Package 'DPQ'

May 7, 2026

Title Density, Probability, Quantile ('DPQ') Computations

Version 0.6-1

Date 2025-10-13

VersionNote Last CRAN: 0.6-0 on 2025-07-08; 0.5-9 on 2024-08-23; 0.5-8 on 2023-11-30

Description Computations for approximations and alternatives for the 'DPQ' (Density (pdf), Probability (cdf) and Quantile) functions for probability distributions in R.
Primary focus is on (central and non-central) beta, gamma and related distributions such as the chi-squared, F, and t.
--
For several distribution functions, provide functions implementing formulas from Johnson, Kotz, and Kemp (1992) <doi:10.1002/bimj.4710360207> and Johnson, Kotz, and Balakrishnan (1995) for discrete or continuous distributions respectively.
This is for the use of researchers in these numerical approximation implementations, notably for my own use in order to improve standard R pbeta(), qgamma(), ..., etc: {'``dpq'"-functions}.

Depends R (>= 4.1.0)

Imports stats, graphics, methods, utils, sfsmisc (>= 1.1-14)

Suggests Rmpfr, DPQmpfr (>= 0.3-3), gmp, MASS, mgcv, scatterplot3d, interp, cobs

SuggestsNote MASS::fractions() in ex | mgcv, scatt.., .., cobs: some tests/

License GPL (>= 2) | file LICENSE

Encoding UTF-8

URL https://specfun.r-forge.r-project.org/,
https://r-forge.r-project.org/R/?group_id=611,
https://r-forge.r-project.org/scm/viewvc.php/pkg/DPQ/?root=specfun,
svn://svn.r-forge.r-project.org/svnroot/specfun/pkg/DPQ

BugReports https://r-forge.r-project.org/tracker/?atid=2462&group_id=611

NeedsCompilation yes

Author Martin Maechler [aut, cre] (ORCID:
<https://orcid.org/0000-0002-8685-9910>),
Morten Welinder [ctb] (pgamma C code, see PR#7307, Jan. 2005; further pdhyper()),
Wolfgang Viechtbauer [ctb] (dtWV(), 2002),
Ross Ihaka [ctb] (src/qchisq_appr.c),
Marius Hofert [ctb] (lsum(), lssum()),
R-core [ctb] (src/{dpq.h, algdiv.c, pnchisq.c, bd0.c}),
R Foundation [cph] (src/qchisq-appr.c)

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

Repository CRAN

Date/Publication 2025-10-13 16:20:12 UTC

Contents

DPQ-package *Density, Probability, Quantile ('DPQ') Computations*

Description

Computations for approximations and alternatives for the 'DPQ' (Density (pdf), Probability (cdf) and Quantile) functions for probability distributions in R. Primary focus is on (central and non-central) beta, gamma and related distributions such as the chi-squared, F, and t. – For several distribution functions, provide functions implementing formulas from Johnson, Kotz, and Kemp (1992) <doi:10.1002/bimj.4710360207> and Johnson, Kotz, and Balakrishnan (1995) for discrete or continuous distributions respectively. This is for the use of researchers in these numerical approximation implementations, notably for my own use in order to improve standard R pbeta(), qgamma(), ..., etc: {'"dpq"'-functions}.

Details

The DESCRIPTION file:

Package:	DPQ
Title:	Density, Probability, Quantile ('DPQ') Computations
Version:	0.6-1
Date:	2025-10-13
VersionNote:	Last CRAN: 0.6-0 on 2025-07-08; 0.5-9 on 2024-08-23; 0.5-8 on 2023-11-30
Authors@R:	c(person("Martin","Maechler", role=c("aut","cre"), email="maechler@stat.math.ethz.ch", comment = c(ORC
Description:	Computations for approximations and alternatives for the 'DPQ' (Density (pdf), Probability (cdf) and Quanti
Depends:	R (>= 4.1.0)
Imports:	stats, graphics, methods, utils, sfsmisc (>= 1.1-14)
Suggests:	Rmpfr, DPQmpfr (>= 0.3-3), gmp, MASS, mgcv, scatterplot3d, interp, cobs
SuggestsNote:	MASS::fractions() in ex | mgcv, scatt.., .., cobs: some tests/
License:	GPL (>= 2) | file LICENSE
Encoding:	UTF-8
URL:	https://specfun.r-forge.r-project.org/, https://r-forge.r-project.org/R/?group_id=611, https://r-forge.r-project.or
BugReports:	https://r-forge.r-project.org/tracker/?atid=2462&group_id=611
Author:	Martin Maechler [aut, cre] (ORCID: <https://orcid.org/0000-0002-8685-9910>), Morten Welinder [ctb] (pga
Maintainer:	Martin Maechler <maechler@stat.math.ethz.ch>

Index of help topics:

```
.D_0                    Distribution Utilities "dpq"
Bern                    Bernoulli Numbers
DPQ-package             Density, Probability, Quantile ('DPQ')
                        Computations
```

Ixpq	Normalized Incomplete Beta Function "Like" 'pbeta()'
M_LN2	Numerical Utilities - Functions, Constants
algdiv	Compute log(gamma(b)/gamma(a+b)) when b >= 8
b_chi	Compute E[chi_nu]/sqrt(nu) useful for t- and chi-Distributions
bd0	Binomial Deviance - Auxiliary Functions for 'dgamma()' Etc
bpser	'pbeta()' 'bpser' series computation
chebyshevPoly	Chebyshev Polynomial Evaluation
dbinom_raw	R's C Mathlib (Rmath) dbinom_raw() Binomial Probability pure R Function
dgamma.R	Gamma Density Function Alternatives
dhyperBinMolenaar	HyperGeometric (Point) Probabilities via Molenaar's Binomial Approximation
dltgammaInc	TOMS 1006 - Fast and Accurate Generalized Incomplete Gamma Function
dnbinomR	Pure R Versions of R's C (Mathlib) dnbinom() Negative Binomial Probabilities
dnchisqR	Approximations of the (Noncentral) Chi-Squared Density
dntJKBf1	Non-central t-Distribution Density - Algorithms and Approximations
dpsifn	Psi Gamma Functions Workhorse from R's API
dtWV	Asymptotic Noncentral t Distribution Density by Viechtbauer
expm1x	Accurate exp(x) - 1 - x (for smallish |x|)
format01prec	Format Numbers in [0,1] with "Precise" Result
frexp	Base-2 Representation and Multiplication of Numbers
gam1d	Compute 1/Gamma(x+1) - 1 Accurately
gamln1	Compute log(Gamma(x+1)) Accurately in [-0.2, 1.25]
gammaVer	Gamma Function Versions
hyper2binomP	Transform Hypergeometric Distribution Parameters to Binomial Probability
lbetaM	(Log) Beta and Ratio of Gammas Approximations
lfastchoose	R versions of Simple Formulas for Logarithmic Binomial Coefficients
lgamma1p	Accurate 'log(gamma(a+1))'
lgammaAsymp	Asymptotic Log Gamma Function
lgammaP11	Log Gamma(p) for Positive 'p' by Pugh's Method (11 Terms)
log1mexp	Compute log(1 - exp(-a)) and log(1 + exp(x)) Numerically Optimally
log1pmx	Accurate 'log(1+x) - x' Computation
logcf	Continued Fraction Approximation of Log-Related Power Series

logspace.add	Logspace Arithmetix - Addition and Subtraction
lssum	Compute Logarithm of a Sum with Signed Large Summands
lsum	Properly Compute the Logarithm of a Sum (of Exponentials)
newton	Simple R level Newton Algorithm, Mostly for Didactical Reasons
p1l1	Numerically Stable p1l1(t) = (t+1)*log(1+t) - t
pbetaAS_eq20	'pbeta()' Approximations of Abramowitz & Stegun, 26.5.{20,21}
pbetaRv1	Pure R Implementation of Old pbeta()
pchisqV	Wienergerm Approximations to (Non-Central) Chi-squared Probabilities
phyper1molenaar	Molenaar's Normal Approximations to the Hypergeometric Distribution
phyperAllBin	Compute Hypergeometric Probabilities via Binomial Approximations
phyperApprAS152	Normal Approximation to cumulative Hyperbolic Distribution - AS 152
phyperBin.1	HyperGeometric Distribution via Approximate Binomial Distribution
phyperBinMolenaar	HyperGeometric Distribution via Molenaar's Binomial Approximation
phyperIbeta	Pearson's incomplete Beta Approximation to the Hyperbolic Distribution
phyperPeizer	Peizer's Normal Approximation to the Cumulative Hyperbolic
phyperR	R-only version of R's original phyper() algorithm
phyperR2	Pure R version of R's C level phyper()
phypers	The Four (4) Symmetric 'phyper()' Calls
pl2curves	Plot 2 Noncentral Distribution Curves for Visual Comparison
pnbetaAppr2	Noncentral Beta Probabilities
pnchi1sq	(Probabilities of Non-Central Chi-squared Distribution for Special Cases
pnchisq	(Approximate) Probabilities of Non-Central Chi-squared Distribution
pnormAsymp	Asymptotic Approxmation of (Extreme Tail) 'pnorm()'
pnormL_LD10	Bounds for 1-Phi(.) - Mill's Ratio related Bounds for pnorm()
pntR	Non-central t Probability Distribution - Algorithms and Approximations
pow	X to Power of Y - R C API 'R_pow()'
pow1p	Accurate (1+x)^y, notably for small |x|
ppoisErr	Direct Computation of 'ppois()' Poisson Distribution Probabilities

pt_Witkovsky_Tab1	Viktor Witosky's Table_1 pt() Examples
qbetaAppr	Compute (Approximate) Quantiles of the Beta Distribution
qbinomR	Pure R Implementation of R's qbinom() with Tuning Parameters
qchisqAppr	Compute Approximate Quantiles of the Chi-Squared Distribution
qgammaAppr	Compute (Approximate) Quantiles of the Gamma Distribution
qnbinomR	Pure R Implementation of R's qnbinom() with Tuning Parameters
qnchisqAppr	Compute Approximate Quantiles of Noncentral Chi-Squared Distribution
qnormAppr	Approximations to 'qnorm()', i.e., z_alpha
qnormAsymp	Asymptotic Approximation to Outer Tail of qnorm()
qnormR	Pure R version of R's 'qnorm()' with Diagnostics and Tuning Parameters
qntR	Pure R Implementation of R's qt() / qnt()
qpoisR	Pure R Implementation of R's qpois() with Tuning Parameters
qtAppr	Compute Approximate Quantiles of the (Non-Central) t-Distribution
qtR	Pure R Implementation of R's C-level t-Distribution Quantiles 'qt()'
qtU	'uniroot()'-based Computing of t-Distribution Quantiles
r_pois	Compute Relative Size of i-th term of Poisson Distribution Series
rexpm1	TOMS 708 Approximation REXP(x) of expm1(x) = exp(x) - 1
stirlerr	Stirling's Error Function - Auxiliary for Gamma, Beta, etc

Further information is available in the following vignettes:

Noncentral-Chisq	Noncentral Chi-Squared Probabilities – Algorithms in R (source)
comp-beta	Computing Beta(a,b) for Large Arguments (source)
log1pmx-etc	log1pmx, bd0, stirlerr - Probability Computations in R (source)
qnorm-asymp	Asymptotic Tail Formulas For Gaussian Quantiles (source)

An important goal is to investigate diverse algorithms and approximations of R's own density (d*()), probability (p*()), and quantile (q*()) functions, notably in "border" cases where the traditional published algorithms have shown to be suboptimal, not quite accurate, or even useless.

Examples are border cases of the beta distribution, or **non-central** distributions such as the non-central chi-squared and t-distributions.

Author(s)

Principal author and maintainer: Martin Maechler <maechler@stat.math.ethz.ch>

See Also

The package **DPQmpfr**, which builds on this package and on **Rmpfr**.

Examples

```
## Show problem in R's non-central t-distrib. density (and approximations):
example(dntJKBf)
```

`algdiv` *Compute log(gamma(b)/gamma(a+b)) when b >= 8*

Description

Computes

$$\texttt{algdiv(a,b)} := \log \frac{\Gamma(b)}{\Gamma(a+b)} = \log \Gamma(b) - \log \Gamma(a+b) = \texttt{lgamma(b) - lgamma(a+b)}$$

in a numerically stable way.

This is an auxiliary function in R's (TOMS 708) implementation of `pbeta()`, aka the incomplete beta function ratio.

Usage

```
algdiv(a, b)
```

Arguments

a, b	numeric vectors which will be recycled to the same length.

Details

Note that this is also useful to compute the Beta function

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

Clearly,

$$\log B(a,b) = \log \Gamma(a) + \mathrm{algdiv(a,b)} = \log \Gamma(a) - \mathrm{logQab}(a,b)$$

In our '`../tests/qbeta-dist.R`' we look into computing $\log(pB(p,q))$ accurately for $p \ll q$.

We are proposing a nice solution there.
How is this related to `algdiv()` ?

Additionally, we have defined

$$Qab = Q_{a,b} := \frac{\Gamma(a+b), \Gamma(b)}{,}$$

such that `logQab(a,b)` $:= \log Qab(a,b)$ fulfills simply

$$\texttt{logQab(a,b)} = -\texttt{algdiv(a,b)}$$

see `logQab_asy`.

Value

a numeric vector of length `max(length(a), length(b))` (if neither is of length 0, in which case the result has length 0 as well).

Author(s)

Didonato, A. and Morris, A., Jr, (1992); `algdiv()`'s C version from the R sources, authored by the R core team; C and R interface: Martin Maechler

References

Didonato, A. and Morris, A., Jr, (1992) Algorithm 708: Significant digit computation of the incomplete beta function ratios, *ACM Transactions on Mathematical Software* **18**, 360–373.

See Also

`gamma`, `beta`; my own `logQab_asy()`.

Examples

```
Qab <- algdiv(2:3, 8:14)
cbind(a = 2:3, b = 8:14, Qab) # recycling with a warning

## algdiv()  and my  logQab_asy()  give *very* similar results for largish b:
all.equal( -   algdiv(3, 100),
           logQab_asy(3, 100), tolerance=0) # 1.283e-16 !!
(lQab <- logQab_asy(3, 1e10))
## relative error
1 + lQab/ algdiv(3, 1e10) # 0 (64b F 30 Linux; 2019-08-15)

## in-and outside of "certified" argument range {b >= 8}:
a. <- c(1:3, 4*(1:8))/32
b. <- seq(1/4, 20, by=1/4)
ad <- t(outer(a., b., algdiv))
## direct computation:
f.algdiv <- function(a,b) lgamma(b) - lgamma(a+b)
ad.d <- t(outer(a., b., f.algdiv))

matplot (b., ad.d, type = "o", cex=3/4,
         main = quote(log(Gamma(b)/Gamma(a+b)) ~"  vs.  algdiv(a,b)"))
```

```
mtext(paste0("a[1:",length(a.),"] = ",
        paste0(paste(head(paste0(formatC(a.*32), "/32")), collapse=", "), ", .., 1")))
matlines(b., ad,   type = "l", lwd=4, lty=1, col=adjustcolor(1:6, 1/2))
abline(v=1, lty=3, col="midnightblue")
# The larger 'b', the more accurate the direct formula wrt algdiv()
all.equal(ad[b. >= 1,], ad.d[b. >= 1,]       )# 1.5e-5
all.equal(ad[b. >= 2,], ad.d[b. >= 2,], tol=0)# 3.9e-9
all.equal(ad[b. >= 4,], ad.d[b. >= 4,], tol=0)# 4.6e-13
all.equal(ad[b. >= 6,], ad.d[b. >= 6,], tol=0)# 3.0e-15
all.equal(ad[b. >= 8,], ad.d[b. >= 8,], tol=0)# 2.5e-15 (not much better)
```

`Bern` *Bernoulli Numbers*

Description

Return the n-th Bernoulli number B_n, (or B_n^+, see the reference), where $B_1 = +\frac{1}{2}$.

Usage

```
Bern(n, verbose = getOption("verbose", FALSE))
```

Arguments

`n`	integer, $n \geq 0$.
`verbose`	logical indicating if computation should be traced.

Value

The number B_n of type `numeric`.

A side effect is the *caching* of computed Bernoulli numbers in the hidden `environment` `.bernoulliEnv`.

Author(s)

Martin Maechler

References

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

See Also

`Bernoulli` in **Rmpfr** in arbitrary precision via Riemann's ζ function.

The next version of package **gmp** is to contain `BernoulliQ()`, providing exact Bernoulli numbers as big rationals (class `"bigq"`).

Examples

```
(B.0.10 <- vapply(0:10, Bern, 1/2))
## [1]  1.00000000 +0.50000000  0.16666667  0.00000000 -0.03333333  0.00000000
## [7]  0.02380952  0.00000000 -0.03333333  0.00000000  0.07575758
if(requireNamespace("MASS")) {
  print( MASS::fractions(B.0.10) )
  ## 1  +1/2   1/6    0  -1/30     0  1/42     0 -1/30     0  5/66
}
```

---

bpser pbeta() *'bpser' series computation*

Description

Compute the bpser series approximation of pbeta, the incomplete beta function. Note that when b is integer valued, the series is a *sum* of $b + 1$ terms.

Usage

```
bpser(a, b, x, log.p = FALSE, eps = 1e-15, verbose = FALSE, warn = TRUE)
```

Arguments

a, b	numeric and non-negative, the two shape parameters of the beta distribution.
x	numeric vector of abscissa values in $[0, 1]$.
log.p	a logical if log(prob) should be returned, allowing to avoid underflow much farther "out in the tails".
eps	series convergence (and other) tolerance, a small positive number.
verbose	a logical indicating if some intermediate results should be printed to the console.
warn	a logical indicating if bpser() computation problems should be warned about *in addition* to return a non-zero error code.

Value

a list with components

r	the resulting numeric vector.
ier	an integer vector of the same length as x, providing one error code for the computation in each r[i].

Author(s)

Martin Maechler, ported to **DPQ**; R-Core team for the code in R.

References

TOMS 708, see pbeta

See Also

R's pbeta; **DPQ**'s pbetaRv1(), and Ixpq(); **Rmpfr**'s pbetaI

Examples

```
with(bpser(100000, 11, (0:64)/64), # all 0 {last one "wrongly"}
     stopifnot(r == c(rep(0, 64), 1), err == 0))
bp1e5.11L <- bpser(100000, 11, (0:64)/64, log.p=TRUE)# -> 2 "underflow to -Inf" warnings!
pbe <- pbeta((0:64)/64, 100000, 11, log.p=TRUE)

## verbose=TRUE showing info on number of terms / iterations
ps11.5 <- bpser(100000, 11.5, (0:64)/64, log.p=TRUE, verbose=TRUE)
```

---

b_chi *Compute $E[\chi_\nu]/\sqrt{\nu}$ useful for t- and chi-Distributions*

Description

$$b_\chi(\nu) := E[\chi(\nu)]/\sqrt{\nu} = \frac{\sqrt{2/\nu}\,\Gamma((\nu+1)/2)}{\Gamma(\nu/2)},$$

where $\chi(\nu)$ denotes a chi-distributed random variable, i.e., the square of a chi-squared variable, and $\Gamma(z)$ is the Gamma function, gamma() in R.

This is a relatively important auxiliary function when computing with non-central t distribution functions and approximations, specifically see Johnson et al.(1994), p.520, after (31.26a), e.g., our pntJW39().

Its logarithm,

$$lb_\chi(\nu) := log\Big(\frac{\sqrt{2/\nu}\,\Gamma((\nu+1)/2)}{\Gamma(\nu/2)}\Big),$$

is even easier to compute via lgamma and log, and I have used Maple to derive an asymptotic expansion in $\frac{1}{\nu}$ as well.

Note that $lb_\chi(\nu)$ also appears in the formula for the t-density (dt) and distribution (tail) functions.

Usage

```
b_chi     (nu, one.minus = FALSE, c1 = 341, c2 = 1000)
b_chiAsymp(nu, order = 2, one.minus = FALSE)
#lb_chi    (nu, ......) # not yet
lb_chiAsymp(nu, order)

c_dt(nu)     # warning("FIXME: current c_dt() is poor -- base it on lb_chi(nu) !")
c_dtAsymp(nu)  # deprecated in favour of lb_chi(nu)
c_pt(nu)     # warning("use better c_dt()") %---> FIXME deprecate even stronger ?
```

Arguments

nu	non-negative numeric vector of degrees of freedom.
one.minus	logical indicating if $1 - b()$ should be returned instead of $b()$.
c1, c2	boundaries for different approximation intervals used: for 0 < nu <= c1, internal b1() is used, for c1 < nu <= c2, internal b2() is used, and for c2 < nu, the b_chiAsymp() function is used, (and you can use that explicitly, also for smaller nu). FIXME: c1 and c2 were defined when the only asymptotic expansion known to me was the order = 2 one. A future version of b_chi will *very likely* use b_chiAsymp(*, order) for higher orders, and the c1 and c2 arguments will change, possibly be abolished.
order	the polynomial order in $\frac{1}{\nu}$ of the asymptotic expansion of $b_\chi(\nu)$ for $\nu \to \infty$. The default, order = 2 corresponds to the order you can get out of the Abramowitz and Stegun (6.1.47) formula. Higher order expansions were derived using Maple by Martin Maechler in 2002, see below, but implemented in b_chiAsymp() only in 2018.

Details

One can see that b_chi() has the properties of a CDF of a continuous positive random variable: It grows monotonely from $b_\chi(0) = 0$ to (asymptotically) one. Specifically, for large nu, b_chi(nu) = b_chiAsymp(nu) and

$$1 - b_\chi(\nu) \sim \frac{1}{4\nu}.$$

More accurately, derived from Abramowitz and Stegun, 6.1.47 (p.257) for a= 1/2, b=0,

$$\Gamma(z + 1/2)/\Gamma(z) \sim \sqrt(z) * (1 - 1/(8z) + 1/(128z^2) + O(1/z^3)),$$

and applied for $b_\chi(\nu)$ with $z = \nu/2$, we get

$$b_\chi(\nu) \sim 1 - (1/(4\nu) * (1 - 1/(8\nu)) + O(\nu^{-3})),$$

which has been implemented in b_chiAsymp(*, order=2) in 1999.

Even more accurately, Martin Maechler, used Maple to derive an asymptotic expansion up to order 15, here reported up to order 5, namely with $r := \frac{1}{4\nu}$,

$$b_\chi(\nu) = c_\chi(r) = 1 - r + \frac{1}{2}r^2 + \frac{5}{2}r^3 - \frac{21}{8}r^4 - \frac{399}{8}r^5 + O(r^6).$$

Value

a numeric vector of the same length as nu.

Author(s)

Martin Maechler

References

Johnson, Kotz, Balakrishnan (1995) *Continuous Univariate Distributions*, Vol 2, 2nd Edition; Wiley; Formula on page 520, after (31.26a)

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain.

See Also

The t-distribution (base R) page pt; our pntJW39().

Examples

```
curve(b_chi, 0, 20); abline(h=0:1, v=0, lty=3)
r <- curve(b_chi, 1e-10, 1e5, log="x")
with(r, lines(x, b_chi(x, one.minus=TRUE), col = 2))

## Zoom in to c1-region
rc1 <- curve(b_chi, 340.5, 341.5, n=1001)# nothing to see
e <- 1e-3; curve(b_chi, 341-e, 341+e, n=1001) # nothing
e <- 1e-5; curve(b_chi, 341-e, 341+e, n=1001) # see noise, but no jump
e <- 1e-7; curve(b_chi, 341-e, 341+e, n=1001) # see float "granularity"+"jump"

## Zoom in to c2-region
rc2 <- curve(b_chi, 999.5, 1001.5, n=1001) # nothing visible
e <- 1e-3; curve(b_chi, 1000-e, 1000+e, n=1001) # clear small jump
c2 <- 1500
e <- 1e-3; curve(b_chi(x,c2=c2), c2-e, c2+e, n=1001)# still
## - - - -
c2 <- 3000
e <- 1e-3; curve(b_chi(x,c2=c2), c2-e, c2+e, n=1001)# ok asymp clearly better!!
curve(b_chiAsymp, add=TRUE, col=adjustcolor("red", 1/3), lwd=3)
if(requireNamespace("Rmpfr")) {
 xm <- Rmpfr::seqMpfr(c2-e, c2+e, length.out=1000)

}
## - - - -
c2 <- 4000
e <- 1e-3; curve(b_chi(x,c2=c2), c2-e, c2+e, n=1001)# ok asymp clearly better!!
curve(b_chiAsymp, add=TRUE, col=adjustcolor("red", 1/3), lwd=3)

grCol <- adjustcolor("forest green", 1/2)
curve(b_chi,                    1/2, 1e11, log="x")
curve(b_chiAsymp, add = TRUE, col = grCol, lwd = 3)
## 1-b(nu) ~= 1/(4 nu) a power function <==> linear in log-log scale:
curve(b_chi(x, one.minus=TRUE), 1/2, 1e11, log="xy")
curve(b_chiAsymp(x, one.minus=TRUE), add = TRUE, col = grCol, lwd = 3)
```

chebyshevPoly *Chebyshev Polynomial Evaluation*

Description

Provides (evaluation of) Chebyshev polynomials, given their coefficients vector `coef` (using $2c_0$, i.e., `2*coef[1]` as the base R mathlib `chebyshev*()` functions. Specifically, the following sum is evaluated:

$$\sum_{j=0}^{n} c_j T_j(x)$$

where c_0 :=`coef[1]` and c_j :=`coef[j+1]` for $j \geq 1$. n := `chebyshev_nc(coef, .)` is the maximal degree and hence one less than the number of terms, and $T_j()$ is the Chebyshev polynomial (of the first kind) of degree j.

Usage

```
chebyshevPoly(coef, nc = chebyshev_nc(coef, eta), eta = .Machine$double.eps/20)

chebyshev_nc(coef, eta = .Machine$double.eps/20)
chebyshevEval(x, coef,
              nc = chebyshev_nc(coef, eta), eta = .Machine$double.eps/20)
```

Arguments

`coef`	a numeric vector of coefficients for the Chebyshev polynomial.
`nc`	the maximal degree, i.e., one less than the number of polynomial terms to use; typically use the default.
`eta`	a positive number; typically keep the default.
`x`	for `chebyshevEval()`: numeric vector of abscissa values at which the polynomial should be evaluated. Typically `x` values are inside the interval $[-1, 1]$.

Value

`chebyshevPoly()` returns `function(x)` which computes the values of the underlying Chebyshev polynomial at `x`.

`chebyshev_nc()` returns an `integer`, and `chebyshevEval(x, coef)` returns a numeric "like" `x` with the values of the polynomial at `x`.

Author(s)

R Core team, notably Ross Ihaka; Martin Maechler provided the R interface.

References

https://en.wikipedia.org/wiki/Chebyshev_polynomials

See Also

`polyn.eval()` from CRAN package **sfsmisc**; as one example of many more.

Examples

```
## The first 5 (base) Chebyshev polynomials:
T0 <- chebyshevPoly(2)  # !! 2, not 1
T1 <- chebyshevPoly(0:1)
T2 <- chebyshevPoly(c(0,0,1))
T3 <- chebyshevPoly(c(0,0,0,1))
T4 <- chebyshevPoly(c(0,0,0,0,1))
curve(T0(x), -1,1, col=1, lwd=2, ylim=c(-1,1))
abline(h=0, lty=2)
curve(T1(x), col=2, lwd=2, add=TRUE)
curve(T2(x), col=3, lwd=2, add=TRUE)
curve(T3(x), col=4, lwd=2, add=TRUE)
curve(T4(x), col=5, lwd=2, add=TRUE)

(Tv <- vapply(c(T0=T0, T1=T1, T2=T2, T3=T3, T4=T4),
              function(Tp) Tp(-1:1), numeric(3)))
x <- seq(-1,1, by = 1/64)
stopifnot(exprs = {
   all.equal(chebyshevPoly(1:5)(x),
             0.5*T0(x) + 2*T1(x) + 3*T2(x) + 4*T3(x) + 5*T4(x))
   all.equal(unname(Tv), rbind(c(1,-1), c(1:-1,0:1), rep(1,5)))# warning on rbind()
})
```

`dbinom_raw` *R's C Mathlib (Rmath) dbinom_raw() Binomial Probability pure R Function*

Description

A pure R implementation of R's C API ('Mathlib' specifically) `dbinom_raw()` function which computes binomial probabilities *and* is continuous in `x`, i.e., also "works" for non-integer `x`.

Usage

```
dbinom_raw (x, n, p, q = 1-p, log = FALSE,
            version = c("2008", "R4.4"),
            verbose = getOption("verbose"))
```

Arguments

`x`	vector with values typically in `0:n`, but here allowed to non-integer values.
`n`	called `size` in R's `dbinom()`.
`p`	called `prob` in R's `dbinom()`, the success probability, hence in $[0, 1]$.

q	mathemtically the same as $1 - p$, but may be (much) more accurate, notably when small.
log	logical indicating if the log() of the resulting probability should be returned; useful notably in case the probability itself would underflow to zero.
version	a character string; originally, "2008" was the only option. Still the default currently, this *may change* in the future.
verbose	integer indicating the amount of verbosity of diagnostic output, 0 means no output, 1 more, etc.

Value

numeric vector of the same length as x which may have to be thought of recycled along n, p and/or q.

Author(s)

R Core and Martin Maechler

See Also

Note that our CRAN package **Rmpfr** provides dbinom, an mpfr-accurate function to be used used instead of R's or this pure R version relying bd0() and stirlerr() where the latter currently only provides accurate double precision accuracy.

Examples

```
for(n in c(3, 10, 27, 100, 500, 2000, 5000, 1e4, 1e7, 1e10)) {
 x <- if(n <= 2000) 0:n else round(seq(0, n, length.out=2000))
 p <- 3/4
 stopifnot(all.equal(dbinom_raw(x, n, p, q=1-p) -> dbin,
                 dbinom    (x, n, p), tolerance = 1e-13))# 1.636e-14 (Apple clang 14.0.3)
 stopifnot(all.equal(dbin, dbinom_raw(x, n, p, q=1-p, version = "R4.4") -> dbin44,
                     tolerance = 1e-13))
 cat("n = ", n, ": ", (aeq <- all.equal(dbin44, dbin, tolerance = 0)), "\n")
 if(n < 3000) stopifnot(is.character(aeq)) # check that dbin44 is "better" ?!
}

n <- 1024 ; x <- 0:n
plot(x, dbinom_raw(x, n, p, q=1-p) - dbinom(x, n, p), type="l", main = "|db_r(x) - db(x)|")
plot(x, dbinom_raw(x, n, p, q=1-p) / dbinom(x, n, p) - 1, type="b", log="y",
     main = "rel.err.  |db_r(x / db(x) - 1)|")
```

dchisqApprox *Approximations of the (Noncentral) Chi-Squared Density*

Description

Compute the density function $f(x, *)$ of the (noncentral) chi-squared distribution.

Usage

```
dnchisqR     (x, df, ncp, log = FALSE,
              eps = 5e-15, termSml = 1e-10, ncpLarge = 1000)
dnchisqBessel(x, df, ncp, log = FALSE)
dchisqAsym   (x, df, ncp, log = FALSE)
dnoncentchisq(x, df, ncp, kmax = floor(ncp/2 + 5 * (ncp/2)^0.5))
```

Arguments

x	non-negative numeric vector.
df	degrees of freedom (parameter), a positive number.
ncp	non-centrality parameter δ; ....
log	logical indicating if the result is desired on the log scale.
eps	positive convergence tolerance for the series expansion: Terms are added while term * q > (1-q)*eps, where q is the term's multiplication factor.
termSml	positive tolerance: in the series expansion, terms are added to the sum as long as they are not smaller than termSml * sum even when convergence according to eps had occured. This was not part of the original C code, but was added later for safeguarding against infinite loops, from PR#14105, e.g., for dchisq(2000, 2, 1000).
ncpLarge	in the case where mid underflows to 0, when log is true, or ncp >= ncpLarge, use a central approximation. In theory, an optimal choice of ncpLarge would not be arbitrarily set at 1000 (hardwired in R's dchisq() here), but possibly also depend on x or df.
kmax	the number of terms in the sum for dnoncentchisq().

Details

dnchisqR() is a pure R implementation of R's own C implementation in the sources, 'R/src/nmath/dnchisq.c', additionally exposing the three "tuning parameters" eps, termSml, and ncpLarge.

dnchisqBessel() implements Fisher(1928)'s exact closed form formula based on the Bessel function I_{nu}, i.e., R's besselI() function; specifically formula (29.4) in Johnson et al. (1995).

dchisqAsym() is the simple asymptotic approximation from Abramowitz and Stegun's formula 26.4.27, p. 942.

dnoncentchisq() uses the (typically defining) infinite series expansion directly, with truncation at kmax, and terms t_k which are products of a Poisson probability and a central chi-square density, i.e., terms t.k := dpois(k, lambda = ncp/2) * dchisq(x, df = 2*k + df) for k = 0, 1, ..., kmax.

Value

numeric vector similar to x, containing the (logged if log=TRUE) values of the density $f(x, *)$.

Note

These functions are mostly of historical interest, notably as R's dchisq() was not always very accurate in the noncentral case, i.e., for ncp > 0.

Note

R's dchisq() is typically more uniformly accurate than the approximations nowadays, apart from dnchisqR() which should behave the same. There may occasionally exist small differences between dnchisqR(x, *) and dchisq(x, *) for the same parameters.

Author(s)

Martin Maechler, April 2008

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain.

Johnson, N.L., Kotz, S. and Balakrishnan, N. (1995) Continuous Univariate Distributions Vol~2, 2nd ed.; Wiley; chapter 29, Section *3 Distribution*, (29.4), p. 436.

See Also

R's own dchisq().

Examples

```
x <- sort(outer(c(1,2,5), 2^(-4:5)))
fRR <- dchisq  (x, 10, 2)
f.R <- dnchisqR(x, 10, 2)
all.equal(fRR, f.R, tol = 0) # 64bit Lnx (F 30): 1.723897e-16
stopifnot(all.equal(fRR, f.R, tol = 4e-15))
```

---

dgamma-utils *Binomial Deviance – Auxiliary Functions for* dgamma() *Etc*

Description

The "binomial deviance" function bd0(x, M) := $D_0(x, M) := M \cdot d_0(x/M)$, where $d_0(r) := r \log(r) + 1 - r \stackrel{!}{=} p_1 l_1(r-1)$. Mostly, pure R transcriptions of the C code utility functions for dgamma(), dbinom(), dpois(), dt(), and similar "base" density functions by Catherine Loader. These have extra arguments with defaults that correspond to R's Mathlib C code hardwired cutoffs and tolerances.

Usage

```
dpois_raw(x, lambda, log=FALSE,
          version,
          ## the defaults for `version` will probably change in the future
          small.x__lambda = .Machine$double.eps,
          bd0.delta = 0.1,
          ## optional arguments of log1pmx() :
        tol_logcf = 1e-14, eps2 = 0.01, minL1 = -0.79149064, trace.lcf = verbose,
          logCF = if (is.numeric(x)) logcf else logcfR,
          verbose = FALSE)

dpois_simpl (x, lambda, log=FALSE)
dpois_simpl0(x, lambda, log=FALSE)

bd0(x, np, delta = 0.1,
    maxit = max(1L, as.integer(-1100 / log2(delta))),
    s0 = .Machine$double.xmin,
    verbose = getOption("verbose"))
bd0C(x, np, delta = 0.1, maxit = 1000L, version = c("R4.0", "R_2025_0510"),
     verbose = getOption("verbose"))
# "simple" log1pmx() based versions :
bd0_p1l1d1(x, M, tol_logcf = 1e-14, ...)
bd0_p1l1d (x, M, tol_logcf = 1e-14, ...)
# p1l1() based version {possibly using log1pmx(), too}:
bd0_p1l1  (x, M, ...)
# using faster formula for large |M-x|/x
bd0_l1pm  (x, M, tol_logcf = 1e-14, ...)

ebd0 (x, M, verbose = getOption("verbose"), ...) # experimental, may disappear !!
ebd0C(x, M, verbose = getOption("verbose"))
```

Arguments

x	numeric (or number-alike such as "mpfr").
lambda, np, M	each numeric (or number-alike ..); distribution parameters.
log	logical indicating if the log-density should be returned, otherwise the density at x.
verbose	logical indicating if some information about the computations are to be printed.
small.x__lambda	small positive number; for dpois_raw(x, lambda), when x/lambda is not larger than small.x__lambda, the direct log poisson formula is used instead of ebd0() or bd0() and stirlerr().
delta, bd0.delta	a non-negative number $\delta < 1$, a cutoff for bd0() where a continued fraction series expansion is used when $\lvert x - M\rvert \le \delta \cdot (x + M)$.

`tol_logcf`, `eps2`, `minL1`, `trace.lcf`, `logCF`, ...
: optional tuning arguments passed to `log1pmx()`, and to its options passed to `logcf()`.

`maxit`
: the number of series expansion terms to be used in `bd0()` when $|x-M|$ is small. The default is k such that $\delta^{2k} \le 2^{-1022-52}$, i.e., will underflow to zero.

`s0`
: the very small s_0 determining that `bd0() = s` already before the locf series expansion.

`version`
: a `character` string specifying the version of $bd0()$ to use. The versions available *and* the current default are partly experimental!

Details

`bd0()`: Loader's "Binomial Deviance" function; for $x, M > 0$ (where the limit $x \to 0$ is allowed). In the case of `dbinom`, x are integers (and $M = np$), but in general `x` is real.

$$bd_0(x, M) := M \cdot D_0(\frac{x}{M}),$$

where $D_0(u) := u\log(u) + 1 - u = u(\log(u) - 1) + 1$. Hence

$$bd_0(x, M) = M \cdot \big(\frac{x}{M}(\log(\frac{x}{M}) - 1) + 1\big) = x\log(\frac{x}{M}) - x + M.$$

A different way to rewrite this from Martyn Plummer, notably for important situation when $|x - M| \ll M$, is using $t := (x - M)/M$ (and $|t| \ll 1$ for that situation), equivalently, $\frac{x}{M} = 1 + t$. Using t,

$$bd_0(x, M) = \log(1+t) - t\cdot M = M\cdot[(t+1)(\log(1+t)-1)+1] = M\cdot[(t+1)\log(1+t) - t] = M\cdot p_1 l_1(t),$$

and

$$p_1 l_1(t) := (t + 1)\log(1 + t) - t = \frac{t^2}{2} - \frac{t^3}{6} \ldots$$

where the Taylor series expansion is useful for small $|t|$, see `p1l1`.

Note that `bd0(x, M)` now also works when `x` and/or `M` are arbitrary-accurate mpfr-numbers (package **Rmpfr**).

`bd0C()` interfaces to C code which corresponds to R's C Mathlib (Rmath) `bd0()`.

Value

a numeric vector "like" `x`; in some cases may also be an (high accuracy) "mpfr"-number vector, using CRAN package **Rmpfr**.

`ebd0()` (R code) and `ebd0C()` (interface to `C` code) are *experimental*, meant to be precision-extended version of `bd0()`, returning `(yh, yl)` (high- and low-part of `y`, the numeric result). In order to work for *long* vectors `x`, `yh`, `yl` need to be `list` components; hence we return a two-column `data.frame` with column names `"yh"` and `"yl"`.

Author(s)

Martin Maechler

References

C. Loader (2000), see dbinom's documentation.

Martin Mächler (2021 ff) log1pmx, ... *Computing ... Probabilities in* R. **DPQ** package vignette https://CRAN.R-project.org/package=DPQ/vignettes/log1pmx-etc.pdf.

See Also

p1l1() and its variants, e.g., p1l1ser(); log1pmx() which is called from bd0_p1l1d*() and bd0_l1pm(). Then, stirlerr for Stirling's error function, complementing bd0() for computation of Gamma, Beta, Binomial and Poisson probabilities. R's own dgamma, dpois.

