Package ‘Rmpfr’

January 24, 2020

Title R MPFR - Multiple Precision Floating-Point Reliable
Version 0.8-1
Date 2020-01-14
DateNote Previous CRAN version 0.7-2 on 2019-01-18
Type Package
SystemRequirements gmp (>= 4.2.3), mpfr (>= 3.0.0)
SystemRequirementsNote 'MPFR' (MP Floating-Point Reliable Library, http://mpfr.org/) and 'GMP' (GNU Multiple Precision library, http://gmplib.org/), see >> README.md
Depends gmp (>= 0.5-8), R (>= 3.3.0)
Imports stats, utils, methods
Suggests MASS, Bessel, polynom, sfsmisc (>= 1.0-20), Matrix
SuggestsNote MASS, polynom, sfsmisc: only for vignette; Matrix: test-tools
URL http://rmpfr.r-forge.r-project.org/
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.
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
Maintainer Martin Maechler <maechler@stat.math.ethz.ch>
Repository CRAN
Date/Publication 2020-01-24 14:30:13 UTC
R topics documented:

- Rmpfr-package ........................................ 3
- array_or_vector-class ................................. 6
- asNumeric-methods ....................................... 7
- atomicVector-class ...................................... 8
- Bernoulli .................................................. 8
- Bessel_mpfr ............................................... 10
- bind-methods ............................................. 11
- chooseMpfr ............................................... 12
- factorialMpfr ............................................ 14
- formatHex ............................................... 15
- formatMpfr ............................................... 18
- gmp-conversions ......................................... 20
- hjkMpfr .................................................... 21
- igamma ..................................................... 24
- integrateR ............................................... 25
- is.whole ................................................. 28
- Mnumber-class ........................................... 29
- mpfr ....................................................... 30
- mpfr-class ............................................... 32
- mpfr-distr-etc .......................................... 37
- mpfr-special-functions ................................. 39
- mpfr-utils ............................................... 40
- mpfr.utils ............................................... 44
- mpfrArray ............................................... 46
- mpfrMatrix ............................................... 47
- mpfrMatrix-utils ....................................... 50
- optimizeR ............................................... 52
- pbetaI .................................................... 54
- pmax ....................................................... 56
- Rmpfr-workarounds ..................................... 57
- roundMpfr ................................................ 58
- sapplyMpfr ............................................... 58
- seqMpfr .................................................... 59
- str.mpfr ................................................... 60
- sumBinomMpfr ............................................ 61
- unirootR .................................................. 63
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: 0.8-1
Date: 2020-01-14
DateNote: Previous CRAN version 0.7-2 on 2019-01-18
Type: Package
Authors@R: c(person("Martin","Maechler", role = c("aut","cre"), email = "maechler@stat.math.ethz.ch", comment = "R Development Core Team"), person("Martin","Maechler", role = c("aut","cre"), email = "maechler@stat.math.ethz.ch", comment = "R Development Core Team"), person("Richard","M.", "Heiberger", role = "ctb", email="rmh@temple.edu", comment = "formatHex(), *Bin, *Dec")

SystemRequirements: gmp (>= 4.2.3), mpfr (>= 3.0.0)
SystemRequirementsNote: 'MPFR' (MP Floating-Point Reliable Library, http://mpfr.org/) and 'GMP' (GNU Multiple Precision Library)
Depends: gmp (>= 0.5-8), R (>= 3.3.0)
Imports: stats, utils, methods
Suggests: MASS, Bessel, polynom, sfsmisc (>= 1.0-20), Matrix
SuggestsNote: MASS, polynom, sfsmisc: only for vignette; Matrix: test-tools
URL: http://rmpfr.r-forge.r-project.org/
Description: Arithmetic (via S4 classes and methods) for arbitrary precision floating point numbers, including transcendental functions.
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_mpf 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 Auxiliary Class "array_or_vector"
asNumeric-methods Methods for 'asNumeric(<mpfr>)'
atomicVector-class Virtual Class "atomicVector" of Atomic Vectors
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.mpfr</td>
<td>MPFR Number Utilities</td>
</tr>
<tr>
<td>cbind</td>
<td>&quot;mpfr&quot; '...' - Methods for Functions cbind(), rbind()</td>
</tr>
<tr>
<td>chooseMpfr</td>
<td>Binomial Coefficients and Pochhammer Symbol aka Rising Factorial</td>
</tr>
<tr>
<td>determinant.mpfrMatrix</td>
<td>Functions for mpfrMatrix Objects</td>
</tr>
<tr>
<td>factorialMpfr</td>
<td>Factorial 'n!' in Arbitrary Precision</td>
</tr>
<tr>
<td>formatHex</td>
<td>Flexibly Format Numbers in Binary, Hex and Decimal Format</td>
</tr>
<tr>
<td>formatMpfr</td>
<td>Formatting MPFR (multiprecision) Numbers</td>
</tr>
<tr>
<td>getPrec</td>
<td>Rmpfr - Utilities for Precision Setting, Printing, etc</td>
</tr>
<tr>
<td>hjkMpfr</td>
<td>Hooke-Jeeves Derivative-Free Minimization R (working for MPFR)</td>
</tr>
<tr>
<td>igamma</td>
<td>Incomplete Gamma Function</td>
</tr>
<tr>
<td>integrateR</td>
<td>One-Dimensional Numerical Integration - in pure R</td>
</tr>
<tr>
<td>is.whole.mpfr</td>
<td>Whole (&quot;Integer&quot;) Numbers</td>
</tr>
<tr>
<td>mpfr</td>
<td>Create &quot;mpfr&quot; Numbers (Objects)</td>
</tr>
<tr>
<td>mpfr-class</td>
<td>Class &quot;mpfr&quot; of Multiple Precision Floating Point Numbers</td>
</tr>
<tr>
<td>mpfrArray</td>
<td>Construct &quot;mpfrArray&quot; almost as by 'array()'</td>
</tr>
<tr>
<td>mpfrMatrix-class</td>
<td>Classes &quot;mpfrMatrix&quot; and &quot;mpfrArray&quot;</td>
</tr>
<tr>
<td>optimizeR</td>
<td>High Precision One-Dimensional Optimization</td>
</tr>
<tr>
<td>outer</td>
<td>Base Functions etc, as an Rmpfr version</td>
</tr>
<tr>
<td>pbetaI</td>
<td>Accurate Incomplete Beta / Beta Probabilities For Integer Shapes</td>
</tr>
<tr>
<td>pmax</td>
<td>Parallel Maxima and Minima</td>
</tr>
<tr>
<td>pnorm</td>
<td>Distribution Functions etc (MPFR)</td>
</tr>
<tr>
<td>roundMpfr</td>
<td>Rounding to Binary bits, &quot;mpfr-internally&quot;</td>
</tr>
<tr>
<td>sapplyMpfr</td>
<td>Apply a Function over a &quot;mpfr&quot; Vector</td>
</tr>
<tr>
<td>seqMpfr</td>
<td>&quot;mpfr&quot; Sequence Generation</td>
</tr>
<tr>
<td>str.mpfr</td>
<td>Compactly Show STRucture of Rmpfr Number Object</td>
</tr>
<tr>
<td>sumBinomMpfr</td>
<td>(Alternating) Binomial Sums via Rmpfr</td>
</tr>
<tr>
<td>unirootR</td>
<td>One Dimensional Root (Zero) Finding - in pure R</td>
</tr>
<tr>
<td>zeta</td>
<td>Special Mathematical Functions (MPFR)</td>
</tr>
</tbody>
</table>

