Package ‘Rmpfr’

January 21, 2024

Title R MPFR - Multiple Precision Floating-Point Reliable

Version 0.9-5

Date 2024-01-20

DateNote Previous CRAN version 0.9-4 on 2023-12-04

Type Package

Description Arithmetic (via S4 classes and methods) for
   arbitrary precision floating point numbers, including transcendental
   ("special") functions. To this end, the package interfaces to
   the 'LGPL' licensed 'MPFR' (Multiple Precision Floating-Point Reliable) Library
   which itself is based on the 'GMP' (GNU Multiple Precision) Library.

SystemRequirements gmp (>= 4.2.3), mpfr (>= 3.0.0), pdfcrop (part of
   TeXLive) is required to rebuild the vignettes.

SystemRequirementsNote 'MPFR' (MP Floating-Point Reliable Library,
   https://www.mpfr.org/) and 'GMP' (GNU Multiple Precision
   library, https://gmplib.org/), see >> README.md

Depends gmp (>= 0.6-1), R (>= 3.6.0)

Imports stats, utils, methods

Suggests MASS, Bessel, polynom, sfsmisc (>= 1.1-14)

SuggestsNote MASS, polynom, sfsmisc: only for vignette;

Enhances dfoptim, pracma, DPQ

EnhancesNote mentioned in Rd xrefs | used in example

URL https://rmpfr.r-forge.r-project.org/

BugReports https://r-forge.r-project.org/tracker/?group_id=386

License GPL (>= 2)

Encoding UTF-8

NeedsCompilation yes

Author Martin Maechler [aut, cre] (<https://orcid.org/0000-0002-8685-9910>),
   Richard M. Heiberger [ctb] (formatHex(), *Bin, *Dec),
   John C. Nash [ctb] (hjkMpfr(), origin of unirootR()),
   Hans W. Borchers [ctb] (optimizeR(*, "GoldenRatio"); origin of
   hjkMpfr())
Maintainer  Martin Maechler <maechler@stat.math.ethz.ch>
Repository  CRAN
Date/Publication  2024-01-21 12:22:45 UTC

R topics documented:

Rmpfr-package .......................................................... 3
array_or_vector-class .................................................. 6
asNumeric-methods ...................................................... 7
atomicVector-class ...................................................... 8
Bernoulli ................................................................. 9
Bessel_mpfr ............................................................... 10
bind-methods ............................................................. 11
chooseMpfr ............................................................... 12
factorialMpfr ............................................................ 14
formatHex ................................................................. 16
formatMpfr ............................................................... 19
frexpMpfr ................................................................. 21
gmp-conversions ......................................................... 23
hjkMpfr ................................................................. 24
igamma ................................................................. 27
integrateR .............................................................. 28
is.whole ................................................................. 31
log1mexp ............................................................... 32
matmult ................................................................. 34
Mnumber-class ........................................................... 35
mpfr ................................................................. 36
mpfr-class .............................................................. 39
mpfr-distr-etc .......................................................... 44
mpfr-special-functions .................................................. 46
mpfr-utils .............................................................. 47
mpfr.utils ............................................................. 53
mpfrArray .............................................................. 54
mpfrMatrix ............................................................. 55
mpfrMatrix-utils ....................................................... 58
optimizeR ............................................................... 60
pbetaI ................................................................. 62
pmax ................................................................. 65
qnormI ................................................................. 66
Rmpfr-workarounds ...................................................... 69
roundMpfr ............................................................... 70
sapplyMpfr ............................................................. 70
seqMpfr ................................................................. 72
str.mpfr .............................................................. 73
sumBinomMpfr .......................................................... 74
unirootR ............................................................... 76
**Description**

Rmpfr provides S4 classes and methods for arithmetic including transcendental ("special") functions for arbitrary precision floating point numbers, here often called "mpfr - numbers". To this end, it interfaces to the LGPL'ed MPFR (Multiple Precision Floating-Point Reliable) Library which itself is based on the GMP (GNU Multiple Precision) Library.

**Details**

Package: Rmpfr
Title: R MPFR - Multiple Precision Floating-Point Reliable
Version: 0.9-5
Date: 2024-01-20
DateNote: Previous CRAN version 0.9-4 on 2023-12-04
Type: Package
Authors@R: c(person("Martin","Maechler", role = c("aut","cre"), email = "maechler@stat.math.ethz.ch", comment = ... role = "ctb", email="hwborchers@googlemail.com", comment = "optimizeR(*, "GoldenRatio"); origin of hjkMpfr()") )
Description: Arithmetic (via S4 classes and methods) for arbitrary precision floating point numbers, including ... (Multiple Precision Floating-Point Reliable) Library.
SystemRequirements: gmp (>= 4.2.3), mpfr (>= 3.0.0), pdfcrop (part of TexLive) is required to rebuild the vignettes.
SystemRequirementsNote: ‘MPFR’ (MP Floating-Point Reliable Library, https://www.mpfr.org/) and ‘GMP’ (GNU Multiple Precision Library, https://gmplib.org/), see » README.md
Depends: gmp (>= 0.6-1), R (>= 3.6.0)
Imports: stats, utils, methods
Suggests: MASS, Bessel, polynom, sfsmisc (>= 1.1-14)
SuggestsNote: MASS, polynom, sfsmisc: only for vignette;
Enhances: dfoptim, pracma, DPQ
EnhancesNote: mentioned in Rd xrefs | used in example
URL: https://rmpfr.r-forge.r-project.org/
BugReports: https://r-forge.r-project.org/tracker/?group_id=386
License: GPL (>= 2)
Encoding: UTF-8
Author: Martin Maechler [aut, cre] (<https://orcid.org/0000-0002-8685-9910>), Richard M. Heiberger [ctb]
Maintainer: Martin Maechler <maechler@stat.math.ethz.ch>

Index of help topics:

- .bigq2mpfr Conversion Utilities gmp <-> Rmpfr
- Bernoulli Bernoulli Numbers in Arbitrary Precision
- Bessel_mpfr Bessel functions of Integer Order in multiple precisions
- Mnumber-class Class "Mnumber" and "mNumber" of "mpfr" and regular numbers and arrays from them
- Rmpfr-package R MPFR - Multiple Precision Floating-Point Reliable
array_or_vector-class
asNumeric-methods
atomicVector-class
c.mpfr
cbind
chooseMpfr
determinant.mpfrMatrix
factorialMpfr
formatHex
formatMpfr
frexpMpfr
getPrec
hjkMpfr
igamma
integrateR
is.whole.mpfr
log1mexp
matmult
mpfr
mpfr-class
mpfrArray
mpfrMatrix-class
optimizeR
outer
pbetaI
pmax
pnorm
qnormI
roundMpfr
sapplyMpfr
seqMpfr
str.mpfr
sumBinomMpfr
unirrootR
zeta

Reliable
Auxiliary Class "array_or_vector"
Methods for 'asNumeric(<mpfr>)'
Virtual Class "atomicVector" of Atomic Vectors
MPFR Number Utilities
"mpfr' '...' - Methods for Functions cbind(), rbind()
Binomial Coefficients and Pochhammer Symbol aka Rising Factorial
Functions for mpfrMatrix Objects
Factorial 'n!' in Arbitrary Precision
Flexibly Format Numbers in Binary, Hex and Decimal Format
Formatting MPFR (multiprecision) Numbers
Base-2 Representation and Multiplication of Mpfr Numbers
Rmpfr - Utilities for Precision Setting, Printing, etc
Hooke-Jeeves Derivative-Free Minimization R (working for MPFR)
Incomplete Gamma Function
One-Dimensional Numerical Integration - in pure R
Whole ("Integer") Numbers
Compute f(a) = log(1 +/- exp(-a)) Numerically Optimally
(MPFR) Matrix (Vector) Multiplication
Create "mpfr" Numbers (Objects)
Class "mpfr" of Multiple Precision Floating Point Numbers
Construct "mpfrArray" almost as by 'array()' Classes "mpfrMatrix" and "mpfrArray"
High Precision One-Dimensional Optimization
Base Functions etc, as an Rmpfr version
Accurate Incomplete Beta / Beta Probabilities For Integer Shapes
Parallel Maxima and Minima
Distribution Functions with MPFR Arithmetic
Gaussian / Normal Quantiles 'qnorm()' via Inversion
Rounding to Binary bits, "mpfr-internally"
Apply a Function over a "mpfr" Vector
"mpfr" Sequence Generation
Compactly Show STRucture of Rmpfr Number Object
(Alternating) Binomial Sums via Rmpfr
One Dimensional Root (Zero) Finding - in pure R
Special Mathematical Functions (MPFR)
Further information is available in the following vignettes:

- Maechler_useR_2011-abstr
- Rmpfr-pkg
- log1mexp-note

The following (help pages) index does not really mention that we provide many methods for mathematical functions, including `gamma`, `digamma`, etc, namely, all of R's (S4) Math group (with the only exception of `trigamma`), see the list in the examples. Additionally also `pnorm`, the “error function”, and more, see the list in `zeta`, and further note the first vignette (below).

**Partial index:**

- mpfr: Create "mpfr" Numbers (Objects)
- mpfrArray: Construct "mpfrArray" almost as by `array()`
- mpfr-class: Class "mpfr" of Multiple Precision Floating Point Numbers
- mpfrMatrix-class: Classes "mpfrMatrix" and "mpfrArray"
- Bernoulli: Bernoulli Numbers in Arbitrary Precision
- Bessel_mpfr: Bessel functions of Integer Order in multiple precisions
- c.mpfr: MPFR Number Utilities
- cbind: "mpfr"... - Methods for Functions cbind(), rbind()
- chooseMpfr: Binomial Coefficients and Pochhammer Symbol aka Rising Factorial
- factorialMpfr: Factorial 'n!' in Arbitrary Precision
- formatMpfr: Formatting MPFR (multiprecision) Numbers
- getPrec: Rmpfr - Utilities for Precision Setting, Printing, etc
- roundMpfr: Rounding to Binary bits, "mpfr-internally"
- seqMpfr: "mpfr" Sequence Generation
- sumBinomMpfr: (Alternating) Binomial Sums via Rmpfr
- zeta: Special Mathematical Functions (MPFR)
- integrateR: One-Dimensional Numerical Integration - in pure R
- unirootR: One Dimensional Root (Zero) Finding - in pure R
- optimizeR: High Precisione One-Dimensional Optimization
- hjkMpfr: Hooke-Jeeves Derivative-Free Minimization R (working for MPFR)

Further information is available in the following vignettes:

- Rmpfr-pkg: Arbitrarily Accurate Computation with R: The 'Rmpfr' package (source, pdf)
- log1mexp-note: Accurately Computing log(1 - exp(.)) – Assessed by Rmpfr (source, pdf)
Author(s)

Martin Maechler

References

MPFR (MP Floating-Point Reliable Library), https://www.mpfr.org/
GMP (GNU Multiple Precision library), https://gmplib.org/
and see the vignettes mentioned above.

See Also

The R package gmp for big integer gmp and rational numbers (bigrational) on which Rmpfr
depends.

Examples

```r
## Using "mpfr" numbers instead of regular numbers...
n1.25 <- mpfr(5, precBits = 256)/4
n1.25
## and then "everything" just works with the desired chosen precision:hig
n1.25 ^ c(1:7, 20, 30) ## fully precise; compare with
print(1.25 ^ 30, digits=19)
exp(n1.25)
## Show all math functions which work with "MPFR" numbers (1 exception: trigamma)
getGroupMembers("Math")
## We provide *many* arithmetic, special function, and other methods:
showMethods(classes = "mpfr")
showMethods(classes = "mpfrArray")
```

array_or_vector-class  Auxiliary Class "array_or_vector"

Description

"array_or_vector" is the class union of c("array", "matrix", "vector") and exists for its use
in signatures of method definitions.

Details

Using "array_or_vector" instead of just "vector" in a signature makes an important difference:
E.g., if we had setMethod(crossprod, c(x="mpfr", y="vector"), function(x,y) CPR(x,y)),
a call crossprod(x, matrix(1:6, 2,3)) would extend into a call of CPR(x, as(y, "vector"))
such that CPR("")'s second argument would simply be a vector instead of the desired 2 \times 3 matrix.
Objects from the Class

A virtual Class: No objects may be created from it.

Examples

showClass("array_or_vector")

asNumeric-methods  Methods for asNumeric(<mpfr>)

Description

Methods for function `asNumeric` (in package `gmp`).

Usage

## S4 method for signature 'mpfrArray'
asNumeric(x)

Arguments

x  a “number-like” object, here, a `mpfr` or typically `mpfrArray` one.

Value

an R object of type (typeof) "numeric", a `matrix` or `array` if x had non-NULL dimension `dim()`.

Methods

signature(x = "mpfrArray") this method also dispatches for `mpfrMatrix` and returns a numeric array.

signature(x = "mpfr") for non-array/matrix, `asNumeric(x)` is basically the same as `as.numeric(x)`.

Author(s)

Martin Maechler

See Also

our lower level (non-generic) `toNum()`. Further, `asNumeric` (package `gmp`), standard R’s `as.numeric()`.
Examples

```r
x <- (0:7)/8 # (exact)
X <- mpfr(x, 99)
stopifnot(identical(asNumeric(x), x),
           identical(asNumeric(X), x))

m <- matrix(1:6, 3,2)
(M <- mpfr(m, 99) / 5) ##-> "mpfrMatrix"
asNumeric(M) # numeric matrix
stopifnot(all.equal(asNumeric(M), m/5),
           identical(asNumeric(m), m))# remains matrix
```

---

atomicVector-class  Virtual Class "atomicVector" of Atomic Vectors

Description

The class "atomicVector" is a virtual class containing all atomic vector classes of base R, as also implicitly defined via `is.atomic`.

Objects from the Class

A virtual Class: No objects may be created from it.

Methods

In the Matrix package, the "atomicVector" is used in signatures where typically "old-style" "matrix" objects can be used and can be substituted by simple vectors.

Extends

The atomic classes "logical", "integer", "double", "numeric", "complex", "raw" and "character" are extended directly. Note that "numeric" already contains "integer" and "double", but we want all of them to be direct subclasses of "atomicVector".

Author(s)

Martin Maechler

See Also

`is.atomic`, `integer`, `numeric`, `complex`, etc.

Examples

```
showClass("atomicVector")
```
Bernoulli Numbers in Arbitrary Precision

Description

Computes the Bernoulli numbers in the desired (binary) precision. The computation happens via the \texttt{zeta} function and the formula

\[ B_k = -k \zeta(1 - k), \]

and hence the only non-zero odd Bernoulli number is \( B_1 = +1/2 \). (Another tradition defines it, equally sensibly, as \(-1/2\).)

Usage

\texttt{Bernoulli(k, precBits = 128)}

Arguments

- \texttt{k} non-negative integer vector
- \texttt{precBits} the precision in \textit{bits} desired.

Value

an \texttt{mpfr} class vector of the same length as \texttt{k}, with i-th component the \texttt{k[i]}-th Bernoulli number.

Author(s)

Martin Maechler

References

- \url{https://en.wikipedia.org/wiki/Bernoulli_number}

See Also

\texttt{zeta} is used to compute them.

The next version of package \texttt{gmp} is to contain \texttt{BernoulliQ()}, providing exact Bernoulli numbers as big rationals (class “bigq”).

Examples

\begin{verbatim}
Bernoulli(0:10)
pplot(as.numeric(Bernoulli(0:15)), type = "h")

curve(-x*zeta(1-x), -.2, 15.03, n=300,
    main = expression(-x %.% zeta(1-x)))
legend("top", paste(c("even","odd "), "Bernoulli numbers"),
    pch=c(1,3), col=2, pt.cex=2, inset=1/64)
\end{verbatim}
### Description

Bessel functions of integer orders, provided via arbitrary precision algorithms from the MPFR library. Note that the computation can be very slow when $n$ and $x$ are large (and of similar magnitude).

#### Usage

- `Ai(x)`
- `j0(x)`
- `j1(x)`
- `jn(n, x, rnd.mode = c("N","D","U","Z","A"))`
- `y0(x)`
- `y1(x)`
- `yn(n, x, rnd.mode = c("N","D","U","Z","A"))`

#### Arguments

- `x` : a numeric or mpfr vector.
- `n` : non-negative integer (vector).
- `rnd.mode` : a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see mpfr.

#### Value

Computes multiple precision versions of the Bessel functions of integer order, $J_n(x)$ and $Y_n(x)$, and—when using MPFR library 3.0.0 or newer—also of the Airy function $Ai(x)$. Note that currently $Ai(x)$ is very slow to compute for large $x$.

#### See Also

besselJ, and besselY compute the same bessel functions but for arbitrary real order and only precision of a bit more than ten digits.
Examples

```r
x <- (0:100)/8  # (have exact binary representation)
stopifnot(exprs = {
  all.equal(besselY(x, 0), bY0 <- y0(x))
  all.equal(besselJ(x, 1), bJ1 <- j1(x))
  all.equal(yn(0,x), bY0)
  all.equal(jn(1,x), bJ1)
})

mpfrVersion() # now typically 4.1.0
if(mpfrVersion() >= "3.0.0") { ## Ai() not available previously
  print( aix <- Ai(x) )
  plot(x, aix, log="y", type="l", col=2)
  stopifnot(
    all.equal(Ai (0) , 1/(3^(2/3) * gamma(2/3)))
    , # see https://dlmf.nist.gov/9.2.ii
    all.equal(Ai(100), mpfr("2.6344821520881844895505525695264981561e-291"), tol=1e-37)
  )
  two3rd <- 2/mpfr(3, 144)
p
  if(Rmpfr:::doExtras()) withAutoprint({
    system.time(ai1k <- Ai(1000)) # 1.4 sec (on 2017 lynne)
    stopifnot(all.equal(print(log10(ai1k)),
      -9157.031193409585185582, tol=2e-16)) # seen 8.8..e-17 | 1.1..e-16
  })
} # ver >= 3.0
```

Description

`cbind` and `rbind` methods for signature ... (see `dotsMethods` are provided for class `Mnumber`, i.e., for binding numeric vectors and class "mpfr" vectors and matrices ("mpfrMatrix") together.

Usage

```r
cbind(..., deparse.level = 1)
rbind(..., deparse.level = 1)
```

Arguments

- `...` matrix-/vector-like `R` objects to be bound together, see the `base` documentation, `cbind`.
- `deparse.level` integer determining under which circumstances column and row names are built from the actual arguments’ ‘expression’, see `cbind`.
### chooseMpfr

**Value**

typically a ‘matrix-like’ object, here typically of class "mpfrMatrix".

**Methods**

... = "Mnumber" is used to (c|r)bind multiprecision “numbers” (inheriting from class "mpfr") together, maybe combined with simple numeric vectors.

... = "ANY" reverts to cbind and rbind from package base.

**Author(s)**

Martin Maechler

**See Also**

cbind2, cbind, Documentation in base R’s methods package

**Examples**

cbind(1, mpfr(6:3, 70)/7, 3:0)

---

### chooseMpfr

**Binomial Coefficients and Pochhammer Symbol aka Rising Factorial**

**Description**

Compute binomial coefficients, chooseMpfr(a, n) being mathematically the same as choose(a, n), but using high precision (MPFR) arithmetic.

chooseMpfr.all(n) means the vector choose(n, 1:n), using enough bits for exact computation via MPFR. However, chooseMpfr.all() is now deprecated in favor of chooseZ from package gmp, as that is now vectorized.

pochMpfr() computes the Pochhammer symbol or “rising factorial”, also called the “Pochhammer function”, “Pochhammer polynomial”, “ascending factorial”, “rising sequential product” or “upper factorial”,

\[ x^{(n)} = x(x + 1)(x + 2) \cdots (x + n - 1) = \frac{(x + n - 1)!}{(x - 1)!} = \frac{\Gamma(x + n)}{\Gamma(x)} \].

**Usage**

chooseMpfr (a, n, rnd.mode = c("N","D","U","Z","A"))
chooseMpfr.all(n, precBits=NULL, k0=1, alternating=FALSE)
pochMpfr(a, n, rnd.mode = c("N","D","U","Z","A"))
chooseMpfr

Arguments

a  a numeric or mpfr vector.
n  an integer vector; if not of length one, n and a are recycled to the same length.
rnd.mode  a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see mpfr.
precBits  integer or NULL for increasing the default precision of the result.
k0  integer scalar
alternating  logical, for chooseMpfr.all(), indicating if alternating sign coefficients should be returned, see below.

Value

For

chooseMpfr(), pochMpfr(): an mpfr vector of length \text{max}(|a|,|n|);
chooseMpfr.all(n, k0): a mpfr vector of length \text{n-k0+1}, of binomial coefficients \( C_{n,m} \) or, if alternating is true, \((-1)^m \cdot C_{n,m} \) for \( m \in k0:n \).

Note

Currently this works via a (C level) for(i in 1:n)-loop which really slow for large \( n \), say \( 10^6 \), with computational cost \( O(n^2) \). In such cases, if you need high precision \( \text{choose(a,n)} \) (or Pochhammer(a,n)) for large \( n \), preferably work with the corresponding \( \text{factorial(mpfr(..))} \), or \( \text{gamma(mpfr(..))} \) terms.

See Also

\texttt{choose(n,m)} (\texttt{base R}) computes the binomial coefficient \( C_{n,m} \) which can also be expressed via Pochhammer symbol as \( C_{n,m} = (n - m + 1)^{(m)}/m! \).
\texttt{chooseZ} from package gmp; for now, \texttt{factorialMpfr}.

For (alternating) binomial sums, directly use \texttt{sumBinomMpfr}, as that is potentially more efficient.

