Ruby - Bug #4576

Range#step miss the last value, if end-exclusive and has float number

04/14/2011 10:46 PM - yimutang (Joey Zhou)

Status:	Closed	
Priority:	Normal	
Assignee:		
Target version:	1.9.4	
ruby -v:	-	**Backport:**

Description

=begin
Hi, I find that:

- if: range.exclude_end? == true
- and: any one in [begin_obj, end_obj, step] is a true Float(f.to_i != f)
- and: unless begin_obj + step*int == end_obj
- then: the result will miss the last value.

for example:

p (1...6.3).step.to_a # => [1.0, 2.0, 3.0, 4.0, 5.0], no 6.0
p (1.1...6).step.to_a # => [1.1, 2.1, 3.1, 4.1], no 5.1
p (1...6).step(1.1).to_a # => [1.0, 2.1, 3.2, 4.300000000000001], no 5.4

p (1.0...6.6).step(1.9).to_a # => [1.0, 2.9], no 4.8
p (1.0...6.7).step(1.9).to_a # => [1.0, 2.9, 4.8]
p (1.0...6.8).step(1.9).to_a # => [1.0, 2.9, 4.8], no 6.7

Maybe the #step is ok on integers, but there's something wrong if the range is end-exclusive and contain float numbers.
=end

Associated revisions

Revision a635de7d - 10/05/2011 07:35 AM - naruse (Yui NARUSE)

- numeric.c (ruby_float_step): improve floating point calculations.
  [ruby-core:35753] [Bug #4576]

- numeric.c (ruby_float_step): correct the error of floating point
  numbers on the excluding case.
  patched by Masahiro Tanaka [ruby-core:39608]

- numeric.c (ruby_float_step): use the end value when the current
  value is greater than or equal to the end value.
  patched by Akira Tanaka [ruby-core:39612]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@33407 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 033244c1 - 11/22/2011 01:47 AM - naruse (Yui NARUSE)

- numeric.c (ruby_float_step): improve floating point calculations.
  [ruby-core:35753] [Bug #4576]

- numeric.c (ruby_float_step): correct the error of floating point
  numbers on the excluding case.
  patched by Masahiro Tanaka [ruby-core:39608]

- numeric.c (ruby_float_step): use the end value when the current
  value is greater than or equal to the end value.

> patched by Akira Tanaka [ruby-core:39612]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@33811 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

History

#1 - 04/18/2011 09:59 AM - naruse (Yui NARUSE)

- Status changed from Open to Closed

Fixed in r31286.

#2 - 04/18/2011 12:06 PM - nagachika (Tomoyuki Chikanaga)

=begin
test_step_ruby_core_35753 seems depend on platform.
On i686-linux it fails.

1. Failure:
   test_step_ruby_core_35753(TestRange) [/home/chikanaga/opt/ruby-trunk/src/ruby/test/ruby/test_range.rb:190]:
   <3> expected but was
   <4>.

Rebuild with "cflags=-ffloat-store" option fixes this failure. It seems a "learn floating point number" issue.
May i partially revert an testcase of test_step_ruby_core_35753?
=end

#3 - 04/18/2011 12:23 PM - usa (Usaku NAKAMURA)

- ruby -v changed from ruby 1.9.2p180 (2011-02-18) [i386-mingw32] to -

Hello,

In message "[ruby-core:35804] [Ruby 1.9 - Bug #4576] Range#step miss the last value, if end-exclusive and has float number"
on Apr.18,2011 12:06:24, redmine@ruby-lang.org wrote:

> test_step_ruby_core_35753 seems depend on platform.
> On i686-linux it fails.

Ah, I doubt it. Thank you.

> May i partially revert an testcase of test_step_ruby_core_35753?

No.
I guess that 1.5 stepping doesn't have error.
Please test r31304.

Regards,

--
U.Nakamura usa@garbagecollect.jp

#4 - 08/30/2011 05:52 PM - vo.x (Vit Ondruch)

- File 0001-Fix-the-ronding-error-causing-wrong-evaluation-of-ra.patch added

Usaku NAKAMURA wrote:

> Hello,
>
> In message "[ruby-core:35804] [Ruby 1.9 - Bug #4576] Range#step miss the last value, if end-exclusive and has float number"
> on Apr.18,2011 12:06:24, redmine@ruby-lang.org wrote:
>
>> test_step_ruby_core_35753 seems depend on platform.
>> On i686-linux it fails.
>
> Ah, I doubt it. Thank you.
>
>> May i partially revert an testcase of test_step_ruby_core_35753?

> No.
> I guess that 1.5 stepping doesn't have error.
> Please test r31304.
>
> ## Regards,
>
> U.Nakamura usa@garbagecollect.jp

Can we reopen this issue please? The fix of unit tests was wrong, fixing consequences instead of reasons. Moreover, it was not backported to 1.8.7, so we hit the issue again when preparing updated package for RHEL 6.2. You can find the discussion about the issue at [1] including proposed solution [2]. Could you please review and apply the attached patch and also backport it to 1.8.7?

[1] https://bugzilla.redhat.com/show_bug.cgi?id=733372
[2] https://bugzilla.redhat.com/attachment.cgi?id=520552

#5 - 09/13/2011 06:59 PM - Anonymous

Can somebody please reopen this issue? Since the test suite fix is
apparently wrong.

Thank you.

Vit

#6 - 09/13/2011 07:29 PM - shyouhei (Shyouhei Urabe)

(09/13/2011 06:56 PM), VÃt Ondruch wrote:

> Can somebody please reopen this issue? Since the test suite fix is
> apparently wrong.

Hi, maybe I'm too unfamiliar with this area, but can you explain a bit
closer about how the test is apparently wrong? It seems OK to me.

#7 - 09/13/2011 07:46 PM - vo.x (Vit Ondruch)

Please first see the commit [1] and then tell me why the original test case should fail? Actually it fails on i386 and succeeds on x86_64 which is a bit suspicious. So I dig a bit deeper with my colleagues and we found that the test was just fine, but the implementation has issues on i386. This should be hopefully fixed with patch attached to this issue [2]. More detailed explanation can be found in Red Hat bugzilla [3].

[1] http://redmine.ruby-lang.org/projects/ruby-19/repository/revisions/31304/diff/test/ruby/test_range.rb
[2] http://redmine.ruby-lang.org/attachments/2039/0001-Fix-the-ronding-error-causing-wrong-evaluation-of-ra.patch
[3] https://bugzilla.redhat.com/show_bug.cgi?id=733372

#8 - 09/13/2011 07:49 PM - vo.x (Vit Ondruch)

This [1] is short C reproducer for the issue [2]. You can compare behavior on i386 and x86_64

[1] https://bugzilla.redhat.com/attachment.cgi?id=520087
[2] http://redmine.ruby-lang.org/issues/4576#note-2

#9 - 09/13/2011 08:23 PM - hramrach (Michal Suchanek)

On 13 September 2011 12:28, Urabe Shyouhei shyouhei@ruby-lang.org wrote:

> (09/13/2011 06:56 PM), Vít Ondruch wrote:
>
>> Can somebody please reopen this issue? Since the test suite fix is
>> apparently wrong.
>
> Hi, maybe I'm too unfamiliar with this area, but can you explain a bit
> closer about how the test is apparently wrong? It seems OK to me.

I see no error:

```
irb(main):001:0> p (1.0...6.8).step(1.9).to_a
[1.0, 2.9, 4.8, 6.7]
=> nil
irb(main):002:0> p (1.0...6.6).step(1.9).to_a
[1.0, 2.9, 4.8]
```

```
=> nil
irb(main):003:0> p (1...6.3).step.to_a
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
=> nil
irb(main):004:0> p (1.1...6).step.to_a
[1.1, 2.1, 3.1, 4.1, 5.1]
=> nil
irb(main):005:0> p (1...6).step(1.1).to_a
[1.0, 2.1, 3.2, 4.3, 5.4]
=> nil
```

ruby 1.8.7 (2011-06-30 patchlevel 352) [x86_64-linux]

```
irb(main):002:0> p (1.0...6.8).step(1.9).to_a
[1.0, 2.9, 4.8, 6.699999999999999]
=> [1.0, 2.9, 4.8, 6.699999999999999]
irb(main):003:0> p (1.0...6.6).step(1.9).to_a
[1.0, 2.9, 4.8]
=> [1.0, 2.9, 4.8]
irb(main):004:0> p (1...6.3).step.to_a
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
=> [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
irb(main):005:0>  p (1.1...6).step.to_a
[1.1, 2.1, 3.1, 4.1, 5.1]
=> [1.1, 2.1, 3.1, 4.1, 5.1]
irb(main):006:0> p (1...6).step(1.1).to_a
[1.0, 2.1, 3.2, 4.300000000000001, 5.4]
=> [1.0, 2.1, 3.2, 4.300000000000001, 5.4]
```

ruby 1.9.2p290 (2011-07-09 revision 32553) [x86_64-linux]

Thanks

Michal

#10 - 09/13/2011 08:50 PM - shyouhei (Shyouhei Urabe)

Vit Ondruch wrote:

> Please first see the commit [1] and then tell me why the original test case should fail?

