

Package ‘Rmpfr’

May 23, 2012

Type Package

Title R MPFR - Multiple Precision Floating-Point Reliable

Version 0.4-8

Date 2012-05-22

Author Martin Maechler

Maintainer Martin Maechler <maechler@stat.math.ethz.ch>

SystemRequirements gmp (>= 4.2.3), mpfr (>= 3.0.0)

SystemReqsNotes MPFR (MP Floating-Point Reliable Library,<http://mpfr.org/>) and GMP (GNU Multiple Precision library,<http://gmplib.org/>), see README

Depends methods, gmp (>= 0.5-2), R (>= 2.12.0)

Imports gmp, stats, utils

Suggests MASS, polynom, sfsmisc

SuggestNotes MASS, polynom, sfsmisc are only needed for vignette

URL <http://rmpfr.r-forge.r-project.org/>

Description Rmpfr provides S4 classes and methods for arithmetic including transcendental (“special”) functions for arbitrary precision floating point 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.

License GPL (>= 2)

Repository CRAN

Date/Publication 2012-05-23 09:35:22

R topics documented:

array_or_vector-class	2
atomicVector-class	3
Bernoulli	4
Bessel_mpfr	5
bind-methods	6
chooseMpfr	7
factorialMpfr	8
formatMpfr	9
gmp-conversions	11
integrateR	12
is.whole	14
Mnumber-class	15
mpfr	16
mpfr-class	17
mpfr-special-functions	21
mpfr-utils	22
mpfr.utils	24
mpfrArray	26
mpfrMatrix	27
pmax	29
roundMpfr	30
seqMpfr	31
sumBinomMpfr	32
unirootR	34
Index	37

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

```
showClass("array_or_vector")
```

atomicVector-class *Virtual Class "atomicVector" of Atomic Vectors*

Description

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

Objects from the Class

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

Methods

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

Extends

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

Author(s)

Martin Maechler

See Also

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

Examples

```
showClass("atomicVector")
```

Description

Computes the Bernoulli numbers in the desired (binary) precision. The computation happens via the [zeta](#) function and the formula

$$B_k = -k\zeta(1 - k),$$

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

Usage

```
Bernoulli(k, precBits = 128)
```

Arguments

k	non-negative integer vector
precBits	the precision in <i>bits</i> 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_mpfr

*Bessel functions of Integer Order in multiple precisions***Description**

Bessel functions of integer orders, provided via arbitrary precision algorithms from the MPFR library.

Usage

```

Ai(x)
j0(x)
j1(x)
jn(n, x)
y0(x)
y1(x)
yn(n, x)

```

Arguments

x	a numeric or mpfr vector.
n	non-negative integer (vector).

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 $Ai(x)$.

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

```

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(yn(0,x), bY0)
           , all.equal(jn(1,x), bJ1)
           )

```

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|r)bind multiprecision "numbers" (inheriting from class `"mpfr"`) together, maybe combined with simple numeric vectors.

`... = "ANY"` reverts to [cbind](#) and `rbind` from package **base**.

Author(s)

Martin Maechler

See Also

[cbind2](#), [cbind](#), [Methods](#).

Examples

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

Description

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

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

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

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

Usage

```
chooseMpfr(a, n)
chooseMpfr.all(n, precBits=NULL, k0=1, alternating=FALSE)
pochMpfr(a, n)
```

Arguments

<code>a</code>	a numeric or <code>mpfr</code> vector.
<code>n</code>	an integer vector; if not of length one, <code>n</code> and <code>a</code> are recycled to the same length.
<code>precBits</code>	integer or <code>NULL</code> for increasing the default precision of the result.
<code>k0</code>	integer scalar
<code>alternating</code>	logical, for <code>chooseMpfr.all()</code> , indicating if <i>alternating sign</i> coefficients should be returned, see below.

Value

For

`chooseMpfr()`, `pochMpfr()`: an `mpfr` vector of length `max(length(a), length(n))`;

`chooseMpfr.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, `factorialMpfr`.

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

Examples

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

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

## --- Experimenting & Checking
n.set <- c(1:10, 20, 50:55, 100:105, 200:203, 300:303, 500:503,
          699:702, 999:1001)
C1 <- C2 <- numeric(length(n.set))
for(i.n in seq_along(n.set)) {
  cat(n <- n.set[i.n], ":")
  C1[i.n] <- system.time(c.c <- chooseMpfr.all(n) )[1]
  C2[i.n] <- system.time(c.2 <- chooseMpfr(n, 1:n) )[1]
  stopifnot(is.whole(c.c), c.c == c.2,
            if(n > 60) TRUE else all.equal(c.c, choose(n, 1:n), tol = 1e-15))
  cat(" [0k]\n")
}
matplot(n.set, cbind(C1,C2), type="b", log="xy",
        xlab = "n", ylab = "system.time(.) [s]")
legend("topleft", c("chooseMpfr.all(n)", "chooseMpfr(n, 1:n)"),
      pch=as.character(1:2), col=1:2, lty=1:2, bty="n")

## Currently, chooseMpfr.all() is faster only for large n (>= 300)
## That would change if we used C-code for the *.all() version
```

factorialMpfr

Factorial 'n!' in Arbitrary Precision

Description

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

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

Usage

```
factorialMpfr(n, precBits = max(2, ceiling(lgamma(n+1)/log(2))))
```

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*.

Value

a number of (S4) class `mpfr`.

See Also

`factorial` and `gamma` in base R.
`factorialZ` (package `gmp`), to *replace* `factorialMpfr`, see above.
`chooseMpfr()` and `pochMpfr()` (on the same page).

Examples

```
factorialMpfr(200)

