Package: DPQmpfr (via r-universe)

August 24, 2024

Title DPQ (Density, Probability, Quantile) Distribution Computations using MPFR

Version 0.3-3

VersionNote Last CRAN: 0.3-2 on 2023-12-05; 0.3-1 on 2021-05-17

Date 2024-08-19

Description An extension to the 'DPQ' package with computations for 'DPQ' (Density (pdf), Probability (cdf) and Quantile) functions, where the functions here partly use the 'Rmpfr' package and hence the underlying 'MPFR' and 'GMP' C libraries.

Depends R (>= 3.6.0)

Imports DPQ (>= 0.5-3), Rmpfr (>= 0.9-0), gmp (>= 0.6-4), sfsmisc, stats, graphics, methods, utils

Suggests Matrix

SuggestsNote Matrix for its test-tools-1.R

License GPL (>= 2)

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/DPQmpfr/?root=specfun, svn://svn.r-forge.r-project.org/svnroot/specfun/pkg/DPQmpfr

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

Repository https://r-forge.r-universe.dev

RemoteUrl https://github.com/r-forge/specfun

RemoteRef HEAD

RemoteSha 43df78b48c92771b25a768b97ba720ef644b5fd2

Contents

`DPQmpfr-package` *DPQ (Density, Probability, Quantile) Distribution Computations using MPFR*

Description

An extension to the 'DPQ' package with computations for 'DPQ' (Density (pdf), Probability (cdf) and Quantile) functions, where the functions here partly use the 'Rmpfr' package and hence the underlying 'MPFR' and 'GMP' C libraries.

Details

The DESCRIPTION file: This package was not yet installed at build time.

Index: This package was not yet installed at build time.

Author(s)

Martin Maechler [aut, cre] (<https://orcid.org/0000-0002-8685-9910>)

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

See Also

Packages **DPQ**, **Rmpfr** are both used by this package.

Examples

```
## An example how mpfr-numbers  "just work" with reasonable R functions:
.srch <- search() ; doAtt <- is.na(match("Rmpfr:package", .srch))
if(doAtt) require(Rmpfr)
nu.s <- 2^seqMpfr(mpfr(-30, 64), mpfr(100, 64), by = 1/mpfr(4, 64))
b0 <- DPQ::b_chi(nu.s)
b1 <- DPQ::b_chi(nu.s, one.minus=TRUE)
stopifnot(inherits(b0,"mpfr"), inherits(b1, "mpfr"),
          b0+b1 == 1,  diff(log(b1)) < 0)
plot(nu.s,          log(b1),  type="l", log="x")
plot(nu.s[-1], diff(log(b1)), type="l", log="x")
if(doAtt) # detach the package(s) we've attached above
  for(pkg in setdiff(search(), .srch)) detach(pkg, character.only=TRUE)
```

`algdivM` *Compute log(gamma(b)/gamma(a+b)) Accurately, also via* **Rmpfr**

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.

The name 'algdiv' is from the auxiliary function in R's (TOMS 708) implementation of `pbeta()`. As package **DPQ** provides R's Mathlib (double precision) as R function `algdiv()`, we append 'M' to show the reliance on the **Rmpfr** package.

Usage

```
algdivM(a, b, usePr = NULL)
```

Arguments

a, b	numeric or numeric-alike vectors (recycled to the same length if needed), typically inheriting from `class "mpfr"`.
usePr	positive integer specifying the precision in **bit**s, or `NULL` when a smart default will be used.

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' file, 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` from package **DPQ**.

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)

Martin Maechler (for the **Rmpfr** version).

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`; the (double precision) version `algdiv()` in **DPQ**, and also in **DPQ**, the asymptotic approximation `logQab_asy()`.

Examples

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

## algdivM()  and my  logQab_asy()  give *very* similar results for largish b:
(lQab <- DPQ::logQab_asy(3, 100))
all.equal( - algdivM(3, 100), lQab, tolerance=0) # 1.283e-16 !!
## relative error
1 + lQab/ algdivM(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., algdivM))
## direct computation:
f.algdiv0 <- function(a,b) lgamma(b) - lgamma(a+b)
```

```
f.algdiv1 <- function(a,b) lgamma(b) - lgamma(a+b)
ad.d <- t(outer(a., b., f.algdiv0))

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

---

betaD94 *Ding(1994) (non-central) Beta Distribution Functions*

Description

The three functions "p" (cumulative distribution, CDF), "d" (density (PDF)), and "q" (quantile) use Ding(1994)'s algorithm A, B, and C, respectively, each of which implements a recursion formula using only simple arithmetic and `log` and `exp`.

These are particularly useful also for using with high precision `"mpfr"` numbers from the **Rmpfr** CRAN package.

Usage

```
dbetaD94(x, shape1, shape2, ncp = 0, log = FALSE,
         eps = 1e-10, itrmax = 100000L, verbose = FALSE)
pbetaD94(q, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE,
  log_scale = (a * b > 0) && (a + b > 100 || c >= 500),
         eps = 1e-10, itrmax = 100000L, verbose = FALSE)

qbetaD94(p, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE,
  log_scale = (a * b > 0) && (a + b > 100 || c >= 500),
         delta = 1e-6,
         eps = delta^2,
         itrmax = 100000L,
         iterN = 1000L,
         verbose = FALSE)
```

Arguments

x, q	numeric vector of values in $[0, 1]$ as beta variates.
shape1, shape2	the two shape parameters of the beta distribution, must be positive.

ncp	the noncentrality parameter; by default zero for the (*central*) beta distribution; if positive, we have a noncentral beta distribution.
p	numeric vector of probabilities, log()ged in case log.p is true.
log, log.p	logical indicating if the density or probability values should be log()ged.
lower.tail	logical indicating if the lower or upper tail probability should be computed, or for qbeta*() are provided.
eps	a non-negative number specifying the desired accuracy for computing F() and f().
itrmax	the maximal number of steps for computing F() and f().
delta	[For qbeta*():] non-negative number indicating the desired accuracy for computing x_p (the root of $pbeta * () == p$), i.e., the convergence tolerance for the Newton iterations. This sets default eps = delta^2 which is sensible but may be too small, such that eps should be specified in addition to delta.
iterN	[For qbeta*():] The maximal number of **N**ewton iterations.
log_scale	logical indicating if most of the computations should happen in log scale, which protects from "early" overflow and underflow but takes more computations. The current default is somewhat *arbitrary*, still derived from the facts that gamma(172) overflows to Inf already and exp(-750) underflows to 0 already.
verbose	logical (or integer) indicating the amount of diagnostic output during computation; by default none.