Because no one guarantees that it should pass.

> Actually it fails on i386 and succeeds on x86_64 which is a bit suspicious.

It is a clear sign that you are "dancing with floats".

As Tomoyoki Chikanaga says in note #note-2 of this issue I believe this is a "learn floating point number" kind of thig.

#11 - 09/13/2011 09:00 PM - vo.x (Vit Ondruch)

Michal Suchanek wrote:

> On 13 September 2011 12:28, Urabe Shyouhei shyouhei@ruby-lang.org wrote:
>
>> (09/13/2011 06:56 PM), Vít Ondruch wrote:
>>
>>> Can somebody please reopen this issue? Since the test suite fix is
>>> apparently wrong.
>>
>> Hi, maybe I'm too unfamiliar with this area, but can you explain a bit
>> closer about how the test is apparently wrong? It seems OK to me.
>
> I see no error:
> ruby 1.9.2p290 (2011-07-09 revision 32553) [x86_64-linux]
>
> Thanks
>
> Michal

As I said, it is i386 or i686 issue, not x86_64.

#12 - 09/13/2011 09:04 PM - vo.x (Vit Ondruch)

Shyouhei Urabe wrote:

> Vit Ondruch wrote:
>
> > Please first see the commit [1] and then tell me why the original test case should fail?
>
> Because no one guarantees that it should pass.

It is a feature I guess, there is even test case for this unfortunately, so why should something pass on one platform and should not pass on another? That doesn't make sense.

> > Actually it fails on i386 and succeeds on x86_64 which is a bit suspicious.
>
> It is a clear sign that you are "dancing with floats".
>
> As Tomoyoki Chikanaga says in note #note-2 of this issue I believe this is a "learn floating point number" kind of thig.

No, it is not ... the main difference is if the float comparison is done in memory or in registers. Each have different precision and it popups on i386.

Here is long explanation copied from RH bugzilla:

Jaroslav Škarvada 2011-08-29 00:15:22 CEST

AFAIK by default on 32 bit machines GCC uses FPU instructions for better
compatibility with older machines, while on 64 bit machines it uses SSE
instructions for better performance.

The SSE offers more registers and consistency - the value always retain 64 bit,
while FPU offers better precision - it uses 80 bit intermediate values when
possible. And that's the source of inconsistency.

The core from your stripped down reproducer:

```
...
double beg = 1.0;
double unit = 1.2;
double end = 9.4;

printf("%d\n", 7 * unit + beg < end);
...
```

On 32 bit it returns 1, while on 64 bit it returns 0. Why? Analysis:

```
FPU instructions (on 32 bit):
7 * 1.199999999999999956 = 8.399999999999999689
8.399999999999999689 + 1 = 9.399999999999999689
9.399999999999999689 < 9.400000000000000355

SSE instructions (on 64 bit):
7 * 1.2 = 8.4000000000000004
8.4000000000000004 + 1 =  9.4000000000000004
9.4000000000000004 == 9.4000000000000004
```

Please note that the intermediate number 9.400000000000000355 can be rounded to
9.4000000000000004, but the comparison is done on FPU stack with the full
precision. And that's the problem.

You can force usage of the FPU on 64 bit, by compiling with -mfpmath=387 and
then the results will be the same on both arches. But please note the floating
points are tricky and it shouldn't be relied on internal rounding as in the
reproducer above.

#13 - 09/13/2011 09:19 PM - shyouhei (Shyouhei Urabe)

Vit Ondruch wrote:

> Shyouhei Urabe wrote:

> > Vit Ondruch wrote:
> >
> > > Please first see the commit [1] and then tell me why the original test case should fail?
> >
> > Because no one guarantees that it should pass.
>
> It is a feature I guess, there is even test case for this unfortunately, so why should something pass on one platform and should not pass on another? That doesn't make sense.

It is a hardware issue. So it is quite natural for one platform behaves differently than another.

> > > Actually it fails on i386 and succeeds on x86_64 which is a bit suspicious.
> >
> > It is a clear sign that you are "dancing with floats".
> >
> > As Tomoyoki Chikanaga says in note #note-2 of this issue I believe this is a "learn floating point number" kind of thig.
>
> No, it is not ... the main difference is if the float comparison is done in memory or in registers. Each have different precision and it popups on i386.

Yes, I know. And you cannot force your C compiler to use specific hardware (except by compiler flags). Your patch is insufficient for your needs. In fact, C lacks a way to specify how a floating number should be handled.

#14 - 09/13/2011 09:48 PM - vo.x (Vit Ondruch)

Shyouhei Urabe wrote:

> Vit Ondruch wrote:
>
> > Shyouhei Urabe wrote:
> >
> > > Vit Ondruch wrote:
> > >
> > > > Please first see the commit [1] and then tell me why the original test case should fail?
> > >
> > > Because no one guarantees that it should pass.
> >
> > It is a feature I guess, there is even test case for this unfortunately, so why should something pass on one platform and should not pass on another? That doesn't make sense.
>
> It is a hardware issue. So it is quite natural for one platform behaves differently than another.

So what is this feature for if you cannot rely on it nor there is way how to detect it? In this case, please remove such feature.

> > > > Actually it fails on i386 and succeeds on x86_64 which is a bit suspicious.
> > >
> > > It is a clear sign that you are "dancing with floats".
> > >
> > > As Tomoyoki Chikanaga says in note #note-2 of this issue I believe this is a "learn floating point number" kind of thig.
> >
> > No, it is not ... the main difference is if the float comparison is done in memory or in registers. Each have different precision and it popups on i386.
>
> Yes, I know. And you cannot force your C compiler to use specific hardware (except by compiler flags). Your patch is insufficient for your needs. In fact, C lacks a way to specify how a floating number should be handled.

My patch is sufficient to behave consistently on i386 and x86_64. It is not perfect, but far better than the current state.

#15 - 09/13/2011 10:13 PM - shyouhei (Shyouhei Urabe)

Vit Ondruch wrote:

> So what is this feature for if you cannot rely on it nor there is way how to detect it? In this case, please remove such feature.

No. Sorry. Ruby is not designed like that. Ruby's design is that it embraces the world we live, no matter it is ugly. Ruby do not hide its ugliness from your eyes (another good example is M17N design).

No one can argue that i386's floaing pointer arithmetic is chaos. Ruby just shows you the way it is.

#16 - 09/13/2011 11:17 PM - alix (Ales Marecek)

Hi!
Shyouhei, I can't agree with you. We have some fact that algorithm written in ruby doesn't work. I know this is not the problem of ruby but it is fixable. If we have something like tests there, we run it, what it is for when it fails? If there is no reason to fix bug like this what is the reason for fixing anything? Then, we could drop all tests and pretend everything works well. I think it should NOT work like this.
As Vit said, changing value in test case is absolutely nonsense, it's good for nothing, it's not a fix! Leave this alone is bad too because we have some algorithm that doesn't work - why use it then? Why have it in code?

Michal, do you think that this kind of information is correct ---> [1.0, 2.9, 4.8, 6.699999999999999]? I do NOT. What's wrong there, try to guess. Yes, you have three numbers with one decimal and the last one with many, that's not correct! That's not correct in Math view and same in Physics. Correct could be [1.0, 2.9, 4.8, 6.6] or [1.0, 2.9, 4.8, 6.7] OR [1.000000..., ...] (depends on round method.

#17 - 09/14/2011 12:01 AM - mrkn (Kenta Murata)

Hi,

you can use BigDecimal as following:

```
ruby-1.9-head -rbigdecimal -ve 'p (BigDecimal("1.0")..BigDecimal("6.8")).step(BigDecimal("1.9")).to_a'
ruby 1.9.4dev (2011-09-06 trunk 33199) [x86_64-darwin11.1.0]
-e:1: warning: (...) interpreted as grouped expression
[#BigDecimal:7fb9db05fd60,'0.1E1',9(18), #BigDecimal:7fb9db05eed8,'0.29E1',18(36), #BigDecimal:7fb9db05ec08,'0.48E1',18(36), #
BigDecimal:7fb9db05e960,'0.67E1',18(36)]
```

#18 - 09/14/2011 12:03 AM - mrkn (Kenta Murata)

Also, you can use Rational:

```
ruby-1.9-head -ve 'p (1 .. 68.quo(10)).step(19.quo(10)).to_a'
ruby 1.9.4dev (2011-09-06 trunk 33199) [x86_64-darwin11.1.0]
[1, (29/10), (24/5), (67/10)]
-e:1: warning: (...) interpreted as grouped expression
```

#19 - 09/14/2011 12:23 AM - mame (Yusuke Endoh)

Hello,

2011/9/13 Vit Ondruch v.ondruch@tiscali.cz:

> Please first see the commit [1] and then tell me why the original test case should fail? Actually it fails on i386 and succeeds on x86_64 which is a bit suspicious. So I dig a bit deeper with my colleagues and we found that the test was just fine, but the implementation has issues on i386. This should be hopefully fixed with patch attached to this issue [2]. More detailed explanation can be found in Red Hat bugzilla [3].

