

1 Repeating decimals

This small Ruby class has its origins on old notes I wrote about reading decimal representations of rational numbers, including repeating decimals. A simple and clear exposition of the theory behind it can be found at [Ross, §16], “Decimal Expansions of Real Numbers”.

This module may serve as a low-level help for the implementation of rational numbers, to handle representation of rationals as decimal numbers. Such rational numbers class could be complemented with an utility to convert floating point numbers to rationals, using the algorithms from `geo/rtnlZR`. (for the implementation of rational numbers, see also Algebra1 library).

The notation used for repeating decimal is either $0.\overline{123}$ or $0.123123123\dots$. Here we’ll use adaptations of both to simple text format.

For a repeating decimal such as $k.d_0d_1\dots d_{l-1}\overline{d_l\dots d_{l+r}}$ we’ll store the integer k in an instance variable `@ip` (integer part), the decimal digits in an array `@d` of numbers, and the position of the first repeating decimal l in `@rep_i` (if `nil` it is assumed to be the first zero digit, at $l+r+1$). We’ll use the number’s absolute value for the integer part and decimals and keep the sign separated in `@sign` because we cannot use `@ip sign` when it is zero.

```

"lib/nio/repdec.rb" 1a ≡
  # repdec.rb -- Repeating Decimals (Repeating Numerals, actually)
  <License 1c>
  require 'nio/tools'
  module Nio
    <Auxiliar classes 2e, ... >
    <RepDec class 2a >
    <Auxiliary functions 12a >
    module_function
    <Nio functions 16b >
  end
  ◇

"test/test_repdec.rb" 1b ≡

  <License 1c>
  #require File.dirname(__FILE__) + '/test_helper.rb'
  require 'test/unit'

  require 'nio/repdec'
  include Nio
  require 'yaml'

  class TestRepdec < Test::Unit::TestCase

    def setup
      <Tests setup? >
    end

    <Tests 17 >
  end
  ◇

<License 1c > ≡
  # 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 [1ab](#).

```

< RepDec class 2a > ≡
  # RepDec handles repeating decimals (repeating numerals actually)
  class RepDec
    include StateEquivalent
    < RepDec class variables and constants 4b, ... >
    < RepDec members 2b, ... >
    #protected
    < RepDec protected members 2d >
  end
  ◇

```

Macro referenced in [1a](#).

We'll initialize new RepDec instances to zero.

```

< RepDec members 2b > ≡
  def initialize (b=10)
    setZ(b)
  end
  ◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).

Macro referenced in [2a](#).

Defines: `initialize` [4b](#), [6b](#).

Member to set to the value 0.

```

< RepDec members 2c > ≡
  def setZ(b=10)
    @ip = 0;
    @d = [];
    @rep_i = nil;
    @sign = 0;
    @radix = b;
  self
  end
  ◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).

Macro referenced in [2a](#).

```

< RepDec protected members 2d > ≡
  attr_reader :d, :ip, :rep_i, :sign;
  attr_writer :d, :ip, :rep_i, :sign;
  ◇

```

Macro referenced in [2a](#).

We will define an exception class for errors occurring here:

