1 Introduction

This module converts floating point numbers to fractions efficiently.

2 Floating point tolerance

First we'll define some classes in a separte source file to handle floating-point telerance. We will support tolerances for the floating point types `Float` and `BigDecimal`.

```
"lib/nio/flttol.rb" 1a ≡
    # Floating point tolerance
    #--
    ⟨License 1b⟩
    #++
    ⟨rdoc commentary for flttol.rb 25c⟩
    ⟨flttol Required Modules 9a⟩
    ⟨flttol definitions 2a, ... ⟩
    ⟨Nio constructor methods rdoc 26g⟩
    module Nio
      ⟨flttol classes 2c, ... ⟩
      module_function
      ⟨flttol functions 13a, ... ⟩
    end
    ◇
```

```
⟨License 1b⟩ ≡
    # Copyright (C) 2003–2005, Javier Goizueta <javier@goizueta.info>
    #
    # This program is free software; you can redistribute it and/or
    # modify it under the terms of the GNU General Public License
    # as published by the Free Software Foundation; either version 2
    # of the License, or (at your option) any later version.◇
```

Macro referenced in 1a, 13b, 14a.

First we'll set up some utilities to deal with floating point types. We need some information about the floating point implementation and we will add define some constants for it. We'll use the contants available in Float since version 1.8 of ruby:

- `MANT_DIG` is the number of bits in the fraction part
- `DIG` is the number of decimal digits of precision: if a decimal number has more digits they may not preserve when converted to Float (with correct rounding) and back to decimal (rounded to `DIG` digits).
- `EPSILON` is the smallest number that, added to 1.0 produces something different from 1.0 (i.e. the difference between 1 and the least value greater than 1 that is representable). Note that `EPSILON` is also the maximum relative error corresponding to half an ulp (unit in the last place), and half an ulp is tha maximum rounding error when a real number is approximated by the closest floating point number.

To these we add a constant, `DECIMAL_DIG` defined as the number of decimal digits necessary for round-trip conversion `Float`→decimal→`Float`.

⟨*flttol definitions* 2a⟩ ≡
```
class Float
  unless const_defined?(:RADIX) # old Ruby versions didn't have this
    # Base of the Float representation
    RADIX = 2
    ⟨compute bits per Float 2b⟩
    # Number of RADIX−base digits of precision in a Float
    MANT_DIG = _bits_
    # Number of decimal digits that can be stored in a Float and recovered
    DIG = ((MANT_DIG−1)*Math.log(RADIX)/Math.log(10)).floor
    # Smallest value that added to 1.0 produces something different from 1.0
    EPSILON = Math.ldexp(*Math.frexp(1).collect{|e| e.kind_of?(Integer) ? e−(MANT_DIG−1) : e})
  end
  # Decimal precision required to represent a Float and be able to recover its value
  DECIMAL_DIG = (MANT_DIG*Math.log(RADIX)/Math.log(10)).ceil+1
end
◇
```
Macro defined by 2a, 27d.
Macro referenced in 1a.

If we need to compute the constants we assume the base is 2. Then the number of bits of precision can be easily compute like this:

⟨*compute bits per Float* 2b⟩ ≡
```
x = 1.0
_bits_ = 0
begin
  _bits_ += 1
  x /= 2
end while 1!=x+1
◇
```
Macro referenced in 2a.

Notes:

- `DIG` is defined to be valid for any decimal representation and and `DECIMAL_DIG` for any `Float` value. Particular decimal expresions with many more than `DIG` digits can be stored exactly in a Float, and particular `Float` values can be unambiguosly defined with less than `DECIMAL_DIG` decimal digits.
- `MANT_DIG = 2-Math.frexp(Float::EPSILON)[1]` `EPSILON` is 1ulp (unit in the least place) for 1.0. Knuth (4.2.2 pg.219) states that a tolerace greater than or equal to `2*EPSILON/(1-0.5*EPSILON)**2` assures the associativity of multiplication, e.g. `ldexp(0.75,3-MANT_DIG)`, since we have `2*EPSILON/(1-0.5*EPS`

This class represents floating point tolerances and allows comparison of numbers within the specified tolerance.

⟨*flttol classes* 2c⟩ ≡
```
⟨rdoc commentary for Tolerance 25d⟩
class Tolerance
  include StateEquivalent

  # The numeric class this tolerance applies to.
  def num_class
    Float
  end

  # The tolerance mode is either :abs (absolute) :rel (relative) or :sig (significant).
  # The last parameter is a flag to specify decimal mode for the :sig mode
  def initialize (t=0.0, mode=:abs, decmode=false)
    set t, mode, decmode
  end
```

```
    ⟨Tolerance constructors 4a, ... ⟩

    #Shortcut notation for get_value
    def [](x)
      return x.nil? ? @t : get_value(x)
    end
    #Return tolerance relative to a magnitude
    def get_value(x)
      rel(x)
    end
    #Essential equality within tolerance
    def equals?(x,y)
      ⟨equals? 7g⟩
    end
    #Approximate equality within tolerance
    def aprx_equals?(x,y)
      ⟨aprxEquals? 7l⟩
    end
    #Comparison within tolerance
    def greater_than?(x,y)
      less_than?(y,x)
    end
    #Comparison within tolerance
    def less_than?(x,y)
     ⟨lessThan? 7b⟩
    end
    #Comparison within tolerance
    def zero?(x,compared_with=nil)
      compared_with.nil? ? x.abs<@t : x.abs<rel(compared_with)
    end

    ⟨Tolerance methods 8a⟩
    ⟨Tolerance attributes 8b⟩

    private
    ⟨Tolerance private 3, ... ⟩
  end
  ◇
```

Macro defined by 2c, 5b, 9b, 11a.
Macro referenced in 1a.

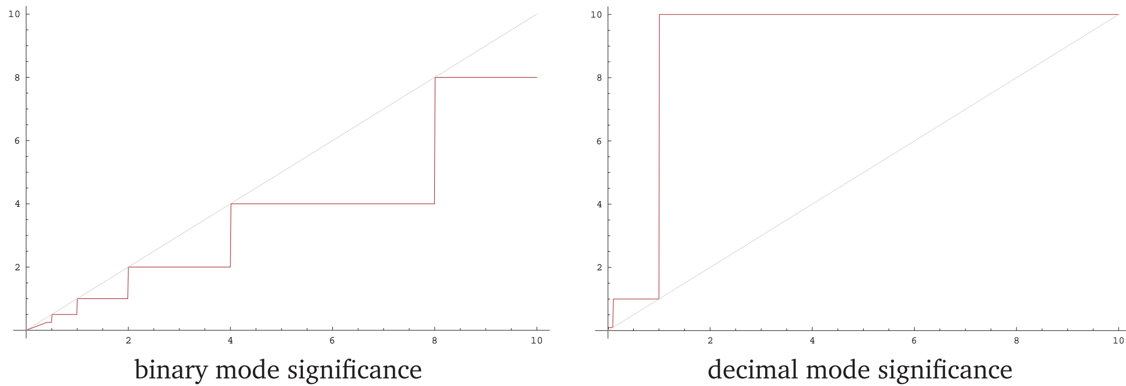
The tolerances can be defined in three basic modes:

- Absolute tolerance (`:abs`) is a fixed value.
- Relative tolerance (`:rel`) is given in relation to the unit 1 and varies proportionally to the magnitud of the tested values.
- Significant tolerance (`:sig`) is given in relation to a reference interval and varies in steps (of exponential size). For the binary floating-point type Float, binary significant is used unless decimal mode is selected; the reference interval is $[1,2)$. Otherwise decimal significance in relation to reference $[0.1,1)$ is used.

This private method defines a tolerance.

```
⟨Tolerance private 3⟩ ≡
  def set(t=0.0, mode=:abs, decmode=false)
    ⟨Initialize Tolerance 5c⟩
    self
  end
  ◇
```

Macro defined by 3, 6d, 7a, 8c.
Macro referenced in 2c.



binary mode significance   decimal mode significance

This defines the tolerance by givind the number of decimal digits of precision; by default as an absolute tolerance, i.e. by a fixed number of decimals; significant mode would also be meaningful here, to define the number of significant digits. By default *r*ounded digits are used.