Examples

```
x <- 800:1200
bd0x1k <- bd0(x, np = 1000)
plot(x, bd0x1k, type="l", ylab = "bd0(x, np=1000)")
bd0x1kC <- bd0C(x, np = 1000)
lines(x, bd0x1kC, col=2)
bd0.1d1 <- bd0_p1l1d1(x, 1000)
bd0.1d  <- bd0_p1l1d (x, 1000)
bd0.1p1 <- bd0_p1l1  (x, 1000)
bd0.1pm <- bd0_l1pm  (x, 1000)
stopifnot(exprs = {
    all.equal(bd0x1kC, bd0x1k,  tol=1e-14) # even tol=0 currently ..
    all.equal(bd0x1kC, bd0.1d1, tol=1e-14)
    all.equal(bd0x1kC, bd0.1d , tol=1e-14)
    all.equal(bd0x1kC, bd0.1p1, tol=1e-14)
    all.equal(bd0x1kC, bd0.1pm, tol=1e-14)
})

str(log1pmx) #--> play with { tol_logcf, eps2, minL1, trace.lcf, logCF }  --> vignette

ebd0x1k <- ebd0 (x, 1000)
exC     <- ebd0C(x, 1000)
stopifnot(all.equal(exC, ebd0x1k,          tol = 4e-16),
          all.equal(bd0x1kC, rowSums(exC), tol = 2e-15))
lines(x, rowSums(ebd0x1k), col=adjustcolor(4, 1/2), lwd=4)

## very large args --> need new version "R_2025..."
np <- 117e306;  np / .Machine$double.xmax # 0.65
x <- (1:116)*1e306
tail(bd0L <-  bd0C(x, np, version = "R4")) # underflow to 0
tail(bd0Ln <- bd0C(x, np, version = "R_2025_0510"))
bd0L.R <- bd0(x, np)
all.equal(bd0Ln, bd0L.R, tolerance = 0) # see TRUE
stopifnot(exprs = {
    is.finite(bd0Ln)
    bd0Ln > 4e303
    tail(bd0L, 54) == 0
    all.equal(bd0Ln, np*p1l1((x-np)/np), tolerance = 5e-16)
    all.equal(bd0Ln, bd0L.R, tolerance = 5e-16)
```

```
})
```

---

dgamma.R *Gamma Density Function Alternatives*

Description

`dgamma.R()` is aimed to be an R level "clone" of R's C level implementation `dgamma` (from package **stats**).

Usage

```
dgamma.R(x, shape, scale = 1, log,
         dpois_r_args = list())
```

Arguments

`x`	non-negative numeric vector.
`shape`	non-negative shape parameter of the Gamma distribution.
`scale`	positive scale parameter; note we do not see the need to have a `rate` parameter as the standard R function.
`log`	logical indicating if the result is desired on the log scale.
`dpois_r_args`	a `list` of optional arguments for `dpois_raw()`; not much checked, must be specified correctly.

Value

numeric vector of the same length as `x` (which may have to be thought of recycled along `shape` and/or `scale`.

Author(s)

Martin Maechler

See Also

(As R's C code) this depends crucially on the "workhorse" function `dpois_raw()`.

Examples

```
xy  <- curve(dgamma  (x, 12), 0,30) # R's dgamma()
xyR <- curve(dgamma.R(x, 12, dpois_r_args = list(verbose=TRUE)), add=TRUE,
             col = adjustcolor(2, 1/3), lwd=3)
stopifnot(all.equal(xy, xyR, tolerance = 4e-15)) # seen 7.12e-16
## TODO: check *vectorization* in x --> add tests/*.R ___ TODO ___


## From R's  <R>/tests/d-p-q-r-tst-2.R -- replacing dgamma() w/ dgamma.R()
## PR#17577 - dgamma(x, shape)  for shape < 1 (=> +Inf at x=0) and very small x
stopifnot(exprs = {
    all.equal(dgamma.R(2^-1027, shape = .99 , log=TRUE), 7.1127667376, tol=1e-10)
    all.equal(dgamma.R(2^-1031, shape = 1e-2, log=TRUE), 702.8889158,  tol=1e-10)
    all.equal(dgamma.R(2^-1048, shape = 1e-7, log=TRUE), 710.30007699, tol=1e-10)
    all.equal(dgamma.R(2^-1048, shape = 1e-7, scale = 1e-315, log=TRUE),
              709.96858768, tol=1e-10)
})
## R's dgamma() gave all Inf in R <= 3.6.1 [and still there in 32-bit Windows !]
```

`dhyperBinMolenaar` *HyperGeometric (Point) Probabilities via Molenaar's Binomial Approximation*

Description

Compute hypergeometric (point) probabilities via Molenaar's binomial approximation, `hyper2binomP()`.

Usage

```
dhyperBinMolenaar(x, m, n, k, log = FALSE)
```

Arguments

`x`	(vector of) the number of white balls drawn without replacement from an urn which contains both black and white balls.
`m`	the number of white balls in the urn.
`n`	the number of black balls in the urn.
`k`	the number of balls drawn from the urn, hence in $0, 1, \ldots, m + n$.
`log`	`logical` indication if the logarithm `log(P)` should be returned (instead of P).

Value

a `numeric` vector, with the length the maximum of the lengths of `x`, `m`, `n`, `k`.

Author(s)

Martin Maechler

References

See those in phyperBinMolenaar.

See Also

hyper2binomP(); R's own dhyper() which uses more sophisticated computations.

Examples

```
## The function is simply defined as
function (x, m, n, k, log = FALSE)
  dbinom(x, size = k, prob = hyper2binomP(x, m, n, k), log = log)
```

---

dltgammaInc — *TOMS 1006 - Fast and Accurate Generalized Incomplete Gamma Function*

Description

This is the deltagammainc algorithm as described in Abergel and Moisan (2020).

It uses one of three different algorithms to compute $G(p, x)$, their conveniently *scaled* generalization of the (upper and lower) incomplete Gamma functions,

$$G(p,x) = e^{x-p\log x} \times \gamma(p,x) \text{ if} x \leq por\Gamma(p,x)\text{otherwise},$$

for $\gamma(p, x)$ and $\Gamma(p, x)$ the lower and upper incomplete gamma functions, and for all $x \geq 0$ and $p > 0$.

dltgammaInc(x,y, mu, p) := $I_{x,y}^{\mu,p}$ computes their (generalized incomplete gamma) integral

$$I_{x,y}^{\mu,p} := \int_x^y s^{p-1} e^{-\mu s}\, ds.$$

Current explorations (via package **Rmpfr**'s igamma()) show that this algorithm is *less accurate* than R's own pgamma() at least for small p.

Usage

```
dltgammaInc(x, y, mu = 1, p)
```

Arguments

x, y, p	numeric vectors, typically of the same length; otherwise recycled to common length.
mu	a number different from 0; must be finite; 1 by default.

Value

a `list` with three components

`rho`, `sigma`	numeric vectors of same length (the common length of `x`,`y`,.., after recycling). `ans := rho * exp(sigma)` are the computed values of $G(p, x)$.
`method`	integer code indicating the algorithm used for the computation of `(rho, sigma)`.

Author(s)

C source from Remy Abergel and Lionel Moisan, see the reference; original is in '`1006.zip`' at `https://calgo.acm.org/`.

Interface to R in **DPQ**: Martin Maechler

References

Rémy Abergel and Lionel Moisan (2020) Algorithm 1006: Fast and accurate evaluation of a generalized incomplete gamma function, *ACM Transactions on Mathematical Software* **46**(1): 1–24. doi:10.1145/3365983

See Also

Rmpfr's `igamma()`, R's `pgamma()`.

Examples

```
p <- pi
x <- y <- seq(0, 10, by = 1/8)

dltg1 <- dltgammaInc(0, y,   mu = 1, p = pi); r1 <- with(dltg1, rho * exp(sigma))  ##
dltg2 <- dltgammaInc(x, Inf, mu = 1, p = pi); r2 <- with(dltg2, rho * exp(sigma))
str(dltg1)
stopifnot(identical(c(rho = "double", sigma = "double", method = "integer"),
                    sapply(dltg1, typeof)))
summary(as.data.frame(dltg1))
pg1 <- pgamma(q = y, shape = pi, lower.tail=TRUE)
pg2 <- pgamma(q = x, shape = pi, lower.tail=FALSE)

stopifnot(all.equal(r1 / gamma(pi), pg1, tolerance = 1e-14)) # seen 5.26e-15
stopifnot(all.equal(r2 / gamma(pi), pg2, tolerance = 1e-14)) #  "   5.48e-15

if(requireNamespace("Rmpfr")) withAutoprint({
    asN  <- Rmpfr::asNumeric
    mpfr <- Rmpfr::mpfr
    iGam <- Rmpfr::igamma(a= mpfr(pi, 128),
                          x= mpfr(y,  128))
    relErrV <- sfsmisc::relErrV
    relE <- asN(relErrV(target = iGam, current = r2))
    summary(relE)
    plot(x, relE, type = "b", col=2, # not very good: at  x ~= 3, goes up to  1e-14 !
         main = "rel.Error(incGamma(pi, x)) vs  x")
    abline(h = 0, col = adjustcolor("gray", 0.75), lty = 2)
```

```
    lines(x, asN(relErrV(iGam, pg2 * gamma(pi))), col = adjustcolor(4, 1/2), lwd=2)
  ## ## ==> R's pgamma() is *much* more accurate !! ==> how embarrassing for TOMS 1006 !?!?
    legend("topright", expression(dltgammaInc(x, Inf, mu == 1, p == pi),
                                  pgamma(x, pi, lower.tail== F) %*% Gamma(pi)),
       col = c(palette()[2], adjustcolor(4, 1/2)), lwd = c(1, 2), pch = c(1, NA), bty = "n")
})
```

---

dnbinomR *Pure R Versions of R's C (Mathlib) dnbinom() Negative Binomial Probabilities*

Description

Compute pure R implementations of R's C Mathlib (Rmath) `dnbinom()` binomial probabilities, allowing to see the effect of the cutoff `eps`.

Usage

```
dnbinomR  (x, size, prob, log = FALSE, eps = 1e-10)
dnbinom.mu(x, size, mu,   log = FALSE, eps = 1e-10)
```

Arguments

`x`, `size`, `prob`, `mu`, `log`	see R's `dnbinom()`.
`eps`	non-negative number specifying the cutoff for "small `x/size`", in which case the 2-term approximation from Abramowitz and Stegun, 6.1.47 (p.257) is preferable to the `dbinom()` based evaluation.

Value

numeric vector of the same length as `x` which may have to be thought of recycled along `size` and `prob` or `mu`.

Author(s)

R Core and Martin Maechler

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain.

See Also

`dbinom_raw`; Note that our CRAN package **Rmpfr** provides `dnbinom`, `dbinom` and more, where mpfr-accurate functions are used instead of R's (and our pure R version of) `bd0()` and `stirlerr()`.

Examples

```
stopifnot( dnbinomR(0, 1, 1) == 1 )
size <- 1000 ; x <- 0:size
dnb <- dnbinomR(x, size, prob = 5/8, log = FALSE, eps = 1e-10)
plot(x, dnb, type="b")
all.equal(dnb, dnbinom(x, size, prob = 5/8)) ## mean rel. diff: 0.00017...

dnbm <- dnbinom.mu(x, size, mu = 123, eps = 1e-10)
all.equal(dnbm, dnbinom(x, size, mu = 123)) #  Mean relative diff: 0.00069...
```

---

dnt *Non-central t-Distribution Density - Algorithms and Approximations*

Description

dntJKBf1 implements the summation formulas of Johnson, Kotz and Balakrishnan (1995), (31.15) on page 516 and (31.15') on p.519, the latter being typo-corrected for a missing factor $1/j!$.

dntJKBf() is Vectorize(dntJKBf1, c("x","df","ncp")), i.e., works vectorized in all three main arguments x, df and ncp.

The functions .dntJKBch1() and .dntJKBch() are only there for didactical reasons allowing to check that indeed formula (31.15') in the reference is missing a $j!$ factor in the denominator.

The dntJKBf*() functions are written to also work with arbitrary precise numbers of class "mpfr" (from package **Rmpfr**) as arguments.

Usage

```
dntJKBf1(x, df, ncp, log = FALSE, M = 1000)
dntJKBf (x, df, ncp, log = FALSE, M = 1000)

## The "checking" versions, only for proving correctness of formula:
.dntJKBch1(x, df, ncp, log = FALSE, M = 1000, check=FALSE, tol.check = 1e-7)
.dntJKBch (x, df, ncp, log = FALSE, M = 1000, check=FALSE, tol.check = 1e-7)
```

Arguments

x, df, ncp	see R's dt(); note that each can be of class "mpfr".
log	as in dt(), a logical indicating if $\log(f(x,*))$ should be returned instead of $f(x,*)$.
M	the number of terms to be used, a positive integer.
check	logical indicating if checks of the formula equalities should be done.
tol.check	tolerance to be used for all.equal() when check is true.

Details

How to choose `M` optimally has not been investigated yet and is probably also a function of the precision of the first three arguments (see `getPrec` from **Rmpfr**).

Note that relatedly, R's source code '`R/src/nmath/dnt.c`' has claimed from 2003 till 2014 but **wrongly** that the noncentral t density $f(x, *)$ was

```
   f(x, df, ncp) =
df^(df/2) * exp(-.5*ncp^2) /
(sqrt(pi)*gamma(df/2)*(df+x^2)^((df+1)/2)) *
 sum_{k=0}^Inf  gamma((df + k + df)/2)*ncp^k / prod(1:k)*(2*x^2/(df+x^2))^(k/2) .
```

These functions (and this help page) prove that it was wrong.

Value

a number for `dntJKBf1()` and `.dntJKBch1()`.

a numeric vector of the same length as the maximum of the lengths of `x`, `df`, `ncp` for `dntJKBf()` and `.dntJKBch()`.

Author(s)

Martin Maechler

References

Johnson, N.L., Kotz, S. and Balakrishnan, N. (1995) Continuous Univariate Distributions Vol~2, 2nd ed.; Wiley; chapter 31, Section *5 Distribution Function*, p.514 ff

See Also

R's `dt`; (an improved version of) Viechtbauer's proposal: `dtWV`.

Examples

```
tt <-  seq(0, 10, length.out = 21)
ncp <- seq(0,  6, length.out = 31)
dt3R   <- outer(tt, ncp, dt,     df = 3)
dt3JKB <- outer(tt, ncp, dntJKBf, df = 3)
all.equal(dt3R, dt3JKB) # Lnx(64-b): 51 NA's in dt3R

x <- seq(-1,12, by=1/16)
fx <- dt(x, df=3, ncp=5)
re1 <- 1 - .dntJKBch(x, df=3, ncp=5) / fx ; summary(warnings()) # slow, with warnings
op <- options(warn = 2) # (=> warning == error, for now)
re2 <- 1 -  dntJKBf (x, df=3, ncp=5) / fx  # faster, no warnings
stopifnot(all.equal(re1[!is.na(re1)], re2[!is.na(re1)], tol=1e-6))
head( cbind(x, fx, re1, re2) , 20)
matplot(x, log10(abs(cbind(re1, re2))), type = "o", cex = 1/4)

## One of the numerical problems in "base R"'s non-central t-density:
```

```
options(warn = 0) # (factory def.)
x <- 2^seq(-12, 32, by=1/8) ; df <- 1/10
dtm <- cbind(dt(x, df=df,           log=TRUE),
             dt(x, df=df, ncp=df/2, log=TRUE),
             dt(x, df=df, ncp=df,   log=TRUE),
             dt(x, df=df, ncp=df*2, log=TRUE)) #.. quite a few warnings:
summary(warnings())
matplot(x, dtm, type="l", log = "x", xaxt="n",
        main = "dt(x, df=1/10, log=TRUE) central and noncentral")
sfsmisc::eaxis(1)
legend("right", legend=c("", paste0("ncp = df",c("/2","","*2"))),
       lty=1:4, col=1:4, bty="n")

(doExtras <- DPQ:::doExtras()) # TRUE e.g. if interactive()
(ncp <- seq(0, 12, by = if(doExtras) 3/4 else 2))
names(ncp) <- nnMs <- paste0("ncp=", ncp)
tt  <- seq(0, 5, by =  1)
dt3R <- outer(tt, ncp, dt,   df = 3)
if(requireNamespace("Rmpfr")) withAutoprint({
   mt  <- Rmpfr::mpfr(tt , 128)
   mcp <- Rmpfr::mpfr(ncp, 128)
   system.time(
       dt3M <- outer(mt, mcp, dntJKBf, df = 3,
                     M = if(doExtras) 1024 else 256)) # M=1024: 7 sec [10 sec on Winb]
   relE <- Rmpfr::asNumeric(sfsmisc::relErrV(dt3M, dt3R))
   relE[tt != 0, ncp != 0]
})

## all.equal(dt3R, dt3V, tol=0) # 1.2e-12

 # ---- using MPFR high accuracy arithmetic (too slow for routine testing) ---
## no such kink here:
x. <- if(requireNamespace("Rmpfr")) Rmpfr::mpfr(x, 256) else x
system.time(dtJKB <- dntJKBf(x., df=df, ncp=df, log=TRUE)) # 43s, was 21s and only 7s ???
lines(x, dtJKB, col=adjustcolor(3, 1/2), lwd=3)
options(op) # reset to prev.

## Relative Difference / Approximation errors :
plot(x, 1 - dtJKB / dtm[,3], type="l", log="x")
plot(x, 1 - dtJKB / dtm[,3], type="l", log="x", xaxt="n", ylim=c(-1,1)*1e-3); sfsmisc::eaxis(1)
plot(x, 1 - dtJKB / dtm[,3], type="l", log="x", xaxt="n", ylim=c(-1,1)*1e-7); sfsmisc::eaxis(1)
plot(x, abs(1 - dtJKB / dtm[,3]), type="l", log="xy", axes=FALSE, main =
     "dt(*, 1/10, 1/10, log=TRUE) relative approx. error",
     sub= paste("Copyright (C) 2019  Martin Maechler  --- ", R.version.string))
for(j in 1:2) sfsmisc::eaxis(j)
```

---

dot-D-utils *Distribution Utilities "dpq"*

Description

Utility functions for "dpq"-computations, parelling those in R's own C source '<Rsource>/src/nmath/dpq.h', ("dpq" := **d**ensity–**p**robability–**q**uantile).

Usage

```
.D_0(log.p) # prob/density == 0   (for log.p=FALSE)
.D_1(log.p) # prob         == 1     "       "

.DT_0(lower.tail, log.p) # == 0  when (lower.tail=TRUE, log.p=FALSE)
.DT_1(lower.tail, log.p) # == 1  when     "                 "

.D_Lval(p, lower.tail) # p    {L}ower
.D_Cval(p, lower.tail) # 1-p  {C}omplementary

.D_val (x, log.p)  #     x  in pF(x,..)
.D_qIv (p, log.p)  #     p  in qF(p,..)
.D_exp (x, log.p)  # exp(x)        unless log.p where it's  x
.D_log (p, log.p)  #     p           "      "     "    "   log(p)
.D_Clog(p, log.p)  #   1-p           "      "     "    "   log(1-p) == log1p(-)

.D_LExp(x, log.p)  ## [log](1 - exp(-x))    == log1p(- .D_qIv(x))) even more stable

.DT_val (x, lower.tail, log.p) # := .D_val(.D_Lval(x, lower.tail), log.p) ==   x  in pF
.DT_Cval(x, lower.tail, log.p) # := .D_val(.D_Cval(x, lower.tail), log.p) == 1-x  in pF

.DT_qIv (p, lower.tail, log.p) # := .D_Lval(.D_qIv(p)) ==    p  in qF
.DT_CIv (p, lower.tail, log.p) # := .D_Cval(.D_qIv(p)) ==  1-p  in qF

.DT_exp (x, lower.tail, log.p) #  exp( x )
.DT_Cexp(x, lower.tail, log.p) #  exp(1-x)

.DT_log (p, lower.tail, log.p) # log ( p )  in qF
.DT_Clog(p, lower.tail, log.p) # log (1-p)  in qF
.DT_Log (p, lower.tail)        # log ( p )  in qF(p,..,log.p=TRUE)
```

Arguments

x	numeric vector.
p	(log) probability–like numeric vector.
lower.tail	logical; if true, probabilities are $P[X \leq x]$, otherwise upper tail probabilities, $P[X > x]$.
log.p	logical; if true, probabilities p are given as $\log(p)$ in argument p.

Value

Typically a numeric vector "as" x or p, respectively.

Author(s)

Martin Maechler

See Also

log1mexp() which is called from .D_LExp() and .DT_Log().

Examples

```
FT <- c(FALSE, TRUE)
stopifnot(exprs = {
    .D_0(log.p = FALSE) ==    (0)
    .D_0(log.p = TRUE ) == log(0)
    identical(c(1,0), vapply(FT, .D_1, double(1)))
})

## all such functions in package DPQ:
eDPQ <- as.environment("package:DPQ")
ls.str(envir=eDPQ, pattern = "^[.]D", all.names=TRUE)
(nD <- local({ n <- names(eDPQ); n[startsWith(n, ".D")] }))
trimW <- function(ch) sub(" +$","", sub("^ +","", ch))
writeLines(vapply(sort(nD), function(nm) {
    B <- deparse(eDPQ[[nm]])
    sprintf("%31s := %s", trimW(sub("function ", nm, B[[1]])),
            paste(trimW(B[-1]), collapse=" "))
                  }, ""))

do.lowlog <- function(Fn, ...) {
    stopifnot(is.function(Fn),
              all(c("lower.tail", "log.p") %in% names(formals(Fn))))
    FT <- c(FALSE, TRUE) ; cFT <- c("F", "T")
   L <- lapply(FT, function(lo) sapply(FT, function(lg) Fn(..., lower.tail=lo, log.p=lg)))
    r <- simplify2array(L)
    `dimnames<-`(r, c(rep(list(NULL), length(dim(r)) - 2L),
                      list(log.p = cFT, lower.tail = cFT)))
}
do.lowlog(.DT_0)
do.lowlog(.DT_1)
do.lowlog(.DT_exp, x = 1/4) ; do.lowlog(.DT_exp, x = 3/4)
do.lowlog(.DT_val, x = 1/4) ; do.lowlog(.DT_val, x = 3/4)
do.lowlog(.DT_Cexp, x = 1/4) ; do.lowlog(.DT_Cexp, x = 3/4)
do.lowlog(.DT_Cval, x = 1/4) ; do.lowlog(.DT_Cval, x = 3/4)
do.lowlog(.DT_Clog, p = (1:3)/4) # w/ warn
do.lowlog(.DT_log,  p = (1:3)/4) # w/ warn
do.lowlog(.DT_qIv,  p = (1:3)/4)

## unfinished: FIXME, the above is *not* really checking
stopifnot(exprs = {

})
```

dpsifn *Psi Gamma Functions Workhorse from R's API*

Description

Log Gamma derivatives, Psi Gamma functions. `dpsifn()` is an R interface to the R API function `R_dpsifn()`.

Usage

```
dpsifn(x, m, deriv1 = 0L, k2 = FALSE)
```

Arguments

`x`	numeric vector.
`m`	number of derivatives to return, an integer >= 0.
`deriv1`	"start" derivative ....
`k2`	a `logical` specifying if `kode = 2` should be applied.

Details

`dpsifn()` is the underlying "workhorse" of R's own `digamma`, `trigamma` and (generalized) `psigamma` functions.

It is useful, e.g., when several derivatives of $\log\Gamma$ =`lgamma` are desired. It computes and returns length-*m* sequence $(-1)^{k+1}/\Gamma(k+1)\cdot\psi^{(k)}(x)$ for $k = n, n+1, \ldots, n+m-1$, where n =`deriv1`, and $\psi^{(k)}(x)$ is the k-th derivative of $\psi(x)$, i.e., `psigamma(x,k)`. For more details, see the comments in '`src/nmath/polygamma.c`'.

Value

A numeric $l_x \times m$ `matrix` (where l_x =`length(x)`) of scaled $\psi^{(k)}(x)$ values. The matrix has `attributes`

`underflow`	of l_x integer counts of the number of under- and over-flows, in computing the corresponding i-th matrix column for `x[i]`.
`ierr`	length-l_x integer vector of error codes, where `0` is normal/successful.

Author(s)

Martin Maechler (R interface); R Core et al., see `digamma`.

References

See those in `psigamma`

See Also

digamma, trigamma, psigamma.

Examples

```
x <- seq(-3.5, 6, by=1/4)
dpx <- dpsifn(x, m = if(getRversion() >= "4.2") 7 else 5)
dpx # in R <= 4.2.1, see that sometimes the 'nz' (under-over-flow count) was uninitialized !!
j <- -1L+seq_len(nrow(dpx)); (fj <- (-1)^(j+1)*gamma(j+1))
## mdpsi <- cbind(di =   digamma(x),      -dpx[1,],
##         tri=  trigamma(x),       dpx[2,],
##         tetra=psigamma(x,2),  -2*dpx[3,],
##         penta=psigamma(x,3),   6*dpx[4,],
##         hexa =psigamma(x,4), -24*dpx[5,],
##         hepta=psigamma(x,5), 120*dpx[6,],
##         octa =psigamma(x,6),-720*dpx[7,])
## cbind(x, ie=attr(dpx,"errorCode"), round(mdpsi, 4))
str(psig <- outer(x, j, psigamma))
dpsi <- t(fj * (`attributes<-`(dpx, list(dim=dim(dpx)))))
if(getRversion() >= "4.2") {
      print( all.equal(psig, dpsi, tol=0) )# -> see 1.185e-16
  stopifnot( all.equal(psig, dpsi, tol=1e-15) )
} else { # R <= 4.1.x; dpsifn(x, ..) *not* ok for x < 0
  i <- x >= 0
      print( all.equal(psig[i,], dpsi[i,], tol=0) )# -> see 1.95e-16
  stopifnot( all.equal(psig[i,], dpsi[i,], tol=1e-15) )
}
```

---

dtWV *Asymptotic Noncentral t Distribution Density by Viechtbauer*

Description

Compute the density function $f(x)$ of the t distribution with df degrees of freedom and non-centrality parameter ncp, according to Wolfgang Viechtbauer's proposal in 2002. This is an asymptotic formula for "large" df$= \nu$, or mathematically $\nu \to \infty$.

Usage

```
dtWV(x, df, ncp = 0, log = FALSE)
```

Arguments

x	numeric vector.
df	degrees of freedom (> 0, maybe non-integer). df = Inf is allowed.
ncp	non-centrality parameter δ; If omitted, use the central t distribution.
log	logical; if TRUE, $log(f(x))$ is returned instead of $f(x)$.

Details

The formula used is "asymptotic": Resnikoff and Lieberman (1957), p.1 and p.25ff, proposed to use recursive polynomials for (*integer !*) degrees of freedom $f = 1, 2, \ldots, 20$, and then, for df$= f > 20$, use the asymptotic approximation which Wolfgang Viechtbauer proposed as a first version of a non-central t density for R (when `dt()` did not yet have an `ncp` argument).

Value

numeric vector of density values, properly recycled in `(x, df, ncp)`.

Author(s)

Wolfgang Viechtbauer (2002) post to R-help (https://stat.ethz.ch/pipermail/r-help/2002-October/026044.html), and Martin Maechler (`log` argument; tweaks, notably recycling).

References

Resnikoff, George J. and Lieberman, Gerald J. (1957) *Tables of the non-central t-distribution*; Technical report no. 32 (LIE ONR 32), April 1, 1957; Applied Math. and Stat. Lab., Stanford University. https://statistics.stanford.edu/technical-reports/tables-non-central-t-distribution-density-functio

See Also

`dt`, R's (C level) implementation of the (non-central) t density; `dntJKBf`, for Johnson et al.'s summation formula approximation.

Examples

```
tt <- seq(0, 10, len = 21)
ncp <- seq(0, 6, len = 31)
dt3R  <- outer(tt, ncp, dt  , df = 3)
dt3WV <- outer(tt, ncp, dtWV, df = 3)
all.equal(dt3R, dt3WV) # rel.err 0.00063
dt25R  <- outer(tt, ncp, dt  , df = 25)
dt25WV <- outer(tt, ncp, dtWV, df = 25)
all.equal(dt25R, dt25WV) # rel.err 1.1e-5

x <- -10:700
fx  <- dt  (x, df = 22, ncp =100)
lfx <- dt  (x, df = 22, ncp =100, log=TRUE)
lfV <- dtWV(x, df = 22, ncp =100, log=TRUE)

head(lfx, 20) # shows that R's dt(*, log=TRUE) implementation is "quite suboptimal"

## graphics
opa <- par(no.readonly=TRUE)
par(mar=.1+c(5,4,4,3), mgp = c(2, .8,0))
plot(fx ~ x, type="l")
par(new=TRUE) ; cc <- c("red", adjustcolor("orange", 0.4))
plot(lfx ~ x, type = "o", pch=".", col=cc[1], cex=2, ann=FALSE, yaxt="n")
sfsmisc::eaxis(4, col=cc[1], col.axis=cc[1], small.args = list(col=cc[1]))
```

```
lines(x, lfV, col=cc[2], lwd=3)
dtt1 <- "      dt"; dtt2 <- "(x, df=22, ncp=100"; dttL <- paste0(dtt2,", log=TRUE)")
legend("right", c(paste0(dtt1,dtt2,")"), paste0(c(dtt1,"dtWV"), dttL)),
       lty=1, lwd=c(1,1,3), col=c("black", cc), bty = "n")
par(opa) # reset
```

`expm1x` *Accurate exp(x) - 1 - x (for smallish |x|)*

Description

Compute $e^x - 1 - x =$ `exp(x) - 1 - x` accurately, notably for small $|x|$.

The last two entries in `cutx[]` denote boundaries where `expm1x(x)` uses direct formulas. For `nC <- length(cutx)`, `exp(x) - 1 - x` is used for `abs(x) >= cutx[nC]`, and when `abs(x) < cutx[nC]` `expm1(x) - x` is used for `abs(x) >= cutx[nC-1]`.

Usage

```
expm1x(x, cutx = c( 4.4e-8, 0.1, 0.385, 1.1, 2),
             k = c(2,      9,  12,    17))

expm1xTser(x, k)
```

Arguments

`x`	numeric-alike vector; goal is to work for `mpfr`-numbers too.
`cutx`	increasing positive numeric vector of cut points defining intervals in which the computations will differ.
`k`	for `exp1mx()`: increasing vector of integers with `length(k) == length(cutx) + 2`, denoting the order of Taylor polynomial approximation by `expm1xTser(.,k)` to `expm1x(.)`. `exp1mxTser()`: an integer ≥ 1, where the Taylor polynomial approximation has degree $k + 1$.

Value

a vector like `x` containing (approximations to) $e^x - x - 1$.

Author(s)

Martin Maechler

See Also

`expm1(x)` for computing $e^x - 1$ is much more widely known, and part of the ISO C standards now.

Examples

```
## a symmetric set of negative and positive
x <- unique(c(2^-seq(-3/8, 54, by = 1/8), seq(7/8, 3, by = 1/128)))
x <- x0 <- sort(c(-x, 0, x)) # negative *and* positive

## Mathematically,  expm1x() = exp(x) - 1 - x  >= 0  (and == 0 only at x=0):
em1x <- expm1x(x)
stopifnot(em1x >= 0, identical(x == 0, em1x == 0))

plot (x, em1x, type='b', log="y")
lines(x, expm1(x)-x, col = adjustcolor(2, 1/2), lwd = 3) ## should nicely cover ..
lines(x, exp(x)-1-x, col = adjustcolor(4, 1/4), lwd = 5) ## should nicely cover ..
cuts <- c(4.4e-8, 0.10, 0.385, 1.1, 2)[-1] # *not* drawing 4.4e-8
v <- c(-rev(cuts), 0, cuts); stopifnot(!is.unsorted(v))
abline(v = v, lty = 3, col=adjustcolor("gray20", 1/2))

stopifnot(diff(em1x[x <= 0]) <= 0)
stopifnot(diff(em1x[x >= 0]) >= 0)

## direct formula - may be really "bad" :
expm1x.0 <- function(x) exp(x) -1 - x
## less direct formula - improved (but still not universally ok):
expm1x.1 <- function(x) expm1(x)  - x

ax <- abs(x) # ==> show negative and positive x on top of each other
plot (ax, em1x, type='l', log="xy", xlab = "|x|  (for negative and positive x)")
lines(ax, expm1(x)-x, col = adjustcolor(2, 1/2), lwd = 3) ## see problem at very left
lines(ax, exp(x)-1-x, col = adjustcolor(4, 1/4), lwd = 5) ## see huge problems for |x| < ~10^{-7}
legend("topleft", c("expm1x(x)", "expm1(x) - x", "exp(x) - 1 - x"), bty="n",
       col = c(1,2,4), lwd = c(1,3,5))

## -------------------- Relative error of Taylor series approximations :
twoP <- seq(-0.75, 54, by = 1/8)
x <- 2^-twoP
x <- sort(c(-x,x)) # negative *and* positive
e1xAll <- cbind(expm1x.0 = expm1x.0(x),
                expm1x.1 = expm1x.1(x),
                vapply(1:15, \(k) expm1xTser(x, k=k), x))
colnames(e1xAll)[-(1:2)] <- paste0("k=",1:15)
head(e1xAll)
## TODO  plot !!
```

`format01prec` *Format Numbers in [0,1] with "Precise" Result*

Description

Format numbers in [0,1] with "precise" result, notably using `"1-..."` if needed.

Usage

```
format01prec(x, digits = getOption("digits"), width = digits + 2,
             eps = 1e-06, ...,
             FUN = function(x, ...) formatC(x, flag = "-", ...))
```

Arguments

x	numbers in [0,1]; (still works if not)
digits	number of digits to use; is used as FUN(*, digits = digits) or FUN(*, digits = digits - 5) depending on x or eps.
width	desired width (of strings in characters), is used as FUN(*, width = width) or FUN(*, width = width - 2) depending on x or eps.
eps	small positive number: Use '1-' for those x which are in $(1 - eps, 1]$. The author has claimed in the last millennium that (the default) 1e-6 is *optimal*.
...	optional further arguments passed to FUN(x, digits, width, ...).
FUN	a function used for format()ing; must accept both a digits and width argument.

Value

a character vector of the same length as x.

Author(s)

Martin Maechler, 14 May 1997

See Also

formatC, format.pval.

Examples

```
## Show that format01prec()  does reveal more precision :
cbind(format      (1 - 2^-(16:24)),
      format01prec(1 - 2^-(16:24)))

## a bit more variety
e <- c(2^seq(-24,0, by=2), 10^-(7:1))
ee <- sort(unique(c(e, 1-e)))
noquote(ff <- format01prec(ee))
data.frame(ee, format01prec = ff)
```

fr_ld_exp *Base-2 Representation and Multiplication of Numbers*

Description

Both are R versions of C99 (and POSIX) standard C (and C++) mathlib functions of the same name.

`frexp(x)` computes base-2 exponent `e` and "mantissa", or *fraction* `r`, such that $x = r * 2^e$, where $r \in [0.5, 1)$ (unless when `x` is in `c(0, -Inf, Inf, NaN)` where `r == x` and `e` is 0), and e is integer valued.

`ldexp(f, E)` is the *inverse* of `frexp()`: Given fraction or mantissa `f` and integer exponent `E`, it returns $x = f * 2^E$. Viewed differently, it's the fastest way to multiply or divide (double precision) numbers with 2^E.

Usage

```
frexp(x)
ldexp(f, E)
```

Arguments

`x`	numeric (coerced to `double`) vector.
`f`	numeric fraction (vector), in $[0.5, 1)$.
`E`	integer valued, exponent of `2`, i.e., typically in `(-1024-50):1024`, otherwise the result will underflow to 0 or overflow to `+/- Inf`.

Value

`frexp` returns a `list` with named components `r` (of type `double`) and `e` (of type `integer`).

Author(s)

Martin Maechler

References

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

See Also

Vaguely relatedly, `log1mexp()`, `lsum`, `logspace.add`.

Examples

```
set.seed(47)
x <- c(0, 2^(-3:3), (-1:1)/0,
       rlnorm(2^12, 10, 20) * sample(c(-1,1), 512, replace=TRUE))
head(x, 12)
which(!(iF <- is.finite(x))) # 9 10 11
rF <- frexp(x)
sapply(rF, summary) # (nice only when x had no NA's ..)
data.frame(x=x[!iF], lapply(rF, `[`, !iF))
##  by C.99/POSIX  'r' should be the same as 'x'  for these,
##      x    r e
## 1 -Inf -Inf 0
## 2  NaN  NaN 0
## 3  Inf  Inf 0
## but on Windows, we've seen  3 NA's :
ar <- abs(rF$r)
ldx <- with(rF, ldexp(r, e))
r0 <- numeric(0L)
stopifnot(exprs = {
  0.5 <= ar[iF & x != 0]
  ar[iF] < 1
  is.integer(rF$e)
  all.equal(x[iF], ldx[iF], tol= 4*.Machine$double.eps)
  ## but actually, they should even be identical, well at least when finite
  identical(x[iF], ldx[iF])
  identical(r0, ldexp(r0, 1)) # (0-length o non-0-l.) did seg.fault
})
```

---

gam1d *Compute 1/Gamma(x+1) - 1 Accurately*

Description

Computes $1/\Gamma(a+1) - 1$ accurately in $[-0.5, 1.5]$ for numeric argument a; For "mpfr" numbers, the precision is increased intermediately such that $a + 1$ should not lose precision.

FIXME: "Pure-R" implementation is in ‘ ~/R/Pkgs/DPQ/TODO_R_versions_gam1_etc.R ’

Usage

```
gam1 (a, useTOMS = is.numeric(a), warnIf=TRUE, verbose=FALSE)
gam1d(a, warnIf = TRUE, verbose = FALSE)
```

Arguments

a	a numeric or numeric-alike, typically inheriting from class "mpfr".
useTOMS	logical indicating if the TOMS 708 C code should be used; in that case a must be (C-level coercible to) numeric.

warnIf — logical indicating if a warning should be signalled when a is not in the "proper" range $[-0.5, 1.5]$.

verbose — logical indicating if some output from C code execution should be printed to the console.

Details

https://dlmf.nist.gov/ states the well-know Taylor series for

$$\frac{1}{\Gamma(z)} = \sum_{k=1}^{\infty} c_k z^k$$

with $c_1 = 1$, $c_2 = \gamma$, (Euler's gamma, $\gamma = 0.5772...$, with recursion $c_k = (\gamma c_{k-1} - \zeta(2)c_{k-2}... + (-1)^k \zeta(k-1)c_1)/(k-1)$.

Hence,

$$\frac{1}{\Gamma(z+1)} = z + 1 + \sum_{k=2}^{\infty} c_k (z+1)^k$$

$$\frac{1}{\Gamma(z+1)} - 1 = z + \gamma * (z+1)^2 + \sum_{k=3}^{\infty} c_k (z+1)^k$$

Consequently, for $\zeta_k := \zeta(k)$, $c_3 = (\gamma^2 - \zeta_2)/2$, $c_4 = \gamma^3/6 - \gamma\zeta_2/2 + \zeta_3/3$.

```
gam <- Const("gamma", 128)
z <- Rmpfr::zeta(mpfr(1:7, 128))
(c3 <- (gam^2 -z[2])/2)                    # -0.65587807152025388107701951514 5
(c4 <- (gam*c3 - z[2]*c2 + z[3])/3)         # -0.042002635034095235529003934888
(c4 <- gam*(gam^2/6 - z[2]/2) + z[3]/3)
(c5 <- (gam*c4 - z[2]*c3 + z[3]*c2 - z[4])/4) # 0.16653861138229148950170079 51
(c5 <- (gam^4/6 - gam^2*z[2] + z[2]^2/2 + gam*z[3]*4/3 - z[4])/4)
```

Value

a numeric-alike vector like a.

Author(s)

Martin Maechler building on C code of TOMS 708

References

TOMS 708, see pbeta, where gam1() is the C function name.

See Also

gamma; package **DPQmpfr**'s gam1M.

