

# Bessel Functions in other CRAN Packages

Martin Mächler  
ETH Zurich

---

## Abstract

Why do I write yet another R package, when R itself has Bessel functions and several CRAN packages also have versions of these?

Short answer: I myself added the Bessel functions to R version 0.63, second half of 1998, but they have been seen to be limited for “large”  $x$  and / or large order  $\nu$ .

*Keywords:* Bessel Functions, Accuracy, R.

---

## 1. Introduction

R itself has had the function `besselI()`, `besselJ()`, `besselK()` and `besselY()`, from very early on. Specifically, I myself added them to R version 0.63, in 1998. This helped quite a bit to attract people from computational finance to R in these early times. For some reason I must have been under the impression that the Fortran code I ported to C and interfaced with R to be state of the art at the time, even though I now doubt it.

However, they had shown deficiencies: First, they did only work for real (`double`) but not for *complex* arguments, even though the Bessel functions are well-defined on the whole complex plain. Second, for  $x \approx 1500$  and larger, `besselI(x, nu, expon.scaled=TRUE)` jumped to zero, as I found, because of an overflow in the backward recursion (via difference equation), which I found elegantly to resolve (by re-scaling), for R2.9.0. However, the algorithm complexity is proportional to  $\lfloor x \rfloor$ , and for large  $x$ , a better algorithm has been desired for years. Hence, I had started experimenting with the two asymptotic expansions from [Abramowitz and Stegun \(1970\)](#).

The following R packages on CRAN (as of Jan.29, 2009) also provide Bessel functions:

**gsl** See Section `refsec:gsl` below

**Rmpfr** provides arbitrary precision Bessel functions of *integer* order  $\nu =: n$  of the first kind only,  $J_n(x) = \text{jn}(n, x)$  and  $Y_n(x) = \text{yn}(n, x)$  (and `j0()`, `j1`, `y0`, `y1`) and—since MPFR version 3.0.0—the Airy function  $Ai(x) = \text{Ai}(x)$ .

```
> suppressPackageStartupMessages(require("Rmpfr"))
```

**QRMLib** Uses many ‘GSL’ (GNU Scientific Library) C functions in its own code, or, rather, has copy-pasted “Bessel-related” parts of GSL into its own ‘src/’ directory.

Notably ‘QRMLib/src/bessel.c’ is a copy (slightly modified to work as “standalone” in the QRMLib sources) of ‘GSL’s ‘specfunc/bessel.c’ but has not been adapted to the latest

GSL sources. Further note that **QRMLib** only provides function `besselM3()`: “M3” for the modified Bessel function of the 3rd kind, i.e.,  $K()$ ; note that it already has optional argument `logvalue=FALSE` and will call ‘GSL’s `gsl_sf_bessel_lnKnu_e()` for `logvalue=TRUE`. Note that it calls different GSL routines for *integer*  $\nu$  ( $=: n$  in that case) than for non-integer which presumably has at least computational advantages.

**GeneralizeHyperbolic** (todo)

**ghyp** (todo)

**CircularDDM** provides (a **Rcpp** and **gsl** based) function `besselzero(nu, k, kind)` to compute the first  $k$  zeros of the  $J_\nu()$  (`kind=1`) and  $Y_\nu()$  (`kind=0`) functions but fails to work for  $I_\nu()$  (`kind=0`) where there is one zero for negative  $\nu \in [-2k, -2k + 1]$ ,  $k = 1, 2, \dots$

## 2. Package ‘gsl’

The R package **gsl** by Robin Hankin provides an R interface on a function-by-function basis to much of the GSL, the GNU Scientific Library. You get a first overview with

```
> library(gsl)
```

```
> ?bessel_Knu
```

```
> ?Airy
```

where the `?bessel_Knu` lists all “Bessel” functions and `?Airy` additionally the “Airy” functions  $Ai()$  and  $Bi()$  and their derivatives which are strongly related to the Bessel functions (and can be defined via them).