```

< Auxiliar classes 2e > ≡
  class RepDecError <StandardError
  end
  ◇

```

Macro defined by [2e](#), [6b](#).

Macro referenced in [1a](#).

1.1 Special values

To handle quotients such as $1/0$ or $0/0$ we'll need special values for decimal representations; namely infinite and indeterminate (also known as not-a-number).

With respect to infinity, the kind of infinity introduced for this purpose, extending the set of real values, is denoted as ∞ . Often, when working with complex numbers, a special form, the directed infinity, which has a direction (an argument or angle θ) is used: $e^{i\theta}\infty$. Two particular cases of directed infinity are $+\infty = e^{i0}\infty$ and $-\infty = e^{i\pi}\infty$.

There are two common, coherent approaches to handling this special quantities.

The first one is the one used by *Mathematica*. This systems handles all the mentioned kinds of infinity and returns exact, algebraic answers. In this approach, both $+1/0$ and $-1/0$ produce a non-directed form of infinity that we could denote as ∞ (different from the directed $+\infty$). This is mathematically correct, since the zero can represent a sequence with that limit, an its values may be negative or positive; we haven't enough information to determine the orientation of the resulting infinity. The effect of this is that we have cases such as $1/(1/\infty) = \infty$ in which $1/(1/x) = x$ is not an identity.

The other common approach is the one employed by IEEE floating point; to have directed infinities and also directed zeros. IEEE floating point applies only to real numbers, so rather than having directed quantities (with any argument) we have signed quantities (with arguments 0 and π . So here we have both $+0$ and -0 . In this case we have directed infinities as the result of division by zero: $+1/+0 = +\infty$, $+1/-0 = -\infty$, $-1/+0 = -\infty$ $-1/-0 = +\infty$. This approach is convenient here, because the infinities may be produced also by overflow, i.e. by division by a small (but different from zero and signed) quantity. This approach also preserves the identity $1/(1/x) = x$.

In our case the first approach seems more appropriate, since handling signed zeros seems too much for this simple class, and overflow is not an issue.

On the other hand, it seems convenient to preserve the identity $Q(x, y) == Q(r.SetQ(x, y).GetQ())$, where we denote by $Q(x, y)$ the rational number x/y . One simple way to do this is to have two infinities $+\infty$ and $-\infty$ and consider 0 as the signed, positive zero of IEEE.

So we will maintain three special values for our decimal representations; with the following text representations (in accordance with class `Float`) `NaN` for the indeterminate $0/0$ (not-a-number in IEEE), `Infinity` for $1/0$, and `-Infinity` for $-1/0$.

```
< Write special values 3a > ≡
  if !ip.is_a?(Integer) then
    str=opt.nan_txt if ip==:indeterminate;
    str=opt.inf_txt if ip==:posinfinity
    str='-' +opt.inf_txt if ip==:neginfinity
    return str;
  end
  ◇
```

Macro referenced in 11a.

```
< Read special values 3b > ≡
  str.upcase!
  if str[i,opt.nan_txt.size]==opt.nan_txt.upcase
    i_str = :indeterminate;
  elsif str[i,opt.inf_txt.size]==opt.inf_txt.upcase
    i_str = neg? :neginfinity : :posinfinity;
  end
  ◇
```

Macro referenced in 9a.

```
< Get special values 3c > ≡
  if !ip.is_a?(Integer) then
    y = 0;
    x=0 if ip==:indeterminate;
    x=1 if ip==:posinfinity
    x=-1 if ip==:neginfinity
    return x,y;
  end if
  ◇
```

Macro referenced in 16a.

```

< Set special values 4a > ≡
  if y==0 then
    if x==0 then
      @ip = :indeterminate
    else
      @ip = xy_sign==-1 ? :neginfinity : :posinfinity
    end
  return self
end
end
◇

```

Macro referenced in [15](#).

2 Text representation

Initially configuration parameters were kept in class variables, now configuration objects have been introduced so that different options can be used for different operations.

```

< RepDec class variables and constants 4b > ≡
  class Opt # :nodoc:
    include StateEquivalent
    def initialize () #default options
      < Default Options 5b, ... >
    end
    attr_accessor :begin_rep, :end_rep, :auto_rep, :dec_sep, :grp_sep, :grp, :max_d
    attr_accessor :nan_txt, :inf_txt
    < class Opt members 5a, ... >
  end
end
◇

```

Macro defined by [4bc](#).

Macro referenced in [2a](#).

Uses: `initialize` [2b](#).

We'll have a default options object for default arguments.

```

< RepDec class variables and constants 4c > ≡
  DEF_OPT=Opt.new
◇

```

Macro defined by [4bc](#).

Macro referenced in [2a](#).

We'll add some `set` members to the options for the convenience of specifying options in a single line:

```
RepDec::Opt.new.set_delim('<') .set_dec(',',')
```

```

⟨ class Opt members 5a ⟩ ≡
  def set_delim(begin_d,end_d='')
    @begin_rep = begin_d
    @end_rep = end_d
    return self
  end
  def set_suffix(a)
    @auto_rep = a
    return self
  end
  def set_sep(d)
    @dec_sep = a
    return self
  end
  def set_grouping(sep,g=[])
    @grp_sep = a
    @grp = g
    return self
  end
  def set_special(nan_txt, inf_txt)
    @nan_txt = nan_txt
    @inf_txt = inf_txt
    return self
  end
  ◇

```

Macro defined by [5a, 7bc](#).

Macro referenced in [4b](#).

First we'll add methods to read a number from and to a text string. We'll use a special character to delimit the start of the repeating section, and optionally another character for the end of the section (this can be an empty string).