Vit, did you run Test E in the original ticket in Red Hat buzilla?
Indeed, Tests A--D and F behave as you expected with your patch applied:

```
$ ./miniruby -e 'p (1.0...9.4).step(1.2).to_a'
[1.0, 2.2, 3.4, 4.6, 5.8, 7.0, 8.2]
$ ./miniruby -e 'p (1.0...6.4).step(1.8).to_a'
[1.0, 2.8, 4.6]
$ ./miniruby -e 'p (1.0...7.3).step(2.1).to_a'
[1.0, 3.1, 5.2]
$ ./miniruby -e 'p (1.0...7.6).step(2.2).to_a'
[1.0, 3.2, 5.4]
$ ./miniruby -e 'p (1.0...146.6).step(18.2).to_a'
[1.0, 19.2, 37.4, 55.599999999999994, 73.8, 92.0,
110.19999999999999, 128.39999999999998]
```

However, Test E still looks like "apparently wrong" on my i386:

```
$ ./miniruby -e 'p (1.0...128.4).step(18.2).to_a'
[1.0, 19.2, 37.4, 55.599999999999994, 73.8, 92.0,
110.19999999999999, 128.39999999999998]
```

--

Yusuke Endoh mame@tsg.ne.jp

#20 - 09/14/2011 12:48 AM - alix (Ales Marecek)

Hi Kenta, thanks for the hint.
The bug is about "three-dotted" range, not "double-dotted". Test with Bigdecimal works, with "quo" does NOT, with "quo" using rational lib does.

```
$ ruby -rbigdecimal -e 'p (BigDecimal("1.0")...BigDecimal("6.4")).step(BigDecimal("1.8")).to_a'
[#BigDecimal:b789602c,'0.1E1',4(8), #BigDecimal:b7895eb0,'0.28E1',8(16), #BigDecimal:b7895e74,'0.46E1',8(16)]
$ ruby -e 'p (1...64.quo(10)).step(18.quo(10)).to_a'
[1.0, 2.8, 4.6, 6.4]
$ ruby -rrational -ve 'p (1...64.quo(10)).step(18.quo(10)).to_a'
[1, Rational(14, 5), Rational(23, 5)]
```

But it doesn't solve the issue we're discussing here: we have method that doesn't work in some circumstances which aren't hacks / exploits / non-standard use / on non-standard hardware... It happens on standard hardware with correct usage.

#21 - 09/14/2011 01:02 AM - mrkn (Kenta Murata)

Alers, what version of ruby do you use?

1.9.2p290, 1.9.3-preview1, and 1.9.4dev works well:

```
$ ruby-1.9-head -ve 'p (1...64.quo(10)).step(18.quo(10)).to_a'
ruby 1.9.4dev (2011-09-06 trunk 33199) [x86_64-darwin11.1.0]
[1, (14/5), (23/5)]
-e:1: warning: (...) interpreted as grouped expression

$ ruby-1.9.3-preview1 -ve 'p (1...64.quo(10)).step(18.quo(10)).to_a'
ruby 1.9.3dev (2011-07-31 revision 32789) [x86_64-darwin11.0.0]
[1, (14/5), (23/5)]
-e:1: warning: (...) interpreted as grouped expression

$ ruby-1.9.2-p290 -ve 'p (1...64.quo(10)).step(18.quo(10)).to_a'
ruby 1.9.2p290 (2011-07-09) [x86_64-darwin11.0.0]
-e:1: warning: (...) interpreted as grouped expression
[1, (14/5), (23/5)]
```

#22 - 09/14/2011 10:23 PM - hramrach (Michal Suchanek)

On 13 September 2011 17:19, Yusuke ENDOH mame@tsg.ne.jp wrote:

> Hello,
>
> 2011/9/13 Vit Ondruch v.ondruch@tiscali.cz:
>
>> Please first see the commit [1] and then tell me why the original test case should fail? Actually it fails on i386 and succeeds on x86_64 which is a bit suspicious. So I dig a bit deeper with my colleagues and we found that the test was just fine, but the implementation has issues on i386. This should be hopefully fixed with patch attached to this issue [2]. More detailed explanation can be found in Red Hat bugzilla [3].
>
> Vit, did you run Test E in the original ticket in Red Hat buzilla?
> Indeed, Tests A--D and F behave as you expected with your patch applied:
>
> ```
>  $ ./miniruby -e 'p (1.0...9.4).step(1.2).to_a'
>  [1.0, 2.2, 3.4, 4.6, 5.8, 7.0, 8.2]
>  $ ./miniruby -e 'p (1.0...6.4).step(1.8).to_a'
>  [1.0, 2.8, 4.6]
>  $ ./miniruby -e 'p (1.0...7.3).step(2.1).to_a'
>  [1.0, 3.1, 5.2]
>  $ ./miniruby -e 'p (1.0...7.6).step(2.2).to_a'
>  [1.0, 3.2, 5.4]
>  $ ./miniruby -e 'p (1.0...146.6).step(18.2).to_a'
>  [1.0, 19.2, 37.4, 55.599999999999994, 73.8, 92.0,
> 110.19999999999999, 128.39999999999998]
> ```
>
> However, Test E still looks like "apparently wrong" on my i386:
>
> ```
>  $ ./miniruby -e 'p (1.0...128.4).step(18.2).to_a'
>  [1.0, 19.2, 37.4, 55.599999999999994, 73.8, 92.0,
> 110.19999999999999, 128.39999999999998]
> ```

That happens on both i386 and amd64 for me.

The problem is different than the one discussed above, though.

Here the excluded end of the interval is reached (through rounding error) whereas in the above post the step before the excluded end was not reached.

Thanks

Michal

#23 - 09/16/2011 02:14 AM - marcandre (Marc-Andre Lafortune)

- Category set to core

- Target version set to 1.9.4

Hi,

In this long thread, I could not find a single argument against the (3-line) patch.

Does the patch cause a problem? No

Does the patch fix a platform inconsistency? Yes

The patch has been committed as r33282

The question that remains: there might be other issues like this one. Would compiling with "cflags=-ffloat-store" on affected platforms cause a significant performance loss?

Below are some comments on what has been said before:

On Sun, Apr 17, 2011 at 8:59 PM, redmine@ruby-lang.org wrote:

> Fixed in r31286.

The commit message of r31304 said "avoid float error". The correct phrasing is "ignore float error". Changing a failing test for another one that doesn't show the problem is not fixing anything. It is playing the ostrich (and yes, I know ostriches don't actually bury their head in the sand).

On Tue, Sep 13, 2011 at 7:50 AM, Shyouhei Urabe shyouhei@ruby-lang.org wrote:

> It is a clear sign that you are "dancing with floats".
>
> As Tomoyoki Chikanaga says in note #note-2 of this issue I believe this is a "learn floating point number" kind of thig.

Yes, floats can be complicated. No I wouldn't recommend to anyone to play with tight limits with floats. But here, it is simply not acceptable that (foo...bar).step(baz).to_a.last == bar. On any platform. This is not a question of learning floating point number.

On Tue, Sep 13, 2011 at 9:13 AM, Shyouhei Urabe shyouhei@ruby-lang.org wrote:

> No. Sorry. Ruby is not designed like that. Ruby's design is that it embraces the world we live, no matter it is ugly. Ruby do not hide its ugliness from your eyes.

What makes you think this? This is simply not true. Math in Ruby aims to be as platform independent as is reasonable. For example see Matz in [ruby-core:28212].

On Tue, Sep 13, 2011 at 11:01 AM, Kenta Murata muraken@gmail.com wrote:

> you can use BigDecimal as following:

I understand this is meant to be helpful to the original poster, but the reasoning "if there is a problem with Float, let's use BigDecimal" is flawed. It does not improve Ruby, it does not address the problem. Float methods should be fixed (within the inherent limits of Floats). Otherwise where draw the limit? Would it be ok if Float("3.0e-31").to_s == "3.0000000000000003e-31"?