Examples

pochMpfr(100, 4) == 100*101*102*103 # TRUE
a <- 100:110
pochMpfr(a, 10) # exact (but too high precision)
x <- mpfr(a, 70)# should be enough
(px <- pochMpfr(x, 10)) # the same as above (needing only 70 bits)
stopifnot(pochMpfr(a, 10) == px,
px[1] == prod(mpfr(100:109, 100)))# used to fail

(c1 <- chooseMpfr(1000:997, 60)) # -> automatic "correct" precision
stopifnot(all.equal(c1, choose(1000:997, 60), tolerance=1e-12))

## --- Experimenting & Checking
if(!Rmpfr:::doExtras()) { ## speed up: smaller set
  n. <- n.set[-(1:10)]
  n.set <- c(1:10, n.[c(TRUE, diff(n.) > 1)])
}
C1 <- C2 <- numeric(length(n.set))
for(i.n in seq_along(n.set)) {
  cat(n <- n.set[i.n], ":")
  C1[i.n] <- system.time(c.c <- chooseMpfr.all(n))[1]
  C2[i.n] <- system.time(c.2 <- chooseMpfr(n, 1:n))[1]
  stopifnot(is.whole(c.c), c.c == c.2,
            if(n > 60) TRUE else all.equal(c.c, choose(n, 1:n), tolerance = 1e-15))
  cat(" [OK]\n")
}
matplot(n.set, cbind(C1,C2), type="b", log="xy",
       xlab = "n", ylab = "system.time(.) [s]"
legend("topleft", c("chooseMpfr.all(n)", "chooseMpfr(n, 1:n)"),
       pch=as.character(1:2), col=1:2, lty=1:2, bty="n")
## Currently, chooseMpfr.all() is faster only for large n (>= 300)
## That would change if we used C-code for the *.all() version

## If you want to measure more:
measureMore <- TRUE
measureMore <- FALSE
if(measureMore) { ## takes ~ 2 minutes (on "lynne", Intel i7-7700T, ~2019)
  n.s <- 2^((5:20)
  r <- lapply(n.s, function(n) {
    N <- ceiling(10000/n)
    cat(sprintf("n=%9g => N=%d: ",n,N))
    ct <- system.time(C <- replicate(N, chooseMpfr(n, n/2)))
    cat("[OK]\n")
    list(C=C, ct=ct/N)
  })
  print(ct.n <- t(sapply(r, mean))
  hasSfS <- requireNamespace("sfsmisc")
  plot(ct.n[,."user.self"] ~ n.s, xlab="n", ylab="system.time(.) [s]",
       main = "CPU Time for chooseMpfr(n, n/2)",
       log = "xy", type = "b", axes = !hasSfS)
  if(hasSfS) for(side in 1:2) sfsmisc::eaxis(side)
  summary(fm <- lm(log(ct.n[,"user.self"]) ~ log(n.s), subset = n.s >= 10^4))
  ## --> slope = 1.980876776577515387760208355741255374519464715284957171019990049950029018378557181915573287034690822211475344830731 = O(n^2)

  nn <- 2^seq(11,21, by=1/16) ; Lcol <- adjustcolor(2, 1/2)
  bet <- coef(fm)
  lines(nn, exp(predict(fm, list(n.s = nn))), col=Lcol, lwd=3)
  text(100000,1, substitute(AA %*% n^EE,
            list(AA = signif(exp(bet[1]),3),
                 EE = signif( bet[2], 3)), col = 2)

) # measure more

factorialMpfr Factorial 'n!' in Arbitrary Precision
factorialMpfr

Description

Efficiently compute \( n! \) in arbitrary precision, using the MPFR-internal implementation. This is mathematically (but not numerically) the same as \( \Gamma(n + 1) \).

factorialZ (package gmp) should typically be used instead of factorialMpfr() nowadays. Hence, factorialMpfr now is somewhat deprecated.

Usage

factorialMpfr(n, precBits = max(2, ceiling(lgamma(n+1)/log(2))),
            rnd.mode = c("N", "D", "U", "Z", "A"))

Arguments

\n
- **n**: non-negative integer (vector).
- **precBits**: desired precision in bits ("binary digits"); the default sets the precision high enough for the result to be exact.
- **rnd.mode**: a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see mpfr.

Value

a number of (S4) class mpfr.

See Also

factorial and gamma in base R.

factorialZ (package gmp), to replace factorialMpfr, see above.

chooseMpfr() and pochMpfr() (on the same page).

Examples

factorialMpfr(200)

n <- 1000:1010
f1000 <- factorialMpfr(n)
stopifnot(1e-15 > abs(as.numeric(1 - lfactorial(n)/log(f1000))))

## Note that---astonishingly---measurements show only
## *small* efficiency gain of ~ 10% : over using the previous "technique"

system.time(replicate(8, f1e4 <- factorialMpfr(10000)))

system.time(replicate(8, f.1e4 <- factorial(mpfr(10000,
prec=1+lfactorial(10000)/log(2)))))
Description

Show numbers in binary, hex and decimal format. The resulting character-like objects can be back-transformed to "mpfr" numbers via `mpfr()`.

Usage

formatHex(x, precBits = min(getPrec(x)), style = "+", expAlign = TRUE)

formatBin(x, precBits = min(getPrec(x)), scientific = TRUE,
          left.pad = ",", right.pad = left.pad, style = "+", expAlign = TRUE)

formatDec(x, precBits = min(getPrec(x)), digits = decdigits,
          nsmall = NULL, scientific = FALSE, style = "+",
          decimalPointAlign = TRUE, ...)

Arguments

x  a numeric or mpfr R object.
precBits  integer, the number of bits of precision, typically derived from x, see `getPrec`. Numeric, i.e., double precision numbers have 53 bits. For more detail, see mpfr.
style  a single character, to be used in sprintf's format (fmt), immediately after the " sets a sign in the output, i.e., "+" or "-", where as style = " " may seem more standard.
expAlign  logical indicating if for scientific ("exponential") representations the exponents should be aligned to the same width, i.e., zero-padded to the same number of digits.
scientific  logical indicating that formatBin should display the binary representation in scientific notation (mpfr(3, 5) is displayed as +0b1.1000p+1). When FALSE, formatBin will display the binary representation in regular format shifted to align binary points (mpfr(3, 5) is displayed +0b11.000).
...  additional optional arguments.

formatHex, formatBin: precBits is the only ... argument acted on. Other ... arguments are ignored.

formatDec: precBits is acted on. Any argument accepted by format (except nsmall) is acted on. Other ... arguments are ignored.
left.pad, right.pad
characters (one-character strings) that will be used for left- and right-padding of
the formatted string when `scientific=FALSE`. Do not change these unless for
display-only purpose !!

nsmall
only used when `scientific` is false, then passed to `format()`. If NULL, the
default is computed from the range of the non-zero values of `x`.

digits
integer; the number of decimal digits displayed is the larger of this argument
and the internally generated value that is a function of `precBits`. This is related
to but different than `digits` in `format`.

decimalPointAlign
logical indicating if padding should be used to ensure that the resulting strings
align on the decimal point (".").

Details
For the hexadecimal representation, when the precision is not larger than double precision, `sprintf()`
is used directly, otherwise `formatMpfr()` is used and post processed. For the binary representation,
the hexadecimal value is calculated and then edited by substitution of the binary representation of
the hex characters coded in the `HextoBin` vector. For binary with `scientific=FALSE`, the result of
the `scientific=TRUE` version is edited to align binary points. For the decimal representation, the
hexadecimal value is calculated with the specified precision and then sent to the `format` function
for `scientific=FALSE` or to the `sprintf` function for `scientific=TRUE`.

Value
a character vector (or matrix) like `x`, say `r`, containing the formatted represention of `x`, with a `class`
(unless `left.pad` or `right.pad` were not "."). In that case, `formatHex()` and `formatBin()` return
class "Ncharacter"; for that, `mpfr(.)` has a method and will basically return `x`, i.e., work as inverse
function.

Since `Rmpfr` version 0.6-2, the S3 class "Ncharacter" extends "character". `(class(.))` is
now of length two and `class(.)[2]` is "character".). There are simple `[]` and `print` methods;
modifying or setting `dim` works as well.

Author(s)
Richard M. Heiberger <rmh@temple.edu>, with minor tweaking by Martin M.

References
R FAQ 7.31: Why doesn’t R think these numbers are equal? `system.file(".././doc/FAQ")`

See Also
`mpfr`, `sprintf`
Examples

```r
FourBits <- mpfr(matrix(0:31, 8, 4, dimnames = list(0:7, c(0,8,16,24))),
    precBits=4) ## 4 significant bits

FourBits

formatHex(FourBits)
formatBin(FourBits, style = " ")
formatBin(FourBits, scientific=FALSE)
formatDec(FourBits)

## as "Ncharacter" 'inherits from' "character", this now works too :
data.frame(Dec = c( formatDec(FourBits) ), formatHex(FourBits),
    Bin = formatBin(FourBits, style = " "))

FBB <- formatBin(FourBits) ; clB <- class(FBB)
(nFBB <- mpfr(FBB))
stopifnot(class(FBB)[1] == "Ncharacter",
    all.equal(nFBB, FourBits, tol=0))

FBH <- formatHex(FourBits) ; clH <- class(FBH)
(nFBH <- mpfr(FBH))
stopifnot(class(FBH)[1] == "Ncharacter",
    all.equal(nFBH, FourBits, tol=0))

## Compare the different "formattings" (details will change, i.e. improve!)% % FIXME
M <- mpfr(c(-Inf, -1.25, 1/(-Inf), NA, 0, .5, 1:2, Inf), 3)
data.frame(fH = formatHex(M), f16 = format(M, base=16),
    fB = formatBin(M), f2 = format(M, base=2),
    fD = formatDec(M), f10 = format(M), # base = 10 is default
    fSci= format(M, scientific=TRUE),
    fFix= format(M, scientific=FALSE))

## Other methods ("[", t()) also work :
stopifnot(dim(F1 <- FBB[, 1, drop=FALSE]) == c(8,1), identical(class( F1), clB),
    dim(t(F1)) == c(1,8), identical(class(t(F1)),clB),
    is.null(dim(F.2 <- FBB[,2])), identical(class( F.2), clB),
    dim(F22 <- FBH[1:2, 3:4]) == c(2,2), identical(class(F22), clH),
    identical(class(FBH[2,3]), clH), is.null(dim(FBH[2,3])),
    identical(FBH[2,3:4], F22[2,]),
    identical(FBH[2,3], unname(FBH[,3][2])), TRUE)

TenFrac <- matrix((1:10)/10, dimnames=list(1:10, expression(1/x)))
TenFrac9 <- mpfr(TenFrac, precBits=9) ## 9 significant bits

formatHex(TenFrac)
formatBin(TenFrac)
formatBin(TenFrac, scientific=FALSE)
formatDec(TenFrac)
formatDec(TenFrac9, precBits=10)
```
**formatMpfr**

*Formatting MPFR (multiprecision) Numbers*

**Description**

Flexible formatting of “multiprecision numbers”, i.e., objects of class `mpfr`. `formatMpfr()` is also the `mpfr` method of the generic `format` function.

The `formatN()` methods for `mpfr` numbers renders them differently than their double precision equivalents, by appending ".M".

Function `.mpfr2str()` is the low level work horse for `formatMpfr()` and hence all `print()`ing of "mpfr" objects.

**Usage**

```r
formatMpfr(x, digits = NULL, trim = FALSE, scientific = NA,
          maybe.full = (!is.null(digits) && is.na(scientific)) || isFALSE(scientific),
          base = 10, showNeg0 = TRUE, max.digits = Inf,
          big.mark = "", big.interval = 3L,
          small.mark = "", small.interval = 5L,
          decimal.mark = ".",
          exponent.char = if(base <= 14) "e" else if(base <= 36) "E" else "|e",
          exponent.plus = TRUE,
          zero.print = NULL, drop0trailing = FALSE, ...)

## S3 method for class 'mpfr'
formatN(x, drop0trailing = TRUE, ...)

.mpfr2str(x, digits = NULL, maybe.full = !is.null(digits), base = 10L)
```

**Arguments**

- **x** an MPFR number (vector or array).
- **digits** how many significant digits (in the base chosen!) are to be used in the result. The default, NULL, uses enough digits to represent the full precision, often one or two digits more than “you” would expect. For bases 2,4,8,16, or 32, MPFR requires digits at least 2. For such bases, digits = 1 is changed into 2, with a message.
- **trim** logical; if FALSE, numbers are right-justified to a common width: if TRUE the leading blanks for justification are suppressed.
- **scientific** either a logical specifying whether MPFR numbers should be encoded in scientific format (“exponential representation”), or an integer penalty (see `options("scipen")`). Missing values correspond to the current default penalty.
- **maybe.full** logical, passed to `.mpfr2str()`.
formatMpfr

base
an integer in 2, 3, ..., 62; the base ("basis") in which the numbers should be represented. Apart from the default base 10, binary (base = 2) or hexadecimal (base = 16) are particularly interesting.

showNeg0
logical indicating if "negative" zeros should be shown with a "-". The default, TRUE is intentionally different from format(<numeric>).

exponent.char
the "exponent" character to be used in scientific notation. The default takes into account that for base \( \leq 15 \), "e" is part of the (mantissa) digits and the same is true for "E" when \( \geq 37 \).

exponent.plus
logical indicating if "+" should be for positive exponents in exponential (aka "scientific") representation. This used to be hardcoded to FALSE; the new default is compatible to \( \text{R}'s \) format()ing of numbers and helps to note visually when exponents are in use.

max.digits
a (large) positive number to limit the number of (mantissa) digits, notably when digits is NULL (as by default). Otherwise, a numeric digits is preferred to setting max.digits (which should not be smaller than digits).

big.mark, big.interval, small.mark, small.interval, decimal.mark, zero.print, drop0trailing
used for prettying decimal sequences, these are passed to prettyNum and that help page explains the details.

... further arguments passed to or from other methods.

Value

a character vector or array, say cx, of the same length as x. Since Rmpfr version 0.5-3 (2013-09), if x is an mpfrArray, then cx is a character array with the same dim and dimnames as x.

Note

Currently, formatMpfr(x, scientific = FALSE) does not work correctly, e.g., for x <- Const("pi", 128) * 2^c(-200, 200), i.e., it uses the scientific / exponential-style format. This is considered bogus and hopefully will change.

Author(s)

Martin Maechler

References

The MPFR manual’s description of ‘mpfr_get_str()’ which is the C-internal workhorse for .mpfr2str() (on which formatMpfr() builds).
See Also

`mpfr` for creation and the `mpfr` class description with its many methods. The `format` generic, and the `prettyNum` utility on which `formatMpfr` is based as well. The S3 generic function `formatN` from package `gmp`.

`.mpfr_formatinfo(x)` provides the (cheap) non-string parts of `.mpfr2str(x)`; the (base 2) exp exponents are also available via `.mpfr2exp(x)`.

Examples

```r
## Printing of MPFR numbers uses formatMpfr() internally.
## Note how each components uses the "necessary" number of digits:
(x3 <- c(Const("pi", 168), mpfr(pi, 140), 3.14))
format(x3[3], 15)
format(x3[3], 15, drop0 = TRUE)# "3.14" .. dropping the trailing zeros
x3[4] <- 2^30
x3[4] # automatically drops trailing zeros
format(x3[1], dig = 41, small.mark = "\\Var") # (41 - 1 = ) 40 digits after "."

rbind(formatN(x3, digits = 15),
      formatN(as.numeric(x3), digits = 15))

(Zero <- mpfr(c(0,1/-Inf), 20)) # 0 and "-0"
xx <- c(Zero, 1:2, Const("pi", 120), -100*pi, -.00987)
format(xx, digits = 2)
format(xx, digits = 1, showNeg0 = FALSE)# "-0" no longer shown

## Output in other bases:
formatMpfr(mpfr(10^6, 40), base=32, drop0trailing=TRUE)
## "ugi0"
mpfr("ugi0", base=32) #-> 000000

## This now works: The large number shows "as" large integer:
x <- Const("pi", 128) * 2^c(-200,200)
formatMpfr(x, scientific = FALSE) # was 1.955...e-60 5.048...e+60

i32 <- mpfr(1:32, precBits = 64)
format(i32, base= 2, drop0trailing=TRUE)
format(i32, base= 16, drop0trailing=TRUE)
format(1/i32, base= 2, drop0trailing=TRUE)# using scientific notation for [17..32]
format(1/i32, base= 32)
format(1/i32, base= 62, drop0trailing=TRUE)
format(mpfr(2, 64)^-(1:16), base=16, drop0trailing=TRUE)
```
Description

MPFR - versions of the C99 (and POSIX) standard C (and C++) mathlib functions `frexp()` and `ldexp()`.

`frexpMpfr(x)` computes base-2 exponent e and “mantissa”, or fraction r, such that \( x = r \times 2^e \), where \( r \in [0.5, 1) \) (unless when \( x \) is in \( c(0, -\infty, \infty, \text{NaN}) \) where \( r = x \) and \( e \) is \( 0 \)), and \( e \) is integer valued.

`ldexpMpfr(f, E)` is the inverse of `frexpMpfr()`: Given fraction or mantissa \( f \) and integer exponent \( E \), it returns \( x = f \times 2^E \). Viewed differently, it’s the fastest way to multiply or divide MPFR numbers with \( 2^E \).

Usage

```r
frexpMpfr(x, rnd.mode = c("N", "D", "U", "Z", "A"))
ldexpMpfr(f, E, rnd.mode = c("N", "D", "U", "Z", "A"))
```

Arguments

- **x** numeric (coerced to `double`) vector.
- **f** numeric fraction (vector), in \([0.5, 1)\).
- **E** integer valued, exponent of 2, i.e., typically in \((-1024-50):1024\), otherwise the result will underflow to 0 or overflow to +/- Inf.
- **rnd.mode** a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see `mpfr`.

Value

`frexpMpfr` returns a list with named components \( r \) (of class mpfr) and \( e \) (integer valued, of type integer is small enough, “double” otherwise).

Author(s)

Martin Maechler

References

On unix-alikes, typically man `frexp` and man `ldexp`

See Also

Somewhat related, `mpfr2exp()`, `frexp()` and `ldexp()` in package `DPQ`.

Examples

```r
set.seed(47)
x <- c(0, 2^(-3:3), (-1:1)/0,
      sort(rlnorm(2^12, 10, 20) * sample(c(-1,1), 512, replace=TRUE)))
head(xM <- mpfr(x, 128), 11)
str(rFM <- frexpMpfr(xM))
```
d.fr <- with(rFM, data.frame(x=x, r=asNumeric(r), e=e))
head(d.fr, 16)
tail(d.fr)
ar <- abs(rFM$r)
stopifnot(0.5 <= ar[is.finite(x) & x != 0], ar[is.finite(x)] < 1,
is.integer(rFM$e))
ldx <- with(rFM, ldexpMpfr(r, e))
iN <- which(is.na(x)) # 10
stopifnot(exprs = {
  all.equal(xM, ldx, tol = 2^-124) # allow 4 bits loss, but apart from the NA, even:
  identical(xM[-iN], ldx[-iN])
  is.na(xM [iN])
  is.na(ldx[iN])
})

---

## gmp-conversions

### Conversion Utilities gmp <-> Rmpfr

#### Description

Coerce from and to big integers (`bigz`) and `mpfr` numbers. Further, coerce from big rationals (`bigq`) to `mpfr` numbers.

#### Usage

- `.bigz2mpfr(x, precB = NULL, rnd.mode = c("N","D","U","Z","A"))`
- `.bigq2mpfr(x, precB = NULL, rnd.mode = c("N","D","U","Z","A"))`
- `.mpfr2bigz(x, mod = NA)`
- `.mpfr2bigq(x)`

#### Arguments

- `x` an R object of class `bigz`, `bigq` or `mpfr` respectively.
- `precB` precision in bits for the result. The default, NULL, means to use the minimal precision necessary for correct representation.
- `rnd.mode` a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see details of `mpfr`.
- `mod` a possible modulus, see `as.bigz` in package `gmp`.

#### Details

Note that we also provide the natural (S4) coercions, `as(x, "mpfr")` for `x` inheriting from class "bigz" or "bigq".

#### Value

a numeric vector of the same length as `x`, of the desired class.
See Also

`mpfr()`, `as.bigz` and `as.bigq` in package `gmp`.

Examples

```r
S <- gmp::Stirling2(50,10)
show(S)
SS <- S * as.bigz(1:3)^128
stopifnot(all.equal(log2(SS[2]) - log2(S), 128, tolerance=1e-15),
          identical(SS, .mpfr2bigz(.bigz2mpfr(SS))))

.bigz2mpfr(S)       # 148 bit precision
.bigz2mpfr(S, precB=256) # 256 bit

## rational --> mpfr:
seq <- SS / as.bigz(2)^100
MP <- as(seq, "mpfr")
stopifnot(identical(MP, .bigq2mpfr(seq)),
          SS == MP * as(2, "mpfr")^100)

## New since 2024-01-20: mpfr --> big rational "bigq"
Pi <- Const("pi", 128)
m <- Pi * 2^(-5:5)
(m <- c(m, mpfr(2, 128)^(-5:5)))

getDenom <- Rmpfr:::getDenom
stopifnot(is.whole(m * (d.m <- getDenom(m))))
stopifnot(all.equal(m, mpfr(.mpfr2bigq(m), 130), tolerance = 2^-130)) # I see even
all.equal(m, mpfr(.mpfr2bigq(m), 130), tolerance = 0) # TRUE

m <- m * mpfr(2, 128)^200 # quite a bit larger
stopifnot(is.whole(m * (d.m <- getDenom(m))))
stopifnot(all.equal(m, mpfr(.mpfr2bigq(m), 130), tolerance = 2^-130)) # I see even
all.equal(m, mpfr(.mpfr2bigq(m), 130), tolerance = 0) # TRUE

m2 <- m * mpfr(2, 128)^20000 ## really huge
stopifnot(is.whole(m2 * (d.m2 <- getDenom(m2))))
stopifnot(all.equal(m2, mpfr(.mpfr2bigq(m2), 130), tolerance = 2^-130)) # I see even
all.equal(m2, mpfr(.mpfr2bigq(m2), 130), tolerance = 0) # TRUE
```

---

**hjkMpf**

**Hooke-Jeeves Derivative-Free Minimization R (working for MPFR)**

Description

An implementation of the Hooke-Jeeves algorithm for derivative-free optimization.