⟨ *Tolerance constructors* 4a ⟩ ≡

```
#This initializes a Tolerance with a given number of decimals
def decimals(d, mode=:abs, rounded=true)
  ⟨Initialize Tolerance from digits 6a⟩
  self
end
◇
```

Macro defined by 4abcd, 5a.
Macro referenced in 2c.

This is a shortcut to define the tolerance by the number of significant digits.

⟨ *Tolerance constructors* 4b ⟩ ≡

```
#This initializes a Tolerance with a number of significant decimal digits
def sig_decimals(d, rounded=true)
  decimals d, :sig, rounded
end
◇
```

Macro defined by 4abcd, 5a.
Macro referenced in 2c.

Initialize with a multiple of the internal floating-point precision.

⟨ *Tolerance constructors* 4c ⟩ ≡

```
# Initialize with a multiple of the internal floating-point precision.
def epsilon(times_epsilon=1, mode=:sig)
  set Float::EPSILON*times_epsilon, mode
end
◇
```

Macro defined by 4abcd, 5a.
Macro referenced in 2c.

Same, with a somewhat (about twice) bigger precision that assures associative multiplication.

⟨ *Tolerance constructors* 4d ⟩ ≡

```
# As #epsilon but using a somewhat bigger (about twice) precision that
# assures associative multiplication.
def big_epsilon(n=1, mode=:sig)
  t = Math.ldexp(0.5*n,3-Float::MANT_DIG) # n*(2*Float::EPSILON/(1-0.5*Float::EPSILON)**2)
  set t, mode
end
◇
```

Macro defined by 4abcd, 5a.
Macro referenced in 2c.

Initialize with a relative fraction, a percentage, or a per-mille value.

⟨*Tolerance constructors* 5a⟩ ≡
```
# Initialize with a relative fraction
def fraction(f)
  set f, :rel
end
# Initialize with a percentage
def percent(x)
  fraction x/100.0
end
# Initialize with a per-mille value
def permille(x)
  fraction x/1000.0
end
```
◇

Macro defined by 4abcd, 5a.
Macro referenced in 2c.

Now we define constructors using the initialization methods defined.

⟨*flttol classes* 5b⟩ ≡
```
def Tolerance.decimals(d=0, mode=:abs,rounded=true)
  Tolerance.new.decimals(d,mode,rounded)
end
def Tolerance.sig_decimals(d=0, mode=:abs,rounded=true)
  Tolerance.new.sig_decimals(d,rounded)
end
def Tolerance.epsilon(n=1, mode=:sig)
  Tolerance.new.epsilon(n, mode)
end
def Tolerance.big_epsilon(n=1, mode=:sig)
  Tolerance.new.big_epsilon(n, mode)
end
def Tolerance.fraction(f)
  Tolerance.new.fraction(f)
end
def Tolerance.percent(p)
  Tolerance.new.percent(p)
end
def Tolerance.permille(p)
  Tolerance.new.permille(p)
end
```
◇

Macro defined by 2c, 5b, 9b, 11a.
Macro referenced in 1a.

This is the code to set the tolerance value: values that are to big or too small are adjusted, and the number of decimal digits implied by the tolerance is computed.

⟨*Initialize Tolerance* 5c⟩ ≡
```
@t = t==0 ? Float::EPSILON : t.abs
@t = 0.5 if @t > 0.5
@mode = mode
@t = Float::EPSILON if @mode!=:abs && @t<Float::EPSILON
@decimal_mode = decmode
@d = @t==0 ? 0 : (-Math.log10(2*@t).floor).to_i
```
◇

Macro referenced in 3.

Note that there is an inconsistency in the relative mode between decimals and non-decimals tolerences: for decimals, the tolerance value refers to the `[0.1,1)` interval; for non-decimals, the reference interval is `[1,2)`.

For decimals there's an option to choose between rounded or truncated decimals; in both cases the rounded or truncated d digit may vary in one unit at most.

⟨ *Initialize Tolerance from digits* 6a ⟩ ≡

```
@mode = mode
@decimal_mode = true
@d = (d<=0 || d>Float::DIG) ? Float::DIG : d
@t = 10**(-@d)
@t *= 0.5 if rounded
◇
```

Macro referenced in 4a.

The mode determines how to compare quantities within the tolerance. First we define relative comparison; this fragment is parameterized by a partial expression that defines what to compare against the tolerance and which comparision operator to use (e.g. `y-x >`) and by a method to apply and choose which of the compared magnitude to use (e.g. `min`).

⟨ *Relative Comparison* 6b ⟩ ≡

```
·1 @t*([x.abs,y.abs].·2) #reference value is 1
◇
```

Macro referenced in 7bgl.

We define a fragment with same parameters for the significant comparison.

⟨ *Significant Comparison* 6c ⟩ ≡

```
if @decimal_mode
  begin
    x_exp = Math.log10(x.abs)
    #x_exp = x_exp.finite? ? x_exp.ceil : 0
    x_exp = x_exp.finite? ? x_exp.floor+1 : 0
  rescue
    x_exp = 0
  end
  begin
    y_exp = Math.log10(y.abs)
    #y_exp = y_exp.finite? ? y_exp.ceil : 0
    y_exp = y_exp.finite? ? y_exp.floor+1 : 0
  rescue
    y_exp = 0
  end
  ·1 @t*(10**([x_exp,y_exp].·2-@@dec_ref_exp))
else
  z,x_exp = Math.frexp(x)
  z,y_exp = Math.frexp(y)
  ·1 Math.ldexp(@t,[x_exp,y_exp].·2-@@ref_exp) # ·1 @t*(2**([x_exp,y_exp].·2-@@ref_exp))
end
◇
```

Macro referenced in 7bgl.

Now we will define reference exponents for significant tolerances; in general a reference exponent r will select the interval $[b^{r-1}, b^r)$ as reference (the interval where the tolerance has the value given in its definition).

The reference exponent is the binary exponent of the reference value for relative tolerances. If we use 1 (which is `Math.frexp(1)[1]`), then the tolerance given applies to $[1, 2)$. If we use 0 the reference interval is $[0.5, 1)$.

⟨ *Tolerance private* 6d ⟩ ≡

```
@@ref_exp = 1 # Math.frexp(1)[1] => tol. relative to [1,2)
◇
```

Macro defined by 3, 6d, 7a, 8c.
Macro referenced in 2c.

For significant decimals mode, we will use the $[0.1, 1)$ as reference by using the reference exponent 0; a reference exponent of 1 the reference interval would be $[1, 10)$.

⟨*Tolerance private* 7a⟩ ≡
```
@@dec_ref_exp = 0 # tol. relative to [0.1,1)
```
◇

Macro defined by 3, 6d, 7a, 8c.
Macro referenced in 2c.

For relative mode, the reference is always 1 (a single value rather than an interval); for absolute mode the reference is $(-\infty, +\infty)$, since the same value is always used.

Now we can define the different specific comparisons.

⟨*lessThan?* 7b⟩ ≡
```
case @mode
  when :sig
    ⟨Significant Comparison ( y−x >, max ) 6c⟩
  when :rel
    ⟨Relative Comparison ( y−x >, max ) 6b⟩
  when :abs
    x−y<@t
end
```
◇

Macro referenced in 2c.

This is essential equality.

⟨*equals?* 7g⟩ ≡
```
case @mode
  when :sig
    ⟨Significant Comparison ( (y−x).abs <=, min ) 6c⟩
  when :rel
    ⟨Relative Comparison ( (y−x).abs <=, min ) 6b⟩
  when :abs
    (x−y).abs<@t
end
```
◇

Macro referenced in 2c.

And this is approximate equality, a weaker form of equality.

⟨*aprxEquals?* 7l⟩ ≡
```
case @mode
  when :sig
    ⟨Significant Comparison ( (y−x).abs <=, max ) 6c⟩
  when :rel
    ⟨Relative Comparison ( (y−x).abs <=, max ) 6b⟩
  when :abs
    (x−y).abs<=@t
end
```
◇

Macro referenced in 2c.