```

⟨ Default Options 5b ⟩ ≡
  @begin_rep = '<'
  @end_rep = '>'
  ◇

```

Macro defined by [5bcd, 6a, 7a, 14c](#).

Macro referenced in [4b](#).

Also we'll use a special character at the end of a representation to specify implicit repetition (of the longest final repeated section of the number). Characters after this will be ignored, so if we use a dot for this purpose we can safely use three dots as an ellipsis.

```

⟨ Default Options 5c ⟩ ≡
  @auto_rep = '...'
  ◇

```

Macro defined by [5bcd, 6a, 7a, 14c](#).

Macro referenced in [4b](#).

```

⟨ Default Options 5d ⟩ ≡
  @dec_sep = '.'
  @grp_sep = ','
  @grp = [] # [3] for thousands separators
  ◇

```

Macro defined by [5bcd, 6a, 7a, 14c](#).

Macro referenced in [4b](#).

```

<Default Options 6a> ≡
  @inf_txt = 'Infinity'
  @nan_txt = 'NaN'
  ◇

```

Macro defined by [5bcd](#), [6a](#), [7a](#), [14c](#).
 Macro referenced in [4b](#).

2.1 Digits and base

We will add an option to define the characters reconigzed as digits and its values.

We will define a class to contain the information about digits definitions because it can be handy elsewhere.

Note that the definition of this class will be before the definition of `RepDec`. It must be so because a constant defined in that class executes indirectly, in its definitions the constructor for a `DigitsDef` object, so this must be defined when the constant is created.

The digits define a numerical base as well (the number of digits). The repeating decimals have been upgraded to work in any base (so they shouldn't be called decimals anymore).

```

<Auxiliar classes 6b> ≡
class DigitsDef
  include StateEquivalent
  def initialize (ds='0123456789', cs=true)
    @digits = ds
    @casesens = cs
    @dncase = (ds.downcase==ds)
    @radix = @digits.size
  end
  def is_digit?(ch_code)
    ch_code = set_case(ch_code) unless @casesens
    @digits.include?(ch_code)
  end
  def digit_value(ch_code)
    ch_code = set_case(ch_code) unless @casesens
    @digits.index(ch_code)
  end
  def digit_char(v)
    @digits[v]
  end
  def digit_char_safe(v)
    v>=0 && v<@radix ? @digits[v] : nil
  end
  def radix
    @radix
  end
  def DigitsDef.base(b,dncase=false,casesens=false)
    dgs = "0123456789ABCDEFGHIJKLMNPOQRSTUVWXYZ"[0,b]
    dgs.downcase! if dncase
    DigitsDef.new(dgs,casesens)
  end
private
  def set_case(ch_code)
    ch_code = ch_code.chr if ch_code.kind_of?(Numeric)
    @dncase ? ch_code.downcase[0] : ch_code.upcase[0]
  end
end
◇

```

Macro defined by [2e](#), [6b](#).
 Macro referenced in [1a](#).
 Uses: `initialize` [2b](#).

```

< Default Options 7a > ≡
  @digits = DigitsDef.new
  @digits_defined = false
  ◇

```

Macro defined by [5bcd](#), [6a](#), [7a](#), [14c](#).
 Macro referenced in [4b](#).

```

< class Opt members 7b > ≡
  def set_digits(ds, dncase=false, casesens=false)
    if ds
      @digits_defined = true
      if ds.kind_of?(DigitsDef)
        @digits = ds
      elsif ds.kind_of?(Numeric)
        @digits = DigitsDef.base(ds, dncase, casesens)
      else
        @digits = DigitsDef.new(ds,casesens)
      end
    end
    else
      @digits = DigitsDef.new
      @digits_defined = false
    end
    self
  end
  ◇

```

Macro defined by [5a](#), [7bc](#).
 Macro referenced in [4b](#).

