Ruby master - Feature #6857

bigdecimal/math BigMath.E/BigMath.exp R. P. Feynman inspired optimization

08/11/2012 11:46 PM - royaltm (Rafał Michalski)

Status:	Assigned	
Priority:	Normal	
Assignee:	mrkn (Kenta Murata)	
Target version:		

Description

The algorythms to calculate E and exp programmed in BigMath module are the very straightforward interpretation of the series 1 + x + $x^{2/2!}$ +
$x^{3/3!}$ + ...
Therefore they are slow.

Try it yourself:

```
require 'bigdecimal/math'

def timer; s=Time.now; yield; puts Time.now-s; end

timer { BigMath.E(1000) }   #->  0.038848
timer { BigMath.E(10000) }  #-> 16.526972
timer { BigMath.E(100000) } #-> lost patience
```

That's because every iteration divides 1 by n! and the dividend grows extremely fast.

In "Surely You're Joking, Mr. Feynman!" (great book, you should read it if you didn't already) R. P. Feynman said:

"One day at Princeton I was sitting in the lounge and overheard some mathematicians talking about the series for $e^{x,}$ which is 1 + x + $x^{2/2!}$ +
$x^{3/3!}$ Each term you get by multiplying the preceding term by x and dividing by the next number. For example, to get the next term after $x^{4/4!}$ you
multiply that term by x and divide by 5. It's very simple."

Yes it's very simple indeed. Why it's not been applied in such a great, modern and popular language? Is it because people just forget about simple solutions today?

Here is a Feynman's optimized version of BigMath.E:

```
def E(prec)
  raise ArgumentError, "Zero or negative precision for E" if prec <= 0
  n = prec + BigDecimal.double_fig
  y = d = i = one = BigDecimal('1')
  while d.nonzero? && (m = n - (y.exponent - d.exponent).abs) > 0
    m = BigDecimal.double_fig if m < BigDecimal.double_fig
    d = d.div(i, m)
    i += one
    y += d
  end
  y
end
```

Now, let's put it to the test:

```
(1..1000).all? {|n| BigMath.E(n).round(n) == E(n).round(n) }
=> true
BigMath.E(10000).round(10000) == E(10000).round(10000)
=> true
```

What about the speed then?

```
timer { E(1_000) }     #-> 0.003832 ~ 10 times faster
timer { E(10_000) }    #-> 0.139862 ~ 100 times faster
```

```
  timer { E(100_000) }   #-> 8.787411 ~ dunno?
  timer { E(1_000_000) } #-> ~11 minutes
```

The same simple rule might be applied to BigDecimal.exp() which originally uses the same straightforward interpretation of power series.
Feynman's pure ruby version of BigMath.exp (the ext version seems now pointless anyway):

```
  def exp(x, prec)
    raise ArgumentError, "Zero or negative precision for exp" if prec <= 0
    x = case x
    when Float
      BigDecimal(x, prec && prec <= Float::DIG ? prec : Float::DIG + 1)
    else
      BigDecimal(x, prec)
    end
    one = BigDecimal('1', prec)
    case x.sign
    when BigDecimal::SIGN_NaN
      return BigDecimal::NaN
    when BigDecimal::SIGN_POSITIVE_ZERO, BigDecimal::SIGN_NEGATIVE_ZERO
      return one
    when BigDecimal::SIGN_NEGATIVE_FINITE
      x = -x
      inv = true
    when BigDecimal::SIGN_POSITIVE_INFINITE
      return BigDecimal::INFINITY
    when BigDecimal::SIGN_NEGATIVE_INFINITE
      return BigDecimal.new('0')
    end
    n = prec + BigDecimal.double_fig
    if x.round(prec) == one
      y = E(prec)
    else
      y = d = i = one
      while d.nonzero? && (m = n - (y.exponent - d.exponent).abs) > 0
        m = BigDecimal.double_fig if m < BigDecimal.double_fig
        d = d.mult(x, m).div(i, m)
        i += one
        y += d
      end
    end
    y = one.div(y, n) if inv
    y.round(prec - y.exponent)
  end

  (1..1000).all? {|n| exp(E(n),n) == BigMath.exp(BigMath.E(n),n) }
  # => true
  (1..1000).all? {|n| exp(-E(n),n) == BigMath.exp(-BigMath.E(n),n) }
  # => true
  (-10000..10000).all? {|n| exp(BigDecimal(n)/1000,100) == BigMath.exp(BigDecimal(n)/1000,100) }
  # => true
  (1..1000).all? {|n| exp(BigMath.PI(n),n) == BigMath.exp(BigMath.PI(n),n) }
  # => true

  timer { BigMath.exp(BigDecimal('1').div(3, 10), 100) }    #-> 0.000496
  timer { exp(BigDecimal('1').div(3, 10), 100) }            #-> 0.000406 faster but not that reall
y

  timer { BigMath.exp(BigDecimal('1').div(3, 10), 1_000) }  #-> 0.029231
  timer { exp(BigDecimal('1').div(3, 10), 1_000) }          #-> 0.004554 here we go...

  timer { BigMath.exp(BigDecimal('1').div(3, 10), 10_000) } #-> 12.554197
  timer { exp(BigDecimal('1').div(3, 10), 10_000) }         #->  0.189462 oops :)

  timer { exp(BigDecimal('1').div(3, 10), 100_000) }        #-> 11.914613 who has the patience to
compare?
```