Examples

```
g1 <- function(u) 1/gamma(u+1) - 1
u <- seq(-.5, 1.5, by=1/16); set.seed(1); u <- sample(u) # permuted (to check logic)

g11   <- vapply(u, gam1d, 1)
gam1d. <- gam1d(u)
stopifnot( all.equal(g1(u), g11) )
stopifnot( identical(g11, gam1d.) )

## Comparison of g1() and gam1d(), slightly extending the [-.5, 1.5] interval:
u <- seq(-0.525, 1.525, length.out = 2001)
mg1 <- cbind(g1 = g1(u), gam1d = gam1d(u))
clr <- adjustcolor(1:2, 1/2)
matplot(u, mg1, type = "l", lty = 1, lwd=1:2, col=clr) # *no* visual difference
## now look at *relative* errors
relErrV <- sfsmisc::relErrV
relE <- relErrV(mg1[,"gam1d"], mg1[,"g1"])
plot(u, relE, type = "l")
plot(u, abs(relE), type = "l", log = "y",
     main = "|rel.diff|  gam1d() vs 'direct' 1/gamma(u+1) - 1")

## now {Rmpfr} for "truth" :
if(requireNamespace("Rmpfr")) withAutoprint({
    asN  <- Rmpfr::asNumeric; mpfr <- Rmpfr::mpfr
    gam1M <- g1(mpfr(u, 512)) # "cheap": high precision avoiding "all" cancellation
    relE <- asN(relErrV(gam1M, gam1d(u)))
    plot(relE ~ u, type="l", ylim = c(-1,1) * 2.5e-15,
        main = expression("Relative Error of " ~~ gam1d(u) %~~% frac(1, Gamma(u+1)) - 1))
    grid(lty = 3); abline(v = c(-.5, 1.5), col = adjustcolor(4, 1/2), lty=2, lwd=2)
})


## FIXME(maybe) : original plan was a  gam1()  in DPQmpfr, but it is now here in DPQ
if(requireNamespace("Rmpfr")) {
    ## Comparison using Rmpfr; slightly extending the [-.5, 1.5] interval:
    ## relErrV(), mpfr(), asN() defined above (!)
    u <- seq(-0.525, 1.525, length.out = 2001)
    gam1M <- gam1(mpfr(u, 128))
    relE <- asN(relErrV(gam1M, gam1d(u)))

    plot(relE ~ u, type="l", ylim = c(-1,1) * 2.5e-15,
         main = expression("Relative Error of " ~~ gam1d(u) == frac(1, Gamma(u+1)) - 1))
    grid(lty = 3); abline(v = c(-.5, 1.5), col = adjustcolor(4, 1/2), lty=2, lwd=2)

    ## what about the direct formula -- how bad is it really ?
    relED <- asN(relErrV(gam1M, g1(u)))

    plot(relE ~ u, type="l", ylim = c(-1,1) * 1e-14,
         main = expression("Relative Error of " ~~ gam1d(u) == frac(1, Gamma(u+1)) - 1))
    lines(relED ~ u, col = adjustcolor(2, 1/2), lwd = 2)
    ## mtext("comparing with direct formula   1/gamma(u+1) - 1")
    legend("top", c("gam1d(u)", "1/gamma(u+1) - 1"), col = 1:2, lwd=1:2, bty="n")
```

```
    ## direct is clearly *worse* , but not catastrophical

} # if {Rmpfr}
```

---

gamln1 *Compute log(Gamma(x+1)) Accurately in [-0.2, 1.25]*

Description

Computes $\log \Gamma(a+1)$ accurately notably when $|a| \ll 1$. Specifically, in the `userTOMS` case, it uses high (double precision) accuracy rational approximations for $-0.2 \le a \le 1.25$.

Usage

```
gamln1 (a, useTOMS = is.numeric(a), warnIf = TRUE)
gamln1.(a, warnIf = TRUE)
```

Arguments

`a`	a numeric or numeric-alike, typically inheriting from `class "mpfr"`.
`useTOMS`	`logical` indicating if the TOMS 708 C code should be used; in that case `a` must be (C-level coercible to) numeric.
`warnIf`	logical if a `warning` should be signalled when `a` is not in the "proper" range $[-0.2, 1.25]$.

Details

It uses $-a * p(a)/q(a)$ for $a < 0.6$, where p and q are polynomials of degree 6 with coefficient vectors $p = [p_0 p_1 \dots p_6]$ and q,

```
p <- c( .577215664901533, .844203922187225, -.168860593646662,
-.780427615533591, -.402055799310489, -.0673562214325671,
-.00271935708322958)
q <- c( 1, 2.88743195473681, 3.12755088914843, 1.56875193295039,
  .361951990101499, .0325038868253937, 6.67465618796164e-4)
```

Similarly, for $a \ge 0.6$, $x := a - 1$, the result is $x * r(x)/s(x)$, with 5th degree polynomials $r()$ and $s()$ and coefficient vectors

```
r <- c(.422784335098467, .848044614534529, .565221050691933,
       .156513060486551, .017050248402265, 4.97958207639485e-4)
s <- c( 1 , 1.24313399877507, .548042109832463,
       .10155218743983, .00713309612391, 1.16165475989616e-4)
```

Value

a numeric-alike vector like a.

Author(s)

Martin Maechler building on C code of TOMS 708

References

TOMS 708, see pbeta

See Also

lgamma1p() for different algorithms to compute $\log\Gamma(a+1)$, notably when outside the interval $[-0.2, 1.35]$. Package **DPQmpfr**'s lgamma1pM() provides very precise such computations. lgamma() (and gamma() (same page)).

Examples

```
lg1 <- function(u) lgamma(u+1) # the simple direct form
## The curve, zeros at  u=0 & u=1:
curve(lg1, -.2, 1.25, col=2, lwd=2, n=999)
title("lgamma(x + 1)"); abline(h=0, v=0:1, lty=3)

u <- (-16:100)/80 ; set.seed(1); u <- sample(u) # permuted (to check logic)
g11   <- vapply(u, gamln1, numeric(1))
gamln1. <- gamln1(u)
stopifnot( identical(g11, gamln1.) )
stopifnot( all.equal(lg1(u), g11) )

u <- (-160:1000)/800
relE <- sfsmisc::relErrV(gamln1(u), lg1(u))
plot(u, relE, type="l", main = expression("rel.diff." ~~ gamln1(u) %~~% lgamma(u+1)))
plot(u, abs(relE), type="l", log="y", yaxt="n",
     main = expression("|rel.diff.|" ~~ gamln1(u) %~~% lgamma(u+1)))
sfsmisc::eaxis(2)


if(requireNamespace("DPQmpfr")) withAutoprint({
  ## Comparison using Rmpfr; extending the [-.2, 1.25] interval a bit
  u <- seq(-0.225, 1.31, length.out = 2000)
  lg1pM <- DPQmpfr::lgamma1pM(Rmpfr::mpfr(u, 128))
  relE <- Rmpfr::asNumeric(sfsmisc::relErrV(lg1pM, gamln1(u, warnIf=FALSE)))

  plot(relE ~ u, type="l", ylim = c(-1,1) * 2.3e-15,
       main = expression("relative error of " ~~ gamln1(u) == log( Gamma(u+1) )))
  grid(lty = 3); abline(v = c(-.2, 1.25), col = adjustcolor(4, 1/2), lty=2, lwd=2)
  ## well... TOMS 708 gamln1() is good (if "only" 14 digits required

  ## what about the direct formula -- how bad is it really ?
  relED <- Rmpfr::asNumeric(sfsmisc::relErrV(lg1pM, lg1(u)))
  lines(relED ~ u, col = adjustcolor(2, 1/2))
```

```
 ## amazingly, the direct formula is partly (around -0.2 and +0.4) even better than gamln1() !

 plot(abs(relE) ~ u, type="l", log = "y", ylim = c(7e-17, 1e-14),
      main = expression("|relative error| of " ~~ gamln1(u) == log( Gamma(u+1) )))
 grid(lty = 3); abline(v = c(-.2, 1.25), col = adjustcolor(4, 1/2), lty=2, lwd=2)
 relED <- Rmpfr::asNumeric(sfsmisc::relErrV(lg1pM, lg1(u)))
 lines(abs(relED) ~ u, col = adjustcolor(2, 1/2))
})
```

`gammaVer` *Gamma Function Versions*

Description

Provide different variants or versions of computing the Gamma (Γ) function.

Usage

```
gammaVer(x, version, stirlerrV = c("R3", "R4..1", "R4.4_0"), traceLev = 0L)
```

Arguments

`x`	numeric vector of absissa value for the Gamma function.
`version`	integer in {1,2,..,5} specifying which variant is desired.
`stirlerrV`	a string, specifying the `stirlerr()` version/variant to use.
`traceLev`	non-negative integer indicating the amount of diagnostic "tracing" output to the console during computation.

Details

All of these are good algorithms to compute $\Gamma(x)$ (for real x), and indeed correspond to the versions R's implementation of `gamma(x)` over time. More specifically, the current version numbers correspond to

1. . TODO
2. .
3. .
4. Used in R from ... up to versions 4.3.z
5. Possibly to be used in R 4.4.z and newer.

The `stirlerrV` must be a string specifying the version of `stirlerr()` to be used:

`"R3"`: the historical version, used in all R version up to R 4.3.z.

`"R4..1"`: only started using `lgamma1p(n)` instead of `lgamma(n + 1.)` in `stirlerr(n)` for $n \leq 15$, in the direct formula.

`"R4.4_0"`: uses 10 cutoffs instead 4, and these are larger to gain accuracy.

Value

numeric vector as x

Author(s)

Martin Maechler

References

.... TODO ....

See Also

gamma(), R's own Gamma function.

Examples

```
xx <- seq(-4, 10, by=1/2)
gx <- sapply(1:5, gammaVer, x=xx)
gamx <- gamma(xx)
cbind(xx, gx, gamma=gamx)
apply(gx, 2, all.equal, target=gamx, tol = 0) # typically: {T,T,T,T, 1.357e-16}
stopifnot( apply(gx, 2, all.equal, target = gamx, tol = 1e-14))
                                                    # even 2e-16 (Lnx, 64b, R 4.2.1)
```

---

hyper2binomP *Transform Hypergeometric Distribution Parameters to Binomial Probability*

Description

Transform the three parameters of the hypergeometric distribution function to the probability parameter of the "corresponding" binomial distribution.

Usage

```
hyper2binomP(x, m, n, k)
```

Arguments

x, m, n, k	see dhyper.

Value

a number, the binomial probability.

References

See those in phyperBinMolenaar.

See Also

phyper, pbinom.

dhyperBinMolenaar(), phyperBinMolenaar.1(), *.2(), etc, all of which are crucially based on hyper2binomP().

Examples

```
hyper2binomP(3,4,5,6) # 0.38856

## The function is simply defined as
function (x, m, n, k) {
    N <- m + n
    p <- m/N
    N.n <- N - (k - 1)/2
    (m - x/2)/N.n - k * (x - k * p - 1/2)/(6 * N.n^2)
 }
```

Ixpq *Normalized Incomplete Beta Function "Like"* pbeta()

Description

Computes the normalized incomplete beta function, in pure R code, derived from Nico Temme's Maple code for computing Table 1 in Gil et al (2023).

It uses a continued fraction, similarly to bfrac() in the TOMS 708 algorithm underlying R's pbeta().

Usage

```
Ixpq(x, l_x, p, q, tol = 3e-16, it.max = 100L, plotIt = FALSE)
```

Arguments

x	numeric
l_x	1 - x; may be specified with higher precision (e.g., when $x \approx 1$, $1 - x$ suffers from cancellation).
p, q	the two shape parameters of the beta distribution.
tol	positive number, the convergence tolerance for the continued fraction computation.
it.max	maximal number of continued fraction steps.
plotIt	a logical, if true, plots show the relative approximation errors in each step.

Value

a vector like x or l_x with corresponding pbeta(x, *) values.

Author(s)

Martin Maechler; based on original Maple code by Nico Temme.

References

Gil et al. (2023)

See Also

pbeta, pbetaRv1(), ..

Examples

```
x <- seq(0, 1, by=1/16)
r <- Ixpq(x, 1-x, p = 4, q = 7, plotIt = TRUE)
cbind(x, r)
## and "test" ___FIXME__
```

---

lbeta *(Log) Beta and Ratio of Gammas Approximations*

Description

Compute log(beta(a,b)) in a simple (fast) or asymptotic way. The asymptotic case is based on the asymptotic Γ (gamma) ratios, provided in Qab_terms() and logQab_asy().

lbeta_asy(a,b, ..) is simply lgamma(a) - logQab_asy(a, b, ..).

Usage

```
lbetaM   (a, b, k.max = 5, give.all = FALSE)
lbeta_asy(a, b, k.max = 5, give.all = FALSE)
lbetaMM  (a, b, cutAsy = 1e-2, verbose = FALSE)

 betaI(a, n)
lbetaI(a, n)

logQab_asy(a, b, k.max = 5, give.all = FALSE)
Qab_terms(a, k)
```

Arguments

a, b, n	the Beta parameters, see beta; n must be a positive integer and "small".
k.max, k	for lbeta*() and logQab_asy(): the number of terms to be used in the series expansion of Qab_terms(), currently must be in $0, 1, .., 5$.
give.all	logical indicating if all terms should be returned (as columns of a matrix) or just the result.

cutAsy	cutoff value from where to switch to asymptotic formula.
verbose	logical (or integer) indicating if and how much monitoring information should be printed to the console.

Details

All `lbeta*()` functions compute `log(beta(a,b))`.

We use $Qab = Qab(a, b)$ for

$$Q_{a,b} := \frac{\Gamma(a+b)}{\Gamma(b)},$$

which is numerically challenging when b becomes large compared to `a`, or $a \ll b$.

With the beta function

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} = \frac{\Gamma(a)}{Qab},$$

and hence

$$\log B(a,b) = \log\Gamma(a) + \log\Gamma(b) - \log\Gamma(a+b) = \log\Gamma(a) - \log Qab,$$

or in R, `lbeta(a,b) := lgamma(a) - logQab(a,b)`.

Indeed, typically everything has to be computed in log scale, as both $\Gamma(b)$ and $\Gamma(a+b)$ would overflow numerically for large b. Consequently, we use `logQab*()`, and for the large b case `logQab_asy()` specifically,

$$\texttt{logQab(a,b)} := \log(Qab(a,b)).$$

The 5 polynomial terms in `Qab_terms()` have been derived by the author in 1997, but not published, about getting asymptotic formula for Γ ratios, related to but *different* than formula (6.1.47) in Abramowitz and Stegun.

We also have a vignette about this, but really the problem has been adressed pragmatically by the authors of TOMS 708, see the 'References' in `pbeta`, by their routine `algdiv()` which also is available in our package **DPQ**, $\texttt{algdiv}(a,b) = -\texttt{logQab}(a,b)$. Note that this is related to computing `qbeta()` in boundary cases. See also `algdiv()` 'Details'.

Value

a fast or simple (approximate) computation of `lbeta(a,b)`.

Author(s)

Martin Maechler

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain; Formula (6.1.47), p.257

See Also

R's `beta` function; `algdiv()`.

Examples

```
(r  <- logQab_asy(1, 50))
(rF <- logQab_asy(1, 50, give.all=TRUE))
r == rF # all TRUE:  here, even the first approx. is good!
(r2  <- logQab_asy(5/4, 50))
(r2F <- logQab_asy(5/4, 50, give.all=TRUE))
r2 == r2F # TRUE only first entry "5"
(r2F.3 <- logQab_asy(5/4, 50, k=3, give.all=TRUE))

## Check relation to Beta(), Gamma() functions:
a <- 1.1 * 2^(-6:4)
b <- 1001.5
rDlgg <- lgamma(a+b) - lgamma(b) # suffers from cancellation for small 'a'
rDlgb <- lgamma(a) - lbeta(a, b) #    (ditto)
ralgd <- - algdiv(a,b)
rQasy <- logQab_asy(a, b)
cbind(a, rDlgg, rDlgb, ralgd, rQasy)
all.equal(rDlgg, rDlgb, tolerance = 0) # 3.0e-14
all.equal(rDlgb, ralgd, tolerance = 0) # 1.2e-16
all.equal(ralgd, rQasy, tolerance = 0) # 4.1e-10
all.equal(rQasy, rDlgg, tolerance = 0) # 3.5e-10

stopifnot(exprs = {
    all.equal(rDlgg, rDlgb, tolerance = 1e-12) # 3e-14 {from cancellations!}
    all.equal(rDlgb, ralgd, tolerance = 1e-13) # 1e-16
    all.equal(ralgd, rQasy, tolerance = 2e-9) # 4.1e-10
    all.equal(rQasy, rDlgg, tolerance = 2e-9) # 3.5e-10
    all.equal(lgamma(a)-lbeta(a, 2*b), logQab_asy(a, 2*b), tolerance =1e-10)# 1.4e-11
    all.equal(lgamma(a)-lbeta(a, b/2), logQab_asy(a, b/2), tolerance = 1e-7)# 1.2e-8
})
if(requireNamespace("Rmpfr")) withAutoprint({
  aM <- Rmpfr::mpfr(a, 512)
  bM <- Rmpfr::mpfr(b, 512)
  rT <- lgamma(aM+bM) - lgamma(bM) # "True" i.e. accurate values
  relE <- Rmpfr::asNumeric(sfsmisc::relErrV(rT, cbind(rDlgg, rDlgb, ralgd, rQasy)))
  cbind(a, signif(relE,4))
  ##          a      rDlgg      rDlgb      ralgd      rQasy
  ##  0.0171875  4.802e-12  3.921e-16  4.145e-17 -4.260e-16
  ##  0.0343750  1.658e-12  1.509e-15 -1.011e-17  1.068e-16
  ##  0.0687500 -2.555e-13  6.853e-16 -1.596e-17 -1.328e-16
  ##  0.1375000  1.916e-12 -7.782e-17  3.905e-17 -7.782e-17
  ##  0.2750000  1.246e-14  7.001e-17  7.001e-17 -4.686e-17
  ##  0.5500000 -2.313e-13  5.647e-17  5.647e-17 -6.040e-17
  ##  1.1000000 -9.140e-14 -1.298e-16 -1.297e-17 -1.297e-17
  ##  2.2000000  9.912e-14  2.420e-17  2.420e-17 -9.265e-17
  ##  4.4000000  1.888e-14  6.810e-17 -4.873e-17 -4.873e-17
  ##  8.8000000 -7.491e-15  1.004e-16 -1.638e-17 -4.118e-13
  ## 17.6000000  2.222e-15  1.207e-16  3.974e-18 -6.972e-10

## ==>  logQab_asy() is very good _here_ as long as  a << b
})
```

`lfastchoose` *R versions of Simple Formulas for Logarithmic Binomial Coefficients*

Description

Provide R versions of simple formulas for computing the logarithm of (the absolute value of) binomial coefficients, i.e., simpler, more direct formulas than what (the C level) code of R's `lchoose()` computes.

Usage

```
lfastchoose(n, k)
 f05lchoose(n, k)
```

Arguments

n	a numeric vector.
k	a integer valued numeric vector.

Value

a numeric vector with the same attributes as n + k.

Author(s)

Martin Maechler

See Also

`lchoose`.

Examples

```
lfastchoose # function(n, k) lgamma(n + 1) - lgamma(k + 1) - lgamma(n - k + 1)
f05lchoose  # function(n, k) lfastchoose(n = floor(n + 0.5), k = floor(k + 0.5))

## interesting cases ?
```

lgamma1p *Accurate* log(gamma(a+1))

Description

Compute

$$l\Gamma_1(a) := \log\Gamma(a+1) = \log(a \cdot \Gamma(a)) = \log a + \log\Gamma(a),$$

which is "in principle" the same as log(gamma(a+1)) or lgamma(a+1), accurately also for (very) small a $(0 < a < 0.5)$.

Usage

```
lgamma1p (a, tol_logcf = 1e-14, f.tol = 1, ...)
lgamma1p.(a, cutoff.a = 1e-6, k = 3)
lgamma1p_series(x, k)
lgamma1pC(x)
```

Arguments

a, x	a numeric vector.
tol_logcf	for lgamma1p(): a non-negative number passed to logcf() (and log1pmx() which calls logcf()).
f.tol	numeric (**f**actor) used in log1pmx(*, tol_logcf = f.tol * tol_logcf).
...	further optional arguments passed on to log1pmx().
cutoff.a	for lgamma1p.(): a positive number indicating the cutoff to switch from ...
k	an integer, the number of terms in the series expansion used internally; currently for lgamma1p.(): $k \le 3$ lgamma1p_series(): $k \le 15$

Details

lgamma1p() is an R translation of the function (in Fortran) in Didonato and Morris (1992) which uses a 40-degree polynomial approximation.

lgamma1p.(u) for small $|u|$ uses up to 4 terms of

$$\Gamma(1+u) = 1 + u * (-\gamma_E + a_0 u + a_1 u^2 + a_2 u^3) + O(u^5),$$

where $a_0 := (\psi'(1) + \psi(1)^2)/2 = (\pi^2/6 + \gamma_E^2)/2$, and a_1 und a_2 are similarly determined. Then log1p(.) of the $\Gamma(1+u) - 1$ approximation above is used.

lgamma1p_series(x, k) is a Taylor series approximation of order k, directly of $l\Gamma_1(a) := \log\Gamma(a+1)$ (mostly via Maple), which starts as $-\gamma_E x + \pi^2 x^2/12 + \ldots$, where γ_E is Euler's constant 0.5772156649.

lgamma1pC() is an interface to R's C API ('Mathlib' / 'Rmath.h') function lgamma1p().

Value

a numeric vector with the same attributes as a.

Author(s)

Morten Welinder (C code of Jan 2005, see R's bug issue PR#7307) for lgamma1p().

Martin Maechler, notably for lgamma1p_series() which works with package **Rmpfr** but otherwise may be *much* less accurate than Morten's 40 term series!

References

Didonato, A. and Morris, A., Jr, (1992) Algorithm 708: Significant digit computation of the incomplete beta function ratios. *ACM Transactions on Mathematical Software*, **18**, 360–373; see also pbeta.

See Also

Yet another algorithm, fully double precision accurate in $[-0.2, 1.25]$, is provided by gamln1().

log1pmx, log1p, pbeta.

Examples

```
curve(lgamma1p, -1.25, 5, n=1001, col=2, lwd=2)
abline(h=0, v=-1:0, lty=c(2,3,2), lwd=c(1, 1/2,1))
for(k in 1:15)
 curve(lgamma1p_series(x, k=k), add=TRUE, col=adjustcolor(paste0("gray",25+k*4), 2/3), lty = 3)

curve(lgamma1p, -0.25, 1.25, n=1001, col=2, lwd=2)
abline(h=0, v=0, lty=2)
for(k in 1:15)
  curve(lgamma1p_series(x, k=k), add=TRUE, col=adjustcolor("gray20", 2/3), lty = 3)

curve(-log(x*gamma(x)), 1e-30, .8, log="xy", col="gray50", lwd = 3,
      axes = FALSE, ylim = c(1e-30,1)) # underflows to zero at x ~= 1e-16
eaxGrid <- function(at.x = 10^(1-4*(0:8)), at.y = at.x) {
    sfsmisc::eaxis(1, sub10 = c(-2, 2), nintLog=16)
    sfsmisc::eaxis(2, sub10 = 2, nintLog=16)
    abline(h = at.y, v = at.x, col = "lightgray", lty = "dotted")
}
eaxGrid()
curve(-lgamma( 1+x), add=TRUE, col="red2", lwd=1/2)# underflows even earlier
curve(-lgamma1p (x), add=TRUE, col="blue") -> lgxy
curve(-lgamma1p.(x), add=TRUE, col=adjustcolor("forest green",1/4),
      lwd = 5, lty = 2)
for(k in 1:15)
  curve(-lgamma1p_series(x, k=k), add=TRUE, col=paste0("gray",80-k*4), lty = 3)
stopifnot(with(lgxy, all.equal(y, -lgamma1pC(x))))

if(requireNamespace("Rmpfr")) { # accuracy comparisons, originally from ../tests/qgamma-ex.R
    x <- 2^(-(500:11)/8)
    x. <- Rmpfr::mpfr(x, 200)
```

```
## versions of lgamma1p(x) := lgamma(1+x)
## lgamma1p(x) = log gamma(x+1) = log (x * gamma(x)) = log(x) + lgamma(x)
xct. <- log(x.  * gamma(x.)) # using  MPFR  arithmetic .. no overflow/underflow ...
xc2. <- log(x.) + lgamma(x.) #  (ditto)

AllEq <- function(target, current, ...)
    Rmpfr::all.equal(target, current, ...,
                     formatFUN = function(x, ...) Rmpfr::format(x, digits = 9))
print(AllEq(xct., xc2., tol = 0)) # 2e-57
rr <- vapply(1:15, function(k) lgamma1p_series(x, k=k), x)
colnames(rr) <- paste0("k=",1:15)
relEr <- Rmpfr::asNumeric(sfsmisc::relErrV(xct., rr))
## rel.error of direct simple computation:
relE.D <- Rmpfr::asNumeric(sfsmisc::relErrV(xct., lgamma(1+x)))

matplot(x, abs(relEr), log="xy", type="l", axes = FALSE,
        main = "|rel.Err(.)| for lgamma(1+x) =~= lgamma1p_series(x, k = 1:15)")
eaxGrid()
p2 <- -(53:52); twp <- 2^p2; labL <- lapply(p2, function(p) substitute(2^E, list(E=p)))
abline(h = twp, lty=3)
axis(4, at=twp, las=2, line=-1, labels=as.expression(labL), col=NA,col.ticks=NA)
legend("topleft", paste("k =", 1:15), ncol=3, col=1:6, lty=1:5, bty="n")
lines(x, abs(relE.D), col = adjustcolor(2, 2/3), lwd=2)
legend("top", "lgamma(1+x)", col=2, lwd=2)

## zoom in:
matplot(x, abs(relEr), log="xy", type="l", axes = FALSE,
        xlim = c(1e-5, 0.1), ylim = c(1e-17, 1e-10),
        main = "|rel.Err(.)| for lgamma(1+x) =~= lgamma1p_series(x, k = 1:15)")
eaxGrid(10^(-5:1), 10^-(17:10))
abline(h = twp, lty=3)
axis(4, at=twp, las=2, line=-1, labels=as.expression(labL), col=NA,col.ticks=NA)
legend("topleft", paste("k =", 1:15), ncol=3, col=1:6, lty=1:5, bty="n")
lines(x, abs(relE.D), col = adjustcolor(2, 2/3), lwd=2)
legend("right", "lgamma(1+x)", col=2, lwd=2)

} # Rmpfr only
```

`lgammaAsymp` *Asymptotic Log Gamma Function*

Description

Compute an n-th order asymptotic approximation to log Gamma function, using Bernoulli numbers `Bern(k)` for k in $1, \ldots, 2n$.

Usage

```
lgammaAsymp(x, n)
```

Arguments

x	numeric vector
n	integer specifying the approximation order.

Value

numeric vector with the same attributes (`length()` etc) as `x`, containing approximate `lgamma(x)` values.

Author(s)

Martin Maechler

See Also

`lgamma`; the n-th Bernoulli number `Bern(n)`, and also *exact* fractions Bernoulli numbers `BernoulliQ()` from package **gmp**.

Examples

```
## The function is currently
lgammaAsymp
```

---

lgammaP11 *Log Gamma(p) for Positive* p *by Pugh's Method (11 Terms)*

Description

Computes $\log(\Gamma(p))$ for $p > 0$, where $\Gamma(p) = \int_0^\infty s^{p-1} e^{-s}\, ds$.

The evaluation hapenss via Pugh's method (approximation with 11 terms), which is a refinement of the Lanczos method (with 6 terms).

This implementation has been used in and for Abergel and Moisan (2020)'s TOMS 1006 algorithm, see the reference and our `dltgammaInc()`. Given the examples below and `example(dltgammaInc)`, their use of "accurate" does not apply to R's "standard" accuracy.

Usage

```
lgammaP11(x)
```

Arguments

x	a numeric vector

Value

a numeric vector "as" `x` (with no attributes, currently).

Author(s)

C source from Remy Abergel and Lionel Moisan, see the reference; original is in '1006.zip' at https://calgo.acm.org/.

Interface to R in **DPQ**: Martin Maechler

References

Rémy Abergel and Lionel Moisan (2020) Algorithm 1006: Fast and accurate evaluation of a generalized incomplete gamma function, *ACM Transactions on Mathematical Software* **46**(1): 1–24. doi:10.1145/3365983

See Also

`dltgammaInc()` partly uses (the underly C code of) `lgammaP11()`; **DPQ**'s `lgamma1p`, R's `lgamma()`.

Examples

```
(r <- lgammaP11(1:20))
stopifnot(all.equal(r, lgamma(1:20)))
summary(rE <- sfsmisc::relErrV(r, lgamma(1:20)))
cbind(x=1:20, r, lgamma(1:20))
## at x=1 and 2 it is *not* fully accurate but gives -4.44e-16  (= - 2*EPS )

if(requireNamespace("Rmpfr")) withAutoprint({
    asN  <- Rmpfr::asNumeric
    mpfr <- Rmpfr::mpfr
    x <- seq(1, 20, by = 1/8)
    lgamM <- lgamma(x = mpfr(x, 128)) # uses Rmpfr::Math(.) -> Rmpfr:::.Math.codes
    lgam  <- lgammaP11(x)
    relErrV <- sfsmisc::relErrV
    relE <- asN(relErrV(target = lgamM, current = lgam))
    summary(relE)
    summary(relE.R <- asN(relErrV(lgamM, base::lgamma(x))))
    plot(x, relE, type = "b", col=2, # not very good: at  x ~= 3, goes up to  1e-14 !
         main = "rel.Error(lgammaP11(x)) vs  x")
    abline(h = 0, col = adjustcolor("gray", 0.75), lty = 2)
    lines(x, relE.R, col = adjustcolor(4, 1/2), lwd=2)
    ## ## ==> R's lgamma() is much more accurate
    legend("topright", expression(lgammaP11(x), lgamma(x)),
        col = c(palette()[2], adjustcolor(4, 1/2)), lwd = c(1, 2), pch = c(1, NA), bty = "n")

    ## |absolute rel.Err| on log-scale:
    cEps <- 2^-52 ; lc <- adjustcolor("gray", 0.75)
   plot(x, abs(relE), type = "b", col=2, log = "y", ylim = cEps * c(2^-4, 2^6), yaxt = "n",
         main = "|rel.Error(lgammaP11(x))| vs  x -- log-scale")
    sfsmisc::eaxis(2, cex.axis = 3/4, col.axis = adjustcolor(1, 3/4))
  abline(h= cEps, col = lc, lty = 2); axis(4, at=cEps, quote(epsilon[C]), las=1, col.axis=lc)
    lines(x, pmax(2^-6*cEps, abs(relE.R)), col = adjustcolor(4, 1/2), lwd=2)
    ## ## ==> R's lgamma() is *much* more accurate
   legend("topright", expression(lgammaP11(x), lgamma(x)), bg = adjustcolor("gray", 1/4),
        col = c(palette()[2], adjustcolor(4, 1/2)), lwd = c(1, 2), pch = c(1, NA), bty = "n")
```

```
    mtext(sfsmisc::shortRversion(), cex = .75, adj = 1)
})
```

`log1mexp` *Compute* log*(1 -* exp*(-a)) and* $\log(1+\exp(x))$ *Numerically Optimally*

Description

Compute f(a) = log(1 - exp(-a)) quickly and numerically accurately.

`log1mexp()` is simple pure R code;
`log1mexpC()` is an interface to R C API ('`Mathlib`' / '`Rmath.h`') function.

`log1pexpC()` is an interface to R's '`Mathlib`' `double` function `log1pexp()` which computes $\log(1+\exp(x))$, accurately, notably for large x, say, $x > 720$.

Usage

```
log1mexp (x)
log1mexpC(x)
log1pexpC(x)
```

Arguments

x	numeric vector of positive values.

Author(s)

Martin Maechler

References

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

See Also

The `log1mexp()` function in CRAN package **copula**, and the corresponding vignette (in the 'References').

Examples

```
l1m.xy <- curve(log1mexp(x), -10, 10, n=1001)
stopifnot(with(l1m.xy, all.equal(y, log1mexpC(x))))

x <- seq(0, 710, length=1+710*2^4); stopifnot(diff(x) == 1/2^4)
l1pm <- cbind(log1p(exp(x)),
              log1pexpC(x))
matplot(x, l1pm, type="l", log="xy") # both look the same
iF <- is.finite(l1pm[,1])
stopifnot(all.equal(l1pm[iF,2], l1pm[iF,1], tol=1e-15))
```

log1pmx *Accurate* log(1+x) - x *Computation*

Description

Compute

$$\log(1 + x) - x$$

accurately also for small x, i.e., $|x| \ll 1$.

Since April 2021, the pure R code version `log1pmx()` also works for "mpfr" numbers (from package **Rmpfr**).

`rlog1(x)`, provided mostly for reference and reproducibility, is used in TOMS Algorithm 708, see e.g. the reference of `lgamma1p`. and computes *minus* `log1pmx(x)`, i.e., $x - \log(1 + x)$, using (argument reduction) and a rational approximation when $x \in [-0.39, 0.57)$.

Usage

```
log1pmx (x, tol_logcf = 1e-14, eps2 = 0.01, minL1 = -0.79149064,
         trace.lcf = FALSE,
         logCF = if(is.numeric(x)) logcf else logcfR)
log1pmxC(x)  # TODO in future: arguments (minL1, eps2, tol_logcf),
             # possibly with *different* defaults (!)
rlog1(x)
```

Arguments

x	numeric (or, for `log1pmx()` only, `"mpfr"` number) vector with values $x > -1$.
tol_logcf	a non-negative number indicating the tolerance (maximal relative error) for the auxiliary `logcf()` function.
eps2	non-negative cutoff ϵ_2 where the algorithm switches from a few terms, to using `logcf()` explicitly. Note that for more accurate mpfr-numbers the default `eps2 = .01` is too large, even more so when the tolerance is lowered (from `1e-14`).
minL1	negative cutoff, called `minLog1Value` in Morten Welinder's C code for `log1pmx()` in 'R/src/nmath/pgamma.c', hard coded there to -0.79149064 which seems not optimal for computation of `log1pmx()`, at least in some cases, and hence **the default may be changed in the future**. Also, for mpfr numbers, the default -0.79149064 may well be far from optimal.
trace.lcf	`logical` used in `logcf(.., trace=trace.lcf)`.
logCF	the `function` to be used as `logcf()`. The default chooses the pure R `logcfR()` when x is not numeric, and chooses the C-based `logcf()` when `is.numeric(x)` is true.

Details

In order to provide full (double precision) accuracy, the computations happens differently in three regions for x,

$$m_l = \texttt{minL1} = -0.79149064$$

is the first cutpoint,

$x < m_l$ **or** $x > 1$**:** use `log1pmx(x) := log1p(x) - x`,

$|x| < \epsilon_2$**:** use $t((((2/9 * y + 2/7)y + 2/5)y + 2/3)y - x)$,

$x \in [ml, 1]$**, and** $|x| \geq \epsilon_2$**:** use $t(2y\text{logcf}(y, 3, 2) - x)$,

where $t := \frac{x}{2+x}$, and $y := t^2$.

Note that the formulas based on t are based on the (fast converging) formula

$$\log(1 + x) = 2\left(r + \frac{r^3}{3} + \frac{r^5}{5} + \ldots\right), \text{where } r := \frac{x}{x + 2},$$

see the reference.

`log1pmxC()` is an interface to R C API ('`Rmathlib`') function.

Value

a numeric vector (with the same attributes as `x`).

Author(s)

A translation of Morten Welinder's C code of Jan 2005, see R's bug issue PR#7307 (comment #6), parametrized and tuned by Martin Maechler.

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain.
Formula (4.1.29), p.68.

Martin Mächler (2021 ff) `log1pmx`, ... *Computing ... Probabilities in* R. **DPQ** package vignette https://CRAN.R-project.org/package=DPQ/vignettes/log1pmx-etc.pdf.

See Also

`logcf`, the auxiliary function, `lgamma1p` which calls `log1pmx`, `log1p`; also `expm1x()` which computes `expm1(x) - x` accurately, whereas `log1pmx(x)` computes `log1p(x) - x` accurately

Examples

```
(doExtras <- DPQ:::doExtras()) # TRUE e.g. if interactive()
n1 <- if(doExtras) 1001 else 201
curve(log1pmx, -.9999, 7, n=n1); abline(h=0, v=-1:0, lty=3)
curve(log1pmx, -.1,  .1,  n=n1); abline(h=0, v=0, lty=3)
curve(log1pmx, -.01, .01, n=n1) -> l1xz2; abline(h=0, v=0, lty=3)
## C and R versions correspond closely:
with(l1xz2, stopifnot(all.equal(y, log1pmxC(x), tol = 1e-15)))

e <- if(doExtras) 2^-12 else 2^-8; by.p <- 1/(if(doExtras) 256 else 64)
xd <- c(seq(-1+e, 0+100*e, by=e), seq(by.p, 5, by=by.p)) # length 676 or 5476 if do.X.
plot(xd, log1pmx(xd), type="l", col=2, main = "log1pmx(x)")
abline(h=0, v=-1:0, lty=3)

## --- Compare rexp1() with log1pmx() ----------------------------
x <- seq(-0.5, 5/8, by=1/256)
all.equal(log1pmx(x), -rlog1(x), tol = 0) # 2.838e-16 {|rel.error| <= 1.33e-15}
stopifnot(all.equal(log1pmx(x), -rlog1(x), tol = 1e-14))
## much more closely:
x <- c(-1+1e-9, -1+1/256, -(127:50)/128, (-199:295)/512, 74:196/128)
if(is.unsorted(x)) stop("x must be sorted for plots")
rlog1.x <- rlog1(x)
summary(relD <- sfsmisc::relErrV(log1pmx(x), -rlog1.x))
n.relD <- relD * 2^53
table(n.relD)
## 64-bit Linux F36 (gcc 12.2.1):
## -6  -5  -4  -3  -2  -1   0   2   4   6   8  10  12  14
##  2   3  13  24  79  93 259 120  48  22  14  15   5   1
stopifnot(-10 <= n.relD, n.relD <= 20) # above Lnx: [-6, 14]

if(requireNamespace("Rmpfr")) {
  relE <- Rmpfr::asNumeric(sfsmisc::relErrV(log1pmx(Rmpfr::mpfr(x,128)), -rlog1(x)))
  plot(x, pmax(2^-54, abs(relE)), log="y", type="l", main= "|rel.Err| of rlog1(x)")
  rl1.c <- c(-.39, 0.57, -.18, .18) # the cutoffs used inside rlog1()
  lc <- "gray"
  abline(v = rl1.c, col=lc, lty=2)
  axis(3, at=rl1.c, col=lc, cex.axis=3/4, mgp=c(2,.5,0))
  abline(h= (1:4)*2^-53,  lty=3, col = (cg <- adjustcolor(1, 1/4)))
  axis(4, at=(1:4)*2^-53, labels=expression(frac(epsilon[c],2), epsilon[c],
                                            frac(3,2)*epsilon[c], 2*epsilon[c]),
       cex.axis = 3/4, tcl=-1/4, las = 1, mgp=c(1.5,.5,0), col=cg)
  ## it seems the -.18 +.18 cutoffs should be slightly moved "outside"
}

## much more graphics etc in ../tests/dnbinom-tst.R  (and the vignette, see above)
```

---

logcf *Continued Fraction Approximation of Log-Related Power Series*

Description

Compute a continued fraction approximation to the series (infinite sum)

$$\sum_{k=0}^{\infty} \frac{x^k}{i + k \cdot d} = \frac{1}{i} + \frac{x}{i+d} + \frac{x^2}{i+2*d} + \frac{x^3}{i+3*d} + \ldots$$

Needed as auxiliary function in `log1pmx()` and `lgamma1p()`.

Usage

```
logcf (x, i, d, eps, maxit = 10000L, trace = FALSE)
logcfR(x, i, d, eps, maxit = 10000L, trace = FALSE)
logcfR_vec(x, i, d, eps, maxit = 10000L, trace = FALSE)
```

Arguments

`x`	numeric vector of values less than 1. "mpfr"-numbers (of potentially high precision, package **Rmpfr**) work in `logcfR*(x,*)`.
`i`	positive numeric
`d`	non-negative numeric
`eps`	positive number, the convergence tolerance.
`maxit`	a positive integer, the maximal number of iterations or terms in the truncated series used.
`trace`	logical (or non-negative integer in the future) indicating if (and how much) diagnostic output should be printed to the console during the computations.

Details

`logcfR()`: a pure R version where the iterations happen vectorized in `x`, only for those components `x[i]` they have not yet converged. This is particularly beneficial for not-very-short `"mpfr"` vectors `x`, and still conceptually equivalent to the `logcfR_vec()` version.

`logcfR_vec()`: a pure R version where each `x[i]` is treated separately, hence "properly" vectorized, but slowly so.

`logcf()`: only for `numeric` `x`, calls into (a clone of) R's own (non-API currently) `logcf()` C Rmathlib function.

Value

a numeric-alike vector with the same attributes as `x`. For the `logcfR*()` versions, an `"mpfr"` vector if `x` is one.

Note

Rescaling is done by (namespace hidden) "global" `scalefactor` which is 2^{256}, represented exactly (in `double` precision).

Author(s)

Martin Maechler, based on R's 'nmath/pgamma.c' implementation.

See Also

lgamma1p, log1pmx, and pbeta, whose prinicipal algorithm has evolved from TOMS 708.