⟨ *Tolerance methods* 8a ⟩ ≡

```
# Returns true if the argument is approximately an integer
def apprx_i?(x)
  equals?(x,x.round)
end
# If the argument is close to an integer it rounds it
# and returns it as an object of the specified class (by default, Integer)
def apprx_i(x, result =Integer)
  r = x.round
  return equals?(x,r) ? r.prec(result) : x
end
◇
```

Macro referenced in 2c, 9b.

Now we will define accessors for the public properties of Tolerance. To modify a property the tolerance must be redefined with any of the initialization methods.

⟨ *Tolerance attributes* 8b ⟩ ≡

```
# Returns the magnitude of the tolerance
def magnitude
  @t
end
# Returns the number of decimal digits of the tolerance
def num_decimals
  @d
end
# Returns true for decimal−mode tolerance
def decimal?
  @decimal_mode
end
# Returns the mode (:abs, :rel, :sig) of the tolerance
def mode
  @mode
end
◇
```

Macro referenced in 2c, 9b.

And we must now define a method to compute the relative value of the tolerance in relation to a magnitude `x`. For absolute mode this returns the tolerance value independently of `x`; otherwise the value is properly scaled to `x`.

⟨ *Tolerance private* 8c ⟩ ≡

```
def rel (x)
  r = @t
  case @mode
    when :sig
      if @decimal_mode
        d = x==0 ? 0 : (Math.log10(x.abs).floor+1).to_i
        r = @t*(10**(d−@@dec_ref_exp))
      else
        x,exp = Math.frexp(x)
        r = Math.ldexp(@t,exp−@@ref_exp)
      end
    when :rel
      r = @t*x.abs
  end
  r
end
◇
```

Macro defined by 3, 6d, 7a, 8c.
Macro referenced in 2c.

2.1 BigDecimal tolerance

Here define a tolerance class for `BigDecimal`. This is not, in general, as useful as `Tolerance` for `Float` is, since `BigDecimal` has arbitrary precision.

⟨*flttol Required Modules* 9a⟩ ≡

```
require 'bigdecimal'
require 'bigdecimal/math' if ::VERSION>='1.8.1'
require 'nio/tools'
◇
```

Macro referenced in 1a.

⟨*flttol classes* 9b⟩ ≡

```
⟨rdoc commentary for BigTolerance 25e⟩
class BigTolerance
  include StateEquivalent
  module BgMth # :nodoc:
    extend BigMath if ::RUBY_VERSION>='1.8.1'
  end

  # The numeric class this tolerance applies to.
  def num_class
    BigDecimal
  end

  #The tolerance mode is either :abs (absolute) :rel (relative) or :sig
  def initialize (t=BigDecimal('0'), mode=:abs, decmode=false)
    set t, mode, decmode
  end

  ⟨BigTolerance constructors 10c, ... ⟩

  #Shortcut notation for get_value
  def [](x)
    return x.nil? ? @t : get_value(x)
  end
  #Return tolerance relative to a magnitude
  def get_value(x)
    rel(x)
  end
  #Essential equality within tolerance
  def equals?(x,y)
    ⟨BigTolerance equals? 12g⟩
  end
  #Approximate equality within tolerance
  def aprx_equals?(x,y)
    ⟨BigTolerance aprxEquals? 12l⟩
  end
  #Comparison within tolerance
  def greater_than?(x,y)
    less_than?(y,x)
  end
  #Comparison within tolerance
  def less_than?(x,y)
   ⟨BigTolerance lessThan? 12b⟩
  end
  #Comparison within tolerance
  def zero?(x,compared_with=nil)
    compared_with.nil? ? x.abs<@t : x.abs<rel(compared_with)
  end

  ⟨Tolerance methods 8a⟩
```

```
    ⟨ Tolerance attributes 8b ⟩

    private
    ⟨ BigTolerance private 10a, ... ⟩
  end
  ◇
```
Macro defined by 2c, 5b, 9b, 11a.
Macro referenced in 1a.

⟨ *BigTolerance private* 10a ⟩ ≡
```
  HALF = BigDecimal('0.5')
  ◇
```
Macro defined by 10ab, 12q.
Macro referenced in 9b.

The initialization methods and constructors are as those of `Tolerance`.

⟨ *BigTolerance private* 10b ⟩ ≡
```
  def set(t=BigDecimal('0'), mode=:abs, decmode=false)
    ⟨ Initialize BigTolerance 11b ⟩
    self
  end
  ◇
```
Macro defined by 10ab, 12q.
Macro referenced in 9b.

Initialize with a given number of decimals.

⟨ *BigTolerance constructors* 10c ⟩ ≡
```
  #This initializes a BigTolerance with a given number of decimals
  def decimals(d, mode=:abs, rounded=true)
    ⟨ Initialize BigTolerance from digits 11c ⟩
    self
  end
  ◇
```
Macro defined by 10cde.
Macro referenced in 9b.

Initialize with a number a number of significant decimal digits.

⟨ *BigTolerance constructors* 10d ⟩ ≡
```
  #This initializes a BigTolerance with a number of significative decimal digits
  def sig_decimals(d, rounded=true)
    decimals d, :sig, rounded
  end
  ◇
```
Macro defined by 10cde.
Macro referenced in 9b.

Initialize with a relative fraction, a percentage, or a per-mille value.

⟨ *BigTolerance constructors* 10e ⟩ ≡
```
  def fraction(f)
    set f, :rel
  end
  def percent(x)
    fraction x*BigDecimal('0.01')
  end
  def permille(x)
    fraction x*BigDecimal('0.001')
  end
  ◇
```
Macro defined by 10cde.
Macro referenced in 9b.

Shortcuts for constructors.

⟨ *flttol classes* 11a ⟩ ≡
```
def BigTolerance.decimals(d=0, mode=:abs)
  BigTolerance.new.decimals(d,mode)
end
def BigTolerance.sig_decimals(d=0, mode=:abs)
  BigTolerance.new.sig_decimals(d)
end
def BigTolerance.fraction(f)
  BigTolerance.new.fraction(f)
end
def BigTolerance.percent(p)
  BigTolerance.new.percent(p)
end
def BigTolerance.permille(p)
  BigTolerance.new.permille(p)
end
◇
```
Macro defined by 2c, 5b, 9b, 11a.
Macro referenced in 1a.

Since the underlying type `BigTolerance` is now decimal we won't use a "binary significance" mode, but we will use a different reference intervals for decimal mode. Here is the normal tolerance mode, with the reference $[1, 10)$ for significative mode.

⟨ *Initialize BigTolerance* 11b ⟩ ≡
```
@t = t
@t = HALF if @t > HALF
raise TypeError,"El valor de tolerancia debe ser de tipo BigDecimal" if @t.class!=BigDecimal
@mode = mode
@decimal_mode = decmode
@d = @t.zero? ? 0 : −(@t∗2).exponent+1
@ref_exp = BigDecimal('1').exponent # reference for significative mode: [1,10)
◇
```
Macro referenced in 10b.

And here is the decimal tolerance mode, with the reference $[0.1, 1)$.

⟨ *Initialize BigTolerance from digits* 11c ⟩ ≡
```
@mode = mode
@decimal_mode = true
@d = d==0 ? 16 : d
if rounded
  @t = BigDecimal("0.5E#{-d}") # HALF∗(BigDecimal(10)∗∗(−d))
else
  @t = BigDecimal("1E#{-d}") # BigDecimal(10)∗∗(−d)
end
@ref_exp = BigDecimal('0.1').exponent # reference for significative mode: [0.1,1)
◇
```
Macro referenced in 10c.

Now we define the parameterized comparison fragments as for `Tolerance`.

⟨ *BigTolerance Significative Comparison* 11d ⟩ ≡
```
x_exp = x.exponent
y_exp = y.exponent
·1 @t∗BigDecimal("1E#{[x_exp,y_exp].·2−@ref_exp}")
◇
```
Macro referenced in 12bgl.

⟨ *BigTolerance Relative Comparison* 12a ⟩ ≡
```
·1 @t*([x.abs,y.abs].·2) #reference value is 1
◇
```
Macro referenced in 12bgl.

And the specific comparisons.