On Tue, Sep 13, 2011 at 11:19 AM, Yusuke ENDOH mame@tsg.ne.jp wrote:

> However, Test E still looks like "apparently wrong" on my i386:
>
> ```
>  $ ./miniruby -e 'p (1.0...128.4).step(18.2).to_a'
>  [1.0, 19.2, 37.4, 55.599999999999994, 73.8, 92.0,
> 110.19999999999999, 128.39999999999998]
> ```

Here we have to accept this result, due to rounding error with Floats. 7 * 18.2 + 1.0 == 128.39999999999998 < 128.4
This result holds accross all IEEE 194 platforms, etc...

#24 - 09/16/2011 09:59 AM - mrkn (Kenta Murata)

Hi,

On Friday, September 16, 2011 at 02:14 , Marc-Andre Lafortune wrote:

> The patch has been committed as r33282

I cannot find tests for the commit r33282. Please tell me where the tests are.
If you write the tests in RubySpec, please describe the commit hashes of the corresponding
commits of RubySpec in the commit message of the commit of CRuby.

> Would it be ok if Float("3.0e-31").to_s == "3.0000000000000003e-31"?

I've objected to the behavior. Please look at the rejected my report.
http://redmine.ruby-lang.org/issues/4656

I think it is not a bug, but is a specification problem.
The issue is related to what is to_s.

--
Kenta Murata
Sent with Sparrow (http://www.sparrowmailapp.com)

#25 - 09/16/2011 10:23 AM - naruse (Yui NARUSE)

2011/9/16 Kenta Murata muraken@gmail.com:

> On Friday, September 16, 2011 at 02:14 , Marc-Andre Lafortune wrote:
>
>> The patch has been committed as r33282
>
> I cannot find tests for the commit r33282. Please tell me where the tests are.
> If you write the tests in RubySpec, please describe the commit hashes of the corresponding
> commits of RubySpec in the commit message of the commit of CRuby.
>
>> Would it be ok if `Float("3.0e-31").to_s

#26 - 09/16/2011 03:53 PM - Anonymous

Hi,

On Thu, Sep 15, 2011 at 8:57 PM, Kenta Murata muraken@gmail.com wrote:

> I cannot find tests for the commit r33282. Please tell me where the tests are.
> If you write the tests in RubySpec, please describe the commit hashes of the corresponding
> commits of RubySpec in the commit message of the commit of CRuby.

Good idea, I'll include the commit hash in future commits, with the
format [rubyspec:a9525edcd], unless there is another suggestion.

Note that the RubySpec is easy to find, either with git log --grep=4576 or grep -r 4576 ., or looking up the path of the
modified method(s) (here core/range/step_spec.rb). The hash in the
commit will be an additional way.

>> Would it be ok if Float("3.0e-31").to_s == "3.0000000000000003e-31"?
>
> I've objected to the behavior. Please look at the rejected my report.
> http://redmine.ruby-lang.org/issues/4656

Actually, in my example, 3.0e-31 != 3.0000000000000003e-31.

As for why up to 17 decimals may be needed to represent a float, see
my original issue #3273, or
http://en.wikipedia.org/wiki/IEEE_754-2008#Character_representation

#27 - 09/16/2011 03:53 PM - Anonymous

I re-committed as r33285 because

- Ruby should not keep it platform dependent with default compile
  flags [ruby-core:39566], [ruby-core:28212]
- this commit has corresponding test [rubyspec:a9525edcd]

Before reverting a commit, please give an example of a problem it can
cause or provide a failing test.

It a commit doesn't address all the possible scenarios, please re-open
the issue so additional patches can be made.

#28 - 09/16/2011 04:18 PM - naruse (Yui NARUSE)

Marc-Andre Lafortune wrote:

> I re-committed as r33285 because
>
> - Ruby should not keep it platform dependent with default compile
>   flags [ruby-core:39566], [ruby-core:28212]
> - this commit has corresponding test [rubyspec:a9525edcd]

Such style doesn't get consensus over the thread [ruby-core:39260].

> Before reverting a commit, please give an example of a problem it can
> cause or provide a failing test.
>
> It a commit doesn't address all the possible scenarios, please re-open
> the issue so additional patches can be made.

See following results, it includes your ruby and RubySpec change:
http://u32.rubyci.org/~chkbuild/ruby-trunk/log/20110915T230102Z.log.html.gz
http://u64.rubyci.org/~chkbuild/ruby-trunk/log/20110915T230102Z.log.html.gz
http://c5632.rubyci.org/~chkbuild/ruby-trunk/log/20110915T230102Z.log.html.gz
http://c5664.rubyci.org/~chkbuild/ruby-trunk/log/20110915T230301Z.log.html.gz
http://www.rubyist.net/~akr/chkbuild/debian/ruby-trunk/log/20110916T000500Z.log.html.gz

Now I doubt your understanding of floating point number, so please learn it and check your understanding
before recommit it.

#29 - 09/16/2011 04:23 PM - Anonymous

PS: I should have modified the commit message to say that this fixes
the problem for i386 (not amd64, I misread Michal's message)

On Fri, Sep 16, 2011 at 2:45 AM, Marc-Andre Lafortune
ruby-core-mailing-list@marc-andre.ca wrote:

> I re-committed as r33285 because
>
> - Ruby should not keep it platform dependent with default compile
>   flags [ruby-core:39566], [ruby-core:28212]
> - this commit has corresponding test [rubyspec:a9525edcd]
>
> Before reverting a commit, please give an example of a problem it can
> cause or provide a failing test.
>
> It a commit doesn't address all the possible scenarios, please re-open
> the issue so additional patches can be made.

#30 - 09/16/2011 05:04 PM - naruse (Yui NARUSE)

For people who get this issue, I describe additional comment,

CRuby doesn't specify its internal calculation of floatin point numbers, and ISO C also not.

This issue is by x87 FPU's internal calculation behavior.
x87 always calculates floats with 80bit precision on their register, even if the number is 64bit double.
And if the number is assigned a 64bit double variable, the number is rounded to 64bit.

So if (1.0+1.8*3) is calculated in 80bit precision, it doesn't equals to 6.4 in 64bit precision.
You may force FPU to compare them with 64bit by following ways:
(1) assign the number to a variable
(2) specify -ffloat-store or similar one
(3) specify -msse2 -mfpmath=sse or similar one

(1) is the one Vit suggested and marcandre commited.
It is still problematic because a smart C compiler will optimize out a variable.
Failed test results I showed in [ruby-core:39579] are because of it.
So you must add volatile to guard the variable from such optimization.
But it is barrier for optimization on non x87 FPU.

(2) is not so good because it effects above volatile effect all over the code.
It has speed penalty.

(3) uses SSE2 to calculate floating point numbers.
This doesn't have such speed penalty, but it is less portable.

#31 - 09/16/2011 05:53 PM - usa (Usaku NAKAMURA)

Hello,

In message "[ruby-core:39580] [Ruby 1.9 - Bug #4576] Range#step miss the last value, if end-exclusive and has float number"
on Sep.16,2011 17:04:46, naruse@airemix.jp wrote:

> (2) specify -ffloat-store or similar one

only for information:
on VC6, specifing -Op has same effect.
on VC8 and later, -fp:precise.
(I've not installed VC7 and VC7.1 to this machine, so I can't
check them. I guess that they are same as VC8.)

Regards,

U.Nakamura usa@garbagecollect.jp

#32 - 09/16/2011 10:53 PM - akr (Akira Tanaka)

2011/9/16 Marc-Andre Lafortune ruby-core@marc-andre.ca:

> Yes, floats can be complicated. No I wouldn't recommend to anyone to play with tight limits with floats. But here, it is simply not acceptable that
> `(foo...bar).step(baz).to_a.last

#33 - 09/16/2011 11:23 PM - hramrach (Michal Suchanek)

On 16 September 2011 15:49, Tanaka Akira akr@fsij.org wrote:

> 2011/9/16 Marc-Andre Lafortune ruby-core@marc-andre.ca:
>
> > Yes, floats can be complicated. No I wouldn't recommend to anyone to play with tight limits with floats. But here, it is simply not acceptable that (foo...bar).step(baz).to_a.last == bar. On any platform. This is not a question of learning floating point number.
>
> Interesting.
>
> Please show us an actual example of (foo...bar).step(baz).to_a.last == bar

== is meaningless with floats.

The previous issue (the value before the excluded end of the range
not being reached) was most likely the result of guarding against
this:

(1.0...128.4).step(18.2).to_a
=> [1.0, 19.2, 37.4, 55.599999999999994, 73.8, 92.0, 110.19999999999999, 128.4]

I am quite sure that the 128.4 are not the same.

Still Ruby does not display the difference so the result is quite confusing.