Further information is available in the following vignettes:

- Maechler_useR_2011-abstr
- useR-2011-abstract
- Rmpfr-pkg
- Arbitrarily Accurate Computation with R Package Rmpfr
- log1mexp-note
- Accurately Computing log(1 - exp(.)) – Assessed by Rmpfr

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

\textbf{Partial index:}

- \texttt{mpfr} Create "mpfr" Numbers (Objects)
- \texttt{mpfrArray} Construct "mpfrArray" almost as by \texttt{array()}
- \texttt{mpfr-class} Class "mpfr" of Multiple Precision Floating Point Numbers
- \texttt{mpfrMatrix-class} Classes "mpfrMatrix" and "mpfrArray"

- \texttt{Bernoulli} Bernoulli Numbers in Arbitrary Precision
- \texttt{Bessel_mpfr} Bessel functions of Integer Order in multiple precisions
- \texttt{c.mpfr} MPFR Number Utilities
- \texttt{cbind} "mpfr" . . . - Methods for Functions cbind(), rbind()
- \texttt{chooseMpfr} Binomial Coefficients and Pochhammer Symbol aka Rising Factorial
- \texttt{factorialMpfr} Factorial 'n!' in Arbitrary Precision
- \texttt{formatMpfr} Formatting MPFR (multiprecision) Numbers
- \texttt{getPrec} Rmpfr - Utilities for Precision Setting, Printing, etc
- \texttt{roundMpfr} Rounding to Binary bits, "mpfr-internally"
- \texttt{seqMpfr} "mpfr" Sequence Generation
- \texttt{sumBinomMpfr} (Alternating) Binomial Sums via Rmpfr
- \texttt{zeta} Special Mathematical Functions (MPFR)

- \texttt{integrateR} One-Dimensional Numerical Integration - in pure R
- \texttt{unirootR} One Dimensional Root (Zero) Finding - in pure R
- \texttt{optimizeR} High Precisione One-Dimensional Optimization
- \texttt{hjkMpfr} Hooke-Jeeves Derivative-Free Minimization R (working for MPFR)

Further information is available in the following vignettes:

- \texttt{Rmpfr-pkg} Rmpfr (source, pdf)
- \texttt{log1mexp-note} Accurately Computing \texttt{log(1 - exp(.))} – Assessed by Rmpfr (source, pdf)

\textbf{Author(s)}

Martin Maechler

\textbf{References}

MPFR (MP Floating-Point Reliable Library), \url{http://mpfr.org/}

GMP (GNU Multiple Precision library), \url{http://gmplib.org/}

and see the vignettes mentioned above.
See Also

The R package `gmp` for big integer and rational numbers (`bigrational`) on which `Rmpfr` now 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:

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 × 3 matrix.

Objects from the Class

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

Examples

```r
showClass("array_or_vector")
```
Methods for function `asNumeric` (in package `gmp`).

Usage

```r
## 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 \texttt{is.atomic}.

**Objects from the Class**

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

**Methods**

In the \texttt{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**

\texttt{is.atomic, integer, numeric, complex}, etc.

**Examples**

\texttt{showClass("atomicVector")}

---

**Bernoulli**  

*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 \(-1/2\).)
Bernoulli

Usage

Bernoulli(k, precBits = 128)

Arguments

k non-negative integer vector
precBits the precision in bits desired.

Value

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

Author(s)

Martin Maechler

References

http://en.wikipedia.org/wiki/Bernoulli_number

See Also

zeta is used to compute them.

Examples

Bernoulli(0:10)
plot(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)
abline(h=0,v=0, lty=3, col="gray")
k <- 0:15; k[1] <- 1e-4
points(k, -k*zeta(1-k), col=2, cex=2, pch=1+2*(k%%2))

## They pretty much explode for larger k :
k2 <- 2*(1:120)
plot(k2, abs(as.numeric(Bernoulli(k2))), log = "y")
title("Bernoulli numbers exponential growth")

Bernoulli(10000)# - 9.0494239636 * 10^27677
Bessel functions of Integer Order in multiple precisions

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

\[
\begin{align*}
\text{Ai}(x) \\
j_0(x) \\
j_1(x) \\
j_n(n, x, \text{rnd.mode} = c(\text{"N"}, \text{"D"}, \text{"U"}, \text{"Z"}, \text{"A"})) \\
y_0(x) \\
y_1(x) \\
y_n(n, x, \text{rnd.mode} = c(\text{"N"}, \text{"D"}, \text{"U"}, \text{"Z"}, \text{"A"}))
\end{align*}
\]

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( all.equal(besselY(x, 0), bY0 <- y0(x)),
           all.equal(besselJ(x, 1), bJ1 <- j1(x)),
           all.equal(y0(0,x), bY0),
           all.equal(j0(1,x), bJ1)
)

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 http://dlmf.nist.gov/9.2.ii
  all.equal(Ai(100), mpfr("2.53448215208818448950552569264981561e-291"), tol=1e-37)
)
two3rd <- 2/mpfr(3, 144)
print( all.equal(Ai(0), 1/(3^two3rd * gamma(two3rd)), tol=0) ) # 1.7e-40
if(Rmpfr:::doExtras()) { # slowish:
system.time(ai1k <- Ai(1000)) # 1.4 sec (on 2017 lynne)
  stopifnot(all.equal(log10(ai1k),
               -9157.03193409585185582, tol=1e-16))
}  # ver >= 3.0

bind-methods

"mpfr" '...' - Methods for Functions cbind(), rbind()

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

  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.

Value

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

Methods

... = "Mnumber" is used to (c)rbind 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
chooseMpfr

See Also

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

Examples

```r
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

```r
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"))
```

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.
chooseMpf

Value

For

chooseMpf(), pochMpf(): an mpfr vector of length max(length(a),length(n));
chooseMpf.all(n, k0): a mpfr vector of length 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

If you need high precision choose(a,n) (or Pochhammer(a,n)) for large n, maybe better work with
the corresponding factorial(mpfr(..)), or gamma(mpfr(..)) terms.