This is a slight adaption `hjk()` from package `dfoptim`. 

---
Usage

hjkMpfr(par, fn, control = list(), ...)  

Arguments

par    Starting vector of parameter values. The initial vector may lie on the boundary. If lower[i]=upper[i] for some i, the i-th component of the solution vector will simply be kept fixed.

fn     Nonlinear objective function that is to be optimized. A scalar function that takes a real vector as argument and returns a scalar that is the value of the function at that point.

control list of control parameters. See Details for more information.

... Additional arguments passed to fn.

Details

Argument control is a list specifying changes to default values of algorithm control parameters. Note that parameter names may be abbreviated as long as they are unique.

The list items are as follows:

tol   Convergence tolerance. Iteration is terminated when the step length of the main loop becomes smaller than tol. This does not imply that the optimum is found with the same accuracy. Default is 1.e-06.

maxfeval Maximum number of objective function evaluations allowed. Default is Inf, that is no restriction at all.

maximize A logical indicating whether the objective function is to be maximized (TRUE) or minimized (FALSE). Default is FALSE.

target A real number restricting the absolute function value. The procedure stops if this value is exceeded. Default is Inf, that is no restriction.

info   A logical variable indicating whether the step number, number of function calls, best function value, and the first component of the solution vector will be printed to the console. Default is FALSE.

If the minimization process threatens to go into an infinite loop, set either maxfeval or target.

Value

A list with the following components:

par    Best estimate of the parameter vector found by the algorithm.

value   value of the objective function at termination.

convergence indicates convergence (TRUE) or not (FALSE).

feval   number of times the objective fn was evaluated.

niter   number of iterations (“steps”) in the main loop.
Note

This algorithm is based on the Matlab code of Prof. C. T. Kelley, given in his book “Iterative methods for optimization”. It has been implemented for package `dfoptim` with the permission of Prof. Kelley.

This version does not (yet) implement a cache for storing function values that have already been computed as searching the cache makes it slower.

Author(s)

Hans W Borchers <hwborchers@googlemail.com>; for `Rmpfr`: John Nash, June 2012. Modifications by Martin Maechler.

References

C.T. Kelley (1999), Iterative Methods for Optimization, SIAM.

See Also

Standard R’s `optim`: `optimizeR` provides one-dimensional minimization methods that work with `mpfr`-class numbers.

Examples

```r
## simple smooth example:
ff <- function(x) sum((x - c(2:4))^2)
str(rr <- hjkMpfr(rep(mpfr(0,128), 3), ff, control=list(info=TRUE)))

doX <- Rmpfr:::doExtras(); cat("doExtras: ", doX, ", \n") # slow parts only if(doX)

## Hooke-Jeeves solves high-dim. Rosenbrock function (but slowly!)
rosenbrock <- function(x) {
  n <- length(x)
  sum (100*((x1 <- x[1:(n-1)])^2 - x[2:n])^2 + (x1 - 1)^2)
}
par0 <- rep(0, 10)
str(rb.db <- hjkMpfr(par0, 10), rosenbrock, control=list(info=TRUE))
if(doX) {
  ## rosenbrook() is quite slow with mpfr-numbers:
  str(rb.M. <- hjkMpfr(mpfr(numeric(10), prec=128), rosenbrock, control = list(tol = 1e-8, info=TRUE)))
}

## Hooke-Jeeves does not work well on non-smooth functions
nsf <- function(x) {
  return(c(f1, f2))
}
```

```r
## Hooke-Jeeves solves high-dim. Rosenbrock function (but slowly!)
rosenbrock <- function(x) {
  n <- length(x)
  sum (100*((x1 <- x[1:(n-1)])^2 - x[2:n])^2 + (x1 - 1)^2)
}
par0 <- rep(0, 10)
str(rb.db <- hjkMpfr(rep(0, 10), rosenbrock, control=list(info=TRUE))
```
\[ \max(f_1, f_2, f_3) \]
\[
\text{par0} <- c(1, 1) \# true min 7.2 at (1.2, 2.4)
\text{h.d} \leftarrow \text{hjkMpfr(par0, nsf)} \# fmin=8 at xmin=(2,2)
\]
\[
\text{if(doX)} { \quad \text{## and this is not at all better (but slower!)} \quad \text{h.M} \leftarrow \text{hjkMpfr(mpfr(c(1,1), 128), nsf, control = list(tol = 1e-15))} \}
\]
\[
\text{## \rightarrow demo(hjkMpfr) \# \rightarrow Fletcher's chebyquad function m = n -- residuals}
\]

\textbf{igamma}

\textit{Incomplete Gamma Function}

\textbf{Description}

For MPFR version \( \geq 3.2.0 \), the following MPFR library function is provided: \texttt{mpfr\_gamma\_inc(a,x)}, the R interface of which is \texttt{igamma(a,x)}, where \texttt{igamma(a,x)} is the “upper” incomplete gamma function

\[ \Gamma(a, x) := \Gamma(a) - \gamma(a, x), \]

where

\[ \gamma(a, x) := \int_0^x t^{a-1}e^{-t}dt, \]

and hence

\[ \Gamma(a, x) := \int_x^\infty t^{a-1}e^{-t}dt, \]

and

\[ \Gamma(a) := \gamma(a, \infty). \]

As R’s \texttt{pgamma(x,a)} is

\[ \text{pgamma(x, a)} := \gamma(a, x)/\Gamma(a), \]

we get

\[ \text{igamma(a,x)} == \text{gamma(a)} * \text{pgamma(x, a, lower.tail=FALSE)} \]

\textbf{Usage}

\[ \text{igamma(a, x, rnd.mode = c("N", "D", "U", "Z", "A"))} \]

\textbf{Arguments}

\begin{itemize}
  \item \texttt{a, x} \hspace{1cm} \text{an object of class mpfr or numeric.}
  \item \texttt{rnd.mode} \hspace{1cm} \text{a 1-letter string specifying how \textit{rounding} should happen at C-level conversion to MPFR, see mpfr.}
\end{itemize}
Value

A numeric vector of “common length”, recyling along a and x.

Author(s)

R interface: Martin Maechler

References


See Also

R’s gamma (function) and pgamma (probability distribution).

Examples

```r
## show how close pgamma() is :
x <- c(seq(0,20, by=1/4), 21:50, seq(55, 100, by=5))
if(mpfrVersion() >= "3.2.0") { print(
  all.equal(igamma(Const("pi", 80), x),
    pgamma(x, pi, lower.tail=FALSE) * gamma(pi),
    tol=0, formatFUN = function(., ...) format(., digits = 7)) #-> 2.75e-16 (was 3.13e-16)
  )
## and ensure *some* closeness:
stopifnot(exprs = {
  all.equal(igamma(Const("pi", 80), x),
    pgamma(x, pi, lower.tail=FALSE) * gamma(pi),
    tol = 1e-15)
})
# only if MPFR version >= 3.2.0
```

Description

Numerical integration of one-dimensional functions in pure R, with care so it also works for "mpfr"-numbers.

Currently, only classical Romberg integration of order ord is available.

Usage

```r
integrateR(f, lower, upper, ..., ord = NULL,
    rel.tol = .Machine$double.eps^0.25, abs.tol = rel.tol,
    max.ord = 19, verbose = FALSE)
```
Arguments

- \( f \): an \( \text{R} \) function taking a numeric or "mpfr" first argument and returning a numeric (or "mpfr") vector of the same length. Returning a non-finite element will generate an error.
- \( \text{lower, upper} \): the limits of integration. Currently must be finite. Do use "mpfr"-numbers to get higher than double precision, see the examples.
- ...: additional arguments to be passed to \( f \).
- \( \text{ord} \): integer, the order of Romberg integration to be used. If this is \( \text{NULL} \), as per default, and either \( \text{rel.tol} \) or \( \text{abs.tol} \) are specified, the order is increased until convergence.
- \( \text{rel.tol} \): relative accuracy requested. The default is 1.2e-4, about 4 digits only, see the Note.
- \( \text{abs.tol} \): absolute accuracy requested.
- \( \text{max.ord} \): only used, when neither \( \text{ord} \) or one of \( \text{rel.tol} \), \( \text{abs.tol} \) are specified: Stop Romberg iterations after the order reaches \( \text{max.ord} \); may prevent infinite loops or memory explosion.
- \( \text{verbose} \): logical or integer, indicating if and how much information should be printed during computation.

Details

Note that arguments after \(...\) must be matched exactly.
For convergence, both relative and absolute changes must be smaller than \( \text{rel.tol} \) and \( \text{abs.tol} \), respectively.

\( \text{rel.tol} \) cannot be less than \( \max(50\cdot\text{Machine\$double\_eps}, 0.5\cdot10^{-28}) \) if \( \text{abs.tol} \leq 0 \).

Value

A list of class "\text{integrateR}" (as from standard \( \text{R} \)'s \text{integrate()}\) with a \text{print} method and components

- \( \text{value} \): the final estimate of the integral.
- \( \text{abs.error} \): estimate of the modulus of the absolute error.
- \( \text{subdivisions} \): for Romberg, the number of function evaluations.
- \( \text{message} \): "OK" or a character string giving the error message.
- \( \text{call} \): the matched call.

Note

\( f \) must accept a vector of inputs and produce a vector of function evaluations at those points. The \text{Vectorize} function may be helpful to convert \( f \) to this form.
If you want to use higher accuracy, you must set \( \text{lower} \) or \( \text{upper} \) to "mpfr" numbers (and typically lower the relative tolerance, \( \text{rel.tol} \)), see also the examples.
Note that the default tolerances (\( \text{rel.tol} \), \( \text{abs.tol} \)) are not very accurate, but the same as for \text{integrate}, which however often returns considerably more accurate results than requested. This is typically not the case for \text{integrateR()}.
Note

We use practically the same print S3 method as \texttt{print.integrate}, provided by \texttt{R}, with a difference when the message component is not "Ok".

Author(s)

Martin Maechler

References


See Also

\texttt{R}'s standard, \texttt{integrate}, is much more adaptive, also allowing infinite integration boundaries, and typically considerably faster for a given accuracy.

Examples

```r
## See more from \texttt{?integrate}
## this is in the region where \texttt{integrate()} can get problems:
integrateR(dnorm,0,2000)
integrateR(dnorm,0,2000, rel.tol=1e-15)
(Id <- integrateR(dnorm,0,2000, rel.tol=1e-15, verbose=TRUE))
Id$value == 0.5 # exactly

## Demonstrating that 'subdivisions' is correct:
Exp <- function(x) { .N <<- .N+ length(x); exp(x) }
.N <- 0; str(integrateR(Exp, 0,1, rel.tol=1e-10), digits=15); .N

## Using high-precision functions ----- 

## Polynomials are very nice:
integrateR(function(x) (x-2)^4 - 3*(x-3)^2, 0, 5, verbose=TRUE)
# n= 1, 2^n= 2 | I = 46.04, abs.err = 98.9583
# n= 2, 2^n= 4 | I = 20, abs.err = 26.0417
# n= 3, 2^n= 8 | I = 20, abs.err = 7.10543e-15
## 20 with absolute error < 7.1e-15
## Now, using higher accuracy:
I <- integrateR(function(x) (x-2)^4 - 3*(x-3)^2, 0, mpfr(5,128),
               rel.tol = 1e-20, verbose=TRUE)
I ; I$value ## all fine

## with floats:
integrateR(exp, 0 , 1, rel.tol=1e-15, verbose=TRUE)
## with "mpfr":
(I <- integrateR(exp, mpfr(0,200), 1, rel.tol=1e-25, verbose=TRUE))
(I.true <- exp(mpfr(1, 200)) - 1)
## true absolute error:
stopifnot(print(as.numeric(I.true - I$value)) < 4e-25)
```
is.whole

## Want absolute tolerance check only (=> set 'rel.tol' very high, e.g. 1):
(Ia <- integrateR(exp, mpfr(0,200), 1, abs.tol = 1e-6, rel.tol=1, verbose=TRUE))

## Set 'ord' (but no '*tol') --> Using 'ord'; no convergence checking
(I <- integrateR(exp, mpfr(0,200), 1, ord = 13, verbose=TRUE))

### is.whole

**Whole ("Integer") Numbers**

#### Description

Check which elements of \(x[\cdot]\) are integer valued aka “whole” numbers, including MPFR numbers (class `mpfr`).

#### Usage

```
## S3 method for class 'mpfr'
is.whole(x)
```

#### Arguments

- **x**: any \(\mathbb{R}\) vector, here of class `mpfr`.

#### Value

Logical vector of the same length as \(x[\cdot]\), indicating where \(x[\cdot]\) is integer valued.

#### Author(s)

Martin Maechler

#### See Also

- `is.integer(x)` (base package) checks for the internal mode or class, not if \(x[i]\) are integer valued.

  The `is.whole()` methods in package `gmp`.

#### Examples

```r
is.integer(3)  # FALSE, it's internally a double
is.whole(3)    # TRUE
x <- c(as(2,"mpfr")^100, 3, 3.2, 1000000, 2^40)
is.whole(x)    # one FALSE, only
```
log1mexp

Compute \( f(a) = \log(1 \pm \exp(-a)) \) Numerically Optimally

Description

Compute \( f(a) = \log(1 - \exp(a)) \), respectively \( g(x) = \log(1 + \exp(x)) \) quickly numerically accurately.

Usage

\[
\begin{align*}
\log1mexp(a, \text{cutoff} = \log(2)) \\
\log1pexp(x, c0 = -37, c1 = 18, c2 = 33.3)
\end{align*}
\]

Arguments

- \( a \) numeric (or \texttt{mpfr}) vector of positive values.
- \( x \) numeric vector, may also be an \texttt{mpfr} object.
- \( \text{cutoff} \) positive number; \( \log(2) \) is “optimal”, but the exact value is unimportant, and anything in \([0.5, 1]\) is fine.
- \( c0, c1, c2 \) cutoffs for \( \log1pexp \); see below.

Value

\[
\log1mexp(a) := f(a) = \log(1 - \exp(-a)) = \log1p(-\exp(-a)) = \log(-\expm1(-a))
\]

or, respectively,

\[
\log1pexp(x) := g(x) = \log(1 + \exp(x)) = \log1p(\exp(x))
\]

computed accurately and quickly.

Author(s)

Martin Maechler, May 2002; \texttt{log1pexp()} in 2012

References

Martin Mächler (2012). Accurately Computing \( \log(1 - \exp(-|a|)) \); \url{https://CRAN.R-project.org/package=Rmpfr/vignettes/log1mexp-note.pdf}. 
Examples

```r
fExpr <- expression(
  DEF = log(1 - exp(-a)),
  expm1 = log(-expm1(-a)),
  log1p = log1p(-exp(-a)),
  F = log1mexp(a))
a <- 2 ^ seq(-58, 10, length = 256)
a <- a ; str(fa <- do.call(cbind, as.list(fExpr)))
head(fa)# expm1() works here
tail(fa)# log1p() works here
## graphically:
lwd <- 1.5*(5:2); col <- adjustcolor(1:4, 0.4)
op <- par(mfcol=c(1,2), mgp = c(1.25, .6, 0), mar = .1+c(3,2,1,1))
  matplot(a, fa, type = "l", log = "x", col=col, lwd=lwd)
  legend("topleft", fExpr, col=col, lwd=lwd, lty=1:4, bty="n")
  # expm1() & log1mexp() work here
  matplot(a, -fa, type = "l", log = "xy", col=col, lwd=lwd)
  legend("left", paste("-",fExpr), col=col, lwd=lwd, lty=1:4, bty="n")
  # log1p() & log1mexp() work here
par(op)
```

```r
aM <- 2 ^ seqMpfr(-58, 10, length=length(a.)) # => default prec = 128
a <- aM; dim(faM <- do.call(cbind, as.list(fExpr))) # 256 x 4, "same" as 'fa'
## Here, for small 'a' log1p() and even 'DEF' is still good enough
l_f <- asNumeric(log(-faM))
all.equal(l_f[,"F"], l_f[,"log1p"], tol=0) # see TRUE (Lnx 64-bit)
io <- a. < 80 # for these, the differences are small
all.equal(l_f[i0,"F"], l_f[i0,"expm1"], tol=0) # see 6.662e-9
all.equal(l_f[i0,"F"], l_f[i0, "DEF" ], tol=0)
stopifnot(exprs = {
  all.equal(l_f[i0,"F"], l_f[i0,"log1p"], tol= 1e-15)
  all.equal(l_f[i0,"F"], l_f[i0,"expm1"], tol= 1e-7)
  all.equal(l_f[i0,"F"], l_f[i0, "DEF" ], tol= 1e-7)
})
## For 128-bit prec, if we go down to 2^-130, "log1p" is no longer ok:
aM2 <- 2 ^ seqMpfr(-130, 10, by = 1/2)
a <- aM2; fa2 <- do.call(cbind, as.list(fExpr))
head(asNumeric(fa2), 12)
tail(asNumeric(fa2), 12)
matplot(a, log(-fa2[,1:3]) -log(-fa2[,"F"])), type="l", log="x",
  lty=1:3, lwd=2*31:1-1, col=adjustcolor(2:4, 1/3))
legend("top", colnames(fa2)[1:3], lty=1:3, lwd=2*(3:1)-1, col=adjustcolor(2:4, 1/3))
cols <- adjustcolor(2:4, 1/3); lwd <- 2*(3:1)-1
matplot(a, 1e-40*abs(log(-fa2[,1:3]) -log(-fa2[,"F"])), type="o", log="xy",
  main = "log1mexp(a) -- approximation rel.errors, mpfr(*, prec=128)",
  pch=21:23, cex=.6, bg=5:7, lty=1:2, lwd=lwd, col=cols)
legend("top", colnames(fa2)[1:3], lty=1:2, lwd=lwd, col=cols,
  pch=21:23, pt.cex=.6, pt.bg=5:7)
```
## log1pexp() [simpler]

```r
curve(log1pexp, -10, 10, asp=1)
abline(0,1, h=0, v=0, lty=3, col="gray")
```

### Cutoff c1 for log1pexp() -- not often "needed":

```r
curve(log1p(exp(x)) - log1pexp(x), 16, 20, n=2049)
```

### Cutoff for log1pexp():

```r
x <- seq(700, 720, by=2)
cbind(x, log1p(exp(x)), log1pexp(x))
```

```r
curve((x+exp(-x)) - x, 20, 40, n=1025)
curve((x+exp(-x)) - x, 33.1, 33.5, n=1025)
```

---

### matmult

**(MPFR) Matrix (Vector) Multiplication**

**Description**

Matrix / vector multiplication of mpfr (and "simple" numeric) matrices and vectors.

`matmult(x, y, fPrec = 2)` or `crossprod(x, y, fPrec = 2)` use higher precision in underlying computations.

**Usage**

```r
matmult(x, y, ...)
```

**Arguments**

- **x, y** numeric or mpfrMatrix-classed R objects, i.e. semantically numeric matrices or vectors.
- **...** arguments passed to the hidden underlying .matmult.R() work horse which is also underlying the `%*%`, `crossprod()`, and `tcrossprod()` methods, see the mpfrMatrix class documentation:
  - **fPrec** a multiplication factor, a positive number determining the number of bits precBits used for the underlying multiplication and summation arithmetic. The default is fPrec = 1. Setting fPrec = 2 doubles the precision which has been recommended, e.g., by John Nash.
  - **precBits** the number of bits used for the underlying multiplication and summation arithmetic; by default precBits = fPrec * max(getPrec(x), getPrec(y)) which typically uses the same accuracy as regular mpfr-arithmetic would use.
Value

a (base R) matrix or mpfrMatrix, depending on the classes of x and y.

Note

Using matmult(x, y) instead of x %*% y, makes sense mainly if you use non-default fPrec or precBits arguments.

The crossprod(), and tcrossprod() function have the identical optional arguments fPrec or precBits.

Author(s)

Martin Maechler

See Also

%*%, crossprod, tcrossprod.

Examples

```r
## FIXME: add example
## 1) matmult() <--> %*%
## 2) crossprod(), tcrossprod() %<-- ./mpfrMatrix-class.Rd examples (!)
```

Mnumber-class

Class "Mnumber" and "mNumber" of "mpfr" and regular numbers and arrays from them

Description

Classes "Mnumber" "mNumber" are class unions of "mpfr" and regular numbers and arrays from them.
Its purpose is for method dispatch, notably defining a cbind(...) method where ... contains objects of one of the member classes of "Mnumber".

Classes "mNumber" is considerably smaller is it does not contain "matrix" and "array" since these also extend "character" which is not really desirable for generalized numbers. It extends the simple "numericVector" class by mpfr* classes.
Methods

%%signature(x = "mpfrMatrix", y = "Mnumber")
crossprod signature(x = "mpfr", y = "Mnumber")
tcrossprod signature(x = "Mnumber", y = "mpfr")
etc. These are documented with the classes mpfr and or mpfrMatrix.

See Also

the array_or_vector sub class; cbind-methods.