Value

In all three cases, a numeric vector with the same attributes as x (or q respectively), containing (an approximation) to the correponding beta distribution function.

Author(s)

Martin Maechler, notably log_scale was not part of Ding's proposals.

References

Cherng G. Ding (1994) On the computation of the noncentral beta distribution. *Computational Statistics & Data Analysis* **18**, 449–455.

See Also

pbeta. Package **Rmpfr**'s pbetaI() needs both shape1 and shape2 to be integer but is typically more efficient than the current pbetaD94() implementation.

Examples

```
## Low precision (eps, delta) values as "e.g." in Ding(94): ------------------

## Compare with  Table 3  of  Baharev_et_al 2017 %% ===> ./qbBaha2017.Rd <<<<<<<<<<<
aa <- c(0.5, 1, 1.5, 2, 2.5, 3, 5, 10, 25)
```

```
bb <- c(1:15, 10*c(2:5, 10, 25, 50))
utime <-
 qbet <- matrix(NA_real_, length(aa), length(bb),
                dimnames = list(a = formatC(aa), b = formatC(bb)))
(doExtras <- DPQmpfr:::doExtras())
if(doExtras) qbetL <- utimeL <- utime

p <- 0.95
delta <- 1e-4
eps   <- 1e-6
system.t.usr <- function(expr)
  system.time(gcFirst = FALSE, expr)[["user.self"]]

system.time(
for(ia in seq_along(aa)) {
    a <- aa[ia]; cat("\n--==--\na=",a,":\n")
    for(ib in seq_along(bb)) {
        b <- bb[ib]; cat("\n>> b=",b,"\n")
        utime [ia, ib] <- system.t.usr(
         qbet[ia, ib] <-   qbetaD94(p, a, b, ncp = 0, delta=delta, eps=eps, verbose = 2))
        if(doExtras)
          utimeL[ia, ib] <- system.t.usr(
           qbetL[ia, ib] <-   qbetaD94(p, a, b, ncp = 0, delta=delta, eps=eps,
                                       verbose = 2, log_scale=TRUE))
    }
    cat("\n")
}
)# system.time(.): ~ 1 sec (lynne i7-7700T, Fedora 32, 2020)
sum(print(table(round(1000*utime)))) # lynne .. :
##  0  1  2  3  4  5  6  7  8  9 10 11 14 15 16 29
## 53 94 15  3  3 12  2  2  2  2  1  2  3  1  2  1
## [1] 198
if(doExtras) print(sum(print(table(round(1000*utimeL))))) # lynne .. :
```

---

dhyperQ *Exact Hypergeometric Distribution Probabilites*

Description

Computes **exact** probabilities for the hypergeometric distribution (see, e.g., `dhyper()` in R), using package **gmp**'s big integer and rational numbers, notably `chooseZ()`.

Usage

```
dhyperQ(x, m, n, k)
phyperQ(x, m, n, k, lower.tail=TRUE)
phyperQall(m, n, k, lower.tail=TRUE)
```

Arguments

x	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 indicating if the lower or upper tail probability should be computed.

Value

a bigrational (class "bigq" from package **gmp**) vector "as" x; currently of length one (as all the function arguments must be "scalar", currently).

Author(s)

Martin Maechler

See Also

chooseZ (pkg **gmp**), and R's own Hypergeometric

Examples

```
## dhyperQ() is simply
 function (x, m, n, k)
 {
    stopifnot(k - x == as.integer(k - x))
    chooseZ(m, x) * chooseZ(n, k - x) / chooseZ(m + n, k)
 }

# a case where  phyper(11, 15, 0, 12, log=TRUE) gave 'NaN'
(phyp5.0.12 <- cumsum(dhyperQ(0:12, m=15,n=0,k=12)))
stopifnot(phyp5.0.12 == c(rep(0, 12), 1))

for(x in 0:9)
  stopifnot(phyperQ(x, 10,7,8) +
            phyperQ(x, 10,7,8, lower.tail=FALSE) == 1)

(ph. <- phyperQall(m=10, n=7, k=8))
## Big Rational ('bigq') object of length 8:
## [1] 1/2431    5/374     569/4862  2039/4862 3803/4862 4685/4862 4853/4862 1
stopifnot(identical(gmp::c_bigq(list(0, ph.)),
                    1- c(phyperQall(10,7,8, lower.tail=FALSE), 0)))

(doExtras <- DPQmpfr:::doExtras())
if(doExtras) { # too slow for standard testing
 k <- 5000
 system.time(ph <-   phyper(k, 2*k, 2*k, 2*k)) #   0 (< 0.001 sec)
 system.time(phQ <- phyperQ(k, 2*k, 2*k, 2*k)) # 5.6 (was 6.3) sec
 ## Relative error of R's phyper()
```

```
 stopifnot(print(gmp::asNumeric(1 - ph/phQ)) < 1e-14) # seen 1.063e-15
}
```

---

dnt *Non-central t-Distribution Density*

Description

dntJKBm is a fully **Rmpfr**-ified vectorized version of dntJKBf() from **DPQ**; see there.

dtWVm(x, df, ncp) computes 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, using an asymptotic formula for "large" df$= \nu$.

Usage

```
dntJKBm(x, df, ncp, log = FALSE, M = 1000) # __ Deprecated __ use DPQ :: dntJKBf
dtWVm  (x, df, ncp, log = FALSE)
```

Arguments

x	numeric or "mpfr" 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	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.