See Also

choose(n,m) (base R) computes the binomial coefficient \( C_{n,m} \) which can also be expressed via
Pochhammer symbol as \( C_{n,m} = (n-m+1)^{(m)}/m! \).

chooseZ from package gmp; for now, factorialMpf.

For (alternating) binomial sums, directly use sumBinomMpf, as that is potentially more efficient.

Examples

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

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

## --- Experimenting & Checking
n.set <- c(1:10, 20, 50:55, 100:105, 200:203, 300:303, 500:503,
699:702, 999:1001)
if(!Rmpfr:::doExtras()) { ## speed up: smaller set
  n. <- n.set[-c(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.set[i.n], ":")
  C1[i.n] <- system.time(c.c <- chooseMpf.all(n ))[1]
  C2[i.n] <- system.time(c.2 <- chooseMpf(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]"
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

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

Arguments

- `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 \( \mathcal{R} \).
- `factorialZ` (package `gmp`), to replace `factorialMpfr`, see above.
- `chooseMpfr()` and `pochMpfr()` (on the same page).

Examples

```r
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"

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

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

---

### formatHex

**Flexibly Format Numbers in Binary, Hex and Decimal Format**

#### Description

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

#### Usage

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

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

```r
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 representation 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

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!)
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(F.2 <- FBB[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
TenFrac9
formatHex(TenFrac9)
formatBin(TenFrac9)
formatBin(TenFrac9, scientific=FALSE)
formatDec(TenFrac9)
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),
base = 10, showNeg0 = TRUE, max.digits = Inf,
big.mark = "", big.interval = 3L,
small.mark = "", small.interval = 5L,
dermal.mark = ".",
exponent.char = if(base <= 14) "e" else if(base <= 36) "E" else "|e",
exponent.plus = TRUE,
zero.print = NULL, drop0trailing = FALSE, ...)
```

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

```r
.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()`.
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 \( B \geq 15 \), \( "e" \) is part of the (mantissa) digits and the same is true for \( "E" \) when \( B \geq 37 \).

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 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 that in scientific notation, the integer exponent is always in decimal, i.e., base 10 (even when base is not 10), but of course meaning base powers, e.g., in base 32, "u.giE3" is the same as "ugi0" which is \( 32^3 \) times "u.gi". This is in contrast, e.g., with sprintf("%a",x) where the powers after "p" are powers of 2.

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

## 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 = "\", # (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) #-> 1'000'000

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)

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)
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:
sq <- SS / as.bigz(2)^100
MP <- as(sq, "mpfr")
stopifnot(identical(MP, .bigq2mpfr(sq)),
          SS == MP * as(2, "mpfr")^100)
```

Description

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

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

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

## 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)))

## 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)))

## 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) {
str(rb.M. <- hjkMpfr(mpfr(numeric(10), prec=128), nsf, control=list(info=TRUE)))

## Hooke-Jeeves does not work well on non-smooth functions
nsf <- function(x) {
str(rb.M. <- hjkMpfr(mpfr(numeric(10), prec=128), nsf, control=list(info=TRUE)))
\text{max}(f1, f2, f3)
}
par0 <- c(1, 1) # true min 7.2 at (1.2, 2.4)
h.d <- hjkMpfr(par0, nsf) # fmin=8 at xmin=(2,2)

## and this is not at all better (but slower!)
h.M <- hjkMpfr(mpfr(c(1,1), 128), nsf, control = list(tol = 1e-15))

## --> demo(hjkMpfr) # -> Fletcher's chebyquad function m = n -- residuals

\textbf{igamma} \\
\textit{Incomplete Gamma Function}

\textbf{Description}

For MPFR version >= 3.2.0, the following MPFR library function is provided: mpfr\_gamma\_inc(a,x), the R interface of which is igamma(a,x), where 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 pgamma(x,a) is

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

we get

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

\textbf{Usage}

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

\textbf{Arguments}

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

A numeric vector of “common length”, recycling 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)) #-> 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

\( f \) \hspace{1cm} \text{an 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.}

\( \text{lower, upper} \) \hspace{1cm} \text{the limits of integration. Currently \textit{must} be finite. Do use "mpfr"-numbers to get higher than double precision, see the examples.}

\( \ldots \) \hspace{1cm} \text{additional arguments to be passed to} \( f \).

\( \text{ord} \) \hspace{1cm} \text{integer, the order of Romberg integration to be used. If this is NULL, as per default, and either rel.tol or abs.tol are specified, the order is increased until convergence.}

\( \text{rel.tol} \) \hspace{1cm} \text{relative accuracy requested. The default is 1.2e-4, about 4 digits only, see the Note.}

\( \text{abs.tol} \) \hspace{1cm} \text{absolute accuracy requested.}

\( \text{max.ord} \) \hspace{1cm} \text{only used, when neither ord or one of rel.tol, abs.tol are specified: Stop Romberg iterations after the order reaches max.ord; may prevent infinite loops or memory explosion.}

\( \text{verbose} \) \hspace{1cm} \text{logical or integer, indicating if and how much information should be printed during computation.}

Details

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

For convergence, \textit{both} relative and absolute changes must be smaller than rel.tol and abs.tol, respectively.

rel.tol cannot be less than max(50*Machine$double.eps,0.5e-28) if abs.tol <= 0.

Value

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

- \texttt{value} \hspace{1cm} \text{the final estimate of the integral.}
- \texttt{abs.error} \hspace{1cm} \text{estimate of the modulus of the absolute error.}
- \texttt{subdivisions} \hspace{1cm} \text{for Romberg, the number of function evaluations.}
- \texttt{message} \hspace{1cm} \text{"OK" or a character string giving the error message.}
- \texttt{call} \hspace{1cm} \text{the matched call.}

Note

\( f \) must accept a vector of inputs and produce a vector of function evaluations at those points. The \texttt{Vectorize} function may be helpful to convert \( f \) to this form.

If you want to use higher accuracy, you \textit{must} set lower or upper to "mpfr" numbers (and typically lower the relative tolerance, rel.tol), see also the examples.

Note that the default tolerances (rel.tol, abs.tol) are not very accurate, but the same as for integrate, which however often returns considerably more accurate results than requested. This is typically \textit{not} the case for integrateR().
Note

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

Author(s)

Martin Maechler

References


See Also

\textsf{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,dnorm,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)
```
## 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[] are integer valued aka “whole” numbers, including MPFR numbers (class mpfr).

### Usage

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

### Arguments

- **x**  
  any R vector, here of class mpfr.