Examples

## "Mnumber" encompasses (i.e., "extends") quite a few
## "vector / array - like" classes:
showClass("Mnumber")
stopifnot(extends("mpfrMatrix", "Mnumber"),
       extends("array", "Mnumber"))

Mnsub <- names(getClass("Mnumber")@subclasses)
(mNsub <- names(getClass("mNumber")@subclasses))
## mNumber has *one* subclass which is not in Mnumber:
setdiff(mNsub, Mnsub)# namely "numericVector"
## The following are only subclasses of "Mnumber", but not of "mNumber":
setdiff(Mnsub, mNsub)

mpfr

Create "mpfr" Numbers (Objects)

Description

Create multiple (i.e. typically high) precision numbers, to be used in arithmetic and mathematical computations with R.

Usage

mpfr(x, precBits, ...)
## Default S3 method:
mpfr(x, precBits, base = 10,
     rnd.mode = c("N","D","U","Z","A"), scientific = NA, ...)
Const(name = c("pi", "gamma", "catalan", "log2"), prec = 120L,
      rnd.mode = c("N","D","U","Z","A"))

is.mpfr(x)
Arguments

- **x**: a numeric, mpfr, bigz, bigq, or character vector or array.
- **precBits**, **prec**: a number, the maximal precision to be used, in bits; i.e. 53 corresponds to double precision. Must be at least 2. If missing, `getPrec(x)` determines a default precision.
- **base**: (only when x is character) the base with respect to which x[i] represent numbers; base b must fulfill 2 ≤ b ≤ 62.
- **rnd.mode**: a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see details.
- **scientific**: (used only when x is the result of `formatBin()`), i.e., of class "Bcharacter": logical indicating that the binary representation of x is in scientific notation. When TRUE, mpfr() will substitute 0 for \_; when NA, mpfr() will guess, and use TRUE when finding a "p" in x; see also `formatBin`.
- **name**: a string specifying the mpfrlib - internal constant computation. "gamma" is Euler's gamma (\(\gamma\)), and "catalan" Catalan's constant.
- ... potentially further arguments passed to and from methods.

Details

The "mpfr" method of mpfr() is a simple wrapper around `roundMpfr()`. MPFR supports the following rounding modes,

- **GMP_RNDN**: round to nearest (roundTiesToEven in IEEE 754-2008).
- **GMP_RNDZ**: round toward zero (roundTowardZero in IEEE 754-2008).
- **GMP_RNDU**: round toward plus infinity ("Up", roundTowardPositive in IEEE 754-2008).
- **GMP_RNDD**: round toward minus infinity ("Down", roundTowardNegative in IEEE 754-2008).
- **GMP_RNDA**: round away from zero (new since MPFR 3.0.0).

The ‘round to nearest’ ("N") mode, the default here, works as in the IEEE 754 standard: in case the number to be rounded lies exactly in the middle of two representable numbers, it is rounded to the one with the least significant bit set to zero. For example, the number 5/2, which is represented by (10.1) in binary, is rounded to (10.0)=2 with a precision of two bits, and not to (11.0)=3. This rule avoids the "drift" phenomenon mentioned by Knuth in volume 2 of The Art of Computer Programming (Section 4.2.2).

When x is character, mpfr() will detect the precision of the input object.

Value

an object of (S4) class mpfr, or for mpfr(x) when x is an array, mpfrMatrix, or mpfrArray which the user should just as a normal numeric vector or array.

`is.mpfr()` returns TRUE or FALSE.

Author(s)

Martin Maechler
References


See Also

The class documentation mpfr contains more details. Use asNumeric to transform back to double precision ("numeric").

Examples

mpfr(pi, 120) ## the double-precision pi "translated" to 120-bit precision

pi. <- Const("pi", prec = 260) # pi "computed" to correct 260-bit precision
pi. # nicely prints 80 digits [260 * log10(2) =~ 78.3 ~ 80]

Const("gamma", 128L) # 0.5772...
Const("catalan", 128L) # 0.9159...

x <- mpfr(0:7, 100)/7 # a more precise version of k/7, k=0,...,7
x
1 / x

## character input :
mpfr("2.718281828459045235360287471352662497757") - exp(mpfr(1, 150))
## = -4 * 10^-40
## Also works for NA, NaN, ... :
cx <- c("1234567890123456", 345, "NA", "NaN", "Inf", "-Inf")
mpfr(cx)

## with some 'base' choices :
print(mpfr("111.1111", base=2)) * 2^4

mpfr("af21.01020300a0b0c", base=16)
## 68 bit prec. 44833.00394694653820642

mpfr("ugi0", base = 32) == 10^6 ## TRUE

## --- Large integers from package 'gmp':
Z <- as.bigz(7)^(1:200)
head(Z, 40)

## mpfr(Z) by default chooses the correct *maximal* default precision:
mZ. <- mpfr(Z)
## more efficiently chooses precision individually
m.Z <- mpfr(Z, precBits = frexpZ(Z)$exp)
## the precBits chosen are large enough to keep full precision:
stopifnot(identical(cZ <- as.character(Z),
 as(mZ.,"character")),
 identical(cZ, as(m.Z,"character")))

## compare mpfr-arithmetic with exact rational one:
stopifnot(all.equal(mpfr(as.bigq(355,113), 99),
        mpfr(355, 99) / 113, tol = 2^-98))

## look at different "rounding modes":
sapply(c("N", "D","U","Z","A"), function(RND)
        mpfr(c(-1,1)/5, 20, rnd.mode = RND), simplify=FALSE)
symnum(sapply(c("N", "D","U","Z","A"),
        function(RND) mpfr(0.2, prec = 5:15, rnd.mode = RND) < 0.2 ))

mpfr-class Class "mpfr" of Multiple Precision Floating Point Numbers

Description
"mpfr" is the class of Multiple Precision Floatingpoint numbers with Reliable arithmetic.

For the high-level user, "mpfr" objects should behave as standard R's numeric vectors. They would just print differently and use the prespecified (typically high) precision instead of the double precision of 'traditional' R numbers (with class(.) == "numeric" and typeof(.) == "double").

hypot(x,y) computes the hypothenuse length z in a rectangular triangle with "leg" side lengths x and y, i.e.,
\[
z = \text{hypot}(x,y) = \sqrt{x^2 + y^2},
\]
in a numerically stable way.

Usage
hypot(x,y, rnd.mode = c("N","D","U","Z","A"))

Arguments
x, y an object of class mpfr.
rnd.mode a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see mpfr.

Objects from the Class
Objects are typically created by mpfr(<number>, precBits).
summary(<mpfr>) returns an object of class "summaryMpfr" which contains "mpfr" but has its own print method.

Slots
Internally, "mpfr" objects just contain standard R lists where each list element is of class "mpfr1", representing one MPFR number, in a structure with four slots, very much parallelizing the C struct in the mpfr C library to which the Rmpfr package interfaces.

An object of class "mpfr1" has slots
prec: "integer" specifying the maximal precision in **bits**.
exp: "integer" specifying the base-2 exponent of the number.
sign: "integer", typically -1 or 1, specifying the sign (i.e. `sign(.)`) of the number.
d: an "integer" vector (of 32-bit “limbs”) which corresponds to the full mantissa of the number.

Methods

**abs** signature(x = "mpfr"): ...

**atan2** signature(y = "mpfr", x = "ANY"), and
atan2 signature(x = "ANY", y = "mpfr"): compute the arc-tangent of two arguments: $\tan^{-1}(y, x)$ returns the angle between the x-axis and the vector from the origin to $(x, y)$, i.e., for positive arguments $\tan^{-1}(y, x) = \tan(y/x)$.

**lbeta** signature(a = "ANY", b = "mpfrArray"), is $\log(|B(a,b)|)$ where $B(a,b)$ is the Beta function, $\beta(a,b)$.

**beta** signature(a = "mpfr", b = "ANY"), **beta** signature(a = "mpfr", b = "mpfr"), ...., etc: Compute the beta function $B(a,b)$, using high precision, building on internal `gamma` or `lgamma`. See the help for R’s base function `beta` for more. Currently, there, $a, b \geq 0$ is required. Here, we provide (non-NaN) for all numeric $a, b$.

When either $a, b$, or $a + b$ is a negative integer, $\Gamma(.)$ has a pole there and is undefined (NaN). However the Beta function can be defined there as “limit”, in some cases. Following other software such as SAGE, Maple or Mathematica, we provide finite values in these cases. However, note that these are not proper limits (two-dimensional in $(a,b)$), but useful for some applications. E.g., $B(a,b)$ is defined as zero when $a + b$ is a negative integer, but neither $a$ nor $b$ is. Further, if $a > b > 0$ are integers, $B(-a,b) = B(b,-a)$ can be seen as $(-1)^b * B(a-b+1,b)$.

**dim<** signature(x = "mpfr"): Setting a dimension **dim** on an "mpfr" object makes it into an object of class "mpfrArray" or (more specifically) "mpfrMatrix" for a length-2 dimension, see their help page; note that t(x) (below) is a special case of this.

**Ops** signature(e1 = "mpfr", e2 = "ANY"): ...

**Ops** signature(e1 = "ANY", e2 = "mpfr"): ...

**Arith** signature(e1 = "mpfr", e2 = "missing"): ...

**Arith** signature(e1 = "mpfr", e2 = "mpfr"): ...

**Arith** signature(e1 = "mpfr", e2 = "integer"): ...

**Arith** signature(e1 = "mpfr", e2 = "numeric"): ...

**Arith** signature(e1 = "integer", e2 = "mpfr"): ...

**Arith** signature(e1 = "numeric", e2 = "mpfr"): ...

**Compare** signature(e1 = "mpfr", e2 = "mpfr"): ...

**Compare** signature(e1 = "mpfr", e2 = "integer"): ...

**Compare** signature(e1 = "mpfr", e2 = "numeric"): ...

**Compare** signature(e1 = "integer", e2 = "mpfr"): ...

**Compare** signature(e1 = "numeric", e2 = "mpfr"): ...
Logic signature(e1 = "mpfr", e2 = "mpfr"): ...

Summary signature(x = "mpfr"): The S4 Summary group functions, max, min, range, prod, sum, any, and all are all defined for MPFR numbers. mean(x, trim) for non-0 trim works analogously to mean.default.

median signature(x = "mpfr"): works via
quantile signature(x = "mpfr"): a simple wrapper of the quantile.default method from stats.
summary signature(object = "mpfr"): modeled after summary.default, ensuring to provide the full "mpfr" range of numbers.

Math signature(x = "mpfr"): All the S4 Math group functions are defined, using multiple precision (MPFR) arithmetic, from getGroupMembers("Math"), these are (in alphabetical order): abs, sign, sqrt, ceiling, floor, trunc, cummax, cummin, cumprod, cumsum, exp, expm1, log, log10, log2, log1p, cos, cosh, sin, sinh, tan, tanh, acos, acosh, asin, asinh, atan, atanh, cospi, sinpi, tanpi, gamma, lgamma, digamma, and trigamma.
Currently, trigamma is not provided by the MPFR library and hence not yet implemented. Further, the cum*() methods are not yet implemented.

factorial signature(x = "mpfr"): this will round the result when x is integer valued. Note however that factorialMpfr(n) for integer n is slightly more efficient, using the MPFR function mpfr_fac_ui.

Math2 signature(x = "mpfr"): round(x,digits) and signif(x, digits) methods. Note that these do not change the formal precision ('prec' slot), and you may often want to apply roundMpfr() in addition or preference.

as.numeric signature(x = "mpfr"): ...

as.vector signature(x = "mpfrArray"): as for standard arrays, this “drops” the dim (and dimnames), i.e., transforms x into an ‘MPFR’ number vector, i.e., class mpfr.

[[ signature(x = "mpfr", i = "ANY"), and

[ signature(x = "mpfr", i = "ANY", j = "missing", drop = "missing"): subsetting aka “indexing” happens as for numeric vectors.

format signature(x = "mpfr"), further arguments digits = NULL, scientific = NA, etc: returns character vector of same length as x; when digits is NULL, with enough digits to recreate x accurately. For details, see formatMpfr.

is.finite signature(x = "mpfr"): ...

is.infinite signature(x = "mpfr"): ...

is.na signature(x = "mpfr"): ...

is.nan signature(x = "mpfr"): ...

log signature(x = "mpfr"): ...

show signature(object = "mpfr"): ...

sign signature(x = "mpfr"): ...

Re, Im signature(z = "mpfr"): simply return z or 0 (as "mpfr" numbers of correct precision), as mpfr numbers are ‘real’ numbers.

Arg, Mod, Conj signature(z = "mpfr"): these are trivial for our ‘real’ mpfr numbers, but defined to work correctly when used in R code that also allows complex number input.
all.equal signature(target = "mpfr", current = "mpfr"),
all.equal signature(target = "mpfr", current = "ANY"), and
all.equal signature(target = "ANY", current = "mpfr"): methods for numerical (approximate)
equality, all.equal of multiple precision numbers. Note that the default tolerance (argument)
is taken to correspond to the (smaller of the two) precisions when both main arguments
are of class "mpfr", and hence can be considerably less than double precision machine epsilon
Machine$double.eps.

coerce signature(from = "numeric", to = "mpfr"): as(. , "mpfr") coercion methods are available
for character strings, numeric, integer, logical, and even raw. Note however, that
mpfr(., precBits, base) is more flexible.

coerce signature(from = "mpfr", to = "bigz"): coerces to biginteger, see bigz in package gmp.

collapse signature(from = "mpfr", to = "numeric"): ...
collapse signature(from = "mpfr", to = "character"): ...

unique signature(x = "mpfr"), and corresponding S3 method (such that unique(<mpfr>) works
inside base functions), see unique.

Note that duplicated() works for "mpfr" objects without the need for a specific method.

t signature(x = "mpfr"): makes x into an $n \times 1$ mpfrMatrix.

which.min signature(x = "mpfr"): gives the index of the first minimum, see which.min.

which.max signature(x = "mpfr"): gives the index of the first maximum, see which.max.

Note

Many more methods ("functions") automagically work for "mpfr" number vectors (and matrices,
see the mpfrMatrix class doc), notably sort, order, quantile, rank.

Author(s)

Martin Maechler

See Also

The "mpfrMatrix" class, which extends the "mpfr" one.

roundMpfr to change precision of an "mpfr" object which is typically desirable instead of or in
addition to signif() or round(); is.whole() etc.

Special mathematical functions such as some Bessel ones, e.g., jn; further, zeta(.) (= \( \zeta(.) \)), Ei()
etc. Bernoulli numbers and the Pochhammer function pochMpfr.

Examples

```r
## 30 digit precision
(x <- mpfr(c(2:3, pi), prec = 30 * log2(10)))
str(x) # str() displays *compact*ly => not full precision
x^2
x[1] / x[2] # 0.666666... ~ 30 digits
```

```r
## indexing - as with numeric vectors
```
### mpfr-class

stopifnot(exprs = {
  identical(x[2], x[[2]])
  ## indexing "outside" gives NA (well: "mpfr-NaN" for now):
  is.na(x[5])
  ## whereas "[" cannot index outside:
  inherits(tryCatch(x[[5]], error=identity), "error")
  ## and only select one element:
  inherits(tryCatch(x[[2:3]], error=identity), "error")
})

## factorial() & lfactorial would work automagically via lgamma(),
## but factorial() additionally has an "mpfr" method which rounds
f200 <- factorial(mpfr(200, prec = 1500)) # need high prec.!

f200

as.numeric(log2(f200)) # 1245.38 -- need precBits >~ 1246 for full precision

##--> see factorialMpfr() for more such computations.

### "Underflow" **much** later -- exponents have 30(+1) bits themselves:

mpfr.min.exp2 <- - (2^30 + 1)
two <- mpfr(2, 55)

stopifnot(two ^ mpfr.min.exp2 == 0)

## whereas

two ^ (mpfr.min.exp2 * (1 - 1e-15))

## 2.382564e-323228497 ["typically"]

### "Assert" that (sort), (order), (quantile), (rank), all work :

p <- mpfr(rpois(32, lambda=500), precBits=128)^10

np <- as.numeric(log(p))

(sp <- summary(p)) # using the print.summaryMpfr() method

stopifnot(all(diff(sort(p)) >= 0),
  identical(order(p), order(np)),
  identical(rank (p), rank (np)),
  all.equal(sapply(1:9, function(Typ) quantile(np, type=Typ, names=FALSE)),
    sapply(lapply(1:9, function(Typ) quantile( p, type=Typ, names=FALSE)),
    function(x) as.numeric(log(x))),
    tol = 1e-3),
  TRUE)

m0 <- mpfr(numeric(), 99)

xy <- expand.grid(x = -2:2, y = -2:2) ; x <- xy[, "x"] ; y <- xy[, "y"]

a2. <- atan2(y,x)

stopifnot(identical(which.min(m0), integer(0)),
  identical(which.max(m0), integer(0)),
  all.equal(a2., atan2(as(y,"mpfr"), x)),
  max(m0) == mpfr(-Inf, 53), # (53 is not a feature, but ok)
  min(m0) == mpfr(+Inf, 53),
  sum(m0) == 0, prod(m0) == 1)

## unique(), now even base::factor() "works" on <mpfr> :
```r
set.seed(17)
p <- rlnorm(20) * mpfr(10, 100)^-999
pp <- sample(p, 50, replace=TRUE)
str(unique(pp)) # length 18 .. (from originally 20)
## Class 'mpfr' [package "Rmpfr"] of length 18 and precision 100
## 5.56520587824e-999 4.41636588227e-1000 ..
facp <- factor(pp)
str(facp) # the factor *levels* are a bit verbose:
# Factor w/ 18 levels "new("mpfr1", ...........)" ...
# At least *some* factor methods work:
stopifnot(exprs = {
is.factor(facp)
  identical(unname(table(facp)),
    unname(table(asNumeric(pp * mpfr(10,100)^1000))))
})
```

```r
## ((unfortunately, the expressions are wrong; should integer "L"))
#
## More useful: levels with which to *invert* factor():
## -- this is not quite ok:
## simplified from 'utils':
deparse1 <- function(x, ...) paste(deparse(x, 500L, ...), collapse = " ")
if(FALSE) {
  str(pp.levs <- vapply(unclass(sort(unique(pp))), deparse1, ")
  facp2 <- factor(pp, levels = pp.levs)
}
```

---

**Distribution Functions with MPFR Arithmetic**

**Description**

For some R standard (probability) density, distribution or quantile functions, we provide MPFR versions.

**Usage**

dpois(x, lambda, log = FALSE, useLog = )
dbinom(x, size, prob, log = FALSE, useLog = )
dnbinom(x, size, prob, mu, log = FALSE, useLog = any(x > 1e6))
dnorm(x, mean = 0, sd = 1, log = FALSE)
dgamma(x, shape, rate = 1, scale = 1/rate, log = FALSE)
dt(x, df, ncp, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
Arguments

- `x, q, lambda, size, prob, mu, mean, sd, shape, rate, scale, df, ncp` numeric or `mpfr` vectors. All of these are “recycled” to the length of the longest one. For their meaning/definition, see the corresponding standard R (`stats`) package function.
- `log, log.p, lower.tail` logical, see `pnorm`, `dpois`, etc.
- `useLog` logical with default depending on `x` etc, indicating if log-scale computation should be used even when `log = FALSE`, for performance or against overflow / underflow.

Details

`pnorm()` is based on `erf()` and `erfc()` which have direct MPFR counter parts and are both reparametrizations of `pnorm`, `erf(x) = 2*pnorm(sqrt(2)*x)` and `erfc(x) = 2* pnorm(sqrt(2)*x, lower=FALSE)`.

Value

A vector of the same length as the longest of `x, q, ...`, of class `mpfr` with the high accuracy results of the corresponding standard R function.

Note

E.g., for `pnorm(*, log.p = TRUE)` to be useful, i.e., not to underflow or overflow, you may want to extend the exponential range of MPFR numbers, using `.mpfr_erange_set()`, see the examples.

See Also

- `dpois` in standard package `stats`.
- `pbetaI(x, a,b)` is a `mpfr` version of `pbeta` only for integer `a` and `b`.

Examples

```r
x <- 1400 + 0:10
print(dpois(x, 1000), digits = 18) ## standard R's double precision
(px <- dpois(mpfr(x, 120), 1000))## more accuracy for the same
px. <- dpois(mpfr(x, 120), 1000, useLog=TRUE)# {failed in 0.8-8}
stopifnot(all.equal(px, px., tol = 1e-31))
dpois(0:5, mpfr(10000, 80)) ## very small exponents (underflowing in dbl.prec.)
print(dbinom(0:8, 8, pr = 4 / 5), digits=18)
dbinom(0:8, 8, pr = 4/mpfr(5, 99)) -> dB; dB
print(dnorm(-5:5), digits=18)
dnorm(mpfr(-5:5, prec=99))
```

```r
## For pnorm() in the extreme tails, need an exponent range
## larger than the (MPFR and Rmpfr) default:
(old_eranges <- .mpfr_erange()) # typically -/+ 2^30:
```

```r
```
log2(abs(old_eranges)) # 30 30
.mpfr_erange_set(value = (1-2^-52)*.mpfr_erange(c("min.emin","max.emax")))
log2(abs(.mpfr_erange()))# 62 62 *if* setup -- 2023-01: *not* on Winbuilder, nor
## other Windows where long is 4 bytes (32 bit) and the erange typically cannot be extended.
tens <- mpfr(10^4:7, 128)

pnorm(tens, lower.tail=FALSE, log.p=TRUE) # "works" (iff ...)
## "the" boundary:

pnorm(mpfr(- 38581.371, 128), log.p=TRUE) # still does not underflow (but *.372 does)
## -744261105.5992838248196753129188937418 (iff ...)