Thanks

Michal

#34 - 09/16/2011 11:23 PM - akr (Akira Tanaka)

2011/9/16 Michal Suchanek hramrach@centrum.cz:

> == is meaningless with floats.
>
> The previous issue (the value before the excluded end of the range
> not being reached) was most likely the result of guarding against
> this:
>
> (1.0...128.4).step(18.2).to_a
> => [1.0, 19.2, 37.4, 55.599999999999994, 73.8, 92.0, 110.19999999999999, 128.4]
>
> I am quite sure that the 128.4 are not the same.

% ./ruby -ve 'p((1.0...128.4).step(18.2).to_a.last == 128.4)'
ruby 1.9.4dev (2011-09-16 trunk 33286) [i686-linux]
true

Hm. It is same value in my environment.

I feel this can be considered as a bug.

It is also possible to argue that this is not a bug because == is meaningless with floats, though.

Tanaka Akira

#35 - 09/16/2011 11:23 PM - mame (Yusuke Endoh)

Hello,

2011/9/16 Michal Suchanek hramrach@centrum.cz:

> == is meaningless with floats.

I guess it is not essential for the problem Marc-Andre says.
We can read it as follows without ==:

(foo...bar).step(baz).all? {|n| n < bar }

--
Yusuke Endoh mame@tsg.ne.jp

#36 - 09/16/2011 11:53 PM - hramrach (Michal Suchanek)

On 16 September 2011 16:18, Yusuke ENDOH mame@tsg.ne.jp wrote:

> Hello,
>
> 2011/9/16 Michal Suchanek hramrach@centrum.cz:
>
>> == is meaningless with floats.
>
> I guess it is not essential for the problem Marc-Andre says.
> We can read it as follows without ==:
>
> (foo...bar).step(baz).all? {|n| n < bar }

Indeed:

(1.0...128.4).step(18.2).to_a.all? {|n| n < 128.4}

=> false

Thanks

Michal

#37 - 09/17/2011 12:05 AM - marcandre (Marc-Andre Lafortune)

- Status changed from Closed to Open

Hi,

Michal Suchanek wrote:

> == is meaningless with floats.
>
> I am quite sure that the 128.4 are not the same.

If foo != bar, it doesn't tell you all that much about foo and bar, as they could differ by a lot, or possibly by very little due to a calculation rounding.

But foo == bar is completely *meaningful* for two Floats. It means they are the exact same value, that the 64 bits are the same. There are no two 128.4 that are somehow the same but somehow different.

This bug is due on some platforms that compare a Float (i.e. double precision 64 bits) with an extended double register (80 bits) which later gets rounded down to a Float.

#38 - 09/17/2011 12:42 AM - shyouhei (Shyouhei Urabe)

Marc-Andre Lafortune wrote:

> On Tue, Sep 13, 2011 at 9:13 AM, Shyouhei Urabe shyouhei@ruby-lang.org wrote:
>
> > No. Sorry. Ruby is not designed like that. Ruby's design is that it embraces the world we live, no matter it is ugly. Ruby do not hide its ugliness from your eyes.
>
> What makes you think this? This is simply not true. Math in Ruby aims to be as platform independent as is reasonable. For example see Matz in [ruby-core:28212].

I'm pretty sure Matz wasn't interested in the current implementation. He always says what he wants i.e. his statements do not apply to the one we already have. OTOH I've never said about any future plans about this area. I'm not against a Ruby in 22nd century to fully comply IEEE floats. But today, it's really considerably painfully difficult to force an Intel chip to behave under IEEE arithmetic[1]. It's clearer than light that the current ruby don't pay the cost to achieve it.

[1] http://www.shudo.net/publications/java-hpc2000/shudo-Java4HPC-strictfp.pdf

#39 - 09/17/2011 01:10 AM - marcandre (Marc-Andre Lafortune)

I'd like to thank Vit Ondruch and Aleš Mareček for pointing out this issue, investigating it and providing insight as to how to fix it.

I am sorry that this problem has not been fixed yet. I completely agree that this is a bug and that Ruby_with_patch >= Ruby_without_patch.

I fail to understand how someone can think it is not a bug that should be fixed and that Ruby_with_patch < Ruby_without_patch. Or that the performance of Ruby_with_patch would be significantly affected (no performance cost has been shown, and I only own macs so I can't measure it).

I am ashamed that tests were modified to cover up the issue instead of addressing it.
I am ashamed that your repeated requests to reopen this issue were not respected.
I am ashamed that commits were made that worsen Ruby.
I am ashamed that so much arguing is going on for such an obvious defect.

I am leaving for a vacation and won't be back until November. I am unsure if I will have the time (and energy) to monitor the mailing list during that time.

With a bit of luck, when I come back, it will be accepted that (foo...bar).step(baz).to_a.last == bar is not acceptable. It will be accepted and remembered that floats can take between 15 and 17 digits to print. It will be obvious that committers should be allowed (if not encouraged) to write implementation independent tests in RubySpec. And respect in actions and posts will be valued.

Or maybe nothing will have changed.

I sure hope that Ruby 2.0 will not get rid of the Float class altogether, with arguments like "use BigDecimal or Rational" and "Float#== is meaningless".

#40 - 09/17/2011 07:29 AM - akr (Akira Tanaka)

2011/9/17 Marc-Andre Lafortune ruby-core@marc-andre.ca:

> I'd like to thank Vit Ondruch and Aleš Mareček for pointing out this issue, investigating it and providing insight as to how to fix it.
>
> I am sorry that this problem has not been fixed yet. I completely agree that this is a bug and that Ruby_with_patch >= Ruby_without_patch.

I don't think the patch is a appropriate fix for this problem.

The test, "n*unit+beg < end", is fragile.

n*unit+beg can be different from the true mathematical value because
floating point calculation errors, as you know.
It can be bigger or smaller than the true value.

So the test should be changed to "n*unit+beg - e < end" for a estimated
maximum error, e.

I beleave the result of "*" and "+" is the nearest representable value of
the true mathematical result.

So I guess the maximum error is abs(a*DBL_EPSILON) + abs(b*DBL_EPSILON)
where a = n*unit and b = a + beg.

The estimate may be too small as considering double rounding problem
of x86 80bit float issue, though.

Note that I am not a expert of floating point calculations. Is there someone can validate the above logic?

Tanaka Akira

#41 - 09/17/2011 09:23 AM - akr (Akira Tanaka)

2011/9/17 Tanaka Akira akr@fsij.org:

> I don't think the patch is a appropriate fix for this problem.
>
> The test, "n*unit+beg < end", is fragile.

I made a sample script to show the fragileness on x86_64 to explain
this is not the x86 80bit float issue.
(x86_64 doesn't use 80bit float.)

```
% ./ruby -v
ruby 1.9.4dev (2011-09-15 trunk 33274) [x86_64-linux]
% ./ruby -e '
h = Hash.new(0)
1000.times {
a = rand
b = a+rand*10000
s = (b - a) / 10
l = (a...b).step(s).to_a.length
h[l] += 1
}
p h
'
{10=>940, 11=>60}
```

In this script, the result length vary.

Some people may think this is not a bug because
floating point calculation may have errors.

But in the following script, which changes "a...b" to "a..b" from the above
script, the result length doesn't vary.

```
% ./ruby -e '
h = Hash.new(0)
1000.times {
a = rand
```

b = a+rand*10000
s = (b - a) / 10
l = (a..b).step(s).to_a.length
h[l] += 1
}
p h
'
{11=>1000}

This is because we tried to consider float errors
in Numeric#step for Ruby 1.8. [ruby-dev:20163]

The implementation is extracted and reused for Range#step for Ruby 1.9.
[ruby-dev:37691] r21298

The exclude_end support in the implementation is new
since Numeric#step don't have correspondence to exclude_end.

So my understanding of this problem is that no one implemented proper exclude_end support with well considered float errors, yet.

Tanaka Akira

#42 - 09/17/2011 02:23 PM - mame (Yusuke Endoh)

Hello,

2011/9/17 Tanaka Akira akr@fsij.org:

> But in the following script, which changes "a...b" to "a..b" from the above
> script, the result length doesn't vary.
>
> % ./ruby -e '
> h

#43 - 09/17/2011 03:59 PM - naruse (Yui NARUSE)

(2011/09/17 9:07), Tanaka Akira wrote:

> So my understanding of this problem is that no one implemented
> proper exclude_end support with well considered float errors, yet.