⟨ *BigTolerance lessThan?* 12b ⟩ ≡
```
case @mode
  when :sig
    ⟨ BigTolerance Significative Comparison ( y−x >, max ) 11d ⟩
  when :rel
    ⟨ BigTolerance Relative Comparison ( y−x >, max ) 12a ⟩
  when :abs
    x−y<@t
end
◇
```
Macro referenced in 9b.

⟨ *BigTolerance equals?* 12g ⟩ ≡
```
case @mode
  when :sig
    ⟨ BigTolerance Significative Comparison ( (y−x).abs <=, min ) 11d ⟩
  when :rel
    ⟨ BigTolerance Relative Comparison ( (y−x).abs <=, min ) 12a ⟩
  when :abs
    (x−y).abs<@t
end
◇
```
Macro referenced in 9b.

⟨ *BigTolerance aprxEquals?* 12l ⟩ ≡
```
case @mode
  when :sig
    ⟨ BigTolerance Significative Comparison ( (y−x).abs <=, max ) 11d ⟩
  when :rel
    ⟨ BigTolerance Relative Comparison ( (y−x).abs <=, max ) 12a ⟩
  when :abs
    (x−y).abs<=@t
end
◇
```
Macro referenced in 9b.

And the tolerance relative value.

⟨ *BigTolerance private* 12q ⟩ ≡
```
def rel(x)
  r = @t
  case @mode
    when :sig
      d = x==0 ? 0 : x.exponent
      r = @t*BigDecimal("1E#{d-@ref_exp}")
    when :rel
      r = @t*x.abs
  end
  r
end
◇
```
Macro defined by 10ab, 12q.
Macro referenced in 9b.

2.2 Tolerance definition and conversion methods

```
⟨flttol functions 13a⟩ ≡
    ⟨rdoc for Tol 27b⟩
    def Tol(x) # :doc:
      case x
        when Tolerance
          x
        when BigTolerance
          x
        when BigDecimal
          BigTolerance.new(x)
        when Float
          Tolerance.new(x)
        when Integer
          Tolerance.sig_decimals(x)
        else # e.g. Rational
          x
      end
    end

    ⟨rdoc for BigTol 27c⟩
    def BigTol(x) # :doc:
      case x
        when BigTolerance
          x
        when Integer
          BigTolerance.sig_decimals(x)
        when Rational
          x
        else
          BigTolerance.new(BigDec(x))
      end
    end
    ◇
```

Macro defined by 13a, 17a.
Macro referenced in 1a.

3 Rationalization of floating point numbers

To find rational aproximations we use algorithms by Joe Horn adaptad from his RPL programs.

```
"lib/nio/rtnlzr.rb" 13b ≡
    # Rationalization of floating point numbers.
    #--
    ⟨License 1b⟩
    #++
    ⟨rdoc commentary for rntlzr.rb 25a⟩
    ⟨Required Modules 14b, ... ⟩
    ⟨definitions 15a, ... ⟩
    ⟨classes 24b, ... ⟩
    module Nio
      ⟨Nio definitions ?⟩
      ⟨Nio classes 18, ... ⟩
      module_function
      ⟨Nio functions ?⟩
    end
    ◇
```

"test/test_rtnlzr.rb" 14a ≡

```
⟨License 1b⟩
require 'test/unit'

require 'nio/rtnlzr'
require 'nio/sugar'
include Nio
require 'yaml'
require 'bigdecimal/math'

class TestRtnlzr < Test::Unit::TestCase

  class BgMth
    extend BigMath
  end

  def setup
    ⟨Tests setup 28a⟩
  end

  ⟨Tests 28b, ... ⟩

end
◇
```

⟨*Required Modules* 14b⟩ ≡

```
require 'nio/tools'
◇
```

Macro defined by 14bc, 24ac.
Macro referenced in 13b.

⟨*Required Modules* 14c⟩ ≡

```
require 'nio/flttol'
◇
```

Macro defined by 14bc, 24ac.
Macro referenced in 13b.

4 Floating point to exact fraction conversion

These utility functions return fractions that yield the exact value of floating point numbers; this is trivial (since floating point numbers have finite precision) and doesn't produce simple fractions.

4.1 Float

This implementation does not always yield the smallest possible fraction, but is efficient. It is based in the definition of `Math.frexp` and `ldexp`, and also relies on the fact that `Float#to_i` converts big integers; in particular big powers of the Float radix to their exact integer value (so we use `Math.frexp` rather than `Integer#**` when possible.)

⟨ *definitions* 15a ⟩ ≡

```
class Float
  ⟨ rdoc commentary for Float#nio_xr 26c ⟩
  def nio_xr
    return Rational(self.to_i,1) if self.modulo(1)==0
    if !self.finite?
      return Rational(0,0) if self.nan?
      return self<0 ? Rational(-1,0) : Rational(1,0)
    end

    f,e = Math.frexp(self)

    if e < Float::MIN_EXP
       bits = e+Float::MANT_DIG-Float::MIN_EXP
    else
       bits = [Float::MANT_DIG,e].max
       #return Rational(self.to_i,1) if bits<e
    end
      p = Math.ldexp(f,bits)
      e = bits - e
      if e<Float::MAX_EXP
        q = Math.ldexp(1,e)
      else
        q = Float::RADIX**e
      end
    return Rational(p.to_i,q.to_i)
  end
end
◇
```

Macro defined by 15a, 16b, 17bc.
Macro referenced in 13b.

Here's alternative implementation for binary floating point that yields smallest fractions when possible and is almost as fast:

⟨ *scratch* 15b ⟩ ≡

```
class Float
  def nio_xr
    p,q = self,1
    while p.modulo(1) != 0
      p *= 2.0
      q <<= 1 # q *= 2
    end
    return Rational(p.to_i,q)
  end
end
◇
```

Macro defined by 15b, 16a.
Macro never referenced.

An a here's a shorter implementation relying on the semantics of the power operator, but which is somewhat slow:

⟨ *scratch* 16a ⟩ ≡

```
class Float
  def nio_xr
    f,e = Math.frexp(self)
    f = Math.ldexp(f, Float::MANT_DIG)
    e -= Float::MANT_DIG
    return Rational( f.to_i*(Float::RADIX**e.to_i), 1)
  end
end
◇
```

Macro defined by 15b, 16a.
Macro never referenced.

4.2 BigDecimal

⟨ *definitions* 16b ⟩ ≡

```
class BigDecimal
  ⟨ rdoc commentary for BigDecimal#nio_xr 26b ⟩
  def nio_xr
    s,f,b,e = split
    p = f.to_i
    p = -p if s<0
    e = f.size-e
    if e<0
      p *= b**(-e)
      e = 0
    end
    q = b**(e)
    return Rational(p,q)
  end
end
◇
```

Macro defined by 15a, 16b, 17bc.
Macro referenced in 13b.

We will define here also a utility to define BigDecimals; when applied to a Float value this uses the method `Nio.nio_float_to_bigdecimal`, defined in rtnlzr.rb; that file is not required here to avoid circular references, but should have been brought in before using BigDec applied to a Float argument.

⟨ *flttol functions* 17a ⟩ ≡

```
  ⟨ rdoc for BigDec 27a ⟩
  def BigDec(x,prec=nil) # :doc:
    if x.respond_to?(:to_str)
      x = BigDecimal(x.to_str, prec||0)
    else
      case x
        when Integer
        x = BigDecimal(x.to_s)
      when Rational
        if prec && prec!=:exact
          x = BigDecimal.new(x.numerator.to_s).div(x.denominator,prec)
        else
          x = BigDecimal.new(x.numerator.to_s)/BigDecimal.new(x.denominator.to_s)
        end
      when BigDecimal
      when Float
        x = nio_float_to_bigdecimal(x,prec)
      end
    end
    x
  end
◇
```

Macro defined by 13a, 17a.
Macro referenced in 1a.

4.3 Integer

⟨ *definitions* 17b ⟩ ≡

```
class Integer
  ⟨ rdoc commentary for Integer#nio_r ? ⟩
  def nio_xr
    return Rational(self,1)
  end
end
◇
```

Macro defined by 15a, 16b, 17bc.
Macro referenced in 13b.