```

< class Opt members 7c > ≡
  attr_accessor :digits
  def digits_defined?
    @digits_defined
  end
  ◇

```

Macro defined by [5a](#), [7bc](#).
 Macro referenced in [4b](#).

The handling of bases, added late in the development is a little messy; here is how it works: Now the options `RepDec::Opt` contain information in `digits` about the numerical base (radix) and the digits used. The `RepDec` class itself contains information about the base (radix) used in it. This base, 10 by default, can be set in the constructor, but it is overridden by the radix defined by the options passed to `setS` and `setQ` which persistently establish the radix of the `RepDec`. Other instance methods that have an options parameter, `getS` and `getQ`, do not change the `RepDec` radix; in these cases the radix of the `RepDec` and that defined by the options should be the same.

3 Reading from text

```

⟨ RepDec members 8a ⟩ ≡
  def setS(str, opt=DEF_OPT)
    setZ(opt.digits_defined? ? opt.digits.radix : @radix);
    sgn, i_str, f_str, ri, detect_rep = RepDec.parse(str, opt)
    if i_str.kind_of?(Symbol)
      @ip = i_str
    else
      @ip = i_str.to_i(@radix); # this assumes conventional digits
    end
    @sign = sgn
    @rep_i = ri if ri
    f_str.each_byte{|b| @d.push opt.digits.digit_value(b)} unless f_str.nil?

    if detect_rep then
      ⟨ Detect repetition 9d ⟩
    end

    ⟨ Remove trailing zeros 10d ⟩
  self
end
◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).

Macro referenced in [2a](#).

```

⟨ RepDec members 8b ⟩ ≡
  def RepDec.parse(str, opt=DEF_OPT)
    sgn, i_str, f_str, ri, detect_rep = nil, nil, nil, nil, nil

    i = 0;
    l = str.length;

    detect_rep = false;

    ⟨ Skip whitespace in str 9c ⟩
    ⟨ Read integral part 9a ⟩
    unless i_str.kind_of?(Symbol)
      j = 0;
      f_str = ''
      while i < l
        ch = str[i, 1];
        if ch == opt.begin_rep then
          ri = j;
        elsif ch == opt.end_rep then
          i = 1;
        elsif ch == opt.auto_rep[0, 1] then
          detect_rep = true;
          i = 1;
        else
          f_str << ch
          j += 1;
        end
        i += 1;
      end
      end
      return [sgn, i_str, f_str, ri, detect_rep]
    end
  ◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).

Macro referenced in [2a](#).

```

⟨ Read integral part 9a ⟩ ≡
  neg = false;
  ⟨ Read sign from str 9b ⟩
  ⟨ Skip whitespace in str 9c ⟩
  ⟨ Read special values 3b ⟩
  unless i_str
    i_str = "0";
    while i < 1 && str[i,1] != opt.dec_sep
      break if str[i,opt.auto_rep.length] == opt.auto_rep && opt.auto_rep != ' '
      i_str += str[i,1] if str[i,1] != opt.grp_sep
      i += 1;
    end
    sgn = neg ? -1 : +1
    i += 1; # skip the decimal separator
  end
  ◇

```

Macro referenced in [8b](#).

```

⟨ Read sign from str 9b ⟩ ≡
  neg = true if str[i,1] == '-'
  i += 1 if str[i,1] == '-' || str[i,1] == '+'
  ◇

```

Macro referenced in [9a](#).

```

⟨ Skip whitespace in str 9c ⟩ ≡
  i += 1 while i < str.length && str[i,1] =~ /\s/
  ◇

```

Macro referenced in [8b](#), [9a](#).

```

⟨ Detect repetition 9d ⟩ ≡
  for l in 1..(@d.length/2)
    ⟨ Reverse order of checking 9e ⟩
    if @d[-l..-1] == @d[-2*l...-1]
      ⟨ Reduce multiple repetitions 10a ⟩
      ⟨ Remove repeated digits 10b ⟩
      ⟨ Find additional repetitions 10c ⟩
      break
    end
  end
  ◇

```

Macro referenced in [8a](#).

To detect longest repetitions instead of the shortest one, we reverse the order of repetition search (first long strings) and then we must check for the possibility of having detected several repetitions (repeated in turn). In that case we'll tidy things up and use the minimal repetition.

