%S,%L")
> => "00:00:12,860"
> ```

Even more interestingly, Ruby 1.9 also has Float#rationalize which leads to this:

irb(main):001:0> Time.utc(1970,1,1,0,0,12.86.rationalize).strftime("%H:%M:%S,%L")
=> "00:00:12,860"

What do people thing about changing num_exact() in time.c to call #rationalize for Floats rather than #to_r? Or perhaps call #rationalize on the object if it responds to #rationalize so that this won't be exclusive to Floats?

Dave

#12 - 04/03/2013 09:23 PM - akr (Akira Tanaka)

2013/2/22 David MacMahon davidm@astro.berkeley.edu:

> What do people thing about changing num_exact() in time.c to call #rationalize for Floats rather than #to_r? Or perhaps call #rationalize on the object if it responds to #rationalize so that this won't be exclusive to Floats?

I'm not sure Float#rationalize is good choice.
At least, I don't understand the behavior and
the document don't explain the behavior.
The document describes eps is choosen automatically.
I think it is not enough explanation.
--
Tanaka Akira

#13 - 04/04/2013 04:53 AM - david_macmahon (David MacMahon)

On Apr 3, 2013, at 5:15 AM, Tanaka Akira wrote:

> 2013/2/22 David MacMahon davidm@astro.berkeley.edu:
>
> > What do people thing about changing num_exact() in time.c to call #rationalize for Floats rather than #to_r? Or perhaps call #rationalize on the object if it responds to #rationalize so that this won't be exclusive to Floats?
>
> I'm not sure Float#rationalize is good choice.
> At least, I don't understand the behavior and
> the document don't explain the behavior.
> The document describes eps is choosen automatically.
> I think it is not enough explanation.

I agree that the documentation is not explicit on how eps is chosen automatically in Float#rationalize.

My assumption has been that eps is chosen such that the resulting Rational will convert to the exact same double precision value that was originally given. In other words, "f.rationalize.to_f == f" will always be true assuming #rationalze doesn't raise FloatDomainError (e.g. if f is NaN or Infinity).

I have assumed that Float#rationalize is based on the same concept as the atod function described in this paper:

http://www.ampl.com/REFS/rounding.pdf

Based on some quick tests, it seems like my assumptions are wrong (as assumptions often are!) at least on "ruby 1.9.3p194 (2012-04-20 revision 35410) [x86_64-linux]". My assumptions about Float#rationalize do not hold for small (absolute) values around and below 1.0e-17. String#to_r has even more problem cases. Here are two examples:

```
f=57563.232824357045
=> 57563.232824357045

puts "%016x\n"*5 % [f, f.to_r.to_f, f.to_s.to_f, f.to_s.to_r.to_f, f.rationalize.to_f].pack('D').unpack('Q*')
40ec1b67734c10e7
40ec1b67734c10e7
40ec1b67734c10e7
40ec1b67734c10e6 <=== String#to_r "error"
40ec1b67734c10e7
=> nil

f=1e-17
=> 1.0e-17

puts "%016x\n"*5 % [f, f.to_r.to_f, f.to_s.to_f, f.to_s.to_r.to_f, f.rationalize.to_f].pack('D').unpack('Q*')
3c670ef54646d497
3c670ef54646d497
```

```
3c670ef54646d497
3c670ef54646d497
3c670ef54646d498 <=== Float#rationalize "error"
=> nil
```

I regard the String#to_r error to be a bug (i.e unintended and undesirable behavior). I find the Float#rationalize error to be undesirable behavior (IMHO), but since the documentation is vague I can't really say that it is unintended behavior.

In both cases, however, the error is very small (just one LSb of the mantissa).