4.4 Rational

⟨ *definitions* 17c ⟩ ≡

```
class Rational
  ⟨ rdoc commentary for Rational#nio_r ? ⟩
  def nio_xr
    return self
  end

  # helper method to return both the numerator and denominator
  def nio_num_den
    return [numerator,denominator]
  end
end
◇
```

Macro defined by 15a, 16b, 17bc.
Macro referenced in 13b.

5 Rationalizer object

Here is the `Rtnlzr` class that encapsulates the rationalization algorithm. It contains several rationalization approaches that has been tested; the most efficient one is them `rationalize` method.

⟨*Nio classes* 18⟩ ≡

```
    ⟨rdoc commentary for Rtnlzr 25b⟩
    class Rtnlzr
      include StateEquivalent

      # Create Rationalizator with given tolerance.
      def initialize(tol=Tolerance.new)
        @tol = tol
      end

      # Rationalization method that finds the fraction with
      # smallest denominator fraction within the tolerance distance
      # of an approximate (floating point) number.
      #
      # It uses the algorithm which has been found most efficient, rationalize_Knuth.
      def rationalize(x)
        rationalize_Knuth(x)
      end

      # This algorithm is derived from exercise 39 of 4.5.3 in
      # "The Art of Computer Programming", by Donald E. Knuth.
      def rationalize_Knuth(x)
        ⟨Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta 19h⟩
      end
      # This is algorithm PDQ2 by Joe Horn.
      def rationalize_Horn(x)
        ⟨Smallest-Denominator Rationalization by Joe Horn 19d⟩
      end
      # This is from a RPL program by Tony Hutchins (PDR6).
      def rationalize_HornHutchins(x)
        ⟨Smallest-Denominator Rationalization by Joe Horn and Tony Hutchins 19f⟩
      end
    end
    ◇
```

Macro defined by 18, 22b, 23.
Macro referenced in 13b.

This is the generic structure of our rationalization methods:

⟨ *Rationalization Procedure* 19a ⟩ ≡
```
num_tol = @tol.kind_of?(Numeric)
if !num_tol && @tol.zero?(x)
  # num,den = x.nio_xr.nio_num_den
  num,den = 0,1
else
  negans=false
  if x<0
    negans = true
    x = -x
  end
  dx = num_tol ? @tol : @tol.get_value(x)

  ·1

  num = -num if negans
end
return num,den
```
◇
Macro referenced in 19bdfhj.

5.1 Rationalization algorithms

Simple rationalization algorithm not currently included in the Rtnlzr class:

⟨ *Simple Rationalization by Joe Horn* 19b ⟩ ≡
⟨ *Rationalization Procedure* (⟨ *Simple Rationalization by Joe Horn Procedure* 20c ⟩) 19a ⟩
◇
Macro never referenced.

Smallest denominator rationalization procedure by Joe Horn.

⟨ *Smallest-Denominator Rationalization by Joe Horn* 19d ⟩ ≡
⟨ *Rationalization Procedure* (⟨ *Smallest-Denominator Rationalization by Joe Horn Procedure* 20a ⟩) 19a ⟩
◇
Macro referenced in 18.

Smallest denominator rationalization procedure by Joe Horn and Tony Hutchins; this is the most efficient method as implemented in RPL.

⟨ *Smallest-Denominator Rationalization by Joe Horn and Tony Hutchins* 19f ⟩ ≡
⟨ *Rationalization Procedure* (⟨ *Smallest-Denominator Rationalization by Joe Horn Procedure* 20a ⟩) 19a ⟩
◇
Macro referenced in 18.

Smallest denominator rationalization based on exercise 39 of [Knuth, §4.5.3]. This has been found the most efficient method (except for big tolerances) as implemented in Ruby.

⟨ *Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta* 19h ⟩ ≡
⟨ *Rationalization Procedure* (⟨ *Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta Procedure* 21b ⟩) 19a ⟩
◇
Macro referenced in 18.

A small modification of this algorthm has been used in tests, but is not currenly included in class Rtnlzr.

⟨ *Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta B* 19j ⟩ ≡
⟨ *Rationalization Procedure* (⟨ *Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta B Procedure* 22a ⟩) 19a ⟩
◇
Macro never referenced.

5.2 Implementation of the algorithms

⟨ *Smallest-Denominator Rationalization by Joe Horn Procedure* 20a ⟩ ≡
⟨ *Rationalization by Joe Horn Procedure* (⟨ *Extra Rationalization Step by Joe Horn* 20e ⟩) 20d ⟩◇

Macro referenced in 19eg.

⟨ *Simple Rationalization by Joe Horn Procedure* 20c ⟩ ≡
⟨ *Rationalization by Joe Horn Procedure* 20d ⟩◇

Macro referenced in 19c.

⟨ *Rationalization by Joe Horn Procedure* 20d ⟩ ≡

```
z,t = x,dx # renaming

a,b = t.nio_xr.nio_num_den
n0,d0 = (n,d = z.nio_xr.nio_num_den)
cn,x,pn,cd,y,pd,lo,hi,mid,q,r = 1,1,0,0,0,1,0,1,1,0,0
begin
  q,r = n.divmod(d)
  x = q*cn+pn
  y = q*cd+pd
  pn = cn
  cn = x
  pd = cd
  cd = y
  n,d = d,r
end until b*(n0*y-d0*x).abs <= a*d0*y
·1
num,den = x,y # renaming
◇
```

Macro referenced in 20ac.

⟨ *Extra Rationalization Step by Joe Horn* 20e ⟩ ≡

```
if q>1
  hi = q
  begin
    mid = (lo+hi).div(2)
    x = cn-pn*mid
    y = cd-pd*mid
    if b*(n0*y-d0*x).abs <= a*d0*y
      lo = mid
    else
      hi = mid
    end
  end until hi-lo <= 1
  x = cn - pn*lo
  y = cd - pd*lo
end
◇
```

Macro referenced in 20b.

Tony Hutchins has come up with PDR6, an improvement over PDQ2; though benchmarking does not show any speed improvement under Ruby.

⟨ *Smallest-Denominator Rationalization by Joe Horn and Tony Hutchins Procedure* 21a ⟩ ≡

```
a,b = dx.nio_xr.nio_num_den
n,d = x.nio_xr.nio_num_den
pc,ce = n,-d
pc,cd = 1,0
t = a*b
begin
  tt = (-pe).div(ce)
  pd,cd = cd,pd+tt*cd
  pe,ce = ce,pe+tt*ce
end until b*ce.abs <= t*cd
tt = t * (pe<0 ? -1 : (pe>0 ? +1 : 0))
tt = (tt*d+b*ce).div(tt*pd+b*pe)
num,den = (n*cd-ce-(n*pd-pe)*tt)/d, tt/(cd-tt*pd)
◇
```

Macro never referenced.

Here's the rationalization procedure based on the exercise by Knuth. We need first to calculate the limits (x-dx, x+dx) of the range where we'll look for the rational number. If we compute them using floating point and then convert then to fractions this method is always more efficient than the other procedures implemented here, but it may be less accurate. We can achieve perfect accuracy as the other methods by doing the substraction and addition with rationals, but then this method becomes less efficient than the others for a low number of iterations (low precision required).

⟨ *Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta Procedure* 21b ⟩ ≡

```
 x = x.nio_xr
 dx = dx.nio_xr
 xp,xq = (x-dx).nio_num_den
 yp,yq = (x+dx).nio_num_den

   a = []
   fin ,odd = false,false
   while !fin  && xp!=0 && yp!=0
     odd = !odd
     xp,xq = xq,xp
     ax = xp.div(xq)
     xp -= ax*xq

     yp,yq = yq,yp
     ay = yp.div(yq)
     yp -= ay*yq

     if  ax!=ay
       fin  = true
       ax,xp,xq = ay,yp,yq if  odd
     end
     a << ax # .to_i
   end
   a[-1] += 1 if xp!=0 && a.size>0
   p,q = 1,0
   (1..a.size).each{|i| p,q=q+p*a[-i],p}
   num,den = q,p
◇
```

Macro referenced in 19i.

La siguiente variante realiza una iteraci n menos si xq<xp y una iteraci n m s si xq>xp.