Indeed, the GSL and hence the R package **gsl** does contain quite an array of Bessel functions and the Airy functions, we can also get via

```
> igsl <- match("package:gsl", search())
> aB <- apropos("Bessel", where=TRUE); unname(aB)[names(aB) == igsl]
```

```
[1] "bessel_I0"           "bessel_I0_scaled"
[3] "bessel_I1"           "bessel_I1_scaled"
[5] "bessel_In"           "bessel_In_array"
[7] "bessel_In_scaled"   "bessel_In_scaled_array"
[9] "bessel_Inu"          "bessel_Inu_scaled"
[11] "bessel_J0"           "bessel_J1"
[13] "bessel_Jn"           "bessel_Jn_array"
[15] "bessel_Jnu"          "bessel_K0"
[17] "bessel_K0_scaled"   "bessel_K1"
[19] "bessel_K1_scaled"   "bessel_Kn"
[21] "bessel_Kn_array"    "bessel_Kn_scaled"
[23] "bessel_Kn_scaled_array" "bessel_Knu"
```

```

[25] "bessel_Knu_scaled"      "bessel_Y0"
[27] "bessel_Y1"              "bessel_Yn"
[29] "bessel_Yn_array"        "bessel_Ynu"
[31] "bessel_i0_scaled"       "bessel_i1_scaled"
[33] "bessel_i2_scaled"       "bessel_i1_scaled"
[35] "bessel_i1_scaled_array" "bessel_j0"
[37] "bessel_j1"              "bessel_j2"
[39] "bessel_j1"              "bessel_j1_array"
[41] "bessel_j1_steep_array"  "bessel_k0_scaled"
[43] "bessel_k1_scaled"       "bessel_k2_scaled"
[45] "bessel_k1_scaled"       "bessel_k1_scaled_array"
[47] "bessel_lnKnu"           "bessel_sequence_Jnu"
[49] "bessel_y0"              "bessel_y1"
[51] "bessel_y2"              "bessel_y1"
[53] "bessel_y1_array"        "bessel_zero_J0"
[55] "bessel_zero_J1"         "bessel_zero_Jnu"

```

```
> aA <- apropos("Airy", where=TRUE); unname(aA)[names(aA) == igs1]
```

```

[1] "airy_Ai"                "airy_Ai_deriv"        "airy_Ai_deriv_scaled"
[4] "airy_Ai_scaled"         "airy_Bi"               "airy_Bi_deriv"
[7] "airy_Bi_deriv_scaled"  "airy_Bi_scaled"       "airy_zero_Ai"
[10] "airy_zero_Ai_deriv"    "airy_zero_Bi"         "airy_zero_Bi_deriv"

```

Features (and drawbacks):

- only real 'x', not complex
- provides separate functions for *integer* and *fractional*  $\nu$  where the latter should be more general than the former (untested in detail though).
- For *fractional*  $\nu$ , the relevant, i.e., interesting functions are

```

bessel_Jnu      (nu, x, give=FALSE, strict=TRUE)

bessel_Ynu      (nu, x, give=FALSE, strict=TRUE)

bessel_Inu      (nu, x, give=FALSE, strict=TRUE)
bessel_Inu_scaled(nu, x, give=FALSE, strict=TRUE)

bessel_Knu      (nu, x, give=FALSE, strict=TRUE)
bessel_Knu_scaled(nu, x, give=FALSE, strict=TRUE)
bessel_lnKnu    (nu, x, give=FALSE, strict=TRUE)

```

where the `*_scaled()` version of each corresponds to our functions `expon.scaled=TRUE`.

- For fractional  $\nu$ , the (only) interesting functions are

```

> lst <- ls(patt="bessel_.*nu", pos="package:gsl")
> l2 <- sapply(lst, function(.) args(get(.)), simplify=FALSE)
> lnms <- setNames(format(lst), lst)
> arglst <- lapply(lst, ## a bit ugly, using deparse(.)
+   function(nm) sub(" *$", "", sub("^function", lnms[[nm]], deparse(l2[[nm]]))[[1]]))
> .tmp <- lapply(arglst, function(.) cat(format(.), "\n"))

```

```

bessel_Inu      (nu, x, give = FALSE, strict = TRUE)
bessel_Inu_scaled (nu, x, give = FALSE, strict = TRUE)
bessel_Jnu      (nu, x, give = FALSE, strict = TRUE)
bessel_Knu      (nu, x, give = FALSE, strict = TRUE)
bessel_Knu_scaled (nu, x, give = FALSE, strict = TRUE)
bessel_Ynu      (nu, x, give = FALSE, strict = TRUE)
bessel_lnKnu    (nu, x, give = FALSE, strict = TRUE)
bessel_sequence_Jnu (nu, v, mode = 0, give = FALSE, strict = TRUE)
bessel_zero_Jnu (nu, s, give = FALSE, strict = TRUE)

```

where the `*_scaled()` version of each function corresponds to our functions with option `expon.scaled=TRUE`.