Arguments with large mantissa should slow down the results of course:

```
  timer { BigMath.exp(BigDecimal('1').div(3, 1_000), 1_000) }   #->  0.119048
  timer { exp(BigDecimal('1').div(3, 1_000), 1_000) }           #->  0.066177

  timer { BigMath.exp(BigDecimal('1').div(3, 10_000), 10_000) } #-> 68.083222
  timer { exp(BigDecimal('1').div(3, 10_000), 10_000) }         #-> 29.439336
```

Though still two times faster than the ext version.

It seems Dick Feynman was not such a joker after all. I think he was a master in treating lightly "serious" things and treating very seriously things that didn't matter to anybody else.

I'd write a patch for ext version if you are with me. Just let me know.

Associated revisions

Revision b8bbc1a3 - 11/23/2013 10:52 AM - mrkn (Kenta Murata)

- ext/bigdecimal/lib/bigdecimal/math.rb (BigMath.E): Use BigMath.exp. [Feature #6857] [ruby-core:47130]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@43817 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 43817 - 11/23/2013 10:52 AM - mrkn (Kenta Murata)

- ext/bigdecimal/lib/bigdecimal/math.rb (BigMath.E): Use BigMath.exp. [Feature #6857] [ruby-core:47130]

Revision 43817 - 11/23/2013 10:52 AM - mrkn (Kenta Murata)

- ext/bigdecimal/lib/bigdecimal/math.rb (BigMath.E): Use BigMath.exp. [Feature #6857] [ruby-core:47130]

Revision 43817 - 11/23/2013 10:52 AM - mrkn (Kenta Murata)

- ext/bigdecimal/lib/bigdecimal/math.rb (BigMath.E): Use BigMath.exp. [Feature #6857] [ruby-core:47130]

Revision 43817 - 11/23/2013 10:52 AM - mrkn (Kenta Murata)

- ext/bigdecimal/lib/bigdecimal/math.rb (BigMath.E): Use BigMath.exp. [Feature #6857] [ruby-core:47130]

Revision 43817 - 11/23/2013 10:52 AM - mrkn (Kenta Murata)

- ext/bigdecimal/lib/bigdecimal/math.rb (BigMath.E): Use BigMath.exp. [Feature #6857] [ruby-core:47130]

Revision 43817 - 11/23/2013 10:52 AM - mrkn (Kenta Murata)

- ext/bigdecimal/lib/bigdecimal/math.rb (BigMath.E): Use BigMath.exp. [Feature #6857] [ruby-core:47130]

History

#1 - 08/11/2012 11:48 PM - royaltm (Rafał Michalski)

- algorithms ;)

#2 - 08/13/2012 12:05 PM - sorah (Sorah Fukumori)

- Assignee set to mrkn (Kenta Murata)

#3 - 08/13/2012 12:05 PM - sorah (Sorah Fukumori)

- Tracker changed from Bug to Feature

#4 - 08/14/2012 02:58 AM - royaltm (Rafał Michalski)

Having fast exp() allows us to speed up BigMath.log(). Especially for calculations with large precision.

The area hyperbolic tangent power series performs better when the domain (x) of the function is closer to 1.

Additionally for x > 10 there is a significant linear performance degradation proportional to x.

So the first thing would be to narrow "no decimal shift" domain limitation to just 0.1 <= x <= 10.
The current implementation of BigMath.log uses range: 0.1 <= x < 100.

But this is just a prerequisite.

The real performance boost we gain from the following rule:

Let's suppose y ~ log(x) where y is calculated with much lesser precision than we actually need.
We may find then such an A:

A = x / exp(y)

which is very close to 1.