## reset to previous

pnorm(tens, lower.tail=FALSE, log.p=TRUE) # all but first underflow to -Inf

---

### mpfr-special-functions

**Special Mathematical Functions (MPFR)**

**Description**

Special Mathematical Functions, supported by the MPFR Library.

Note that additionally, all the Math and Math2 group member functions are “mpfr-ified”, too; ditto, for many more standard R functions. See see the methods listed in mpfr (aka `?mpfr-class`).

**Usage**

- `zeta(x)`
- `Ei(x)`
- `Li2(x)`
- `erf(x)`
- `erfc(x)`

**Arguments**

- `x` a numeric or mpfr vector.

**Details**

`zeta(x)` computes Riemann’s Zeta function $\zeta(x)$ important in analytical number theory and related fields. The traditional definition is

$$\zeta(x) = \sum_{n=1}^{\infty} \frac{1}{n^x}.$$
mpfr-utils

Ei(x) computes the exponential integral,
\[ \int_{-\infty}^{x} \frac{e^t}{t} dt. \]

Li2(x) computes the dilogarithm,
\[ \int_{0}^{x} \frac{-\log(1-t)}{t} dt. \]

erf(x) and erfc(x) are the error, respectively complementary error function which are both reparametrizations of pnorm, \( \text{erf}(x) = 2\times pnorm(\sqrt{2}\times x) \) and \( \text{erfc}(x) = 2\times pnorm(\sqrt{2}\times x, \text{lower}=	ext{FALSE}) \), and hence Rmpfr provides its own version of pnorm.

Value
A vector of the same length as x, of class mpfr.

See Also
pnorm in standard package stats; the class description mpfr mentioning the generic arithmetic and mathematical functions (sin, log, ...., etc) for which "mpfr" methods are available.

Note the (integer order, non modified) Bessel functions \( j_0() \), \( y_n() \), etc, named \( j0, \) \( yn \) etc, and Airy function \( Ai() \) in Bessel_mpfr.

Examples
```r
curve(Ei, 0, 5, n=2001)
## As we now require (mpfrVersion() >= "2.4.0"):
curve(Li2, 0, 5, n=2001)
curve(Li2, -2, 13, n=2000); abline(h=0,v=0, lty=3)
curve(Li2, -200, 400, n=2000); abline(h=0,v=0, lty=3)

curve(erf, -3.3, col = "red", ylim = c(-1,2))
curve(erfc, add = TRUE, col = "blue")
abline(h=0, v=0, lty=3)
legend(-3.1, c("erf(x)", "erfc(x)"), col = c("red","blue"), lty=1)
```

Description
This page documents utilities from package Rmpfr which are typically not called by the user, but may come handy in some situations.

Notably, the (base-2) maximal (and minimal) precision and the “erange”, the range of possible (base-2) exponents of mpfr-numbers can be queried and partly extended.
Usage

getPrec(x, base = 10, doNumeric = TRUE, is.mpfr = NA, bigq. = 128L)
.getPrec(x)
.getD(x)
mpfr_default_prec(prec)
## S3 method for class 'mpfrArray'
print(x, digits = NULL, drop0trailing = FALSE,
   right = TRUE,
   max.digits = getOption("Rmpfr.print.max.digits", 999L),
   exponent.plus = getOption("Rmpfr.print.exponent.plus", TRUE),
   ...)
## S3 method for class 'mpfr'
print(x, digits = NULL, drop0trailing = TRUE,
   right = TRUE,
   max.digits = getOption("Rmpfr.print.max.digits", 999L),
   exponent.plus = getOption("Rmpfr.print.exponent.plus", TRUE),
   ...)
toNum(from, rnd.mode = c('N','D','U','Z','A'))
.mpfr2d(from)
.mpfr2i(from)

mpfr2array(x, dim, dimnames = NULL, check = FALSE)

.mpfr2list(x, names = FALSE)

mpfrXport(x, names = FALSE)
mpfrImport(mx)

.mpfr_formatinfo(x)

.mpfr2exp(x)

.mpfr_erange(kind = c("Emin", "Emax"), names = TRUE)
.mpfr_erange_set(kind = c("Emin", "Emax"), value)
.mpfr_erange_kinds
.mpfr_erange_is_int()
.mpfr_maxPrec()
.mpfr_minPrec()

.mpfr_gmp_numbbits()

.mpfrVersion()

## Really Internal and low level, no error checking (for when you know ..)
.mpfr (x, precBits)
.mpfr.(x, precBits, rnd.mode)
.getSign(x)

.mpfr_negative(x)
.mpfr_sign(x)
bigq2mpfr(x, precB = NULL, rnd.mode = c("N", "D", "U", "Z", "A"))
.bigz2mpfr(x, precB = NULL, rnd.mode = c("N", "D", "U", "Z", "A"))

Arguments

x, from a typical, an R object of class "mpfr", or "mpfrArray", respectively. For getPrec(), any number-like R object, or NULL.

base (only when x is character) the base with which to represent numbers; base b must fulfill 2 ≤ b ≤ 62.

doNumeric logical indicating integer or double typed x should be accepted and a default precision be returned. Should typically be kept at default TRUE.

is.mpfr logical indicating if class(x) is already known to be "mpfr"; typically should be kept at default, NA.

bigq. for getPrec(), the precision to use for a big rational (class "bigq"); if not specified gives warning when used.

prec, precB, precBits a positive integer, or missing.

drop0trailing logical indicating if trailing "0"s should be omitted.

right logical indicating print()ing should right justify the strings; see print.default() to which it is passed.

digits, ... further arguments to print methods.

max.digits a number (possibly Inf) to limit the number of (mantissa) digits to be printed, simply passed to formatMpfr(). The default is finite to protect from printing very long strings which is often undesirable, notably in interactive use.

exponent.plus logical, simply passed to formatMpfr(). Was FALSE hardwired in Rmpfr versions before 0.8-0, and hence is allowed to be tweaked by an options() setting.

rnd.mode a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see details of mpfr.

dim, dimnames for "mpfrArray" construction.

check logical indicating if the mpfrArray construction should happen with internal safety check. Previously, the implicit default used to be true.

names (for .mpfr2list()) logical or character vector, indicating if the list returned should have names. If character, it specifies the names; if true, the names are set to format(x).

mxp an "mpfrXport" object, as resulting from mpfrXport().

kind a character string or vector, specifying the kind of "erange" value; must be an element of .mpfr_erange_kinds, i.e., one of "Emin", "Emax", "min.emin", "max.emin", "min.emax", "max.emax".

value numeric, for .mpfr_erange_set() one number per kind. Must be in range specified by the *."emin" and *."emax" erange values.
Details

The print method is currently built on the format method for class mpfr. This, currently does not format columns jointly which leads to suboptimally looking output. There are plans to change this.

Note that formatMpfr() which is called by print() (or show() or R’s implicit printing) uses max.digits = Inf, differing from our print()’s default on purpose. If you do want to see the full accuracy even in cases it is large, use options(Rmpfr.print.max.digits = Inf) or (. = 1e7), say.

The .mpfr_erange* functions (and variable) allow to query and set the allowed range of values for the base-2 exponents of "mpfr" numbers. See the examples below and GNU MPFR library documentation on the C functions mpfr_get_emin(), mpfr_set_emin()., mpfr_get_emin_min(), and mpfr_get_emin_max(), (and those four with '_emin' replaced by '_emax' above).

Value

getPrec(x) returns a integer vector of length one or the same length as x when that is positive, whereas getPrec(NULL) returns mpfr_default_prec(), see below.

If you need to change the precision of x, i.e., need something like “setPrec”, use roundMpfr().

.getPrec(x) is a simplified version of getPrec() which only works for "mpfr" objects x.

getD(x) is intended to be a fast version of x@.Data, and should not be used outside of lower level functions.

mpfr_default_prec() returns the current MPFR default precision, an integer. This is currently not made use of much in package Rmpfr, where functions have their own default precision where needed, and otherwise we’d rather not be dependent of such a global setting.

mpfr_default_prec(prec) sets the current MPFR default precision and returns the previous one; see above.

.mpfr_maxPrec() and (less interestingly) .mpfr_minPrec() give the maximal and minimal base-2 precision allowed in the current version of the MPFR library linked to by R package Rmpfr. The maximal precision is typically $2^{63}$, i.e.,

```
all.equal(.mpfr_maxPrec(), 2^63)
```

is typically true.

toNum(m) returns a numeric array or matrix, when m is of class "mpfrArray" or "mpfrMatrix", respectively. It should be equivalent to as(m, "array") or ... "matrix". Note that the slightly more general asNumeric() is preferred now. .mpfr2d() is similar to but simpler than toNum(), whereas .mpfr2i() is an analogue low level utility for as.integer(<mpfr>).

mpfr2array() a slightly more flexible alternative to dim(.) <- dd.

.mpfr2exp(x) returns the base-2 (integer valued) exponents of x, i.e., it is the R interface to MPFR C’s mpfr_get_exp(). The result is integer iff .mpfr_erange_is_int() is true, otherwise double. Note that the MPFR (4.0.1) manual says about mpfr_get_exp(): The behavior for NaN, infinity or zero is undefined.

.mpfr_erange_is_int() returns TRUE iff the .mpfr_erange(c("Emin","Emax")) range lies inside the range of R’s integer limits, i.e., has absolute values not larger than .Machine$integer.max ( = $2^{31} – 1).

.mpfr_erange_set() invisibly (see invisible()) returns TRUE iff the change was successful.
.mpfr_gmp_numbbits() returns the ‘GMP’ library “numb” size, which is either 32 or 64 bit (as integer, i.e., 64L or 32L). If it is not 64, you typically cannot enlarge the exponential range of mpfr numbers via .mpfr_erange(), see above.

.mpfrVersion() returns a string, the version of the ‘MPFR’ library we are linking to.

.mpfr_formatinfo(x) returns conceptually a subset of .mpfr2str()’s result, a list with three components

- **exp**: the base-2 exponents of x, identical to .mpfr2exp(x).
- **finite**: logical identical to is.finite(x).
- **is.0**: logical indicating if the corresponding x[i] is zero; identical to mpfrIs0(x).

(Note that .mpfr2str(x, ..., base)$exp is wrt base and is not undefined but ...)

.mpfr_sign(x) only works for mpfr objects, then identical to sign(x). Analogously, .mpfr_negative(x) is -x in that case. .getSign(x) is a low-level version of sign(x) returning -1 or +1, but not 0. Finally, .bigq2mpfr(x, ...) and .bigz2mpfr(x, ...) are fast ways to coerce bigz and bigq number objects (created by package gmp’s functionality) to our “mpfr” class.

**Note**

mpfrXport() and mpfrImport() are **experimental** and used to explore reported platform incompatibilities of saved and loaded “mpfr” objects between Windows and non-Windows platforms.

In other words, the format of the result of mpfrXport() and hence the mxp argument to mpfrImport() are considered internal, not part of the API and subject to change.

**See Also**

Start using mpfr(...), and compute with these numbers.

mpfrArray(x) is for numeric (“non-mpfr”) x, whereas mpfr2array(x) is for “mpfr” classed x, only.

**Examples**

```r
getPrec(as(c(1, pi), "mpfr")) # 128 for both

(opr <- mpfr_default_prec()) ## typically 53, the MPFR system default
stopifnot(opr == (oprec <- mpfr_default_prec(70)),
  70 == mpfr_default_prec())
## and reset it:
mpfr_default_prec(opr)
```

```
## Explore behavior of rounding modes 'rnd.mode':
x <- mpfr(10, 99)^512 # too large for regular (double prec. / numeric):
sapply(c("N", "D", "U", "Z", "A"), function(RM)
  sapply(list(-x, x), function(.) toNum(., RM)))
## N  D  U  Z  A
## -Inf -Inf -1.797693e+308 -1.797693e+308 -Inf
## Inf 1.797693e+308 Inf 1.797693e+308 Inf
```
## Printing of "MPFR" matrices is less nice than R's usual matrix printing:
m <- outer(c(1, 3.14, -1024.5678), c(1, 1e-3, 10, 100))
m[3,3] <- round(m[3,3])
m
mpfr(m, 50)

B6 <- mpfr2array(Bernoulli(1:6, 60), c(2,3),
                 dimnames = list(LETTERS[1:2], letters[1:3]))
B6

## Ranges of (base 2) exponents of MPFR numbers:
.mpfr_erange() # the currently active range of possible base 2 exponents:

## A factory fresh setting fulfills
.mpfr_erange(c("Emin","Emax")) == c(-1,1) * (2^30 - 1)

## There are more 'kind's, the latter 4 showing how you could change the first two:
.mpfr_erange_kinds
.mpfr_erange(.mpfr_erange_kinds)
eLimits <- .mpfr_erange(c("min.emin","max.emin", "min.emax", "max.emax"))

## Typically true in MPFR versions *iff* long is 64-bit, i.e. *not* on Windows
if(.Machine$sizeof.long == 8L) {
eLimits == c(-1,1, -1,1) * (2^62 - 1)
} else if(.Machine$sizeof.long == 4L) # on Windows
  eLimits == c(-1,1, -1,1) * (2^30 - 1)

## Looking at internal representation [for power users only!]:
i8 <- mpfr(-2:5, 32)
x4 <- mpfr(c(NA, NaN, -Inf, Inf), 32)
stopifnot(exprs = {
  identical(x4[1], x4[2])
  is.na(x4[1]) == x4[2] # <- was *wrong* in Rmpfr <= 0.9-4
  is.na(x4[2]) != x4[2] # (ditto)
  identical(x4 < i8[1:4], c(NA,NA, TRUE,FALSE))
  !is.finite(x4)
  identical(is.infinite(x4), c(FALSE,FALSE, TRUE,TRUE))
})

## The output of the following depends on the GMP "numb" size
## (32 bit vs. 64 bit), *and* additionally
## on sizeof.long (mostly non-Windows <-> Windows, see above):
str( .mpfr2list(i8) )
str( .mpfr2list(x4, names = TRUE) )

str(xp4 <- mpfrXport(x4, names = TRUE))
stopifnot(identical(x4, mpfrImport(mpfrXport(x4))),
          identical(i8, mpfrImport(mpfrXport(i8))))

## FIXME, need c(.), as dim(.) "get lost":
stopifnot(identical(c(B6), mpfrImport(mpfrXport(B6))))
Description

`mpfrVersion()` returns the version of the MPFR library which `Rmpfr` is currently linked to.

`c(x,y,...)` can be used to combine MPFR numbers in the same way as regular numbers IFF the first argument `x` is of class `mpfr`.

`mpfrIs0(.)` uses the MPFR library in the documented way to check if (a vector of) MPFR numbers are zero. It was called `mpfr.is.0` which is strongly deprecated now.

`.mpfr.is.whole(x)` uses the MPFR library in the documented way to check if (a vector of) MPFR numbers is integer valued. This is equivalent to `x == round(x)`, but not at all to `is.integer(as(x, "numeric"))`.

You should typically rather use (the "mpfr" method of the generic function) `is.whole(x)` instead. The former name `mpfr.is.integer` is deprecated now.

Usage

```r
mpfrVersion()
mpfrIs0(x)
```

```r
## S3 method for class 'mpfr'
c(...)
## S3 method for class 'mpfr'
diff(x, lag = 1L, differences = 1L, ...)
```

Arguments

- `x` : an object of class `mpfr`.
- `...` : for `diff`, further `mpfr` class objects or simple numbers (numeric vectors) which are coerced to `mpfr` with default precision of 128 bits.
- `lag, differences` : for `diff()`: exact same meaning as in `diff()`’s default method, `diff.default`.

Value

`mpfrIs0` returns a logical vector of length `length(x)` with values `TRUE` iff the corresponding `x[i]` is an MPFR representation of zero (0).

Similarly, `.mpfr.is.whole` and `is.whole` return a logical vector of length `length(x)`.

`mpfrVersion` returns an object of S3 class "numeric_version", so it can be used in comparisons.

The other functions return MPFR number (vectors), i.e., extending class `mpfr`. 
mpfrArray

See Also

str.mpfr for the str method. erf for special mathematical functions on MPFR.

The class description mpfr page mentions many generic arithmetic and mathematical functions for which "mpfr" methods are available.

Examples

mpfrVersion()

(x <- c(Const("pi", 64), mpfr(-2:2, 64)))
mpfrIs0(x) # one of them is
x[mpfrIs0(x)] # but it may not have been obvious..
str(x)

x <- rep(-2:2, 5)
stopifnot(is.whole(mpfr(2, 500) ^ (1:200)),
  all.equal(diff(x), diff(as.numeric(x))))

mpfrArray

Construct "mpfrArray" almost as by `array()`

Description

Utility to construct an R object of class mpfrArray, very analogously to the numeric array function.

Usage

mpfrArray(x, precBits, dim = length(x), dimnames = NULL, rnd.mode = c("N","D","U","Z","A"))

Arguments

x numeric(like) vector, typically of length prod(dim) or shorter in which case it is recycled.
precBits a number, the maximal precision to be used, in bits; i.e., 53 corresponds to double precision. Must be at least 2.
dim the dimension of the array to be created, that is a vector of length one or more giving the maximal indices in each dimension.
dimnames either NULL or the names for the dimensions. This is a list with one component for each dimension, either NULL or a character vector of the length given by dim for that dimension.
rnd.mode a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see details of mpfr.

Value

an object of class "mpfrArray", specifically "mpfrMatrix" when length(dim) == 2.
See Also

`mpfr`, `array`, `asNumeric()` as “inverse” of `mpfrArray()`, to get back a numeric array.

`mpfr2array(x)` is for “mpfr” classed `x`, only, whereas `mpfrArray(x)` is for numeric ("non-mpfr") `x`.

Examples

```r
## preallocating is possible here too
ma <- mpfrArray(NA, prec = 80, dim = 2:4)
validObject(A2 <- mpfrArray(1:24, prec = 64, dim = 2:4))

## recycles, gives an "mpfrMatrix" and dimnames :
mat <- mpfrArray(1:5, 64, dim = c(5,3), dimnames=list(NULL, letters[1:3]))
mat
asNumeric(mat)
stopifnot(identical(asNumeric(mat),
    matrix(1:5 +0, 5,3, dimnames=dimnames(mat))))

## Testing the apply() method :
apply(mat, 2, range)
apply(A2, 1:2, range)
apply(A2, 2:3, max)
(fA2 <- apply(A2, 2, fivenum))
a2 <- as(A2, "array")
stopifnot(identical(as(apply(A2, 2, range), "matrix") ==
    apply(a2, 2, range),
    all.equal(fA2, apply(a2, 2, fivenum)),
    all.equal(apply(A2, 2, quantile),
        apply(a2, 2, quantile)),
    all.equal(A2, apply(A2, 2:3, identity) -> aA2, check.attributes=FALSE),
    dim(A2) == dim(aA2))
```

mpfrMatrix

Classes "mpfrMatrix" and "mpfrArray"

Description

The classes "mpfrMatrix" and "mpfrArray" are, analogously to the base functions and classes simply "numbers" of class mpfr with an additional Dim and Dimnames slot.

Objects from the Class

Objects should typically be created by `mpfrArray()`, but can also be created by `new("mpfrMatrix", ...)` or `new("mpfrArray", ...)`, or also by `t(x)`, `dim(x) <- dd`, or `mpfr2array(x, dim=dd)` where `x` is a mpfr "number vector".

A (slightly more flexible) alternative to `dim(x) <- dd` is `mpfr2array(x, dd, dimnames)."
Slots

.Data: as for the mpfr class, a "list" of mpfr numbers.
Dim: of class "integer", specifying the array dimension.
Dimnames: of class "list" and the same length as Dim, each list component either NULL or a character vector of length Dim[j].

Extends

Class "mpfrMatrix" extends "mpfrArray", directly.
Class "mpfrArray" extends class "mpfr", by class "mpfrArray", distance 2; class "list", by class "mpfrArray", distance 3; class "vector", by class "mpfrArray", distance 4.

Methods

Arith signature(e1 = "mpfr", e2 = "mpfrArray"): ...
Arith signature(e1 = "numeric", e2 = "mpfrArray"): ...
Arith signature(e1 = "mpfrArray", e2 = "mpfrArray"): ...
Arith signature(e1 = "mpfrArray", e2 = "mpfr"): ...
Arith signature(e1 = "mpfrArray", e2 = "numeric"): ...

as.vector signature(x = "mpfrArray", mode = "missing"): drops the dimension ‘attribute’, i.e., transforms x into a simple mpfr vector. This is an inverse of t(.) or dim(.) <- * on such a vector.

atan2 signature(y = "ANY", x = "mpfrArray"): ...
atan2 signature(y = "mpfrArray", x = "mpfrArray"): ...
atan2 signature(y = "mpfrArray", x = "ANY"): ...
[ signature(x = "mpfrArray", i = "ANY", j = "ANY", value = "ANY"): ...
[ signature(x = "mpfrArray", i = "ANY", j = "ANY", drop = "ANY"): ...
[ signature(x = "mpfrArray", i = "ANY", j = "missing", drop = "missing"): "mpfrArray"s can be subset ("indexed") as regular R arrays.

%*% signature(x = "mpfr", y = "mpfrMatrix"): Compute the matrix/vector product xy when the dimensions (dim) of x and y match. If x is not a matrix, it is treated as a 1-row or 1-column matrix (aka "row vector" or "column vector") depending on which one makes sense, see the documentation of the base function %*%.

%*% signature(x = "mpfr", y = "Mnumber"): method definition for cases with one mpfr and any "number-like" argument are to use MPFR arithmetic as well.