In my current understanding, the error is
fabs(beg) * epsilon + fabs(unit) * epsilon * n + fabs(end) * epsilon
= (fabs(beg) + fabs(end) + fabs(end-beg)) * epsilon
// ignore error over unit*n -> end-beg
But the correct error may be less than it.

diff --git a/numeric.c b/numeric.c
index 18f5e1c..459e209 100644
--- a/numeric.c
+++ b/numeric.c
@@ -1691,7 +1691,9 @@ ruby_float_step(VALUE from, VALUE to, VALUE step, int excl)
else {
if (err>0.5) err=0.5;
n = floor(n + err);

- [redacted]

- [redacted]

- [redacted]

- [redacted]
```
      for (i=0; i<n; i++) {
          rb_yield(DBL2NUM(i*unit+beg));
      }
```

--

NARUSE, Yui naruse@airemix.jp

#44 - 09/17/2011 04:23 PM - akr (Akira Tanaka)

2011/9/17 Yusuke ENDOH mame@tsg.ne.jp:

> Maybe this "consideration" causes the following behavior:
>
> p (1.0..12.7).step(1.3).all? {|n| n <= 12.7 } #=> false
> p (1.0..12.7).step(1.3).to_a
> #=> [1.0, 2.3, 3.6, 4.9, 6.2, 7.5, 8.8, 10.1, 11.4, 12.700000000000001]
>
> Is this ok?
> I guess the result length will vary if this is fixed simply.

I agree.

I can't remember a discussion about it.

However I can't remember a bug report about it around Ruby 1.8 era.

Tanaka Akira

#45 - 09/17/2011 04:23 PM - mame (Yusuke Endoh)

Hello,

2011/9/17 NARUSE, Yui naruse@airemix.jp:

> In my current understanding, the error is
> fabs(beg) * epsilon + fabs(unit) * epsilon * n + fabs(end) * epsilon
> = (fabs(beg) + fabs(end) + fabs(end-beg)) * epsilon
> // ignore error over unit*n -> end-beg
> But the correct error may be less than it.

I have no understanding of the error, but your patch does not
work in this case:

e = 1.0 + 1E-12
p (1.0 ... e).step(1E-16).all? {|n| n < e } #=> false
p (1.0 ... e).step(1E-16).to_a.last == e #=> true

--
Yusuke Endoh mame@tsg.ne.jp

#46 - 09/17/2011 06:29 PM - naruse (Yui NARUSE)

(2011/09/17 16:14), Yusuke ENDOH wrote:

> Hello,
>
> 2011/9/17 NARUSE, Yui naruse@airemix.jp:
>
>> In my current understanding, the error is
>> fabs(beg) * epsilon + fabs(unit) * epsilon * n + fabs(end) * epsilon
>> = (fabs(beg) + fabs(end) + fabs(end-beg)) * epsilon
>> // ignore error over unit*n -> end-beg
>> But the correct error may be less than it.
>
> I have no understanding of the error, but your patch does not
> work in this case:
>
> e = 1.0 + 1E-12
> p (1.0 ... e).step(1E-16).all? {|n| n < e } #=> false
> p (1.0 ... e).step(1E-16).to_a.last == e #=> true

Your case is not the target of my patch.

--
NARUSE, Yui naruse@airemix.jp

#47 - 09/17/2011 07:23 PM - masa16 (Masahiro Tanaka)

I have not been watching ruby-core, but let me give a comment for this issue.
I proposed Numeric#step algorithm for Float in [ruby-dev:20177],
but that was only for the include_end-case.

> (1...6.3).step.to_a # => [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
> (1.1...6).step.to_a # => [1.1, 2.1, 3.1, 4.1, 5.1]
> (1...6).step(1.1).to_a # => [1.0, 2.1, 3.2, 4.3, 5.4]
>
> (1.0...6.6).step(1.9).to_a # => [1.0, 2.9, 4.0]
> (1.0...6.7).step(1.9).to_a # => [1.0, 2.9, 4.8]
> (1.0...6.8).step(1.9).to_a # => [1.0, 2.9, 4.8, 6.7]

If this behaviour is expected, a possible algorithm is:

--- numeric.c (revision 33288)
+++ numeric.c (working copy)
@@ -1690,8 +1690,16 @@
}
else {
if (err>0.5) err=0.5;

- [redacted]

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
  ```
        for (i=0; i<n; i++) {
            rb_yield(DBL2NUM(i*unit+beg));
        }
  ```

Masahiro Tanaka

#48 - 09/18/2011 08:29 AM - akr (Akira Tanaka)

2011/9/17 Yusuke ENDOH mame@tsg.ne.jp:

> Maybe this "consideration" causes the following behavior:
>
> p (1.0..12.7).step(1.3).all? {|n| n <= 12.7 } #=> false
> p (1.0..12.7).step(1.3).to_a
> #=> [1.0, 2.3, 3.6, 4.9, 6.2, 7.5, 8.8, 10.1, 11.4, 12.700000000000001]
>
> Is this ok?
> I guess the result length will vary if this is fixed simply.

One idea is that yield the end value instead.

Index: numeric.c

--- numeric.c (revision 33291)
+++ numeric.c (working copy)

@@ -1693,7 +1693,10 @@ ruby_float_step(VALUE from, VALUE to, VA
n = floor(n + err);
if (!excl || ((long)n)*unit+beg < end) n++;
for (i=0; i<n; i++) {

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
  ```
      }
  }
  return TRUE;
  ```

--
Tanaka Akira

#49 - 09/18/2011 06:53 PM - hramrach (Michal Suchanek)

On 17 September 2011 07:05, Yusuke ENDOH mame@tsg.ne.jp wrote:

> Hello,
>
> 2011/9/17 Tanaka Akira akr@fsij.org:
>
>> But in the following script, which changes "a...b" to "a..b" from the above
>> script, the result length doesn't vary.
>>
>> % ./ruby -e '
>> h = Hash.new(0)
>> 1000.times {
>> a = rand
>> b = a+rand*10000
>> s = (b - a) / 10
>> l = (a..b).step(s).to_a.length
>> h[l] += 1
>> }
>> p h
>> '
>> {11=>1000}
>>
>> This is because we tried to consider float errors
>> in Numeric#step for Ruby 1.8. [ruby-dev:20163]
>
> Maybe this "consideration" causes the following behavior:
>
> p (1.0..12.7).step(1.3).all? {|n| n <= 12.7 } #=> false
> p (1.0..12.7).step(1.3).to_a
>   #=> [1.0, 2.3, 3.6, 4.9, 6.2, 7.5, 8.8, 10.1, 11.4, 12.700000000000001]
>
> Is this ok?
> I guess the result length will vary if this is fixed simply.

In absence of a good estimation of the floating point error which
would allow for this pre-calculated loop to work flawlessly in all
cases perhaps the loop could be made shorter and the last values
checked explicitly?

Thanks

Michal

#50 - 09/20/2011 05:59 PM - akr (Akira Tanaka)

2011/9/17 Masahiro TANAKA masa16.tanaka@gmail.com:

> I have not been watching ruby-core, but let me give a comment for this issue.

> I proposed Numeric#step algorithm for Float in [ruby-dev:20177],
> but that was only for the include_end-case.
>
> ```
> p (1...6.3).step.to_a # => [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
> p (1.1...6).step.to_a # => [1.1, 2.1, 3.1, 4.1, 5.1]
> p (1...6).step(1.1).to_a # => [1.0, 2.1, 3.2, 4.3, 5.4]
>
> p (1.0...6.6).step(1.9).to_a # => [1.0, 2.9, 4.0]
> p (1.0...6.7).step(1.9).to_a # => [1.0, 2.9, 4.8]
> p (1.0...6.8).step(1.9).to_a # => [1.0, 2.9, 4.8, 6.7]
> ```
>
> If this behaviour is expected, a possible algorithm is:

I'm glad to see your opinion.

However the algorithm doesn't solve [ruby-core:39602] and [ruby-core:39606].

```
% ./ruby -e 'a = (1.0..12.7).step(1.3).to_a; p a.all? {|n| n <= 12.7 }, a.last'
false
12.700000000000001
% ./ruby -e 'e = 1+1E-12; a = (1.0 ... e).step(1E-16).to_a; p a.all?
{|n| n < e }, a.last'
false
1.000000000001
```

They show (s..e).step(u) yields a number greater than e and
(s...e).step(u) still yields a number greater than or equal to e.

Do you have an opinion on that?

Tanaka Akira

#51 - 09/21/2011 11:27 AM - naruse (Yui NARUSE)

I made a patch which fixes following 3 cases:

- the error of loop count
- duplicated values from underflow
- excess of the end value

diff --git a/ChangeLog b/ChangeLog
index 0abb211..0f0c28b 100644
--- a/ChangeLog
+++ b/ChangeLog
@@ -1,3 +1,16 @@
+Wed Sep 21 11:17:22 2011 NARUSE, Yui naruse@ruby-lang.org
+

  - numeric.c (ruby_float_step): improve floating point calculations.