Details

See dtWV's details (package **DPQ**).

As **DPQ**'s dntJKBf() **is** already fully mpfr-ized, dntJKBm() is deprecated.

Value

an mpfr vector of the same length as the maximum of the lengths of x, df, ncp.

Note

Package **DPQ**'s dntJKBf() **is** already fully mpfr-ized, and hence dntJKBm() is redundant, and therefore deprecated.

Author(s)

Martin Maechler

See Also

R's dt, and package **DPQ**'s dntJKBf() and dtWV().

Examples

```
tt <- seq(0, 10, len = 21)
ncp <- seq(0, 6, len = 31)
dt3R  <- outer(tt, ncp, dt  , df = 3)
dt3WV <- outer(tt, ncp, dtWVm, df = 3)
all.equal(dt3R, dt3WV) # rel.err 0.00063
dt25R  <- outer(tt, ncp, dt  , df = 25)
dt25WV <- outer(tt, ncp, dtWVm, 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 <- dtWVm(x, df = 22, ncp =100, log=TRUE)

head(lfx, 15) # 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,"dtWVm"), dttL)),
       lty=1, lwd=c(1,1,3), col=c("black", cc), bty = "n")
par(opa) # reset


## For dntJKBm(), see  example(dntJKBf, package="DPQ")
```

DPQmpfr-utils *Numeric / Mpfr Utilities for DPQmpfr*

Description

Utilities for package **DPQmpfr**

Usage

```
ldexp(f, E)
```

Arguments

f	'fraction', as such with absolute value in $[0.5, 1)$, but can be any numbers.
E	integer-valued exponent(s).

Details

`ldexp()` is a simple wrapper, either calling `DPQ::ldexp` from **DPQ** or `ldexpMpfr` from the **Rmpfr** package,

$$ldexp(f, E) := f \times 2^E,$$

computed accurately and fast on typical platforms with internally binary arithmetic.

Value

either a numeric or a `"mpfr"`, depending on the type of `f`, vector as (the recyled) combination of `f` and `E`.

See Also

`ldexp` from package **DPQ** and `ldexpMpfr` from package **Rmpfr**.

Examples

```
ldexp(1:10, 2)
ldexp(Rmpfr::Const("pi", 96), -2:2) # =  pi * (1/4  1/2  1  2  4)
```

---

gam1M *Compute 1/Gamma(x+1) - 1 Accurately*

Description

FIXME: "R's own" double prec version is now in package DPQ: e.g. ~/R/Pkgs/DPQ/man/gam1.Rd

FIXME2: R-only implementation is in ~/R/Pkgs/DPQ/TODO_R_versions_gam1_etc.R

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.

Usage

```
gam1M(a, usePr = NULL)
```

Arguments

a	a numeric or numeric-alike, typically inheriting from `class "mpfr"`.
usePr	the precision to use; the default, `NULL`, means to use a default which depends on `a`, specifically `getPrec(a)`.

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

```
require(Rmpfr) # Const(), mpfr(), zeta()
gam <- Const("gamma", 128)
z <- zeta(mpfr(1:7, 128))
(c3 <- (gam^2 -z[2])/2)                    # -0.65587807152025388107701951514 5
(c4 <- (gam*c3 - z[2]*c2 + z[3])/3)         # -0.042002635034095235529003934 88
(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

See Also

gamma.

Examples

```
##' naive direct formula:
g1 <- function(u) 1/gamma(u+1) - 1


##' @title gam1() from TOMS 708 -- translated to R (*and* vectorized)
```

```
##' @author Martin Maechler
gam1R <- function(a, chk=TRUE) { ##  == 1/gamma(a+1) - 1  -- accurately  ONLY for  -0.5 <= a <= 1.5
    if(!length(a)) return(a)
    ## otherwise:
    if(chk) stopifnot(-0.5 <= a, a <= 1.5) # if not, the computation below is non-sense!
    d  <- a - 0.5
    ## t := if(a > 1/2)  a-1  else  a  ==> t in [-0.5, 0.5]  <==>  |t| <= 0.5
    R <- t <- a
    dP <- d > 0
    t[dP] <- d[dP] - 0.5
    if(any(N <- (t < 0.))) { ## L30: */
        r <- c(-.422784335098468, -.771330383816272,
               -.244757765222226, .118378989872749, 9.30357293360349e-4,
               -.0118290993445146, .00223047661158249, 2.66505979058923e-4,
               -1.32674909766242e-4)
        s1 <- .273076135303957
        s2 <- .0559398236957378
        t_ <- t[N]
       top  <- (((((((r[9] * t_ + r[8]) * t_ + r[7]) * t_ + r[6]) * t_ + r[5]) * t_ + r[4]
        ) * t_ + r[3]) * t_ + r[2]) * t_ + r[1]
        bot <- (s2 * t_ + s1) * t_ + 1.
        w <- top / bot
        ## if (d > 0.) :
        if(length(iP <- which(dP[N])))
            R[N &  dP] <- (t_ * w)[iP] / a[N & dP]
        ## else d <= 0 :
        if(length(iN <- which(!dP[N])))
            R[N & !dP] <- a[N & !dP] * (w[iN] + 0.5 + 0.5)
    }
    if(any(Z <- (t == 0))) ## L10: a in {0, 1}
        R[Z] <- 0.
    if(any(P <- t > 0)) { ## t > 0;  L20: */
        p <- c( .577215664901533, -.409078193005776,
               -.230975380857675, .0597275330452234, .0076696818164949,
               -.00514889771323592, 5.89597428611429e-4 )
     q <- c(1., .427569613095214, .158451672430138, .0261132021441447, .00423244297896961)
        t <- t[P]
        top <- (((((p[7] * t + p[6])*t + p[5])*t + p[4])*t + p[3])*t + p[2])*t + p[1]
        bot <- (((q[5] * t + q[4]) * t + q[3]) * t + q[2]) * t + 1.
        w <- top / bot
        ## if (d > 0.) ## L21: */
        if(length(iP <- which(dP[P])))
            R[P &  dP] <- t[iP] / a[P &  dP] * (w[iP] - 0.5 - 0.5)
        ## else d <= 0 :
        if(length(iN <- which(!dP[P])))
            R[P & !dP] <- a[P & !dP] * w[iN]
    }
    R
} ## gam1R()

u <- seq(-.5, 1.5, by=1/16); set.seed(1); u <- sample(u) # permuted (to check logic)
g11  <- vapply(u, gam1R, 1) # [-.5, 1.5] == the interval for which the above gam1() was made
gam1. <- gam1R(u)
```