Examples

```
x <- (-2:1)/2
logcf (x, 2,3, eps=1e-7, trace=TRUE) # shows iterations for each x[]
logcfR_vec(x, 2,3, eps=1e-7, trace=TRUE) # 1 line per x[]
logcfR_vec(x, 2,3, eps=1e-7, trace= 2  ) # shows iterations for each x[]

n <- 2049; x <- seq(-1,1, length.out = n)[-n] ; stopifnot(diff(x) == 1/1024)
plot(x, (lcf <- logcf(x, 2,3, eps=1e-12)), type="l", col=2)
lcR <- logcfR_vec (x, 2,3, eps=1e-12); all.equal(lcf, lcR , tol=0)
lcR.<- logcfR(x, 2,3, eps=1e-12); all.equal(lcf, lcR., tol=0)
all.equal(lcR, lcR., tol=0) # TRUE
all.equal(lcf, lcR., tol=0) # TRUE (x86_64, Lnx)
stopifnot(exprs = {
  all.equal(lcf, lcR., tol=1e-14)# seen 0 (!)
  all.equal(lcR, lcR., tol=5e-16)# seen 0 above
})

l32 <- curve(logcf(x, 3,2, eps=1e-7), -3, 1, n = 1000)
abline(h=0,v=1, lty=3, col="gray50")
##
plot(y~x, l32, log="y", type="l", main= "logcf(x, i, d)  in log-scale",
     ylim = c(.01, 10), col=2); abline(v=1, lty=3, col="gray50")
## other (i,d) than the (3,2) needed for log1pmx():
curve(logcfR(x,  5,  4,  eps = 1e-5), n=1000, add=TRUE, col = 4)
curve(logcfR(x, 20,  1,  eps = 1e-5), n=1000, add=TRUE, col = 5)
curve(logcfR(x, 1/4,1/2, eps = 1e-5), n=1000, add=TRUE, col = 6)
i.d <- cbind(c(2,3), c(5,4), c(20,1), c(1/4, 1/2))
legend("topleft", apply(i.d, 2, \(k) paste0("(i=",k[1],", d=", k[2],")")),
       lwd = 2, col = c(2,4:6), bty="n")
```

---

logspace.add *Logspace Arithmetix – Addition and Subtraction*

Description

Compute the log(arithm) of a sum (or difference) from the log of terms without causing overflows and without throwing away large handfuls of accuracy.

`logspace.add(lx, ly)`**:=**

$$\log(\exp(lx) + \exp(ly))$$

`logspace.sub(lx, ly)`**:=**

$$\log(\exp(lx) - \exp(ly))$$

Usage

```
logspace.add(lx, ly)
logspace.sub(lx, ly)
```

Arguments

lx, ly	numeric vectors, typically of the same length, but will be recycled to common length as with other R arithmetic.

Value

a numeric vector of the same length as x+y.

Note

This is really from R's C source code for pgamma(), i.e., '<R>/src/nmath/pgamma.c'

The function definitions are very simple, logspace.sub() using log1mexp().

Author(s)

Morten Welinder (for R's pgamma()); Martin Maechler

See Also

lsum, lssum; then pgamma()

Examples

```
set.seed(12)
ly <- rnorm(100, sd= 50)
lx <- ly + abs(rnorm(100, sd=100))  # lx - ly must be positive for *.sub()
stopifnot(exprs = {
   all.equal(logspace.add(lx,ly),
             log(exp(lx) + exp(ly)), tol=1e-14)
   all.equal(logspace.sub(lx,ly),
             log(exp(lx) - exp(ly)), tol=1e-14)
})
```

---

lssum *Compute Logarithm of a Sum with Signed Large Summands*

Description

Properly compute $\log(x_1 + \ldots + x_n)$ for given log absolute values `lxabs` = $log(|x_1|), .., log(|x_n|)$ and corresponding signs `signs` = $sign(x_1), .., sign(x_n)$. Here, x_i is of arbitrary sign.

Notably this works in many cases where the direct sum would have summands that had overflown to `+Inf` or underflown to `-Inf`.

This is a (simpler, vector-only) version of `copula:::lssum()` (CRAN package **copula**).

Note that the *precision* is often not the problem for the direct summation, as R's `sum()` internally uses `"long double"` precision on most platforms.

Usage

```
lssum(lxabs, signs, l.off = max(lxabs), strict = TRUE)
```

Arguments

`lxabs`	n-vector of values $\log(\|x_1\|), \ldots, \log(\|x_n\|)$.
`signs`	corresponding signs $sign(x_1), \ldots, sign(x_n)$.
`l.off`	the offset to substract and re-add; ideally in the order of `max(.)`.
`strict`	`logical` indicating if the function should stop on some negative sums.

Value

$log(x_1+..+x_n) == log(sum(x)) = log(sum(sign(x)*|x|)) == log(sum(sign(x)*exp(log(|x|)))) == log(exp(log(x0$

Author(s)

Marius Hofert and Martin Maechler (for package **copula**).

See Also

`lsum()` which computes an exponential sum in log scale with *out* signs.

Examples

```
rSamp <- function(n, lmean, lsd = 1/4, roundN = 16) {
  lax <- sort((1+1e-14*rnorm(n))*round(roundN*rnorm(n, m = lmean, sd = lsd))/roundN)
  sx <- rep_len(c(-1,1), n)
  list(lax=lax, sx=sx, x = sx*exp(lax))
}

set.seed(101)
L1 <- rSamp(1000, lmean = 700) # here, lssum() is not needed (no under-/overflow)
summary(as.data.frame(L1))
ax <- exp(lax <- L1$lax)
hist(lax); rug(lax)
hist( ax); rug( ax)
sx <- L1$sx
```

```
table(sx)
(lsSimple <- log(sum(L1$x)))           # 700.0373
(lsS <- lssum(lxabs = lax, signs = sx))# ditto
lsS - lsSimple # even exactly zero (in 64b Fedora 30 Linux which has nice 'long double')
stopifnot(all.equal(700.037327351478, lsS, tol=1e-14), all.equal(lsS, lsSimple))

L2 <- within(L1, { lax <- lax + 10; x <- sx*exp(lax) }) ; summary(L2$x) # some -Inf, +Inf
(lsSimpl2 <- log(sum(L2$ x)))                    # NaN
(lsS2 <- lssum(lxabs = L2$ lax, signs = L2$ sx)) # 710.0373
stopifnot(all.equal(lsS2, lsS + 10, tol = 1e-14))
```

`lsum` *Properly Compute the Logarithm of a Sum (of Exponentials)*

Description

Properly compute $\log(x_1 + \ldots + x_n)$. for given $log(x_1), .., log(x_n)$. Here, $x_i > 0$ for all i.

If the inputs are denoted $l_i = log(x_i)$ for $i = 1, 2, .., n$, we compute `log(sum(exp(l[])))`, numerically stably.

Simple vector version of `copula:::lsum()` (CRAN package **copula**).

Usage

```
lsum(lx, l.off = max(lx))
```

Arguments

`lx`	n-vector of values log(x_1),..,log(x_n).
`l.off`	the offset to substract and re-add; ideally in the order of the maximum of each column.

Value

$$log(x_1+..+x_n) = log(sum(x)) = log(sum(exp(log(x)))) == log(exp(log(x_max))*sum(exp(log(x)-log(x_max)))) =$$

Author(s)

Originally, via paired programming: Marius Hofert and Martin Maechler.

See Also

`lssum()` which computes a sum in log scale with specified (typically alternating) signs.

Examples

```
## The "naive" version :
lsum0 <- function(lx) log(sum(exp(lx)))

lx1 <- 10*(-80:70) # is easy
lx2 <- 600:750     # lsum0() not ok [could work with rescaling]
lx3 <- -(750:900)  # lsum0() = -Inf - not good enough
m3 <- cbind(lx1,lx2,lx3)
lx6 <- lx5 <- lx4 <- lx3
lx4[149:151] <- -Inf ## = log(0)
lx5[150] <- Inf
lx6[1] <- NA_real_
m6 <- cbind(m3,lx4,lx5,lx6)
stopifnot(exprs = {
  all.equal(lsum(lx1), lsum0(lx1))
  all.equal((ls1 <- lsum(lx1)),  700.000045400960403, tol=8e-16)
  all.equal((ls2 <- lsum(lx2)),  750.458675145387133, tol=8e-16)
  all.equal((ls3 <- lsum(lx3)), -749.541324854612867, tol=8e-16)
  ## identical: matrix-version <==> vector versions
  identical(lsum(lx4), ls3)
  identical(lsum(lx4), lsum(head(lx4, -3))) # the last three were -Inf
  identical(lsum(lx5), Inf)
  identical(lsum(lx6), lx6[1])
  identical((lm3 <- apply(m3, 2, lsum)), c(lx1=ls1, lx2=ls2, lx3=ls3))
  identical(apply(m6, 2, lsum), c(lm3, lx4=ls3, lx5=Inf, lx6=lx6[1]))
})
```

`newton` *Simple R level Newton Algorithm, Mostly for Didactical Reasons*

Description

Given the function `G()` and its derivative `g()`, `newton()` uses the Newton method, starting at `x0`, to find a point xp at which G is zero. `G()` and `g()` may each depend on the same parameter (vector) `z`.

Convergence typically happens when the stepsize becomes smaller than `eps`.

`keepAll = TRUE` to also get the vectors of consecutive values of x and G(x, z);

Usage

```
newton(x0, G, g, z,
       xMin = -Inf, xMax = Inf, warnRng = TRUE,
       dxMax = 1000, eps = 0.0001, maxiter = 1000L,
       warnIter = missing(maxiter) || maxiter >= 10L,
       keepAll = NA)
```

Arguments

x0	numeric start value.
G, g	must be functions, mathematically of their first argument, but they can accept parameters; g() must be the derivative of G.
z	parameter vector for $G()$ and $g()$, to be kept fixed.
xMin, xMax	numbers defining the allowed range for x during the iterations; e.g., useful to set to 0 and 1 during quantile search.
warnRng	logical specifying if a warning should be signalled when start value x0 is outside [xMin, xMax] and hence will be changed to one of the boundary values.
dxMax	maximal step size in x-space. (The default 1000 is quite arbitrary, do set a good maximal step size yourself!)
eps	positive number, the *absolute* convergence tolerance.
maxiter	positive integer, specifying the maximal number of Newton iterations.
warnIter	logical specifying if a warning should be signalled when the algorithm has not converged in maxiter iterations.
keepAll	logical specifying if the full sequence of x- and G(x,*) values should be kept and returned: NA, the default: newton returns a small list of final "data", with 4 components x= $x*$, G= $G(x*, z)$, it, and converged. TRUE: returns an extended list, in addition containing the vectors x.vec and G.vec. FALSE: returns only the $x*$ value.

Details

Because of the quadrativc convergence at the end of the Newton algorithm, often x^* satisfies approximately $|G(x*, z)| < eps^2$.

newton() can be used to compute the quantile function of a distribution, if you have a good starting value, and provide the cumulative probability and density functions as R functions G and g respectively.

Value

The result always contains the final x-value $x*$, and typically some information about convergence, depending on the value of keepAll, see above:

x	the optimal x^* value (a number).
G	the function value $G(x*, z)$, typically very close to zero.
it	the integer number of iterations used.
convergence	logical indicating if the Newton algorithm converged within maxiter iterations.
x.vec	the full vector of x values, $\{x0, \ldots, x^*\}$.
G.vec	the vector of function values (typically tending to zero), i.e., G(x.vec, .) (even when G(x, .) would not vectorize).

Author(s)

Martin Maechler, ca. 2004

References

Newton's Method on Wikipedia, https://en.wikipedia.org/wiki/Newton%27s_method.

See Also

uniroot() is much more sophisticated, works without derivatives and is generally faster than newton().

newton(.) is currently crucially used (only) in our function qchisqN().

Examples

```
## The most simple non-trivial case :  Computing SQRT(a)
  G <- function(x, a) x^2 - a
  g <- function(x, a) 2*x

  newton(1, G, g, z = 4  ) # z = a -- converges immediately
  newton(1, G, g, z = 400) # bad start, needs longer to converge

## More interesting, and related to non-central (chisq, e.t.) computations:
## When is  x * log(x) < B,  i.e., the inverse function of G = x*log(x) :
xlx <- function(x, B) x*log(x) - B
dxlx <- function(x, B) log(x) + 1

Nxlx <- function(B) newton(B, G=xlx, g=dxlx, z=B, maxiter=Inf)$x
N1   <- function(B) newton(B, G=xlx, g=dxlx, z=B, maxiter = 1)$x
N2   <- function(B) newton(B, G=xlx, g=dxlx, z=B, maxiter = 2)$x

Bs <- c(outer(c(1,2,5), 10^(0:4)))
plot (Bs, vapply(Bs, Nxlx, pi), type = "l", log ="xy")
lines(Bs, vapply(Bs, N1  , pi), col = 2, lwd = 2, lty = 2)
lines(Bs, vapply(Bs, N2  , pi), col = 3, lwd = 3, lty = 3)

BL <- c(outer(c(1,2,5), 10^(0:6)))
plot (BL, vapply(BL, Nxlx, pi), type = "l", log ="xy")
lines(BL, BL, col="green2", lty=3)
lines(BL, vapply(BL, N1  , pi), col = 2, lwd = 2, lty = 2)
lines(BL, vapply(BL, N2  , pi), col = 3, lwd = 3, lty = 3)
## Better starting value from an approximate 1 step Newton:
iL1 <- function(B) 2*B / (log(B) + 1)
lines(BL, iL1(BL), lty=4, col="gray20") ## really better ==> use it as start

Nxlx <- function(B) newton(iL1(B), G=xlx, g=dxlx, z=B, maxiter=Inf)$x
N1   <- function(B) newton(iL1(B), G=xlx, g=dxlx, z=B, maxiter = 1)$x
N2   <- function(B) newton(iL1(B), G=xlx, g=dxlx, z=B, maxiter = 2)$x

plot (BL, vapply(BL, Nxlx, pi), type = "o", log ="xy")
lines(BL, iL1(BL),  lty=4, col="gray20")
```

```
lines(BL, vapply(BL, N1  , pi), type = "o", col = 2, lwd = 2, lty = 2)
lines(BL, vapply(BL, N2  , pi), type = "o", col = 3, lwd = 2, lty = 3)
## Manual 2-step Newton
iL2 <- function(B) { lB <- log(B) ; B*(lB+1) / (lB * (lB - log(lB) + 1)) }
lines(BL, iL2(BL), col = adjustcolor("sky blue", 0.6), lwd=6)
##==>  iL2() is very close to true curve
## relative error:
iLtrue <- vapply(BL, Nxlx, pi)
cbind(BL, iLtrue, iL2=iL2(BL), relErL2 = 1-iL2(BL)/iLtrue)
## absolute error (in log-log scale; always positive!):
plot(BL, iL2(BL) - iLtrue, type = "o", log="xy", axes=FALSE)
if(requireNamespace("sfsmisc")) {
  sfsmisc::eaxis(1)
  sfsmisc::eaxis(2, sub10=2)
} else {
  cat("no 'sfsmisc' package; maybe  install.packages(\"sfsmisc\")  ?\n")
  axis(1); axis(2)
}
## 1 step from iL2()  seems quite good:
B. <- BL[-1] # starts at 2
NL2 <- lapply(B., function(B) newton(iL2(B), G=xlx, g=dxlx, z=B, maxiter=1))
str(NL2)
iL3 <- sapply(NL2, `[[`, "x")
cbind(B., iLtrue[-1], iL2=iL2(B.), iL3, relE.3 = 1- iL3/iLtrue[-1])
x. <- iL2(B.)
all.equal(iL3, x. - xlx(x., B.) / dxlx(x.)) ## 7.471802e-8
## Algebraic simplification of one newton step :
all.equal((x.+B.)/(log(x.)+1), x. - xlx(x., B.) / dxlx(x.), tol = 4e-16)
iN1 <- function(x, B) (x+B) / (log(x) + 1)
B <- 12345
iN1(iN1(iN1(B, B),B),B)
Nxlx(B)
```

---

numer-utils *Numerical Utilities - Functions, Constants*

Description

The **DPQ** package provides some numeric constants used in some of its distribution computations.

`all_mpfr()` and `any_mpfr()` return `TRUE` iff all (or 'any', respectively) of their arguments inherit from `class` `"mpfr"` (from package **Rmpfr**).

`logr(x,a)` computes `log(x / (x + a))` in a numerically stable way.

`modf(x)` splits each `x` into integer part (as `trunc(x)`) and fractional (remainder) part in $(-1, 1)$ and corresponds to the R version of the C99 (and POSIX) standard C (and C++) mathlib functions of the same name.

Usage

```
## Numeric Constants : % mostly in   ../R/beta-fns.R
M_LN2        # = log(2)  = 0.693....
M_SQRT2      # = sqrt(2) = 1.4142...
M_cutoff     # := If |x| > |k| * M_cutoff, then  log[ exp(-x) * k^x ]  =~=  -x
             #  = 3196577161300663808 ~= 3.2e+18
M_minExp     # = log(2) * .Machine$double.min.exp # ~= -708.396..
G_half       # = sqrt(pi) = Gamma( 1/2 )

## Functions :
all_mpfr(...)
any_mpfr(...)
logr(x, a)    # == log(x / (x + a)) -- but numerically smart; x >= 0, a > -x
modf(x)
okLongDouble(lambda = 999, verbose = 0L, tol = 1e-15)
```

Arguments

`...`	numeric or `"mpfr"` numeric vectors.
`x, a`	number-like, not negative, now may be vectors of `length(.) > 1`.
`lambda`	a number, typically in the order of 500–10'000.
`verbose`	a non-negative integer, if not zero, `okLongDouble()` prints the intermediate long double computations' results.
`tol`	numerical tolerance used to determine the accuracy required for near equality in `okLongDouble()`.

Details

`all_mpfr()`,

`all_mpfr()`: test if `all` or `any` of their arguments or of class `"mpfr"` (from package **Rmpfr**). The arguments are evaluated only until the result is determined, see the example.

`logr()` computes $\log(x/(x+a))$ in a numerically stable way.

Value

The numeric constant in the first case; a numeric (or "mpfr") vector of appropriate size in the 2nd case.

`okLongDouble()` returns a `logical`, `TRUE` iff the long double arithmetic with `expl()` and `logl()` seems to work accurately and consistently for `exp(-lambda)` and `log(lambda)`.

Author(s)

Martin Maechler

See Also

`.Machine`

Examples

```
(Ms <- ls("package:DPQ", pattern = "^M"))
lapply(Ms, function(nm) { cat(nm,": "); print(get(nm)) }) -> .tmp

logr(1:3, a=1e-10)

okLongDouble(verbose=TRUE) # verbose: show (C-level) computations
## typically TRUE, but not e.g. in a valgrinded R-devel of Oct.2019
## Here is typically the "boundary":
rr <- try(uniroot(function(x) okLongDouble(x) - 1/2,
             c(11350, 11400), tol=1e-7, extendInt = "yes"))
str(rr, digits=9) ## seems somewhat platform dependent: now see
## $ root      : num 11376.563
## $ estim.prec: num 9.313e-08
## $ iter      : int 29

set.seed(2021); x <- runif(100, -7,7)
mx <- modf(x)
with(mx, head( cbind(x, i=mx$i, fr=mx$fr) )) # showing the first cases
with(mx, stopifnot(   x == fr + i,
                      i == trunc(x),
                sign(fr) == sign(x)))
```

---

p1l1 *Numerically Stable p1l1(t) = (t+1)*log(1+t) - t*

Description

The binomial deviance function `bd0(x,M)` := $D_0(x,M) := M \cdot d_0(x/M)$ (see `bd0`) can mathematically be re-written as $D_0(x,M) = M \cdot p_1 l_1\left(\frac{x-M}{M}\right)$ where we look into providing numerically stable formulas for `p1l1(t)`$= p_1 l_1(t)$, as its direct mathematical formula $p_1 l_1(t) = (t+1)\log(1+t) - t$ suffers from cancellation for small $|t|$, even when `log1p(t)` is used instead of `log(1+t)`.

Using a hybrid implementation, `p1l1()` uses a direct formula, now the stable one in `p1l1p(t) := log1pmx(t) + t*log1p(t)`, for $|t| > c$ and a series approximation for basically $|t| \le c$, $c \approx 0.07$.

NB: The re-expression via `log1pmx(x) := log(1+x) - x` is almost perfect; it fixes the cancellation problem entirely (and analysis further suggests that `log1pmx()`'s internal cutoff seems sub optimal).

Usage

```
 p1l1p  (t, ...)
 p1l1.  (t)
 p1l1   (t,    F = t^2/2, ...)
 p1l1ser(t, k, F = t^2/2)
.p1l1ser(t, k, F = t^2/2)
```

Arguments

t	numeric a-like vector ("mpfr" included), larger (or equal) to -1.
...	optional (tuning) arguments, passed to log1pmx().
k	small positive integer, the number of terms to use in the Taylor series approximation p1l1ser(t,k) of p1l1(t).
F	(numeric vector) multiplication **f**actor; *must* be t^2/2 for the p1l1() function, but can be modified, e.g. in more direct bd0() computations.

Details

for now see in bd0().

Value

numeric vector "as" t.

Author(s)

Martin Maechler

See Also

Both log1pmx() and bd0; our package vignette log1pmx, bd0, stirlerr - *Probability Computations in R*. Further, dbinom, also for the C. Loader (2000) reference.

Examples

```
(doExtras <- DPQ:::doExtras()) # TRUE e.g. if interactive()

t <- seq(-1, 4, by=1/64)
plot(t, p1l1ser(t, 1), type="l")
lines(t, p1l1.(t), lwd=5, col=adjustcolor(1, 1/2)) # direct formula
for(k in 2:6) lines(t, p1l1ser(t, k), col=k)

## zoom in
t <- 2^seq(-59,-1, by=1/4)
t <- c(-rev(t), 0, t)
stopifnot(!is.unsorted(t))
k.s <- 1:12; names(k.s) <- paste0("k=", 1:12)

## True function values: use Rmpfr with 256 bits precision: ---
### eventually move this to ../tests/ & ../vignettes/log1pmx-etc.Rnw
#### FIXME: eventually replace with  if(requireNamespace("Rmpfr")){ ......}
#### =====
if((needRmpfr <- is.na(match("Rmpfr", (srch0 <- search())))))
    require("Rmpfr")
p1l1.T <- p1l1.(mpfr(t, 256)) # "true" values
p1l1.n <- asNumeric(p1l1.T)
all.equal(sapply(k.s, function(k)  p1l1ser(t,k)) -> m.p1l1,
          sapply(k.s, function(k) .p1l1ser(t,k)) -> m.p1l., tolerance = 0)
```

```
p1tab <-
    cbind(b1 = bd0(t+1, 1),
          b.10 = bd0(10*t+10,10)/10,
          dirct = p1l1.(t),
          p1l1p = p1l1p(t),
          p1l1  = p1l1 (t),
          sapply(k.s, function(k) p1l1ser(t,k)))
matplot(t, p1tab, type="l", ylab = "p1l1*(t)")
## (absolute) error:
##' legend for matplot()
mpLeg <- function(leg = colnames(p1tab), xy = "top", col=1:6, lty=1:5, lwd=1,
                  pch = c(1L:9L, 0L, letters, LETTERS)[seq_along(leg)], ...)
    legend(xy, legend=leg, col=col, lty=lty, lwd=lwd, pch=pch, ncol=3, ...)

titAbs <- "Absolute errors of p1l1(t) approximations"
matplot(t, asNumeric(p1tab - p1l1.T), type="o", main=titAbs); mpLeg()
i <- abs(t) <= 1/10 ## zoom in a bit
matplot(t[i], abs(asNumeric((p1tab - p1l1.T)[i,])), type="o", log="y",
        main=titAbs, ylim = c(1e-18, 0.003)); mpLeg()
## Relative Error
titR <- "|Relative error| of p1l1(t) approximations"
matplot(t[i], abs(asNumeric((p1tab/p1l1.T - 1)[i,])), type="o", log="y",
        ylim = c(1e-18, 2^-10), main=titR)
mpLeg(xy="topright", bg= adjustcolor("gray80", 4/5))
i <- abs(t) <= 2^-10 # zoom in more
matplot(t[i], abs(asNumeric((p1tab/p1l1.T - 1)[i,])), type="o", log="y",
        ylim = c(1e-18, 1e-9))
mpLeg(xy="topright", bg= adjustcolor("gray80", 4/5))


## Correct number of digits
corDig <- asNumeric(-log10(abs(p1tab/p1l1.T - 1)))
cbind(t, round(corDig, 1))# correct number of digits

matplot(t, corDig, type="o", ylim = c(1,17))
(cN <- colnames(corDig))
legend(-.5, 14, cN, col=1:6, lty=1:5, pch = c(1L:9L, 0L, letters), ncol=2)

## plot() function >>>> using global (t, corDig) <<<<<<<<<
p.relEr <- function(i, ylim = c(11,17), type = "o",
                    leg.pos = "left", inset=1/128,
                    main = sprintf(
                       "Correct #{Digits} in p1l1() approx., notably Taylor(k=1 .. %d)",
                                   max(k.s)))
{
    if((neg <- all(t[i] < 0)))
        t  <- -t
    stopifnot(all(t[i] > 0), length(ylim) == 2) # as we use log="x"
    matplot(t[i], corDig[i,], type=type, ylim=ylim, log="x", xlab = quote(t), xaxt="n",
            main=main)
    legend(leg.pos, cN, col=1:6, lty=1:5, pch = c(1L:9L, 0L, letters), ncol=2,
           bg=adjustcolor("gray90", 7/8), inset=inset)
    t.epsC <- -log10(c(1,2,4)* .Machine$double.eps)
```

```
    axis(2, at=t.epsC, labels = expression(epsilon[C], 2*epsilon[C], 4*epsilon[C]),
         las=2, col=2, line=1)
    tenRs <- function(t) floor(log10(min(t))) : ceiling(log10(max(t)))
    tenE <- tenRs(t[i])
    tE <- 10^tenE
    abline (h = t.epsC,
            v = tE, lty=3, col=adjustcolor("gray",.8), lwd=2)
    AX <- if(requireNamespace("sfsmisc")) sfsmisc::eaxis else axis
    AX(1, at= tE, labels = as.expression(
                      lapply(tenE,
                             if(neg)
                                 function(e) substitute(-10^{E}, list(E = e +0))
                             else
                                 function(e) substitute( 10^{E}, list(E = e +0)))))
}

p.relEr(t > 0, ylim = c(1,17))
p.relEr(t > 0) # full positive range
p.relEr(t < 0) # full negative range
if(FALSE) {## (actually less informative):
 p.relEr(i = 0 < t & t < .01)  ## positive small t
 p.relEr(i = -.1 < t & t < 0) ## negative small t
}

## Find approximate formulas for accuracy of k=k*  approximation
d.corrD <- cbind(t=t, as.data.frame(corDig))
names(d.corrD) <- sub("k=", "nC_",  names(d.corrD))

fmod <- function(k, data, cut.y.at = -log10(2 * .Machine$double.eps),
                 good.y = -log10(.Machine$double.eps), # ~ 15.654
                 verbose=FALSE) {
    varNm <- paste0("nC_",k)
    stopifnot(is.numeric(y <- get(varNm, data, inherits=FALSE)),
              is.numeric(t <- data$t))# '$' works for data.frame, list, environment
    i <- 3 <= y & y <= cut.y.at
    i.pos <- i & t > 0
    i.neg <- i & t < 0
    if(verbose) cat(sprintf("k=%d >> y <= %g ==> #{pos. t} = %d ;  #{neg. t} = %d\n",
                            k, cut.y.at, sum(i.pos), sum(i.neg)))
    nCoefLm <- function(x,y) `names<-`(.lm.fit(x=x, y=y)$coeff, c("int", "slp"))
   nC.t <- function(x,y) { cf <- nCoefLm(x,y); c(cf, t.0 = exp((good.y - cf[[1]])/cf[[2]])) }
    cbind(pos = nC.t(cbind(1, log( t[i.pos])), y[i.pos]),
          neg = nC.t(cbind(1, log(-t[i.neg])), y[i.neg]))
}
rr <- sapply(k.s, fmod, data=d.corrD, verbose=TRUE, simplify="array")
stopifnot(rr["slp",,] < 0) # all slopes are negative (important!)
matplot(k.s, t(rr["slp",,]), type="o", xlab = quote(k), ylab = quote(slope[k]))
## fantastcally close to linear in k
## The numbers, nicely arranged
ftable(aperm(rr, c(3,2,1)))
signif(t(rr["t.0",,]),3) # ==> Should be boundaries for the hybrid p1l1()
##           pos      neg
## k=1  6.60e-16 6.69e-16
```

```
## k=2  3.65e-08 3.65e-08
## k=3  1.30e-05 1.32e-05
## k=4  2.39e-04 2.42e-04
## k=5  1.35e-03 1.38e-03
## k=6  4.27e-03 4.34e-03
## k=7  9.60e-03 9.78e-03
## k=8  1.78e-02 1.80e-02
## k=9  2.85e-02 2.85e-02
## k=10 4.13e-02 4.14e-02
## k=11 5.62e-02 5.64e-02
## k=12 7.24e-02 7.18e-02

###------------- Well,  p1l1p()  is really basically good enough ... with a small exception:
rErr1k <- curve(asNumeric(p1l1p(x) / p1l1.(mpfr(x, 4096)) - 1), -.999, .999,
                n = if(doExtras) 4000 else 800, col=2, lwd=2)
abline(h = c(-8,-4,-2:2,4,8)* 2^-52, lty=2, col=adjustcolor("gray20", 1/4))
## well, have a "spike" at around -0.8 -- why?

plot(abs(y) ~ x, data = rErr1k, ylim = c(4e-17, max(abs(y))),
     ylab = expression(abs(hat(p)/p - 1)),
     main = "p1l1p(x) -- Relative Error wrt mpfr(*. 4096) [log]",
     col=2, lwd=1.5, type = "b", cex=1/2, log="y", yaxt="n")
sfsmisc::eaxis(2)
eps124 <-  c(1, 2,4,8)* 2^-52
abline(h = eps124, lwd=c(3,1,1,1), lty=c(1,2,2,2), col=adjustcolor("gray20", 1/4))
axLab <- expression(epsilon[c], 2*epsilon[c], 4*epsilon[c], 8*epsilon[c])
axis(4, at = eps124, labels = axLab, col="gray20", las=1)
abline(v= -.791, lty=3, lwd=2, col="blue4") # -.789  from visual ..
##--> The "error" is in log1pmx() which has cutoff minLog1Value = -0.79149064
##--> which is clearly not optimal, at least not for computing p1l1p()

d <- if(doExtras) 1/2048 else 1/512; x <- seq(-1+d, 1, by=d)
p1l1Xct <- p1l1.(mpfr(x, if(doExtras) 4096 else 512))
rEx.5 <- asNumeric(p1l1p(x, minL1 = -0.5) / p1l1Xct - 1)
lines(x, abs(rEx.5), lwd=2.5, col=adjustcolor(4, 1/2)); abline(v=-.5, lty=2,col=4)
rEx.25 <- asNumeric(p1l1p(x, minL1 = -0.25) / p1l1Xct - 1)
lines(x, abs(rEx.25), lwd=3.5, col=adjustcolor(6, 1/2)); abline(v=-.25, lty=2,col=6)
lines(lowess(x, abs(rEx.5),  f=1/20), col=adjustcolor(4,offset=rep(1,4)/3), lwd=3)
lines(lowess(x, abs(rEx.25), f=1/20), col=adjustcolor(6,offset=rep(1,4)/3), lwd=3)

rEx.4 <- asNumeric(p1l1p(x, tol_logcf=1e-15, minL1 = -0.4) / p1l1Xct - 1)
lines(x, abs(rEx.4), lwd=5.5, col=adjustcolor("brown", 1/2)); abline(v=-.25, lty=2,col="brown")

if(needRmpfr && isNamespaceLoaded("Rmpfr"))
    detach("package:Rmpfr")
```

---

pbetaAS_eq20 pbeta() *Approximations of Abramowitz & Stegun,* $26.5.\{20, 21\}$

Description

Compute (simple) `pbeta()` approximations; Abramowitz & Stegun, eq. 26.5.20 and 26.5.21. The first, eq. 26.5.20, approximates via a χ^2 distribution, i.e., calls `pchisq()` with appropriate arguments, where the 2nd, eq. 26.5.21., approximates via the normal (aka "Gaussian") distribution, i.e., calls `pnorm()`.

In addition, rather for "didactical" reasons or simple comparisons, we also provide `pbetaNorm2()` which simply uses the normal approximation with matching moments, i.e., uses $Phi^{-1}((x-\mu)/\sigma)$ where $\mu = \mu(\alpha, \beta)$ is the mean, and $\sigma = \sigma(\alpha, \beta)$ is the standard deviation of the $B(\alpha, \beta)$ Beta distribution, where $\alpha =$`shape1`, $\beta =$`shape2`,

$$\mu(\alpha, \beta) = \frac{\alpha}{\alpha + \beta}, \text{ and}$$

$$\sigma(\alpha, \beta) = \frac{\sqrt{\alpha\beta}, (\alpha + \beta)\sqrt{\alpha + \beta + 1}}{.}$$

If the "validity" conditions are not fulfilled, the functions will try to "swap tails", i.e., using the (anti-) symmetries of the Beta distribution, viz. computing $1 - F(1 - x, b, a)$ instead of $F(x, a, b)$.

Usage

```
pbetaAS_eq20(q, shape1, shape2, lower.tail = TRUE, log.p = FALSE, warnBad = TRUE)
pbetaAS_eq21(q, shape1, shape2, lower.tail = TRUE, log.p = FALSE, warnBad = TRUE)
pbetaNorm2  (q, shape1, shape2, lower.tail = TRUE, log.p = FALSE)

## lower-level -- direct formula without range checks :
.pbeta.eq20(q, a,b, lower.tail = TRUE, log.p = FALSE, ._1_q = .5 - q + .5)
.pbetaSeq20(q, a,b, lower.tail = TRUE, log.p = FALSE)
.pbeta.eq21(q, a,b, lower.tail = TRUE, log.p = FALSE, ._1_q = .5 - q + .5)
.pbetaSeq21(q, a,b, lower.tail = TRUE, log.p = FALSE)
```

Arguments

q, shape1, shape2	non-negative vectors (recycled to common length), q in $[0, 1]$, see `pbeta`.
a, b	the same as shape1 and shape2, respectively; for convenience in (R) formulas only.
lower.tail	indicating if $F(q; *)$ should be returned or the upper tail probability $1 - F(q)$.
log.p	logical indicating if `log(<pr>)`, log-probabilities should be returned instead of (default) the probabilities <pr>.
warnBad	logical indicating if a `warning` should be signalled when for some `(q, shape1, shape2, lower.tail)` combinations, A.&S. indicate the approximation to be bad, i.e., potentially quite inaccurate.
._1_q	$= 1 - q$, computed numerically carefully (potentially with less cancellation error), or pre-computed.

Value

a numeric vector, the length of e.g., `length(q + shape1 + shape2)`, i.e., recycled to common length. With the corresponding `pbeta()` approximations, i.e., probabilities (in $[0, 1]$) or log-probabilities in $[-\infty, 0]$.

Author(s)

Martin Maechler

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain; formulas (26.5.20) & (26.5.21), p.945.

Beta Distribution (in Wikipedia) https://en.wikipedia.org/wiki/Beta_distribution

See Also

`pbeta`; our `pbetaRv1`, and approximations to the non-central beta `pnbetaAppr2`.

Examples

```
str(alf <- 2^seq(1,15, by=1/8)) # 2 1.18 ... 32768
bet <- seq_along(alf) # 1 2 .. 113
E <- alf/(alf + bet) # = E[X]
pb   <- pbeta       (E, alf, bet)
pb20 <- pbetaAS_eq20(E, alf, bet) # warning .. 112 potentially bad
pb21 <- pbetaAS_eq21(E, alf, bet)
pbN2 <- pbetaNorm2  (E, alf, bet); summary(pbN2)# all == 0.5 as q = E[ Beta(a,b) ]
pbM <- cbind(pb20, pb21, pbN2)
tit <- quote(list(pbeta[...](E, alpha, beta) ~~" ", alpha == 2^{1~".."~15},
                  beta==1:113, E == frac(alpha,alpha+beta)))
matplot(alf, cbind(pb, pbM), type = "l", lwd = 1:4, log = "x",
        xlab = quote(alpha ~~ "[log]"), xaxt = "n", main = tit)
sfsmisc::eaxis(1, sub10=2)
legend("topright", ncol=2, bty="n", lty = 1:4, col = adjustcolor(1:4, 3/4), lwd = 1:4,
       expression(pbeta(), pbetaAS_eq20(), pbetaAS_eq21(), pbetaNorm2()))
stopifnot(apply(pbM, 2, all.equal, target = pb, tolerance = 0.03))
## relative errors, compared to R's pbeta()
round(cbind(sort(apply(pbM, 2, sfsmisc::relErr, target = pb))), 5)
## pb21 0.00013 << good here
## pb20 0.00356 << "ok" (got warning about "bad")
## pbN2 0.02742 << (constant)

## 2.  below E[]: mu - 2 s ================================================
## ----------------------  --> use  xmin :
sdB <- sqrt(alf * bet)/(alf + bet)/sqrt(alf + bet + 1)
xmin <- unique(pmax(0, E - 2*sdB))
xmax <- unique(pmin(1, E + 2*sdB))
##
pb   <- pbeta       (xmin, alf, bet)
```

```
pb20 <- pbetaAS_eq20(xmin, alf, bet) # warning .. 104 potentially bad
## --> analysis shows the 'swap' logical to be clearly sub optimal.  ==> functions .pbeta.eq20() etc
pb21 <- pbetaAS_eq21(xmin, alf, bet)
pbN2 <- pbetaNorm2  (xmin, alf, bet); summary(pbN2)
pbM <- cbind(pb20, pb21, pbN2)
round(cbind(sort(apply(pbM, 2, sfsmisc::relErr, target = pb))), 5)
## pb21 0.00594
## pbN2 0.27069
## pb20 0.68668

tit <- quote(list(pbeta[...](xmin, alpha, beta) ~~" ", alpha == 2^{1~".."~15},
                  beta==1:113, xmin == (mu - 2*sigma)(alpha,beta)))
matplot(alf, cbind(pb, pbM), type = "l", lwd = 1:4, log = "x",
        xlab = quote(alpha ~~ "[log]"), xaxt = "n", main = tit)
sfsmisc::eaxis(1, sub10=2)
legend("topright", ncol=2, bty="n", lty = 1:4, lwd = 1:4, col = 1:4, # col = adjustcolor(1:4, 3/4),
       legend = expression(pbeta(), pbetaAS_eq20(), pbetaAS_eq21(), pbetaNorm2()))
##
pb20. <- .pbeta.eq20(xmin, alf,bet)
pb20S <- .pbetaSeq20(xmin, alf,bet)
pb21. <- .pbeta.eq21(xmin, alf,bet)
pb21S <- .pbetaSeq21(xmin, alf,bet)
matlines(alf, cbind(pb20., pb20S, pb21., pb21S), lwd = 4, lty = 1, col = adjustcolor(2:5, 1/4))
legend("right", ncol=2, bty="n", lty = 1, lwd = 4, col = adjustcolor(2:5, 1/4),
       legend = paste0(".pbeta", c(".eq20", "Seq20", ".eq21", "Seq21"), "()"))

## 3.  above E[]: mu + 2 s ===========================================================
## ----------------------  --> use  xmax
##
pb   <- pbeta       (xmax, alf, bet)
pb20 <- pbetaAS_eq20(xmax, alf, bet) # warning .. 104 potentially bad
## --> analysis shows the 'swap' logical to be clearly sub optimal.  ==> functions .pbeta.eq20() etc
pb21 <- pbetaAS_eq21(xmax, alf, bet)
pbN2 <- pbetaNorm2  (xmax, alf, bet); summary(pbN2)
pbM <- cbind(pb20, pb21, pbN2)
round(cbind(sort(apply(pbM, 2, sfsmisc::relErr, target = pb))), 5)
## pb21 0.00009
## pb20 0.00559
## pbN2 0.00610
tit <- quote(list(pbeta[...](xmax, alpha, beta) ~~" ", alpha == 2^{1~"."~15},
                  beta==1:113, xmax == (mu + 2*sigma)(alpha,beta)))
matplot(alf, cbind(pb, pbM), type = "l", lwd = 1:4, log = "x",
        xlab = quote(alpha ~~ "[log]"), xaxt = "n", main = tit)
sfsmisc::eaxis(1, sub10=2)
legend("topright", ncol=2, bty="n", lty = 1:4, lwd = 1:4, col = 1:4, # col = adjustcolor(1:4, 3/4),
       legend = expression(pbeta(), pbetaAS_eq20(), pbetaAS_eq21(), pbetaNorm2()))
pb20. <- .pbeta.eq20(xmax, alf,bet)
pb20S <- .pbetaSeq20(xmax, alf,bet)
pb21. <- .pbeta.eq21(xmax, alf,bet)
pb21S <- .pbetaSeq21(xmax, alf,bet)
matlines(alf, cbind(pb20., pb20S, pb21., pb21S), lwd = 4, lty = 1, col = adjustcolor(2:5, 1/4))
legend("right", ncol=2, bty="n", lty = 1, lwd = 4, col = adjustcolor(2:5, 1/4),
       legend = paste0(".pbeta", c(".eq20", "Seq20", ".eq21", "Seq21"), "()"))
```

`pbetaRv1` *Pure R Implementation of Old pbeta()*

Description

`pbetaRv1()` is an implementation of the original ("version 1" `pbeta()` function in R (versions <= 2.2.x), before we started using TOMS 708 `bratio()` instead, see the current `pbeta` help page also for references.