### Value

logical vector of the same length as x, indicating where x[i] 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
```
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:
sediff(mNsub, Mnsub)# namely "numericVector"
## The following are only subclasses of "Mnumber", but not of "mNumber":
sediff(Mnsub, mNsub)
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"))

Arguments

x
precBits, prec a numeric, mpfr, bigz, bigq, or character vector or array.
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.
(only when x is character) the base with respect to which x[i] represent numbers; base b must fulfill 2 ≤ b ≤ 62.
a 1-letter string specifying how rounding should happen at C-level conversion to MPFR, see details.
(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.
a string specifying the mpfr - internal constant computation. "gamma" is Euler's gamma (γ), 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_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.

**Author(s)**

Martin Maechler

**References**


**See Also**

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

**Examples**

```r
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. 44833.003694653820642

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**

### 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.,

\[
z = hypot(x, y) = \sqrt{x^2 + y^2},
\]

  in a numerically stable way.

**Usage**

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

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: atan2(y,x) returns the angle between the x-axis and the vector from the origin to (x,y), i.e., for positive arguments atan2(y,x) == atan(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 ≥ 0 is required. Here, we provide (non-NaN) for all numeric a,b. When either a, b, or a + b is a negative integer, Γ(.) 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"): ...

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, cosp, sinh, tan, tanh, acos, acosh, asin, asinh, atan, atanh, 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.

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

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

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 × 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(.) (= ζ(.)), Ei() etc. Bernoulli numbers and the Pochhammer function pochMpfr.

Examples

```r
## 30 digit precision
str(x <- mpfr(c(2:3, pi), prec = 30 * log2(10)))
x^2
x[1] / x[2] # 0.66666... ~ 30 digits

## indexing - as with numeric vectors
stopifnot(identical(x[2], x[[2]]),
  ## indexing "outside" gives NA (well: "mpfr-NaN" for now):
  is.na(x[5]),
  ## whereas "[[" cannot index outside:
  is(try(x[[5]]), "try-error"),
  ## and only select *one* element:
  is(try(x[[2:3]]), "try-error"))

## factorial() & lfactorial would work automagically via [l]gamma(),
## 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

## 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))
## 2.38256490488795107e-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(diffl(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),# quantiles: interpolated in orig. <-> log scale
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> :
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
## 5.66520587824e-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)))))
})

## ((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 etc (MPFR)**

**Description**

For some \texttt{R} standard (probability) density, distribution or quantile functions, we provide MPFR versions.
Usage

\[
\begin{align*}
dpois(x, \lambda, \log = FALSE) \\
dbinom(x, n, p, \log = FALSE) \\
dnorm(x, \mu, \sigma, \log = FALSE) \\
dgamma(x, \alpha, \beta, \log = FALSE) \\
pnorm(q, \mu, \sigma, \text{lower.tail} = TRUE, \log.p = FALSE)
\end{align*}
\]

Arguments

\[
x, q, \lambda, n, p, \mu, \sigma, \alpha, \beta
\]
numeric or \texttt{mpfr} vectors. All of these are “recycled” to the length of the longest one. For their meaning/definition, see the corresponding standard R (\texttt{stats} package) function.

\[
\log, \log.p, \text{lower.tail}
\]
logical, see \texttt{pnorm, dpois}, etc.

Details

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

Value

A vector of the same length as the longest of \(x, q, \ldots\), of class \texttt{mpfr} with the high accuracy results of the corresponding standard R function.

See Also

\texttt{pnorm, dbinom, dgamma, dpois} in standard package \texttt{stats}.

\texttt{pbetaI(x,a,b)} is a \texttt{mpfr} version of \texttt{pbeta} only for integer \(a\) and \(b\).

Examples

\[
x <- 1400 + 0:10 \\
\text{print(dpois(x, 1000), digits =18)} \quad \text{## standard R's double precision} \\
\text{dpois(mpfr(x, 120), 1000)} \quad \text{## more accuracy for the same} \\
\text{dpois(0:5, mpfr(10000, 80))} \quad \text{## very small exponents} \\
\text{print(dbinom(0:8, 8, pr = 4 / 5), digits=18)} \\
\text{dbinom(0:8, 8, pr = 4/mpfr(5, 99)) \rightarrow dB; dB} \\
\text{print(dnorm(-5:5), digits=18)} \\
\text{dnorm(mpfr(-5:5, prec=99))}
\]
Special Mathematical Functions (MPFR)

Description
Special Mathematical Functions, supported by the MPFR Library.

Usage
zeta(x)
Ei(x)
Li2(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

\[
\zeta(x) = \sum_{n=1}^{\infty} \frac{1}{n^x}.
\]

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 \texttt{pnorm}, \( \texttt{erf}(x) = 2 \times \texttt{pnorm}(\sqrt{2} \times x) \) and \( \texttt{erfc}(x) = 2 \times \texttt{pnorm}(\sqrt{2} \times x, \text{lower=FALSE}) \), and hence \texttt{Rmpfr} provides its own version of \texttt{pnorm}.

Value
A vector of the same length as \( x \), of class \texttt{mpfr}.
See Also

`pnorm` in standard package `stats`; the class description `mpfr` mentioning the generic arithmetic and mathematical functions (\(\text{sin}, \text{log}, \ldots\), etc) for which "mpfr" methods are available.

Note the (integer order, non modified) Bessel functions \(j_0(), y_n()\), etc, named \(j_0, y_n\) 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

```r
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),
)```
...)
toBeNum(from, rnd.mode = c('N', 'D', 'U', 'Z', 'A'))
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()

Arguments

x, from typically, 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 respect to which x[i] 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 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

gPrec(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().

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

gGetD(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.

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_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

Note

mpfrXport() and mpfrImport() are experimental and used to explore reported platform incompatibilities of save()d and load()ed "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

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)
```r
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

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 "current" MPFR versions:
eLimits == c(-1,1, -1,1) * (2^62 - 1)

## Looking at internal representation [for power users only!]:
i8 <- mpfr(-2:5, 32)
x4 <- mpfr(c(NA, NaN, -Inf, Inf), 32)
## The output of the following depends on the GMP "numb" size
## (32 bit vs. 64 bit), and may be even more platform specifics:
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))))

stopifnot(identical(B6, mpfrImport(mpfrXport(B6))))
```

### mpfr.utils

**MPFR Number Utilities**

### Description

`mpfrVersion()` returns the version of the MPFR library which `Rmpfr` is currently linked to.
\texttt{c(x,y,...)} can be used to combine MPFR numbers in the same way as regular numbers \textbf{IFF} the first argument \texttt{x} is of class \texttt{mpfr}.