```
cbind(u, gam1., D = sfsmisc::relErrV(gam1., g1(u)))[order(u),]
                          # looks "too good", as we are not close (but different) to {0, 1}
stopifnot( identical(g11, gam1.) )
           all.equal(g1(u), gam1., tolerance = 0) # 6.7e-16  ("too good", see above)
stopifnot( all.equal(g1(u), gam1.) )

## Comparison using Rmpfr; slightly extending [-.5, 1.5] interval (and getting much closer to {0,1})
u <- seq(-0.525, 1.525, length.out = 2001)
uM <- Rmpfr::mpfr(u, 128)
gam1M. <- gam1M(uM)
relE <- Rmpfr::asNumeric(sfsmisc::relErrV(gam1M., gam1R(u, chk=FALSE)))
rbind(rErr = summary(relE),
     `|rE|` = summary(abs(relE)))
##            Min.    1st Qu.    Median      Mean   3rd Qu.     Max.
## rErr -3.280e-15 -3.466e-16 1.869e-17 1.526e-16 4.282e-16 1.96e-14
## |rE|  1.343e-19  2.363e-16 3.861e-16 6.014e-16 6.372e-16 1.96e-14
stopifnot(max(abs(relE)) < 1e-13)

relEtit <- expression("Relative Error of " ~~ gam1(u) %~%{} == frac(1, Gamma(u+1)) - 1) #%
plot(relE ~ u, type="l", ylim = c(-1,1) * 2.5e-15, main = relEtit)
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 <- Rmpfr::asNumeric(sfsmisc::relErrV(gam1M., g1(u)))

plot(relE ~ u, type="l", ylim = c(-1,1) * 1e-14, main = relEtit)
lines(relED ~ u, col = adjustcolor(2, 1/2), lwd = 2); abline(v = (-1:3)/2, lty=2, col="orange3")
mtext("comparing with direct formula   1/gamma(u+1) - 1", col=2, cex=3/4)
legend("top", c("gam1R(u)", "1/gamma(u+1) - 1"), col = 1:2, lwd=1:2, bty="n")
## direct is clearly *worse* , but not catastrophical
```

`lgamma1pM` *Compute log(Gamma(x+1)) Arbitrarily (MPFR) Accurately*

Description

Computes $\log\Gamma(x+1)$ accurately notably when $|x| \ll 1$. For `"mpfr"` numbers, the precision is increased intermediately such that $a+1$ should not lose precision.

R's "own" double prec version is soon available in package in **DPQ**, under the name `gamln1()` (from TOMS 708).

Usage

```
lgamma1pM(a, usePr = NULL, DPQmethod = c("lgamma1p", "algam1"))
```

Arguments

a	a numeric or numeric-alike vector, typically inheriting from class `"mpfr"`.

usePr	positive integer specifying the precision in **bit**s, or NULL when a smart default will be used.
DPQmethod	a character string; must be the name of an lgamma1p()-alike function from package **DPQ**. It will be called in case of is.numeric(a) (and when **DPQ** is available).

Value

a numeric-alike vector like a.

Author(s)

Martin Maechler

References

TOMS 708, see pbeta

See Also

lgamma() (and gamma() (same page)), and our algdivM(); further, package **DPQ**'s lgamma1p() and (if already available) gamln1().

Examples

```
## Package {DPQ}'s  lgamma1p():
lgamma1p <- DPQ::lgamma1p
lg1 <- function(u) lgamma(u+1) # the simple direct form
u <- seq(-.5, 1.5, by=1/16); set.seed(1); u <- sample(u) # permuted (to check logic)
g11   <- vapply(u, lgamma1p, numeric(1))
lgamma1p. <- lgamma1p(u)
all.equal(lg1(u), g11, tolerance = 0) # see 3.148e-16
stopifnot(exprs = {
    all.equal(lg1(u), g11, tolerance = 2e-15)
    identical(g11, lgamma1p.)
})

## Comparison using Rmpfr; slightly extending the [-.5, 1.5] interval:
u <- seq(-0.525, 1.525, length.out = 2001)
lg1p  <- lgamma1pM(   u)
lg1pM <- lgamma1pM(Rmpfr::mpfr(u, 128))
asNumeric <- Rmpfr::asNumeric
relErrV   <- sfsmisc::relErrV
if(FALSE) { # DPQ "latest" version __FIXME__
lng1  <- DPQ::lngam1(u)
relE <- asNumeric(relErrV(lg1pM, cbind(lgamma1p = lg1p, lngam1 = lng1)))
} else {
relE <- asNumeric(relErrV(lg1pM, cbind(lgamma1p = lg1p)))#, lngam1 = lng1)))
}

