Ruby master - Feature #15811

Propsing new method for comparing equality of 2 (float) numbers relatively

04/29/2019 10:58 AM - yennguyenh (yen nguyen)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

Background

Equal comparison method between 2 float numbers returns unexpected results sometimes. Therefore, a relative comparison method is needed!

Proposal

A relative equal comparison method has been written based on a Python project! This method gives the approximation for the equal comparison based on two values: realative tolerance and absolute tolerance. Near zero value will also be considered carefully!

Implementation

The function for that would be called close?
close?(a, b, rel_tol, abs_tol)

a and b: are the two values to be tested to relative closeness

rel_tol: is the relative tolerance -- it is the amount of error allowed, relative to the larger absolute value of a or b. For example, to set a tolerance of 5%, pass tol=0.05. The default tolerance is 1E-9, which assures that the two values are the same within about 9 decimal digits. rel_tol must be greater than 0.0

abs_tol: is a minimum absolute tolerance level -- useful for comparisons near zero.

Evaluation of your implementation

By default, relative tolerance is 1E-9 which is relatively precise enough to compare two float numbers. However it can also be adjusted in case higher accuracy is requested. The absolute tolerance is by default 0.0 and need to be set in case of near-zero numbers.

Discussion

There are some test cases available for the method which has approved the accuracy of the method. BigNumbers and integers are also tested. However, more test cases are still needed to assure even better the accuracy of the method.

Gist

Relative equal comparison

https://gist.github.com/yennguyenh/63d5e7a11f354f796b43ada037c4b2c5

Test cases

https://gist.github.com/yennguyenh/2e81dc72b310cb9d886a82faf3d536ef

Related issues:

Related to Ruby master - Feature #10425: A predicate method to tell if a numb...	**Open**	**10/26/2014**

History

#1 - 04/29/2019 11:03 AM - yennguyenh (yen nguyen)

I don't know how to add the new method correctly. Please have a look at those gist and tell me what to do! Thank you :)

#2 - 04/29/2019 11:35 AM - osyo (manga osyo)

hi.
Here are the guidelines.
https://github.com/ko1/rubyhackchallenge

#3 - 04/29/2019 11:55 AM - shevegen (Robert A. Heiler)

Not sure if the name .close? is a good name to indicate a relative comparison. Note that
I have no real pro or con opinion, just pointing out that the name may not be ideal. I
don't have a good alternative proposal either; perhaps something a bit longer, two words?

Perhaps you could add a more specific example - might be useful. Will this method reside
on **Math** , for example? E. g.:

```
Math.close?(a, b, rel_tol, abs_tol)
```

Again, not having any preference myself here, just asking to see more details added to
the suggestion. The ruby team is open to adding/discussing changes/modifications based
on use cases; in my opinion, the more details can be added (when they are important),
the better.

To make it easier for others, I copy/pasted a part of your gist sample (excluding the
documents), just to make it simpler for people to read the code here; you can indent
via 4 ' ' and then the bugtracker here will correctly highlight the ruby code:

```
def self.close?(a, b, rel_tol: RELATIVE_TOLERANCE, abs_tol: ABSOLUTE_TOLERANCE)
  raise ArgumentError.new('Arguments must be numeric') unless (a.is_a?(Numeric) && b.is_a?(Numeric))
  raise ArgumentError.new('Error tolerance must positive') if (rel_tol < 0.0 || abs_tol < 0.0)

  # short-cut exact equality
  return true if a == b

  # check if any attribute is Infinite
  return false if a.infinite? || b.infinite?

  # weak comparition - the tolerance is scaled by the larger of 2 values
  abs_diff = (a - b).abs
  ((abs_diff <= (rel_tol * b).abs) ||
    (abs_diff <= (rel_tol * a).abs) ||
    (abs_diff <= abs_tol))
end
```

#4 - 04/29/2019 02:55 PM - wishdev (John Higgins)

The tests for this are incorrect and show why this does not work.