Dave

#14 - 04/04/2013 10:53 AM - akr (Akira Tanaka)

2013/4/4 David MacMahon davidm@astro.berkeley.edu:

```
f=57563.232824357045
=> 57563.232824357045

puts "%016x\n"5 % [f, f.to_r.to_f, f.to_s.to_f, f.to_s.to_r.to_f, f.rationalize.to_f].pack('D').unpack('Q*')
40ec1b67734c10e7
40ec1b67734c10e7
40ec1b67734c10e7
40ec1b67734c10e6 <=== String#to_r "error"
40ec1b67734c10e7
=> nil
```

I don't think that String#to_r is wrong.

```
% ruby -e 'f=57563.232824357045
p f, f.to_s, f.to_s.to_r
'
57563.232824357045
"57563.232824357045"
(11512646564871409/200000000000)
```

String#to_r is correct because
57563.232824357045 == 11512646564871409/200000000000 in mathematical sense.

The "error" is caused by Froat#to_s and it is expected.

> I regard the String#to_r error to be a bug (i.e unintended and undesirable behavior).

I don't think so.

> I find the Float#rationalize error to be undesirable behavior (IMHO), but since the documentation is vague I can't really say that it is unintended behavior.

I have no idea about Float#rationalize.

Anyway, I'm sure now that Float#rationalize should not be used
internally/automatically.

Anyone can use it as Time.utc(1970,1,1,0,0,12.860.rationalize) and it
may (or may not) solve problem, though.
--
Tanaka Akira

#15 - 04/04/2013 10:37 PM - akr (Akira Tanaka)

- Status changed from Open to Closed

- % Done changed from 0 to 100

This issue was solved with changeset r40109.
Philippe, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

- time.c (time_strftime): Describe %L and %N truncates digits under the specified length. [ruby-core:52130] [Bug #7829]

#16 - 04/05/2013 10:23 AM - david_macmahon (David MacMahon)

On Apr 3, 2013, at 6:30 PM, Tanaka Akira wrote:

> 2013/4/4 David MacMahon davidm@astro.berkeley.edu:
>
> ```
> f=57563.232824357045
> => 57563.232824357045
>
> puts "%016x\n"*5 % [f, f.to_r.to_f, f.to_s.to_f, f.to_s.to_r.to_f, f.rationalize.to_f].pack('D').unpack('Q*')
> 40ec1b67734c10e7
> 40ec1b67734c10e7
> 40ec1b67734c10e7
> 40ec1b67734c10e6 <=== String#to_r "error"
> 40ec1b67734c10e7
> => nil
> ```
>
> I don't think that String#to_r is wrong.
>
> ```
> % ruby -e 'f=57563.232824357045
> p f, f.to_s, f.to_s.to_r
> '
> 57563.232824357045
> "57563.232824357045"
> (11512646564871409/200000000000)
> ```
>
> String#to_r is correct because
> 57563.232824357045 == 11512646564871409/200000000000 in mathematical sense.
>
> The "error" is caused by Froat#to_s and it is expected.

Of course you're right about String#to_r being correct. I think Float#to_s is correct as well. I think the problem is actually in Rational#to_f.

Each distinct Float value has (or should have, IMHO) an unambiguous String representation such that f.to_s.to_f == f, discounting NaN and Infinity for which this relationship doesn't hold due to a limitation (bug?) of String#to_f.

String#to_r works correctly as you pointed out.

The problem occurs because the Rational returned by String#to_r is reduced. When converting the reduced fraction of this example to Float, Rational#to_f effectively computes:

```
11512646564871409.to_f/200000000000.to_f
=> 57563.23282435704 <=== does NOT equal original value
```

instead of the un-reduced computation of:

```
57563232824357045.to_f/1000000000000.to_f
=> 57563.232824357045 <=== DOES equal original value
```

As you can see, these two expressions do not product equal answers. This limitation of Rational#to_f can also be seen by using BigDecimal to convert from Rational to Float:

```
class Rational
def to_f_via_bd
(BigDecimal.new(numerator)/denominator).to_f
end
end

f=57563.232824357045
=> 57563.232824357045

f.to_s.to_r.to_f
=> 57563.23282435704 <=== does NOT equal f
```

```
f.to_s.to_r.to_f_via_bd
=> 57563.232824357045 <=== DOES equal f
```

This same limitation also explains the problem I saw with Float.rationalize:

```
1.501852784991644e-17.rationalize.to_f
=> 1.5018527849916442e-17 <=== does NOT equal original value

1.501852784991644e-17.rationalize.to_f_via_bd
=> 1.501852784991644e-17 <=== DOES equal original value
```

In an earlier message I wrote: "Converting Float to Rational is 'perfect' in that the conversion back to Float results in the same (limited precision) value." The above examples shows that this is not true. I think this could be considered a bug in Rational#to_f.