%*% signature(x = "mpfrMatrix", y = "mpfrMatrix").
%*% signature(x = "mpfrMatrix", y = "mpfr"), etc. Further method definitions with identical semantic.

crossprod signature(x = "mpfr", y = "missing"): Computes x'x, i.e., t(x) %*% x, typically more efficiently.

crossprod signature(x = "mpfr", y = "mpfrMatrix"): Computes x'y, i.e., t(x) %*% y, typically more efficiently.
mpfrMatrix

crossprod signature(x = "mpfrMatrix", y = "mpfrMatrix"): ...
crossprod signature(x = "mpfrMatrix", y = "mpfr"): ...
tcrossprod signature(x = "mpfr", y = "missing"): Computes $xx'$, i.e., $x \times t(x)$, typically more efficiently.
tcrossprod signature(x = "mpfrMatrix", y = "mpfrMatrix"): Computes $xy'$, i.e., $x \times t(y)$, typically more efficiently.
tcrossprod signature(x = "mpfrMatrix", y = "mpfr"): ...
tcrossprod signature(x = "mpfr", y = "mpfrMatrix"): ...
coerce signature(from = "mpfrArray", to = "array"): coerces from to a numeric array of the same dimension.
coerce signature(from = "mpfrArray", to = "vector"): as for standard arrays, this “drops” the dim (and dimnames), i.e., returns an mpfr vector.
Compare signature(e1 = "mpfr", e2 = "mpfrArray"): ...
Compare signature(e1 = "numeric", e2 = "mpfrArray"): ...
Compare signature(e1 = "mpfrArray", e2 = "mpfr"): ...
Compare signature(e1 = "mpfrArray", e2 = "numeric"): ...
dim signature(x = "mpfrArray"): ...
dimnames< signature(x = "mpfrArray"): ...
dimnames signature(x = "mpfrArray"): ...
show signature(object = "mpfrArray"): ...
sign signature(x = "mpfrArray"): ...
norm signature(x = "mpfrMatrix", type = "character"): computes the matrix norm of x, see norm or the one in package Matrix.
t signature(x = "mpfrMatrix"): transpose the mpfrMatrix.
aperm signature(a = "mpfrArray"): aperm(a, perm) is a generalization of t(.) to permute the dimensions of an mpfrArray; it has the same semantics as the standard aperm() method for simple R arrays.

Author(s)

Martin Maechler

See Also

mpfrArray, also for more examples.

Examples

showClass("mpfrMatrix")

validObject(mm <- new("mpfrMatrix"))
validObject(aa <- new("mpfrArray"))

v6 <- mpfr(1:6, 128)
m6 <- new("mpfrMatrix", v6, Dim = c(2L, 3L))
validObject(m6)
m6
which(m6 == 3, arr.ind = TRUE) # |--> (1, 2)
## Coercion back to "vector": Both of these work:
stopifnot(identical(as(m6, "mpfr"), v6),
  identical(as.vector(m6), v6)) # < but this is a "coincidence"
S2 <- m6[,-3] # 2 x 2
S3 <- rbind(m6, c(1:2,10)); s3 <- asNumeric(S3)
det(S2)
str(determinant(S2))
det(S3)
stopifnot(all.equal(det(S2), det(asNumeric(S2)), tol=1e-15),
  all.equal(det(S3), det(s3), tol=1e-15))
## 2-column matrix indexing and replacement:
(sS <- S3[i2 <- cbind(1:2, 2:3)])
stopifnot(identical(asNumeric(sS), s3[i2]))
C3 <- S3; c3 <- s3
C3[i2] <- 10:11
c3[i2] <- 10:11
stopifnot(identical(asNumeric(C3), c3))
AA <- new("mpfrArray", as.vector(cbind(S3, -S3)), Dim=c(3L,3:2))
stopifnot(identical(AA[,,1], S3), identical(AA[,,2], -S3))
aa <- asNumeric(AA)
i3 <- cbind(3:1, 1:3, c(2L, 1:2))
ii3 <- Rmpfr:::.mat2ind(i3, dim(AA), dimnames(AA))
stopifnot(aa[i3] == new("mpfr", getD(AA)[ii3]))
stopifnot(identical(aa[i3], asNumeric(AA[i3])))
CA <- AA; ca <- aa
cai3 <- ca[i3] ^ 3
CA[i3] <- CA[i3] ^ 3
## scale():
S2. <- scale(S2)
stopifnot(all.equal(abs(as.vector(S2.)), rep(sqrt(1/mpfr(2, 128)), 4),
  tol = 1e-30))
## norm():
norm(S2)
stopifnot(identical(norm(S2), norm(S2, "1")),
  norm(S2, "1") == 6,
  norm(S2, "M") == 4,
  abs(norm(S2, "F") - 5.477225575051661) < 1e-15)
Description
determinant(x, ...) computes the determinant of the mpfr square matrix x. May work via coercion to "numeric", i.e., compute determinant(asNumeric(x), logarithm), if asNumeric is true, by default, if the dimension is larger than three. Otherwise, use precision precBits for the "accumulator" of the result, and use the recursive mathematical definition of the determinant (with computational complexity $n!$, where $n$ is the matrix dimension, i.e., very inefficient for all but small matrices!)

Usage
## S3 method for class 'mpfrMatrix'
determinant(x, logarithm = TRUE, 
asNumeric = (d[1] > 3), precBits = max(.getPrec(x)), ...)

Arguments
  x an mpfrMatrix object of square dimension.
  logarithm logical indicating if the log of the absolute determinant should be returned.
  asNumeric logical .. if rather determinant(asNumeric(x), ...) should be computed.
  precBits the number of binary digits for the result (and the intermediate accumulations).
  ... unused (potentially further arguments passed to methods).

Value
as determinant(), an object of S3 class "det", a list with components
  modulus the (logarithm of) the absolute value (abs) of the determinant of x.
  sign the sign of the determinant.

Author(s)
  Martin Maechler

See Also
determinant in base R, which relies on a fast LU decomposition. mpfrMatrix

Examples
m6 <- mpfrArray(1:6, prec=128, dim = c(2L, 3L))
m6
S2 <- m6[,3] # 2 x 2
S3 <- rbind(m6, c(1:2,10))
det(S2)
str(determinant(S2))
det(S3)
stopifnot(all.equal(det(S2), det(asNumeric(S2)), tolerance=1e-15),
  all.equal(det(S3), det(asNumeric(S3)), tolerance=1e-15))
**optimizeR**  
*High Precision One-Dimensional Optimization*

**Description**

optimizeR searches the interval from lower to upper for a minimum of the function \( f \) with respect to its first argument.

**Usage**

```r
optimizeR(f, lower, upper, ..., tol = 1e-20,
method = c("Brent", "GoldenRatio"),
maximum = FALSE,
precFactor = 2.0, precBits = -log2(tol) * precFactor,
maxiter = 1000, trace = FALSE)
```

**Arguments**

- **f**  
  the function to be optimized. \( f(x) \) must work *in Rmpfr arithmetic* for optimizer() to make sense. The function is either minimized or maximized over its first argument depending on the value of maximum.

- **...**  
  additional named or unnamed arguments to be passed to \( f \).

- **lower**  
  the lower end point of the interval to be searched.

- **upper**  
  the upper end point of the interval to be searched.

- **tol**  
  the desired accuracy, typically higher than double precision, i.e., \( tol < 2e-16 \).

- **method**  
  character string specifying the optimization method.

- **maximum**  
  logical indicating if \( f() \) should be maximized or minimized (the default).

- **precFactor**  
  only for default precBits construction: a factor to multiply with the number of bits directly needed for tol.

- **precBits**  
  number of bits to be used for mpfr numbers used internally.

- **maxiter**  
  maximal number of iterations to be used.

- **trace**  
  integer or logical indicating if and how iterations should be monitored; if an integer \( k \), print every \( k \)-th iteration.

**Details**

"Brent": Brent(1973)'s simple and robust algorithm is a hybrid, using a combination of the golden ratio and local quadratic ("parabolic") interpolation. This is the same algorithm as standard R's optimize(), adapted to high precision numbers.

In smooth cases, the convergence is considerably faster than the golden section or Fibonacci ratio algorithms.

"GoldenRatio": The golden ratio method, aka ‘golden-section search’ works as follows: from a given interval containing the solution, it constructs the next point in the golden ratio between the interval boundaries.
Value

A list with components minimum (or maximum) and objective which give the location of the minimum (or maximum) and the value of the function at that point; iter specifying the number of iterations, the logical convergence indicating if the iterations converged and estim.prec which is an estimate or an upper bound of the final precision (in \(x\)). method the string of the method used.

Author(s)

"GoldenRatio" is based on Hans Werner Borchers' golden_ratio (package pracma); modifications and "Brent" by Martin Maechler.

See Also

R's standard optimize; for multivariate optimization, Rmpfr's hjkMpfr(); for root finding, Rmpfr's unirootR.

Examples

```r
## The minimum of the Gamma (and lgamma) function (for x > 0):
Gmin <- optimizeR(gamma, .1, 3, tol = 1e-50)
str(Gmin, digits = 8)
## high precision chosen for "objective"; minimum has "estim.prec" = 1.79e-50
Gmin[c("minimum","objective")]
## it is however more accurate to 59 digits:
asNumeric(optimizeR(gamma, 1, 2, tol = 1e-100)$minimum - Gmin$minimum)

iG5 <- function(x) -exp(-(x-5)^2/2)
curve(iG5, 0, 10, 200)
o.dp <- optimize (iG5, c(0, 10)) #-> 5 of course
o.M.gs <- optimizeR(iG5, 0, 10, method="Golden")
o.M.Br <- optimizeR(iG5, 0, 10, method="Brent", trace=TRUE)
oM.gs$min ; oM.gs$iter
oM.Br$min ; oM.Br$iter
(doExtras <- Rmpfr:::doExtras())
if(doExtras) {## more accuracy {takes a few seconds}
oM.gs <- optimizeR(iG5, 0, 10, method="Golden", tol = 1e-70)
oM.Br <- optimizeR(iG5, 0, 10, tol = 1e-70)
}
rbind(Golden = c(err = as.numeric(oM.gs$min -5), iter = oM.gs$iter),
       Brent = c(err = as.numeric(oM.Br$min -5), iter = oM.Br$iter))

## ==> Brent is orders of magnitude more efficient !

## Testing on the sine curve with 40 correct digits:
sol <- optimizeR(sin, 2, 6, tol = 1e-40)
str(sol)
sol <- optimizeR(sin, 2, 6, tol = 1e-50,
                 precFactor = 3.0, trace = TRUE)
pi.. <- 2*sol$min/3
print(pi.., digits=51)
```
stopifnot(all.equal(pi.., Const("pi", 256), tolerance = 10*1e-50))

if(doExtras) { # considerably more expensive

## a harder one:
  f.sq <- function(x) sin(x-2)^4 + sqrt(pmax(0,(x-1)*(x-4)))*(x-2)^2
  curve(f.sq, 0, 4.5, n=1000)
  msq <- optimizeR(f.sq, 0, 5, tol = 1e-50, trace=5)
  str(msq) # ok
  stopifnot(abs(msq$minimum - 2) < 1e-49)

## find the other local minimum: -- non-smooth ==> Golden ratio -section is used
  msq2 <- optimizeR(f.sq, 3.5, 5, tol = 1e-50, trace=10)
  stopifnot(abs(msq2$minimum - 4) < 1e-49)

## and a local maximum:
  msq3 <- optimizeR(f.sq, 3, 4, maximum=TRUE, trace=2)
  stopifnot(abs(msq3$maximum - 3.57) < 1e-2)
}

##----- "impossible" one to get precisely ------------------------

ff <- function(x) exp(-1/(x-8)^2)
  curve(exp(-1/(x-8)^2), -3, 13, n=1001)
  (opt. <- optimizeR(function(x) exp(-1/(x-8)^2), -3, 13, trace = 5))
## -> close to 8 (but not very close!)
  ff(opt.$minimum) # gives 0
  if(doExtras) {
    ## try harder ... in vain ..
    str(opt1 <- optimizeR(ff, -3,13, tol = 1e-60, precFactor = 4))
    print(opt1$minimum, digits=20)
    ## still just 7.99998038 or 8.000036655 (depending on method)
  }

---

### pbetaI

**Accurate Incomplete Beta / Beta Probabilities For Integer Shapes**

**Description**

For integers $a$, $b$, $I_x(a,b)$ aka pbeta(x, a,b) is a polynomial in $x$ with rational coefficients, and hence arbitrarily accurately computable.

TODO (not yet): It’s sufficient for one of $a,b$ to be integer such that the result is a finite sum (but the coefficients will no longer be rational, see Abramowitz and Stegun, 26.5.6 and .7, p.944).

**Usage**
pbetaI(q, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE,
precBits = NULL,
useRational = !log.p && !is.mpfr(q) && is.null(precBits) && int2,
rnd.mode = c("N","D","U","Z","A"))

Arguments

q called x, above; vector of quantiles, in \([0, 1]\); can be numeric, or of class "mpfr"
or also "bigq" ("big rational" from package gmp); in the latter case, if log.p = FALSE as by default, all computations are exact, using big rational arithmetic.

shape1, shape2 the positive Beta “shape” parameters, called \(a, b\), above. Must be integer valued for this function.

ncp unused, only for compatibility with pbeta, must be kept at its default, 0.

lower.tail logical; if TRUE (default), probabilities are \(P[X \leq x]\), otherwise, \(P[X > x]\).

log.p logical; if TRUE, probabilities p are given as log(p).

precBits the precision (in number of bits) to be used in sumBinomMpfr().

useRational optional logical, specifying if we should try to do everything in exact rational arithmetic, i.e, using package gmp functionality only, and return bigq numbers instead of mpfr numbers.

rnd.mode a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see mpfr.

Value

an "mpfr" vector of the same length as q.

Note

For upper tail probabilities, i.e., when lower.tail=FALSE, we may need large precBits, because the implicit or explicit \(1 – P\) computation suffers from severe cancellation.

Author(s)

Martin Maechler

References

https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain.

See Also

pbeta, sumBinomMpfr chooseZ.
Examples

```r
x <- (0:12)/16 # not all the way up ..
a <- 7; b <- 788

p. <- pbetaI(x, a, b) ## a bit slower:
system.time(
  pp <- pbetaI(x, a, b, precBits = 2048)
) # 0.23 -- 0.50 sec
## Currently, the lower.tail=FALSE are computed "badly":
lp <- log(pp) ## = pbetaI(x, a, b, log.p=TRUE)
lIp <- log1p(-pp) ## = pbetaI(x, a, b, lower.tail=FALSE, log.p=TRUE)
Ip <- 1 - pp ## = pbetaI(x, a, b, lower.tail=FALSE)

if(Rmpfr:::doExtras()) { ## somewhat slow
  stopifnot(
    all.equal(lp, pbetaI(x, a, b, precBits = 2048, log.p=TRUE)),
    all.equal(lIp, pbetaI(x, a, b, precBits = 2048, lower.tail=FALSE, log.p=TRUE),
      tol = 1e-230),
    all.equal(Ip, pbetaI(x, a, b, precBits = 2048, lower.tail=FALSE))
  )
}

rErr <- function(approx, true, eps = 1e-200) {
  true <- as.numeric(true) # for "mpfr"
  ifelse(Mod(true) >= eps,
    ## relative error, catching '-Inf' etc :
    ifelse(true == approx, 0, 1 - approx / true),
    ## else: absolute error (e.g. when true=0)
    true - approx)
}

rErr(pbeta(x, a, b), pp)
rErr(pbeta(x, a, b, lower=FALSE), Ip)
rErr(pbeta(x, a, b, log = TRUE), lp)
rErr(pbeta(x, a, b, lower=FALSE, log = TRUE), lIp)

a.EQ <- function(..., tol=1e-15) all.equal(..., tolerance=tol)
stopifnot(
  a.EQ(pp, pbeta(x, a, b)),
  a.EQ(lp, pbeta(x, a, b, log.p=TRUE)),
  a.EQ(lIp, pbeta(x, a, b, lower.tail=FALSE, log.p=TRUE)),
  a.EQ(Ip, pbeta(x, a, b, lower.tail=FALSE))
)

## When 'q' is a bigrational (i.e., class "bigq", package 'gmp'), everything
## is computed *exactly* with bigrational arithmetic:
(q4 <- as.bigq(1, 2^(0:4)))
pb4 <- pbetaI(q4, 10, 288, lower.tail=FALSE)
stopifnot( is.bigq(pb4) )
mpb4 <- as(pb4, "mpfr")
mpb4[1:2]
getPrec(mp4) # 128 349 1100 1746 2362
```
(pb. <- pbeta(asNumeric(q4), 10, 288, lower.tail=FALSE))
stopifnot(mpb4[1] == 0,
  all.equal(mpb4, pb., tol=4e-15))
$qbetaI. <- function(p, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE,
  precBits = NULL, rnd.mode = c("N", "D", "U", "Z", "A"),
  tolerance = 1e-20, ...)
{
  if(is.na(a <- as.integer(shape1))) stop("a = shape1 is not coercable to finite integer")
  if(is.na(b <- as.integer(shape2))) stop("b = shape2 is not coercable to finite integer")
  unirootR(function(q) pbetaI(q, a, b, lower.tail=lower.tail, log.p=log.p,
    precBits=precBits, rnd.mode=rnd.mode) - p,
    interval = if(log.p) c(-double.xmax, 0) else 0:1,
    tol = tolerance, ...)
} # end{qbetaI}

(p <- 1 - mpfr(1,128)/20) # 'p' must be high precision
q95.1.3 <- qbetaI.(p, 1,3, tolerance = 1e-29) # -> ~29 digits accuracy
str(q95.1.3) ; roundMpfr(q95.1.3$root, precBits = 29 * log2(10))
## relative error is really small:
(relE <- asNumeric(1 - pbetaI(q95.1.3$root, 1,3) / p))
stopifnot(abs(relE) < 1e-28)

---

pmax

Parallel Maxima and Minima

Description

Returns the parallel maxima and minima of the input values.

The functions pmin and pmax have been made S4 generics, and this page documents the “... method for class "mNumber"", i.e., for arguments that are numeric or from class "mpfr".

Usage

pmax(..., na.rm = FALSE)
pmin(..., na.rm = FALSE)

Arguments

... numeric or arbitrary precision numbers (class mpfr).
na.rm a logical indicating whether missing values should be removed.

Details

See pmax, the documentation of the base functions, i.e., default methods.

Value

vector-like, of length the longest of the input vectors; typically of class mpfr, for the methods here.
Methods

... = "ANY"  the default method, really just base::pmin or base::pmax, respectively.
... = "mNumber"  the method for mpfr arguments, mixed with numbers; designed to follow the
same semantic as the default method.

See Also

The documentation of the base functions, pmin and pmax; also min and max; further,
range (both min and max).

Examples

(pm <- pmin(1.35, mpfr(0:10, 77)))
stopifnot(pm == pmin(1.35, 0:10))

qnormI

Gaussian / Normal Quantiles qnorm() via Inversion

Description

Compute Gaussian or Normal Quantiles qnorm(p, *) via inversion of our “mpfr-ified” arbitrary
accurate pnorm(), using our unirootR() root finder.

Usage

qnormI(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE,
trace = 0, verbose = as.logical(trace),
tol,
useMpfr = any(prec > 53),
give.full = FALSE,
...)

Arguments

p  vector of probabilities.
mean  vector of means.
sd  vector of standard deviations.
log.p  logical; if TRUE, probabilities p are given as log(p).
lower.tail  logical; if TRUE (default), probabilities are \( P[X \leq x] \) otherwise, \( P[X > x] \).
trace  integer passed to unirootR(). If positive, information about a search interval
extension will be printed to the console.
verbose  logical indicating if progress details should be printed to the console.
tol  optionally the desired accuracy (convergence tolerance); if missing or not finite,
it is computed as \( 2^{-pr} + 2 \) where the precision \( pr \) is basically \( \max(getPrec(p+mean+sd)) \).
**useMpfr**  logical indicating if `mpfr` arithmetic should be used.

**give.full**  logical indicating if the full result of `unirootR()` should be returned (when applicable).

...  optional further arguments passed to `unirootR()` such as `maxiter`, `verbDigits`, `check.conv`, `warn.no.convergence`, and `epsC`.

**Value**

If `give.full` is true, return a [list](https://www.r-project.org/doc/licenses/stdlib/Lib/licenses.html) say `r`, of `unirootR()` results, with `length(r) == length(p)`.

Otherwise, return a “numeric vector” like `p`, e.g., of class "mpfr" when `p` is.

**Author(s)**

Martin Maechler

**See Also**

Standard R’s `qnorm`.