`pbetaRv1()` is basically a manual translation from C to R of the underlying `pbeta_raw()` C function, see in R's source tree at https://svn.r-project.org/R/branches/R-2-2-patches/src/nmath/pbeta.c

For consistency within R, we are using R's argument names `(q, shape1, shape2)` instead of C code's `(x, pin, qin)`.

It is only for the *central* beta distribution.

Usage

```
pbetaRv1(q, shape1, shape2, lower.tail = TRUE,
         eps = 0.5 * .Machine$double.eps,
         sml = .Machine$double.xmin,
         verbose = 0)
```

Arguments

`q, shape1, shape2`	non-negative numbers, q in $[0, 1]$, see `pbeta`.
`lower.tail`	indicating if $F(q; *)$ should be returned or the upper tail probability $1 - F(q)$.
`eps`	the tolerance used to determine congerence. `eps` has been hard coded in C code to `0.5 * .Machine$double.eps` which is equal to 2^{-53} or `1.110223e-16`.
`sml`	the smallest positive number on the typical platform. The default `.Machine$double.xmin` is hard coded in the C code (as `DBL_MIN`), and this is equal to 2^{-1022} or `2.225074e-308` on all current platforms.
`verbose`	integer indicating the amount of verbosity of diagnostic output, `0` means no output, 1 more, etc.

Value

a number.

Note

The C code contains
This routine is a translation into C of a Fortran subroutine by W. Fullerton of Los Alamos Scientific Laboratory.

Author(s)

Martin Maechler

References

(From the C code:)

Nancy E. Bosten and E.L. Battiste (1974). Remark on Algorithm 179 (S14): Incomplete Beta Ratio. *Communications of the ACM*, **17**(3), 156–7.

See Also

pbeta; our approximations to the non-central beta pnbetaAppr2.

Examples

```
all.equal(pbetaRv1(1/4, 2, 3),
          pbeta   (1/4, 2, 3))
set.seed(101)

N <- 1000
x <- sample.int(7, N, replace=TRUE) / 8
a <-   rlnorm(N)
b <- 5*rlnorm(N)
pbt <- pbeta(x, a, b)
for(i in 1:N) {
   stopifnot(all.equal(pbetaRv1(x[i], a[i], b[i]), pbt[i]))
   cat(".", if(i %% 20 == 0) paste0(i, "\n")) #%
}

if(requireNamespace("Rmpfr"))
    pbetaRv1(.1, 1e-3, Rmpfr::mpfr(1e7, 128), sml = 2^Rmpfr::mpfr(-1e20, 128),
             lower.tail = FALSE, verbose=2)
```

---

phyperAllBin *Compute Hypergeometric Probabilities via Binomial Approximations*

Description

Simple utilities for ease of comparison of the different phyper approximation in package **DPQ**:

- phyperAllBinM() computes all four Molenaar binomial approximations to the hypergeometric cumulative distribution function phyper().
- phyperAllBin() computes Molenaar's four and additionally the other four phyperBin.1(), *.2, *.3, and *.4.
- .suppHyper(), *supp*ort of the Hyperbolic, is a simple 1-liner, providing all sensible integer values for the first argument q (or also x) of the hyperbolic probability functions (dhyper() and phyper()), and their approximations (here in **DPQ**).

Usage

```
phyperAllBin (m, n, k, q = .suppHyper(m, n, k), lower.tail = TRUE, log.p = FALSE)
phyperAllBinM(m, n, k, q = .suppHyper(m, n, k), lower.tail = TRUE, log.p = FALSE)
.suppHyper(m, n, k)
```

Arguments

m	the number of white balls in the urn.
n	the number of black balls in the urn.
k	the number of balls drawn from the urn, hence must be in $0, 1, \ldots, m + n$.
q	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls. The default, `.suppHyper(m, n, k)` provides the full (finite) support.
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).

Value

the `phyperAllBin*()` functions return a numeric `matrix`, with each column a different approximation to `phyper(m,n,k,q, lower.tail, log.p)`.

Note that the columns of `phyperAllBinM()` are a *subset* of those from `phyperAllBin()`.

Author(s)

Martin Maechler

References

See those in `phyperBinMolenaar`.

See Also

`phyperBin.1` etc, and `phyperBinMolenaar`.

`phyper`

Examples

```
.suppHyper # very simple:
stopifnot(identical(.suppHyper, ignore.environment = TRUE,
         function (m, n, k) max(0, k-n):min(k, m)))

phBall <- phyperAllBin (5,15, 7)
phBalM <- phyperAllBinM(5,15, 7)
stopifnot(identical( ## indeed, ph...AllBinM() gives a *subset* of ph...AllBin():
            phBall[, colnames(phBalM)] ,
            phBalM)
         , .suppHyper(5, 15, 7) == 0:5
)
```

```
round(phBall, 4)
cbind(q = 0:5, round(-log10(abs(1 - phBall / phyper(0:5, 5,15,7))),  digits=2))

require(sfsmisc)## -->  relErrV() {and eaxis()}:
qq <-    .suppHyper(20, 47, 31)
phA <- phyperAllBin(20, 47, 31)
rE <- relErrV(target = phyper(qq, 20,47,31), phA)
signif(cbind(qq, rE), 4)
## Relative approximation error [ log scaled ] :
matplot(qq, abs(rE), type="b", log="y", yaxt="n")
eaxis(2)
## ---> approximations useful only "on the right", aka the right tail
```

`phyperApprAS152` *Normal Approximation to cumulative Hyperbolic Distribution – AS 152*

Description

Compute the normal approximation (via `pnorm(.)` from AS 152 to the cumulative hyperbolic distribution function `phyper()`.

Usage

```
phyperApprAS152(q, m, n, k)
```

Arguments

`q`	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
`m`	the number of white balls in the urn.
`n`	the number of black balls in the urn.
`k`	the number of balls drawn from the urn, hence must be in $0, 1, \ldots, m + n$.

Value

a `numeric` vector of the same length (etc) as `q`.

Note

I have Fortran (and C code translated from Fortran) which says

```
ALGORITHM AS R77  APPL. STATIST. (1989), VOL.38, NO.1
Replaces AS 59 and AS 152
Incorporates AS R86 from vol.40(2)
```

Author(s)

Martin Maechler, 19 Apr 1999

References

Lund, Richard E. (1980) Algorithm AS 152: Cumulative Hypergeometric Probabilities. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, **29**(2), 221–223. doi:10.2307/2986315

Shea, B. (1989) Remark AS R77: A Remark on Algorithm AS 152: Cumulative Hypergeometric Probabilities. *JRSS C (Applied Statistics)*, **38**(1), 199–204. doi:10.2307/2347696

Berger, R. (1991) Algorithm AS R86: A Remark on Algorithm AS 152: Cumulative Hypergeometric Probabilities. *JRSS C (Applied Statistics)*, **40**(2), 374–375. doi:10.2307/2347606

See Also

phyper

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (q, m, n, k)
{
    kk <- n
    nn <- m
    mm <- m + n
    ll <- q
    mean <- kk * nn/mm
    sig <- sqrt(mean * (mm - nn)/mm * (mm - kk)/(mm - 1))
    pnorm(ll + 1/2, mean = mean, sd = sig)
  }
```

---

phyperBin *HyperGeometric Distribution via Approximate Binomial Distribution*

Description

Compute hypergeometric cumulative probabilities via (good) binomial distribution approximations. The arguments of these functions are *exactly* those of R's own phyper().

Usage

```
phyperBin.1(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
phyperBin.2(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
phyperBin.3(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
phyperBin.4(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
```

Arguments

q	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
m	the number of white balls in the urn.
n	the number of black balls in the urn.
k	the number of balls drawn from the urn, hence must be in $0, 1, \ldots, m + n$.
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).

Details

TODO

Value

a numeric vector, with the length the maximum of the lengths of q, m, n, k.

Author(s)

Martin Maechler

See Also

phyper, pbinom

Examples

```
## The 1st function is
function (q, m, n, k, lower.tail = TRUE, log.p = FALSE)
  pbinom(q, size = k, prob = m/(m + n), lower.tail = lower.tail,
         log.p = log.p)
```

---

phyperBinMolenaar *HyperGeometric Distribution via Molenaar's Binomial Approximation*

Description

Compute hypergeometric cumulative probabilities via Molenaar's binomial approximations. The arguments of these functions are *exactly* those of R's own phyper().

Usage

```
phyperBinMolenaar.1(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
phyperBinMolenaar.2(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
phyperBinMolenaar.3(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
phyperBinMolenaar.4(q, m, n, k, lower.tail = TRUE, log.p = FALSE)

phyperBinMolenaar  (q, m, n, k, lower.tail = TRUE, log.p = FALSE) # Deprecated !
```

Arguments

q	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
m	the number of white balls in the urn.
n	the number of black balls in the urn.
k	the number of balls drawn from the urn, hence must be in $0, 1, \ldots, m + n$.
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).

Details

Molenaar(1970), as cited in Johnson et al (1992), proposed `phyperBinMolenaar.1()`; the other three are just using the mathematical symmetries of the hyperbolic distribution, swapping k and m, and using `lower.tail = TRUE` or `FALSE`.

Value

a `numeric` vector, with the length the maximum of the lengths of `q, m, n, k`.

Author(s)

Martin Maechler

References

Johnson, N.L., Kotz, S. and Kemp, A.W. (1992) Univariate Discrete Distributions, 2nd ed.; Wiley, doi:10.1002/bimj.4710360207.
Chapter 6, mostly Section *5 Approximations and Bounds*, p.256 ff

Johnson, N.L., Kotz, S. and Kemp, A.W. (2005) Univariate Discrete Distributions, 3rd ed.; Wiley; doi:10.1002/0471715816.
Chapter 6, Section *6.5 Approximations and Bounds*, p.268 ff

See Also

`phyper`, the hypergeometric distribution, and R's own "exact" computation. `pbinom`, the binomial distribution functions.

Our utility `phyperAllBin()`.

Examples

```
## The first function is simply
function (q, m, n, k, lower.tail = TRUE, log.p = FALSE)
  pbinom(q, size = k, prob = hyper2binomP(q, m, n, k), lower.tail = lower.tail,
        log.p = log.p)
```

phyperIbeta *Pearson's incomplete Beta Approximation to the Hyperbolic Distribution*

Description

Pearson's incomplete Beta function approximation to the cumulative hyperbolic distribution function phyper(.).

Note that in R, pbeta() provides a version of the incomplete Beta function.

Usage

```
phyperIbeta(q, m, n, k)
```

Arguments

q	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
m	the number of white balls in the urn.
n	the number of black balls in the urn.
k	the number of balls drawn from the urn, hence must be in $0, 1, \ldots, m + n$.

Value

a numeric vector "like" q with values approximately equal to phyper(q,m,n,k).

Author(s)

Martin Maechler

References

Johnson, Kotz & Kemp (1992): (6.90), p.260 –> Bol'shev (1964)

See Also

phyper.

Examples

```
## The function is currently defined as
function (q, m, n, k)
{
    Np <- m
    N <- n + m
    n <- k
    x <- q
    p <- Np/N
```

```
  np <- n * p
  xi <- (n + Np - 1 - 2 * np)/(N - 2)
  d.c <- (N - n) * (1 - p) + np - 1
  cc <- n * (n - 1) * p * (Np - 1)/((N - 1) * d.c)
  lam <- (N - 2)^2 * np * (N - n) * (1 - p)/((N - 1) * d.c *
      (n + Np - 1 - 2 * np))
  pbeta(1 - xi, lam - x + cc, x - cc + 1)
}
```

---

phyperMolenaar *Molenaar's Normal Approximations to the Hypergeometric Distribution*

Description

Compute Molenaar's two normal approximations to the (cumulative hypergeometric distribution `phyper()`.

Usage

```
phyper1molenaar(q, m, n, k)
phyper2molenaar(q, m, n, k)
```

Arguments

`q`	(vector of) the number of white balls drawn without replacement from an urn which contains both black and white balls.
`m`	the number of white balls in the urn.
`n`	the number of black balls in the urn.
`k`	the number of balls drawn from the urn, hence in $0, 1, \ldots, m + n$.

Details

Both approximations are from page 261 of Johnson, Kotz & Kemp (1992). `phyper1molenaar` is formula (6.91), and `phyper2molenaar` is formula (6.92).

Value

a `numeric` vector, with the length the maximum of the lengths of `q`, `m`, `n`, `k`.

Author(s)

Martin Maechler

References

Johnson, Kotz & Kemp (1992): p.261

See Also

phyper, pnorm.

Examples

```
## TODO -- maybe see  ../tests/hyper-dist-ex.R
```

---

phyperPeizer *Peizer's Normal Approximation to the Cumulative Hyperbolic*

Description

Compute Peizer's extremely good normal approximation to the cumulative hyperbolic distribution.

This implementation corrects a typo in the reference.

Usage

```
phyperPeizer(q, m, n, k)
```

Arguments

q	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
m	the number of white balls in the urn.
n	the number of black balls in the urn.
k	the number of balls drawn from the urn, hence must be in $0, 1, \ldots, m + n$.

Value

a numeric vector, with the length the maximum of the lengths of q, m, n, k.

Author(s)

Martin Maechler

References

Johnson, Kotz & Kemp (1992): (6.93) & (6.94), p.261 *CORRECTED* by M.M.

See Also

phyper.

Examples

```
## The function is defined as

phyperPeizer <- function(q, m, n, k)
{
  ## Purpose: Peizer's extremely good Normal Approx. to cumulative Hyperbolic
  ##  Johnson, Kotz & Kemp (1992):  (6.93) & (6.94), p.261 __CORRECTED__
  ## ------------------------------------------------------------------------
  Np <- m; N <- n + m; n <- k; x <- q
  ## (6.94) -- in proper order!
  nn <- Np ;  n. <- Np     + 1/6
  mm <- N - Np                  ;  m. <- N - Np + 1/6
  r <- n                        ;  r. <- n      + 1/6
  s <- N - n                    ;  s. <- N - n  + 1/6
                                   N. <- N      - 1/6
  A <- x + 1/2                  ;  A. <- x      + 2/3
  B <- Np - x - 1/2             ;  B. <- Np - x - 1/3
  C <- n  - x - 1/2             ;  C. <- n  - x - 1/3
  D <- N - Np - n + x + 1/2     ;  D. <- N - Np - n + x + 2/3

  n <- nn
  m <- mm
  ## After (6.93):
  L <-
    A * log((A*N)/(n*r)) +
    B * log((B*N)/(n*s)) +
    C * log((C*N)/(m*r)) +
    D * log((D*N)/(m*s))
  ## (6.93) :
  pnorm((A.*D. - B.*C.) / abs(A*D - B*C) *
        sqrt(2*L* (m* n* r* s* N.)/
                  (m.*n.*r.*s.*N )))
  # The book wrongly has an extra "2*" before `m* ' (after "2*L* (" ) above
}
```

---

phyperR — R*-only version of* R*'s original phyper() algorithm*

Description

An R version of the first `phyper()` algorithm in R, which was used up to svn rev `30227` on 2004-07-09.

Usage

```
phyperR(q, m, n, k, lower.tail=TRUE, log.p=FALSE)
```

Arguments

q	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
m	the number of white balls in the urn.
n	the number of black balls in the urn.
k	the number of balls drawn from the urn, hence must be in $0, 1, \ldots, m + n$.
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).

Value

a numeric vector similar to `phyper(q, m, n, k)`.

Note

The original argument list in `C` was `(x, NR, NB, n)` where there were *red* and *black* balls in the urn.

Note that we have *vectorized* a translation to R of the original C code.

Author(s)

Martin Maechler

See Also

`phyper` and our `phyperR2()` for the pure R version of the newer (Welinder) `phyper()` algorithm

Examples

```
m <- 9:12; n <- 7:10; k <- 10
x <- 0:(k+1) # length 12
## confirmation that recycling + lower.tail, log.p now work:
for(lg in c(FALSE,TRUE))
  for(lt in c(FALSE, TRUE)) {
    cat("(lower.tail = ", lt, " -- log = ", lg,"):\n", sep="")
    withAutoprint({
      (rr <-
           cbind(x, m, n, k, # recycling (to 12 rows)
                 ph  = phyper (x, m, n, k, lower.tail=lt, log.p=lg),
                 phR = phyperR(x, m, n, k, lower.tail=lt, log.p=lg)))
      all.equal(rr[,"ph"], rr[,"phR"], tol = 0)
      ## saw   4.706e-15 1.742e-15 7.002e-12 1.086e-15  [x86_64 Lnx]
      stopifnot(all.equal(rr[,"ph"], rr[,"phR"],
                          tol = if(lg && !lt) 2e-11 else 2e-14))
    })
  }
```

phyperR2 *Pure R version of R's C level phyper()*

Description

Use pure R functions to compute (less efficiently and usually even less accurately) hypergeometric (point) probabilities with the same "Welinder"-algorithm as R's C level code has been doing since 2004.

Apart from boundary cases, each `phyperR2()` call uses one corresponding `pdhyper()` call.

Usage

```
phyperR2(q, m, n, k, lower.tail = TRUE, log.p = FALSE, ...)
pdhyper (q, m, n, k,                    log.p = FALSE,
         epsC = .Machine$double.eps, verbose = getOption("verbose"))
```

Arguments

`q`	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
`m`	the number of white balls in the urn.
`n`	the number of black balls in the urn.
`k`	the number of balls drawn from the urn, hence must be in $0, 1, \ldots, m + n$.
`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).
`...`	further arguments, passed to `pdhyper()`.
`epsC`	a non-negative number, the computer epsilon to be used; effectively a relative convergence tolerance for the `while()` loop in `pdhyper()`.
`verbose`	logical indicating if the `pdhyper()` calls, typically one per `phyperR2()` call, should show how many terms have been computed and summed up.

Value

a number (as `q`).

pdhyper(q, m,n,k) computes the ratio `phyper(q, m,n,k) / dhyper(q, m,n,k)` but without computing numerator or denominator explicitly.

phyperR2() (in the non-boundary cases) then just computes the product `dhyper(..) * pdhyper(..)`, of course "modulo" `lower.tail` and `log.p` transformations.

Consequently, it typically returns values very close to the corresponding R `phyper(q, m,n,k, ..)` call.

Note

For now, all arguments of these functions must be of length **one**.

Author(s)

Martin Maechler, based on R's C code originally provided by Morton Welinder from the Gnumeric project, who thanks Ian Smith for ideas.

References

Morten Welinder (2004) phyper accuracy and efficiency; R bug report PR#6772; https://bugs.r-project.org/show_bug.cgi?id=6772

See Also

phyper

Examples

```
## same example as phyper()
m <- 10; n <- 7; k <- 8
vapply(0:9, phyperR2, 0.1, m=m, n=n, k=k)  ==  phyper(0:9, m,n,k)
##  *all* TRUE (for 64b FC30)

## 'verbose=TRUE' to see the number of terms used:
vapply(0:9, phyperR2, 0.1, m=m, n=n, k=k,  verbose=TRUE)

## Larger arguments:
k <- 100 ; x <- .suppHyper(k,k,k)
ph  <- phyper (x, k,k,k)
ph1 <- phyperR(x, k,k,k) # ~ old R version
ph2 <- vapply(x, phyperR2, 0.1, m=k, n=k, k=k)
cbind(x, ph, ph1, ph2, rE1 = 1-ph1/ph, rE = 1-ph2/ph)
stopifnot(abs(1 -ph2/ph) < 8e-16) # 64bit FC30: see -2.22e-16 <= rE <= 3.33e-16

## Morten Welinder's example:
(p1R <- phyperR (59, 150, 150, 60, lower.tail=FALSE))
## gave 6.372680161e-14 in "old R";, here -1.04361e-14 (worse!!)
(p1x <-  dhyper ( 0, 150, 150, 60))# is 5.111204798e-22.
(p1N <- phyperR2(59, 150, 150, 60, lower.tail=FALSE)) # .. "perfect"
(p1. <- phyper  (59, 150, 150, 60, lower.tail=FALSE))# R's own
all.equal(p1x, p1N, tol=0) # on Lnx even perfectly
all.equal(p1x, p1., tol=0) # on Lnx even perfectly
```

---

phypers *The Four (4) Symmetric 'phyper()' Calls*

Description

Compute the four (4) symmetric phyper() calls which mathematically would be identical but in practice typically slightly differ numerically.

Usage

```
phypers(m, n, k, q = .suppHyper(m, n, k), tol = sqrt(.Machine$double.eps))
```

Arguments

m	the number of white balls in the urn.
n	the number of black balls in the urn.
k	the number of balls drawn from the urn, hence must be in $0, 1, \ldots, m + n$.
q	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls. By default all "non-trivial" abscissa values i.e., for which the mathematical value is strictly inside $(0, 1)$.
tol	a non-negative number, the tolerance for the all.equal() checks.

Value

a list with components

q	Description of 'comp1'
phyp	a numeric matrix of 4 columns with the 4 different calls to phyper() which are theoretically equivalent because of mathematical symmetry.

Author(s)

Martin Maechler

References

Johnson et al

See Also

R's phyper. In package **DPQmpfr**, phyperQ() uses (package **gmp** based) exact rational arithmetic, summing up dhyperQ(), terms computed by chooseZ(), exact (long integer) arithmetic binomial coefficients.

Examples

```
## The function is defined as
function(m,n,k, q = .suppHyper(m,n,k), tol = sqrt(.Machine$double.eps)) {
    N <- m+n
    pm <- cbind(ph = phyper(q,     m,  n , k), # 1 = orig.
                p2 = phyper(q,     k, N-k, m), # swap m <-> k (keep N = m+n)
                ## "lower.tail = FALSE"  <==>  1 - p..(..)
                Ip2= phyper(m-1-q, N-k, k, m, lower.tail=FALSE),
                Ip1= phyper(k-1-q, n,   m, k, lower.tail=FALSE))

    ## check that all are (approximately) the same :
    stopifnot(all.equal(pm[,1], pm[,2], tolerance=tol),
```

```
              all.equal(pm[,2], pm[,3], tolerance=tol),
              all.equal(pm[,3], pm[,4], tolerance=tol))
    list(q = q, phyp = pm)
}


str(phs <- phypers(20, 47, 31))
with(phs, cbind(q, phyp))
with(phs,
     matplot(q, phyp, type = "b"), main = "phypers(20, 47, 31)")

## differences:
with(phs, phyp[,-1] - phyp[,1])
## *relative*
relE <- with(phs, { phM <- rowMeans(phyp); 1 - phyp/phM })
print.table(cbind(q = phs$q, relE / .Machine$double.eps), zero.print = ".")
```

---

pl2curves *Plot 2 Noncentral Distribution Curves for Visual Comparison*

Description

Plot two noncentral (chi-squared or t or ..) distribution curves for visual comparison.

Usage

```
pl2curves(fun1, fun2, df, ncp, log = FALSE,
          from = 0, to = 2 * ncp, p.log = "", n = 2001,
          leg = TRUE, col2 = 2, lwd2 = 2, lty2 = 3, ...)
```

Arguments

fun1, fun2	function()s, both to be used via curve(), and called with the same 4 arguments, (., df, ncp, log) (the name of the first argument is not specified).
df, ncp, log	parameters to be passed and used in both functions, which hence typically are non-central chi-squared or t density, probability or quantile functions.
from, to	numbers determining the x-range, passed to curve().
p.log	string, passed as curve(...., log = log.p).
n	the number of evaluation points, passed to curve().
leg	logical specifying if a legend() should be drawn.
col2, lwd2, lty2	color, line width and line type for the second curve. (The first curve uses defaults for these graphical properties.)
...	further arguments passed to *first* curve(..) call.

Value

TODO: inivisible return both curve() results, i.e., (x,y1, y2), possibly as data frame

Author(s)

Martin Maechler

See Also

curve, ..

Examples

```
p.dnchiBessel <- function(df, ncp, log=FALSE, from=0, to = 2*ncp, p.log="", ...)
{
    pl2curves(dnchisqBessel, dchisq, df=df, ncp=ncp, log=log,
              from=from, to=to, p.log=p.log, ...)
}

  ## TODO the p.dnchiB()  examples  >>>>>> ../tests/chisq-nonc-ex.R <<<
```

---

pnbeta *Noncentral Beta Probabilities*

Description

pnbetaAppr2() and its inital version pnbetaAppr2v1() provide the "approximation 2" of Chattamvelli and Shanmugam(1997) to the noncentral Beta probability distribution.

pnbetaAS310() is an R level interface to a C translation (and "Rification") of the AS 310 Fortran implementation.

Usage

```
pnbetaAppr2(x, a, b, ncp = 0, lower.tail = TRUE, log.p = FALSE)

pnbetaAS310(x, a, b, ncp = 0, lower.tail = TRUE, log.p = FALSE,
            useAS226 = (ncp < 54.),
            errmax = 1e-6, itrmax = 100)
```

Arguments

x	numeric vector (of quantiles), typically from inside $[0, 1]$.
a, b	the shape parameters of Beta, aka as shape1 and shape2.
ncp	non-centrality parameter.
log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
useAS226	logical specifying if AS 226 (with R84 and R95 amendments) should be used which is said to be sufficient for small ncp. The default ncp < 54 had been hardwired in AS 310.
errmax	non-negative number determining convergence for AS 310.
itrmax	positive integer number, only if(useAS226) is passed to AS 226.

Value

a numeric vector of (log) probabilities of the same length as `x`.

Note

The authors in the reference compare AS 310 with Lam(1995), Frick(1990) and Lenth(1987) and state to be better than them. R's current (2019) noncentral beta implementation builds on these, too, with some amendments though; still, `pnbetaAS310()` may potentially be better, at least in certain corners of the 4-dimensional input space.

Author(s)

Martin Maechler; `pnbetaAppr2()` in Oct 2007.

References

– not yet implemented –
Gil, A., Segura, J., and Temme, N. M. (2019) On the computation and inversion of the cumulative noncentral beta distribution function. *Applied Mathematics and Computation* **361**, 74–86; doi:10.1016/j.amc.2019.05.014 . Chattamvelli, R., and Shanmugam, R. (1997) Algorithm AS 310: Computing the Non-Central Beta Distribution Function. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* **46**(1), 146–156, for "approximation 2" notably p.154; doi:10.1111/14679876.00055 .

Lenth, R. V. (1987) Algorithm AS 226, . . . , Frick, H. (1990)'s AS R84, . . . , and Lam, M.L. (1995)'s AS R95 : See 'References' in R's `pbeta` page.

See Also

R's own `pbeta`.

Examples

```
## Same arguments as for Table 1 (p.151) of the reference
a <- 5*rep(1:3, each=3)
aargs <- cbind(a = a, b = a,
               ncp = rep(c(54, 140, 170), 3),
               x = 1e-4*c(8640, 9000, 9560, 8686, 9000, 9000, 8787, 9000, 9220))
aargs
pnbA2 <- apply(aargs, 1, function(aa) do.call(pnbetaAppr2, as.list(aa)))
pnA310<- apply(aargs, 1, function(aa) do.call(pnbetaAS310, as.list(aa)))
aar2 <- aargs; dimnames(aar2)[[2]] <- c(paste0("shape", 1:2), "ncp", "q")
pnbR  <- apply(aar2,  1, function(aa) do.call(pbeta, as.list(aa)))
range(relD2   <- 1 - pnbA2 /pnbR)
range(relD310 <- 1 - pnA310/pnbR)
cbind(aargs, pnbA2, pnA310, pnbR,
      relD2 = signif(relD2, 3), relD310 = signif(relD310, 3)) # <------> Table 1
stopifnot(abs(relD2)   < 0.009) # max is 0.006286
stopifnot(abs(relD310) < 1e-5 ) # max is 6.3732e-6

## Arguments as for Table 2 (p.152) of the reference :
```

```
aarg2 <- cbind(a = c( 10, 10, 15, 20, 20, 20, 30, 30),
               b = c( 20, 10,  5, 10, 30, 50, 20, 40),
               ncp=c(150,120, 80,110, 65,130, 80,130),
               x = c(868,900,880,850,660,720,720,800)/1000)
pnbA2 <- apply(aarg2, 1, function(aa) do.call(pnbetaAppr2, as.list(aa)))
pnA310<- apply(aarg2, 1, function(aa) do.call(pnbetaAS310, as.list(aa)))
aar2 <- aarg2; dimnames(aar2)[[2]] <- c(paste0("shape", 1:2), "ncp", "q")
pnbR  <- apply(aar2,  1, function(aa) do.call(pbeta, as.list(aa)))
range(relD2   <- 1 - pnbA2 /pnbR)
range(relD310 <- 1 - pnA310/pnbR)
cbind(aarg2, pnbA2, pnA310, pnbR,
      relD2 = signif(relD2, 3), relD310 = signif(relD310, 3)) # <------> Table 2
stopifnot(abs(relD2  ) < 0.006) # max is 0.00412
stopifnot(abs(relD310) < 1e-5 ) # max is 5.5953e-6

## Arguments as for Table 3 (p.152) of the reference :
aarg3 <- cbind(a = c( 10, 10, 10, 15, 10, 12, 30, 35),
               b = c(  5, 10, 30, 20,  5, 17, 30, 30),
               ncp=c( 20, 54, 80,120, 55, 64,140, 20),
               x = c(644,700,780,760,795,560,800,670)/1000)
pnbA3 <- apply(aarg3, 1, function(aa) do.call(pnbetaAppr2, as.list(aa)))
pnA310<- apply(aarg3, 1, function(aa) do.call(pnbetaAS310, as.list(aa)))
aar3 <- aarg3; dimnames(aar3)[[2]] <- c(paste0("shape", 1:2), "ncp", "q")
pnbR  <- apply(aar3,  1, function(aa) do.call(pbeta, as.list(aa)))
range(relD2   <- 1 - pnbA3 /pnbR)
range(relD310 <- 1 - pnA310/pnbR)
cbind(aarg3, pnbA3, pnA310, pnbR,
      relD2 = signif(relD2, 3), relD310 = signif(relD310, 3)) # <------> Table 3
stopifnot(abs(relD2  ) < 0.09) # max is 0.06337
stopifnot(abs(relD310) < 1e-4) # max is 3.898e-5
```

`pnchi1sq` *(Probabilities of Non-Central Chi-squared Distribution for Special Cases*

Description

Computes probabilities for the non-central chi-squared distribution, in special cases, currently for `df = 1` and `df = 3`, using 'exact' formulas only involving the standard normal (Gaussian) cdf $\Phi()$ and its derivative $\phi()$, i.e., R's `pnorm()` and `dnorm()`.

Usage

```
pnchi1sq(q, ncp = 0, lower.tail = TRUE, log.p = FALSE, epsS = .01)
pnchi3sq(q, ncp = 0, lower.tail = TRUE, log.p = FALSE, epsS = .04)
```

Arguments

`q`	number ('quantile', i.e., abscissa value.)

ncp	non-centrality parameter δ; ....
lower.tail, log.p	logical, see, e.g., pchisq().
epsS	small number, determining where to switch from the "small case" to the regular case, namely by defining small <- sqrt(q/ncp) <= epsS.

Details

In the "small case" (epsS above), the direct formulas suffer from cancellation, and we use Taylor series expansions in $s := \sqrt{q}$, which in turn use "probabilists"' Hermite polynomials $He_n(x)$.

The default values epsS have currently been determined by experiments as those in the 'Examples' below.

Value

a numeric vector "like" q+ncp, i.e., recycled to common length.

Author(s)

Martin Maechler, notably the Taylor approximations in the "small" cases.

References

Johnson et al.(1995), see 'References' in pnchisqPearson.

https://en.wikipedia.org/wiki/Hermite_polynomials for the notation.

See Also

pchisq, the (simple and R-like) approximations, such as pnchisqPearson and the wienergerm approximations, pchisqW() etc.

Examples

```
qq <- seq(9500, 10500, length=1000)
m1 <- cbind(pch = pchisq  (qq, df=1, ncp = 10000),
            p1  = pnchi1sq(qq,       ncp = 10000))
matplot(qq, m1, type = "l"); abline(h=0:1, v=10000+1, lty=3)
all.equal(m1[,"p1"], m1[,"pch"], tol=0) # for now,  2.37e-12

m3 <- cbind(pch = pchisq  (qq, df=3, ncp = 10000),
             p3 = pnchi3sq(qq,       ncp = 10000))
matplot(qq, m3, type = "l"); abline(h=0:1, v=10000+3, lty=3)
all.equal(m3[,"p3"], m3[,"pch"], tol=0) # for now,  1.88e-12

stopifnot(exprs = {
  all.equal(m1[,"p1"], m1[,"pch"], tol=1e-10)
  all.equal(m3[,"p3"], m3[,"pch"], tol=1e-10)
})

### Very small 'x' i.e., 'q' would lead to cancellation: -----------
```

```
##  df = 1 ------------------------------------------------------

qS <- c(0, 2^seq(-40,4, by=1/16))
m1s <- cbind(pch = pchisq  (qS, df=1, ncp = 1)
           , p1.0= pnchi1sq(qS,       ncp = 1, epsS = 0)
           , p1.4= pnchi1sq(qS,       ncp = 1, epsS = 1e-4)
           , p1.3= pnchi1sq(qS,       ncp = 1, epsS = 1e-3)
           , p1.2= pnchi1sq(qS,       ncp = 1, epsS = 1e-2)
        )
cols <- adjustcolor(1:5, 1/2); lws <- seq(4,2, by = -1/2)
abl.leg <- function(x.leg = "topright", epsS = 10^-(4:2), legend = NULL)
{
   abline(h = .Machine$double.eps, v = epsS^2,
          lty = c(2,3,3,3), col= adjustcolor(1, 1/2))
   if(is.null(legend))
     legend <- c(quote(epsS == 0), as.expression(lapply(epsS,
                            function(K) substitute(epsS == KK,
                                                   list(KK = formatC(K, w=1))))))
   legend(x.leg, legend, lty=1:4, col=cols, lwd=lws, bty="n")
}
matplot(qS, m1s, type = "l", log="y" , col=cols, lwd=lws)
matplot(qS, m1s, type = "l", log="xy", col=cols, lwd=lws) ; abl.leg("right")
## ====  "Errors" ===================================================
## Absolute: -------------------------
matplot(qS,     m1s[,1] - m1s[,-1] , type = "l", log="x" , col=cols, lwd=lws)
matplot(qS, abs(m1s[,1] - m1s[,-1]), type = "l", log="xy", col=cols, lwd=lws)
abl.leg("bottomright")
rbind(all     = range(aE1e2 <- abs(m1s[,"pch"] - m1s[,"p1.2"])),
      less.75 = range(aE1e2[qS <= 3/4]))
##            Lnx(F34;i7)  M1mac(BDR)
## all        0 7.772e-16  1.110e-15
## less.75    0 1.665e-16  2.220e-16
stopifnot(aE1e2[qS <= 3/4] <= 4e-16, aE1e2 <= 2e-15) # check
## Relative: -------------------------
matplot(qS,     1 - m1s[,-1]/m1s[,1] , type = "l", log="x",  col=cols, lwd=lws)
abl.leg()
matplot(qS, abs(1 - m1s[,-1]/m1s[,1]), type = "l", log="xy", col=cols, lwd=lws)
abl.leg()
## number of correct digits ('Inf' |--> 17) :
corrDigs <- pmin(round(-log10(abs(1 - m1s[,-1]/m1s[,1])[-1,]), 1), 17)
table(corrDigs > 9.8) # all
range(corrDigs[qS[-1] > 1e-8,  1 ], corrDigs[, 2:4]) # [11.8 , 17]
(min (corrDigs[qS[-1] > 1e-6, 1:2], corrDigs[, 3:4]) -> mi6) # 13
(min (corrDigs[qS[-1] > 1e-4, 1:3], corrDigs[,   4]) -> mi4) # 13.9
stopifnot(exprs = {
   corrDigs >= 9.8
   c(corrDigs[qS[-1] > 1e-8,  1 ], corrDigs[, 2]) >= 11.5
   mi6 >= 12.7
   mi4 >= 13.6
})

##  df = 3 -------------- NOTE:  epsS=0 for small qS is "non-sense" --------
```

```
qS <- c(0, 2^seq(-40,4, by=1/16))
ee <- c(1e-3, 1e-2, .04)
m3s <- cbind(pch = pchisq  (qS, df=3, ncp = 1)
           , p1.0= pnchi3sq(qS,       ncp = 1, epsS = 0)
           , p1.3= pnchi3sq(qS,       ncp = 1, epsS = ee[1])
           , p1.2= pnchi3sq(qS,       ncp = 1, epsS = ee[2])
           , p1.1= pnchi3sq(qS,       ncp = 1, epsS = ee[3])
        )
matplot(qS, m3s, type = "l", log="y" , col=cols, lwd=lws)
matplot(qS, m3s, type = "l", log="xy", col=cols, lwd=lws); abl.leg("right", ee)
## ====  "Errors" ================================================
## Absolute: -------------------------
matplot(qS,     m3s[,1] - m3s[,-1] , type = "l", log="x" , col=cols, lwd=lws)
matplot(qS, abs(m3s[,1] - m3s[,-1]), type = "l", log="xy", col=cols, lwd=lws)
abl.leg("right", ee)
## Relative: -------------------------
matplot(qS,     1 - m3s[,-1]/m3s[,1] , type = "l", log="x",  col=cols, lwd=lws)
abl.leg(, ee)
matplot(qS, abs(1 - m3s[,-1]/m3s[,1]), type = "l", log="xy", col=cols, lwd=lws)
abl.leg(, ee)
```

---

pnchisqAppr *(Approximate) Probabilities of Non-Central Chi-squared Distribution*

Description

Compute (approximate) probabilities for the non-central chi-squared distribution.

The non-central chi-squared distribution with `df`$= n$ degrees of freedom and non-centrality parameter `ncp` $= \lambda$ has density

$$f(x) = f_{n,\lambda}(x) = e^{-\lambda/2} \sum_{r=0}^{\infty} \frac{(\lambda/2)^r}{r!} f_{n+2r}(x)$$

for $x \geq 0$, where $f_m(x)$ (= `dchisq(x,m)`) is the (*central* chi-squared density with m degrees of freedom; for more, see R's help page for `pchisq`.

- R's own historical and current versions, but with more tuning parameters;

Historical relatively simple approximations listed in Johnson, Kotz, and Balakrishnan (1995):

- Patnaik(1949)'s approximation to the non-central via central chi-squared. Is also the formula 26.4.27 in Abramowitz & Stegun, p.942. Johnson et al mention that the approximation error is $O(1/\sqrt(\lambda))$ *for* $\lambda \to \infty$.
- Pearson(1959) is using 3 moments instead of 2 as Patnaik (to approximate via a central chi-squared), and therefore better than Patnaik for the right tail; further (in Johnson et al.), the approximation error is $O(1/\lambda)$ *for* $\lambda \to \infty$.
- Abdel-Aty(1954)'s "first approximation" based on Wilson-Hilferty via Gaussian (`pnorm`) probabilities, is partly *wrongly* cited in Johnson et al., p.463, eq.$(29.61a)$.

- Bol'shev and Kuznetzov (1963) concentrate on the case of **small** ncp λ and provide an "approximation" via *central* chi-squared with the same degrees of freedom df, but a modified q ('x'); the approximation has error $O(\lambda^3)$ for $\lambda \to 0$ and is from Johnson et al., p.465, eq.(29.62) and (29.63).
- Sankaran(1959, 1963) proposes several further approximations base on Gaussian probabilities, according to Johnson et al., p.463. pnchisqSankaran_d() implements its formula $(29.61d)$.

pnchisq(): an R implementation of R's own C pnchisq_raw(), but almost only up to Feb.27, 2004, long before the log.p=TRUE addition there, including *logspace arithmetic* in April 2014, its finish on 2015-09-01. Currently for historical reference only.

pnchisqV(): a Vectorize()d pnchisq.

pnchisqRC(): R's C implementation as of Aug.2019; but with many more options. Currently extreme cases tend to hang on Winbuilder (?)

pnchisqIT: using C code "paralleling" R's own, returns a list containing the full vector of terms computed (and the resulting probability).

pnchisqT93: pure R implementations of approximations when both q and ncp are large, by Temme(1993), from Johnson et al., p.467, formulas $(29.71a)$, and $(29.71b)$, using auxiliary functions pnchisqT93a() and pnchisqT93b() respectively, with adapted formulas for the log.p=TRUE cases.

pnchisq_ss(): based on ss(), shows pure R code to sum up the non-central chi-squared distribution value.

ss: pure R function providing all the summation terms making up non-central chi-squared distribution values; terms computed carefully only using simple arithmetic plus exp() and log(), via cumsum and cumprod and possibly in log space.

pnchisqTerms: pure R providing these terms *directly*, calling (central) pchisq(x, df + 2*(k-1)) for the whole length vector k <- 1:i.max.

ss2(): computes "statistics" on ss(), whereas

ss2.(): uses pnchisqIT()'s C code providing similar stats as ss2().

Usage