## FIXME: lgamma1p() is *NOT* good around u =1. -- even though it should
```

```
##        and the R-only vs (not installed) *does* "work" (is accurate there) ?????
## --> ~/R/Pkgs/DPQ/TODO_R_versions_gam1_etc.R
if(FALSE) {
matplot(u, relE, type="l", ylim = c(-1,1) * 2.5e-15,
     main = expression("relative error of " ~~ lgamma1p(u) == log( Gamma(u+1) )))
} else {
plot(relE ~ u, type="l", ylim = c(-1,1) * 2.5e-15,
     main = expression("relative error of " ~~ lgamma1p(u) == log( Gamma(u+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 <- asNumeric(relErrV(lg1pM, lg1(u)))
lines(relED ~ u, col = adjustcolor(2, 1/2), lwd = 2)
```

`pbeta_ser` *Beta Distribution Function – 'BPSER' Series Expansion from TOMS 708*

Description

Compute a version of the Beta cumulative distribution function (`pbeta()` in R), namely using the series expansion, named `BPSER()`, from "TOMS 708", i.e., Didonato and Morris (1992).

This "pure R" function exists for didactical or documentational reasons on one hand, as R's own `pbeta()` uses this expansion when appropriate and other algorithms otherwise. On the other hand, using high precision q and MPFR arithmetic (via package **Rmpfr**) may allow to get highly accurate `pbeta()` values.

Usage

```
pbeta_ser(q, shape1, shape2, log.p = FALSE, eps = 1e-15, errPb = 0, verbose = FALSE)
```

Arguments

q, shape1, shape2	quantiles and shape parameters of the Beta distribution, `q` typically in $[0, 1]$, see `pbeta`. Here, `q` *must be scalar*, i.e., of length one, and may inherit from `class "mpfr"`, in order to be more accurate (than with the double precision computations).
`log.p`	if TRUE, probabilities `p` are given as `log(p)`.
`eps`	non-negative number; `tol <- eps/shape1` will be used for convergence checks in the series computations.
`errPb`	an integer code, typically in `-2, -1, 0` to determine how warnings on convergence failures are handled.
`verbose`	logical indicating if console output about intermediate results should be printed.

Details

pbeta_ser() crucially needs three auxiliary functions which we "mpfr-ized" as well: gam1M(), lgamma1pM(), and algdivM.

Value

An approximation to the Beta probability $P[X \le q]$ for $X \sim B(a, b)$, (where a =shape1, and b =shape2).

Author(s)

Didonato and Morris and R Core team; separate packaging by 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; doi:10.1145/131766.131776.

See Also

pbeta, **DPQmpfr**'s own pbetaD94; even more pbeta() approximations in package **DPQ**, e.g., pnbetaAS310, or pbetaRv1.

In addition, for integer shape parameters, the potentially "fully accurate" finite sum base pbetaI() in package **Rmpfr**.

Examples

```
(p. <- pbeta_ser(1/2, shape1 = 2, shape2 = 3, verbose=TRUE))
(lp <- pbeta_ser(1/2, shape1 = 2, shape2 = 3, log.p = TRUE))
          all.equal(lp, log(p.), tolerance=0) # 1.48e-16
stopifnot(all.equal(lp, log(p.), tolerance = 1e-13))

## Using  Vectorize() in order to allow vector 'q' e.g. for curve():
str(pbetaSer <- Vectorize(pbeta_ser, "q"))
curve(pbetaSer(x, 1.5, 4.5)); abline(h=0:1, v=0:1, lty=2, col="gray")
curve(pbeta   (x, 1.5, 4.5), add=TRUE, col = adjustcolor(2, 1/4), lwd=3)

## now using mpfr-numbers:
half <- 1/Rmpfr::mpfr(2, 256)
(p2 <- pbeta_ser(half, shape1 = 1, shape2 = 123))
```

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 `"mpfr"` vector (or `array`).
`lower.tail, log.p`	logical, see, e.g., `pnorm()`.

Value

vector/array/mpfr 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`. The same functions "numeric-only" are in my **DPQ** package.

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") # 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)
col4 <- adjustcolor(1:4, 1/2)
options(width = 111) -> oop # (nicely printing "tables")
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))
doLegTit <- function(col=1:4) {
  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=col, 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))

if(x[length(x)] > 1e150) # the "HUGE" case (not default)
  print( tail(cbind(x, px), 20) )
 ##--> For very large x ~= 1e154, the approximations overflow *later* than pnorm() itself !!


## 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)
abline(h=(1:4)*2^-53, col=adjustcolor(1, 1/4))

## lower.tail=TRUE (w/ log.p=TRUE) works "the same" for x < 0:
require(Rmpfr)
x <- - 2^seq(0,    30, by = 1/64)
##   ==
log1mexp <- Rmpfr::log1mexp # Rmpfr version >= 0.8-2 (2020-11-11 on CRAN)
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))

## Comparison with  Rmpfr::erf() / erfc() based pnorm():

## Set the exponential ranges to maximal -- to evade underflow as long as possible
.mpfr_erange_set(value = (1-2^-52) * .mpfr_erange(c("min.emin","max.emax")))
l2t <- seq(0, 32, by=1/4)
twos <- mpfr(2, 1024)^l2t
Qt  <- pnorm(twos, lower.tail=FALSE)
pnU <- pnormU_S53 (twos, log.p=TRUE)
pnL <- pnormL_LD10(twos, log.p=TRUE)
logQt <- log(Qt)
M <- cbind(twos, Qt, logQt = logQt, pnU)
roundMpfr(M, 40)
dM <- asNumeric(cbind(dU = pnU - logQt,    dL = logQt - pnL,
                      # NB: the numbers are *negative*
                      rdU= 1 - pnU/logQt, rdL = pnL/logQt - 1))
data.frame(l2t, dM)
## The bounds are ok (where Qt does not underflow): L < p < U :
stopifnot(pnU > pnL, pnU > logQt, (logQt > pnL)[Qt > 0])
roundMpfr(cbind(twos, pnL, pnU, D=pnU-pnL, relD=(pnU-pnL)/((pnU+pnL)/2)), 40)

## ----- R's pnorm() -- is it always inside [L, U]  ?? ---------------------
nQt <- stats::pnorm(asNumeric(twos), lower.tail=FALSE, log.p=TRUE)
data.frame(l2t, check.names=FALSE
         , nQt
         , "L <= p" = c(" ", "W")[2 -(pnL <= nQt)]
```

```
         , "p <= U" = c(" ", "W")[2- (nQt <= pnU)])
## ==> pnorm() is *outside* sometimes for l2t >= 7.25; always as soon as l2t >= 9.25

## *but* the relative errors are around c_epsilon  in all these cases :
plot (2^l2t, asNumeric(abs(nQt-pnL)/abs(pnU)), type="o", cex=1/4, log="xy", axes=FALSE)
sfsmisc::eaxis(1, sub10 = 2)
sfsmisc::eaxis(2)
lines(2^l2t, asNumeric(abs(nQt-pnU)/abs(pnU)), type="o", cex=1/4, col=2)
abline(h=c(1:4)*2^-53, lty=2, col=adjustcolor(1, 1/4))

options(oop)# reverting
```