```

⟨ Reverse order of checking 9e ⟩ ≡
  l = @d.length/2 + 1 - 1; ◇

```

Macro referenced in [9d](#).

```

⟨ Reduce multiple repetitions 10a ⟩ ≡
  for m in 1..l
    if l.modulo(m)==0 then
      reduce_l = true;
      for i in 2..l/m
        if @d[-m..-1]!=@d[-i*m...-i*m+m] then
          reduce_l = false;
          break;
        end
      end
      if reduce_l then
        l = m
        break
      end
    end
  end
  end
  ◇

```

Macro referenced in [9d](#).

By matching the longest possible repetition, rather than the shortest one, we can safely specify possibly ambiguous cases. For example, $1.234545234545\dots$ can be used to specify $1.\overline{234545}$ and $1.23454523454545\dots$ can be used to specify $1.2345452345\overline{45}$.

We don't need repeated digits.

```

⟨ Remove repeated digits 10b ⟩ ≡
  @rep_i = @d.length - 2*1;
  1.times { @d.pop }
  ◇

```

Macro referenced in [9d](#), [10c](#).

And finally, there may be additional previous instances of the repeated section.

```

⟨ Find additional repetitions 10c ⟩ ≡
  while @d.length >= 2*1 && @d[-1..-1]==@d[-2*1...-1]
    ⟨ Remove repeated digits 10b ⟩
  end
  ◇

```

Macro referenced in [9d](#).

```

⟨ Remove trailing zeros 10d ⟩ ≡
  if @rep_i!=nil then
    if @d.length==@rep_i+1 && @d[@rep_i]==0 then
      @rep_i = nil;
      @d.pop;
    end
  end
  @d.pop while @d[@d.length-1]==0
  ◇

```

Macro referenced in [8a](#).

4 Writing to text

This method will return a text representation of a number. If the parameter is 0 (the default), the repetition will be delimited. If it is 1 or a greater number, the ellipsis notation is used with at least as many repetitions as the parameter indicates.

```

< RepDec members 11a > ≡
  def getS(nrep=0, opt=DEF_OPT)
    raise RepDecError, "Base_mismatch: _#{opt.digits.radix}_when_#{@radix}_was_expected." if
    opt.digits_defined? && @radix!=opt.digits.radix
    < Write special values 3a >
    s = "";
    s += '-' if @sign<0
    s += RepDec.group_digits(@ip.to_s(@radix),opt);
    s += opt.dec_sep if @d.length>0;
    for i in 0... @d.length
      break if nrep>0 && @rep_i==i;
      s += opt.begin_rep if i==@rep_i;
      s << opt.digits.digit_char(@d[i])
    end;
    if nrep>0 then
      if @rep_i!=nil then
        nrep += 1;
        nrep.times do
          for i in @rep_i...@d.length
            s << opt.digits.digit_char(@d[i])
          end
        end
        < Check for ambiguity 12b >
        s += opt.auto_rep;
      end
    else
      s += opt.end_rep if @rep_i!=nil;
    end
    return s;
  end
  ◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).
 Macro referenced in [2a](#).

```

< RepDec members 11b > ≡
  def to_s()
    getS
  end
  ◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).
 Macro referenced in [2a](#).

```

< Auxiliary functions 12a > ≡
  def RepDec.group_digits(digits, opt)
    if opt.grp_sep!=nil && opt.grp_sep!=' ' && opt.grp.length>0
      grouped = ''
      i = 0
      while digits.length>0
        l = opt.grp[i]
        l = digits.length if l>digits.length
        grouped = opt.grp_sep + grouped if grouped.length>0
        grouped = digits[-l,l] + grouped
        digits = digits [0, digits .length-l]
        i += 1 if i<opt.grp.length-1
      end
      grouped
    else
      digits
    end
  end
  ◇

```

Macro referenced in [1a](#).

Defines: `group_digits` Never used.

We must check if additional repetitions are needed for disambiguation.