```
pnchisq          (q, df, ncp = 0, lower.tail = TRUE,
                 cutOffncp = 80, itSimple = 110, errmax = 1e-12, reltol = 1e-11,
                  maxit = 10* 10000, verbose = 0, xLrg.sigma = 5)
pnchisqV(x, ..., verbose = 0)

pnchisqRC        (q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE,
                  no2nd.call = FALSE,
                  cutOffncp = 80, small.ncp.logspace = small.ncp.logspaceR2015,
                  itSimple = 110, errmax = 1e-12,
                 reltol = 8 * .Machine$double.eps, epsS = reltol/2, maxit = 1e6,
                  verbose = FALSE)
pnchisqAbdelAty  (q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
pnchisqBolKuz    (q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
pnchisqPatnaik   (q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
pnchisqPearson   (q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
```

```
pnchisqSankaran_d(q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
pnchisq_ss      (x, df, ncp = 0, lower.tail = TRUE, log.p = FALSE, i.max = 10000,
                 ssr = ss(x=x, df=df, ncp=ncp, i.max = i.max))

pnchisqT93  (q, df, ncp, lower.tail = TRUE, log.p = FALSE, use.a = q > ncp)
pnchisqT93.a(q, df, ncp, lower.tail = TRUE, log.p = FALSE)
pnchisqT93.b(q, df, ncp, lower.tail = TRUE, log.p = FALSE)

pnchisqTerms     (x, df, ncp,     lower.tail = TRUE, i.max = 1000)
ss   (x, df, ncp, i.max = 10000, useLv = !(expMin < -lambda && 1/lambda < expMax))
ss2  (x, df, ncp, i.max = 10000, eps = .Machine$double.eps)
ss2. (q, df, ncp = 0, errmax = 1e-12, reltol = 2 * .Machine$double.eps,
      maxit = 1e+05, eps = reltol, verbose = FALSE)
```

Arguments

x	numeric vector (of 'quantiles', i.e., abscissa values).
q	number ('quantile', i.e., abscissa value.)
df	degrees of freedom > 0, maybe non-integer.
ncp	non-centrality parameter δ.
lower.tail, log.p	logical, see, e.g., pchisq().
i.max	number of terms in evaluation ...
ssr	an ss()-like list to be used in pnchisq_ss().
use.a	logical vector for Temme pnchisqT93*() formulas, indicating to use formula 'a' over 'b'. The default is as recommended in the references, but they did not take into account log.p = TRUE situations.
cutOffncp	a positive number, the cutoff value for ncp at which the algorithm branches; has been hard coded to 80 in R.
itSimple	positive integer, the maximal number of "simple" iterations; has been hard coded to 110 in R.
errmax	absolute error tolerance.
reltol	convergence tolerance for *rel*ative error.
maxit	maximal number of iterations.
xLrg.sigma	positive number ...
no2nd.call	logical indicating if a 2nd call is made to the internal function pnchisq_raw(), swapping lower.tail.
small.ncp.logspace	logical vector or function, indicating if the logspace computations for "small" ncp (defined to fulfill ncp < cutOffncp !).
epsS	small positive number, the convergence tolerance of the 'simple' iterations; has been hard coded to 1e-15 in R.
verbose	logical or integer specifying if or how much the algorithm progress should be monitored.

...	further arguments passed from pnchisqV() to pnchisq().
useLv	logical indicating if logarithmic scale should be used for λ computations.
eps	convergence tolerance, a positive number.

Details

pnchisq_ss() uses si <- ss(x, df, ..) to get the series terms, and returns 2*dchisq(x, df = df +2) * sum(si$s).

ss() computes the terms needed for the expansion used in pnchisq_ss() only using arithmetic, exp() and log().

ss2() computes some simple "statistics" about ss(..).

Value

ss() returns a list with 3 components

s	the series
i1	location (in s[]) of the first change from 0 to positive.
max	(first) location of the maximal value in the series (i.e., which.max(s)).

Author(s)

Martin Maechler, from May 1999; starting from a post to the S-news mailing list by Ranjan Maitra (@ math.umbc.edu) who showed a version of our pchisqAppr.0() thanking Jim Stapleton for providing it.

References

Johnson, N.L., Kotz, S. and Balakrishnan, N. (1995) Continuous Univariate Distributions Vol 2, 2nd ed.; Wiley; chapter 29 *Noncentral χ^2-Distributions*; notably Section *8 Approximations*, p.461 ff.

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun

See Also

pchisq and the wienergerm approximations for it: pchisqW() etc.

r_pois() and its plot function, for an aspect of the series approximations we use in pnchisq_ss().

Examples

```
## set of quantiles to use :
qq <- c(.001, .005, .01, .05, (1:9)/10, 2^seq(0, 10, by= 0.5))
## Take "all interesting" pchisq-approximation from our pkg :
pkg <- "package:DPQ"
pnchNms <- c(paste0("pchisq", c("V", "W", "W.", "W.R")),
             ls(pkg, pattern = "^pnchisq"))
pnchNms <- pnchNms[!grepl("Terms$", pnchNms)]
```

```
pnchF <- sapply(pnchNms, get, envir = as.environment(pkg))
str(pnchF)
ncps <- c(0, 1/8, 1/2)
pnchR <- as.list(setNames(ncps, paste("ncp",ncps, sep="=")))
for(i.n in seq_along(ncps)) {
  ncp <- ncps[i.n]
  pnF <- if(ncp == 0) pnchF[!grepl("chisqT93", pnchNms)] else pnchF
  pnchR[[i.n]] <- sapply(pnF, function(F)
            Vectorize(F, names(formals(F))[[1]])(qq, df = 3, ncp=ncp))
}
str(pnchR, max=2)


## A case where the non-central P[] should be improved :
## First, the central P[] which is close to exact -- choosing df=2 allows
## truly exact values: chi^2 = Exp(1) !
opal <- palette()
palette(c("black", "red", "green3", "blue", "cyan", "magenta", "gold3", "gray44"))
cR  <- curve(pchisq   (x, df=2,        lower.tail=FALSE, log.p=TRUE), 0, 4000, n=2001)
cRC <- curve(pnchisqRC(x, df=2, ncp=0, lower.tail=FALSE, log.p=TRUE),
             add=TRUE, col=adjustcolor(2,1/2), lwd=10, lty=4, n=2001)
cR0 <- curve(pchisq   (x, df=2, ncp=0, lower.tail=FALSE, log.p=TRUE),
             add=TRUE, col=adjustcolor(3,1/2), lwd=12,        n=2001)
## cRC & cR0 jump to -Inf much too early:
pchisq(1492, df=2, ncp=0, lower.tail=FALSE,log.p=TRUE) #--> -Inf *wrongly*
##' list_() := smart "named list" constructor
list_ <- function(...)
   `names<-`(list(...), vapply(sys.call()[-1L], as.character, ""))
JKBfn <-list_(pnchisqPatnaik,
              pnchisqPearson,
              pnchisqAbdelAty,
              pnchisqBolKuz,
              pnchisqSankaran_d)
cl. <- setNames(adjustcolor(3+seq_along(JKBfn), 1/2), names(JKBfn))
lw. <- setNames(2+seq_along(JKBfn),                   names(JKBfn))
cR.JKB <- sapply(names(JKBfn), function(nmf) {
  curve(JKBfn[[nmf]](x, df=2, ncp=0, lower.tail=FALSE, log.p=TRUE),
        add=TRUE, col=cl.[[nmf]], lwd=lw.[[nmf]], lty=lw.[[nmf]], n=2001)$y
})
legend("bottomleft", c("pchisq", "pchisq.ncp=0", "pnchisqRC", names(JKBfn)),
       col=c(palette()[1], adjustcolor(2:3,1/2), cl.),
       lwd=c(1,10,12, lw.), lty=c(1,4,1, lw.))
palette(opal)# revert
##
## the problematic "jump" :
as.data.frame(cRC)[744:750,]
cRall <- cbind(as.matrix(as.data.frame(cR)), R0 = cR0$y, RC = cRC$y, cR.JKB)
cbind(r1492 <- local({ r1 <- cRall[cRall[,"x"] == 1492, ]; r1[1+c(0, order(r1[-1]))] }))
cbind(r1492[-1] - r1492[["y"]])
## ==> Patnaik, Pearson, BolKuz are perfect for this x.
all.equal(cRC, cR0, tol = 1e-15) # TRUE [for now]
if(.Platform$OS.type == "unix")
  ## verbose=TRUE  may reveal which branches of the algorithm are taken:
```

```
  pnchisqRC(1500, df=2, ncp=0, lower.tail=FALSE, log.p=TRUE, verbose=TRUE) #
  ## |-->  -Inf currently

## The *two*  principal cases (both lower.tail = {TRUE,FALSE} !), where
##  "2nd call"  happens *and* is currently beneficial :
dfs <- c(1:2, 5, 10, 20)
pL. <- pnchisqRC(.00001, df=dfs, ncp=0, log.p=TRUE, lower.tail=FALSE, verbose = TRUE)
pR. <- pnchisqRC(   100, df=dfs, ncp=0, log.p=TRUE,                   verbose = TRUE)
## R's own non-central version (specifying 'ncp'):
pL0 <- pchisq   (.00001, df=dfs, ncp=0, log.p=TRUE, lower.tail=FALSE)
pR0 <- pchisq   (   100, df=dfs, ncp=0, log.p=TRUE)
## R's *central* version, i.e., *not* specifying 'ncp' :
pL  <- pchisq   (.00001, df=dfs,        log.p=TRUE, lower.tail=FALSE)
pR  <- pchisq   (   100, df=dfs,        log.p=TRUE)
cbind(pL., pL, relEc = signif(1-pL./pL, 3), relE0 = signif(1-pL./pL0, 3))
cbind(pR., pR, relEc = signif(1-pR./pR, 3), relE0 = signif(1-pR./pR0, 3))
```

`pnchisqWienergerm` *Wienergerm Approximations to (Non-Central) Chi-squared Probabilities*

Description

Functions implementing the two Wiener germ approximations to `pchisq()`, the (non-central) chi-squared distribution, and to `qchisq()` its inverse, the quantile function.

These have been proposed by Penev and Raykov (2000) who also listed a Fortran implementation.

In order to use them in numeric boundary cases, Martin Maechler has improved the original formulas.

Auxiliary functions:

`sW()`: The $s()$ as in the Wienergerm approximation, but using Taylor expansion when needed, i.e., `(x*ncp / df^2)` << 1.

`qs()`: ...

`z0()`: ...

`z.f()`: ...

`z.s()`: ...

....

Usage

```
pchisqW. (q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE,
          Fortran = TRUE, variant = c("s", "f"))
pchisqV  (q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE,
          Fortran = TRUE, variant = c("s", "f"))
pchisqW  (q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE, variant = c("s", "f"))
pchisqW.R(x, df, ncp = 0, lower.tail = TRUE, log.p = FALSE, variant = c("s", "f"),
```

```
          verbose = getOption("verbose"))

sW(x, df, ncp)
qs(x, df, ncp, f.s = sW(x, df, ncp), eps1 = 1/2, sMax = 1e+100)
z0(x, df, ncp)
z.f(x, df, ncp)
z.s(x, df, ncp, verbose = getOption("verbose"))
```

Arguments

q, x	vector of quantiles (main argument, see pchisq).
df	degrees of freedom (non-negative, but can be non-integer).
ncp	non-centrality parameter (non-negative).
lower.tail, log.p	logical, see pchisq.
variant	a character string, currently either "f" for the **f**irst or "s" for the **s**econd Wienergerm approximation in Penev and Raykov (2000).
Fortran	logical specifying if the Fortran or the C version should be used.
verbose	logical (or integer) indicating if or how much diagnostic output should be printed to the console during the computations.
f.s	a number must be a "version" of $s(x, df, ncp)$.
eps1	for qs(): use direct approximation instead of h(1 - 1/s) for s < eps1.
sMax	for qs(): cutoff to switch the $h(.)$ formula for s > sMax.

Details

....TODO... or write vignette

Value

all these functions return numeric vectors according to their arguments.

Note

The exact auxiliary function names etc, are still considered *provisional*; currently they are exported for easier documentation and use, but may well all disappear from the exported functions or even completely.

Author(s)

Martin Maechler, mostly end of Jan 2004

References

Penev, Spiridon and Raykov, Tenko (2000) A Wiener Germ approximation of the noncentral chi square distribution and of its quantiles. *Computational Statistics* **15**, 219–228. doi:10.1007/s001800000029

Dinges, H. (1989) Special cases of second order Wiener germ approximations. *Probability Theory and Related Fields*, **83**, 5–57.

See Also

pchisq, and other approximations for it: pnchisq() etc.

Examples

```
## see  example(pnchisqAppr)   which looks at all of the pchisq() approximating functions
```

---

pnormAsymp *Asymptotic Approxmation of (Extreme Tail) 'pnorm()'*

Description

Provide the first few terms of the asymptotic series approximation to pnorm()'s (extreme) tail, from Abramawitz and Stegun's 26.2.13 (p.932).

Usage

```
pnormAsymp(x, k, lower.tail = FALSE, log.p = FALSE)
```

Arguments

x	positive (at least non-negative) numeric vector.
lower.tail, log.p	logical, see, e.g., pnorm().
k	integer ≥ 0 indicating how many terms the approximation should use; currently $k \leq 5$.

Value

a numeric vector "as" x; see the examples, on how to use it with arbitrary precise mpfr-numbers from package **Rmpfr**.

Author(s)

Martin Maechler

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain.

See Also

pnormU_S53 for (also asymptotic) upper and lower bounds.

Examples

```
x <- c((2:10)*2, 25, (3:9)*10, (1:9)*100, (1:8)*1000, (2:4)*5000)
Px <- pnorm(x, lower.tail = FALSE, log.p=TRUE)
PxA <- sapply(setNames(0:5, paste("k =",0:5)),
              pnormAsymp, x=x, lower.tail = FALSE, log.p=TRUE)
## rel.errors :
signif(head( cbind(x, 1 - PxA/Px) , 20))

## Look more closely with high precision computations
if(requireNamespace("Rmpfr")) {
  ## ensure our function uses Rmpfr's dnorm(), etc:
  environment(pnormAsymp) <- asNamespace("Rmpfr")
  environment(pnormU_S53) <- asNamespace("Rmpfr")
  x. <- Rmpfr::mpfr(x, precBits=256)
  Px. <- Rmpfr::pnorm(x., lower.tail = FALSE, log.p=TRUE)
  ## manual, better sapplyMpfr():
  PxA. <- sapply(setNames(0:5, paste("k =",0:5)),
                 pnormAsymp, x=x., lower.tail = FALSE, log.p=TRUE)
  PxA. <- new("mpfrMatrix", unlist(PxA.), Dim=dim(PxA.), Dimnames=dimnames(PxA.))
  PxA2 <- Rmpfr::cbind(pn_dbl = Px, PxA.,
                       pnormU_S53 = pnormU_S53(x=x., lower.tail = FALSE, log.p=TRUE))
  ## rel.errors : note that pnormU_S53() is very slightly better than "k=2":
  print( Rmpfr::roundMpfr(Rmpfr::cbind(x., 1 - PxA2/Px.), precBits = 13), width = 111)
  pch <- c("R", 0:5, "U")
  matplot(x, abs(1 -PxA2/Px.), type="o", log="xy", pch=pch,
          main="pnorm(<tail>) approximations' relative errors - pnormAsymp(*, k=k)")
  legend("bottomleft", colnames(PxA2), col=1:6, pch=pch, lty=1:5, bty="n", inset=.01)
  at1 <- axTicks(1, axp = c(par("xaxp")[1:2], 3))
  axis(1, at=at1)
  abline(h = 1:2* 2^-53, v = at1, lty=3, col=adjustcolor("gray20", 1/2))
  axis(4, las=2, at= 2^-53, label = quote(epsilon[C]), col="gray20")
}
```

---

pnormLU *Bounds for 1-Phi(.) – Mill's Ratio related Bounds for pnorm()*

Description

Bounds for $1 - \Phi(x)$, i.e., `pnorm(x, *, lower.tail=FALSE)`, typically related to Mill's Ratio.

Usage

```
pnormL_LD10(x, lower.tail = FALSE, log.p = FALSE)
pnormU_S53 (x, lower.tail = FALSE, log.p = FALSE)
```

Arguments

`x`	positive (at least non-negative) numeric vector.
`lower.tail`, `log.p`	logical, see, e.g., `pnorm()`.

Value

a numeric vector like x

Author(s)

Martin Maechler

References

Lutz Duembgen (2010) *Bounding Standard Gaussian Tail Probabilities*; arXiv preprint 1012.2063, https://arxiv.org/abs/1012.2063

See Also

pnorm.

Examples

```
x <- seq(1/64, 10, by=1/64)
px <- cbind(
    lQ = pnorm      (x, lower.tail=FALSE, log.p=TRUE)
  , Lo = pnormL_LD10(x, lower.tail=FALSE, log.p=TRUE)
  , Up = pnormU_S53 (x, lower.tail=FALSE, log.p=TRUE))
matplot(x, px, type="l") # all on top of each other

matplot(x, (D <- px[,2:3] - px[,1]), type="l") # the differences
abline(h=0, lty=3, col=adjustcolor(1, 1/2))

## check they are lower and upper bounds indeed :
stopifnot(D[,"Lo"] < 0, D[,"Up"] > 0)

matplot(x[x>4], D[x>4,], type="l") # the differences
abline(h=0, lty=3, col=adjustcolor(1, 1/2))

### zoom out to larger x : [1, 1000]
x <- seq(1, 1000, by=1/4)
px <- cbind(
    lQ = pnorm      (x, lower.tail=FALSE, log.p=TRUE)
  , Lo = pnormL_LD10(x, lower.tail=FALSE, log.p=TRUE)
  , Up = pnormU_S53 (x, lower.tail=FALSE, log.p=TRUE))
matplot(x, px, type="l") # all on top of each other
matplot(x, (D <- px[,2:3] - px[,1]), type="l", log="x") # the differences
abline(h=0, lty=3, col=adjustcolor(1, 1/2))

## check they are lower and upper bounds indeed :
table(D[,"Lo"] < 0) # no longer always true
table(D[,"Up"] > 0)
## not even when equality (where it's much better though):
table(D[,"Lo"] <= 0)
table(D[,"Up"] >= 0)

## *relative* differences:
```

```
matplot(x, (rD <- 1 - px[,2:3] / px[,1]), type="l", log = "x")
abline(h=0, lty=3, col=adjustcolor(1, 1/2))
## abs()
matplot(x, abs(rD), type="l", log = "xy", axes=FALSE, # NB: curves *cross*
        main = "relative differences 1 - pnormUL(x, *)/pnorm(x,*)")
legend("top", c("Low.Bnd(D10)", "Upp.Bnd(S53)"), bty="n", col=1:2, lty=1:2)
sfsmisc::eaxis(1, sub10 = 2)
sfsmisc::eaxis(2)
abline(h=(1:4)*2^-53, col=adjustcolor(1, 1/4))

### zoom out to LARGE x : ---------------------------

x <- 2^seq(0,    30, by = 1/64)
if(FALSE)## or even HUGE:
   x <- 2^seq(4, 513, by = 1/16)
px <- cbind(
    lQ = pnorm      (x, lower.tail=FALSE, log.p=TRUE)
  , a0 = dnorm(x, log=TRUE)
  , a1 = dnorm(x, log=TRUE) - log(x)
  , Lo = pnormL_LD10(x, lower.tail=FALSE, log.p=TRUE)
  , Up = pnormU_S53 (x, lower.tail=FALSE, log.p=TRUE))
col4 <- adjustcolor(1:4, 1/2)
doLegTit <- function() {
  title(main = "relative differences 1 - pnormUL(x, *)/pnorm(x,*)")
  legend("top", c("phi(x)", "phi(x)/x", "Low.Bnd(D10)", "Upp.Bnd(S53)"),
         bty="n", col=col4, lty=1:4)
}
## *relative* differences are relevant:
matplot(x, (rD <- 1 - px[,-1] / px[,1]), type="l", log = "x",
            ylim = c(-1,1)/2^8, col=col4) ; doLegTit()
abline(h=0, lty=3, col=adjustcolor(1, 1/2))

## abs(rel.Diff)  ---> can use log-log:
matplot(x, abs(rD), type="l", log = "xy", xaxt="n", yaxt="n"); doLegTit()
sfsmisc::eaxis(1, sub10=2)
sfsmisc::eaxis(2, nintLog=12)
abline(h=(1:4)*2^-53, col=adjustcolor(1, 1/4))

## lower.tail=TRUE (w/ log.p=TRUE) works "the same" for x < 0:
x <- - 2^seq(0,    30, by = 1/64)
##   ==
px <- cbind(
    lQ = pnorm   (x, lower.tail=TRUE, log.p=TRUE)
  , a0 = log1mexp(- dnorm(-x, log=TRUE))
  , a1 = log1mexp(-(dnorm(-x, log=TRUE) - log(-x)))
  , Lo = log1mexp(-pnormL_LD10(-x, lower.tail=TRUE, log.p=TRUE))
  , Up = log1mexp(-pnormU_S53 (-x, lower.tail=TRUE, log.p=TRUE)) )
matplot(-x, (rD <- 1 - px[,-1] / px[,1]), type="l", log = "x",
            ylim = c(-1,1)/2^8, col=col4) ; doLegTit()
abline(h=0, lty=3, col=adjustcolor(1, 1/2))
```

pnt *Non-central t Probability Distribution - Algorithms and Approximations*

Description

Compute different approximations for the non-central t-Distribution cumulative probability distribution function.

Usage

```
pntR      (t, df, ncp, lower.tail = TRUE, log.p = FALSE,
           use.pnorm = (df > 4e5 ||
                        ncp^2 > 2*log(2)*1021), # .Machine$double.min.exp = -1022
                                  itrmax = 1000, errmax = 1e-12, verbose = TRUE)
pntR1     (t, df, ncp, lower.tail = TRUE, log.p = FALSE,
           use.pnorm = (df > 4e5 ||
                         ncp^2 > 2*log(2)*1021),
                                  itrmax = 1000, errmax = 1e-12, verbose = TRUE)

pntP94    (t, df, ncp, lower.tail = TRUE, log.p = FALSE,
                                  itrmax = 1000, errmax = 1e-12, verbose = TRUE)
pntP94.1  (t, df, ncp, lower.tail = TRUE, log.p = FALSE,
                                  itrmax = 1000, errmax = 1e-12, verbose = TRUE)

pnt3150   (t, df, ncp, lower.tail = TRUE, log.p = FALSE, M = 1000, verbose = TRUE)
pnt3150.1 (t, df, ncp, lower.tail = TRUE, log.p = FALSE, M = 1000, verbose = TRUE)

pntLrg    (t, df, ncp, lower.tail = TRUE, log.p = FALSE)

pntJW39   (t, df, ncp, lower.tail = TRUE, log.p = FALSE)
pntJW39.0 (t, df, ncp, lower.tail = TRUE, log.p = FALSE)


pntVW13 (t, df, ncp, lower.tail = TRUE, log.p = FALSE,
           keepS = FALSE, verbose = FALSE)

pntGST23_T6  (t, df, ncp, lower.tail = TRUE, log.p = FALSE,
              y1.tol = 1e-8, Mterms = 20, alt = FALSE, verbose = TRUE)
pntGST23_T6.1(t, df, ncp, lower.tail = TRUE, log.p = FALSE,
              y1.tol = 1e-8, Mterms = 20, alt = FALSE, verbose = TRUE)

## *Non*-asymptotic, (at least partly much) better version of R's Lenth(1998) algorithm
pntGST23_1(t, df, ncp, lower.tail = TRUE, log.p = FALSE,
           j0max = 1e4, # for now
           IxpqFUN = Ixpq,
           alt = FALSE, verbose = TRUE, ...)
```

Arguments

t	vector of quantiles (called q in pt(..)).
df	degrees of freedom (> 0, maybe non-integer). df = Inf is allowed.
ncp	non-centrality parameter $\delta \geq 0$; If omitted, use the central t distribution.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
use.pnorm	logical indicating if the pnorm() approximation of Abramowitz and Stegun (26.7.10) should be used, which is available as pntLrg(). The default corresponds to R pt()'s own behaviour (which is suboptimal).
itrmax	number of iterations / terms.
errmax	convergence bound for the iterations.
verbose	logical or integer determining the amount of diagnostic print out to the console.
M	positive integer specifying the number of terms to use in the series.
keepS	logical indicating if the function should return a list with component cdf and other informational elements, or just the CDF values directly (by default).
y1.tol	positive tolerance for warning if $y := t^2/(t^2 + df)$ is too close to 1 (as the formulas use $1/(1 - y)$).
Mterms	number of summation terms for pntGST23_T6().
j0max	*experimental*: large integer limiting the summation terms in pntGST23_1() .
IxpqFUN	the (scaled) incomplete beta function $I_x(p, q)$ to be used; currently, it defaults to the Ixpq function derived from Nico Temme's Maple code for "Table 1" in Gil et al. (2023).
alt	logical specifying if and how log-scale should be used. **Experimental** and not-yet-tested.
...	further arguments passed to IxpqFUN().

Details

pntR1(): a pure R version of the (C level) code of R's own pt(), additionally giving more flexibility (via arguments use.pnorm, itrmax, errmax whose defaults here have been hard-coded in R's C code called by pt()).

This implements an improved version of the AS 243 algorithm from Lenth(1989);

R's help on non-central pt() says: *This computes the lower tail only, so the upper tail suffers from cancellation and a warning will be given when this is likely to be significant.*

and (in 'Note:') *The code for non-zero* ncp *is principally intended to be used for moderate values of* ncp*: it will not be highly accurate, especially in the tails, for large values.*

pntR(): the Vectorize()d version of pntR1().

pntP94(), pntP94.1(): New versions of pntR1(), pntR(); using the Posten (1994) algorithm. pntP94() is the Vectorize()d version of pntP94.1().

`pnt3150()`, `pnt3150.1()`: Simple inefficient but hopefully correct version of pntP94..() This is really a direct implementation of formula (31.50), p.532 of Johnson, Kotz and Balakrishnan (1995)

`pntLrg()`: provides the `pnorm()` approximation (to the non-central t) from Abramowitz and Stegun (26.7.10), p.949; which should be employed only for *large* `df` and/or `ncp`.

`pntJW39.0()`: use the Jennett & Welch (1939) approximation see Johnson et al. (1995), p. 520, after (31.26a). This is still *fast* for huge `ncp` but has *wrong* asymptotic tail for $|t| \to \infty$. Crucially needs $b =$`b_chi(df)`.

`pntJW39()`: is an improved version of `pntJW39.0()`, using $1 - b =$`b_chi(df, one.minus=TRUE)` to avoid cancellation when computing $1 - b^2$.

`pntGST23_T6()`: (and `pntGST23_T6.1()` for informational purposes only) use the Gil et al.(2023)'s approximation of their Theorem 6.

`pntGST23_1()`: implements Gil et al.(2023)'s direct `pbeta()` based formula (1), which is very close to Lenth's algorithm.

`pntVW13()`: use MM's R translation of Viktor Witkowský (2013)'s matlab implementation.

Value

a number for `pntJKBf1()` and `.pntJKBch1()`.

a numeric vector of the same length as the maximum of the lengths of `x`, `df`, `ncp` for `pntJKBf()` and `.pntJKBch()`.

Author(s)

Martin Maechler

References

Johnson, N.L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions Vol~2*, 2nd ed.; Wiley; chapter 31, Section *5 Distribution Function*, p.514 ff

Lenth, R. V. (1989). *Algorithm AS 243* — Cumulative distribution function of the non-central t distribution, *JRSS C (Applied Statistics)* **38**, 185–189.

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover; formula (26.7.10), p.949

Posten, Harry O. (1994) A new algorithm for the noncentral t distribution function, *Journal of Statistical Computation and Simulation* **51**, 79–87; doi:10.1080/00949659408811623.

– not yet implemented –
Chattamvelli, R. and Shanmugam, R. (1994) An enhanced algorithm for noncentral t-distribution, *Journal of Statistical Computation and Simulation* **49**, 77–83. doi:10.1080/00949659408811561

– not yet implemented –
Akahira, Masafumi. (1995). A higher order approximation to a percentage point of the noncentral t distribution, *Communications in Statistics - Simulation and Computation* **24**:3, 595–605; doi:10.1080/03610919508813261

Michael Perakis and Evdokia Xekalaki (2003) On a Comparison of the Efficacy of Various Approximations of the Critical Values for Tests on the Process Capability Indices CPL, CPU, and

Cpk, *Communications in Statistics - Simulation and Computation* **32**, 1249–1264; doi:10.1081/SAC120023888

Witkovský, Viktor (2013) A Note on Computing Extreme Tail Probabilities of the Noncentral T Distribution with Large Noncentrality Parameter, *Acta Universitatis Palackianae Olomucensis, Facultas Rerum Naturalium, Mathematica* **52**(2), 131–143.

Gil A., Segura J., and Temme N.M. (2023) New asymptotic representations of the noncentral t-distribution, *Stud Appl Math.* **151**, 857–882; doi:10.1111/sapm.12609 ; acronym "GST23".

See Also

pt, for R's version of non-central t probabilities.

Examples

```
tt <- seq(0, 10, len = 21)
ncp <- seq(0, 6, len = 31)
pt3R   <- outer(tt, ncp, pt, , df = 3)
pt3JKB <- outer(tt, ncp, pntR, df = 3)# currently verbose
stopifnot(all.equal(pt3R, pt3JKB, tolerance = 4e-15))# 64-bit Lnx: 2.78e-16


## Gil et al.(2023) -- Table 1 p.869
str(GST23_tab1 <- read.table(header=TRUE, text = "
 x     pnt_x_delta              Rel.accuracy   l_y   j_max
 5     0.7890745035061528e-20    0.20e-13    0.29178   254
 8     0.1902963697413609e-07    0.40e-12    0.13863   294
11     0.4649258368179092e-03    0.12e-09    0.07845   310
14     0.2912746016055676e-01    0.11e-07    0.04993   317
17     0.1858422833307925e-00    0.41e-06    0.03441   321
20     0.4434882973203470e-00    0.82e-05    0.02510   323"))

x1 <- c(5,8,11,14,17,20)
(p1  <- pt  (x1, df=10.3, ncp=20))
(p1R <- pntR(x1, df=10.3, ncp=20)) # verbose=TRUE  is default
all.equal(p1, p1R, tolerance=0) # 4.355452e-15 {on x86_64} as have *no* LDOUBLE on R level
stopifnot(all.equal(p1, p1R))
## NB: According to Gil et al., the first value (x=5) is really wrong
## p1.23 <- .. Gil et al., Table 1:
p1.23.11 <- pntGST23_T6(x1, df=10.3, ncp=20, Mterms = 11)
p1.23.20 <- pntGST23_T6(x1, df=10.3, ncp=20, Mterms = 20, verbose=TRUE)
                                        # ==> Mterms = 11 is good only for x=5
p1.23.50 <- pntGST23_T6(x1, df=10.3, ncp=20, Mterms = 50, verbose=TRUE)

x <- 4:40 ; df <- 10.3
ncp <- 20
p1     <- pt        (x, df=df, ncp=ncp)
pG1    <- pntGST23_1(x, df=df, ncp=ncp)
pG1.bR <- pntGST23_1(x, df=df, ncp=ncp,
                     IxpqFUN = \(x, l_x=.5-x+.5, p, q) Ixpq(x,l_x, p,q))
pG1.BR <- pntGST23_1(x, df=df, ncp=ncp,
                     IxpqFUN = \(x, l_x, p, q)   pbeta(x, p,q))
```

```
cbind(x, p1, pG1, pG1.bR, pG1.BR)
all.equal(pG1, p1,     tolerance=0) # 1.034 e-12
all.equal(pG1, pG1.bR, tolerance=0) # 2.497031 e-13
all.equal(pG1, pG1.BR, tolerance=0) # 2.924698 e-13
all.equal(pG1.BR,pG1.bR,tolerance=0)# 1.68644  e-13
stopifnot(exprs = {
    all.equal(pG1, p1,     tolerance = 4e-12)
    all.equal(pG1, pG1.bR, tolerance = 1e-12)
    all.equal(pG1, pG1.BR, tolerance = 1e-12)
  })

ncp <- 40 ## is  > 37.62 = "critical" for Lenth' algorithm

### --------- pntVW13() -------------------------------------------------
## length 1 arguments:
str(rr <- pntVW13(t = 1, df = 2, ncp = 3, verbose=TRUE, keepS=TRUE))
all.equal(rr$cdf, pt(1,2,3), tol = 0)#  "Mean relative difference: 4.956769e-12"
stopifnot( all.equal(rr$cdf, pt(1,2,3)) )

str(rr <- pntVW13(t = 1:19, df = 2,    ncp = 3,    verbose=TRUE, keepS=TRUE))
str(r2 <- pntVW13(t = 1,    df = 2:20, ncp = 3,    verbose=TRUE, keepS=TRUE))
str(r3 <- pntVW13(t = 1,    df = 2:20, ncp = 3:21, verbose=TRUE, keepS=TRUE))

pt1.10.5_T <- 4.34725285650591657e-5 # Ex. 7 of Witkovsky(2013)
pt1.10.5 <- pntVW13(1, 10, 5)
all.equal(pt1.10.5_T, pt1.10.5, tol = 0)# TRUE! (Lnx Fedora 40; 2024-07-04);
# 3.117e-16 (Macbuilder R 4.4.0, macOS Ventura 13.3.1)
stopifnot(exprs = {
    identical(rr$cdf, r1 <- pntVW13(t = 1:19, df = 2, ncp = 3))
    identical(r1[1], pntVW13(1, 2, 3))
    identical(r1[7], pntVW13(7, 2, 3))
    all.equal(pt1.10.5_T, pt1.10.5, tol = 9e-16)# NB even tol=0 (64 Lnx)
})
## However, R' pt() is only equal for the very first
cbind(t = 1:19, pntVW = r1, pt = pt(1:19, 2,3))
```

---

pow *X to Power of Y – R C API* R_pow()

Description

pow(x,y) calls R C API 'Rmathlib"s R_pow(x,y) function to compute x^y *or* when try.int.y is true (as by default), and y is integer valued and fits into integer range, R_pow_di(x,y).

pow_di(x,y) with integer y calls R mathlib's R_pow_di(x,y).

Usage

```
pow   (x, y, try.int.y = TRUE)
pow_di(x, y)
.pow  (x, y)
```

Arguments

x	a numeric vector.
y	a numeric or in the case of pow_di() integer vector.
try.int.y	logical indicating if pow() should check if y is integer valued and fits into integer range, and in that case call pow_di() automatically.

Details

In January 2024, I found (e.g., in 'tests/pow-tst.R') that the accuracy of pow_di(), i.e., also the C function R_pow_di() in R's API is of much lower precision than R's x^y or (equivalently) R_pow(x,y) in R's API, notably on Linux and macOS, using glib etc, sometimes as soon as $y \geq 6$ or so.

.pow(x,y) is identical to pow(x,y, try.int.y = FALSE)

Value

a numeric vector like x or y which are recycled to common length, of course.

Author(s)

Martin Maechler

See Also

Base R's ^ "operator".

Examples

```
set.seed(27)
x <- rnorm(100)
y <- 0:9
stopifnot(exprs = {
    all.equal(x^y, pow(x,y))
    all.equal(x^y, pow(x,y, FALSE))
    all.equal(x^y, pow_di(x,y))
})
```

---

pow1p *Accurate $(1+x)\hat{}y$, notably for small $|x|$*

Description

Compute $(1 + x)^y$ accurately, notably also for small $|x|$, where the naive formula suffers from cancellation, returning 1, often.

Usage

```
pow1p(x, y,
      pow = ((x + 1) - 1) == x || abs(x) > 0.5 || is.na(x))
```

Arguments

x, y	numeric or number-like; in the latter case, arithmetic incl. ^, comparison, exp, log1p, abs, and is.na methods must work.
pow	logical indicating if the "naive" / direct computation (1 + x)^y should be used (unless y is in 0:4, where the binomial is used, see 'Details'). The current default is the one used in R's C-level function (but beware of compiler optimization there!).

Details

A pure R-implementation of R 4.4.0's new C-level pow1p() function which was introduced for more accurate dbinom_raw() computations.

Currently, we use the "exact" (nested) polynomial formula for $y \in \{0, 1, 2, 3, 4\}$.

MM is conjecturing that the default pow=FALSE for (most) $x \leq \frac{1}{2}$ is sub-optimal.

Value

numeric or number-like, as x + y.

Author(s)

Originally proposed by Morten Welinder, see PR#18642; tweaked, notably for small integer y, by Martin Maechler.

See Also

^, log1p, dbinom_raw.

Examples

```
x <- 2^-(1:50)
y <- 99
f1 <- (1+x)^99
f2 <- exp(y * log1p(x))
fp <- pow1p(x, 99)
matplot(x, cbind(f1, f2, fp), type = "l", col = 2:4)
legend("top", legend = expression((1+x)^99, exp(99 * log1p(x)), pow1p(x, 99)),
       bty="n", col=2:4, lwd=2)
cbind(x, f1, f2, sfsmisc::relErrV(f2, f1))
```

ppoisson *Direct Computation of 'ppois()' Poisson Distribution Probabilities*

Description

Direct computation and errors of ppois Poisson distribution probabilities.

Usage

```
ppoisD(q, lambda, all.from.0 = TRUE, verbose = 0L)
ppoisErr (lambda, ppFUN = ppoisD, iP = 1e-15,
          xM = qpois(iP, lambda=lambda, lower.tail=FALSE),
          verbose = FALSE)
```

Arguments

q	numeric vector of non-negative integer values, "quantiles" at which to evaluate ppois(q, la) and ppFUN(q, la).
lambda	positive parameter of the Poisson distribution, lambda$= \lambda = E[X] = Var[X]$ where $X \sim Pois(\lambda)$.
all.from.0	logical indicating if q is positive integer, and the probabilities should computed for all quantile values of 0:q.
ppFUN	alternative ppois evaluation, by default the **d**irect summation of dpois(k, lambda).
iP	small number, iP << 1, used to construct the abscissa values x at which to evaluate and compare ppois() and ppFUN(), see xM:
xM	(specified instead of iP:) the maximal x-value to be used, i.e., the values used will be x <- 0:iM. The default, qpois(1-iP, lambda = lambda) is the upper tail iP-quantile of Poi(lambda).
verbose	integer (≥ 0) or logical indicating if extra information should be printed.

Value

ppoisD() contains the poisson probabilities along q, i.e., is a numeric vector of length length(q).

re <- ppoisErr() returns the relative "error" of ppois(x0, lambda) where ppFUN(x0, lambda) is assumed to be the truth and x0 the "worst case", i.e., the value (among x) with the largest such difference.

Additionally, attr(re, "x0") contains that value x0.

Author(s)

Martin Maechler, March 2004; 2019 ff

See Also

ppois

Examples

```
(lams <- outer(c(1,2,5), 10^(0:3)))# 10^4 is already slow!
system.time(e1 <- sapply(lams, ppoisErr))
e1 / .Machine$double.eps