**Examples**

```r
doX <- Rmpfr:::doExtras() # slow parts only if(doX)
cat("doExtras: ", doX, "\n")
p <- (0:32)/32
lp <- -c(1000, 500, 200, 100, 50, 20:1, 2^-(1:8))
if(doX) {
  tol1 <- 2.3e-16
tolM <- 1e-20
tolRIlog <- 4e-14
} else { # use one more than a third of the points:
ip <- c(TRUE,FALSE, rep_len(c(TRUE,FALSE,FALSE), length(p)-2L))
p <- p[ip]
lp <- lp[ip]
tol1 <- 1e-9
tolM <- 1e-12
tolRIlog <- 25*tolM
}

f.all.eq <- function(a,b)
  sub("Mean relative difference: *, \", '', format(all.equal(a, b, tol=0)))
for(logp in c(FALSE,TRUE)) {
p <- if(logp) lp else p
mp <- mpfr(pp, precBits = if(doX) 80 else 64) # precBits = 128 gave "the same" as 80
for(l.tail in c(FALSE,TRUE)) {
  qn <- qnorm (pp, lower.tail = 1.tail, log.p = logp)
  qnI <- qnormI(pp, lower.tail = 1.tail, log.p = logp, tol = tol)
  qnM <- qnormI(mp, lower.tail = 1.tail, log.p = logp, tol = tolM)
cat(sprintf("Accuracy of qnorm(*, lower.t=%-5s, log.p=%-5s): %s || qnI: %s\n", l.tail, logp, 1.tail, logp, f.all.eq(qnM, qn ), f.all.eq(qnM, qnI)))
}
}
tolRI <- 1e-12
tolM <- 1e-10

for(l.tail in c(FALSE,TRUE)) {
  qn <- qnorm (pp, lower.tail = 1.tail, log.p = logp)
  qnI <- qnormI(pp, lower.tail = 1.tail, log.p = logp, tol = tol)
  qnM <- qnormI(mp, lower.tail = 1.tail, log.p = logp, tol = tolM)
cat(sprintf("Accuracy of qnorm(*, lower.t=%-5s, log.p=%-5s): %s || qnI: %s\n", l.tail, logp, 1.tail, logp, f.all.eq(qnM, qn ), f.all.eq(qnM, qnI)))
}
```


all.equal(qn, qnI, tol = if(logp) tolRIlog else 4*tol1)
all.equal(qnM, qnI, tol = tol1)
}

## useMpfr, using mpfr() :
if(doX) {
p2 <- 2^-c(1:27, 5*(6:20), 20*(6:15))
e2 <- 88
} else {
p2 <- 2^-c(1:2, 7, 77, 177, 307)
e2 <- 60
}

system.time( pn2 <- pnorm(qnormI(mpfr(p2, e2))) ) # 4.1 or 0.68
all.equal(p2, pn2, tol = 0) # 5.48e-29 // 5.2e-18
2^-e2

stopifnot(all.equal(p2, pn2, tol = 6 * 2^-e2)) # '4 *' needed

## Boundary -- from limits in mpfr maximal exponent range!
## 1) Use maximal ranges:
(old_eranges <- .mpfr_erange()) # typically -/* 2^30
(myERng <- (1-2^-52) * .mpfr_erange(c("min.emin","max.emax")))

(doIncr <- !isTRUE(all.equal(unname(myERng), unname(old_eranges)))) # ==>
## TRUE only if long is 64-bit, i.e., *not* on Windows
if(doIncr) .mpfr_erange_set(value = myERng)

log2(abs(.mpfr_erange()))# 62 62 if(doIncr) i.e. not on Windows
(lrgOK <- all(log2(abs(.mpfr_erange())) >= 62)) # FALSE on Windows

## The largest quantile for which our mpfr-ized qnorm() does *NOT* underflow :
cM <- if(doX) { "2528468770.342934368107681591972815143739328158515856314908759369469064" }
## |--> 0 on Windows {limited erange}; otherwise and if(doX) :
## 7.648906825456998451356334684958984619457083435832560693304396661634460003e-1388255822130839040

if(lrgOK) withAutoprint({

try( qnormI(pM) ) ## Error: lower < upper not fulfilled (evt. TODO)
## but this works
print(qnI <- qnormI(log(pM), log.p=TRUE)) # -2528468770.34329343681
all.equal(-qM, qnI, tol = 0) # << show how close; seen 1.084202e-19
stopifnot( all.equal(-qM, qnI, tol = 1e-18) )
})
if(doX) ## Show how bad it is (currently ca. 220 iterations, and then *wrong*)
str(qnormI(round(log(pM)), log.p=TRUE, trace=1, give.full = TRUE))
if(requireNamespace("DPQ")){
  new("mpfr", as(DPQ::qnormR(pM, trace=1), "mpfr")) # as(*, "mpfr") also works for +/- Inf
  # qnormI(p= 0, m=0, s=1, l.t.= 1, log= 0): q = -0.5
  # somewhat close to 0 or 1: r := sqrt(-lp) = 1.7879e+09
  # r > 5, using rational form R_3(t), for t=1.787897e+09 -- that is *not* accurate
  # [1] -94658744.36929586546046206270...........
}
## reset to previous status if needed
if(doIncr) .mpfr_erange_set( , old_eranges)

---

Rmpfr-workarounds       Base Functions etc, as an Rmpfr version

Description

Functions from base etc which need a copy in the Rmpfr namespace so they correctly dispatch.

Usage

outer(X, Y, FUN = "*", ...)

Arguments

X, Y, FUN, ... See base package help: outer.

See Also

outer.

Examples

outer(1/mpfr(1:10, 70), 0:2)
Rounding to binary bits, not decimal digits. Closer to the number representation, this also allows to increase or decrease a number’s precBits. In other words, it acts as setPrec(), see getPrec().

Usage

roundMpfr(x, precBits, rnd.mode = c("N","D","U","Z","A"))

Arguments

- x: an mpfr number (vector)
- precBits: integer specifying the desired precision in bits.
- rnd.mode: a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see mpfr.

Value

an mpfr number as x but with the new ‘precBits’ precision

See Also

The mpfr class group method Math2 implements a method for round(x, digits) which rounds to decimal digits.

Examples

\[(p1 <- Const("pi", 100)) \# 100 bit prec\]

roundMpfr(p1, 120) \# 20 bits more, but "random noise"

Const("pi", 120) \# same "precision", but really precise

Apply a Function over a "mpfr" Vector

Users may be disappointed to note that sapply() or vapply() typically do not work with "mpfr" numbers.

This is a simple (but strong) approach to work around the problem, based on lapply().

\[
\begin{align*}
\text{sapplyMpfr} & \quad Rounding to Binary bits, "mpfr-internally" \\
\text{Description} & \quad \text{Rounding to binary bits, not decimal digits. Closer to the number representation, this also allows to increase or decrease a number's precBits. In other words, it acts as setPrec(), see getPrec().} \\
\text{Usage} & \quad \text{roundMpfr(x, precBits, rnd.mode = c("N","D","U","Z","A"))} \\
\text{Arguments} & \quad \begin{align*}
& \text{x: an mpfr number (vector)} \\
& \text{precBits: integer specifying the desired precision in bits.} \\
& \text{rnd.mode: a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see mpfr.} \\
\text{Value} & \quad \text{an mpfr number as x but with the new 'precBits' precision} \\
\text{See Also} & \quad \text{The mpfr class group method Math2 implements a method for round(x, digits) which rounds to decimal digits.} \\
\text{Examples} & \quad \begin{align*}
& (p1 <- Const("pi", 100)) \# 100 bit prec \\
& \text{roundMpfr(p1, 120) \# 20 bits more, but "random noise"} \\
& \text{Const("pi", 120) \# same "precision", but really precise} \\
\end{align*}
\end{align*}
\]
sapplyMpfr

Usage

sapplyMpfr(X, FUN, ..., drop_1_ = TRUE)

Arguments

X
  a vector, possibly of class "mpfr".

FUN
  a function returning an "mpfr" vector or even an "mpfrArray". May also be a function returning a numeric vector or array for numeric X, and which returns "mpfr(Array)" for an X argument inheriting from "mpfr".

...  
  further arguments passed to lapply, typically further arguments to FUN.

drop_1_  
  logical (with unusual name on purpose!) indicating if 1-column matrices ("mpfrMatrix") should be “dropped” to vectors ("mpfr"), the same as in base R’s own sapply. This has been implicitly FALSE in Rmpfr versions 0.8-5 to 0.8-9 (Oct 2021 to June 2022), accidentally. Since Rmpfr 0.9-0, this has been made an argument with default TRUE to be compatible by default with R’s sapply.

Details

In the case FUN(<length-1>) returns an array or "mpfrArray", i.e., with two or more dimensions, sapplyMpfr() returns an "mpfrArray": this is analogous to sapply(X, FUN, simplify = "array") (rather than the default sapply() behaviour which returns a matrix also when a higher array would be more “logical”.)

Value

an "mpfr" vector, typically of the same length as X.

Note

This may still not always work as well as sapply() does for atomic vectors. The examples seem to indicate that it typically does work as desired, since Rmpfr version 0.9-0.

If you want to transform back to regular numbers anyway, it maybe simpler and more efficient to use

```
res <- lapply(....)
sapply(res, asNumeric, simplify = "array")
```

instead of sapplyMpfr().

Author(s)

Martin Maechler

See Also

sapply, lapply, etc.
Examples
sapplyMpfr0 <- ## Originally, the function was simply defined as
    function (X, FUN, ...) new("mpfr", unlist(lapply(X, FUN, ...), recursive = FALSE))

(m1 <- sapply ( 3, function(k) (1:3)^k)) # 3 x 1 matrix (numeric)
(p1 <- sapplyMpfr(mpfr(3, 64), function(k) (1:3)^k))
stopifnot(m1 == p1, is(p1, "mpfrMatrix"), dim(p1) == c(3,1), dim(p1) == dim(m1))

k.s <- c(2, 5, 10, 20)
(mk <- sapply ( k.s, function(k) (1:3)^k)) # 3 x 4 " "
(pm <- sapplyMpfr(mpfr(k.s, 64), function(k) (1:3)^k))
stopifnot(mk == pm, is(pm, "mpfrMatrix"), dim(pm) == 3:4, 3:4 == dim(mk))
# was *wrongly* 4x3 in Rmpfr 0.8-x

(f5k <- function(k) outer(1:5, k+0:2, "^"))# matrix-valued
(mk5 <- sapply ( k.s, f5k)) # sapply()'s default; not "ideal"
(ak5 <- sapply ( k.s, f5k, simplify = "array") ) # what we want
(pm5 <- sapplyMpfr(mpfr(k.s, 64), f5k))
stopifnot(c(mk5) == c(ak5), ak5 == pm5, is(pm5, "mpfrArray"), is.array(ak5),
          dim(pm5) == dim(ak5), dim(pm5) == c(5,3, 4))

if(require("Bessel")) { # here X, is simple
    bI1 <- function(k) besselI.nuAsym(mpfr(1.31e9, 128), 10, expon.scaled=TRUE, k.max=k)
bImp1 <- sapplyMpfr (0:4, bI1, drop_= FALSE) # 1x5 mpfrMatrix -- as in DPQ 0.8-8
(bImp <- sapplyMpfr (0:4, bI1, drop_= TRUE) ) # 5 "mpfr" vector (by default)
bImp0 <- sapplyMpfr0(0:4, bI1) # 5-vector
stopifnot(identical(bImp, bImp0), bImp == bImp1,
          is(bImp, "mpfr"), is(bImp1, "mpfrMatrix"), dim(bImp1) == c(1, 5))
}

seqMpfr

Description
Generate ‘regular’, i.e., arithmetic sequences. This is in lieu of methods for seq (dispatching on all three of from, to, and by).

Usage
seqMpfr(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
          length.out = NULL, along.with = NULL, ...)

Arguments
from, to the starting and (maximal) end value (numeric or "mpfr") of the sequence.
by number (numeric or "mpfr"): increment of the sequence.
length.out desired length of the sequence. A non-negative number, which will be rounded up if fractional.
along.with take the length from the length of this argument.
... arguments passed to or from methods.
Details

see seq (default method in package base), whose semantic we want to replicate (almost).

Value

a ‘vector’ of class "mpfr", when one of the first three arguments was.

Author(s)

Martin Maechler

See Also

The documentation of the base function seq; mpfr

Examples

seqMpfr(0, 1, by = mpfr(0.25, prec=88))

seqMpfr(7, 3) # -> default prec.

Description

The str method for objects of class mpfr produces a bit more useful output than the default method str.default.

Usage

## S3 method for class 'mpfr'
str(object, nest.lev, internal = FALSE,
     give.head = TRUE, digits.d = 12, vec.len = NULL, drop@trailing=TRUE,
     width = getOption("width"), ...)

Arguments

object an object of class mpfr.

nest.lev for str(), typically only used when called by a higher level str().

internal logical indicating if the low-level internal structure should be shown; if true (not by default), uses str(object@.Data).

give.head logical indicating if the “header” should be printed.
digits.d  the number of digits to be used, will be passed \texttt{formatMpfr()} and hence \texttt{NULL} will use “as many as needed”, i.e. often too many. If this is a number, as per default, less digits will be used in case the precision (\texttt{getPrec(object)}) is smaller.

vec.len  the number of \texttt{elements} that will be shown. The default depends on the precision of \texttt{object} and \texttt{width} (since \texttt{Rmpfr 0.6-0}, it was 3 previously).

drop0trailing  \texttt{logical}, passed to \texttt{formatMpfr()} (with a different default here).

width  the (approximately) desired width of output, see \texttt{options(width = .)}.

...  further arguments, passed to \texttt{formatMpfr()}.

See Also

\texttt{.mpfr2list()} puts the internal structure into a \texttt{list}, and its help page documents many more (low level) utilities.

Examples

\begin{verbatim}
(x <- c(Const("pi", 64), mpfr(-2:2, 64)))
str(x)
str(list(pi = pi, x.mpfr = x))
str(x ^ 1000)
str(x ^ -1e4, digits=NULL) # full precision
str(x, internal = TRUE) # internal low-level (for experts)

uu <- Const("pi", 16)# unaccurate
str(uu) # very similar to just 'uu'
\end{verbatim}

\begin{verbatim}
sumBinomMpfr
\end{verbatim}

\begin{verbatim}
(Alternating) Binomial Sums via \texttt{Rmpfr}
\end{verbatim}

Description

Compute (alternating) binomial sums via high-precision arithmetic. If \( sBn(f, n) := \text{sumBinomMpfr}(n, f) \), (default alternating is true, and \( n0 = 0 \)),

\[
sBn(f, n) = \sum_{k=n0}^{n} (-1)^{n-k} \binom{n}{k} \cdot f(k) = \Delta^n f,
\]

see Details for the \( n \)-th forward difference operator \( \Delta^n f \). If alternating is false, the \((-1)^{n-k}\) factor is dropped (or replaced by 1) above.

Such sums appear in different contexts and are typically challenging, i.e., currently impossible, to evaluate reliably as soon as \( n \) is larger than around 50 − − 70.

Usage

\begin{verbatim}
sumBinomMpfr(n, f, n0 = 0, alternating = TRUE, precBits = 256,
  f.k = f(mpfr(k, precBits=precBits)))
\end{verbatim}
Arguments

n
upper summation index (integer).
f
_function_ to be evaluated at \( k \) for \( k \in n0:n \) (and which must return one value per \( k \)).
n0
lower summation index, typically 0 (= default) or 1.
alternating
logical indicating if the sum is alternating, see below.
precBits
the number of bits for MPFR precision, see mpfr.
f.k
can be specified instead of \( f \) and precBits, and must contain the equivalent of its default, \( f(\text{mpfr}(k, \text{precBits}=\text{precBits})) \).

Details

The alternating binomial sum \( sB(f, n) := \text{sumBinom}(n, f, n0 = 0) \) is equal to the \( n \)-th forward difference operator \( \Delta^n f \),

\[
sB(f, n) = \Delta^n f,
\]

where

\[
\Delta^n f = \sum_{k=0}^{n} (-1)^{n-k} \binom{n}{k} \cdot f(k),
\]

is the \( n \)-fold iterated forward difference \( \Delta f(x) = f(x+1) - f(x) \) (for \( x = 0 \)).

The current implementation might be improved in the future, notably for the case where \( sB(f, n) = \text{sumBinomMpfr}(n, f, *) \) is to be computed for a whole sequence \( n = 1, \ldots, N \).

Value

an mpfr number of precision precBits. \( s \). If alternating is true (as per default),

\[
s = \sum_{k=n0}^{n} (-1)^{k} \binom{n}{k} \cdot f(k),
\]

if alternating is false, the \((-1)^k\) factor is dropped (or replaced by 1) above.

Author(s)

Martin Maechler, after conversations with Christophe Dutang.

References


See Also

chooseMpfr, chooseZ from package gmp.
Examples

```r
## "naive" R implementation:
sumBinom <- function(n, f, n0=0, ...) {
  k <- n0:n
  sum( choose(n, k) * (-1)^(n-k) * f(k, ...))
}

## compute sumBinomMpfr(.) for a whole set of 'n' values:
sumBin.all <- function(n, f, n0=0, precBits = 256, ...) {
  N <- length(n)
  precBits <- rep(precBits, length = N)
  ll <- lapply(seq_len(N), function(i)
      sumBinomMpfr(n[i], f, n0=n0, precBits=precBits[i], ...))
  sapply(ll, as, "double")
}

sumBin.all.R <- function(n, f, n0=0, ...)
  sapply(n, sumBinom, f=f, n0=n0, ...)

n.set <- 5:80
system.time(res.R <- sumBin.all.R(n.set, f = sqrt)) ## instantaneous..
system.time(resMpfr <- sumBin.all (n.set, f = sqrt)) ## ~ 0.6 seconds
matplot(n.set, cbind(res.R, resMpfr), type = "l", lty=1,
    ylim = extendrange(resMpfr, f = 0.25), xlab = "n",
    main = "sumBinomMpfr(n, f = sqrt) vs. R double precision")
legend("topleft", leg=c("double prec.", "mpfr"), lty=1, col=1:2, bty = "n")
```

unirootR

One Dimensional Root (Zero) Finding – in pure R

Description

The function unirootR searches the interval from lower to upper for a root (i.e., zero) of the function f with respect to its first argument.

unirootR() is “clone” of uniroot(), written entirely in R, in a way that it works with mpfr-numbers as well.

Usage

```r
unirootR(f, interval, ..., lower = min(interval), upper = max(interval),
  f.lower = f(lower, ...), f.upper = f(upper, ...),
  extendInt = c("no", "yes", "downX", "upX"),
  trace = 0, verbose = as.logical(trace),
  verbDigits = max(3, min(20, -log10(tol)/2)),
  tol = .Machine$double.eps^0.25, maxiter = 1000L,
  check.conv = FALSE,
  warn.no.convergence = !check.conv,
  epsC = NULL)
```
Arguments

\( f \) the function for which the root is sought.

interval a vector containing the end-points of the interval to be searched for the root.

... additional named or unnamed arguments to be passed to \( f \)

lower, upper the lower and upper end points of the interval to be searched.

\( f.\)lower, \( f.\)upper the same as \( f(\)upper\) and \( f(\)lower\), respectively. Passing these values from the caller where they are often known is more economical as soon as \( f() \) contains non-trivial computations.

extendInt character string specifying if the interval \( c(\)lower, upper\) should be extended or directly produce an error when \( f() \) does not have differing signs at the endpoints. The default, "no", keeps the search interval and hence produces an error. Can be abbreviated.

trace integer number; if positive, tracing information is produced. Higher values giving more details.

verbose logical (or integer) indicating if (and how much) verbose output should be produced during the iterations.

verbDigits used only if \( \)verbose\( \) is true, indicates the number of digits numbers should be printed with, using \( \)format\( (., \)digits=verbDigits)\).

tol the desired accuracy (convergence tolerance).

maxiter the maximum number of iterations.

check.conv logical indicating whether non convergence should be caught as an error, notably non-convergence in maxiter iterations should be an error instead of a warning.

warn.no.convergence if set to FALSE there’s no warning about non-convergence. Useful to just run a few iterations.

epsC positive number or NULL in which case a smart default is sought. This should specify the “achievable machine precision” for the given numbers and their arithmetic.

The default will set this to \( .\)Machine\$double.\)eps for double precision numbers, and will basically use \( 2^{\)–\}(minimum(getPrec(f.\)lower), getPrec(f.\)upper)) \)when that works (as, e.g., for \( mpfr\)-numbers) otherwise.

This is factually a lower bound for the achievable lower bound, and hence, setting \( tol \) smaller than \( epsC \) is typically non-sensical and produces a warning.

Details

Note that arguments after \( ... \) must be matched exactly.

Either interval or both lower and upper must be specified: the upper endpoint must be strictly larger than the lower endpoint. The function values at the endpoints must be of opposite signs (or zero), for extendInt="no", the default. Otherwise, if extendInt="yes", the interval is extended on both sides, in search of a sign change, i.e., until the search interval \([l, u]\) satisfies \( f(l) \cdot f(u) \leq 0 \).

If it is known how \( f \) changes sign at the root \( x_0 \), that is, if the function is increasing or decreasing there, extendInt can (and typically should) be specified as "upX" (for "upward crossing") or
“downX”, respectively. Equivalently, define $S := \pm 1$, to require $S = \text{sign}(f(x_0 + \epsilon))$ at the solution. In that case, the search interval $[l, u]$ possibly is extended to be such that $S \cdot f(l) \leq 0$ and $S \cdot f(u) \geq 0$.

The function only uses R code with basic arithmetic, such that it should also work with “generalized” numbers (such as mpfr-numbers) as long the necessary Ops methods are defined for those.

The underlying algorithm assumes a continuous function (which then is known to have at least one root in the interval).

Convergence is declared either if $f(x) == 0$ or the change in $x$ for one step of the algorithm is less than tol (plus an allowance for representation error in $x$).

If the algorithm does not converge in maxiter steps, a warning is printed and the current approximation is returned.

$f$ will be called as $f(x, \ldots)$ for a (generalized) numeric value of $x$.

**Value**

A list with four components: root and $f$.root give the location of the root and the value of the function evaluated at that point. iter and estim.prec give the number of iterations used and an approximate estimated precision for root. (If the root occurs at one of the endpoints, the estimated precision is NA.)

**Source**

Based on zeroin() (in package rootoned) by John Nash who manually translated the C code in R’s zeroin.c and on uniroot() in R’s sources.

**References**

Brent, R. (1973), see uniroot.

**See Also**

R’s own (stats package) uniroot, polyroot for all complex roots of a polynomial; optimize, nlm.