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

\texttt{.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 \texttt{x == round(x)}, but \textit{not} at all to \texttt{is.integer(as(x,"numeric")).}

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

\textbf{Usage}

\begin{verbatim}
mpfrVersion()
mpfrIs0(x)
## S3 method for class 'mpfr'
c(...)  # S3 method for class 'mpfr'
diff(x, lag = 1L, differences = 1L, ...)
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \texttt{x} an object of class \texttt{mpfr}.
  \item \texttt{...} for \texttt{diff}, further \texttt{mpfr} class objects or simple numbers (\texttt{numeric} vectors) which are coerced to \texttt{mpfr} with default precision of 128 bits.
  \item \texttt{lag, differences} for \texttt{diff}(): exact same meaning as in \texttt{diff()}’s default method, \texttt{diff.default}.
\end{itemize}

\textbf{Value}

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

Similarly, \texttt{.mpfr.is.whole} and \texttt{is.whole} return a logical vector of length \texttt{length(x)}.

\texttt{mpfrVersion} returns an object of S3 class "\texttt{numeric\_version}", so it can be used in comparisons.

The other functions return MPFR number (vectors), i.e., extending class \texttt{mpfr}.

\textbf{See Also}

\texttt{str.mpfr} for the \texttt{str} method. \texttt{erf} for special mathematical functions on MPFR.

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

\textbf{Examples}

\begin{verbatim}
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..
\end{verbatim}
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

```r
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(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 matrix and array 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 an 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 mpfr1 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[].

mpfrMatrix

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"): ...
[c] signature(x = "mpfrArray", i = "ANY", j = "ANY", value = "ANY"): ...
[c] signature(x = "mpfrArray", i = "ANY", j = "ANY", drop = "ANY"): ...
[c] 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.
crossprod signature(x = "mpfrMatrix", y = "mpfrMatrix"): ...
crossprod signature(x = "mpfrMatrix", y = "mpfr"): ...
tcrossprod signature(x = "mpfr", y = "missing"): Computes xx’, i.e., x %*% t(x), typically more efficiently.
tcrossprod signature(x = "mpfrMatrix", y = "mpfrMatrix"): Computes xy’, i.e., x %*% 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"): tranpose 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))
i13 <- Rmpfr:::.mat2ind(i3, dim(AA), dimnames(AA))
stopifnot(aa[i13] == new("mpfr", getD(AA)[i13]))
stopifnot(identical(aa[i13], asNumeric(AA[i13])))
CA <- AA; ca <- aa
c[i13] <- ca[i13] ^ 3
CA[i13] <- CA[i13] ^ 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, "1") == 6,
          abs(norm(S2, "F") - 5.477225575051661) < 1e-15)

---

**mpfrMatrix-utils**

*Functions for mpfrMatrix Objects*

**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!)

## 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

```r
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 \texttt{Rmpfr} arithmetic” for \texttt{optimizer()} to make sense. The function is either minimized or maximized over its first argument depending on the value of \texttt{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., \( \text{tol} < 2e^{-16} \).
- **method**: character string specifying the optimization method.
- **maximum**: logical indicating if \( f() \) should be maximized or minimized (the default).
- **precFactor**: only for default \texttt{precBits} construction: a factor to multiply with the number of bits directly needed for \text{tol}.
- **precBits**: number of bits to be used for \texttt{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 \texttt{R}'s \texttt{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 W Borchert’s golden_ratio; modifications and "Brent" by Martin Maechler.

**See Also**

R’s standard optimize; 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)
pil <- 2*sol$min/3
print(pil, digits=51)
stopifnot(all.equal(pil, Const("pi", 256), tolerance = 10*1e-50))
```

---

*optimizeR*

53

**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 W Borchert’s golden_ratio; modifications and "Brent" by Martin Maechler.

**See Also**

R’s standard optimize; 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)
pil <- 2*sol$min/3
print(pil, digits=51)
stopifnot(all.equal(pil, 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 7.9999038 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.

**Usage**

pbetaI(q, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE, precBits = NULL, rnd.mode = c("N","D","U","Z","A"))
**Arguments**

- **q**: called \( x \), above; vector of quantiles, in \([0, 1]\).
- **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()`.
- **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

**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) ## still slow: % TOO slow -- FIXME
pp <- pbetaI(x, a, b, precBits = 2048)
## Currently, the lower.tail=FALSE are computed "badly":
lp <- log(pp)  ### = pbetaI(x, a, b, log.p=TRUE)
llp <- log1p(-pp)  ### = pbetaI(x, a, b, lower.tail=FALSE, log.p=TRUE)
lp <- 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(llp, 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))
)

---

**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**

```r
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))

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)
roundMpfr  
*Rounding to Binary bits, “mpfr-internally”*

**Description**

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**

```r
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**

```r
(p1 <- Const("pi", 100)) # 100 bit prec
roundMpfr(p1, 120) # 20 bits more, but "random noise"
Const("pi", 120)  # same "precision", but really precise
```

sapplyMpfr  
*Apply a Function over a "mpfr" Vector*

**Description**

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().

Note that this is not yet as flexible as sapply() for atomic vectors.
seqMpfr

Usage

sapplyMpfr(X, FUN, ...)

Arguments

- `X`: a vector, possibly of class "mpfr".
- `FUN`: a function returning an "mpfr" number. (TODO: A function returning a vector of mpfr numbers or even "mpfrArray").
- `...`: further arguments passed to `lapply`, typically further arguments to `FUN`.

Value

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

Note

Another workaround could be to use

```
res <- lapply(....)
sapply(res, asNumeric)
```

Author(s)

Martin Maechler

Examples

```r
## The function is simply defined as
function (X, FUN, ...)
  new("mpfr", unlist(lapply(X, FUN, ...), recursive = FALSE))

if(require("Bessel")) # here X, is simple
  bImp <- sapplyMpfr(0:4, function(k)
    besselI.nuAsym(mpfr(1.31e9, 128), 10, expon.scaled=TRUE, k.max=k))
```

seqMpfr

"mpfr" Sequence Generation

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.

str.mpfr

Compactly Show STRucture of Rmpfr Number Object

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, drop0trailing=TRUE,
   width = getOption("width"), ...)
sumBinomMpfr

(Alternating) Binomial Sums via Rmpfr

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

```r
sumBinomMpfr(n, f, n0 = 0, alternating = TRUE, precBits = 256,
             f.k = f(mpfr(k, precBits=precBits)))
```

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, \( f(mpfr(k,precBits=precBits)) \).

Details

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

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

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 \( s_B(f,n) = sumBinomMpfr(n,f,* \) 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

```
unirootR(f, interval, ..., 
  lower = min(interval), upper = max(interval),
  f.lower = f(lower, ...), f.upper = f(upper, ...),
  verbose = FALSE,
  tol = .Machine$double.eps^0.25, maxiter = 1000,
  warn.no.convergence = TRUE,
  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.
- `verbose` logical (or integer) indicating if (and how much) verbose output should be produced during the iterations.
- `tol` the desired accuracy (convergence tolerance).
- `maxiter` the maximum number of iterations.
- `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 $2^{-\min\{\text{getPrec}(f\_{\text{lower}}),\text{getPrec}(f\_{\text{upper}})\}}$ 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 sense 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).