> Anyway, I'm sure now that Float#rationalize should not be used
> internally/automatically.

I agree with this. Float#rationalize returns a Rational that is an approximation of the Float. This approximation is good enough that converting the Rational back to Float (avoiding intermediate rounding errors!) returns the original Float, but the Rational is NOT an exact representation. This is not a problem when using a single DateTime object, but performing math on a DateTime object that contains such an approximation seems like a bad idea.

On the other hand, Float#to_s works well and String#to_r returns a Rational that exactly equals the floating point number represented by the String. What about changing num_exact() in time.c to handle Floats by converting to String and then to Rational rather than calling Float#to_r?

> Anyone can use it as Time.utc(1970,1,1,0,0,12.860.rationalize) and it
> may (or may not) solve problem, though.

Or even better: Time.utc(1970,1,1,0,0,12.860.to_s.to_r).

Dave

#17 - 04/05/2013 11:23 AM - akr (Akira Tanaka)

2013/4/5 David MacMahon davidm@astro.berkeley.edu:

> Of course you're right about String#to_r being correct. I think Float#to_s is correct as well. I think the problem is actually in Rational#to_f.

It is expected that Rational#to_f can error because Float has only
53bit mantissa but Rational can hold more digits.
(Ruby uses double type in C and it is usally IEEE 754 double which has
53bit mantissa.)

> Each distinct Float value has (or should have, IMHO) an unambiguous String representation such that f.to_s.to_f == f, discounting NaN and Infinity for which this relationship doesn't hold due to a limitation (bug?) of String#to_f.
>
> String#to_r works correctly as you pointed out.
>
> The problem occurs because the Rational returned by String#to_r is reduced. When converting the reduced fraction of this example to Float, Rational#to_f effectively computes:
>
> ```
> 11512646564871409.to_f/200000000000.to_f
> => 57563.23282435704 <=== does NOT equal original value
> ```

Float cannot represent 11512646564871409 exactly.

```
% ruby -e 'i = 11512646564871409; p i.to_s(2), i.to_s(2).length'
"1010001110011010110000111010001110110001111110011110001"
54
```

The result is just an (good) approximation.

> instead of the un-reduced computation of:
>
> ```
> 57563232824357045.to_f/1000000000000.to_f
> ```

```
=> 57563.232824357045 <=== DOES equal original value
```

Float cannot represent 57563232824357045 exactly, too.

```
% ruby -e 'i = 57563232824357045; p i.to_s(2), i.to_s(2).length'
"11001100100000010111010010001100100111100111000010110101"
56
```

The equality is just a luck.
There are 7 integers to print 57563.232824357045.

```
% ruby -e '-7.upto(7) {|off|
p((57563232824357045+off).to_f/1000000000000.to_f) }'
57563.23282435704
57563.23282435704
57563.23282435704
57563.23282435704
57563.23282435704
57563.23282435704
57563.23282435704
57563.232824357045
57563.232824357045
57563.232824357045
57563.232824357045
57563.232824357045
57563.232824357045
57563.232824357045
57563.23282435706
```

It seems your requirement is too strong for Float.
--
Tanaka Akira

#18 - 04/06/2013 04:29 AM - david_macmahon (David MacMahon)

On Apr 4, 2013, at 7:02 PM, Tanaka Akira wrote:

> It is expected that Rational#to_f can error because Float has only
> 53bit mantissa but Rational can hold more digits.
> (Ruby uses double type in C and it is usally IEEE 754 double which has
> 53bit mantissa.)

I understand that the Float returned by Rational#to_f has limited precision and often will only approximate but not equal the value represented by the Rational. But in the example of 57563.232824357045 we are talking about a Float value that is representable. I think it is reasonable to expect f.to_r.to_f == f. I think that this is possible, but it requires changing both Float#to_r and Rational#to_f and I do not have a sense of whether it is practical from a performance point of view.