```

< Check for ambiguity 12b > ≡
  check = RepDec.new;
  check.setS s+opt.auto_rep, opt;
  #print " s=",s,"\n"
  #print " self=",self.to_s,"\n"
  while check!=self
    for i in @rep_i...@d.length
      s << opt.digits .digit_char(@d[i])
    end
    check.setS s+opt.auto_rep, opt;
  end
  ◇

```

Macro referenced in [11a](#).

Some rational numbers have two possible decimal representations; one ending in $\bar{0}$ and another with $\bar{9}$. The first one is stored here as having no repetition (`@rep_i==nil`), i.e. that ending is implicit when no explicit repetition exists. This function normalizes a number avoiding the second representation.

Note than apart from the mentioned ambiguity in ending, the methods that modify decimals, namely `initialize`, `setZ`, `setS` and `setQ`, always produce unique representations in the sense that the repeating sequence is as short as possible and it is the first appearance after the decimal point which is marked.

```

< RepDec members 12c > ≡
  def normalize!(remove_trailing_zeros=true)
    if ip.is_a?(Integer)
      if @rep_i!=nil && @rep_i==@d.length-1 && @d[@rep_i]==(@radix-1) then
        @d.pop;
        @rep_i = nil;
        < Add one ulp to the decimal 13a >
      end
      < Additional normalizations 13b, ... >
    end
  end
  ◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).

Macro referenced in [2a](#).

Add one unit in the least significant digit to the decimal. Note that if the decimal has no decimal digits, a unit (least non-decimal digit) is added, rather than a unit to the least significant, non-zero digit.

```

< Add one ulp to the decimal 13a > ≡
  i = @d.length-1;
  carry = 1;
  while carry>0 && i>=0
    @d[i] += carry;
    carry = 0;
    if @d[i]>(@radix) then
      carry = 1;
      @d[i]=0;
      @d.pop if i==@d.length;
    end
    i -= 1;
  end
  @ip += carry;
  ◇

```

Macro referenced in [12c](#).

When normalizing a number we'll also avoid unnecessary repetitions.

```

< Additional normalizations 13b > ≡
  if @rep_i!=nil && @rep_i>=@d.length
    @rep_i = nil
  end
  ◇

```

Macro defined by [13bc](#).

Macro referenced in [12c](#).

```

< Additional normalizations 13c > ≡
  if @rep_i!=nil && @rep_i>=0
    unless @d[@rep_i..-1].find {|x| x!=0}
      @d = @d[0...@rep_i]
      @rep_i = nil
    end
  end
  if @rep_i==nil && remove_trailing_zeros
    while @d[@d.length-1]==0
      @d.pop
    end
  end
  ◇

```

Macro defined by [13bc](#).

Macro referenced in [12c](#).

Comparison of decimals: comparison is done by value, i.e., normalized values are compared, rather than representation.

```

< RepDec members 13d > ≡
  def copy()
    c = clone
    c.d = d.clone
    return c;
  end
  ◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).

Macro referenced in [2a](#).

```

⟨ RepDec members 14a ⟩ ≡
  def ==(c)
    a = copy;
    b = c.copy;
    a.normalize!
    b.normalize!
    return a.ip==b.ip && a.d==b.d && a.rep_i==b.rep_i
  end
  ◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).
 Macro referenced in [2a](#).

```

⟨ RepDec members 14b ⟩ ≡
  #def !=(c)
  # return !(self==c);
  #end
  ◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).
 Macro referenced in [2a](#).

5 Quotients

```

⟨ Default Options 14c ⟩ ≡
  @max_d = 2048
  ◇

```

Macro defined by [5bcd](#), [6a](#), [7a](#), [14c](#).
 Macro referenced in [4b](#).

5.1 Quotient to decimal

Convert a quotient to a repeating decimal. We'll use the classic long division algorithm, keeping all partial dividends to detect repetition.