From the test gist

```
context 'Numbers between 1 and 0' do
  let (:absolute_tolerance) { 1.0E-14 }
  it 'returns true for same positive numbers' do
    expect(Math.close?(1000001.0E-15, 1000002.0E-15, abs_tol: absolute_tolerance)).to be_truthy
    expect(Math.close?(1000002.0E-15, 1000001.0E-15, abs_tol: absolute_tolerance)).to be_truthy
  end
  it 'returns true for same negative numbers' do
    expect(Math.close?(-1000001.0E-15, -1000002.0E-15, abs_tol: absolute_tolerance)).to be_truthy
    expect(Math.close?(-1000002.0E-15, -1000001.0E-15, abs_tol: absolute_tolerance)).to be_truthy
  end
  it 'returns false for different positive numbers' do
    expect(Math.close?(1000010.0E-15, 1000020.0E-15, abs_tol: absolute_tolerance)).to be_falsey
    expect(Math.close?(1000020.0E-15, 1000010.0E-15, abs_tol: absolute_tolerance)).to be_falsey
  end
  it 'returns false for different negative numbers' do
    expect(Math.close?(-1000010.0E-15, -1000020.0E-15, abs_tol: absolute_tolerance)).to be_falsey
    expect(Math.close?(-1000020.0E-15, -1000010.0E-15, abs_tol: absolute_tolerance)).to be_falsey
  end
end
```

10E-15 == 1E-14 therefore since the absolute tolerance is equal to the difference of the bottom two "return false" specs - they must be true - they are not true because subtracting those floats ends up with garbage.

For example

1000020.0E-15 - 1000010.0E-15

equals

1.0000000000085785e-14

Which places it outside of 1E14 but common sense (and looking at the numbers in front of us) obviously the correct answer is 1E14.

Floating numbers cannot be acted upon and then the result used to prove something.

This does not provide what it claims to provide - it is not possible to provide what you wish to provide here when dealing with floats.

Sorry

John

#5 - 04/30/2019 07:23 AM - nobu (Nobuyoshi Nakada)

- Description updated

I think it should be under Math or Float, and a independent gem could be a good first step.

#6 - 04/30/2019 08:16 AM - duerst (Martin Dürst)

Ruby is an object-oriented language. So I think this should be something like:

```
a.close_to?(b, abs_tolerance: t)
```

or so, not a function with two main numbers.

#7 - 04/30/2019 10:50 AM - sawa (Tsuyoshi Sawada)

Related to https://bugs.ruby-lang.org/issues/10425

#8 - 04/30/2019 12:33 PM - mame (Yusuke Endoh)

- Related to Feature #10425: A predicate method to tell if a number is near another added

#9 - 04/30/2019 01:29 PM - nobu (Nobuyoshi Nakada)

duerst (Martin Dürst) wrote:

> Ruby is an object-oriented language. So I think this should be something like:
>
> ```
> a.close_to?(b, abs_tolerance: t)
> ```
>
> or so, not a function with two main numbers.

If it is an instance method, the relative tolerance feels relative to the absolute value of the receiver, not the larger one.

#10 - 05/09/2019 08:09 AM - yennguyenh (yen nguyen)

nobu (Nobuyoshi Nakada) wrote:

> I think it should be under Math or Float, and a independent gem could be a good first step.

I have updated the first gist. It is under Math! I just forgot putting it in gist . Thank you for reminding anyway :D

#11 - 05/09/2019 10:22 AM - yennguyenh (yen nguyen)

wishdev (John Higgins) wrote:

> The tests for this are incorrect and show why this does not work.
>
> From the test gist
>
> ```
> context 'Numbers between 1 and 0' do
>   let (:absolute_tolerance) { 1.0E-14 }
>   it 'returns true for same positive numbers' do
>     expect(Math.close?(1000001.0E-15, 1000002.0E-15, abs_tol: absolute_tolerance)).to be_truthy
>     expect(Math.close?(1000002.0E-15, 1000001.0E-15, abs_tol: absolute_tolerance)).to be_truthy
>   end
>   it 'returns true for same negative numbers' do
> ```

```
    expect(Math.close?(-1000001.0E-15, -1000002.0E-15, abs_tol: absolute_tolerance)).to be_truthy
    expect(Math.close?(-1000002.0E-15, -1000001.0E-15, abs_tol: absolute_tolerance)).to be_truthy
  end
  it 'returns false for different positive numbers' do
    expect(Math.close?(1000010.0E-15, 1000020.0E-15, abs_tol: absolute_tolerance)).to be_falsey
    expect(Math.close?(1000020.0E-15, 1000010.0E-15, abs_tol: absolute_tolerance)).to be_falsey
  end
  it 'returns false for different negative numbers' do
    expect(Math.close?(-1000010.0E-15, -1000020.0E-15, abs_tol: absolute_tolerance)).to be_falsey
    expect(Math.close?(-1000020.0E-15, -1000010.0E-15, abs_tol: absolute_tolerance)).to be_falsey
  end
end
```

> 10E-15 == 1E-14 therefore since the absolute tolerance is equal to the difference of the bottom two "return false" specs - they must be true - they are not true because subtracting those floats ends up with garbage.
>
> For example
>
> 1000020.0E-15 - 1000010.0E-15
>
> equals
>
> 1.0000000000085785e-14
>
> Which places it outside of 1E14 but common sense (and looking at the numbers in front of us) obviously the correct answer is 1E14.
>
> Floating numbers cannot be acted upon and then the result used to prove something.
>
> This does not provide what it claims to provide - it is not possible to provide what you wish to provide here when dealing with floats.
>
> Sorry
>
> John

Sorry but I do not really understand what you meant. What I get so far is that you mean the difference of that pair of number (1000020.0E-15 - 1000010.0E-15) results not as expected, 1.0000000000085785e-14 instead of 1.0e-14. I have taken a look on that and realize one mistake on the algorithm. The absolute tolerance is set to check the accuracy to a certain decimal place and so at that place the difference should be less than 1 which is 0. Therefore the equal case should not be considered as the case for equal numbers. It should be fixed like below (I have also updated the code!)

current method:

```
abs_diff = (a - b).abs
    ((abs_diff <= (rel_tol * b).abs) ||
      (abs_diff <= (rel_tol * a).abs) ||
      (abs_diff <= abs_tol))
```

fixed method:

```
abs_diff = (a - b).abs
    ((abs_diff <= (rel_tol * b).abs) ||
      (abs_diff <= (rel_tol * a).abs) ||
      (abs_diff < abs_tol))
```

For ex:

absolute tolerance: 1e-2
a: 0.01
b: 0.02
(a-b).abs: 0.001 == 1e-2
At the second decimal place, there is the difference of '1' which should return false for the equal comparison, so it return false in case absolute tolerance == (a-b).abs

If it is not what you meant, please explain me more! Anyway thank you for the feedback, that I could find out that mistake!

#12 - 07/29/2019 06:54 AM - ko1 (Koichi Sasada)

could you share how about other languages (code examples)?

anyway, if you are interest about this ticket yet, could you file on our dev-meeting agenda?
https://bugs.ruby-lang.org/issues/15996

Thanks.

#13 - 07/29/2019 07:34 AM - wishdev (John Higgins)

yennguyenh (yen nguyen) wrote:

> wishdev (John Higgins) wrote:
> ....
> Sorry but I do not really understand what you meant. What I get so far is that you mean the difference of that pair of number (1000020.0E-15 - 1000010.0E-15) results not as expected, 1.0000000000085785e-14 instead of 1.0e-14. I have taken a look on that and realize one mistake on the algorithm. The absolute tolerance is set to check the accuracy to a certain decimal place and so at that place the difference should be less than 1 which is 0. Therefore the equal case should not be considered as the case for equal numbers. It should be fixed like below (I have also updated the code!)

First, my sincere apologies - I missed an email along the way and this just popped up on my radar this evening with the newest message.