- [redacted] `[Bug #4576]`
  - numeric.c (ruby_float_step): correct the error of floating point

- [redacted]
- [redacted][ruby-core:39608]
  - numeric.c (ruby_float_step): use the end value when the current

- [redacted]
- [redacted][ruby-core:39612]

Tue Sep 20 18:08:51 2011 Nobuyoshi Nakada nobu@ruby-lang.org

```
* vm_insnhelper.c (vm_get_cvar_base): reduce duplicated checks and
```

diff --git a/numeric.c b/numeric.c

index 18f5e1c..973da1f 100644
--- a/numeric.c
+++ b/numeric.c
@@ -1689,11 +1689,27 @@ ruby_float_step(VALUE from, VALUE to, VALUE step, int excl)
if (unit > 0 ? beg <= end : beg >= end) rb_yield(DBL2NUM(beg));
}
else {

- [redacted]
  ```
    if (err>0.5) err=0.5;
  ```

- [redacted]
- [redacted]

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
  ```
    for (i=0; i<n; i++) {
  ```

- [redacted]

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
  ```
    }
  ```

  }
  return TRUE;
  diff --git a/test/ruby/test_float.rb b/test/ruby/test_float.rb
  index e77b9e6..4fc8a6b 100644
  --- a/test/ruby/test_float.rb
  +++ b/test/ruby/test_float.rb
  @@ -508,4 +508,39 @@ class TestFloat < Test::Unit::TestCase
  sleep(0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1)
  end
  end

- def test_step
- 1000.times do

- [redacted]

- [redacted]
- [redacted]
- [redacted]
- end
- prev = 0
- (1.0..(1.0+1E-15)).step(1E-16) do |current|
- [redacted]
- [redacted]
- end
- (1.0..12.7).step(1.3).each do |n|
- [redacted]
- end
- end
- def test_step_excl
- 1000.times do
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- end
- assert_equal([1.0, 2.9, 4.8, 6.699999999999999], (1.0...6.8).step(1.9).to_a)
- e = 1+1E-12
- (1.0 ... e).step(1E-16) do |n|
- [redacted]
- end
- end
  end

#52 - 09/21/2011 07:23 PM - masa16 (Masahiro Tanaka)

2011/9/20 Tanaka Akira akr@fsij.org:

> However the algorithm doesn't solve [ruby-core:39602] and [ruby-core:39606].
>
> % ./ruby -e 'a = (1.0..12.7).step(1.3).to_a; p a.all? {|n| n <= 12.7 }, a.last'
> false
> 12.700000000000001

In my opinion, this behaviour is acceptable because the last value
generated by step method is close to the given Range-end value.

In the numerical calculation, the uniformity of sequence is more important.
Uniformity means that, if the generated sequence is
{ x(0), x(1), ..., x(n-1) },
then x(i+1)-x(i) is constant.

I think your solution [ruby-core:39612] is acceptable because the
modification of the last value is small. If we need more uniformity of
the sequence, a possible algorithm is:

```
if (end < (n-1)*unit+beg) {
for (i=0; i<n; i++) {
rb_yield(DBL2NUM((n-1-i)/(n-1)*beg+i/(n-1)*end));
}
} else ..
```

```
    % ./ruby -e 'e = 1+1E-12; a = (1.0 ... e).step(1E-16).to_a; p a.all?
    {|n| n < e }, a.last'
    false
    1.000000000001
```

This problem is hard to solve because this is due to the accuracy of
floating point value. The last part of this sequence is;

```
$  ruby -e 'e=1+1E-12; y=0; a=(1.0..e).step(1E-16).map{|x|"%.20f"%x};
p a[-6..-1]'
["1.00000000000099964481",
"1.00000000000099964481",
"1.00000000000099986686",
"1.00000000000099986686",
"1.00000000000100008890",
"1.00000000000100008890"]
```

The same value appears consecutively. Therefore, even after the last
value is excluded, the last value is equal to the range-end. This is
because this calculation exceeds the capability of floating point
arithmetic. In my opinion, this case is not suitable for the test
case. You can also see

```
$ ruby -e 'e=1+1E-12; y=0; a=(1.0..e).step(1E-16).map{|x|s=x-y;y=x;s};
p a[-6..-1]'
[2.220446049250313e-16, 0.0, 2.220446049250313e-16, 0.0,
2.220446049250313e-16, 0.0]
```

The difference is not equal to given step argument. Even though
step*(n-1) == last-begin is still holds, This does not hold if you
decrease n, so I think the repeat times must not be decreased.

Masahiro Tanaka

#53 - 09/21/2011 09:29 PM - masa16 (Masahiro Tanaka)

I haven't explained the reason of the error estimation in
Range#step for Float;

```
   double n = (end - beg)/unit;
   double err = (fabs(beg) + fabs(end) + fabs(end-beg)) / fabs(unit) * epsilon;
```

The reason is as follows. (including unicode characters)
This is based on the theory of the error propagation;
http://en.wikipedia.org/wiki/Propagation_of_uncertainty

If f(x,y,z) is given as a function of x, y, z,
Δf (the error of f) can be estimated as:

```
Δf^2 = |∂f/∂x|^2*Δx^2 + |∂f/∂y|^2*Δy^2 + |∂f/∂z|^2*Δz^2
```

This is a kind of statistical' error. Instead, maximum' error
can be expressed as:

```
Δf = |∂f/∂x|*Δx + |∂f/∂y|*Δy + |∂f/∂z|*Δz
```

I considered the latter is enough for this case.
Now, the target function here is:

```
n = f(e,b,u) = (e-b)/u
```

The partial differentiations of f are:

```
∂f/∂e = 1/u
∂f/∂b = -1/u
∂f/∂u = -(e-b)/u^2
```

The errors of floating point values are estimated as:

```
Δe = |e|*ε
Δb = |b|*ε
Δu = |u|*ε
```

Finally, the error is derived as:

Δn = |∂n/∂e|*Δe + |∂n/∂b|*Δb + |∂n/∂u|*Δu*
*= |1/u||e|ε + |1/u||b|ε + |(e-b)/u^2||u|*ε
= (|e| + |b| + |e-b|)/|u|*ε

Masahiro Tanaka

#54 - 09/21/2011 11:23 PM - hramrach (Michal Suchanek)

On 21 September 2011 14:25, masa masa16.tanaka@gmail.com wrote:

> I haven't explained the reason of the error estimation in
> Range#step for Float;
>
>   double n = (end - beg)/unit;
>   double err = (fabs(beg) + fabs(end) + fabs(end-beg)) / fabs(unit) *
> epsilon;
>
> The reason is as follows. (including unicode characters)
> This is based on the theory of the error propagation;
> http://en.wikipedia.org/wiki/Propagation_of_uncertainty
>
> If f(x,y,z) is given as a function of x, y, z,
> Δf (the error of f) can be estimated as:
>
> Δf^2 = |∂f/∂x|^2*Δx^2 + |∂f/∂y|^2*Δy^2 + |∂f/∂z|^2*Δz^2
>
> This is a kind of statistical' error. Instead, maximum' error
> can be expressed as:
>
> Δf = |∂f/∂x|*Δx + |∂f/∂y|*Δy + |∂f/∂z|*Δz
>
> I considered the latter is enough for this case.
> Now, the target function here is:
>
> n = f(e,b,u) = (e-b)/u
>
> The partial differentiations of f are:
>
> ∂f/∂e = 1/u
> ∂f/∂b = -1/u
> ∂f/∂u = -(e-b)/u^2
>
> The errors of floating point values are estimated as:
>
> Δe = |e|*ε
> Δb = |b|*ε
> Δu = |u|*ε
>
> Finally, the error is derived as:
>
> Δn = |∂n/∂e|*Δe + |∂n/∂b|*Δb + |∂n/∂u|*Δu*
> *= |1/u||e|ε + |1/u||b|ε + |(e-b)/u^2||u|*ε
> = (|e| + |b| + |e-b|)/|u|*ε

Well, if you can calculate the maximum error and minimum error then
you can get the range over which you need to check *every* value if it
exceeds the end of the range. The estimated (~expected ~average) error
is not useful in this case. Or you can iterate over the intersection
of the original range and error range again with smaller error.