```

⟨ RepDec members 15 ⟩ ≡
  def setQ(x,y, opt=DEF_OPT)
    @radix = opt.digits.radix if opt.digits_defined?
    xy_sign = x==0 ? 0 : x<0 ? -1 : +1;
    xy_sign = -xy_sign if y<0;
    @sign = xy_sign
    x = x.abs;
    y = y.abs;

    @d = [];
    @rep_i = nil;
    ⟨ Set special values 4a ⟩
    k = [];
    @ip = x.div(y) #x/y;
    x -= @ip*y;
    i = 0;
    ended = false;

    max_d = opt.max_d
    while x>0 && @rep_i==nil && (max_d<=0 || i<max_d)
      @rep_i = k.index(x)
      if @rep_i.nil? then
        k.push x;
        x *= @radix
        @d.push x.div(y) # x/y;
        x -= @d[i]*y;
        i += 1;
      end
    end
  end
  self
end
◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).

Macro referenced in [2a](#).

Defines: [setQ 17](#).

5.2 Decimal to quotient

Convert a repeating decimal to a quotient.

```

< RepDec members 16a > ≡
  def getQ(opt=DEF_OPT)
    raise RepDecError, "Base_mismatch: _#{opt.digits.radix}_when_#{@radix}_was_expected." if
    opt.digits_defined? && @radix!=opt.digits.radix
    < Get special values 3c >
    n = @d.length
    a = @ip
    b = a
    for i in 0.. n
      a*=@radix
      a+=@d[i];
      if @rep_i!=nil && i<@rep_i
        b *= @radix
        b += @d[i];
      end
    end
    end

    x = a
    x -= b if @rep_i!=nil

    y = @radix**n
    y -= @radix**@rep_i if @rep_i!=nil

    d = Nio.gcd(x,y)
    x /= d
    y /= d

    x = -x if @sign<0

    return x,y;
  end
  ◇

```

Macro defined by [2bc](#), [8ab](#), [11ab](#), [12c](#), [13d](#), [14ab](#), [15](#), [16a](#).

Macro referenced in [2a](#).

Defines: `getQ` [17](#).

We'll provisionally have this here; a function is needed to compute the greatest common divisor of two integers in order to simplify quotients.

```

< Nio functions 16b > ≡
  def gcd(a,b)
    while b!=0 do
      a,b = b, a.modulo(b)
    end
    return a.abs;
  end
  ◇

```

Macro referenced in [1a](#).

Defines: `gcd` Never used.

6 Tests

⟨ Tests 17 ⟩ ≡

```

def test_basic_repdec
  r = RepDec.new
  assert_equal "2.<3>", r.setQ(7,3).getS(0)
  assert_equal [7, 3], r.setQ(7,3).getQ

  assert_equal "1.<3>", r.setS("1.<3>").getS(0)
  assert_equal [4, 3], r.setS("1.<3>").getQ

  assert_equal "254.34212<678>", r.setS("254.34212<678>").getS(0)
  assert_equal [4234796411, 16650000], r.setS("254.34212<678>").getQ

  assert_equal "254.34212<678>", r.setS("254.34212678678...").getS(0)
  assert_equal [4234796411, 16650000], r.setS("254.34212678678...").getQ

  assert_equal "254.34212<678>", r.setS("254.34212678678678678...").getS(0)
  assert_equal [4234796411, 16650000], r.setS("254.34212678678678678...").getQ

  assert_equal "0.<3>", r.setS("0.3333333...").getS(0)
  assert_equal [1, 3], r.setS("0.3333333...").getQ

  assert_equal "-7.2<14>", r.setS("-7.2141414...").getS(0)
  assert_equal [-3571, 495], r.setS("-7.2141414...").getQ

  assert_equal "-7.21414...", r.setS("-7.2141414...").getS(1)
  assert_equal [-3571, 495], r.setS("-7.2141414...").getQ

  assert_equal "1.<234545>", r.setS("1.234545234545...").getS(0)
  assert_equal [1234544, 999999], r.setS("1.234545234545...").getQ

  assert_equal "1.234545234545...", r.setS("1.234545234545...").getS(1)
  assert_equal [1234544, 999999], r.setS("1.234545234545...").getQ

  assert_equal "1.23454523<45>", r.setS("1.23454523454545...").getS(0)
  assert_equal [678999879, 550000000], r.setS("1.23454523454545...").getQ

  assert_equal "1.23454523454545...", r.setS("1.23454523454545...").getS(1)
  assert_equal [678999879, 550000000], r.setS("1.23454523454545...").getQ

  assert_equal "0.<9>", r.setS("0.<9>").getS(0)
  assert_equal [1, 1], r.setS("0.<9>").getQ

  assert_equal "0.1<9>", r.setS("0.1999999...", RepDec::DEF_OPT.dup.set_digits(DigitsDef.base(16))).getS(0)
  assert_equal [1, 10], r.setS("0.1999999...", RepDec::DEF_OPT.dup.set_digits(DigitsDef.base(16))).getQ

  assert_equal "Infinity", r.setQ(10,0).getS(0)
  assert_equal [1, 0], r.setQ(10,0).getQ

  assert_equal "-Infinity", r.setQ(-10,0).getS(0)
  assert_equal [-1, 0], r.setQ(-10,0).getQ

  assert_equal "NaN", r.setQ(0,0).getS(0)
  assert_equal [0, 0], r.setQ(0,0).getQ

  assert_equal "NaN", r.setS("NaN").getS(0)
  assert_equal [0, 0], r.setS("NaN").getQ

  assert_equal "Infinity", r.setS("Infinity").getS(0)
  assert_equal [1, 0], r.setS("Infinity").getQ

  assert_equal "-Infinity", r.setS("-Infinity").getS(0)
  assert_equal [-1, 0], r.setS("-Infinity").getQ
end