---

pqnormAsymp *Asymptotic Approximations of Extreme Tail 'pnorm()' and 'qnorm()'*

Description

These functions provide the first terms of asymptotic series approximations to `pnorm()`'s (extreme) tail, from Abramawitz and Stegun's 26.2.13 (p.932), or `qnorm()` where the approximations have been derived via iterative plugin using Abramowitz and Stegun's formula.

Usage

```
pnormAsymp(x, k, lower.tail = FALSE, log.p = FALSE)
qnormAsymp(p, lp = .DT_Clog(p, lower.tail = lower.tail, log.p = log.p),
           order, M_2PI =,
           lower.tail = TRUE, log.p = missing(p))
```

Arguments

`x`	positive (at least non-negative) numeric vector.
`k`	integer ≥ 0 indicating how many terms the approximation should use; currently $k \leq 5$.
`p`	numeric vector of probabilities, possibly transformed, depending on `log.p`. Does not need to be specified, if `lp` is 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.
`M_2PI`	the number 2π in the same precision as `p` or `lp`, i.e., `numeric` or of class `"mpfr"`.
`lower.tail`	logical; if true, probabilities are $P[X \leq x]$, otherwise upper tail probabilities, $P[X > x]$.
`log.p`	logical; if `TRUE` (default for `qnormAsymp` !!), probabilities p are given as $\log(p)$ in argument `p` or $\log(1 - p)$ in `lp`.

Details

see *both* help pages `pnormAsymp` and `qnormAsymp` from our package **DPQ**.

Value

vector/array/mpfr like first argument x or p or lp, respectively.

Author(s)

Martin Maechler

See Also

pnorm. The same functions "numeric-only" are in my **DPQ** package with more extensive documentation.

Examples

```
require("Rmpfr") # (in strong dependencies of this pkg {DPQmpfr})
x <- seq(1/64, 10, by=1/64)
xm  <- mpfr( x, 96)
"TODO"

## More extreme tails: -----------------------------------------------
##
## 1. pnormAsymp() ---------------------
lx <- c((2:10)*2, 25, (3:9)*10, (1:9)*100, (1:8)*1000, (2:7)*5000)
lxm <- mpfr(lx, 256)
Px <- pnorm(lxm, lower.tail = FALSE, log.p=TRUE)
PxA <- sapplyMpfr(setNames(0:5, paste("k =",0:5)),
                  pnormAsymp, x=lxm, lower.tail = FALSE, log.p=TRUE)
if(interactive())
  roundMpfr(PxA, 40)
# rel.errors :
relE <- asNumeric(1 - PxA/Px)
options(width = 99) -> oop # (nicely printing the matrices)
cbind(lx, relE)
matplot(lx, abs(relE), type="b", cex = 1/2, log="xy", pch=as.character(0:5),
        axes=FALSE,
        main = "|relE( <pnormAsymp(lx, k=*, lower.tail=FALSE, log.p=TRUE) )|")
sfsmisc::eaxis(1, sub10=2); sfsmisc::eaxis(2)
legend("bottom", paste("k =", 0:5), col=1:6, lty=1:5,
       pch = as.character(0:5), pt.cex=1/2, bty="n")
## NB: rel.Errors go down to  7e-59 ==> need precision of  -log2(7e-59) ~ 193.2 bits

## 2. qnormAsymp() ---------------------
QPx <- sapplyMpfr(setNames(0:5, paste("k =",0:5)),
                  function(k) qnormAsymp(Px, order=k, lower.tail = FALSE, log.p=TRUE))
(relE.q <- asNumeric(QPx/lx - 1))
         # note how consistent the signs are (!) <==> have upper/lower bounds

matplot(-asNumeric(Px), abs(relE.q), type="b", cex = 1/2, log="xy", pch=as.character(0:5),
        xlab = quote(-Px), axes=FALSE,
        main = "|relE( <qnormAsymp(Px, k=*, lower.tail=FALSE, log.p=TRUE) )|")
sfsmisc::eaxis(1, sub10=2); sfsmisc::eaxis(2)
legend("bottom", paste("k =", 0:5), col=1:6, lty=1:5,
```

```
        pch = as.character(0:5), pt.cex=1/2, bty="n")

options(oop) # {revert to previous state}
```

---

qbBaha2017 *Accurate qbeta() values from Baharev et al (2017)'s Program*

Description

Compuate "accurate" `qbeta()` values from Baharev et al (2017)'s Program.

Usage

```
data("qbBaha2017")
```

Format

FIXME: Their published table only shows 6 digits, but running their (32-bit statically linked) Linux executable '`mindiffver`' (from their github repos, see "source") with their own '`input.txt`' gives *12* digits accuracy, which we should be able to increase even more, see https://github.com/baharev/mindiffver/blob/master/README.md

A numeric matrix, 9×22 with guaranteed accuracy `qbeta(0.95, a,b)` values, for $a = 0.5, 1, 1.5, 2, 2.5, 3, 5, 10, 25$ and $b =$ with `str()`

```
num [1:9, 1:22] 0.902 0.95 0.966 0.975 0.98 ...
- attr(*, "dimnames")=List of 2
 ..$ a: chr [1:9] "0.5" "1" "1.5" "2" ...
 ..$ b: chr [1:22] "1" "2" "3" "4" ...
```