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

## Note that---astoundingly--- measurements show only
## *small* efficiency gain of ~ 10% : over using the previous "technique"
system.time(replicate(8, f1e4 <- factorialMpfr(10000)))
system.time(replicate(8, f.1e4 <- factorial(mpfr(10000,
                                           prec=1+lfactorial(10000)/log(2))))))
```

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 e.g., their double precision equivalents.

Usage

```
formatMpfr(x, digits = NULL, trim = FALSE, scientific = NA,
           showNeg0 = TRUE,
           big.mark = "", big.interval = 3L,
           small.mark = "", small.interval = 5L, decimal.mark = ".",
           zero.print = NULL, drop0trailing = FALSE, ...)
## S3 method for class 'mpfr'
formatN(x, drop0trailing = TRUE, ...)
```

Arguments

<code>x</code>	an MPFR number (vector or array).
<code>digits</code>	how many significant digits are to be used for numeric and complex <code>x</code> . The default, <code>NULL</code> , uses enough digits to represent the full precision.
<code>trim</code>	logical; if <code>FALSE</code> , numbers are right-justified to a common width: if <code>TRUE</code> the leading blanks for justification are suppressed.
<code>scientific</code>	either a logical specifying whether MPFR numbers should be encoded in scientific format, or an integer penalty (see <code>options("scipen")</code>). Missing values correspond to the current default penalty.
<code>showNeg0</code>	logical indicating if “ negative ” zeros should be shown with a “-”. The default, <code>TRUE</code> is intentionally different from <code>format(<numeric>)</code> .
<code>big.mark</code> , <code>big.interval</code> , <code>small.mark</code> , <code>small.interval</code> , <code>decimal.mark</code> , <code>zero.print</code> , <code>drop0trailing</code>	used for prettying decimal sequences, these are passed to <code>prettyNum</code> and that help page explains the details.
<code>...</code>	further arguments passed to or from other methods.

Value

a character vector of the same length as `x`.

Author(s)

Martin Maechler

References

The MPFR manual’s description of ‘`mpfr_get_str()`’ which is the C-internal workhorse for the (internal) R function `.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`.

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
```

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)
.mpfr2bigz(x, mod = NA)
.bigq2mpfr(from)
```

Arguments

`x` an R object of class `bigz` or `mpfr` respectively.
`from` an R object of class `bigq`.
`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

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

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

integrateR

One-Dimensional Numerical Integration - in pure R

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

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

Arguments

<code>f</code>	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.
<code>lower, upper</code>	the limits of integration. Currently <i>must</i> be finite.
<code>...</code>	additional arguments to be passed to <code>f</code> .
<code>ord</code>	integer, the order of Romberg integration to be used. If this is NULL, as per default, the order is increased until convergence, see <code>rel.tol</code> and <code>abs.tol</code> .
<code>rel.tol</code>	relative accuracy requested. The default is 1.2e-4.
<code>abs.tol</code>	absolute accuracy requested.
<code>verbose</code>	logical or integer, indicating if and how much information should be printed during computation.

Details

Note that arguments after ... must be matched exactly.

rel.tol cannot be less than $\max(50 \cdot \text{Machine}\$double.eps, 0.5e-28)$ if abs.tol ≤ 0 .

Value

A list of class "integrate" with components

value	the final estimate of the integral.
abs.error	estimate of the modulus of the absolute error.
subdivisions	for Romberg, the number of function evaluations.
message	"OK" or a character string giving the error message.
call	the matched call.

Note

f must accept a vector of inputs and produce a vector of function evaluations at those points. The [Vectorize](#) function may be helpful to convert f to this form.

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 *not* the case for integrateR().

Author(s)

Martin Maechler

References

Bauer, F.L. (1961) Algorithm 60 – Romberg Integration, *Communications of the ACM* **4**(6), p.255.

See Also

R's standard, [integrate](#), is much more adaptive, also allowing infinite integration boundaries, and typically considerably faster for a given accuracy.

We use the same (actually a copy) print S3 method, [print.integrate\(\)](#), as provided by R.

Examples

```
## See more from ?integrate
## this is in the region where integrate() can get problems:
integrateR(dnorm,0,2000)
integrateR(dnorm,0,2000, rel.tol=1e-15)
integrateR(dnorm,0,2000, rel.tol=1e-15, verbose=TRUE)

## 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
I <- integrateR(function(x) (x-2)^4 - 3*(x-3)^2, 0, mpfr(5,128),
  rel.tol = 1e-20, verbose=TRUE)
I ; I$value ## all fine

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

```

is.whole

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[.]` 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

```
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
```

Mnumber-class	<i>Class "Mnumber" of "mpfr" and regular numbers and arrays from them</i>
---------------	---

Description

Class "Mnumber" is a class union 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".

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"))
```

mpfr *Create "mpfr" Numbers (Objects)*

Description

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

Usage

```
mpfr(x, precBits, base = 10, rnd.mode = c("N", "D", "U", "Z"))
Const(name = c("pi", "gamma", "catalan", "log2"), prec = 120L)
```

Arguments

x	a numeric or character vector or array .
precBits, prec	a number, the maximal precision to be used, in <i>bits</i> ; i.e. 53 corresponds to double precision. Must be at least 2.
base	(only when x is character) the base with respect to which x[i] represent numbers; base <i>b</i> must fulfill $2 \leq b \leq 36$.
rnd.mode	a 1-letter string specifying how <i>rounding</i> should happen at C-level conversion to MPFR, see details.
name	a string specifying the mpfrlib - internal constant computation. "gamma" is Euler's gamma (γ), and "catalan" Catalan's constant.

Details

MPFR supports four rounding modes,

GMP_RNDN: round to **nearest** (roundTiesToEven in IEEE 754-2008).

GMP_RNDZ: round toward **zero** (roundTowardZero in IEEE 754-2008).

GMP_RNDU: round toward **plus infinity** ("Up", roundTowardPositive in IEEE 754-2008).