However to be clear

Tolerance = 0.01

0.01 - 0.02 = -0.01 (False/True depending on which version of the code)
0.02 - 0.03 = -0.009999999999999998 (True always)
0.03 - 0.04 = -0.010000000000000002 (False always)

Those are not weird e-15 vs e-14 numbers - that's pennies on a dollar transaction.

Sorry

John

#14 - 07/29/2019 07:35 AM - mrkn (Kenta Murata)

Julia provides isapprox function in Base module. This returns true if norm(x-y) <= max(atol, rtol*max(norm(x), norm(y))).
The detail documentation is here.

The definition is here:

```
function isapprox(x::Number, y::Number; atol::Real=0, rtol::Real=rtoldefault(x,y,atol), nans::Bool=false)
    x == y || (isfinite(x) && isfinite(y) && abs(x-y) <= max(atol, rtol*max(abs(x), abs(y)))) || (nans && isna
n(x) && isnan(y))
end

const ≈ = isapprox
```

#15 - 07/29/2019 07:49 AM - wishdev (John Higgins)

mrkn (Kenta Murata) wrote:

> Julia provides isapprox function in Base module. This returns true if norm(x-y) <= max(atol, rtol*max(norm(x), norm(y))).
> The detail documentation is here.
>
> The definition is here:
>
> ```
> function isapprox(x::Number, y::Number; atol::Real=0, rtol::Real=rtoldefault(x,y,atol), nans::Bool=false)
>     x == y || (isfinite(x) && isfinite(y) && abs(x-y) <= max(atol, rtol*max(abs(x), abs(y)))) || (nans &&
> isnan(x) && isnan(y))
> end
>
> const ≈ = isapprox
> ```

From https://repl.it/languages/julia (online julia repl)

▯ isapprox(0.01-0.02, 0, atol=1e-2)
true

▯ isapprox(0.02-0.03, 0, atol=1e-2)
true

▯ isapprox(0.03-0.04, 0, atol=1e-2)
false

This does not work here either. Which mirrors the examples above (they take the earlier code from this ticket and treat exactly 0.01 as true for this example).

#16 - 07/29/2019 04:40 PM - wishdev (John Higgins)

So there is one path that might work here (it has limits but there are limits in general that cannot be worked around).

If one takes the following scenario

We have a set of items in a store where the base unit of price is 0.01 meaning that something may cost 1.23, or 2.45, but we would not see anything like 1.234 or 45.4555 because those are lower than the working "unit of change" (0.01).

So we could construct a method like this which would allow us to determine if two items where priced within a set of "units of change"

```
def within_amount(item_a, item_b, amount, unit_of_change)
(item_a - item_b).abs < ((within_amount).abs + (unit_of_change).abs /2)
end
```

So going back to earlier examples - if we take

0.01 vs 0.02 we would have a unit of change of no more than 0.01 because while we might have 0.015 as a valid option - we know that 0.03 is a valid option.

So taking the new method
item_a = 0.01
item_b = 0.02
amount = 0.01
unit_of_change = 0.01

(0.01 - 0.02).abs = 0.01
(0.01).abs + (0.01).abs /2 = 0.015

0.01 < 0.015 so it returns true

This works because we know that the unit of change implies the limits of the possible answers of (item_a - item_b).abs - so in this case we should have the following options

0.01, 0.02, 0.03, and so on

Since floating point subtraction is not exact we might end up with 0.0099999999994747, 0.01, or 0.010000000001 when comparing two adjacent numbers within our set

However, if we take the unit of change and halve it then we have better boundaries 0.010000000001 does not worked against a tight comparison to 0.01 but it works just fine if we compare it against 0.015 or 0.005 because those options lay well outside of the error bounds that floating point subtraction will offer us.

So the answer remains that simply comparing two numbers does not work - however, one can compare two numbers within a defined space and obtain the desired conceptual results.

The obvious rules are that within_amount must be at least equal to the unit of change. There are also issues if the within_amount is not a multiple of the unit of change. The validation check for that (checking if x is a multiple of y) starts us back down the road of inexact comparisons so it may be something that is a documented limit as opposed to a test within the method itself.

John