- `bessel_Inu_scaled()` works for large  $x$ , comparably to our `BesselI(.)` which give warnings about accuracy loss here :

```

> x <- (1:500)*50000; b2 <- BesselI(x, pi, expo=TRUE)
> b1 <- bessel_Inu_scaled(pi, x)
> all.equal(b1,b2,tol=0) ## "Mean relative difference: 1.544395e-12"

```

```
[1] "Mean relative difference: 1.849828e-12"
```

```

> ## the accuracy is *as* limited (probably):
> b1 <- bessel_Inu_scaled(pi, x, give=TRUE)
> summary(b1$err)

```

```

      Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
8.299e-08 9.580e-08 1.173e-07 1.606e-07 1.655e-07 1.856e-06

```

where the GSL (info) manual says that `err` is an *absolute* error estimate, hence for *relative* error estimates, we look at

```
> range(b1$err/ b1$val)
```

```
[1] 0.001040159 0.001040161
```

So, we see that either the error estimate is too conservative, or the results only have 3 digit accuracy.

- $J_\nu(.)$ : Here (also), the GSL employs different algorithms in different regions, notably also several asymptotic formula. When  $x < \nu$ , notably  $0 \approx x \ll \nu$ , it does not seem to be ok, in the the “left tail”, returning NaN, for moderate  $\nu$ :

```
> bessel_Jnu(100, 2seq(-5,1, by=1/4))

 [1]          NaN          NaN          NaN          NaN          NaN
 [6] 1.098354e-301 3.685445e-294 1.236620e-286 4.149362e-279 1.392272e-271
 [11] 4.671585e-264 1.567474e-256 5.259330e-249 1.764625e-241 5.920564e-234
 [16] 1.986357e-226 6.663900e-219 2.235461e-211 7.498243e-204 2.514703e-196
 [21] 8.431829e-189 2.826353e-181 9.469925e-174 3.171070e-166 1.060953e-158
```

```
> bessel_Jnu( 20, 2seq(-50,-40, by=1/2))

 [1]          NaN          NaN          NaN          NaN          NaN
 [6]          NaN 4.217737e-308 4.318963e-305 4.422618e-302 4.528761e-299
 [11] 4.637451e-296 4.748750e-293 4.862720e-290 4.979426e-287 5.098932e-284
 [16] 5.221306e-281 5.346617e-278 5.474936e-275 5.606335e-272 5.740887e-269
 [21] 5.878668e-266
```

```
> bessel_Jnu( 5, 2seq(-210,-200, by=.5))

 [1]          NaN          NaN          NaN          NaN          NaN
 [6]          NaN          NaN          NaN          NaN          NaN
 [11]          NaN          NaN          NaN          NaN          NaN
 [16]          NaN 2.373412e-308 1.342605e-307 7.594919e-307 4.296335e-306
 [21] 2.430374e-305
```

giving NaN instead of just underflowing to zero. However, looking at the phenomenon shows that it is only because of the `gsl`'s default optional argument `strict = TRUE`: The underflow to zero which no longer allows the error to be controlled (and returned in `err` when `give = TRUE`), giving `status = 15` here:

```
> as.data.frame(bessel_Jnu( 20, 2seq(-50,-40, by=1/2), give=TRUE, strict=FALSE))

   val          err status
1  0.000000e+00 2.225074e-308    15
2  0.000000e+00 2.225074e-308    15
3  0.000000e+00 2.225074e-308    15
4  0.000000e+00 2.225074e-308    15
5  0.000000e+00 2.225074e-308    15
6  0.000000e+00 2.225074e-308    15
7  4.217737e-308 2.371515e-322     0
8  4.318963e-305 2.397503e-319     0
9  4.422618e-302 2.455046e-316     0
10 4.528761e-299 2.513967e-313     0
11 4.637451e-296 2.574303e-310     0
12 4.748750e-293 2.636086e-307     0
13 4.862720e-290 2.699352e-304     0
14 4.979426e-287 2.764136e-301     0
15 5.098932e-284 2.830476e-298     0
16 5.221306e-281 2.898407e-295     0
```

```

17 5.346617e-278 2.967969e-292    0
18 5.474936e-275 3.039200e-289    0
19 5.606335e-272 3.112141e-286    0
20 5.740887e-269 3.186832e-283    0
21 5.878668e-266 3.263316e-280    0