GMP_RNDD: round toward **minus infinity** ("Down", roundTowardNegative in IEEE 754-2008).

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).

Value

an object of (S4) class [mpfr](#), [mpfrMatrix](#), or [mpfrArray](#) which the user should just as a normal numeric vector or array.

Author(s)

Martin Maechler

References

The MPFR team. (2009). *GNU MPFR – The Multiple Precision Floating-Point Reliable Library*; Edition 2.4.2, November 2009; see <http://www.mpfr.org/mpfr-current/#doc> or directly <http://www.mpfr.org/mpfr-current/mpfr.pdf>.

See Also

The class documentation [mpfr](#) contains more details.

Examples

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

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

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

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

## character input :
mpfr("2.718281828459045235360287471352662497757") - exp(mpfr(1, 150))
## ~= -4 * 10^-40

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

mpfr("af21.01020300a0b0c", base=16)
## 68 bit prec. 44833.00393694653820642

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

symnum(sapply(c("N", "D", "U", "Z"),
  function(RND) mpfr(0.2, prec = 5:15, rnd.mode = RND) < 0.2 ))
```

Description

"mpfr" is the class of **M**ultiple **P**recision **F**loatingpoint numbers with **R**eliable arithmetic.

For the high-level user, "mpfr" objects should behave as standard R's `numeric` vectors, just with prespecified (typically high) precision.

Objects from the Class

Objects are typically created by `mpfr(<number>, precBits)`.

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"): ...

beta signature(a = "mpfr", b = "ANY"),

lbeta signature(a = "ANY", b = "mpfrArray"),

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Summary signature(x = "mpfr"): The S4 [Summary](#) group functions, [max](#), [min](#), [range](#), [prod](#), [sum](#), [any](#), and [all](#) are all defined for MPFR numbers.

Math signature(x = "mpfr"): All the S4 [Math](#) group functions are defined, using multiple precision (MPFR) arithmetic, from [getGroupMembers\("Math"\)](#), these are (in alphabetical order):

[abs](#), [sign](#), [sqrt](#), [ceiling](#), [floor](#), [trunc](#), [cummax](#), [cummin](#), [cumprod](#), [cumsum](#), [exp](#), [expm1](#), [log](#), [log10](#), [log2](#), [log1p](#), [cos](#), [cosh](#), [sin](#), [sinh](#), [tan](#), [tanh](#), [acos](#), [acosh](#), [asin](#), [asinh](#), [atan](#), [atanh](#), [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.

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"): ...

all.equal signature(target = "mpfr", current = "mpfr"),

all.equal signature(target = "mpfr", current = "ANY"), and

all.equal signature(target = "ANY", current = "mpfr"): methods for numerical (approximate) equality, [all.equal](#) of multiple precision numbers. Note that the default tolerance (argument) is taken to correspond to the (smaller of the two) precisions when both main arguments are of class "mpfr", and hence can be considerably less than double precision machine epsilon [.Machine\\$double.eps](#).

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

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

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

unique signature(x = "mpfr"): and

duplicated signature(x = "mpfr"): just work as with numbers.

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

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; `is.whole()` etc.

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

Examples

```
## 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

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

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

mpfr.min.exp2 <- - (2^30 + 1)
two <- mpfr(2, 55)
stopifnot(two ^ mpfr.min.exp2 == 0)
## whereas
two ^ (mpfr.min.exp2 * (1 - 1e-15))
## 2.38256490488795107e-323228497 ["typically"]
```

mpfr-special-functions

*Special Mathematical Functions (MPFR)***Description**

Special Mathematical Functions, supported by the MPFR Library.

Usage

```
zeta(x)
Ei(x)
Li2(x)

erf(x)
erfc(x)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
```

Arguments

`x`, `q`, `mean`, `sd` a [numeric](#) or [mpfr](#) vector.
`lower.tail`, `log.p`
 logical, see [pnorm](#) (or other `p*`() functions).

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 `pnorm`, $\text{erf}(x) = 2 \cdot \text{pnorm}(\sqrt{2} \cdot x)$ and $\text{erfc}(x) = 2 \cdot \text{pnorm}(\sqrt{2} \cdot x, \text{lower}=\text{FALSE})$.

Value

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

See Also

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

Examples

```
curve(Ei, 0, 5, n=2001)

if(mpfrVersion() >= "2.4.0") { ## Li2() is not available in older MPFR versions
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. In some case they may come handy.

Usage

```
getPrec(x, base = 10, doNumeric = TRUE, is.mpfr = NA)
getD(x)
mpfr_default_prec(prec)
## S3 method for class 'mpfrArray'
print(x, digits = NULL, drop0trailing = FALSE,
      right = TRUE, ...)
## S3 method for class 'mpfr'
print(x, digits = NULL, drop0trailing = TRUE,
      right = TRUE, ...)
toNum(from)
mpfr2array(x, dim, dimnames = NULL, check = FALSE)
```

Arguments

<code>x, from</code>	an R object of class "mpfr", or "mpfrArray", respectively.
<code>base</code>	(only when <code>x</code> is <code>character</code>) the base with respect to which <code>x[i]</code> represent numbers; base b must fulfill $2 \leq b \leq 36$.
<code>doNumeric</code>	logical indicating <code>integer</code> or <code>double</code> typed <code>x</code> should be accepted and a default precision be returned. Should typically be kept at default TRUE.
<code>is.mpfr</code>	logical indicating if <code>class(x)</code> is already known to be "mpfr"; typically should be kept at default, NA.
<code>prec</code>	a positive integer, or missing.
<code>drop0trailing</code>	logical indicating if trailing "0"s should be omitted.
<code>right</code>	logical indicating <code>print()</code> ing should right justify the strings; see <code>print.default()</code> to which it is passed.
<code>digits, ...</code>	further arguments to print methods.
<code>dim, dimnames</code>	for "mpfrArray" construction.
<code>check</code>	logical indicating if the <code>mpfrArray</code> construction should happen with internal safety check. Previously, the implicit default used to be true.