Now we can use it to calculate logarithm with the accurate precision from:

log(x) = y + log(a)

The implementation:

```
  def log(x, prec)
    raise ArgumentError, "Zero or negative precision for log" if prec <= 0
    raise ArgumentError, "Zero or negative argument for log" if x.round(prec) <= 0
    return BigDecimal('0') if x.round(prec) == BigDecimal('1')
    return BigDecimal::INFINITY if x.infinite?

    n = prec + BigDecimal.double_fig

    shift = x.exponent
    ten = BigDecimal('10')
    if shift < 0 || x > 10
      x = x.mult(BigDecimal("1E#{-shift}"), n)
    else
      shift = 0
    end

    if prec < 26 # 26 was chosen based on experiments
      y = BigMath.log(x, prec)
    else
      y  = log(x, Math.exp(Math.log(prec)/2).round)

      a = x.div(exp(y, n), n)
      y += BigMath.log(a, prec)
    end

    y += log(ten, prec).mult(shift, n) unless shift.zero?
    y
  end
```

Get ready for some benchmarks:

```
  require 'benchmark'
  require 'bigdecimal/util'

  def testlog(p, range=100.0, iter=100, count=1000)
    Benchmark.bm(20, 'ext', 'new') do |b|
      count.times.map { rand*range }.inject([0,0]) do |(tt1,tt2), n|
        nbig = n.to_d
        a1 = a2 = nil
        GC.disable
        t1 = b.report("#{n} ext") { iter.times { a1 = BigMath.log(nbig, p) } }
        t2 = b.report("#{n} new") { iter.times { a2 = log(nbig, p) } }
        GC.enable
        unless a1.round(p - a1.exponent) == a2.round(p - a2.exponent)
          raise "bad #{a1.round(p - a1.exponent)} <> #{a2.round(p - a2.exponent)}"
        end
        [t1/count + tt1, t2/count + tt2]
      end
    end
    nil
  end
```

To get the idea of speed up factor I'll present some summaries:

```
testlog(9, 10.0)
ext                0.026100   0.000000   0.026100 (  0.025777)
new                0.025600   0.000000   0.025600 (  0.025944)
```

we didn't optimize anything within the domain range of 0 < x < 10.0 and precision (< 26) so the new implementation performs similarly
(it's slightly slower due to some overhead of wrapper code)

```
testlog(9, 100.0)
ext                0.236000   0.000000   0.236000 (  0.235998)
new                0.055900   0.000000   0.055900 (  0.055529)
```

just narrowing the domain range calculated wihtout decimal shift to 0.1 <= x <= 10 gives as a significant speed increase.

Now let's try some serious BigDecimal precision:

```
testlog(99, 10.0)
ext                0.202900   0.000000   0.202900 (  0.201852)
new                0.075600   0.000000   0.075600 (  0.076487)
```

we can now see the effect of approximation algorithm

let's increase the domain range:

```
testlog(99, 100.0)
ext                2.387300   0.004000   2.391300 (  2.390849)
new                0.158300   0.001700   0.160000 (  0.160178)
```

the combined effect of both approximation and domain decimal shift range limitation gives us more than 10 times performance boost (average)

```
testlog(999, 10.0, 2)
ext                1.470000   0.000000   1.470000 (  1.469803)
new                0.031300   0.000000   0.031300 (  0.031546)
```

Large mantissa tests:

```
e = E(10000)
l1 = timer{ BigMath.log(e, 10000) } # -> 318.629882
l2 = timer{ log(e, 10000) }         # -> 1.524671
l1.round(10000) == l2.round(10000)
=> true
l1.round(10000) == 1
=> true

pi = BigMath.PI(10000)
l1 = timer{ BigMath.log(pi, 10000) } # -> 371.913958
l2 = timer{ log(pi,10000) }         # -> 1.892104

l1.round(10000) == l2.round(10000)
=> true
```

#5 - 11/20/2012 11:24 PM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Target version set to 2.6

#6 - 11/23/2013 07:52 PM - mrkn (Kenta Murata)

- Status changed from Assigned to Closed

- % Done changed from 0 to 100

This issue was solved with changeset r43817.
Rafał, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

- ext/bigdecimal/lib/bigdecimal/math.rb (BigMath.E): Use BigMath.exp. [Feature #6857] [ruby-core:47130]

#7 - 11/23/2013 07:55 PM - mrkn (Kenta Murata)

- Status changed from Closed to Assigned

- % Done changed from 100 to 50

The optimization of BigMath.log is remaining.

#8 - 12/25/2017 06:15 PM - naruse (Yui NARUSE)

- Target version deleted (2.6)