Details

MM constructed this data as follows (TODO: say more..):

```
ff <- "~/R/MM/NUMERICS/dpq-functions/beta-gamma-etc/Baharev_et_al-2017_table3.txt"
   qbB2017 <- t( data.matrix(read.table(ff)) )
   dimnames(qbB2017) <- dimnames(qbet)
   saveRDS(qbB2017, "..../qbBaha2017.rds")
```

Source

This matrix comprises all entries of Table 3, p. 776 of
Baharev, A., Schichl, H. and Rév, E. (2017) Computing the noncentral-F distribution and the power of the F-test with guaranteed accuracy; *Comput. Stat.* **32**(2), 763–779. doi:10.1007/s00180016-07013

The paper mentions the first author's '`github`' repos where source code and executables are available from: https://github.com/baharev/mindiffver/

Examples

```
data(qbBaha2017)
str(qbBaha2017)
str(ab <- lapply(dimnames(qbBaha2017), as.numeric))
stopifnot(ab$a == c((1:6)/2, 5, 10, 25),
          ab$b == c(1:15, 10*c(2:5, 10, 25, 50)))
matplot(ab$b, t(qbBaha2017)[,9:1], type="l", log = "x", xlab = "b",
        ylab = "qbeta(.95, a,b)",
        main = "Guaranteed accuracy 95% percentiles of Beta distribution")
legend("right", paste("a = ", format(ab$a)),
       lty=1:5, col=1:6, bty="n")

## Relative error of R's qbeta() -- given that the table only shows 6
## digits, there is *no* relevant error: R's qbeta() is accurate enough:
x.ab <- do.call(expand.grid, ab)
matplot(ab$b, 1 - t(qbeta(0.95, x.ab$a, x.ab$b) / qbBaha2017),
        main = "rel.error of R's qbeta() -- w/ 6 digits, it is negligible",
        ylab = "1 - qbeta()/'true'",
        type = "l", log="x", xlab="b")
abline(h=0, col=adjustcolor("gray", 1/2))
```

`stirlerrM` *Stirling Formula Approximation Error*

Description

Compute the `log()` of the error of Stirling's formula for $n!$. Used in certain accurate approximations of (negative) binomial and Poisson probabilities.

`stirlerrM()` currently simply uses the direct mathematical formula, based on `lgamma()`, adapted for use with `mpfr`-numbers.

Usage

```
stirlerrM(n, minPrec = 128L)
stirlerrSer(n, k)
```

Arguments

`n`	numeric or "numeric-alike" vector, typically "large" positive integer or half integer valued, here typically an `"mpfr"`-number vector.
`k`	integer *scalar*, now in `1:22`.
`minPrec`	minimal precision (in bits) to be used when coercing number-alikes, say, biginteger (`bigz`) to `"mpfr"`.

Details

Stirling's approximation to $n!$ has been

$$n! \approx (\frac{n}{e})^n \sqrt{2\pi n},$$

where by definition the error is the difference of the left and right hand side of this formula, in log-scale,

$$\delta(n) = \log\Gamma(n+1) - n\log(n) + n - \log(2\pi n)/2.$$

See the vignette *log1pmx, bd0, stirlerr, ...* from package **DPQ**, where the series expansion of $\delta(n)$ is used with 11 terms, starting with

$$\delta(n) = \frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5} \pm O(n^{-7}).$$

Value

a numeric or other "numeric-alike" class vector, e.g., `mpfr`, of the same length as `n`.

Note

In principle, the direct formula should be replaced by a few terms of the series in powers of $1/n$ for large `n`, but we assume using high enough precision for `n` should be sufficient and "easier".

Author(s)

Martin Maechler

References

Catherine Loader, see `dbinom`;

Martin Maechler (2021) log1pmx(), bd0(), stirlerr() – Computing Poisson, Binomial, Gamma Probabilities in R. https://CRAN.R-project.org/package=DPQ/vignettes/log1pmx-etc.pdf

See Also

`dbinom`; rational exact `dbinomQ()` in package **gmp**. `stirlerr()` in package **DPQ** which is a pure R version R's mathlib-internal C function.