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.

Value

`getPrec(x)` returns a `integer` vector of the same length as `x`. If you need to *change* the precision of `x`, i.e., need something like "setPrec", use `roundMpfr()`.

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

`mpfr_default_prec()` returns the current MPFR default precision, an `integer`. This is currently not made use of, in all of package **Rmpfr**, where functions have their own default precision where needed.

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

`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"`.

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

See Also

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

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)

## Printing of "MPFR" matrices is less nice than R's usual matrix printing:
m <- outer(c(1, 3.14, -1024.5678), c(1, 1e-3, 10,100))
m[3,3] <- round(m[3,3])
m
mpfr(m, 50)

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

```

mpfr.utils

MPFR Number Utilities

Description

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

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

`mpfr.is.0(.)` uses the MPFR library in the documented way to check if (a vector of) MPFR numbers are zero.

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

`hypot(x,y)` computes the hypotenuse length z in a rectangular triangle with “leg” side lengths x and y , i.e.,

$$z = \text{hypot}(x, y) = \sqrt{x^2 + y^2},$$

in a numerically stable way.

Usage

```

mpfrVersion()
mpfr.is.0(x)
mpfr.is.integer(x)
## S3 method for class 'mpfr'
c(...)
## S3 method for class 'mpfr'
diff(x, lag = 1L, differences = 1L, ...)
## S3 method for class 'mpfr'

```

```
str(object, nest.lev, ...)
```

```
hypot(x,y)
```

Arguments

`x,y, object` an object of class `mpfr`.
`...` further `mpfr` class objects or simple numbers (`numeric` vectors) which are coerced to `mpfr` with default precision of 128 bits.
`lag, differences` for `diff()`: exact same meaning as in `diff()`'s default method, `diff.default`.
`nest.lev` for `str()`, typically only used when called by a higher level `str()`.

Value

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

Similarly, `mpfr.is.integer` returns a logical vector of length `length(x)`.

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

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

Methods

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)`.

See Also

`erf` for special mathematical functions on MPFR; The class description `mpfr` mentioning the generic arithmetic and mathematical functions for which `"mpfr"` methods are available.

Examples

```
mpfrVersion()
```

```
(x <- c(Const("pi", 64), mpfr(-2:2, 64)))
mpfr.is.0(x) # one of them is
x[mpfr.is.0(x)] # but it may not have been obvious..
str(x)
```

```
xy <- expand.grid(x = -2:2, y = -2:2) ; x <- xy[,"x"] ; y <- xy[,"y"]
a2. <- atan2(y,x)
```

```
stopifnot(all.equal(a2., atan2(as(y,"mpfr"), x)),
          mpfr.is.integer(mpfr(2, 500) ^ (1:200)),
          all.equal(diff(x), diff(as.numeric(x))),
          TRUE)
```

mpfrArray *Construct "mpfrArray" almost as by 'array()'*

Description

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

Usage

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

Arguments

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

Value

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

See Also

`mpfr`, `array`.

Examples

```
## 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

## Testing the apply() method :
apply(mat, 2, range)
apply(A2, 1:2, range)
```

```

apply(A2, 2:3, max)
apply(A2, 2, fivenum)
stopifnot(as(apply(A2, 2, range), "matrix") ==
          apply(as(A2,"array"), 2, range))

```

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 can be created by calls of the form `new("mpfrMatrix", ...)` or `new("mpfrArray", ...)`, or also by `dim(x) <- dd` or `t(x)` where `x` is a an `mpfr` "number vector".

A (slightly more) 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[j]`.

Extends

Class "mpfrMatrix" extends "mpfrArray", directly.

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

Methods

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

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

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

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

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

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

atan2 signature(y = "ANY", x = "mpfrArray"): ...

```

atan2 signature(y = "mpfrArray", x = "mpfrArray"): ...
atan2 signature(y = "mpfrArray", x = "ANY"): ...
[<- signature(x = "mpfrArray", i = "ANY", j = "ANY", value = "ANY"): ...
[ signature(x = "mpfrArray", i = "ANY", j = "ANY", drop = "ANY"): ...
[ signature(x = "mpfrArray", i = "ANY", j = "missing", drop = "missing"):
  "mpfrArray"s can be subset ("indexed") as regular R arrays.
%% signature(x = "mpfr", y = "mpfrMatrix"): Compute the matrix/vector product  $xy$ 
  when the dimensions (dim) of  $x$  and  $y$  match. If  $x$  is not a matrix, it is treated as a 1-row
  or 1-column matrix (aka "row vector" or "column vector") depending on which one makes
  sense, see the documentation of the base function %*%.
%% signature(x = "mpfr", y = "Mnumber"): method definition for cases with one mpfr
  and any "number-like" argument are to use MPFR arithmetic as well.
%% signature(x = "mpfrMatrix", y = "mpfrMatrix"),
%% signature(x = "mpfrMatrix", y = "mpfr"), etc. Further method definitions with
  identical semantic.
crossprod signature(x = "mpfr", y = "missing"): Computes  $x'x$ , i.e.,  $t(x) \%*\% x$ , typically
  more efficiently.
crossprod signature(x = "mpfr", y = "mpfrMatrix"): Computes  $x'y$ , i.e.,  $t(x) \%*\% y$ ,
  typically more efficiently.
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"): ...

```

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"
```

pmax

Parallel Maxima and Minima

Description

Returns the parallel maxima and minima of the input values.

The functions `pmin` and `pmax` have been made S4 generics, and this page documents the "... method for class "Mnumber"", i.e., for arguments that are numeric or from [class "mpfr"](#).