## Try another 'ppFUN' :---------------------------------
## this relies on the fact that it's *only* used on an 'x' of the form  0:M :
ppD0 <- function(x, lambda, all.from.0=TRUE)
           cumsum(dpois(if(all.from.0) 0:x else x, lambda=lambda))
## and test it:
p0 <- ppD0 (  1000, lambda=10)
p1 <- ppois(0:1000, lambda=10)
stopifnot(all.equal(p0,p1, tol=8*.Machine$double.eps))

system.time(p0.slow <- ppoisD(0:1000, lambda=10, all.from.0=FALSE))# not very slow, here
p0.1 <- ppoisD(1000, lambda=10)
if(requireNamespace("Rmpfr")) {
 ppoisMpfr <- function(x, lambda) cumsum(Rmpfr::dpois(x, lambda=lambda))
 p0.best <- ppoisMpfr(0:1000, lambda = Rmpfr::mpfr(10, precBits = 256))
 AllEq. <- Rmpfr::all.equal
 AllEq <- function(target, current, ...)
    AllEq.(target, current, ...,
           formatFUN = function(x, ...) Rmpfr::format(x, digits = 9))
 print(AllEq(p0.best, p0,      tol = 0)) # 2.06e-18
 print(AllEq(p0.best, p0.slow, tol = 0)) # the "worst" (4.44e-17)
 print(AllEq(p0.best, p0.1,    tol = 0)) # 1.08e-18
}

## Now (with 'all.from.0 = TRUE',  it is fast too):
p15    <- ppoisErr(2^13)
p15.0. <- ppoisErr(2^13, ppFUN = ppD0)
c(p15, p15.0.) / .Machine$double.eps # on Lnx 64b, see (-10  2.5), then (-2 -2)

## lapply(), so you see "x0" values :
str(e0. <- lapply(lams, ppoisErr, ppFUN = ppD0))

## The first version [called 'err.lambd0()' for years] used simple  cumsum(dpois(..))
## NOTE: It is *stil* much faster, as it relies on special  x == 0:M  relation
## Author: Martin Maechler, Date:  1 Mar 2004, 17:40
##
e0 <- sapply(lams, function(lamb) ppoisErr(lamb, ppFUN = ppD0))
all.equal(e1, e0) # typically TRUE,  though small "random" differences:
cbind(e1, e0) * 2^53 # on Lnx 64b, seeing integer values in {-24, .., 33}
```

---

pt_Witkovsky_Tab1 *Viktor Witosky's Table_1 pt() Examples*

Description

A data frame with 17 `pt()` examples from Witosky (2013)'s 'Table 1'. We provide the results for the FOSS Softwares, additionally including `octave`'s, running the original 2013 matlab code, and

the corrected one from 2022.

Usage

```
data(pt_Witkovsky_Tab1)
```

Format

A data frame with 17 observations on the following `numeric` variables.

`x` the abscissa, called `q` in `pt()`.

`nu` the *positive* degrees of freedom, called `df` in `pt()`.

`delta` the noncentrality parameter, called `ncp` in `pt()`.

`true_pnt` "true" values (computed via higher precision, see Witkovsky(2013)).

`NCTCDFVW` the `pt()` values computed with Witkovsky's matlab implementation. Confirmed by using `octave` (on Fedora 40 Linux). These correspond to our R (package **DPQ**) `pntVW13()` values.

`Boost` computed via the Boost C++ library; reported by Witkovsky.

`R_3.3.0` computed by R version 3.3.0; confirmed to be identical using R 4.4.1

`NCT2013_octave_7.3.0` values computed using Witkovsky's original matlab code, by `octave` 7.3.0

`NCT2022_octave_8.4.0` values computed using Witkovsky's 2022 corrected matlab code, by `octave` 8.4.0

Source

The table was extracted (by MM) from the result of `pdftotext --layout <*>.pdf` from the publication. The `NCT2013_octave_7.3.0` column was computed from the 2013 code, using GPL `octave 7.3.0` on Linux Fedora 38, whereas `NCT2013_octave_8.4..0` from the 2022 code, using GPL `octave 8.4.0` on Linux Fedora 40.

Note that the '`arXiv`' pre-publication has very slightly differing numbers in its `R` column, e.g., first entry ending in `00200` instead of `00111`.

References

Witkovský, Viktor (2013) A Note on Computing Extreme Tail Probabilities of the Noncentral T Distribution with Large Noncentrality Parameter, *Acta Universitatis Palackianae Olomucensis, Facultas Rerum Naturalium, Mathematica* **52**(2), 131–143.

Examples

```
data(pt_Witkovsky_Tab1)
stopifnot(is.data.frame(d.W <- pt_Witkovsky_Tab1), # shorter
          nrow(d.W) >= 17)
mW <- as.matrix(d.W); row.names(mW) <- NULL # more efficient
colnames(mW)[1:3] #  "x" "nu" "delta"
## use 'R pt() - compatible' names:
(n3 <- names(formals(pt)[1:3]))# "q" "df" "ncp"
```

```
colnames(mW)[1:3] <- n3
ptR <- apply(mW[, 1:3], 1, \(a3) unname(do.call(pt, as.list(a3))))
cNm <- paste0("R_", with(R.version, paste(major, minor, sep=".")))
mW <- cbind(mW, `colnames<-`(cbind(ptR), cNm),
            relErr = sfsmisc::relErrV(mW[,"true_pnt"], ptR))
mW
## is current R better than R 3.3.0?  -- or even "the same"?
all.equal(ptR, mW[,"R_3.3.0"])                    # still true in R 4.4.1
all.equal(ptR, mW[,"R_3.3.0"], tolerance = 1e-14) # (ditto)
table(ptR == mW[,"R_3.3.0"]) # {see only 4 (out of 17) *exactly* equal ??}

## How close to published NCTCDFVW is octave's run of the 2022 code?
with(d.W, all.equal(NCTCDFVW, NCT2022_octave_8.4.0, tolerance = 0)) # 3.977e-16

pVW <- apply(unname(mW[, 1:3]), 1, \(a3) unname(do.call(pntVW13, as.list(a3))))
all.equal(pVW, d.W$NCT2013_oct, tolerance = 0)# 2013-based pntVW13() --> 5.6443e-16
all.equal(pVW, d.W$NCT2022_oct, tolerance = 0)
```

---

qbetaAppr *Compute (Approximate) Quantiles of the Beta Distribution*

Description

Compute quantiles (inverse distribution values) of the beta distribution, using diverse approximations.

Usage

```
qbetaAppr.1(a, p, q, lower.tail=TRUE, log.p=FALSE,
            y = qnormUappr(a, lower.tail=lower.tail, log.p=log.p))

qbetaAppr.2(a, p, q, lower.tail=TRUE, log.p=FALSE, logbeta = lbeta(p,q))
qbetaAppr.3(a, p, q, lower.tail=TRUE, log.p=FALSE, logbeta = lbeta(p,q))
qbetaAppr.4(a, p, q, lower.tail=TRUE, log.p=FALSE,
            y = qnormUappr(a, lower.tail=lower.tail, log.p=log.p),
            verbose = getOption("verbose"))

qbetaAppr  (a, p, q, lower.tail=TRUE, log.p=FALSE,
            y = qnormUappr(a, lower.tail=lower.tail, log.p=log.p),
            logbeta = lbeta(p,q),
            verbose = getOption("verbose") && length(a) == 1)

qbeta.R    (alpha, p, q,
            lower.tail = TRUE, log.p = FALSE,
    logbeta = lbeta(p,q),
    low.bnd = 3e-308, up.bnd = 1-2.22e-16,
            method = c("AS109", "Newton-log"),
            tol.outer = 1e-15,
```

```
    f.acu = function(a,p,q) max(1e-300, 10^(-13- 2.5/pp^2 - .5/a^2)),
    fpu = .Machine$ double.xmin,
    qnormU.fun = function(u, lu) qnormUappr(p=u, lp=lu)
          , R.pre.2014 = FALSE
  , verbose = getOption("verbose")
          , non.finite.report = verbose
           )
```

Arguments

a, alpha	vector of probabilities (otherwise, e.g., in qbeta(), called p).
p, q	the two shape parameters of the beta distribution; otherwise, e.g., in qbeta(), called shape1 and shape2.
y	an approximation to $\Phi^{-1}(1-\alpha)$ (aka $z_{1-\alpha}$) where $\Phi(x)$ is the standard normal cumulative probability function and $\Phi-1(x)$ its inverse, i.e., R's qnorm(x).
lower.tail, log.p	logical, see, e.g., qchisq(); must have length 1.
logbeta	must be lbeta(p,q); mainly an option to pass a value already computed.
verbose	logical or integer indicating if and how much "monitoring" information should be produced by the algorithm.
low.bnd, up.bnd	lower and upper bounds for ...TODO...
method	a string specifying the approximation method to be used.
tol.outer	the "outer loop" convergence tolerance; the default 1e-15 has been hardwired in R's qbeta().
f.acu	a function with arguments (a,p,q) ...TODO...
fpu	a very small positive number.
qnormU.fun	a function with arguments (u,lu) to compute "the same" as qnormUappr(), the upper standard normal quantile.
R.pre.2014	a logical ... TODO ...
non.finite.report	logical indicating if during the "outer loop" refining iterations, if y becomes non finite and the iterations have to stop, it should be reported (before the current best value is returned).

Value

...

Author(s)

The R Core Team for the C version of qbeta in R's sources; Martin Maechler for the R port, and the approximations.

See Also

qbeta.

Examples

```
qbeta.R(0.6, 2, 3) # 0.4445
qbeta.R(0.6, 2, 3) - qbeta(0.6, 2,3) # almost 0

qbetaRV <- Vectorize(qbeta.R, "alpha") # now can use
curve(qbetaRV(x, 1.5, 2.5))
curve(qbeta  (x, 1.5, 2.5), add=TRUE, lwd = 3, col = adjustcolor("red", 1/2))

## an example of disagreement (and doubt, as borderline, close to underflow):
qbeta.R(0.5078, .01, 5) # ->  2.77558e-15    # but
qbeta  (0.5078, .01, 5) # now gives 4.651188e-31  -- correctly!
qbeta  (0.5078, .01, 5, ncp=0)# ditto
## which is because qbeta() now works in log-x scale here:
curve(pbeta(x, .01, 5), 1e-40, 1, n=10001, log="x", xaxt="n")
sfsmisc::eaxis(1); abline(h=.5078, lty=3); abline(v=4.651188e-31,col=2)
```

---

qbinomR *Pure R Implementation of R's qbinom() with Tuning Parameters*

Description

A pure R implementation, including many tuning parameter arguments, of R's own Rmathlib C code algorithm, but with more flexibility.

It is using `Vectorize(qbinomR1, *)` where the hidden `qbinomR1` works for numbers (aka 'scalar', length one) arguments only, the same as the C code.

Usage

```
qbinomR(p, size, prob, lower.tail = TRUE, log.p = FALSE,
        yLarge = 4096, # was hard wired to 1e5
        incF = 1/64,   # was hard wired to .001
        iShrink = 8,   # was hard wired to 100
        relTol = 1e-15,# was hard wired to 1e-15
        pfEps.n = 8,   # was hard wired to 64: "fuzz to ensure left continuity"
        pfEps.L = 2,   # was hard wired to 64:   "   "   ..
        fpf = 4, # *MUST* be >= 1 (did not exist previously)
        trace = 0)
```

Arguments

`p, size, prob, lower.tail, log.p`	`qbinom()` standard argument, see its help page.
`yLarge`	when $y >= y_L, y_L =$ `yLarge`, the binary root finding search is made "cleverer", taking larger increments, determined by `incF` and `iShrink`:
`incF`	a positive "increment factor" (originally hardwired to 0.001), used only when `y >= yLarge`; defines the initial increment in the search algorithm as `incr <- floor(incF * y)`.

iShrink	a positive increment shrinking factor, used only when y >= yLarge to define the new increment from the old one as incr <- max(1, floor(incr/iShrink)) where the LHS was hardired original to (incr/100).
relTol	relative tolerance, > 0; the search terminates when the (integer!) increment is less than relTol * y or the previous increment was not larger than 1.
pfEps.n	fuzz factor to ensure left continuity in the **n**ormal case log.p=FALSE; used to be hardwired to 64 (in R up to 2021-05-08).
pfEps.L	fuzz factor to ensure left continuity in case log.p=TRUE; used to be hardwired to 64 (in R up to 2021-05-08).
fpf	factor $f \geq 1$ for the **n**ormal upper tail case (log.p=FALSE, lower.tail=FALSE): p is only "fuzz-corrected", i.e., multiplied by $1 + e$ when 1 - p > fpf*e for e <- pfEps.n * c_e and $c_e = 2^{-52}$, the .Machine$double_epsilon.
trace	logical (or integer) specifying if (and how much) output should be produced from the algorithm.

Details

as mentioned on qbinom help page, qbinom uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, thus defining y := Fn(p, size, prob, ..).

The following (root finding) binary search is tweaked by the yLarge, ..., fpf arguments.

Value

a numeric vector like p recycled to the common lengths of p, size, and prob.

Author(s)

Martin Maechler

See Also

qbinom, qpois.

Examples

```
set.seed(12)
pr <- (0:16)/16 # supposedly recycled
x10 <- rbinom(500, prob=pr, size =  10); p10 <- pbinom(x10, prob=pr, size= 10)
x1c <- rbinom(500, prob=pr, size = 100); p1c <- pbinom(x1c, prob=pr, size=100)
## stopifnot(exprs = {
table( x10  == (qp10  <- qbinom (p10, prob=pr, size= 10) ))
table( qp10 == (qp10R <- qbinomR(p10, prob=pr, size= 10) )); summary(warnings()) # 30 x NaN
table( x1c  == (qp1c  <- qbinom (p1c, prob=pr, size=100) ))
table( qp1c == (qp1cR <- qbinomR(p1c, prob=pr, size=100) )); summary(warnings()) # 30 x NaN
## })
```

qchisqAppr *Compute Approximate Quantiles of the Chi-Squared Distribution*

Description

Compute quantiles (inverse distribution values) for the chi-squared distribution. using Johnson,Kotz,.. .....TODO.....

Usage

```
qchisqKG    (p, df, lower.tail = TRUE, log.p = FALSE)
qchisqWH    (p, df, lower.tail = TRUE, log.p = FALSE)
qchisqAppr  (p, df, lower.tail = TRUE, log.p = FALSE, tol = 5e-7)
qchisqAppr.R(p, df, lower.tail = TRUE, log.p = FALSE, tol = 5e-07,
             maxit = 1000, verbose = getOption("verbose"), kind = NULL)
```

Arguments

p	vector of probabilities.
df	degrees of freedom > 0, maybe non-integer; must have length 1.
lower.tail, log.p	logical, see, e.g., qchisq(); must have length 1.
tol	non-negative number, the convergence tolerance
maxit	the maximal number of iterations
verbose	logical indicating if the algorithm should produce "monitoring" information.
kind	the *kind* of approximation; if NULL, the default, the approximation chosen depends on the arguments; notably it is chosen separately for each p. Otherwise, it must be a character string. The main approximations are Wilson-Hilferty versions, when the string contains "WH". More specifically, it must be one of the strings **"chi.small"** particularly useful for small chi-squared values p;... ... **"WH"** ... ... **"p1WH"** ... ... **"WHchk"** ... ... **"df.small"** particularly useful for small degrees of freedom df... ...

Value

...

Author(s)

Martin Maechler

See Also

qchisq. Further, our approximations to the *non-central* chi-squared quantiles, qnchisqAppr

Examples

```
## TODO
```

---

qgammaAppr *Compute (Approximate) Quantiles of the Gamma Distribution*

Description

Compute approximations to the quantile (i.e., inverse cumulative) function of the Gamma distribution.

Usage

```
qgammaAppr(p, shape, lower.tail = TRUE, log.p = FALSE,
           tol = 5e-07)
qgamma.R  (p, alpha, scale = 1, lower.tail = TRUE, log.p = FALSE,
           EPS1 = 0.01, EPS2 = 5e-07, epsN = 1e-15, maxit = 1000,
           pMin = 1e-100, pMax = (1 - 1e-14),
           verbose = getOption("verbose"))

qgammaApprKG(p, shape, lower.tail = TRUE, log.p = FALSE)


qgammaApprSmallP(p, shape, lower.tail = TRUE, log.p = FALSE)
```

Arguments

p	numeric vector (possibly log tranformed) probabilities.
shape, alpha	shape parameter, non-negative.
scale	scale parameter, non-negative, see qgamma.
lower.tail, log.p	logical, see, e.g., qgamma(); must have length 1.
tol	tolerance of maximal approximation error.
EPS1	small positive number. ...
EPS2	small positive number. ...
epsN	small positive number. ...
maxit	maximal number of iterations. ...
pMin, pMax	boundaries for p. ...
verbose	logical indicating if the algorithm should produce "monitoring" information.

Details

`qgammaApprSmallP(p, a)` should be a good approximation in the following situation when both `p` and `shape` $= \alpha =: a$ are small :

If we look at Abramowitz&Stegun $gamma * (a, x) = x^{-}a * P(a, x)$ and its series $g * (a, x) = 1/gamma(a) * (1/a - 1/(a + 1) * x + ...)$,

then the first order approximation $P(a, x) = x^a * g * (a, x) = x^a/gamma(a + 1)$ and hence its inverse $x = qgamma(p, a) = (p * gamma(a + 1))^(1/a)$ should be good as soon as $1/a >> 1/(a + 1) * x$

<==> x « (a+1)/a = (1 + 1/a)

<==> x < eps *(a+1)/a

<==> log(x) < log(eps) + log((a+1)/a) = log(eps) + log((a+1)/a) ~ -36 - log(a) where log(x) ~= log(p * gamma(a+1)) / a = (log(p) + lgamma1p(a))/a

such that the above

<==> (log(p) + lgamma1p(a))/a < log(eps) + log((a+1)/a)

<==> log(p) + lgamma1p(a) < a*(-log(a)+ log(eps) + log1p(a))

<==> log(p) < a*(-log(a)+ log(eps) + log1p(a)) - lgamma1p(a) =: bnd(a)

Note that `qgammaApprSmallP()` indeed also builds on `lgamma1p()`.

`.qgammaApprBnd(a)` provides this bound $bnd(a)$; it is simply `a*(logEps + log1p(a) - log(a)) - lgamma1p(a)`, where `logEps` is $\log(\epsilon)$ = `log(eps)` where `eps <- .Machine$double.eps`, i.e. typically (always?) `logEps`$= \log \epsilon = -52 * \log(2) = -36.04365$.

Value

numeric

Author(s)

Martin Maechler

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain.

See Also

`qgamma` for R's Gamma distribution functions.

Examples

```
## TODO :  Move some of the curve()s from ../tests/qgamma-ex.R !!
```

qnbinomR *Pure R Implementation of R's qnbinom() with Tuning Parameters*

Description

A pure R implementation, including many tuning parameter arguments, of R's own Rmathlib C code algorithm, but with more flexibility.

It is using `Vectorize(qnbinomR1, *)` where the hidden `qnbinomR1` works for numbers (aka 'scalar', length one) arguments only, the same as the C code.

Usage

```
qnbinomR(p, size, prob, mu, lower.tail = TRUE, log.p = FALSE,
         yLarge = 4096, # was hard wired to 1e5
         incF = 1/64,   # was hard wired to .001
         iShrink = 8,   # was hard wired to 100
         relTol = 1e-15,# was hard wired to 1e-15
        pfEps.n = 8,   # was hard wired to 64: "fuzz to ensure left continuity"
         pfEps.L = 2,   # was hard wired to 64:   "   "   ..
         fpf = 4, # *MUST* be >= 1 (did not exist previously)
         trace = 0)
```

Arguments

`p`, `size`, `prob`, `mu`, `lower.tail`, `log.p`	`qnbinom()` standard argument, see its help page.
`yLarge`, `incF`, `iShrink`, `relTol`, `pfEps.n`, `pfEps.L`, `fpf`	numeric arguments tweaking the "root finding" search after the initial Cornish-Fisher approximation, see `qbinomR`, for details. The defaults should be more reliable (but also a bit more "expensive") than R's (original) `qnbinom()` hard wired values.
`trace`	logical (or integer) specifying if (and how much) output should be produced from the algorithm.

Value

a numeric vector like `p` recycled to the common lengths of `p`, `size`, and either `prob` or `mu`.

Author(s)

Martin Maechler

See Also

`qnbinom`, `qpois`.

Examples

```
set.seed(12)
x10 <- rnbinom(500, mu = 4,       size = 10) ; p10 <- pnbinom(x10, mu=4,       size=10)
x1c <- rnbinom(500, prob = 31/32, size = 100); p1c <- pnbinom(x1c, prob=31/32, size=100)
stopifnot(exprs = {
    x10 == qnbinom (p10, mu=4, size=10)
    x10 == qnbinomR(p10, mu=4, size=10)
    x1c == qnbinom (p1c, prob=31/32, size=100)
    x1c == qnbinomR(p1c, prob=31/32, size=100)
})
```

---

qnchisqAppr *Compute Approximate Quantiles of Noncentral Chi-Squared Distribution*

Description

Compute quantiles (inverse distribution values) for the *non-central* chi-squared distribution.

..... using Johnson,Kotz, and other approximations .....

Usage

```
qchisqAppr.0 (p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qchisqAppr.1 (p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qchisqAppr.2 (p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qchisqAppr.3 (p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qchisqApprCF1(p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qchisqApprCF2(p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)

qchisqCappr.2 (p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qchisqN       (p, df, ncp = 0, qIni = qchisqAppr.0, ...)

qnchisqAbdelAty  (p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qnchisqBolKuz    (p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qnchisqPatnaik   (p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qnchisqPearson   (p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qnchisqSankaran_d(p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)
```

Arguments

p	vector of probabilities.
df	degrees of freedom > 0, maybe non-integer.
ncp	non-centrality parameter δ; ....
lower.tail, log.p	logical, see, e.g., qchisq().
qIni	a function that computes an approximate noncentral chi-squared quantile as starting value x0 for the Newton algorithm newton().
...	further arguments to newton(), notably eps or maxiter.

Details

Compute (approximate) quantiles, using approximations analogous to those for the probabilities, see pnchisqPearson.

qchisqAppr.0(): ...TODO...

qchisqAppr.1(): ...TODO...

qchisqAppr.2(): ...TODO...

qchisqAppr.3(): ...TODO...

qchisqApprCF1(): ...TODO...

qchisqApprCF2(): ...TODO...

qchisqCappr.2(): ...TODO...

qchisqN(): Uses Newton iterations with pchisq() and dchisq() to determine qchisq(.) values.

qnchisqAbdelAty(): ...TODO...

qnchisqBolKuz(): ...TODO...

qnchisqPatnaik(): ...TODO...

qnchisqPearson(): ...TODO...

qnchisqSankaran_d(): ...TODO...

Value

numeric vectors of (noncentral) chi-squared quantiles, corresponding to probabilities p.

Author(s)

Martin Maechler, from May 1999; starting from a post to the S-news mailing list by Ranjan Maitra (@ math.umbc.edu) who showed a version of our qchisqAppr.0() thanking Jim Stapleton for providing it.

References

Johnson, N.L., Kotz, S. and Balakrishnan, N. (1995) Continuous Univariate Distributions Vol 2, 2nd ed.; Wiley; chapter 29 *Noncentral χ^2-Distributions*; notably Section *8 Approximations*, p.461 ff.

See Also

qchisq.

Examples

```
pp <- c(.001, .005, .01, .05, (1:9)/10, .95, .99, .995, .999)
pkg <- "package:DPQ"
qnchNms <- c(paste0("qchisqAppr.",0:3), paste0("qchisqApprCF",1:2),
             "qchisqN", "qchisqCappr.2", ls(pkg, pattern = "^qnchisq"))
qnchF <- sapply(qnchNms, get, envir = as.environment(pkg))
for(ncp in c(0, 1/8, 1/2)) {
```

```
   cat("\n~~~~~~~~~~~~~~\nncp: ", ncp,"\n=======\n")
   print(sapply(qnchF, function(F) Vectorize(F, "p")(pp, df = 3, ncp=ncp)))
 }

## Bug: qnchisqSankaran_d() has numeric overflow problems for large df:
qnchisqSankaran_d(pp, df=1e200, ncp = 100)

## One current (2019-08) R bug: Noncentral chi-squared quantiles on *LOG SCALE*

## a)  left/lower tail : ------------------------------------------------------
qs <- 2^seq(0,11, by=1/16)
pqL <- pchisq(qs, df=5, ncp=1, log.p=TRUE)
plot(qs, -pqL, type="l", log="xy") # + expected warning on log(0) -- all fine
qpqL <- qchisq(pqL, df=5, ncp=1, log.p=TRUE) # severe overflow :
qm <- cbind(qs, pqL, qchisq=qpqL
, qchA.0 = qchisqAppr.0 (pqL, df=5, ncp=1, log.p=TRUE)
, qchA.1 = qchisqAppr.1 (pqL, df=5, ncp=1, log.p=TRUE)
, qchA.2 = qchisqAppr.2 (pqL, df=5, ncp=1, log.p=TRUE)
, qchA.3 = qchisqAppr.3 (pqL, df=5, ncp=1, log.p=TRUE)
, qchACF1= qchisqApprCF1(pqL, df=5, ncp=1, log.p=TRUE)
, qchACF2= qchisqApprCF2(pqL, df=5, ncp=1, log.p=TRUE)
, qchCa.2= qchisqCappr.2(pqL, df=5, ncp=1, log.p=TRUE)
, qnPatnaik   = qnchisqPatnaik   (pqL, df=5, ncp=1, log.p=TRUE)
, qnAbdelAty  = qnchisqAbdelAty  (pqL, df=5, ncp=1, log.p=TRUE)
, qnBolKuz    = qnchisqBolKuz    (pqL, df=5, ncp=1, log.p=TRUE)
, qnPearson   = qnchisqPearson   (pqL, df=5, ncp=1, log.p=TRUE)
, qnSankaran_d= qnchisqSankaran_d(pqL, df=5, ncp=1, log.p=TRUE)
)

round(qm[ qs %in% 2^(0:11) , -2])
#=> Approximations don't overflow but are not good enough

## b)  right/upper tail , larger ncp ------------------------------------------
qS <- 2^seq(-3, 3, by=1/8)
pqLu <- pchisq(qS, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
## using "the alternative" (here is currently identical):
identical(pqLu, (pqLu.<- log1p(-pchisq(qS, df=5, ncp=100)))) # here TRUE
plot (qS, -pqLu, type="l", log="xy") # fine
qpqLu <- qchisq(pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
cbind(qS, pqLu, pqLu, qpqLu)# # severe underflow
qchMat <- cbind(qchisq = qpqLu
, qchA.0 = qchisqAppr.0 (pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qchA.1 = qchisqAppr.1 (pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qchA.2 = qchisqAppr.2 (pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qchA.3 = qchisqAppr.3 (pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qchACF1= qchisqApprCF1(pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qchACF2= qchisqApprCF2(pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qchCa.2= qchisqCappr.2(pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qnPatnaik   = qnchisqPatnaik   (pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qnAbdelAty  = qnchisqAbdelAty  (pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qnBolKuz    = qnchisqBolKuz    (pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qnPearson   = qnchisqPearson   (pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
, qnSankaran_d= qnchisqSankaran_d(pqLu, df=5, ncp=100, log.p=TRUE, lower.tail=FALSE)
```

```
)
cbind(L2err <- sort(sqrt(colSums((qchMat - qS)^2))))
##--> "Sankaran_d", "CF1" and "CF2" are good here

plot (qS, qpqLu, type = "b", log="x", lwd=2)
lines(qS, qS, col="gray", lty=2, lwd=3)
top3 <- names(L2err)[1:3]
use <- c("qchisq", top3)
matlines(qS, qchMat[, use]) # 3 of the approximations are "somewhat ok"
legend("topleft", c(use,"True"), bty="n", col=c(palette()[1:4], "gray"),
                  lty = c(1:4,2), lwd = c(2, 1,1,1, 3))
```

---

qnormAppr *Approximations to 'qnorm()', i.e., z_α*

Description

Approximations to the standard normal (aka "Gaussian") quantiles, i.e., the inverse of the normal cumulative probability function.

The `qnormUappr*()` are relatively simple approximations from Abramowitz and Stegun, computed by Hastings(1955): `qnormUappr()` is the 4-coefficient approximation to (the **u**pper tail) standard normal quantiles, `qnorm()`, used in some `qbeta()` computations.

`qnormUappr6()` is the "traditional" 6-coefficient approximation to `qnorm()`, see in 'Details'.

Usage

```
qnormUappr(p, lp = .DT_Clog(p, lower.tail=lower.tail, log.p=log.p),
           lower.tail = FALSE, log.p = missing(p),
           tLarge = 1e10)
qnormUappr6(p,
            lp = .DT_Clog(p, lower.tail=lower.tail, log.p=log.p),
               # ~= log(1-p) -- independent of lower.tail, log.p
            lower.tail = FALSE, log.p = missing(p),
            tLarge = 1e10)

qnormCappr(p, k = 1) ## *implicit* lower.tail=TRUE, log.p=FALSE  >>> TODO: add! <<

qnormAppr(p) # << deprecated; use qnormUappr(..) instead!
```

Arguments

`p`	numeric vector of probabilities, possibly transformed, depending on `log.p`. Does not need to be specified, if `lp` is instead.
`lp`	`log(1 - p*)`, assuming $p*$ is the `lower.tail=TRUE, log.p=FALSE` version of `p`. If passed as argument, it can be much more accurate than when computed from `p` by default.

`lower.tail`	logical; if TRUE (*not* the default here!), probabilities are $P[X \le x]$, otherwise (by default) upper tail probabilities, $P[X > x]$.
`log.p`	logical; if TRUE, probabilities p are given as $\log(p)$ in argument `p`. Note that it is *not used*, when `missing(p)` and `lp` is specified.
`tLarge`	a large number $t0$; if $t >= t0$, where $t := sqrt(-2 * lp)$, the result will be $= t$.
`k`	positive integer, specifying the iterative plugin 'order'.

Details

This is now *deprecated*; use `qnormUappr()` instead! `qnormAppr(p)` uses the simple 4 coefficient rational approximation to `qnorm(p)`, provided by Abramowitz and Stegun (26.2.22), p.933, to be used *only* for $p > 1/2$ and typically `qbeta()` computations, e.g., `qbeta.R`.
The relative error of this approximation is quite *asymmetric*: It is mainly < 0.

`qnormUappr(p)` uses the same rational approximation directly for the **U**pper tail where it is relatively good, and for the lower tail via "swapping the tails", so it is good there as well.

`qnormUappr6(p, *)` uses the 6 coefficient rational approximation to `qnorm(p, *)`, from Abramowitz and Stegun (26.2.23), again mostly useful in the outer tails.

`qnormCappr(p, k)` inverts formula (26.2.24) of Abramowitz and Stegun, and for $k \ge 2$ improves it, by iterative recursive plug-in, using A.&S. (26.2.25).

Value

numeric vector of (approximate) normal quantiles corresponding to probabilities `p`

Author(s)

Martin Maechler

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain.

Hastings jr., Cecil (1955) *Approximations for Digital Computers*. Princeton Univ. Press.

See Also

`qnorm` (in base R package **stats**), and importantly, `qnormR` and `qnormAsymp()` in this package (**DPQ**).

Examples

```
pp <- c(.001, .005, .01, .05, (1:9)/10, .95, .99, .995, .999)
z_p <- qnorm(pp)
assertDeprecation <- function(expr, verbose=TRUE)
  tools::assertCondition(expr, verbose=verbose, "deprecatedWarning")
assertDeprecation(qA <- qnormAppr(pp))
(R <- cbind(pp, z_p, qA,
```

```
            qUA = qnormUappr(pp, lower.tail= TRUE),
            qA6 = qnormUappr6(pp, lower.tail=TRUE)))
## Errors, absolute and relative:
relEr <- function(targ, curr) { ## simplistic "smart" rel.error
    E <- curr - targ
    r <- E/targ  # simple, but fix 0/0:
    r[targ == 0 & E == 0] <- 0
    r
}
mER <- cbind(pp,
             errA  = z_p - R[,"qA" ],
             errUA = z_p - R[,"qUA"],
             rE.A  = relEr(z_p, R[,"qA" ]),
             rE.UA = relEr(z_p, R[,"qUA"]),
             rE.A6 = relEr(z_p, R[,"qA6"]))
signif(mER)

lp <- -c(1000, 500, 200, 100, 50, 20:10, seq(9.75, 0, by = -1/8))
signif(digits=5, cbind(lp # 'p' need not be specified if 'lp' is !
    , p.  = -expm1(lp)
    , qnU = qnormUappr (lp=lp)
    , qnU6= qnormUappr6(lp=lp)
    , qnA1= qnormAsymp(lp=lp, lower.tail=FALSE, order=1)
    , qnA5= qnormAsymp(lp=lp, lower.tail=FALSE, order=5)
    , qn  = qnorm(lp, log.p=TRUE)
      )) ## oops! shows *BUG* for last values where qnorm() > 0 !

curve(qnorm(x, lower.tail=FALSE), n=1001)
curve(qnormUappr(x), add=TRUE,    n=1001, col = adjustcolor("red", 1/2))

## Error curve:
curve(qnormUappr(x) - qnorm(x, lower.tail=FALSE), n=1001,
      main = "Absolute Error of  qnormUappr(x)")
abline(h=0, v=1/2, lty=2, col="gray")

curve(qnormUappr(x) / qnorm(x, lower.tail=FALSE) - 1, n=1001,
      main = "Relative Error of  qnormUappr(x)")
 abline(h=0, v=1/2, lty=2, col="gray")

curve(qnormUappr(lp=x) / qnorm(x, log.p=TRUE) - 1, -200, -1, n=1001,
    main = "Relative Error of  qnormUappr(lp=x)"); mtext(" & qnormUappr6() [log.p scale]", col=2)
curve(qnormUappr6(lp=x) / qnorm(x, log.p=TRUE) - 1, add=TRUE, col=2, n=1001)
abline(h=0, lty=2, col="gray")

curve(qnormUappr(lp=x) / qnorm(x, log.p=TRUE) - 1,
      -2000, -.1, ylim = c(-2e-4, 1e-4), n=1001,
    main = "Relative Error of  qnormUappr(lp=x)"); mtext(" & qnormUappr6() [log.p scale]", col=2)
curve(qnormUappr6(lp=x) / qnorm(x, log.p=TRUE) - 1, add=TRUE, col=2, n=1001)
abline(h=0, lty=2, col="gray")

## zoom out much more - switch x-axis {use '-x'} and log-scale:
curve(qnormUappr6(lp=-x) / qnorm(-x, log.p=TRUE) - 1,
      .1, 1.1e10, log = "x", ylim = 2.2e-4*c(-2,1), n=2048,
```

```
      main = "Relative Error of  qnormUappr6(lp = -x)  [log.p scale]") -> xy.q
abline(h=0, lty=2, col="gray")

## 2023-02: qnormUappr6() can be complemented with
## an approximation around center p=1/2: qnormCappr()
p <- seq(0,1, by=2^-10)
M <- cbind(p, qn=(qn <- qnorm(p)),
           reC1 = relEr(qn, qnormCappr(p)),
           reC2 = relEr(qn, qnormCappr(p, k=2)),
           reC3 = relEr(qn, qnormCappr(p, k=3)),
           reU6 = relEr(qn, qnormUappr6(p,lower.tail=TRUE)))
matplot(M[,"p"], M[,-(1:2)], type="l", col=2:7, lty=1, lwd=2,
        ylim = c(-.004, +1e-4), xlab=quote(p), ylab = "relErr")
abline(h=0, col="gray", lty=2)
oo <- options(width=99)
summary(    M[,-(1:2)])
summary(abs(M[,-(1:2)]))
options(oo)
```

---

qnormAsymp *Asymptotic Approximation to Outer Tail of qnorm()*

Description

Implementing new asymptotic tail approximations of normal quantiles, i.e., the R function `qnorm()`, mostly useful when `log.p=TRUE` and log-scale `p` is relatively large negative, i.e., $p \ll -1$.

Usage

```
qnormAsymp(p,
           lp = .DT_Clog(p, lower.tail = lower.tail, log.p = log.p),
           order, lower.tail = TRUE, log.p = missing(p))
```

Arguments

`p`	numeric vector of probabilities, possibly transformed, depending on `log.p`. Does not need to be specified, if `lp` is used instead.
`lp`	numeric (vector) of `log(1-p)` values; if not specified, computed from `p`, depending on `lower.tail` and `log.p`.
`order`	an integer in $\{0, 1, \ldots, 5\}$, specifying the approximation order.
`lower.tail`	logical; if true, probabilities are $P[X \le x]$, otherwise upper tail probabilities, $P[X > x]$.
`log.p`	logical; if `TRUE` (as typical here!), probabilities p are given as $\log(p)$ in argument `p`.

Details

These *asymptotic* approximations have been derived by Maechler (2022) via iterative plug-in to the well known asymptotic approximations of $Q(x) = 1 - \Phi(x)$ from Abramowitz and Stegun (26.2.13), p.932, which are provided in our package **DPQ** as `pnormAsymp()`. They will be used in R >= 4.3.0's `qnorm()` to provide very accurate quantiles in the extreme tails.

Value

a numeric vector like `p` or `lp` if that was specified instead.

The simplemost (for extreme tails) is `order = 0`, where the asymptotic approximation is simply $\sqrt{-2s}$ and s is `-lp`.

Author(s)

Martin Maechler

References

Martin Maechler (2022). Asymptotic Tail Formulas For Gaussian Quantiles; **DPQ** vignette, see https://CRAN.R-project.org/package=DPQ/vignettes/qnorm-asymp.pdf.

See Also

The upper tail approximations in Abramowitz & Stegun, in **DPQ** available as `qnormUappr()` and `qnormUappr6()`, are less accurate than our `order >= 1` formulas in the tails.

Examples

```
lp <- -c(head(c(outer(c(5,2,1), 10^(18:1))), -2), 20:10, seq(9.75, 2, by = -1/8))
qnU6 <- qnormUappr6(lp=lp) # 'p' need not be specified if 'lp' is
qnAsy <- sapply(0:5, function(ord) qnormAsymp(lp=lp, lower.tail=FALSE, order=ord))
matplot(-lp, cbind(qnU6, qnAsy), type = "b", log = "x", pch=1:7)# all "the same"
legend("center", c("qnormUappr6()",
                   paste0("qnormAsymp(*, order=",0:5,")")),
       bty="n", col=1:6, lty=1:5, pch=1:7) # as in matplot()

p.ver <- function() mtext(R.version.string, cex=3/4, adj=1)
matplot(-lp, cbind(qnU6, qnAsy) - qnorm(lp, lower.tail=TRUE, log.p=TRUE),
        pch=1:7, cex = .5, xaxt = "n", # and use eaxis() instead
        main = "absolute Error of qnorm() approximations", type = "b", log = "x")
sfsmisc::eaxis(1, sub10=2); p.ver()
legend("bottom", c("qnormUappr6()",
                   paste0("qnormAsymp(*, order=",0:5,")")),
       bty="n", col=1:6, lty=1:5, pch=1:7, pt.cex=.5)

## If you look at the numbers, in versions of R <= 4.2.x,
## qnorm() is *worse* for large -lp than the higher order approximations
## ---> using qnormR() here:
absP <- function(re) pmax(abs(re), 2e-17) # not zero, so log-scale "shows" it
qnT <- qnormR(lp, lower.tail=TRUE, log.p=TRUE, version="2022") # ~= TRUE qnorm()
matplot(-lp, absP(cbind(qnU6, qnAsy) / qnT - 1),
```

```
        ylim = c(2e-17, .01), xaxt = "n", yaxt = "n", col=1:7, lty=1:7,
        main = "relative |Error| of qnorm() approximations", type = "l", log = "xy")
abline(h = .Machine$double.eps * c(1/2, 1, 2), col=adjustcolor("bisque",3/4),
       lty=c(5,2,5), lwd=c(1,3,1))
sfsmisc::eaxis(1, sub10 = 2, nintLog=20)
sfsmisc::eaxis(2, sub10 = c(-3, 2), nintLog=16)
mtext("qnT <- qnormR(*, version=\"2022\")", cex=0.9, adj=1)# ; p.ver()
legend("topright", c("qnormUappr6()",
                paste0("qnormAsymp(*, order=",0:5,")")),
       bty="n", col=1:7, lty=1:7, cex = 0.8)


###=== Optimal cut points / regions for different approximation orders k =================