Examples

```
### ----------------  Regular R  double precision ------------------------------

n <- n. <- c(1:10, 15, 20, 30, 50*(1:6), 100*(4:9), 10^(3:12))
(stE <- stirlerrM(n)) # direct formula is *not* good when n is large:
require(graphics)
plot(stirlerrM(n) ~ n, log = "x", type = "b", xaxt="n") # --> *negative for large n!
sfsmisc::eaxis(1, sub10=3) ; abline(h = 0, lty=3, col=adjustcolor(1, 1/2))
oMax <- 22 # was 8 originally
str(stirSer <- sapply(setNames(1:oMax, paste0("k=",1:oMax)),
                      function(k) stirlerrSer(n, k)))
```

```
cols <- 1:(oMax+1)
matlines(n, stirSer, col = cols, lty=1)
leg1 <- c("stirlerrM(n): [dble prec] direct f()",
          paste0("stirlerrSer(n, k=", 1:oMax, ")"))
legend("top", leg1, pch = c(1,rep(NA,oMax)), col=cols, lty=1, bty="n")
## for larger n, current values are even *negative* ==> dbl prec *not* sufficient

## y in log-scale [same conclusion]
plot (stirlerrM(n) ~ n, log = "xy", type = "b", ylim = c(1e-13, 0.08))
matlines(n, stirSer, col = cols, lty=1)
legend("bottomleft", leg1, pch=c(1,rep(NA,oMax)), col=1:(oMax+1), lty=1, bty="n")

## the numbers:
options(digits=4, width=111)
stEmat. <- cbind(sM = setNames(stirlerrM(n),n), stirSer)
stEmat.
# note *bad* values for (n=1, k >= 8) !


## for printing n=<nice>: 8
N <- Rmpfr::asNumeric
dfm <- function(n, mm) data.frame(n=formatC(N(n)), N(mm), check.names=FALSE)
## relative differences:
dfm(n, stEmat.[,-1]/stEmat.[,1] - 1)
    # => stirlerrM() {with dbl prec} deteriorates after ~ n = 200--500



### ----------------  MPFR High Accuracy -------------------------------

stopifnot(require(gmp),
          require(Rmpfr))
n <- as.bigz(n.)
## now repeat everything .. from above ... FIXME shows bugs !
## fully accurate using big rational arithmetic
class(stEserQ <- sapply(setNames(1:oMax, paste0("k=",1:oMax)),
                        function(k) stirlerrSer(n=n, k=k))) # list ..
stopifnot(sapply(stEserQ, class) == "bigq") # of exact big rationals
str(stEsQM  <- lapply(stEserQ, as, Class="mpfr"))# list of oMax;  each prec. 128..702
    stEsQM. <- lapply(stEserQ, .bigq2mpfr, precB = 512) # constant higher precision
    stEsQMm <- sapply(stEserQ, asNumeric) # a matrix -- "exact" values of Stirling series

stEM  <- stirlerrM(mpfr(n, 128)) # now ok (loss of precision, but still ~ 10 digits correct)
stEM4k <- stirlerrM(mpfr(n, 4096))# assume practically "perfect"/ "true" stirlerr() values
## ==> what's the accuracy of the 128-bit 'stEM'?
N <- asNumeric # short
dfm <- function(n, mm) data.frame(n=formatC(N(n)), N(mm), check.names=FALSE)
dfm(n, stEM/stEM4k - 1)
## ...........
## 29 1e+06  4.470e-25
## 30 1e+07 -7.405e-23
## 31 1e+08 -4.661e-21
## 32 1e+09 -7.693e-20
```

```
## 33 1e+10  3.452e-17  (still ok)
## 34 1e+11 -3.472e-15  << now start losing  --> 128 bits *not* sufficient!
## 35 1e+12 -3.138e-13  <<<<
## same conclusion via  number of correct (decimal) digits:
dfm(n, log10(abs(stEM/stEM4k - 1)))

plot(N(-log10(abs(stEM/stEM4k - 1))) ~ N(n), type="o", log="x",
     xlab = quote(n), main = "#{correct digits} of 128-bit stirlerrM(n)")
ubits <- c(128, 52) # above 128-bit and double precision
abline(h = ubits* log10(2), lty=2)
text(1, ubits* log10(2), paste0(ubits,"-bit"), adj=c(0,0))

stopifnot(identical(stirlerrM(n), stEM)) # for bigz & bigq, we default to precBits = 128
all.equal(roundMpfr(stEM4k, 64),
          stirlerrSer (n., oMax)) # 0.00212 .. because of 1st few n.  ==> drop these
all.equal(roundMpfr(stEM4k,64)[n. >= 3], stirlerrSer (n.[n. >= 3], oMax)) # 6.238e-8

plot(asNumeric(abs(stirlerrSer(n., oMax) - stEM4k)) ~ n.,
     log="xy", type="b", main="absolute error of stirlerrSer(n, oMax)  & (n, 5)")
abline(h = 2^-52, lty=2); text(1, 2^-52, "52-bits", adj=c(1,-1)/oMax)
lines(asNumeric(abs(stirlerrSer(n., 5) - stEM4k)) ~ n., col=2)

plot(asNumeric(stirlerrM(n)) ~ n., log = "x", type = "b")
for(k in 1:oMax) lines(n, stirlerrSer(n, k), col = k+1)
legend("top", c("stirlerrM(n)", paste0("stirlerrSer(n, k=", 1:oMax, ")")),
       pch=c(1,rep(NA,oMax)), col=1:(oMax+1), lty=1, bty="n")

## y in log-scale
plot(asNumeric(stirlerrM(n)) ~ n., log = "xy", type = "b", ylim = c(1e-13, 0.08))
for(k in 1:oMax) lines(n, stirlerrSer(n, k), col = k+1)
legend("topright", c("stirlerrM(n)", paste0("stirlerrSer(n, k=", 1:oMax, ")")),
       pch=c(1,rep(NA,oMax)), col=1:(oMax+1), lty=1, bty="n")
## all "looks" perfect (so we could skip this)

## The numbers ...  reused
## stopifnot(sapply(stEserQ, class) == "bigq") # of exact big rationals
## str(stEsQM  <- lapply(stEserQ, as, Class="mpfr"))# list of oMax;  each prec. 128..702
##     stEsQM. <- lapply(stEserQ, .bigq2mpfr, precB = 512) # constant higher precision
##    stEsQMm <- sapply(stEserQ, asNumeric) # a matrix -- "exact" values of Stirling series

## stEM  <- stirlerrM(mpfr(n, 128)) # now ok (loss of precision, but still ~ 10 digits correct)
## stEM4k <- stirlerrM(mpfr(n, 4096))# assume "perfect"
stEmat <- cbind(sM = stEM4k, stEsQMm)
signif(asNumeric(stEmat), 6) # prints nicely -- large n = 10^e: see ~= 1/(12 n)  =  0.8333 / n
## print *relative errors* nicely :
## simple double precision version of direct formula (cancellation for n >> 1 !):
stE <- stirlerrM(n.) # --> bad for small n;  catastrophically bad for n >= 10^7
## relative *errors*
dfm(n , cbind(stEsQMm, dbl=stE)/stEM4k - 1)
## only "perfect" Series (showing true mathematical approx. error; not *numerical*
relE <- N(stEsQMm / stEM4k - 1)
dfm(n, relE)
matplot(n, relE, type = "b", log="x", ylim = c(-1,1) * 1e-12)
```

```
## |rel.Err|  in  [log log]
matplot(n, abs(N(relE)), type = "b", log="xy")
matplot(n, pmax(abs(N(relE)), 1e-19), type = "b", log="xy", ylim = c(1e-17, 1e-12))
matplot(n, pmax(abs(N(relE)), 1e-19), type = "b", log="xy", ylim = c(4e-17, 1e-15))
abline(h = 2^-(53:52), lty=3)
```

Index