Usage

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

Arguments

... numeric or arbitrary precision numbers (class [mpfr](#)).

na.rm a logical indicating whether missing values should be removed.

Details

See [pmax](#), the documentation of the base functions, i.e., default methods.

Value

vector-like, of length the longest of the input vectors; typically of class [mpfr](#), for the methods here.

Methods

... = "ANY" the default method, really just `base::pmin` or `base::pmax`, respectively.

... = "Mnumber" the method for [mpfr](#) arguments, mixed with numbers; designed to follow the same semantic as the default method.

See Also

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

Examples

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

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

```
roundMpfr(x, precBits)
```

Arguments

`x` an [mpfr](#) number (vector)
`precBits` integer specifying the desired precision in bits.

Value

an [mpfr](#) number as `x` but with the new 'precBits' precision

See Also

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

Examples

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

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 <i>"mpfr"</i>) of the sequence.
by	number (numeric or <i>"mpfr"</i>): 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.
```

sumBinomMpfr

(Alternating) Binomial Sums via Rmpfr

Description

Compute (alternating) binomial sums via high-precision arithmetic. 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 $50 - -70$.

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

$$sB(f, n) = (-1)^n \Delta^n f,$$

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 $x = 0$).

Usage

```
sumBinomMpfr(n, f, n0 = 0, alternating = TRUE, precBits = 256)
```

Arguments

n	upper summation index (integer).
f	function to be evaluated at k for k in $n0:n$.
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 .

Details

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

Value

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

$$s = \sum_{k=n_0}^n (-1)^k \binom{n}{k} \cdot f(k),$$

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

Wikipedia (2012) The Nörlund-Rice integral, http://en.wikipedia.org/wiki/Rice_integral

Flajolet, P. and Sedgewick, R. (1995) Mellin Transforms and Asymptotics: Finite Differences and Rice's Integrals, *Theoretical Computer Science* **144**, 101–124.

See Also

[chooseMpfr](#), [chooseZ](#) from package `gmp`.

Examples

```
## "naive" R implementation:
sumBinom <- function(n, f, n0=0, ...) {
  k <- n0:n
  sum( choose(n, k) * (-1)^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)) ## takes ~2 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")
```

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,
         epsC = NULL)
```

Arguments

<code>f</code>	the function for which the root is sought.
<code>interval</code>	a vector containing the end-points of the interval to be searched for the root.
<code>...</code>	additional named or unnamed arguments to be passed to <code>f</code>
<code>lower, upper</code>	the lower and upper end points of the interval to be searched.
<code>f.lower, f.upper</code>	the same as <code>f(upper)</code> and <code>f(lower)</code> , respectively. Passing these values from the caller where they are often known is more economical as soon as <code>f()</code> contains non-trivial computations.
<code>verbose</code>	logical (or integer) indicating if (and how much) verbose output should be produced during the iterations.
<code>tol</code>	the desired accuracy (convergence tolerance).
<code>maxiter</code>	the maximum number of iterations.
<code>epsC</code>	positive number or NULL in which case a smart default is sought. This should specify the “achievable machine precision” <i>for</i> the given numbers and their arithmetic. The default will set this to <code>.Machine\$double.eps</code> for double precision numbers, and will basically use $2^{-\min(\text{getPrec}(f.lower), \text{getPrec}(f.upper))}$ when that works (as, e.g., for <code>mpfr</code> -numbers) otherwise. This is factually a lower bound for the achievable lower bound, and hence, setting <code>tol</code> smaller than <code>epsC</code> is typically non-sensical and produces a warning.

Details

Note that arguments after `...` must be matched exactly.

Either `interval` or both `lower` and `upper` must be specified: the upper endpoint must be strictly larger than the lower endpoint. The function values at the endpoints must be of opposite signs (or zero).

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

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

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

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

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

Value

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

Source

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

References

Brent, R. (1973), see `uniroot`.

See Also

`polyroot` for all complex roots of a polynomial; `optimize`, `nlm`.

Examples

```
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,
            tol = 0.0001), digits.d = 10)
str(unirootR(function(x) x*(x^2-1) + .5, lower = -2, upper = 2,
            tol = 1e-10 ), digits.d = 10)

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

## Find the smallest value x for which exp(x) > 0 (numerically):
r <- unirootR(function(x) 1e80*exp(x)-1e-300, c(-1000,0), tol = 1e-15)
str(r, digits.d = 15) ##> around -745, depending on the platform.

exp(r$root)      # = 0, but not for r$root * 0.999...
minexp <- r$root * (1 - 10*.Machine$double.eps)
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 } :
(ep.M <- mpfr(2, 55) ^ - ((2^30 + 1) * (1 - 1e-15)))
r <- unirootR(function(x) 1e99* exp(x) - ep.M, mpfr(c(-1e20, 0), 200))
r # 97 iterations; f.root is very similar to ep.M