⟨ *Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta B Procedure* 22a ⟩ ≡

```
x = x.nio_xr
dx = dx.nio_xr
xq,xp = (x-dx).nio_num_den
yq,yp = (x+dx).nio_num_den

  a = []
  fin ,odd = false,true
  while !fin  && xp!=0 && yp!=0
    odd = !odd
    xp,xq = xq,xp
    ax = xp.div(xq)
    xp -= ax*xq

    yp,yq = yq,yp
    ay = yp.div(yq)
    yp -= ay*yq

    if  ax!=ay
      fin  = true
      ax,xp,xq = ay,yp,yq if  odd
    end
    a << ax # .to_i
  end
  a[-1] += 1 if xp!=0 && a.size>0
  p,q = 1,0
  (1..a.size).each{|i| p,q=q+p*a[-i],p}
  num,den = p,q
```
◇

Macro referenced in 19k.

5.3 Maximum denominator algorithm

This is a method by Joseph K. Horn that finds the best possible fraction given a maximum denominator. It computes continued fractions by a fast recursion formula, then make a single calculated jump backwards to the best possible fraction before the specified maximum denominator. He traces back the algorithm to "Textbook of Algebra" by G. Chrystal, 1st edition in 1889, in Part II, Chapter 32.

This algorithm was implementd in the User-RPL program `DEC2FRAC`, which I have adapted here for Ruby.

As this is a different approach, which uses no tolerance value, but instead needs a maximum denominator, I'll add it as a class-method. Note thas this method operates on floating point quantities (rather than integers as the other methods here do) and requires `ceil`, `floor`, `abs` and `round` on the floating point type (so it is applicable to `Float` and `BigDecimal`). As only the denominator is computed first, and then the numerator is computed using floating point math, this limits the magnitude and precision of the numerator. The precision of the floating point type may also make this method to miss the best approximation and yield something worse (with a lower denominator). For example, the fraction 39/329 if correctly approximated with a denominator not greater than 200 as 23/194 using `BigDecimal`. But the accumulated error when using `Float` make the method return 16/135.

⟨ *Nio classes* 22b ⟩ ≡

```
# Best  fraction  given  maximum denominator
# Algorithm Copyright (c)  1991 by Joseph K. Horn.
#
# The implementation of this  method uses floating  point
# arithmetic which limits  the magnitude and precision  of  the  results ,  specially
# using Float  values.
def Rtnlzr.max_denominator(f, max_den=1000000000, num_class=nil)
  return nil  if  max_den<1
  num_class ||= f.class
  return mth.ip(f),1  if  mth.fp(f)==0
```

```
    one = 1.prec(num_class)

      sign = f<0
      f = -f if sign

      a,b,c = 0,1,f
      while b<max_den and c!=0
        cc = one/c
        a,b,c = b, mth.ip(cc)*b+a, mth.fp(cc)
      end


      if b>max_den
        b -= a*mth.ceil((b-max_den)/Float(a))
      end


      f1,f2 = [a,b].collect{|x| mth.abs(mth.rnd(x*f)/x.prec(num_class)-f)}

      a = f1>f2 ? b : a

      num,den = mth.rnd(a*f).to_i,a
      den = 1 if mth.abs(den)<1

      num = -num if sign

    return num,den
  end
  ◇
```

Macro defined by 18, 22b, 23.
Macro referenced in 13b.

To simplifly the code I've defined this, RPL-like, functions:

⟨ *Nio classes* 23 ⟩ ≡

```
  class Rtnlzr
    private
    #Auxiliary floating-point functions
    module Mth # :nodoc:
      def self.fp(x)
        # y =x.modulo(1); return x<0 ? -y : y;
        x-ip(x)
      end

      def self.ip(x)
        # x.to_i.to_f
        (x<0 ? x.ceil : x.floor).to_i
      end

      def self.rnd(x)
        #x.round.to_i
        x.round
      end

      def self.abs(x)
        x.abs
      end

      def self.ceil(x)
        x.ceil.to_i
      end
    end
```

```
    def self.mth; Mth; end
  end
  ◇
```

Macro defined by 18, 22b, 23.
Macro referenced in 13b.

5.4 Float to Rational conversion

Having added `Rtnlzr.max_denominator`, I'll use it if the parameter to `nio_r` is not a `Tolerance`.

⟨*Required Modules* 24a⟩ ≡

```
  require 'rational'
  ◇
```

Macro defined by 14bc, 24ac.
Macro referenced in 13b.

⟨*classes* 24b⟩ ≡

```
  class Float
    ⟨rdoc commentary for Float#nio_r 25h⟩
    def nio_r(tol = Nio::Tolerance.big_epsilon)
      case tol
        when Integer
          Rational(*Nio::Rtnlzr.max_denominator(self,tol,Float))
        else
          Rational(*Nio::Rtnlzr.new(Nio::Tol(tol)).rationalize(self))
      end
    end
  end
  ◇
```

Macro defined by 24bd.
Macro referenced in 13b.

5.5 BigDecimal to Rational conversion

⟨*Required Modules* 24c⟩ ≡

```
  require 'bigdecimal'
  ◇
```

Macro defined by 14bc, 24ac.
Macro referenced in 13b.

⟨*classes* 24d⟩ ≡

```
  class BigDecimal
    ⟨rdoc commentary for BigDecimal#nio_r 25f⟩
    def nio_r(tol = nil)
      tol ||= BigTolerance.decimals([precs[0],Float::DIG].max,:sig)
      case tol
        when Integer
          Rational(*Nio::Rtnlzr.max_denominator(self,tol,BigDecimal))
        else
          Rational(*Nio::Rtnlzr.new(Nio::BigTol(tol)).rationalize(self))
      end
    end
  end
  ◇
```

Macro defined by 24bd.
Macro referenced in 13b.

6 rdoc documentation

⟨ *rdoc commentary for rntlzr.rb* 25a ⟩ ≡

```
#
# Author::     Javier Goizueta (mailto:javier@goizueta.info)
# Copyright:: Copyright (c) 2002-2004 Javier Goizueta & Joe Horn
# License ::   Distributes under the GPL license
#
# This file provides conversion from floating point numbers
# to rational numbers.
# Algorithms by Joe Horn are used.
#
# The rational approximation algorithms are implemented in the class Nio::Rtnlzr
# and there's an interface to the chosen algorithms through:
# * Float#nio_r
# * BigDecimal#nio_r
# There's also exact rationalization implemented in:
# * Float#nio_xr
# * BigDecimal#nio_r
```
◇

Macro referenced in 13b.

⟨ *rdoc commentary for Rtnlzr* 25b ⟩ ≡

```
# This class provides conversion of fractions
# (as approximate floating point numbers)
# to rational numbers.
```
◇

Macro referenced in 18.

⟨ *rdoc commentary for flttol.rb* 25c ⟩ ≡

```
#
# Author::     Javier Goizueta (mailto:javier@goizueta.info)
# Copyright:: Copyright (c) 2002-2004 Javier Goizueta
# License ::   Distributes under the GPL license
#
# This module provides a numeric tolerance class for Float and BigDecimal.
```
◇

Macro referenced in 1a.

⟨ *rdoc commentary for Tolerance* 25d ⟩ ≡

```
# This class represents floating point tolerances for Float numbers
# and allows comparison within the specified tolerance.
```
◇

Macro referenced in 2c.

⟨ *rdoc commentary for BigTolerance* 25e ⟩ ≡

```
# This class represents floating point tolerances for BigDecimal numbers
# and allows comparison within the specified tolerance.
```
◇

Macro referenced in 9b.

⟨ *rdoc commentary for BigDecimal#nio_r* 25f ⟩ ≡

⟨ *rdoc commentary for nio_r* (BigTolerance) 26a ⟩◇

Macro referenced in 24d.

⟨ *rdoc commentary for Float#nio_r* 25h ⟩ ≡

⟨ *rdoc commentary for nio_r* (Tolerance) 26a ⟩◇

Macro referenced in 24b.