Thanks

Michal

#55 - 09/22/2011 01:53 PM - Anonymous

A basis that sums to unity reduces error, due to convex hull property:

int n; double min, max;
for (int i = 0; i <= n; ++ i) {
double u   = (double) i / (double) n;
double _u_1 = 1.0 - u;
double v   = min * u1 + max * u;
}

-- KAS

On 9/21/11 9:16 AM, Michal Suchanek wrote:

> On 21 September 2011 14:25, masamasa16.tanaka@gmail.com wrote:
>
>> I haven't explained the reason of the error estimation in
>> Range#step for Float;
>>
>> ```
>> double n = (end - beg)/unit;
>> double err = (fabs(beg) + fabs(end) + fabs(end-beg)) / fabs(unit) *
>> ```
>>
>> epsilon;
>>
>> The reason is as follows. (including unicode characters)
>> This is based on the theory of the error propagation;
>> http://en.wikipedia.org/wiki/Propagation_of_uncertainty
>>
>> If f(x,y,z) is given as a function of x, y, z,
>> Δf (the error of f) can be estimated as:
>>
>> Δf^2 = |∂f/∂x|^2*Δx^2 + |∂f/∂y|^2*Δy^2 + |∂f/∂z|^2*Δz^2
>>
>> This is a kind of statistical' error. Instead, maximum' error
>> can be expressed as:
>>
>> Δf = |∂f/∂x|*Δx + |∂f/∂y|*Δy + |∂f/∂z|*Δz
>>
>> I considered the latter is enough for this case.
>> Now, the target function here is:
>>
>> n = f(e,b,u) = (e-b)/u
>>
>> The partial differentiations of f are:
>>
>> ∂f/∂e = 1/u
>> ∂f/∂b = -1/u
>> ∂f/∂u = -(e-b)/u^2
>>
>> The errors of floating point values are estimated as:
>>
>> Δe = |e|*ε
>> Δb = |b|*ε
>> Δu = |u|*ε
>>
>> Finally, the error is derived as:
>>
>> Δn = |∂n/∂e|*Δe + |∂n/∂b|*Δb + |∂n/∂u|*Δu
>> = |1/u|*|e|*ε + |1/u|*|b|*ε + |(e-b)/u^2|*|u|*ε
>> = (|e| + |b| + |e-b|)/|u|*ε
>
> Well, if you can calculate the maximum error and minimum error then
> you can get the range over which you need to check *every* value if it
> exceeds the end of the range. The estimated (~expected ~average) error
> is not useful in this case. Or you can iterate over the intersection
> of the original range and error range again with smaller error.
>
> Thanks
>
> Michal

#56 - 09/26/2011 11:53 AM - naruse (Yui NARUSE)

2011/9/22 Kurt Stephens ks@kurtstephens.com:

> A basis that sums to unity reduces error, due to convex hull property:
>
> int n; double min, max;
> for (int i

#57 - 09/27/2011 07:47 PM - naruse (Yui NARUSE)

Masahiro Tanaka wrote:

> I think your solution [ruby-core:39612] is acceptable because the
> modification of the last value is small. If we need more uniformity of
> the sequence, a possible algorithm is:

I'm pro of this side.

> $ ruby -e 'e=1+1E-12; y=0; a=(1.0..e).step(1E-16).map{|x|s=x-y;y=x;s};
> p a[-6..-1]'
> [2.220446049250313e-16, 0.0, 2.220446049250313e-16, 0.0,
> 2.220446049250313e-16, 0.0]
>
> The difference is not equal to given step argument. Even though
> step*(n-1) == last-begin is still holds, This does not hold if you
> decrease n, so I think the repeat times must not be decreased.

Hmm, you're correct, I fixed a patch.

Updated patch is following:

diff --git a/numeric.c b/numeric.c
index 973da1f..5702d08 100644
--- a/numeric.c
+++ b/numeric.c
@@ -1689,7 +1689,6 @@ ruby_float_step(VALUE from, VALUE to, VALUE step, int excl)
if (unit > 0 ? beg <= end : beg >= end) rb_yield(DBL2NUM(beg));
}
else {

- [redacted]
  ```
    if (err>0.5) err=0.5;
    if (excl) {
    if (n>0) {
  ```

@@ -1701,15 +1700,15 @@ ruby_float_step(VALUE from, VALUE to, VALUE step, int excl)
} else {
n = floor(n + err) + 1;
}

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]

- [redacted]
- [redacted]
- [redacted]
  }
- [redacted]
- [redacted]
  }

  }
  return TRUE;
  diff --git a/test/ruby/test_float.rb b/test/ruby/test_float.rb
  index 4fc8a6b..531ff04 100644
  --- a/test/ruby/test_float.rb
  +++ b/test/ruby/test_float.rb
  @@ -517,11 +517,7 @@ class TestFloat < Test::Unit::TestCase
  assert_equal(11, (a..b).step(s).to_a.length)
  end

- prev = 0
- (1.0..(1.0+1E-15)).step(1E-16) do |current|
- [redacted]
- [redacted]
- end

- assert_equal(11, (1.0..(1.0+1E-15)).step(1E-16).to_a.length)

  (1.0..12.7).step(1.3).each do |n|
  assert_operator(n, :<=, 12.7)

#58 - 10/02/2011 06:53 PM - masa16 (Masahiro Tanaka)

2011/9/17 Masahiro TANAKA masa16.tanaka@gmail.com:

> --- numeric.c (revision 33288)
> +++ numeric.c (working copy)
> @@ -1690,8 +1690,16 @@
> }
> else {
> if (err>0.5) err=0.5;
>
> - [redacted]
> - [redacted]
>
> - [redacted]
> - [redacted]
> - [redacted]
> - [redacted]
> - [redacted]
> - [redacted]
> - [redacted]
> - [redacted]
> - [redacted]

- [redacted]
  ```
      for (i=0; i<n; i++) {
          rb_yield(DBL2NUM(i*unit+beg));
      }
  ```

This patch involves a problem:
a=(1.0..1-2e-16).step(1.0).to_a; p a #=> [1.0] ; should be []

The following patch would be better.

--- numeric.c (revision 33377)
+++ numeric.c (working copy)
@@ -1690,9 +1690,18 @@
}
else {
if (err>0.5) err=0.5;

- [redacted]
- [redacted]
- [redacted]

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
  ```
          rb_yield(DBL2NUM(i*unit+beg));
      }
  }
  ```

Masahiro Tanaka

#59 - 10/05/2011 04:35 PM - naruse (Yui NARUSE)

- Status changed from Open to Closed

- % Done changed from 0 to 100

This issue was solved with changeset r33407.
Joey, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

- numeric.c (ruby_float_step): improve floating point calculations.
  [ruby-core:35753] [Bug #4576]

- numeric.c (ruby_float_step): correct the error of floating point
  numbers on the excluding case.
  patched by Masahiro Tanaka [ruby-core:39608]

- numeric.c (ruby_float_step): use the end value when the current
  value is greater than or equal to the end value.
  patched by Akira Tanaka [ruby-core:39612]

#60 - 10/07/2011 08:11 AM - naruse (Yui NARUSE)

- Status changed from Closed to Open

- % Done changed from 100 to 0

#61 - 10/12/2011 07:33 AM - naruse (Yui NARUSE)

I updated a patch.

```
% ./ruby -e 'a = (1.0..12.7).step(1.3).to_a; p a.all? {|n| n <= 12.7 }, a.last'

    false
    12.700000000000001
```

This issue has 3 options:
(1) ignore
(2) return end
(3) change step

In this thread, we don't ignore it, so (2) or (3).
On (3) its middle results are also changed but it seems hard to predict.
So I use (2) on this patch.

```
diff --git a/numeric.c b/numeric.c
index 6d3c143..37f91bc 100644
--- a/numeric.c
+++ b/numeric.c
@@ -1690,10 +1690,21 @@ ruby_float_step(VALUE from, VALUE to, VALUE step, int excl)
}
else {
if (err>0.5) err=0.5;
```

- [redacted]
- [redacted]
- [redacted]
- [redacted]

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]

- [redacted]
- [redacted]
  }

  }
  return TRUE;
  diff --git a/test/ruby/test_float.rb b/test/ruby/test_float.rb
  index e77b9e6..d163848 100644
  --- a/test/ruby/test_float.rb
  +++ b/test/ruby/test_float.rb
  @@ -508,4 +508,33 @@ class TestFloat < Test::Unit::TestCase
  sleep(0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1)
  end
  end

- def test_step
- 1000.times do

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- end
- (1.0..12.7).step(1.3).each do |n|

- [redacted]
- end
- end

- def test_step_excl
- 1000.times do

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- end
- assert_equal([1.0, 2.9, 4.8, 6.699999999999999], (1.0...6.8).step(1.9).to_a)
- e = 1+1E-12
- (1.0 ... e).step(1E-16) do |n|

- [redacted]
- end
- end
  end

#62 - 11/22/2011 10:47 AM - naruse (Yui NARUSE)

- Status changed from Open to Closed

- % Done changed from 0 to 100

This issue was solved with changeset r33811.
Joey, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

- numeric.c (ruby_float_step): improve floating point calculations.
  [ruby-core:35753] [Bug #4576]

- numeric.c (ruby_float_step): correct the error of floating point
  numbers on the excluding case.
  patched by Masahiro Tanaka [ruby-core:39608]

- numeric.c (ruby_float_step): use the end value when the current
  value is greater than or equal to the end value.
  patched by Akira Tanaka [ruby-core:39612]

Files

0001-Fix-the-ronding-error-causing-wrong-evaluation-of-ra.patch	1.02 KB	08/30/2011	vo.x (Vit Ondruch)