```

7 Índices

7.1 Archivos

"lib/nio/repdec.rb" Defined by [1a](#).

"test/test_repdec.rb" Defined by [1b](#).

7.2 Fragmentos

[Add one ulp to the decimal 13a](#) } Referenced in [12c](#).
[Additional normalizations 13bc](#) } Referenced in [12c](#).
[Auxiliar classes 2e, 6b](#) } Referenced in [1a](#).
[Auxiliary functions 12a](#) } Referenced in [1a](#).
[Check for ambiguity 12b](#) } Referenced in [11a](#).
[Default Options 5bcd, 6a, 7a, 14c](#) } Referenced in [4b](#).
[Detect repetition 9d](#) } Referenced in [8a](#).
[Find additional repetitions 10c](#) } Referenced in [9d](#).
[Get special values 3c](#) } Referenced in [16a](#).
[License 1c](#) } Referenced in [1ab](#).
[Nio functions 16b](#) } Referenced in [1a](#).
[Read integral part 9a](#) } Referenced in [8b](#).
[Read sign from str 9b](#) } Referenced in [9a](#).
[Read special values 3b](#) } Referenced in [9a](#).
[Reduce multiple repetitions 10a](#) } Referenced in [9d](#).
[Remove repeated digits 10b](#) } Referenced in [9d, 10c](#).
[Remove trailing zeros 10d](#) } Referenced in [8a](#).
[RepDec class variables and constants 4bc](#) } Referenced in [2a](#).
[RepDec class 2a](#) } Referenced in [1a](#).
[RepDec members 2bc, 8ab, 11ab, 12c, 13d, 14ab, 15, 16a](#) } Referenced in [2a](#).
[RepDec protected members 2d](#) } Referenced in [2a](#).
[Reverse order of checking 9e](#) } Referenced in [9d](#).
[Set special values 4a](#) } Referenced in [15](#).
[Skip whitespace in str 9c](#) } Referenced in [8b, 9a](#).
[Tests setup ?](#) } Referenced in [1b](#).
[Tests 17](#) } Referenced in [1b](#).
[Write special values 3a](#) } Referenced in [11a](#).
[class Opt members 5a, 7bc](#) } Referenced in [4b](#).

7.3 Identificadores

gcd: [16b](#).

getQ: [16a, 17](#).

group_digits: [12a](#).

initialize: [2b, 4b, 6b](#).

setQ: [15, 17](#).

References

[Ross] "Elementary Analysis: The Theory of Calculus"
 Kenneth A. Ross
 1980
 Springer-Verlag New York Inc.
 ISBN: 0-387-90459-X