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 \( \text{tol} \) (plus an allowance for representation error in \( x \)).

If the algorithm does not converge in \( \text{maxiter} \) 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: \( \text{root} \) and \( f.\text{root} \) give the location of the root and the value of the function evaluated at that point. \( \text{iter} \) and \( \text{estim.prec} \) give the number of iterations used and an approximate estimated precision for \( \text{root} \). (If the root occurs at one of the endpoints, the estimated precision is NA.)

Source

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

References

Brent, R. (1973), see \texttt{uniroot}.

See Also

\texttt{polyroot} for all complex roots of a polynomial; \texttt{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)
rcM
x0 <- rcM$root
stopifnot(all.equal(cos(x0), x0,
              tol = 1e-40))## 40 digits accurate!

str(unirootR(function(x) x*(x^2-1) + .5, lower = -2, upper = 2,
\[
\text{tol} = 0.0001), \text{digits.d} = 10
\]
\[
\text{str(unirootR(function(x) x*(x^2-1) + .5, lower = -2, upper = 2,
\text{tol = 1e-10 }, \text{digits.d = 10})}
\]

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

## Find the smallest value \( x \) for which \( \exp(x) > 0 \) (numerically):
\[
\text{r <- unirootR(function(x) 1e80*exp(x)-1e-300, c(-1000,0), tol = 1e-15)}
\text{str(r, digits.d = 15) ##> around -745, depending on the platform.}
\[
\text{exp(r$root) # = 0, but not for r$root * 0.999...}
\text{minexp <- r$root * (1 - 10*.Machine$double.eps)}
\text{exp(minexp) # typically denormalized}
## --- using mpfr-numbers :

## Find the smallest value \( x \) for which \( \exp(x) > 0 \) ("numerically");
## Note that mpfr-numbers underflow *MUCH* later than doubles:
## one of the smallest mpfr-numbers (see also ?mpfr-class ) :
\[
(\text{ep.M <- mpfr(2, 55) ^ - ((2^30 + 1) * (1 - 1e-15))})
\text{r <- unirootR(function(x) 1e99* exp(x) - ep.M, mpfr(c(-1e20, 0), 200))}
\text{r # 97 iterations; f.root is very similar to ep.M}
**Index**

*Topic arith
Bernoulli, 8
chooseMpfr, 12
factorialMpfr, 14
formatHex, 15
gmp-conversions, 20
mpfr.utils, 44
pbetaI, 54
pmax, 56
roundMpfr, 58
sumBinomMpfr, 61

*Topic array
mpfrArray, 46
mpfrMatrix-utils, 50

*Topic character
formatMpfr, 18

*Topic classes
array_or_vector-class, 6
atomicVector-class, 8
Mnumber-class, 29
mpfr, 30
mpfr-class, 32
mpfrMatrix, 47

*Topic distribution
mpfr-distr-etc, 37
pbetaI, 54

*Topic manip
sapplyMpfr, 58
seqMpfr, 59

*Topic math
Bessel_mpfr, 10
igamma, 24
integrateR, 25
is.whole, 28
mpfr-special-functions, 39