Float#to_r effectively converts the sign and binary mantissa of the Float to Rational and then either multiplies it by 2**exponent or divides it by 2**(-exponent) if exponent is negative. This creates a Rational which accurately represents the value represented by the bits underlying the Float. IOW, it rationalizes the binary approximation represented by the Float rather than the corresponding decimal approximation of the Float (which is what f.to_s.to_r does). IMHO, it would be better to rationalize the decimal approximation as these examples show:

```
1e23.to_r
=> (99999999999999991611392/1) <=== That's not exactly 1e23
```

But luckily it rounds back to the original Float:

```
1e23.to_r.to_f
=> 1.0e+23
```

Unfortunately, doing math on it can bring us "bad luck":

```
(1e23.to_r/100).to_f
=> 99999999999999900000.0 <=== Should be 1.0e+21
```

It's only differs by the least significant bit of the mantissa, but doesn't follow the principle of least surprise. Converting the Float to Rational via String (i.e. rationalizing the decimal approximation) avoids this issue:

```
(1e23.to_s.to_r/100).to_f
=> 1.0e+21
```

While changing Float#to_r to do the equivalent of "self.to_s.to_r" leads to "better" (can you find any counter examples?) rationalizations, it does not deal with rounding issues when converting to Float. The current Rational#to_f converts numerator and denominator to Floats then divides them. This results in three potential roundings: one for numerator to Float, one for denominator to Float, and one for the quotient. Using higher than double precision internally (e.g. via BigDecimal) and then rounding only at the end when converting to Float will lead to higher quality results as this example (again) shows:

```
57563.232824357045.to_s.to_r.to_f
=> 57563.23282435704

require 'bigdecimal'; require 'bigdecimal/util'
=> true

57563.232824357045.to_s.to_r.to_d(18).to_f
=> 57563.232824357045
```

The only Floats I have found for which f.to_s.to_r.to_d(18).to_f == f does NOT hold are subnormals and I think that is exposing a bug in BigDecimal#to_f:

```
2.58485e-319.to_s.to_r.to_d(18)
=> #

2.58485e-319.to_s.to_r.to_d(18).to_f
=> Infinity
```

Maybe this is fixed in newer versions. I am running "ruby 1.9.3p194 (2012-04-20 revision 35410) [x86_64-linux]".

> It seems your requirement is too strong for Float.

I think having Float#to_r represent the decimal approximation of the Float would lead to less surprise. Until someone creates a patch and it is accepted, this can be accomplished by monkey patching Float#to_r (though performance may suffer).

I think having Rational#to_f use higher precision internally would lead to higher precision results. This could also be accomplished via monkey patching, perhaps as part of bigdecimal.

Dave

#19 - 04/06/2013 07:53 AM - akr (Akira Tanaka)

2013/4/6 David MacMahon davidm@astro.berkeley.edu:

> I understand that the Float returned by Rational#to_f has limited precision and often will only approximate but not equal the value represented by the Rational. But in the example of 57563.232824357045 we are talking about a Float value that is representable. I think it is reasonable to expect f.to_r.to_f == f. I think that this is possible, but it requires changing both Float#to_r and Rational#to_f and I do not have a sense of whether it is practical from a performance point of view.

57563.232824357045 is not representable as a Float.
f.to_r.to_f == f is true.

```
% ruby -e 'f = 57563.232824357045; p "%.1000g" % f, f.to_r.to_f == f'
"57563.232824357044592034071683883666992187 5"
true
```

The actual value of the Float value is
57563.232824357044592034071683883666992187 5.
--
Tanaka Akira

#20 - 04/09/2013 03:29 PM - david_macmahon (David MacMahon)

On Apr 5, 2013, at 3:34 PM, Tanaka Akira wrote:

> 57563.232824357045 is not representable as a Float.