## Zoom into each each cut-point region :
p.qnormAsy2 <- function(r0, k, # use k-1 and k in region around r0
                        n = 2048, verbose=TRUE, ylim = c(-1,1) * 2.5e-16,
                        rr = seq(r0 * 0.5, r0 * 1.25, length = n), ...)
{
  stopifnot(is.numeric(rr), !is.unsorted(rr), # the initial 'r'
            length(k) == 1L, is.numeric(k), k == as.integer(k), k >= 1)
  k.s <- (k-1L):k; nks <- paste0("k=", k.s)
  if(missing(r0)) r0 <- quantile(rr, 2/3)# allow specifying rr instead of r0
  if(verbose) cat("Around r0 =", r0,";  k =", deparse(k.s), "\n")
  lp <- (-rr^2) # = -r^2 = -s  <==> rr = sqrt(- lp)
 q. <- qnormR(lp, lower.tail=FALSE, log.p=TRUE, version="2022-08")# *not* depending on R ver!
  pq <- pnorm(q., lower.tail=FALSE, log.p=TRUE) # ~= lp
  ## the arg of pnorm() is the true qnorm(pq, ..) == q.  by construction
  ## cbind(rr, lp, q., pq)
  r <- sqrt(- pq)
  stopifnot(all.equal(rr, r, tol=1e-15))
  qnAsy <- sapply(setNames(k.s, nks), function(ord)
                  qnormAsymp(pq, lower.tail=FALSE, log.p=TRUE, order=ord))
  relE <- qnAsy / q. - 1
  m <- cbind(r, pq, relE)
  if(verbose) {
    print(head(m, 9)); for(j in 1:2) cat(" ..........\n")
    print(tail(m, 4))
  }
  ## matplot(r, relE, type = "b", main = paste("around r0 = ", r0))
  matplot(r, relE, type = "l", ylim = ylim,
     main = paste("Relative error of qnormAsymp(*, k) around r0 = ", r0,
                  "for  k =", deparse(k.s)),
     xlab = quote(r == sqrt(-log(p))), ...)
  legend("topleft", nks, col=1:2, lty=1:2, bty="n", lwd=2)
  for(j in seq_along(k.s))
    lines(smooth.spline(r, relE[,j]), col=adjustcolor(j, 2/3), lwd=4, lty=2)
  cc <- "blue2"; lab <- substitute(r[0] == R, list(R = r0))
  abline(v  = r0, lty=2, lwd=2, col=cc)
  axis(3, at= r0, labels=lab, col=cc, col.axis=cc, line=-1)
  abline(h = (-1:1)*.Machine$double.eps, lty=c(3,1,3),
         col=c("green3", "gray", "tan2"))
  invisible(cbind(r = r, qn = q., relE))
```

```
}

r0 <- c(27, 55, 109, 840, 36000, 6.4e8) # <--> in ../R/norm_f.R {and R's qnorm.c eventually}
## use k =   5   4    3    2      1       0    e.g.  k = 0  good for r >= 6.4e8
for(ir in 2:length(r0)) {
  p.qnormAsy2(r0[ir], k = 5 +2-ir) # k = 5, 4, ..
  if(interactive() && ir < length(r0)) {
       cat("[Enter] to continue: "); cat(readLines(stdin(), n=1), "\n") }
}
```

---

qnormR *Pure R version of* R*'s* qnorm() *with Diagnostics and Tuning Parameters*

Description

Computes R level implementations of R's qnorm() as implemented in C code (in R's 'Rmathlib'), historically and present.

Usage

```
qnormR1(p, mu = 0, sd = 1, lower.tail = TRUE, log.p = FALSE, trace = 0, version = )
qnormR (p, mu = 0, sd = 1, lower.tail = TRUE, log.p = FALSE, trace = 0,
        version = c("4.0.x", "1.0.x", "1.0_noN", "2020-10-17", "2022-08-04"))
```

Arguments

p	probability p, $1-p$, or $\log(p)$, $\log(1-p)$, depending on lower.tail and log.p.
mu	mean of the normal distribution.
sd	standard deviation of the normal distribution.
lower.tail, log.p	logical, see, e.g., qnorm().
trace	logical or integer; if positive or TRUE, diagnostic output is printed to the console during the computations.
version	a character string specifying which version or variant is used. The *current* default, "4.0.x" is the one used in R versions up to 4.0.x. The two "1.0*" versions are as used up to R 1.0.1, based on Algorithm AS 111, improved by a branch for extreme tails by Wichura, *and* a final Newton step which is only sensible when log.p=FALSE. That final stepped is skipped for version = "1.0_noN", "noN" := "no Newton". "2020-10-17" is the one committed to the R development sources on 2020-10-17, which prevents the worst for very large $\lvert p\rvert$ when log.p=TRUE. "2022-08-04" uses very accurate asymptotic formulas found on that date and provides full double precision accuracy also for extreme tails.

Details

For `qnormR1(p, ..)`, `p` must be of length one, whereas `qnormR(p, m, s, ..)` works vectorized in `p`, `mu`, and `sd`. In the **DPQ** package source, `qnormR` is simply the result of `Vectorize(qnormR1, ...)`.

Value

a numeric vector like the input `q`.

Author(s)

Martin Maechler

References

For `versions` `"1.0.x"` and `"1.0_noN"`:
Beasley, J.D. and Springer, S.G. (1977) Algorithm AS 111: The Percentage Points of the Normal Distribution. *JRSS C (Appied Statistics)* **26**, 118–121; doi:10.2307/2346889.

For the asymptotic approximations used in `versions` newer than `"4.0.x"`, i.e., `"2020-10-17"` and later, see the reference(s) on `qnormAsymp`'s help page.

See Also

`qnorm`, `qnormAsymp`.

Examples

```
qR <- curve(qnormR, n = 2^11)
abline(h=0, v=0:1, lty=3, col=adjustcolor(1, 1/2))
with(qR, all.equal(y, qnorm(x), tol=0)) # currently shows TRUE
with(qR, all.equal(pnorm(y), x, tol=0)) # currently: mean rel. diff.: 2e-16
stopifnot(with(qR, all.equal(pnorm(y), x, tol = 1e-14)))

(ver.qn <- eval(formals(qnormR)$version)) # the possible versions
(doExtras <- DPQ:::doExtras()) # TRUE e.g. if interactive()
lp <- - 4^(1:30) # effect of  'trace = *' :
qpAll <- sapply(ver.qn, function (V)
    qnormR(lp, log.p=TRUE, trace=doExtras, version = V))
head(qpAll) # the "1.0" versions underflow quickly ..

cAdj <- adjustcolor(palette(), 1/2)
matplot(-lp, -qpAll, log="xy", type="l", lwd=3, col=cAdj, axes=FALSE,
        main = "- qnormR(lp, log.p=TRUE, version = * )")
sfsmisc::eaxis(1, nintLog=15, sub=2); sfsmisc::eaxis(2)
lines(-lp, sqrt(-2*lp), col=cAdj[ncol(qpAll)+1])
leg <- as.expression(c(paste("version=", ver.qn), quote(sqrt(-2 %.% lp))))
matlines(-lp, -qpAll[,2:3], lwd=6, col=cAdj[2:3])
legend("top", leg, bty='n', col=cAdj, lty=1:3, lwd=2)

## Showing why/where R's qnorm() was poor up to 2020: log.p=TRUE extreme tail
##% MM: more TODO? --> ~/R/MM/NUMERICS/dpq-functions/qnorm-extreme-bad.R
```

```
qs <- 2^seq(0, 155, by=1/8)
lp <- pnorm(qs, lower.tail=FALSE, log.p=TRUE)
## The inverse of pnorm() fails BADLY for extreme tails:
## this is identical to qnorm(..) in R <= 4.0.x:
qp <- qnormR(lp, lower.tail=FALSE, log.p=TRUE, version="4.0.x")
## asymptotically correct approximation :
qpA <- sqrt(- 2* lp)
##^
col2 <- c("black", adjustcolor(2, 0.6))
col3 <- c(col2, adjustcolor(4, 0.6))
## instead of going toward infinity, it converges at  9.834030e+07 :
matplot(-lp, cbind(qs, qp, qpA), type="l", log="xy", lwd = c(1,1,3), col=col3,
        main = "Poorness of qnorm(lp, lower.tail=FALSE, log.p=TRUE)",
        ylab = "qnorm(lp, ..)", axes=FALSE)
sfsmisc::eaxis(1); sfsmisc::eaxis(2)
legend("top", c("truth", "qnorm(.) = qnormR(., \"4.0.x\")", "asymp. approx"),
       lwd=c(1,1,3), lty=1:3, col=col3, bty="n")

rM <- cbind(lp, qs, 1 - cbind(relE.qnorm=qp, relE.approx=qpA)/qs)
rM[ which(1:nrow(rM) %% 20 == 1) ,]
```

---

qntR *Pure R Implementation of R's qt() / qnt()*

Description

A pure R implementation of R's C API ('Mathlib' specifically) `qnt()` function which computes (non-central) t quantiles.

The simple inversion (of `pnt()`) scheme has seen to be deficient, even in cases where `pnt()`, i.e., R's `pt(.., ncp=*)` does not loose accuracy.

Usage

```
qntR1(p, df, ncp, lower.tail = TRUE, log.p = FALSE,
      pnt = stats::pt, accu = 1e-13, eps = 1e-11)
qntR (p, df, ncp, lower.tail = TRUE, log.p = FALSE,
      pnt = stats::pt, accu = 1e-13, eps = 1e-11)
```

Arguments

p, df, ncp	vectors of probabilities, degrees of freedom, and non-centrality parameter; see `qt`.
lower.tail, log.p	logical; see `qt`.
pnt	a `function` for computing the CDF of the (non-central) t-distribution.
accu	a non-negative number, the "accu"racy desired in the "root finding" loop.
eps	a non-negative number, used for determining the start interval for the root finding.

Value

numeric vector of t quantiles, properly recycled in `(p, df, ncp)`.

Author(s)

Martin Maechler

See Also

Our `qtU()` and `qtAppr()`; non-central density and probability approximations in `dntJKBf`, and e.g., `pntR`. Further, R's `qt`.

Examples

```
## example where qt() and qntR() "fail" {warnings; --> Inf}
lp <- seq(-30, -24, by=1/4)
summary(p <- exp(lp))
(qp <- qntR( p, df=35, ncp=-7, lower.tail=FALSE))
qp2 <- qntR(lp, df=35, ncp=-7, lower.tail = FALSE, log.p=TRUE)
all.equal(qp, qp2)## same warnings, same values
```

---

qpoisR *Pure R Implementation of R's qpois() with Tuning Parameters*

Description

A pure R implementation, including many tuning parameter arguments, of R's own Rmathlib C code algorithm, but with more flexibility.

It is using `Vectorize(qpoisR1, *)` where the hidden `qpoisR1` works for numbers (aka 'scalar', length one) arguments only, the same as the C code.

Usage

```
qpoisR(p, lambda, lower.tail = TRUE, log.p = FALSE,
       yLarge = 4096, # was hard wired to 1e5
       incF = 1/64,   # was hard wired to .001
       iShrink = 8,   # was hard wired to 100
       relTol = 1e-15,# was hard wired to 1e-15
       pfEps.n = 8,   # was hard wired to 64: "fuzz to ensure left continuity"
       pfEps.L = 2,   # was hard wired to 64:   "   "   ..
       fpf = 4, # *MUST* be >= 1 (did not exist previously)
       trace = 0)
```

Arguments

p, lambda, lower.tail, log.p
: qpois() standard argument, see its help page.

yLarge
: a positive number; in R up to 2021, was internally hardwired to yLarge = 1e5: Uses more careful search for $y \geq y_L$, where y is the initial approximate result, derived from a Cornish-Fisher expansiion.

incF
: a positive "increment factor" (originally hardwired to 0.001), used only when y >= yLarge; defines the initial increment in the search algorithm as incr <- floor(incF * y).

iShrink
: a positive increment shrinking factor, used only when y >= yLarge to define the new increment from the old one as incr <- max(1, floor(incr/iShrink)) where the LHS was hardired original to (incr/100).

relTol
: originally hard wired to 1e-15, defines the convergence tolerance for the search iterations when y >= yLarge; the iterations stop when (new) incr <= y * relTol.

pfEps.n, pfEps.L
: positive factors defining "fuzz to ensure left continuity", both originally hard-wired to 64, the fuzz adjustment was

```
p <- p * (1 - 64 *.Machine$double.eps)
```

Now, pfEps.L is used if(log.p) is true and pfEps.n is used otherwise ("n"ormal case), and the adjustments also depend on lower.tail, and also on fpf :

fpf
: a number larger than 1, together with pfEps.n determines the fuzz-adjustment to p in the case (lower=tail=FALSE, log.p=FALSE): with e <- pfEps.n * .Machine$double.eps, the adjustment p <- p * (1 + e) is made *iff* 1 - p > fpf*e.

trace
: logical (or integer) specifying if (and how much) output should be produced from the algorithm.

Details

The defaults and exact meaning of the algorithmic tuning arguments from yLarge to fpf were experimentally determined are subject to change.

Value

a numeric vector like p recycled to the common lengths of p and lambda.

Author(s)

Martin Maechler

See Also

qpois.

Examples

```
x <- 10*(15:25)
Pp <- ppois(x, lambda = 100, lower.tail = FALSE)  # no cancellation
qPp <- qpois(Pp, lambda = 100, lower.tail=FALSE)
table(x == qPp) # all TRUE ?
## future: if(getRversion() >= "4.2") stopifnot(x == qPp) # R-devel
qpRp <- qpoisR(Pp, lambda = 100, lower.tail=FALSE)
all.equal(x, qpRp, tol = 0)
stopifnot(all.equal(x, qpRp, tol = 1e-15))
```

---

qtAppr *Compute Approximate Quantiles of the (Non-Central) t-Distribution*

Description

Compute quantiles (inverse distribution values) for the non-central t distribution. using Johnson,Kotz,.. p.521, formula (31.26 a) (31.26 b) & (31.26 c)

Note that `qt(.., ncp=*)` did not exist yet in 1999, when MM implemented `qtAppr()`.

`qtNappr()` approximates t-quantiles for large `df`, i.e., when close to the Gaussian / normal distribution, using up to 4 asymptotic terms from Abramowitz & Stegun 26.7.5 (p.949).

Usage

```
qtAppr (p, df, ncp, lower.tail = TRUE, log.p = FALSE, method = c("a", "b", "c"))
qtNappr(p, df,      lower.tail = TRUE, log.p=FALSE, k)
```

Arguments

`p`	vector of probabilities.
`df`	degrees of freedom > 0, maybe non-integer.
`ncp`	non-centrality parameter δ; ....
`lower.tail, log.p`	logical, see, e.g., `qt()`.
`method`	a string specifying the approximation method to be used.
`k`	an integer in {0,1,2,3,4}, choosing the number of terms in `qtNappr()`.

Value

numeric vector of length `length(p + df + ncp)` with approximate t-quantiles.

References

Johnson, N.L., Kotz, S. and Balakrishnan, N. (1995) Continuous Univariate Distributions Vol~2, 2nd ed.; Wiley; chapter 31, Section *6 Approximation*, p.519 ff

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover; formula (26.7.5), p.949; https://en.wikipedia.org/wiki/Abramowitz_and_Stegun provides links to the full text which is in public domain.

See Also

Our qtU(); several non-central density and probability approximations in dntJKBf, and e.g., pntR. Further, R's qt.

Examples

```
qts <- function(p, df) {
    cbind(qt = qt(p, df=df)
        , qtN0 = qtNappr(p, df=df, k=0)
        , qtN1 = qtNappr(p, df=df, k=1)
        , qtN2 = qtNappr(p, df=df, k=2)
        , qtN3 = qtNappr(p, df=df, k=3)
        , qtN4 = qtNappr(p, df=df, k=4)
          )
}
p <- (0:100)/100
ii <- 2:100 # drop p=0 & p=1  where q*(p, .) == +/- Inf

df <- 100 # <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
qsp1c <- qts(p, df = df)
matplot(p, qsp1c, type="l") # "all on top"
(dq <- (qsp1c[,-1] - qsp1c[,1])[ii,])
matplot(p[ii], dq, type="l", col=2:6,
        main = paste0("difference qtNappr(p,df) - qt(p,df), df=",df), xlab=quote(p))
matplot(p[ii], pmax(abs(dq), 1e-17), log="y", type="l", col=2:6,
      main = paste0("abs. difference |qtNappr(p,df) - qt(p,df)|, df=",df), xlab=quote(p))
legend("bottomright", paste0("k=",0:4), col=2:6, lty=1:5, bty="n")
matplot(p[ii], abs(dq/qsp1c[ii,"qt"]), log="y", type="l", col=2:6,
        main = sprintf("rel.error qtNappr(p, df=%g, k=*)",df), xlab=quote(p))
legend("left", paste0("k=",0:4), col=2:6, lty=1:5, bty="n")

df <- 2000 # <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
qsp1c <- qts(p, df=df)
(dq <- (qsp1c[,-1] - qsp1c[,1])[ii,])
matplot(p[ii], dq, type="l", col=2:6,
        main = paste0("difference qtNappr(p,df) - qt(p,df), df=",df), xlab=quote(p))
legend("top", paste0("k=",0:4), col=2:6, lty=1:5)
matplot(p[ii], pmax(abs(dq), 1e-17), log="y", type="l", col=2:6,
        main = paste0("abs.diff. |qtNappr(p,df) - qt(p,df)|, df=",df), xlab=quote(p))
legend("right", paste0("k=",0:4), col=2:6, lty=1:5, bty="n")

matplot(p[ii], abs(dq/qsp1c[ii,"qt"]), log="y", type="l", col=2:6,
        main = sprintf("rel.error qtNappr(p, df=%g, k=*)",df), xlab=quote(p))
legend("left", paste0("k=",0:4), col=2:6, lty=1:5, bty="n")
```

Description

A pure R implementation of R's Mathlib own C-level `qt()` function.
`qtR()` is simply defined as

```
qtR <- Vectorize(qtR1, c("p","df"))
```

where in `qtR1(p, df, *)` both `p` and `df` must be of length one.

Usage

```
qtR1(p, df, lower.tail = TRUE, log.p = FALSE,
     eps = 1e-12, d1_accu = 1e-13, d1_eps = 1e-11,
     itNewt = 10L, epsNewt = 1e-14, logNewton = log.p,
     verbose = FALSE)
qtR (p, df, lower.tail = TRUE, log.p = FALSE,
     eps = 1e-12, d1_accu = 1e-13, d1_eps = 1e-11,
     itNewt = 10L, epsNewt = 1e-14, logNewton = log.p,
     verbose = FALSE)
```

Arguments

`p`, `df`	vectors of probabilities and degrees of freedom, see `qt`.
`lower.tail`, `log.p`	logical; see `qt`.
`eps`	non-negative tolerance for checking if `df` is "very close" to 1 or 2, respectively (when a special branch will be chosen).
`d1_accu`, `d1_eps`	non-negative tolerances only for the `df` < 1 cases.
`itNewt`	integer, the maximal number of final Newton(-Raphson) steps.
`epsNewt`	non-negative convergence tolerance for the final Newton steps.
`logNewton`	logical, in case of `log.p=TRUE` indicating if final Newton steps should happen in log-scale.
`verbose`	logical indicating if diagnostic console output should be produced.

Value

numeric vector of t quantiles, properly recycled in `(p, df)`.

Author(s)

Martin Maechler

See Also

`qtU` and R's `qt`.

Examples

```
## Inspired from Bugzilla PR#16380
pxy <- curve(pt(-x, df = 1.09, log.p = TRUE), 4e152, 1e156, log="x", n=501)
qxy <- curve(-qt(x, df = 1.09, log.p = TRUE), -392, -385, n=501, log="y", col=4, lwd=2)
lines(x ~ y, data=pxy, col = adjustcolor(2, 1/2), lwd=5, lty=3)
## now our "pure R" version:

qRy <- -qtR(qxy$x, df = 1.09, log.p = TRUE)
all.equal(qRy, qxy$y) # "'is.NA' value mismatch: 14 in current 0 in target" for R <= 4.2.1
cbind(as.data.frame(qxy), qRy, D = qxy$y - qRy)
plot((y - qRy) ~ x, data = qxy, type="o", cex=1/4)

qtR1(.1, .1, verbose=TRUE)
pt(qtR(-390.5, df=1.10, log.p=TRUE, verbose=TRUE, itNewt = 100), df=1.10, log.p=TRUE)/-390.5 - 1
## qt(p=     -390.5, df=        1.1, *) -- general case
##  -> P=2.55861e-170, neg=TRUE, is_neg_lower=TRUE; -> final P=5.11723e-170
## usual 'df' case:  P_ok:= P_ok1 = TRUE, y=3.19063e-308, P..., !P_ok: log.p2=-390.5, y=3.19063e-308
## !P_ok && x < -36.04: q=5.87162e+153
## P_ok1: log-scale Taylor (iterated):
## it= 1, .. d{q}1=exp(lF - dt(q,df,log=T))*(lF - log(P/2)) = -5.03644e+152; n.q=5.36798e+153
## it= 2, .. d{q}1=exp(lF - dt(q,df,log=T))*(lF - log(P/2)) =  2.09548e+151; n.q=5.38893e+153
## it= 3, .. d{q}1=exp(lF - dt(q,df,log=T))*(lF - log(P/2)) =  4.09533e+148; n.q=5.38897e+153
## it= 4, .. d{q}1=exp(lF - dt(q,df,log=T))*(lF - log(P/2)) =   1.5567e+143; n.q=5.38897e+153
## [1] 0
##    === perfect!
pt(qtR(-391, df=1.10, log.p=TRUE, verbose=TRUE),
   df=1.10, log.p=TRUE)/-391 - 1 # now perfect
```

---

qtU *'uniroot()'-based Computing of t-Distribution Quantiles*

Description

Currently, R's own `qt()` (aka `qnt()` in the non-central case) uses simple inversion of `pt` to compute quantiles in the case where `ncp` is specified.
That simple inversion (of `pnt()`) has seen to be deficient, even in cases where `pnt()`, i.e., R's `pt(.., ncp=*)` does not loose accuracy.

This `uniroot()`-based inversion does *not* suffer from these deficits in some cases.
`qtU()` is simply defined as

```
qtU <- Vectorize(qtU1, c("p","df","ncp"))
```

where in `qtU1(p, df, ncp, *)` each of `(p, df, ncp)` must be of length one.

Usage

```
qtU1(p, df, ncp, lower.tail = TRUE, log.p = FALSE, interval = c(-10, 10),
     tol = 1e-05, verbose = FALSE, ...)
qtU (p, df, ncp, lower.tail = TRUE, log.p = FALSE, interval = c(-10, 10),
     tol = 1e-05, verbose = FALSE, ...)
```

Arguments

p, df, ncp	vectors of probabilities, degrees of freedom, and non-centrality parameter; see qt. As there, ncp may be missing which amounts to being zero.
lower.tail, log.p	logical; see qt.
interval	the interval in which quantiles should be searched; passed to uniroot(); the current default is arbitrary and suboptimal; when pt(q,*) is accurate enough and hence *montone* (increasing iff lower.tail), this interval is automatically correctly extended by uniroot().
tol	non-negative convergence tolerance passed to uniroot().
verbose	logical indicating if *every* call of the objective function should produce a line of console output.
...	optional further arguments passed to uniroot().

Value

numeric vector of t quantiles, properly recycled in (p, df, ncp).

Author(s)

Martin Maechler

See Also

uniroot and pt are the simple R level building blocks. The length-1 argument version qtU1() is short and simple to understand.

Examples

```
qtU1 # simple definition {with extras only for  'verbose = TRUE'}

## An example, seen to be deficient
## Stephen Berman to R-help, 13 June 2022,
## "Why does qt() return Inf with certain negative ncp values?"
q2 <- seq(-3/4, -1/4, by=1/128)
pq2 <- pt(q2, 35, ncp=-7, lower.tail=FALSE)
### ==> via qtU(), a simple uniroot() - based inversion of pt()
qpqU  <- qtU(pq2, 35, ncp=-7, lower.tail=FALSE, tol=1e-10)
stopifnot(all.equal(q2, qpqU, tol=1e-9)) # perfect!

## These two currently (2022-06-14) give Inf  whereas qtU() works fine
qt  (9e-12, df=35, ncp=-7, lower.tail=FALSE) # warnings; --> Inf
qntR(9e-12, df=35, ncp=-7, lower.tail=FALSE) #  (ditto)
## verbose = TRUE  shows all calls to pt():
qtU1(9e-12, df=35, ncp=-7, lower.tail=FALSE, verbose=TRUE)
```

`rexpm1` *TOMS 708 Approximation REXP(x) of expm1(x) = exp(x) - 1*

Description

Originally `REXP()`, now `rexpm1()` is a numeric (double precision) approximation of $exp(x) - 1$, notably for small $|x| \ll 1$ where direct evaluation looses accuracy through cancellation.

Fully accurate computations of $exp(x) - 1$ are now known as `expm1(x)` and have been provided by math libraries (for C, C++, ..) and R, (and are typically more accurate than `rexp1()`).

The `rexpm1()` approximation was developed by Didonato & Morris (1986) and uses a minimax rational approximation for $|x| <= 0.15$; the authors say "*accurate to within 2 units of the 14th significant digit*" (top of p.379).

Usage

```
rexpm1(x)
```

Arguments

`x`	a numeric vector.

Value

a numeric vector (or array) as `x`,

Author(s)

Martin Maechler, for the C to R *vectorized* translation.

References

Didonato, A.R. and Morris, A.H. (1986) Computation of the Incomplete Gamma Function Ratios and their Inverse. *ACM Trans. on Math. Softw.* **12**, 377–393, doi:10.1145/22721.23109; The above is the "flesh" of 'TOMS 654':

Didonato, A.R. and Morris, A.H. (1987) Algorithm 654: FORTRAN subroutines for Compute the Incomplete Gamma Function Ratios and their Inverse. *ACM Transactions on Mathematical Software* **13**, 318–319, doi:10.1145/29380.214348.

See Also

`pbeta`, where the C version of `rexpm1()` has been used in several places, notably in the original TOMS 708 algorithm.

Examples

```
x <- seq(-3/4, 3/4, by=1/1024)
plot(x,     rexpm1(x)/expm1(x) - 1, type="l", main = "Error wrt expm1()")
abline(h = (-8:8)*2^-53, lty=1:2, col=adjustcolor("gray", 1/2))
cb2 <- adjustcolor("blue", 1/2)
do.15 <- function(col = cb2) {
    abline(v = 0.15*(-1:1), lty=3, lwd=c(3,1,3), col=col)
    axis(1, at=c(-.15, .15), col=cb2, col.axis=cb2)
}
do.15()

 op <- par(mar = par("mar") + c(0,0,0,2))
plot(x, abs(rexpm1(x)/expm1(x) - 1),type="l", log = 'y',
     main = "*Relative* Error wrt expm1() [log scale]")#, yaxt="n"
abline(h = (1:9)*2^-53, lty=2, col=adjustcolor("gray", 1/2))
axis(4, at = (1:9)*2^-53, las = 1, labels =
     expression(2^-53, 2^-52, 3 %*% 2^-53, 2^-51, 5 %*% 2^-53,
                6 %*% 2^-53, 7 %*% 2^-53, 2^-50, 9 %*% 2^-53))
do.15()
 par(op)

## "True" Accuracy comparison of  rexpm1() with [OS mathlib based] expm1():
if(require("Rmpfr")) withAutoprint({
  xM <- mpfr(x, 128); Xexpm1 <- expm1(xM)
  REr1 <- asNumeric(rexpm1(x)/Xexpm1 - 1)
  REe1 <- asNumeric(expm1(x) /Xexpm1 - 1)
  absC <- function(E) pmax(2^-55, abs(E))

  plot(x, absC(REr1), type= "l", log="y",
       main = "|rel.Error|  of exp(x)-1 computations wrt 128-bit MPFR ")
  lines(x, absC(REe1), col = (c2 <- adjustcolor(2, 3/4)))
  abline(h = (1:9)*2^-53, lty=2, col=adjustcolor("gray60", 1/2))
  do.15()
 axis(4, mgp=c(2,1/4,0),tcl=-1/8, at=2^-(53:51), labels=expression(2^-53, 2^-52, 2^-51), las=1)
  legend("topleft", c("rexpm1(x)", " expm1(x)"), lwd=2, col=c("black", c2),
         bg = "gray90", box.lwd=.1)

})
```

`r_pois` *Compute Relative Size of i-th term of Poisson Distribution Series*

Description

Compute

$$r_\lambda(i) := (\lambda^i/i!)/e_{i-1}(\lambda),$$

where $\lambda =$`lambda`, and

$$e_n(x) := 1 + x + x^2/2! + ... + x^n/n!$$

is the n-th partial sum of $\exp(x) = e^x$.

Questions: As function of i

- Can this be put in a simple formula, or at least be well approximated for large λ and/or large i?
- For which i ($:= i_m(\lambda)$) is it maximal?
- When does $r_\lambda(i)$ become smaller than (f+2i-x)/x = a + b*i ?

NB: This is relevant in computations for non-central chi-squared (and similar non-central distribution functions) defined as weighted sum with "Poisson weights".

Usage

```
r_pois(i, lambda)
r_pois_expr  # the R expression() for the asymptotic branch of r_pois()

plRpois(lambda, iset = 1:(2*lambda), do.main = TRUE,
        log = 'xy', type = "o", cex = 0.4, col = c("red","blue"),
        do.eaxis = TRUE, sub10 = "10")
```

Arguments

i	integer ..
lambda	non-negative number ...
iset	.....
do.main	logical specifying if a main title should be drawn via (main = r_pois_expr).
type	type of (line) plot, see lines.
log	string specifying if (and where) logarithmic scales should be used, see plot.default().
cex	character expansion factor.
col	colors for the two curves.
do.eaxis	logical specifying if eaxis() (package **sfsmisc**) should be used.
sub10	argument for eaxis() (with a different default than the original).

Details

r_pois() is related to our series expansions and approximations for the non-central chi-squared; in particular pnchisq_ss()

plRpois() simply produces a "nice" plot of r_pois(ii, *) vs ii.

Value

r_pois() returns a numeric vector $r_\lambda(i)$ values.

r_pois_expr() an expression.

Author(s)

Martin Maechler, 20 Jan 2004

See Also

dpois().

Examples

```
plRpois(12)
plRpois(120)
```

---

stirlerr *Stirling's Error Function - Auxiliary for Gamma, Beta, etc*

Description

Stirling's approximation (to the factorial or Γ function) error in $\log$ scale is the difference of the left and right hand side of Stirling's approximation to $n!$, $n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$, i.e., `stirlerr(n)` := $\delta(n)$, where

$$\delta(n) = \log\Gamma(n+1) - n\log(n) + n - \log(2\pi n)/2.$$

Partly, pure R transcriptions of the C code utility functions for dgamma(), dbinom(), dpois(), dt(), and similar "base" density functions by Catherine Loader.

These **DPQ** versions typically have extra arguments with defaults that correspond to R's Mathlib C code hardwired cutoffs and tolerances.

`lgammacor(x)` is "the same" as `stirlerr(x)`, both computing $delta(x)$ accurately, however is only defined for $x \geq 10$, and has been crucially used for R's own lgamma() and lbeta() computations.

Usage

```
stirlerr(n, scheme = c("R3", "R4.4_0"),
         cutoffs = switch(scheme
                          , R3     = c(15, 35, 80, 500)
                         , R4.4_0 = c(5.25, rep(6.5, 4), 7.1, 7.6, 8.25, 8.8, 9.5, 11,
                                      14, 19,   25, 36, 81, 200, 3700, 17.4e6)
                          ),
         use.halves = missing(cutoffs),
         direct.ver = c("R3", "lgamma1p", "MM2", "n0"),
         order = NA,
         verbose = FALSE)

stirlerrC(n, version = c("R3", "R4..1", "R4.4_0"))

stirlerr_simpl(n, version = c("R3", "lgamma1p", "MM2", "n0"), minPrec = 128L)

lgammacor(x, nalgm = 5, xbig = 2^26.5)
```

Arguments

x, n	numeric (or number-alike such as "mpfr").
verbose	logical indicating if some information about the computations are to be printed.
version	a character string specifying the version of stirlerr_simpl() or stirlerrC().
scheme	a character string specifying the cutoffs scheme for stirlerr().
cutoffs	an increasing numeric vector, required to start with with cutoffs[1] <= 15 specifying the cutoffs to switch from 2 to 3 to ..., up to 10 term approximations for non-small n, where the direct formula loses precision. When missing (as by default), scheme is used, where scheme = "R3" chooses (15, 35, 80, 500), the cutoffs in use in R versions up to (and including) 4.3.z.
use.halves	logical indicating if the full-accuracy prestored values should be use when $2n \in \{0, 1, \ldots, 30\}$, i.e., $n \le 15$ and n is integer or integer + $\frac{1}{2}$. Turn this off to judge the underlying approximation accuracy by comparison with MPFR. However, keep the default TRUE for back-compatibility.
direct.ver	a character string specifying the version of stirlerr_simpl() to be used for the "direct" case in stirlerr(n).
order	approximation order, 1 <= order <= 20 or NA for stirlerr(). If not NA, it specifies the number of terms to be used in the Stirling series which will be used for all n, i.e., scheme, cutoffs, use.halves, and direct.ver are irrelevant.
minPrec	a positive integer; for stirlerr_simpl the minimal accuracy or precision in bits when mpfr numbers are used.
nalgm	number of terms to use for Chebyshev polynomial approximation in lgammacor(). The default, 5, is the value hard wired in R's C Mathlib.
xbig	a large positive number; if x >= xbig, the simple asymptotic approximation lgammacor(x) := 1/(12*x) is used. The default, $2^{26.5} = 94906265.6$, is the value hard wired in R's C Mathlib.

Details

stirlerr(): Stirling's error, stirlerr(n) := $\delta(n)$ has asymptotic ($n \to \infty$) expansion

$$\delta(n) = \frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5} \pm O(n^{-7}),$$

and this expansion is used up to remainder $O(n^{-35})$ in current (package **DPQ**) stirlerr(n); different numbers of terms between different cutoffs for n, and using the direct formula for $n <= c_1$, where c_1 is the first cutoff, cutoff[1].

Note that (new in 2024-01) stirlerr(n, order = k) will *not* use cutoffs nor the direct formula (with its direct.ver), nor halves (use.halves=TRUE), and allows $k \le 20$. Tests seem to indicate that for current double precision arithmetic, only $k \le 17$ seem to make sense.

lgammacor(x): The "same" Stirling's error, but only defined for $x \ge 10$, returning NaN otherwise. The example below suggests that R's hardwired default of nalgm = 5 loses more than one digit accuracy, and nalgm = 6 seems much better. OTOH, the use of lgammacor() in R's (Mathlib/'libRmath') C code is always in conjunction with considerably larger terms such that small inaccuracies in lgammacor() will not become visible in the values of the functions using lgammacor() internally, notably lbeta() and lgamma().

Value

a numeric vector "like" `x`; in some cases may also be an (high accuracy) "mpfr"-number vector, using CRAN package **Rmpfr**.

`lgammacor(x)` originally returned `NaN` for all $|x| < 10$, as its Chebyshev polynomial approximation has been constructed for $x \in [10, xbig]$, specifically for $u \in [-1, 1]$ where $t := 10/x \in [1/x_B, 1]$ and $u := 2t^2 - 1 \in [-1 + \epsilon_B, 1]$.

Author(s)

Martin Maechler

References

C. Loader (2000), see `dbinom`'s documentation.

Our package vignette *log1pmx, bd0, stirlerr - Probability Computations in R*.

See Also

`dgamma`, `dpois`. High precision versions `stirlerrM(n)` and `stirlerrSer(n,k)` in package **DPQmpfr** (via the **Rmpfr** and **gmp** packages).

Examples

```
n <- seq(1, 50, by=1/4)
st.n <- stirlerr(n) # now vectorized
stopifnot(identical(st.n, sapply(n, stirlerr)))
st3. <- stirlerr(n, "R3", direct.ver = "R3") # previous default
st3  <- stirlerr(n, "R3", direct.ver = "lgamma1p") # new? default
## for these n, there is *NO* difference:
stopifnot(st3 == st3.)
plot(n, st.n, type = "b", log="xy", ylab = "stirlerr(n)")
st4 <- stirlerr(n, "R4.4_0", verbose = TRUE) # verbose: give info on cases
## order = k = 1:20  terms in series approx:
k <- 1:20
stirlOrd <- sapply(k, function(k) stirlerr(n, order = k))
matlines(n, stirlOrd)
matplot(n, stirlOrd - st.n, type = "b", cex=1/2, ylim = c(-1,1)/10, log = "x",
     main = substitute(list(stirlerr(n, order=k) ~~"error", k == 1:mK), list(mK = max(k))))

matplot(n, abs(stirlOrd - st.n), type = "b", cex=1/2, log = "xy",
        main = "| stirlerr(n, order=k) error |")
mtext(paste("k =", deparse(k))) ; abline(h = 2^-(53:51), lty=3, lwd=1/2)
colnames(stirlOrd) <- paste0("k=", k)

stCn <- stirlerrC(n)
all.equal(st.n, stCn, tolerance = 0)  # see 6.7447e-14
stopifnot(all.equal(st.n, stCn, tolerance = 1e-12))
stC2 <- stirlerrC(n, version = "R4..1")
stC4 <- stirlerrC(n, version = "R4.4_0")
```

```
## lgammacor(n) : only defined for n >= 10
lgcor <- lgammacor(n)
lgcor6 <- lgammacor(n, nalgm = 6) # more accurate?

all.equal(lgcor[n >= 10], st.n[n >= 10], tolerance=0)# .. rel.diff.: 4.687e-14
stopifnot(identical(is.na(lgcor), n < 10),
          all.equal(lgcor[n >= 10],
                    st.n [n >= 10], tolerance = 1e-12))

## look at *relative* errors -- need "Rmpfr" for "truth" % Rmpfr / DPQmpfr in 'Suggests'
if(requireNamespace("Rmpfr") && requireNamespace("DPQmpfr")) {
    ## stirlerr(n) uses DPQmpfr::stirlerrM()  automagically when n is <mpfr>
    relErrV <- sfsmisc::relErrV; eaxis <- sfsmisc::eaxis
    mpfr <- Rmpfr::mpfr;     asNumeric <- Rmpfr::asNumeric
    stM <- stirlerr(mpfr(n, 512))
    relE <- asNumeric(relErrV(stM, cbind(st3, st4, stCn, stC4,
                                         lgcor, lgcor6, stirlOrd)))

    matplot(n, pmax(abs(relE),1e-20), type="o", cex=1/2, log="xy", ylim =c(8e-17, 0.1),
            xaxt="n", yaxt="n", main = quote(abs(relErr(stirlerr(n)))))
    ## mark "lgcor*" -- lgammacor() particularly !
    col.lgc <- adjustcolor(c(2,4), 2/3)
    matlines(n, abs(relE[,c("lgcor","lgcor6")]), col=col.lgc, lwd=3)
    lines(n, abs(relE[,"lgcor6"]), col=adjustcolor(4, 2/3), lwd=3)
    eaxis(1, sub10=2); eaxis(2); abline(h = 2^-(53:51), lty=3, col=adjustcolor(1, 1/2))
    axis(1, at=15, col=NA, line=-1); abline(v=c(10,15), lty=2, col=adjustcolor(1, 1/4))
    legend("topright", legend=colnames(relE), cex = 3/4,
           col=1:6, lty=1:5, pch= c(1L:9L, 0L, letters)[seq_len(ncol(relE))])
    legend("topright", legend=colnames(relE)[1:6], cex = 3/4, lty=1:5, lwd=3,
           col=c(rep(NA,4), col.lgc), bty="n")
    ## Note that lgammacor(x) {default, n=5} is clearly inferior,
    ## but lgammacor(x, 6) is really good in [10, 50]

    ## ===> Larger n's:
    nL <- c(seq(50, 99, by = 1/2), 100*2^seq(0,8, by = 1/4))
    stMl <- stirlerr(mpfr(nL, 512))
    lgc5 <- lgammacor(nL, nalgm = 5)
    lgc6 <- lgammacor(nL, nalgm = 6)
    stir7 <- stirlerr(nL, order = 7)
    relEl <- asNumeric(relErrV(stMl,
                cbind(lgammacor.5 = lgc5, lgammacor.6 = lgc6, 'stirlerr_k=7' = stir7)))
    matplot(nL, pmax(abs(relEl),2^-55), type="o", cex=2/3, log="xy",
            ylim = c(2^-54.5, max(abs(relEl))), ylab = quote(abs(rE)),
            xaxt="n", yaxt="n", main = quote(abs(relErr(stirlerr(n)))))
    eaxis(1, sub10=2); eaxis(2, cex.axis=.8)
    abline(h = 2^-(54:51), lty=3, col=adjustcolor(1, 1/2))
    legend("center", legend=colnames(relEl), lwd=2, pch = paste(1:3), col=1:3, lty=1:3)
}# end if( <Rmpfr> )
```

Index