```

Index

- *Topic **arith**
 - Bernoulli, [4](#)
 - chooseMpfr, [7](#)
 - factorialMpfr, [8](#)
 - gmp-conversions, [11](#)
 - mpfr.utils, [24](#)
 - pmax, [29](#)
 - roundMpfr, [30](#)
 - sumBinomMpfr, [32](#)
- *Topic **array**
 - mpfrArray, [26](#)
- *Topic **character**
 - formatMpfr, [9](#)
- *Topic **classes**
 - array_or_vector-class, [2](#)
 - atomicVector-class, [3](#)
 - Mnumber-class, [15](#)
 - mpfr, [16](#)
 - mpfr-class, [17](#)
 - mpfrMatrix, [27](#)
- *Topic **manip**
 - seqMpfr, [31](#)
- *Topic **math**
 - Bessel_mpfr, [5](#)
 - integrateR, [12](#)
 - is.whole, [14](#)
 - mpfr-special-functions, [21](#)
- *Topic **methods**
 - bind-methods, [6](#)
 - pmax, [29](#)
- *Topic **optimize**
 - unirootR, [34](#)
- *Topic **print**
 - formatMpfr, [9](#)
- *Topic **univar**
 - pmax, [29](#)
- *Topic **utilities**
 - integrateR, [12](#)
 - mpfr-utils, [22](#)
 - .Machine, [19](#), [34](#)
 - .bigq2mpfr (gmp-conversions), [11](#)
 - .bigz2mpfr (gmp-conversions), [11](#)
 - .mpfr2bigz (gmp-conversions), [11](#)
 - [,mpfr,ANY,missing,missing-method (mpfr-class), [17](#)
 - [,mpfrArray,ANY,ANY,ANY-method (mpfrMatrix), [27](#)
 - [,mpfrArray,ANY,missing,missing-method (mpfrMatrix), [27](#)
 - [<-,mpfr,ANY,missing,ANY-method (mpfr-class), [17](#)
 - [<-,mpfr,ANY,missing,mpfr-method (mpfr-class), [17](#)
 - [<-,mpfr,missing,missing,ANY-method (mpfr-class), [17](#)
 - [<-,mpfrArray,ANY,ANY,ANY-method (mpfrMatrix), [27](#)
 - [<-,mpfrArray,ANY,ANY,mpfr-method (mpfrMatrix), [27](#)
 - [<-,mpfrArray,ANY,missing,ANY-method (mpfrMatrix), [27](#)
 - [<-,mpfrArray,ANY,missing,mpfr-method (mpfrMatrix), [27](#)
 - [<-,mpfrArray,missing,ANY,ANY-method (mpfrMatrix), [27](#)
 - [<-,mpfrArray,missing,ANY,mpfr-method (mpfrMatrix), [27](#)
 - [<-,mpfrArray,missing,missing,ANY-method (mpfrMatrix), [27](#)
 - [<-,mpfrArray,missing,missing,mpfr-method (mpfrMatrix), [27](#)
 - [[,mpfr-method (mpfr-class), [17](#)
 - %%%,Mnumber,mpfr-method (mpfrMatrix), [27](#)
 - %%%,array_or_vector,mpfr-method (mpfr-class), [17](#)
 - %%%,mpfr,Mnumber-method (mpfrMatrix), [27](#)
 - %%%,mpfr,array_or_vector-method (mpfr-class), [17](#)

- %%, mpfr, mpfr-method (mpfrMatrix), 27
- %%, mpfr, mpfrMatrix-method (mpfrMatrix), 27
- %%, mpfrMatrix, mpfr-method (mpfrMatrix), 27
- %%, mpfrMatrix, mpfrMatrix-method (mpfrMatrix), 27
- %%, 28
- abs, 19
- abs, mpfr-method (mpfr-class), 17
- acos, 19
- acosh, 19
- Ai (Bessel_mpfr), 5
- all, 19
- all.equal, 19
- all.equal, ANY, mpfr-method (mpfr-class), 17
- all.equal, mpfr, ANY-method (mpfr-class), 17
- all.equal, mpfr, mpfr-method (mpfr-class), 17
- any, 19
- aperm, 29
- aperm, mpfrArray-method (mpfrMatrix), 27
- apply, mpfrArray-method (mpfrMatrix), 27
- Arith, array, mpfr-method (mpfr-class), 17
- Arith, integer, mpfr-method (mpfr-class), 17
- Arith, mpfr, array-method (mpfr-class), 17
- Arith, mpfr, integer-method (mpfr-class), 17
- Arith, mpfr, missing-method (mpfr-class), 17
- Arith, mpfr, mpfr-method (mpfr-class), 17
- Arith, mpfr, mpfrArray-method (mpfrMatrix), 27
- Arith, mpfr, numeric-method (mpfr-class), 17
- Arith, mpfrArray, mpfr-method (mpfrMatrix), 27
- Arith, mpfrArray, mpfrArray-method (mpfrMatrix), 27
- Arith, mpfrArray, numeric-method (mpfrMatrix), 27
- Arith, numeric, mpfr-method (mpfr-class), 17
- Arith, numeric, mpfrArray-method (mpfrMatrix), 27
- array, 16, 19, 23, 26–29
- array_or_vector, 15
- array_or_vector-class, 2
- as, 20
- as.bigq, 12
- as.bigz, 11, 12
- as.integer, mpfr-method (mpfr-class), 17
- as.numeric, mpfr-method (mpfr-class), 17
- as.vector, mpfrArray, missing-method (mpfrMatrix), 27
- as.vector, mpfrArray-method (mpfr-class), 17
- asin, 19
- asinh, 19
- atan, 19, 25
- atan2, ANY, mpfr-method (mpfr.utils), 24
- atan2, ANY, mpfrArray-method (mpfr.utils), 24
- atan2, mpfr, ANY-method (mpfr.utils), 24
- atan2, mpfr, mpfr-method (mpfr.utils), 24
- atan2, mpfrArray, ANY-method (mpfr.utils), 24
- atan2, mpfrArray, mpfrArray-method (mpfr.utils), 24
- atanh, 19
- atomicVector-class, 3
- base::pmin, 30
- Bernoulli, 4, 20
- Bessel_mpfr, 5
- besselJ, 5
- besselY, 5
- beta, 18
- beta, ANY, mpfr-method (mpfr-class), 17
- beta, ANY, mpfrArray-method (mpfr-class), 17
- beta, mpfr, ANY-method (mpfr-class), 17
- beta, mpfr, mpfr-method (mpfr-class), 17
- beta, mpfrArray, ANY-method (mpfr-class), 17
- beta, mpfrArray, mpfrArray-method (mpfr-class), 17
- bigq, 11
- bigz, 11
- bind-methods, 6
- c, 24
- c.mpfr (mpfr.utils), 24
- cbind, 6