Sorry, poor wording on my part. What I meant was that the Float created from the floating point literal 57563.232824357045 is displayed by #inspect and #to_s as "57563.232824357045". IOW, calling #to_s on the 57563.232824357045 literal returns a string that represents the same value as the literal even though the underlying bits store a different value. This is not true for all literals. For example, calling #to_s on the literal 57563.232824357044 will also return the string "57563.232824357045".

Essentially, Float#to_s returns the shortest decimal string (or one of several equally short strings) that is not closer to any other Float value. A Rational that exactly equals the value represented by that decimal string is also not closer to any other Float value, so converting it to Float via #to_f would be expected to return the original Float value, but that is not always the case.

> f.to_r.to_f == f is true.

Yes, I now realize that this will always be true. Even though Rational#to_f rounds the numerator and denominator to double precision before dividing and then rounds the quotient after dividing, this will never cause problems for Rationals created via Float#to_r (assuming Bignum#to_f works sanely) due to the way Float.to_r works.

I would also like f.to_s.to_r.to_f == f to always be true. This is is not always the case because f.to_s.to_r has factors of 2 and 5 in the denominator, so the reduced Rational in this case runs the risk of #to_f not returning the closest approximation to the original value. In extreme cases, f.to_s.to_r.to_f can return values two representable values away from the original:

```
f1=4.7622749438937484e-07
=> 4.7622749438937484e-07
f2=4.762274943893749e-07
=> 4.762274943893749e-07
f1 < f2
=> true
```

After converting to String then to Rational then back to Float, f2 retains its original value, but f1 becomes larger than f2!

```
f1srf=f1.to_s.to_r.to_f
=> 4.7622749438937494e-07
f2srf=f2.to_s.to_r.to_f
=> 4.762274943893749e-07
f1srf > f2srf <== NB: GREATER THAN
=> true
```

Getting back to the original post, Time.new converts its "seconds" parameter using num_exact(), which converts Floats (among other types) to Rational using #to_r. It then divmod's the value by 1 to get integer seconds and fractional seconds. The complaint in the original post was that using the literal 12.68 for the seconds parameter led Time#strftime's %L specifier to show 679 milliseconds rather than 680.

In an earlier post, I suggested modifying num_exact to convert Floats to Rational via Float#rationalize, but now I think that converting to String and then to Rational (or preferably a more direct approach that does the equivalent) would lead to the best user experience.

Converting Floats passed as "seconds" using Float#to_r assumes that people have 53 bits of precision in their "seconds" values. I suspect that this is not true for the vast majority of users. More likely they have millisecond, microsecond, or maybe nanosecond precision. People with higher (or non-decimal) precision will (or at least should, IMHO) be using Rationals already. Converting Float "seconds" to String and then to Rational makes the most sense (IMHO) as it preserves the decimal precision of the input. The (debatable) rounding issue of Rational#to_f is not really problematic for this use case since it does not affect values that are likely to be used for seconds.

Dave

#21 - 04/11/2013 01:53 PM - zzak (Zachary Scott)

So is this a documentation bug? I haven't read the entire discussion

On Tue, Apr 9, 2013 at 2:24 AM, David MacMahon
davidm@astro.berkeley.edu wrote:

> On Apr 5, 2013, at 3:34 PM, Tanaka Akira wrote:
>
> > 57563.232824357045 is not representable as a Float.
>
> Sorry, poor wording on my part. What I meant was that the Float created from the floating point literal 57563.232824357045 is displayed by #inspect and #to_s as "57563.232824357045". IOW, calling #to_s on the 57563.232824357045 literal returns a string that represents the same value as the literal even though the underlying bits store a different value. This is not true for all literals. For example, calling #to_s on the literal 57563.232824357044 will also return the string "57563.232824357045".

Essentially, Float#to_s returns the shortest decimal string (or one of several equally short strings) that is not closer to any other Float value. A Rational that exactly equals the value represented by that decimal string is also not closer to any other Float value, so converting it to Float via #to_f would be expected to return the original Float value, but that is not always the case.

> f.to_r.to_f == f is true.