*Topic methods
asNumeric-methods, 7
bind-methods, 11
pmax, 56

*Topic misc
Rmpfr-workarounds, 57

*Topic optimize
hjkMpfr, 21
optimizeR, 52
unirootR, 63

*Topic package
Rmpfr-package, 3

*Topic print
formatMpfr, 18

*Topic univar
pmax, 56

*Topic utilities
integrateR, 25
mpfr-utils, 40
str.mpfr, 60
.Machine, 35, 43, 64
.bigq2mpfr (gmp-conversions), 20
.bigz2mpfr (gmp-conversions), 20
.getPrec (mpfr-utils), 40
.mpfr.is.whole (mpfr.utils), 44
.mpfr2bigz (gmp-conversions), 20
.mpfr2exp, 19
.mpfr2exp (mpfr-utils), 40
.mpfr2list, 61
.mpfr2list (mpfr-utils), 40
.mpfr2str, 18, 43
.mpfr2str (formatMpfr), 18
.mpfr_erange (mpfr-utils), 40
.mpfr_erange_is_int (mpfr-utils), 40
.mpfr_erange_kinds (mpfr-utils), 40
.mpfr_erange_set (mpfr-utils), 40
.mpfr_formatinfo, 19
.mpfr_formatinfo (mpfr-utils), 40
.mpfr_maxPrec (mpfr-utils), 40
.mpfr_minPrec (mpfr-utils), 40
[,mpfr,ANY,missing,missing-method (mpfr-class), 32
[,mpfrArray,ANY,ANY,ANY-method
(mpfrMatrix), 47
[, mpfrArray, ANY, missing, missing-method
(mpfrMatrix), 47
[, mpfrArray, matrix, missing, missing-method
(mpfrMatrix), 47
<-, mpfr, ANY, missing, ANY-method
(mpfr-class), 32
<-, mpfr, ANY, missing, mpfr-method
(mpfr-class), 32
<-, mpfr, missing, missing, ANY-method
(mpfr-class), 32
<-, mpfrArray, ANY, ANY, ANY-method
(mpfrMatrix), 47
<-, mpfrArray, ANY, ANY, mpfr-method
(mpfrMatrix), 47
<-, mpfrArray, ANY, missing, ANY-method
(mpfrMatrix), 47
<-, mpfrArray, ANY, missing, mpfr-method
(mpfrMatrix), 47
<-, mpfrArray, matrix, missing, ANY-method
(mpfrMatrix), 47
<-, mpfrArray, matrix, mpfr-method
(mpfrMatrix), 47
<-, mpfrArray, missing, ANY, ANY-method
(mpfrMatrix), 47
<-, mpfrArray, missing, ANY, mpfr-method
(mpfrMatrix), 47
<-, mpfrArray, missing, missing, ANY-method
(mpfrMatrix), 47
<-, mpfrArray, missing, missing, mpfr-method
(mpfrMatrix), 47
[[, mpfr-method (mpfr-class), 32
**%, Mnumber, mpfr-method (mpfrMatrix), 47
**%, array_or_vector, mpfr-method
(mpfr-class), 32
**%, mpfr, Mnumber-method (mpfrMatrix), 47
**%, mpfr, array_or_vector-method
(mpfr-class), 32
**%, mpfr, mpfr-method (mpfrMatrix), 47
**%, mpfr, mpfrMatrix-method
(mpfrMatrix), 47
**%, mpfrMatrix, mpfrMatrix-method
(mpfrMatrix), 47
**%, 48
abs, 34, 51
abs, mpfr-method (mpfr-class), 32
acos, 34
acosh, 34
Ai (Bessel_mpfr), 10
all, 34
all.equal, 35
all.equal, ANY, mpfr-method (mpfr-class), 32
all.equal, mpfr, ANY-method (mpfr-class), 32
all.equal, mpfr, mpfr-method (mpfr-class), 32
any, 34
aperm, 49
aperm, mpfrArray-method (mpfrMatrix), 47
apply, mpfrArray-method (mpfrMatrix), 47
Arg, mpfr-method (mpfr-class), 32
Arith, array, mpfr-method (mpfr-class), 32
Arith, integer, mpfr-method (mpfr-class), 32
Arith, mpfr, array-method (mpfr-class), 32
Arith, mpfr, integer-method (mpfr-class), 32
Arith, mpfr, missing-method (mpfr-class), 32
Arith, mpfr, mpfr-method (mpfr-class), 32
Arith, mpfr, mpfrArray-method (mpfrMatrix), 47
Arith, mpfr, numeric-method (mpfr-class), 32
Arith, mpfrArray, mpfr-method
(mpfrMatrix), 47
Arith, mpfrArray, mpfrArray-method
(mpfrMatrix), 47
Arith, mpfrArray, numeric-method
(mpfrMatrix), 47
Arith, numeric, mpfr-method (mpfr-class), 32
Arith, numeric, mpfrArray-method
(mpfrMatrix), 47
array, 5, 7, 19, 30, 34, 43, 46–49
array_or_vector, 29
array_or_vector-class, 6
as, 35
as.bigq, 21
as.bigz, 21
as.integer, mpfr-method (mpfr-class), 32
as.numeric, 7
as.numeric, mpfr-method (mpfr-class), 32
as.vector, mpfrArray, missing-method
(mpfrMatrix), 47
as.vector, mpfrArray-method
(mpfr-class), 32
asin, 34
asinh, 34
asNumeric, mpfr-method
(asNumeric-methods), 7
asNumeric, mpfrArray-method
(asNumeric-methods), 7
asNumeric-methods, 7
atan, 33, 34
atan2, ANY, mpfr-method
(mpfr-class), 32
atan2, ANY, mpfrArray-method
(mpfr-class), 32
atan2, mpfr, ANY-method
(mpfr-class), 32
atan2, mpfr, mpfr-method
(mpfr-class), 32
atan2, mpfr, numeric-method
(mpfr-class), 32
atan2, mpfrArray, ANY-method
(mpfr-class), 32
atan2, mpfrArray, mpfrArray-method
(mpfr-class), 32
atan2, numeric, mpfr-method
(mpfr-class), 32
atanh, 34
atomicVector-class, 8
base::pmin, 57
Bernoulli, 5, 8, 36
Bessel_mpfr, 5, 10, 40
besselJ, 10
besselY, 10
beta, 33
beta, ANY, mpfr-method (mpfr-class), 32
beta, ANY, mpfrArray-method (mpfr-class), 32
beta, mpfr, ANY-method (mpfr-class), 32
beta, mpfr, mpfr-method (mpfr-class), 32
beta, mpfr, numeric-method (mpfr-class), 32
beta, mpfrArray, ANY-method (mpfr-class), 32
beta, mpfrArray, mpfrArray-method
(mpfr-class), 32
beta, numeric, mpfr-method (mpfr-class), 32
bigq, 20, 30
bigrational, 6
bigz, 20, 30, 35
bind-methods, 11
c, 45
c.mpfr, 5
c.mpfr (mpfr.utils), 44
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, 34
character, 30, 31, 35, 41, 42, 47, 52
choose, 12, 13
chooseMpfr, 5, 12, 63
chooseZ, 12, 13, 55, 63
class, 8, 11, 16, 28, 32, 41, 56
coerce, array, mpfr-method (mpfr-class), 32
coerce, bigq, mpfr-method
(gmp-conversions), 20
coerce, bigz, mpfr-method
(gmp-conversions), 20
coerce, character, mpfr-method
(mpfr-class), 32
coerce, integer, mpfr-method
(mpfr-class), 32
coerce, logical, mpfr-method
(mpfr-class), 32
coerce, mpfr, bigz-method (mpfr-class), 32
coerce, mpfr, character-method
(mpfr-class), 32
coerce, mpfr, integer-method
(mpfr-class), 32
coerce, mpfr, mpfr1-method (mpfr-class), 32
coerce, mpfr, numeric-method
(mpfr-class), 32
coerce, mpfr1, mpfr-method (mpfr-class), 32
coerce, mpfr1, numeric-method
(mpfr-class), 32
coerce, mpfrArray, array-method
(mpfrMatrix), 47
coerce, mpfrArray, matrix-method
(mpfrMatrix), 47
coerce, mpfrArray, vector-method (mpfrMatrix), 47
coerce, mpfrMatrix, matrix-method (mpfrMatrix), 47
coerce, numeric, mpfr-method (mpfr-class), 32
coerce, numeric, mpfr1-method (mpfr-class), 32
coerce, raw, mpfr-method (mpfr-class), 32
coerce<-, mpfrArray, vector-method (mpfrMatrix), 47
colMeans, mpfrArray-method (mpfrMatrix), 47
colSums, mpfrArray-method (mpfrMatrix), 47
Compare, array, mpfr-method (mpfr-class), 32
Compare, integer, mpfr-method (mpfr-class), 32
Compare, mpfr, array-method (mpfr-class), 32
Compare, mpfr, integer-method (mpfr-class), 32
Compare, mpfr, mpfr-method (mpfr-class), 32
Compare, mpfr, mpfrArray-method (mpfrMatrix), 47
Compar, mpfr, numeric-method (mpfr-class), 32
Compare, mpfrArray, mpfr-method (mpfrMatrix), 47
Compare, mpfrArray, numeric-method (mpfrMatrix), 47
Compare, numeric, mpfr-method (mpfr-class), 32
Compare, numeric, mpfrArray-method (mpfrMatrix), 47
cummax, 34
cummin, 34
cumprod, 34
cumsum, 34
dbinom, 38
dbinom (mpfr-distr-etc), 37
determinant, 51
determinant.mpfrMatrix (mpfrMatrix-utils), 50
dgamma, 38
dgamma (mpfr-distr-etc), 37
diag, mpfrMatrix-method (mpfrMatrix), 47
diag<-, mpfrMatrix-method (mpfrMatrix), 47
diff, 45
diff.default, 45
diff.mpfr (mpfr.utils), 44
digamma, 5, 34
dim, 7, 19, 33, 48
dim, mpfrArray-method (mpfrMatrix), 47
dim<-, mpfr-method (mpfr-class), 32
dimnames, 19
dimnames, mpfrArray-method (mpfrMatrix), 47
dimnames<-, mpfrArray-method (mpfrMatrix), 47
dnorm (mpfr-distr-etc), 37
dotsMethods, 11
double, 41, 43
dpois, 38
dpois (mpfr-distr-etc), 37
duplicated, 35
Ei (mpfr-special-functions), 39
erf, 38, 45
erf (mpfr-special-functions), 39
erfc (mpfr-special-functions), 39
exp, 34
expm1, 34