- cbind (bind-methods), 6
- cbind, ANY-method (bind-methods), 6
- cbind, Mnumber-method (bind-methods), 6
- cbind-methods (bind-methods), 6
- cbind2, 6
- ceiling, 19
- character, 16, 19, 20, 23, 27
- choose, 7, 8
- chooseMpfr, 7, 33
- chooseZ, 7, 8, 33
- class, 3, 6, 14, 23, 29
- coerce, array, mpfr-method (mpfr-class), 17
- coerce, bigq, mpfr-method (gmp-conversions), 11
- coerce, bigz, mpfr-method (gmp-conversions), 11
- coerce, character, mpfr-method (mpfr-class), 17
- coerce, integer, mpfr-method (mpfr-class), 17
- coerce, logical, mpfr-method (mpfr-class), 17
- coerce, mpfr, character-method (mpfr-class), 17
- coerce, mpfr, integer-method (mpfr-class), 17
- coerce, mpfr, mpfr1-method (mpfr-class), 17
- coerce, mpfr, numeric-method (mpfr-class), 17
- coerce, mpfr1, mpfr-method (mpfr-class), 17
- coerce, mpfr1, numeric-method (mpfr-class), 17
- coerce, mpfrArray, array-method (mpfrMatrix), 27
- coerce, mpfrArray, matrix-method (mpfrMatrix), 27
- coerce, mpfrArray, vector-method (mpfrMatrix), 27
- coerce, mpfrMatrix, matrix-method (mpfrMatrix), 27
- coerce, numeric, mpfr-method (mpfr-class), 17
- coerce, numeric, mpfr1-method (mpfr-class), 17
- coerce, raw, mpfr-method (mpfr-class), 17
- coerce<-, mpfrArray, vector-method (mpfrMatrix), 27
- colMeans, mpfrArray-method (mpfrMatrix), 27
- colSums, mpfrArray-method (mpfrMatrix), 27
- Compare, array, mpfr-method (mpfr-class), 17
- Compare, integer, mpfr-method (mpfr-class), 17
- Compare, mpfr, array-method (mpfr-class), 17
- Compare, mpfr, integer-method (mpfr-class), 17
- Compare, mpfr, mpfr-method (mpfr-class), 17
- Compare, mpfr, mpfrArray-method (mpfrMatrix), 27
- Compare, mpfr, numeric-method (mpfr-class), 17
- Compare, mpfrArray, mpfr-method (mpfrMatrix), 27
- Compare, mpfrArray, numeric-method (mpfrMatrix), 27
- Compare, numeric, mpfr-method (mpfr-class), 17
- Compare, numeric, mpfrArray-method (mpfrMatrix), 27
- complex, 3
- Const (mpfr), 16
- cos, 19
- cosh, 19
- crossprod, array_or_vector, mpfr-method (mpfr-class), 17
- crossprod, Mnumber, mpfr-method (mpfrMatrix), 27
- crossprod, mpfr, array_or_vector-method (mpfr-class), 17
- crossprod, mpfr, missing-method (mpfrMatrix), 27
- crossprod, mpfr, Mnumber-method (mpfrMatrix), 27
- crossprod, mpfr, mpfr-method (mpfrMatrix), 27
- crossprod, mpfr, mpfrMatrix-method (mpfrMatrix), 27
- crossprod, mpfrMatrix, mpfr-method (mpfrMatrix), 27

- crossprod, mpfrMatrix, mpfrMatrix-method (mpfrMatrix), 27
- cummax, 19
- cummin, 19
- cumprod, 19
- cumsum, 19
- diff, 25
- diff.default, 25
- diff.mpfr (mpfr.utils), 24
- digamma, 19
- dim, 18, 28
- dim, mpfrArray-method (mpfrMatrix), 27
- dim<-, mpfr-method (mpfr-class), 17
- dimnames, mpfrArray-method (mpfrMatrix), 27
- dimnames<-, mpfrArray-method (mpfrMatrix), 27
- dotsMethods, 6
- double, 23
- Ei (mpfr-special-functions), 21
- erf, 25
- erf (mpfr-special-functions), 21
- erfc (mpfr-special-functions), 21
- exp, 19
- expm1, 19
- factorial, 7, 9
- factorial, mpfr-method (mpfr-class), 17
- factorialMpfr, 8, 8, 19
- factorialZ, 8, 9
- floor, 19
- format, 9, 10, 23
- format, mpfr-method (mpfr-class), 17
- formatMpfr, 9, 19
- formatN, 10
- formatN.mpfr (formatMpfr), 9
- function, 32
- gamma, 7, 9, 18, 19
- getD (mpfr-utils), 22
- getGroupMembers, 19
- getPrec, 30
- getPrec (mpfr-utils), 22
- gmp-conversions, 11
- hypot (mpfr.utils), 24
- integer, 3, 20, 23
- integrate, 13
- integrateR, 12
- is.atomic, 3
- is.finite, mpfr-method (mpfr-class), 17
- is.infinite, mpfr-method (mpfr-class), 17
- is.integer, 14
- is.na, mpfr-method (mpfr-class), 17
- is.nan, mpfr-method (mpfr-class), 17
- is.whole, 14, 14, 20, 24
- j0 (Bessel_mpfr), 5
- j1 (Bessel_mpfr), 5
- jn, 20
- jn (Bessel_mpfr), 5
- lbeta, ANY, mpfr-method (mpfr-class), 17
- lbeta, ANY, mpfrArray-method (mpfr-class), 17
- lbeta, mpfr, ANY-method (mpfr-class), 17
- lbeta, mpfr, mpfr-method (mpfr-class), 17
- lbeta, mpfrArray, ANY-method (mpfr-class), 17
- lbeta, mpfrArray, mpfrArray-method (mpfr-class), 17
- lgamma, 18, 19
- Li2 (mpfr-special-functions), 21
- list, 18, 27
- log, 19
- log, mpfr-method (mpfr-class), 17
- log10, 19
- log1p, 19
- log2, 19
- Logic, mpfr, mpfr-method (mpfr-class), 17
- Logic, mpfr, numeric-method (mpfr-class), 17
- Logic, numeric, mpfr-method (mpfr-class), 17
- logical, 20
- Math, 19
- Math, mpfr-method (mpfr-class), 17
- Math2, mpfr-method (mpfr-class), 17
- matrix, 23, 27
- max, 19
- mean, mpfr-method (mpfr-class), 17
- Methods, 6
- min, 19, 30
- Mnumber, 6
- Mnumber-class, 15