Yes, I now realize that this will always be true. Even though Rational#to_f rounds the numerator and denominator to double precision before dividing and then rounds the quotient after dividing, this will never cause problems for Rationals created via Float#to_r (assuming Bignum#to_f works sanely) due to the way Float.to_r works.

I would also like f.to_s.to_r.to_f == f to always be true. This is is not always the case because f.to_s.to_r has factors of 2 and 5 in the denominator, so the reduced Rational in this case runs the risk of #to_f not returning the closest approximation to the original value. In extreme cases, f.to_s.to_r.to_f can return values two representable values away from the original:

```
f1=4.7622749438937484e-07
=> 4.7622749438937484e-07
f2=4.762274943893749e-07
=> 4.762274943893749e-07
f1 < f2
=> true
```

After converting to String then to Rational then back to Float, f2 retains its original value, but f1 becomes larger than f2!

```
f1srf=f1.to_s.to_r.to_f
=> 4.7622749438937494e-07
f2srf=f2.to_s.to_r.to_f
=> 4.762274943893749e-07
f1srf > f2srf <== NB: GREATER THAN
=> true
```

Getting back to the original post, Time.new converts its "seconds" parameter using num_exact(), which converts Floats (among other types) to Rational using #to_r. It then divmod's the value by 1 to get integer seconds and fractional seconds. The complaint in the original post was that using the literal 12.68 for the seconds parameter led Time#strftime's %L specifier to show 679 milliseconds rather than 680.

In an earlier post, I suggested modifying num_exact to convert Floats to Rational via Float#rationalize, but now I think that converting to String and then to Rational (or preferably a more direct approach that does the equivalent) would lead to the best user experience.

Converting Floats passed as "seconds" using Float#to_r assumes that people have 53 bits of precision in their "seconds" values. I suspect that this is not true for the vast majority of users. More likely they have millisecond, microsecond, or maybe nanosecond precision. People with higher (or non-decimal) precision will (or at least should, IMHO) be using Rationals already. Converting Float "seconds" to String and then to Rational makes the most sense (IMHO) as it preserves the decimal precision of the input. The (debatable) rounding issue of Rational#to_f is not really problematic for this use case since it does not affect values that are likely to be used for seconds.

Dave

#22 - 04/16/2013 04:29 PM - david_macmahon (David MacMahon)

On Apr 10, 2013, at 9:36 PM, Zachary Scott wrote:

> So is this a documentation bug? I haven't read the entire discussion

The discussion has wandered some from the original bug report. I don't think there is consensus yet on the disposition of this report. While Time's handling of Float arguments is numerically correct, it is generally inappropriate for most users (IMHO). I would classify it as a "real" bug rather than a documentation bug.

The Time class converts Float arguments (primarily seconds and microseconds) to Rationals using Float#to_r. Float#to_r creates a Rational that exactly represents the double precision value stored in the Float.

The problem with using Float#to_r to convert seconds (or microseconds) to Rational is that it assumes the double precision value stored in the Float is the true value (i.e. the exact value to the nearest unit of precision). In the vast majority of cases the double precision value stored in the Float is not the true value but rather a binary approximation of the true value.

Instead of using Float#to_r to capture the Float's binary approximation of the true value as a Rational, I think it would be preferable to capture a decimal approximation of the Float as a Rational One way to do this is to use Float#to_s to convert the seconds (or microseconds) to a decimal String approximation of the (binary approximation of the) true value, then use String#to_r to capture the decimal approximation of the Float as a Rational.

The three main reasons for preferring the decimal approximation for Float seconds (or microseconds) are:

1) No unpleasant surprises for users, especially when using Float literals as in the original bug report.

2) Almost all users will have a fairly limited decimal precision of time (i.e. milliseconds, microseconds, nanoseconds) so the decimal approximation of the Float is likely to be equal to the true value whereas the binary approximation will not be.

3) Users with very high and/or non-decimal precision of time are unlikely to be using Floats anyway.

Dave

#23 - 04/16/2013 04:53 PM - david_macmahon (David MacMahon)

I think using floor is correct when reducing the precision of time. We don't round "15:00 on Monday" to "Tuesday", it's still Monday. Likewise, we don't round "July 15, 2012" to "2013", it's still 2012. Why should we round "859999" microseconds to "860" milliseconds? It's still millisecond 859.