⟨*rdoc commentary for nio_r* 26a⟩ ≡
```
# Conversion to Rational. The optional argument must be one of:
# - a Nio::·1 that defines the admisible tolerance;
#   in that case, the smallest denominator rational within the
#   tolerance will be found (which may take a long time for
#   small tolerances.)
# - an integer that defines a maximum value for the denominator.
#   in which case, the best approximation with that maximum
#   denominator will be returned.◇
```
Macro referenced in 25fh.

⟨*rdoc commentary for BigDecimal#nio_xr* 26b⟩ ≡
```
⟨rdoc commentary for nio_xr 26f⟩◇
```
Macro referenced in 16b.

⟨*rdoc commentary for Float#nio_xr* 26c⟩ ≡
```
⟨rdoc commentary for nio_xr 26f⟩◇
```
Macro referenced in 15a.

⟨*rdoc commentary for Integer#nio_xr* 26d⟩ ≡
```
⟨rdoc commentary for nio_xr 26f⟩◇
```
Macro never referenced.

⟨*rdoc commentary for Rational#nio_xr* 26e⟩ ≡
```
⟨rdoc commentary for nio_xr 26f⟩◇
```
Macro never referenced.

⟨*rdoc commentary for nio_xr* 26f⟩ ≡
```
# Conversion to Rational preserving the exact value of the number.◇
```
Macro referenced in 26bcde.

The constructor methods are module functions with capitalized names that need to be documented apart.

⟨*Nio constructor methods rdoc* 26g⟩ ≡
```

# This module contains some constructor-like module functions
# to help with the creation of tolerances and big-decimals.
#
# =BigDec
⟨rdoc for BigDec 27a⟩
#
# =Tol
⟨rdoc for Tol 27b⟩
#
# =BigTol
⟨rdoc for BigTol 27c⟩◇
```
Macro referenced in 1a.

⟨*rdoc for BigDec* 27a⟩ ≡

```
# BigDec(x) -> a BigDecimal
# BigDec(x, precision) -> a BigDecimal
# BigDec(x,:exact) -> a BigDecimal
# This is a shortcut to define a BigDecimal without using quotes
# and a general conversion to BigDecimal method.
#
# The second parameter can be :exact to try for an exact conversion
#
# Conversions from Float have issues that should be understood; :exact
# conversion will use the exact internal value of the Float, and when
# no precision is specified, a value as simple as possible expressed as
# a fraction will be used.◇
```

Macro referenced in 17a, 26g.

⟨*rdoc for Tol* 27b⟩ ≡

```
# Tol(x) -> a Tolerance
# This module function will convert its argument to a Noi::Tolerance
# or a Noi::BigTolerance depending on its argument;
#
# Values of type Tolerance,Float,Integer (for Tolerance) or
# BigTolerance,BigDecimal (for BigTolerance) are accepted.◇
```

Macro referenced in 13a, 26g.

⟨*rdoc for BigTol* 27c⟩ ≡

```
# BigTol(x) -> a BigTolerance
# This module function will convert its argument to a Noi::BigTolerance
#
# Values of type BigTolerance or Numeric are accepted.◇
```

Macro referenced in 13a, 26g.

7 Patch

In some Ruby implementations there's a bug in `Float#to_i` which produces incorrect results. This has been detected in Ruby 1.8.4 compiled for `x86_64_linux`. Here we'll try to detect the problem and apply a quick patch. The resulting method will be slower but will produce correct results.

⟨*flttol definitions* 27d⟩ ≡

```
# :stopdoc:
# A problem has been detected with Float#to_i() in some Ruby versiones
# (it has been found in Ruby 1.8.4 compiled for x86_64_linux|)
# This problem makes to_i produce an incorrect sign on some cases.
# Here we try to detect the problem and apply a quick patch,
# although this will slow down the method.
if 4.611686018427388e+018.to_i < 0
  class Float
    alias _to_i to_i
    def to_i
      neg = (self < 0)
      i = _to_i
      i_neg = (i < 0)
      i = -i if neg != i_neg
      i
    end
  end
end
# :startdoc:
◇
```

Macro defined by 2a, 27d.
Macro referenced in 1a.

8 Tests

8.1 Test data

We'll load the data for the tests in a global variable.

⟨ *Tests setup* 28a ⟩ ≡
```
    $data = YAML.load(File.read(File.join(File.dirname(__FILE__),'data.yaml'))).collect{|x| [x].pack('H*').
unpack('E')[0]}
```
◇

Macro referenced in 14a.

8.2 Test methods

⟨ *Tests* 28b ⟩ ≡
```
  def test_basic_rtnlzr
    # basic Rtnlzr tests
    r = Rtnlzr.new
    assert_equal [13,10], r.rationalize(1.3)
    assert_equal [13,10], Rtnlzr.max_denominator(1.3,10)
    assert_equal [13,10], Rtnlzr.max_denominator(BigDecimal('1.3'),10)
    assert_equal [1,3], Rtnlzr.max_denominator(1.0/3,10)
    assert_equal [1,3], Rtnlzr.max_denominator(BigDecimal('1')/3,10)

    # basic tests of Float#nio_r
    assert_equal Rational(1,3), (1.0/3.0).nio_r
    assert_equal Rational(2,3), (2.0/3.0).nio_r
    assert_equal Rational(1237,1234), (1237.0/1234.0).nio_r
    assert_equal Rational(89,217), (89.0/217.0).nio_r

    # rationalization of Floats using a tolerance
    t = Tolerance.new(1e-15,:sig)
    assert_equal Rational(540429, 12500),43.23432.nio_r(t)
    assert_equal Rational(6636649, 206596193),0.032123772.nio_r(t)
    assert_equal Rational(280943, 2500000), 0.1123772.nio_r(t)
    assert_equal Rational(39152929, 12500), 3132.23432.nio_r(t)
    assert_equal Rational(24166771439, 104063), 232232.123223432.nio_r(t)
    assert_equal Rational(792766404965, 637), 1244531247.98273123.nio_r(t)
    #$data.each do |x|
    # assert t.equals?(x, x.nio_r(t).to_f), "out of tolerance: #{x.inspect} #{x.nio_r(t).inspect}"
    #end

    # rationalization with maximum denominator
    assert_equal Rational(9441014047197, 7586), (1244531247.98273123.nio_r(10000))
    assert_equal Rational(11747130449709, 9439), BigDecimal('1244531247.982731230').nio_r(10000)


    # approximate a value in [0.671,0.672];
    # Float
    assert_equal [43,64], Rtnlzr.new(Tolerance.new(0.0005)).rationalize(0.6715)
    assert_equal [43,64], Rtnlzr.new(Tol(0.0005)).rationalize(0.6715)
    assert_equal [43,64], Rtnlzr.new(Rational(5,10000)).rationalize(0.6715)
    # BigDecimal
    assert_equal [43,64], Rtnlzr.new(BigTolerance.new(BigDecimal('0.0005'))).rationalize(BigDecimal('
0.6715'))
    assert_equal [43,64], Rtnlzr.new(Tol(BigDecimal('0.0005'))).rationalize(BigDecimal('0.6715'))
    assert_equal [43,64], Rtnlzr.new(Rational(5,10000)).rationalize(BigDecimal('0.6715'))
    #
    assert_equal Rational(43,64), 0.6715.nio_r(0.0005)
    assert_equal Rational(43,64), 0.6715.nio_r(Rational(5,10000))
    assert_equal Rational(47,70), 0.6715.nio_r(70)
    assert_equal Rational(45,67), 0.6715.nio_r(69)
```

```
    assert_equal Rational(2,3), 0.6715.nio_r(10)

    # some PI tests
    assert_equal Rational(899125804609,286200632530), BgMth.PI(64).nio_r(BigTolerance.new(BigDec('261E
  -24')))
    assert_equal Rational(899125804609,286200632530), BgMth.PI(64).nio_r(Tol(BigDec('261E-24')))
    assert_equal Rational(899125804609,286200632530), BgMth.PI(64).nio_r(BigDec('261E-24'))
    assert_equal Rational(899125804609,286200632530), BgMth.PI(64).nio_r(BigDec(261E-24))
    assert_equal Rational(899125804609,286200632530), BgMth.PI(64).nio_r(261E-24)

    # BigDecimal tests
    #t = BigTolerance.new(BigDecimal('1e-15'),:sig)
    t = BigTolerance.decimals(20,:sig)
    $data.each do |x|
      x = BigDec(x,:exact)
      q = x.nio_r(t)
      assert t.equals?(x, BigDec(q)), "out of tolerance: #{x.inspect} #{BigDec(q)}"
    end
  end
◇
```

Macro defined by 28b, 29.
Macro referenced in 14a.