```

If we do use `strict = FALSE`, consequently, all is fine:

```

> gslJ <- function(nu, f1 = .90, f2 = 1.10, nout = 512, give=FALSE, strict=FALSE) {
+   stopifnot(is.numeric(nu), length(nu) == 1, nout >= 1, f1 <= 1, f2 >= 1)
+   x <- seq(f1*nu, f2*nu, length.out = nout)
+   list(x=x, Jnu.x = bessel_Jnu(nu, x, give=give, strict=strict))
+ }
> plJ <- function(nu, f1 = .90, f2=1.10, nout=512,
+   col=2, lwd=2, main = bquote(nu == .(nu)), ...) {
+   dJ <- gslJ(nu, f1=f1, f2=f2, nout=nout)
+   plot(Jnu.x ~ x, data=dJ, type="l", col=col, lwd=lwd, main=main, ...)
+   abline(h=0, lty=3, col=adjustcolor(1, 0.5))
+   invisible(dJ)
+ }
> sfsmisc::mult.fig(4)
> plJ(500, f1=0)
> r1k <- plJ(1000, f1=0)
> head(as.data.frame(r1k)) # all 0 now (NaN's for 'strict=TRUE' !!)

```

```

      x Jnu.x
1 0.000000    0
2 2.152642    0
3 4.305284    0
4 6.457926    0
5 8.610568    0
6 10.763209   0

```

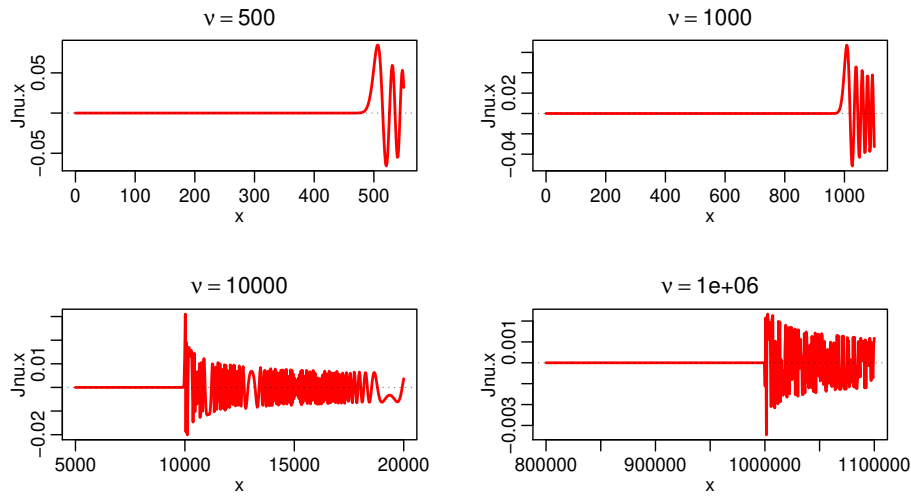
```

> r10k <- plJ(10000, f1=0.5, f2=2)
> str( with(r10k, x[!is.finite(Jnu.x)]) ) # empty; had all NaN upto x = 8317

```

```
num(0)
```

```
> r1M <- plJ(1e6, f1=0.8)
```



### 3. Session Info

```
> toLatex(sessionInfo(), locale=FALSE)
```

- R version 3.6.0 RC (2019-04-24 r76422), x86\_64-pc-linux-gnu
- Running under: Fedora 28 (Twenty Eight)
- Matrix products: default
- BLAS: /sfs/u/maechler/R/D/r-pre-rel/64-linux-inst/lib/libRblas.so
- LAPACK: /sfs/u/maechler/R/D/r-pre-rel/64-linux-inst/lib/libRlapack.so
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: Bessel 0.6-0, Rmpfr 0.7-2, gmp 0.5-13.5, gsl 2.1-6
- Loaded via a namespace (and not attached): compiler 3.6.0, sfsmisc 1.1-3, tools 3.6.0

Date (run in R): 2019-04-25

### References

Abramowitz M, Stegun IA (1970). *Handbook of Mathematical Functions*. Dover Publications, N. Y.

**Affiliation:**

Martin Mächler

Seminar für Statistik, HG G 16

ETH Zurich

8092 Zurich, Switzerland

E-mail: [maechler@stat.math.ethz.ch](mailto:maechler@stat.math.ethz.ch)

URL: <http://stat.ethz.ch/people/maechler>