The real problem (IMHO) is that Time treats Float values as the exact value the user intended rather than an approximation of the value they intended. Please see my other reply to this topic that crossed paths with yours.

Dave

On Feb 11, 2013, at 5:48 PM, drbrain (Eric Hodel) wrote:

> Issue #7829 has been updated by drbrain (Eric Hodel).
>
> Category changed from core to DOC
> Target version changed from 1.9.3 to next minor
>
> ## Seems like %L uses floor, not rounding should be documented so I'll switch this to a DOC ticket.
>
> Bug #7829: Rounding error in Ruby Time
> https://bugs.ruby-lang.org/issues/7829#change-36155
>
> Author: loirotte (Philippe Dosch)
> Status: Open
> Priority: Normal
> Assignee:
> Category: DOC
> Target version: next minor
> ruby -v: ruby 1.9.3p194 (2012-04-20 revision 35410) [i686-linux]
>
> Even if I know the precision errors related to the implementation of IEEE 754 floating values, I'm very surprised of:
>
> irb(main):001:0> Time.utc(1970,1,1,0,0,12.860).strftime("%H:%M:%S,%L")
> => "00:00:12,859"
>
> The fact is that I obtain:
>
> irb(main):002:0> Time.utc(1970, 1, 1, 0, 0, 12.860).subsec
> => (60517119992791/70368744177664)
> irb(main):003:0> Time.utc(1970, 1, 1, 0, 0, 12.860).subsec.to_f
> => 0.8599999999999994
>
> If I well understand the precision error that is reported for the 12th or 14th digit after the comma, I don't understand why the rounding process gives an unexpected result for this value. In this case, the last significant digit of my value is impacted, and it appears to be a embarrassing behavior. For other values, the obtained result is as expected:
>
> irb(main):001:0> Time.utc(1970,1,1,0,0,12.880).strftime("%H:%M:%S,%L")
> => "00:00:12,880"
>
> Moreover, this is a part of the Time class and I don't know any way to fix it in a program (and I don't know the full list of values reproducing this issue...)
>
> --
> http://bugs.ruby-lang.org/

#24 - 04/16/2013 04:59 PM - david_macmahon (David MacMahon)

On Apr 16, 2013, at 12:44 AM, David MacMahon wrote:

> Please see my other reply to this topic that crossed paths with yours.
>
> Dave
>
> On Feb 11, 2013, at 5:48 PM, drbrain (Eric Hodel) wrote:

Wow! Talk about a rounding error in time! Sorry for replying as if your message from two month's ago were current.

Dave

#25 - 10/25/2013 12:33 PM - mpglover (Matt Glover)

david_macmahon (David MacMahon) wrote:

> The three main reasons for preferring the decimal approximation for Float seconds (or microseconds) are:
>
> 1) No unpleasant surprises for users, especially when using Float literals as in the original bug report.
>
> 2) Almost all users will have a fairly limited decimal precision of time (i.e. milliseconds, microseconds, nanoseconds) so the decimal approximation of the Float is likely to be equal to the true value whereas the binary approximation will not be.
>
> 3) Users with very high and/or non-decimal precision of time are unlikely to be using Floats anyway.

Just wanted to second this. I came rather close to filing a bug report about, I think, the same core issue. In part because these two tests led me to believe Ruby did prefer the decimal approximation:

- https://github.com/ruby/ruby/blob/f79aeb60e7e49c0430f2685719c8c4772cfc6f95/test/ruby/test_time.rb#L57
- https://github.com/ruby/ruby/blob/f79aeb60e7e49c0430f2685719c8c4772cfc6f95/test/ruby/test_time.rb#L75

However a similar looking test fails:

> assert_equal(0, (Time.at(1.9) + 0.1).usec)
> expected but was .

I assume that 999999 is the expected result for reasons described earlier in the ticket. Unless/until Dave's suggestions are implemented it may be helpful to adjust those tests to make it obvious the Float-related behavior is expected.