factorial, 13, 14
factorial, mpfr-method (mpfr-class), 32
factorialMpf, 5, 13, 14, 34
factorialZ, 14
floor, 34
format, 16, 18, 19, 42
format, mpfr-method (mpfr-class), 32
formatBin, 30
formatBin (formatHex), 15
formatDec (formatHex), 15
formatHex, 15
formatMpf, 5, 16, 18, 35, 41, 42, 61
formatN, 19
formatN.mpfr (formatMpf), 18
function, 62
gamma, 5, 13, 14, 25, 33, 34
gmp, 6
gmp-conversions, 20
golden_ratio, 53
hjk, 21
hjkMpf, 5, 21
hypot (mpfr-class), 32
igamma, 24
Im, mpfr-method (mpfr-class), 32
integer, 8, 12, 35, 41–43
integrate, 26, 27
integrateR, 5, 25
interactive, 41
is.atomic, 8
is.finite, 43
is.finite, mpfr-method (mpfr-class), 32
is.finite, mpfrArray-method (mpfr-class), 32
is.infinite, mpfr-method (mpfr-class), 32
is.infinite, mpfrArray-method (mpfr-class), 32
is.integer, 28
is.na, mpfr-method (mpfr-class), 32

is.na, mpfrArray-method (mpfr-class), 32
is.nan, mpfr-method (mpfr-class), 32
is.nan, mpfrArray-method (mpfr-class), 32
is.whole, 28, 36, 45
j0, 40
j0 (Bessel_mpf), 10
j1 (Bessel_mpf), 10
jn, 36
jn (Bessel_mpf), 10
lapply, 58, 59
lbeta, ANY, mpfr-method (mpfr-class), 32
lbeta, ANY, mpfrArray-method (mpfr-class), 32
lbeta, mpfr, ANY-method (mpfr-class), 32
lbeta, mpfr, mpfr-method (mpfr-class), 32
lbeta, mpfr, numeric-method (mpfr-class), 32
lbeta, mpfrArray, ANY-method (mpfr-class), 32
lbeta, mpfrArray, mpfrArray-method (mpfr-class), 32
lbeta, numeric, mpfr-method (mpfr-class), 32
lgamma, 33, 34
Li2 (mpfr-special-functions), 39
list, 22, 33, 48, 51, 53, 61
load, 43
log, 34, 51
log, mpfr-method (mpfr-class), 32
log10, 34
log1p, 34
log2, 34
Logic, mpfr, mpfr-method (mpfr-class), 32
Logic, mpfr, numeric-method (mpfr-class), 32
Logic, numeric, mpfr-method (mpfr-class), 32
logical, 15, 18, 19, 35, 42, 43
Math, 34
Math, mpfr-method (mpfr-class), 32
Math2, mpfr-method (mpfr-class), 32
matrix, 7, 43, 47
max, 34
mean, 34
mean, mpfr-method (mpfr-class), 32
mean.default, 34
INDEX

range, 34, 57
rank, 35
raw, 35
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), 32
Rmpfr (Rmpfr-package), 3
Rmpfr-package, 3
Rmpfr-workarounds, 57
round, 34, 58
roundMpfr, 5, 30, 34, 36, 42, 58
rowMeans,mpfrArray-method (mpfrMatrix), 47
rowSums,mpfrArray-method (mpfrMatrix), 47
sapply, 58
sapplyMpfr, 58
save, 43
seq, 59, 60
seqMpfr, 5, 59
setPrec (roundMpfr), 58
show,integrateR-method (integrateR), 25
show,mpfr-method (mpfr-class), 32
show,mpfr1-method (mpfr-class), 32
show,mpfrArray-method (mpfrMatrix), 47
show,summaryMpfr-method (mpfr-class), 32
sign, 33, 34
sign,mpfr-method (mpfr-class), 32
sign,mpfrArray-method (mpfrMatrix), 47
signif, 34
sin, 34
sinh, 34
sort, 35
sprintf, 15, 16, 19
sqrt, 34
str, 45, 60, 61
str.default, 60
str.mpfr, 45, 60
sum, 34
sumBinomMpfr, 5, 13, 55, 61
Summary, 34
Summary,mpfr-method (mpfr-class), 32
summary,mpfr-method (mpfr-class), 32
summary.default, 34
summaryMpfr-class (mpfr-class), 32
t,mpfr-method (mpfr-class), 32
t,mpfrMatrix-method (mpfrMatrix), 47
tan, 34
tanh, 34
tcrossprod,array_or_vector,mpfr-method (mpfr-class), 32
tcrossprod,Mnumber,mpfr-method (mpfrMatrix), 47
tcrossprod,mpfr,array_or_vector-method (mpfr-class), 32
tcrossprod,mpfr,missing-method (mpfrMatrix), 47
tcrossprod,mpfr,Mnumber-method (mpfrMatrix), 47
tcrossprod,mpfr,mpfr-method (mpfrMatrix), 47
tcrossprod,mpfrMatrix,mpfr-method (mpfrMatrix), 47
tcrossprod,mpfrMatrix,mpfrMatrix-method (mpfrMatrix), 47
toNum, 7
toNum (mpfr-utils), 40
trigamma, 5, 34
trunc, 34
typeof, 7, 32
unique, 35
unique,mpfr,ANY-method (mpfr-class), 32
unique,mpfr-method (mpfr-class), 32
unique.mpfr (mpfr-class), 32
uniroot, 63, 65
unirootR, 5, 53, 63
vapply, 58
vector, 48
Vectorize, 26
which.max, 35
which.max,mpfr-method (mpfr-class), 32
which.min, 35
which.min,mpfr-method (mpfr-class), 32
y0 (Bessel_mpfr), 10
y1 (Bessel_mpfr), 10
yn, 40
yn (Bessel_mpfr), 10
zeta, 5, 8, 9, 36
zeta(mpfr-special-functions), 39