⟨ *Tests 29* ⟩ ≡

```
    def test_compare_algorithms
      r = Rtnlzr.new(Tolerance.new(1e-5,:sig))
      ($data + $data.collect{|x| -x}).each do |x|
        q1 = r.rationalize_Knuth(x)
        q2 = r.rationalize_Horn(x)
        q3 = r.rationalize_HornHutchins(x)
        #q4 = r.rationalize_KnuthB(x)
        q1 = [-q1[0],-q1[1]] if q1[1] < 0
        q2 = [-q2[0],-q2[1]] if q2[1] < 0
        q3 = [-q3[0],-q3[1]] if q3[1] < 0
        assert_equal q1, q2
        assert_equal q1, q3
        #assert_equal q1, q4
      end
      r = Rtnlzr.new(Tolerance.epsilon)
      ($data + $data.collect{|x| -x}).each do |x|
        q1 = r.rationalize_Knuth(x)
        q2 = r.rationalize_Horn(x)
        q3 = r.rationalize_HornHutchins(x)
        q1 = [-q1[0],-q1[1]] if q1[1] < 0
        q2 = [-q2[0],-q2[1]] if q2[1] < 0
        q3 = [-q3[0],-q3[1]] if q3[1] < 0
        #q4 = r.rationalize_KnuthB(x)
        assert_equal q1, q2
        assert_equal q1, q3
        #assert_equal q1, q4
      end

    end
◇
```

Macro defined by 28b, 29.
Macro referenced in 14a.

9 Indices

9.1 Files

"lib/nio/flttol.rb" Defined by 1a.

"lib/nio/rtnlzr.rb" Defined by 13b.
"test/test_rtnlzr.rb" Defined by 14a.

9.2 Macros

⟨ BigTolerance Relative Comparison 12a ⟩ Referenced in 12bgl.
⟨ BigTolerance Significative Comparison 11d ⟩ Referenced in 12bgl.
⟨ BigTolerance aprxEquals? 12l ⟩ Referenced in 9b.
⟨ BigTolerance constructors 10cde ⟩ Referenced in 9b.
⟨ BigTolerance equals? 12g ⟩ Referenced in 9b.
⟨ BigTolerance lessThan? 12b ⟩ Referenced in 9b.
⟨ BigTolerance private 10ab, 12q ⟩ Referenced in 9b.
⟨ Extra Rationalization Step by Joe Horn 20e ⟩ Referenced in 20b.
⟨ Initialize BigTolerance from digits 11c ⟩ Referenced in 10c.
⟨ Initialize BigTolerance 11b ⟩ Referenced in 10b.
⟨ Initialize Tolerance from digits 6a ⟩ Referenced in 4a.
⟨ Initialize Tolerance 5c ⟩ Referenced in 3.
⟨ License 1b ⟩ Referenced in 1a, 13b, 14a.
⟨ Nio classes 18, 22b, 23 ⟩ Referenced in 13b.
⟨ Nio constructor methods rdoc 26g ⟩ Referenced in 1a.
⟨ Nio definitions ? ⟩ Referenced in 13b.
⟨ Nio functions ? ⟩ Referenced in 13b.
⟨ Rationalization Procedure 19a ⟩ Referenced in 19bdfhj.
⟨ Rationalization by Joe Horn Procedure 20d ⟩ Referenced in 20ac.
⟨ Relative Comparison 6b ⟩ Referenced in 7bgl.
⟨ Required Modules 14bc, 24ac ⟩ Referenced in 13b.
⟨ Significant Comparison 6c ⟩ Referenced in 7bgl.
⟨ Simple Rationalization by Joe Horn Procedure 20c ⟩ Referenced in 19c.
⟨ Simple Rationalization by Joe Horn 19b ⟩ Not referenced.
⟨ Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta B Procedure 22a ⟩ Referenced in 19k.
⟨ Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta B 19j ⟩ Not referenced.
⟨ Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta Procedure 21b ⟩ Referenced in 19i.
⟨ Smallest-Denominator Rationalization by Donald Knuth and Javier Goizueta 19h ⟩ Referenced in 18.
⟨ Smallest-Denominator Rationalization by Joe Horn Procedure 20a ⟩ Referenced in 19eg.
⟨ Smallest-Denominator Rationalization by Joe Horn and Tony Hutchins Procedure 21a ⟩ Not referenced.
⟨ Smallest-Denominator Rationalization by Joe Horn and Tony Hutchins 19f ⟩ Referenced in 18.
⟨ Smallest-Denominator Rationalization by Joe Horn 19d ⟩ Referenced in 18.
⟨ Tests setup 28a ⟩ Referenced in 14a.
⟨ Tests 28b, 29 ⟩ Referenced in 14a.
⟨ Tolerance attributes 8b ⟩ Referenced in 2c, 9b.
⟨ Tolerance constructors 4abcd, 5a ⟩ Referenced in 2c.
⟨ Tolerance methods 8a ⟩ Referenced in 2c, 9b.
⟨ Tolerance private 3, 6d, 7a, 8c ⟩ Referenced in 2c.
⟨ aprxEquals? 7l ⟩ Referenced in 2c.
⟨ classes 24bd ⟩ Referenced in 13b.
⟨ compute bits per Float 2b ⟩ Referenced in 2a.
⟨ definitions 15a, 16b, 17bc ⟩ Referenced in 13b.
⟨ equals? 7g ⟩ Referenced in 2c.
⟨ flttol Required Modules 9a ⟩ Referenced in 1a.
⟨ flttol classes 2c, 5b, 9b, 11a ⟩ Referenced in 1a.
⟨ flttol definitions 2a, 27d ⟩ Referenced in 1a.
⟨ flttol functions 13a, 17a ⟩ Referenced in 1a.
⟨ lessThan? 7b ⟩ Referenced in 2c.
⟨ rdoc commentary for BigDecimal#nio_r 25f ⟩ Referenced in 24d.
⟨ rdoc commentary for BigDecimal#nio_xr 26b ⟩ Referenced in 16b.
⟨ rdoc commentary for BigTolerance 25e ⟩ Referenced in 9b.
⟨ rdoc commentary for Float#nio_r 25h ⟩ Referenced in 24b.
⟨ rdoc commentary for Float#nio_xr 26c ⟩ Referenced in 15a.
⟨ rdoc commentary for Integer#nio_r ? ⟩ Referenced in 17b.
⟨ rdoc commentary for Integer#nio_xr 26d ⟩ Not referenced.
⟨ rdoc commentary for Rational#nio_r ? ⟩ Referenced in 17c.
⟨ rdoc commentary for Rational#nio_xr 26e ⟩ Not referenced.

⟨ rdoc commentary for Rtnlzr 25b ⟩ Referenced in 18.
⟨ rdoc commentary for Tolerance 25d ⟩ Referenced in 2c.
⟨ rdoc commentary for flttol.rb 25c ⟩ Referenced in 1a.
⟨ rdoc commentary for nio_r 26a ⟩ Referenced in 25fh.
⟨ rdoc commentary for nio_xr 26f ⟩ Referenced in 26bcde.
⟨ rdoc commentary for rntlzr.rb 25a ⟩ Referenced in 13b.
⟨ rdoc for BigDec 27a ⟩ Referenced in 17a, 26g.
⟨ rdoc for BigTol 27c ⟩ Referenced in 13a, 26g.
⟨ rdoc for Tol 27b ⟩ Referenced in 13a, 26g.
⟨ scratch 15b, 16a ⟩ Not referenced.

9.3 Identifiers

References

[Knuth] "The Art of Computer Programming 2d ed. Vol.2"
Seminumerical Algorithms
Donald E. Knuth
1981
Addison-Wesley
ISBN: 0-201-03822-6