**Examples**

```r
require(utils) # for str

## some platforms hit zero exactly on the first step:
## if so the estimated precision is 2/3.
f <- function (x,a) x - a
str(xmin <- unirootR(f, c(0, 1), tol = 0.0001, a = 1/3))

## handheld calculator example: fixpoint of cos(.):
rc <- unirootR(function(x) cos(x) - x, lower=-pi, upper=pi, tol = 1e-9)
rc$root

## the same with much higher precision:
rcM <- unirootR(function(x) cos(x) - x,
   interval= mpfr(c(-3,3), 300), tol = 1e-40)
```
\texttt{rcM}
\begin{verbatim}
x0 <- rcM$root
stopifnot(all.equal(cos(x0), x0,
     tol = 1e-40))## 40 digits accurate!
\end{verbatim}
\begin{verbatim}
str(unirootR(function(x) x*(x^2-1) + .5, lower = -2, upper = 2,
     tol = 0.0001), digits.d = 10)
str(unirootR(function(x) x*(x^2-1) + .5, lower = -2, upper = 2,
     tol = 1e-10 ), digits.d = 10)
\end{verbatim}

\texttt{## A sign change of \textit{f(\cdot)}, but not a zero but rather a "pole":
\tan. <- function(x) tan(x \times (\text{Const("pi",200)/180}))# == tan( <angle> )
(rtan <- unirootR(tan., interval = mpfr(c(80,100), 200), tol = 1e-40))
## finds 90 ("ok"), and now gives a warning

\texttt{## Find the smallest value \textit{x} for which \textit{exp}(\textit{x}) > 0 (numerically):
\texttt{r <- unirootR(function(x) 1e99*exp(x) - ep.M, mpfr(c(-1e20, 0), 200))
r # 97 iterations; \textit{f.root} is very similar to \textit{ep.M}
}

\texttt{## interval extension 'extendInt' -----------
}
\texttt{f1 <- function(x) (121 - x^2)/(x^2+1)
f2 <- function(x) exp(-x)*(x - 12)
tools::assertError(unirootR(f1, c(0,10)), verbose=TRUE)
##---> error: \textit{f()} .. end points not of opposite sign

\texttt{## where as 'extendInt="yes"' simply first enlarges the search interval:
u1 <- unirootR(f1, c(0,10),extendInt="yes", trace=1)
u2 <- unirootR(f2, mpfr(c(0,2), 128), extendInt="yes", trace=2, verbose=FALSE, tol = 1e-25)
stopifnot(all.equal(u1$root, 11, tolerance = 1e-5),
     all.equal(u2$root, 12, tolerance = 1e-23))

\texttt{## The *danger* of interval extension:
\texttt{## No way to find a zero of a positive function, but
\texttt{## numerically, \textit{f(-|M|)} becomes zero :
u3 <- unirootR(exp, c(0,2), extendInt="yes", trace=TRUE)

\texttt{## Nonsense example (must give an error):
tools::assertCondition( unirootR(function(x) 1, 0:1, extendInt="yes"),
}
"error", verbose=TRUE)
Index

* Forward Difference
  sumBinomMpfr, 74
* Rice integral
  sumBinomMpfr, 74
* arithmetic
  frexpMpfr, 21
* arith
  Bernoulli, 9
  chooseMpfr, 12
  factorialMpfr, 14
  formatHex, 16
  gmp-conversions, 23
  matmult, 34
  mpfr.utils, 53
  pbetaI, 62
  pmax, 65
  roundMpfr, 70
  sumBinomMpfr, 74
* array
  mpfrArray, 54
  mpfrMatrix-utils, 58
* character
  formatMpfr, 19
* classes
  array_or_vector-class, 6
  atomicVector-class, 8
  Mnumber-class, 35
  mpfr, 36
  mpfr-class, 39
  mpfrMatrix, 55
* distribution
  mpfr-distr-etc, 44
  pbetaI, 62
  qnormI, 66
* manip
  sapplyMpfr, 70
  seqMpfr, 72
* math
  Bessel_mpfr, 10
  igamma, 27
  integrateR, 28
  is.whole, 31
  log1mexp, 32
  mpfr-special-functions, 46
  qnormI, 66
* methods
  asNumeric-methods, 7
  bind-methods, 11
  pmax, 65
* misc
  Rmpfr-workarounds, 69
* optimize
  hjkMpfr, 24
  optimizeR, 60
  unirootR, 76
* package
  Rmpfr-package, 3
* print
  formatMpfr, 19
* univar
  pmax, 65
* utilities
  frexpMpfr, 21
  integrateR, 28
  mpfr-utils, 47
  str.mpfr, 73
  ..bigq2mpfr (mpfr-utils), 47
  ..bigz2mpfr (mpfr-utils), 47
  .Machine, 42, 50, 77
  .bigq2mpfr (gmp-conversions), 23
  .bigz2mpfr (gmp-conversions), 23
  .getPrec (mpfr-utils), 47
  .getSign (mpfr-utils), 47
  .matmult.R (matmult), 34
  .mpfr (mpfr-utils), 47
  .mpfr.is.whole (mpfr.utils), 53
  .mpfr2bigq (gmp-conversions), 23
  .mpfr2bigz (gmp-conversions), 23
Arith,mpfr,mpfr-method (mpfr-class), 39
Arith,mpfr,mpfrArray-method (mpfrMatrix), 55
Arith,mpfr,numeric-method (mpfr-class), 39
Arith,mpfrArray,mpfr-method (mpfrMatrix), 55
Arith,mpfrArray,mpfrArray-method (mpfrMatrix), 55
Arith,mpfrArray,numeric-method (mpfrMatrix), 55
Arith,numeric,mpfr-method (mpfr-class), 39
Arith,numeric,mpfrArray-method (mpfrMatrix), 55
array, 5, 7, 20, 37, 41, 50, 54–57, 71
array_or_vector, 36
array_or_vector-class, 6
as, 42
as.bigq, 24
as.bigz, 23, 24
as.integer, 50
as.integer,mpfr-method (mpfr-class), 39
as.numeric, 7
as.numeric,mpfr-method (mpfr-class), 39
as.vector,mpfrArray,missing-method (mpfrMatrix), 55
atan, 40
atan, ANY,mpfr-method (mpfr-class), 39
atan, ANY,mpfrArray-method (mpfrMatrix), 55
atan,mpfr,ANY-method (mpfr-class), 39
atan,mpfr,mpfr-method (mpfr-class), 39
atan,mpfr,numeric-method (mpfr-class), 39
atan,mpfrArray,ANY-method (mpfr-class), 39
atan,mpfrArray,mpfrArray-method (mpfr-class), 39
atan,numeric,mpfr-method (mpfr-class), 39
atan2,mpfr,ANY-method (mpfr-class), 39
atan2,mpfr,mpfr-method (mpfr-class), 39
atan2,mpfr,numeric-method (mpfr-class), 39
atan2,mpfrArray,ANY-method (mpfr-class), 39
atan2,mpfrArray,mpfrArray-method (mpfr-class), 39
atan2,numeric,mpfr-method (mpfr-class), 39
atanh, 41
atomicVector-class, 8
base::pmin, 66
Bernoulli, 5, 9, 42
Bessel_mpfr, 5, 10, 47
besselI, 10
besselY, 10
beta, 40
beta, ANY,mpfr-method (mpfr-class), 39
beta, ANY,mpfrArray-method (mpfr-class), 39
beta, mpfr, ANY-method (mpfr-class), 39
beta, mpfr, mpfr-method (mpfr-class), 39
beta, mpfr, numeric-method (mpfr-class), 39
beta, mpfrArray, ANY-method (mpfr-class), 39
beta, mpfrArray, mpfrArray-method (mpfr-class), 39
beta, numeric, mpfr-method (mpfr-class), 39
beta, numeric, mpfrArray-method (mpfr-class), 39
bigq, 23, 37, 63
bigrational, 6
bigz, 23, 37, 42
bind-methods, 11
c, 53
c.mpfr, 5
c.mpfr (mpfr.utils), 53
cbind, 5, 11, 12
cbind (bind-methods), 11
cbind, ANY-method (bind-methods), 11
cbind, Mnumber-method (bind-methods), 11
cbind-methods (bind-methods), 11
cbind2, 12
ceiling, 41
character, 37, 41, 42, 49, 56, 60
choose, 12, 13
chooseMpfr, 5, 12, 75
chooseZ, 12, 13, 63, 75
class, 8, 12, 17, 31, 39, 49, 65
coerce, array, mpfr-method (mpfr-class), 39
coerce, array, mpfrArray-method (mpfrMatrix), 55
coerce, bigq, mpfr-method
  (gmp-conversions), 23
coerce, bigz, mpfr-method
  (gmp-conversions), 23
coerce, character, mpfr-method
  (mpfr-class), 39
coerce, integer, mpfr-method
  (mpfr-class), 39
coerce, logical, mpfr-method
  (mpfr-class), 39
coerce, matrix, mpfrMatrix-method
  (mpfrMatrix), 55
coerce, mpfr, bigz-method (mpfr-class), 39
coerce, mpfr, character-method
  (mpfr-class), 39
coerce, mpfr, integer-method
  (mpfr-class), 39
coerce, mpfr, mpfr1-method (mpfr-class), 39
coerce, mpfr, mpfr1-method (mpfr-class), 39
coerce, mpfr, numeric-method
  (mpfr-class), 39
coerce, mpfr1, mpfr-method
  (mpfr-class), 39
coerce, mpfr1, mpfr1-method (mpfr-class), 39
coerce, mpfr1, mpfr-method
  (mpfr-class), 39
coerce, mpfr1, mpfr1-method (mpfr-class), 39
coerce, mpfrArray, array-method
  (mpfrMatrix), 55
crossprod, array_or_vector, mpfr-method
  (mpfr-class), 39
crossprod, Mnumber, mpfr-method
  (mpfrMatrix), 55
crossprod, mpfr, array_or_vector-method
  (mpfr-class), 39
crossprod, mpfr, missing-method
  (mpfrMatrix), 55
crossprod, mpfr, Mnumber-method
  (mpfrMatrix), 55
crossprod, mpfr, mpfr-method
  (mpfrMatrix), 55
crossprod, mpfr, mpfrMatrix-method
  (mpfrMatrix), 55
crossprod, mpfrMatrix, mpfr-method
  (mpfrMatrix), 55
crossprod, mpfrMatrix, mpfrMatrix-method
  (mpfrMatrix), 55
cummmax, 41
cummin, 41
cumprod, 41
cumsum, 41
dbinom, 45
INDEX

dbinom (mpfr-distr-etc), 44

determinant, 59
determinant.mpfrMatrix
    (mpfrMatrix-utils), 58
dgamma, 45
dgamma (mpfr-distr-etc), 44
diag,mpfrMatrix-method (mpfrMatrix), 55
diag<-,mpfrMatrix-method (mpfrMatrix), 55
diff, 53
diff.default, 53
diff.mpfr (mpfr.utils), 53
digamma, 5

dim, 7, 20, 40, 56
dim,mpfrArray-method (mpfrMatrix), 55
dim<-,mpfr-method (mpfr-class), 39
dimnames, 20
dimnames,mpfrArray-method (mpfrMatrix), 55
dimnames<-,mpfrArray-method
    (mpfrMatrix), 55
dnbinom, 45
dnbinom (mpfr-distr-etc), 44
dnorm (mpfr-distr-etc), 44
dotsMethods, 11
double, 49, 50
dpois, 45
dpois (mpfr-distr-etc), 44
dt, 45
dt (mpfr-distr-etc), 44
duplicated, 42

ei (mpfr-special-functions), 46
erf, 45, 54
erf (mpfr-special-functions), 46
erfc (mpfr-special-functions), 46
exp, 41
expm1, 41

factorial, 13, 15
factorial,mpfr-method (mpfr-class), 39
factorialMpfr, 5, 13, 14, 41
factorialZ, 15
floor, 41
format, 16, 17, 19–21, 50, 77
format,mpfr-method (mpfr-class), 39
formatBin, 37
formatBin (FormatHex), 16
formatDec (FormatHex), 16

formatHex, 16
formatMpfr, 5, 17, 19, 41, 49, 50, 74
formatN, 21
formatN.mpfr (formatMpfr), 19
frexp, 22
frexpMpfr, 21
function, 71, 75

gamma, 5, 13, 15, 28, 40, 41
gamma (mpfr-utils), 47
groupMembers, 41
getPrec, 5, 16, 37, 66, 70, 74
gmp, 6
gmp-conversions, 23
golden_ratio, 61

hjk, 24
hjkMpfr, 5, 24, 61
hypot (mpfr-class), 39

igamma, 27
Im,mpfr-method (mpfr-class), 39
integer, 8, 13, 42, 49–51
integrate, 29, 30
integrateR, 5, 28
interactive, 49
invisible, 50
is.atomic, 8
is.finite, 51
is.finite,mpfr-method (mpfr-class), 39
is.finite,mpfrArray-method
    (mpfr-class), 39
is.infinite,mpfr-method (mpfr-class), 39
is.infinite,mpfrArray-method
    (mpfr-class), 39
is.integer, 31
is.mpfr (mpfr), 36
is.mpfr (mpfr-class), 39
is.mpfr (mpfr-utils), 47
is.nan,mpfr-method (mpfr-class), 39
is.nan,mpfrArray-method (mpfr-class), 39
is.whole, 31, 31, 42, 53

j0, 47
j0 (Bessel_mpfr), 10
j1 (Bessel_mpfr), 10
jn, 42
jn (Bessel_mpfr), 10
lapply, 70, 71
lbeta, ANY, mpfr-method (mpfr-class), 39
lbeta, ANY, mpfrArray-method (mpfr-class), 39
lbeta, mpfr, ANY-method (mpfr-class), 39
lbeta, mpfr, mpfr-method (mpfr-class), 39
lbeta, mpfr, numeric-method (mpfr-class), 39
lbeta, mpfrArray, ANY-method (mpfr-class), 39
lbeta, mpfrArray, mpfr-method (mpfr-class), 39
lapply, 86
INDEX
Li2 (mpfr-special-functions), 46
list, 22, 25, 39, 56, 59, 61, 67, 74
load, 51
log, 41, 59
log, mpfr-method (mpfr-class), 39
log10, 41
log1mexp, 32
log1p, 41
log1pexp (log1mexp), 32
log2, 41
Logic, mpfr, mpfr-method (mpfr-class), 39
Logic, mpfr, numeric-method (mpfr-class), 39
Logic, numeric, mpfr-method (mpfr-class), 39
logical, 16, 19, 20, 42, 45, 49, 51, 63
Math, 41, 46
Math, mpfr-method (mpfr-class), 39
Math2, 46
Math2, mpfr-method (mpfr-class), 39
matmult, 34
matrix, 7, 35, 50, 55
max, 41
mean, 41
mean, mpfr-method (mpfr-class), 39
mean.default, 41
median, mpfr-method (mpfr-class), 39
min, 41, 66
missing, 37
Mnumber, 11
Mnumber-class, 35
mNumber-class (Mnumber-class), 35
Mod, mpfr-method (mpfr-class), 39
mpfr-class, 5, 39
mpfr-distr (mpfr-distr-etc), 44
mpfr-distr-etc, 44
mpfr-special-functions, 46
mpfr-utils, 47
mpfr.is.0 (mpfr.utils), 53
mpfr.is.integer (mpfr.utils), 53
mpfr.utils, 53
mpfr1, 56
mpfr1-class (mpfr-class), 39
mpfr2array, 55
mpfr2array (mpfr-utils), 47
mpfr_default_prec (mpfr-utils), 47
mpfrArray, 5, 7, 20, 37, 40, 49–51, 54, 54, 55, 57, 71
mpfrArray-class (mpfrMatrix), 55
mpfrImport (mpfr-utils), 47
mpfrIs0, 51
mpfrIs0 (mpfr.utils), 53
mpfrMatrix, 7, 11, 12, 34–37, 42, 50, 54, 55, 59
mpfrMatrix-class, 5
mpfrMatrix-class (mpfrMatrix), 55
mpfrMatrix-utils, 58
mpfrVersion (mpfr.utils), 53
mpfrXport (mpfr-utils), 47
names, 49
NaN, 40
nlm, 78
norm, 57
norm, ANY, missing-method (mpfrMatrix), 55
norm, mpfrMatrix, character-method (mpfrMatrix), 55
NULL, 49, 56
numeric, 8, 10, 27, 34, 37–39, 42, 45, 46, 49, 53, 63, 71
numeric_version, 53
numericVector-class (Mnumber-class), 35
Ops, 78
Ops, ANY, mpfr-method (mpfr-class), 39
Ops, array, mpfr-method (mpfr-class), 39
Ops, bigq, mpfr-method (mpfr-class), 39
INDEX

Ops, bigz, mpfr-method (mpfr-class), 39
Ops, mpfr, ANY-method (mpfr-class), 39
Ops, mpfr, array-method (mpfr-class), 39
Ops, mpfr, bigz-method (mpfr-class), 39
Ops, mpfr, bigz-method (mpfr-class), 39
Ops, mpfr, vector-method (mpfr-class), 39
Ops, vector, mpfr-method (mpfr-class), 39
optim, 26
optimize, 60, 61, 78
optimizeR, 5, 26, 60
options, 19, 49, 50, 74
order, 42
outer, 69
outer (Rmpfr-workarounds), 69

pbeta, 45, 63
pbetaI, 45, 62
pgamma, 27, 28
pmax, 65, 65
pmax, ANY-method (pmax), 65
pmax, mNumber-method (pmax), 65
pmax-methods (pmax), 65
pmin, 66
pmin (pmax), 65
pmin, ANY-method (pmax), 65
pmin, mNumber-method (pmax), 65
pmin-methods (pmax), 65
pnorm, 5, 45, 47, 66
pnorm (mpfr-distr-etc), 44
pochMpfr, 15, 42
pochMpfr (chooseMpfr), 12
polyroot, 78
prettyNum, 20, 21
print, 17, 19, 29, 39
print.default, 49
print.integrate, 30
print.integrateR (integrateR), 28
print.mpfr (mpfr-utils), 47
print.mpfr1 (mpfr-class), 39
print.mpfrArray (mpfr-utils), 47
print.Ncharacter (formatHex), 16
print.summaryMpfr (mpfr-class), 39
prod, 41

qnorm, 66, 67
qnormI, 66
quantile, 42
quantile, mpfr-method (mpfr-class), 39
range, 41, 66
rank, 42
raw, 42
rbind, 11
rbind (bind-methods), 11
rbind, ANY-method (bind-methods), 11
rbind, MNumber-method (bind-methods), 11
rbind-methods (bind-methods), 11
Re, mpfr-method (mpfr-class), 39
Rmpfr (Rmpfr-package), 3
Rmpfr-package, 3
Rmpfr-workarounds, 69
round, 41, 70
roundMpfr, 5, 37, 41, 42, 50, 70
rowMeans, mpfrArray-method (mpfrMatrix), 55
rowSums, mpfrArray-method (mpfrMatrix), 55
sapply, 70, 71
sapplyMpfrr, 70
save, 51
seq, 72, 73
seqMpfrr, 5, 72
setPrec (roundMpfrr), 70
show, integrateR-method (integrateR), 28
show, mpfr-method (mpfr-class), 28
show, mpfr1-method (mpfr-class), 39
show, mpfrArray-method (mpfrMatrix), 55
show, summaryMpfrr-method (mpfr-class), 39
sign, 40, 41, 51
sign, mpfr-method (mpfr-class), 39
sign, mpfrArray-method (mpfrMatrix), 55
signif, 41
sin, 41
sinh, 41
sinpi, 41
sort, 42
sprintf, 16, 17, 20
sqrt, 41
str, 54, 73
str.default, 73
str.mpfr, 54, 73
sum, 41
sumBinomMpfrr, 5, 13, 63, 74
Summary, 41
Summary, mpfr-method (mpfr-class), 39
summary, mpfr-method (mpfr-class), 39
summary.default, 41
summaryMpfr-class (mpfr-class), 39

t, mpfr-method (mpfr-class), 39
t, mpfrMatrix-method (mpfrMatrix), 55
tan, 41
tanh, 41
tanpi, 41
tcrossprod, 34, 35
tcrossprod, array_or_vector, mpfr-method
  (mpfr-class), 39
tcrossprod, Mnumber, mpfr-method
  (mpfrMatrix), 55
tcrossprod, mpfr, array_or_vector-method
  (mpfr-class), 39
tcrossprod, mpfr, missing-method
  (mpfrMatrix), 55
tcrossprod, mpfr, Mnumber-method
  (mpfrMatrix), 55
tcrossprod, mpfr, mpfr-method
  (mpfrMatrix), 55
tcrossprod, mpfr, mpfrMatrix-method
  (mpfrMatrix), 55
tcrossprod, mpfrMatrix, mpfr-method
  (mpfrMatrix), 55
tcrossprod, mpfrMatrix, mpfrMatrix-method
  (mpfrMatrix), 55
toNum, 7
toNum (mpfr-utils), 47
trigamma, 5, 41
trunc, 41
typeof, 7, 39
unique, 42
unique, mpfr, ANY-method (mpfr-class), 39
unique, mpfr-method (mpfr-class), 39
unique.mpfr (mpfr-class), 39
uniroot, 76, 78
unirootR, 5, 61, 66, 67, 76
vapply, 70
vector, 56
Vectorize, 29

which.max, 42
which.max, mpfr-method (mpfr-class), 39
which.min, 42
which.min, mpfr-method (mpfr-class), 39

y0 (Bessel_mpfr), 10

yn, 47
yn (Bessel_mpfr), 10

zeta, 5, 9, 42
zeta (mpfr-special-functions), 46