- mpfr, 4–7, 9–12, 14, 15, 16, 16–35
- mpfr-class, 17
- mpfr-special-functions, 21
- mpfr-utils, 22
- mpfr.is.0 (mpfr.utils), 24
- mpfr.is.integer (mpfr.utils), 24
- mpfr.utils, 24
- mpfr1, 27
- mpfr1-class (mpfr-class), 17
- mpfr2array, 27
- mpfr2array (mpfr-utils), 22
- mpfr_default_prec (mpfr-utils), 22
- mpfrArray, 16, 18, 23, 26, 26, 29
- mpfrArray-class (mpfrMatrix), 27
- mpfrMatrix, 6, 15, 16, 20, 23, 26, 27
- mpfrMatrix-class (mpfrMatrix), 27
- mpfrVersion (mpfr.utils), 24

- nlm, 35
- NULL, 27
- numeric, 3, 5, 16, 18, 20, 21, 25
- numeric_version, 25

- Ops, 35
- Ops, ANY, mpfr-method (mpfr-class), 17
- Ops, array, mpfr-method (mpfr-class), 17
- Ops, bigq, mpfr-method (mpfr-class), 17
- Ops, bigz, mpfr-method (mpfr-class), 17
- Ops, mpfr, ANY-method (mpfr-class), 17
- Ops, mpfr, array-method (mpfr-class), 17
- Ops, mpfr, bigq-method (mpfr-class), 17
- Ops, mpfr, bigz-method (mpfr-class), 17
- Ops, mpfr, vector-method (mpfr-class), 17
- Ops, vector, mpfr-method (mpfr-class), 17
- optimize, 35
- options, 10
- order, 20

- pmax, 29, 30
- pmax, ANY-method (pmax), 29
- pmax, Mnumber-method (pmax), 29
- pmax-methods (pmax), 29
- pmin, 30
- pmin (pmax), 29
- pmin, ANY-method (pmax), 29
- pmin, Mnumber-method (pmax), 29
- pmin-methods (pmax), 29
- pnorm, 21, 22
- pnorm (mpfr-special-functions), 21

- pochMpfr, 9, 20
- pochMpfr (chooseMpfr), 7
- polyroot, 35
- prettyNum, 10
- print.default, 23
- print.integrate, 13
- print.integrate (integrateR), 12
- print.mpfr (mpfr-utils), 22
- print.mpfr1 (mpfr-class), 17
- print.mpfrArray (mpfr-utils), 22
- prod, 19

- quantile, 20

- range, 19, 30
- rank, 20
- raw, 20
- rbind, 6
- rbind (bind-methods), 6
- rbind, ANY-method (bind-methods), 6
- rbind, Mnumber-method (bind-methods), 6
- rbind-methods (bind-methods), 6
- round, 19, 30
- roundMpfr, 20, 23, 30
- rowMeans, mpfrArray-method (mpfrMatrix), 27
- rowSums, mpfrArray-method (mpfrMatrix), 27

- seq, 31
- seqMpfr, 31
- setPrec (roundMpfr), 30
- show, integrate-method (integrateR), 12
- show, mpfr-method (mpfr-class), 17
- show, mpfr1-method (mpfr-class), 17
- show, mpfrArray-method (mpfrMatrix), 27
- sign, 18, 19
- sign, mpfr-method (mpfr-class), 17
- sign, mpfrArray-method (mpfrMatrix), 27
- signif, 19
- sin, 19
- sinh, 19
- sort, 20
- sqrt, 19
- str, 25
- str.mpfr (mpfr.utils), 24
- sum, 19
- sumBinomMpfr, 8, 32
- Summary, 19

Summary, mpfr-method (mpfr-class), 17

t, mpfr-method (mpfr-class), 17

t, mpfrMatrix-method (mpfrMatrix), 27

tan, 19

tanh, 19

tcrossprod, array_or_vector, mpfr-method (mpfr-class), 17

tcrossprod, Mnumber, mpfr-method (mpfrMatrix), 27

tcrossprod, mpfr, array_or_vector-method (mpfr-class), 17

tcrossprod, mpfr, missing-method (mpfrMatrix), 27

tcrossprod, mpfr, Mnumber-method (mpfrMatrix), 27

tcrossprod, mpfr, mpfr-method (mpfrMatrix), 27

tcrossprod, mpfr, mpfrMatrix-method (mpfrMatrix), 27

tcrossprod, mpfrMatrix, mpfr-method (mpfrMatrix), 27

tcrossprod, mpfrMatrix, mpfrMatrix-method (mpfrMatrix), 27

toNum (mpfr-utils), 22

trigamma, 19

trunc, 19

unique, mpfr, missing-method (mpfr-class), 17

uniroot, 34, 35

unirootR, 34

vector, 27

Vectorize, 13

y0 (Bessel_mpfr), 5

y1 (Bessel_mpfr), 5

yn (Bessel_mpfr), 5

zeta, 4, 20

zeta (mpfr-special-functions), 21