

Helping R to be (even more) Accurate

Martin Mächler

maechler@R-project.org

The R Core Team

maechler@stat.math.ethz.ch

Seminar für Statistik

ETH Zurich, Switzerland

useR! – July 12, 2018



Outline

- 1 Preamble – Kudos to an Aussie Artist
- 2 Introduction
- 3 FAQ 7.31 — generalized: Loss of Accuracy
- 4 Accurately computing $\log 1\text{mexp}(x) := \log(1 - \exp(-x))$
- 5 R pkg Rmpfr

Outline

- 1 Preamble – Kudos to an Aussie Artist
- 2 Introduction
- 3 FAQ 7.31 — generalized: Loss of Accuracy
- 4 Accurately computing $\log 1\text{mexp}(x) := \log(1 - \exp(-x))$
- 5 R pkg Rmpfr

Kudos: useR! 2014, my talk on “Good Practices in R Programming”

- “Practice”: What I’ve learned over the years, with examples
- → Seven Guidelines (or *Rules*) for Good Practices in R Programming.

Rule 4: Do learn from the masters

An art is learned from the master artists:

Picasso, Van Gogh, Gauguin, Manet, Klimt . . .

John Chambers, **Bill Venables**, Bill Dunlap, Luke Tierney, Brian Ripley, R-core in general :-), Dirk Eddelbuettel, Hadley Wickham . . .

one of the *S* masters is an Aussie and has been here in Brisbane !! . . .

Read others’ source — Learn by examples

Kudos: useR! 2014, my talk on “Good Practices in R Programming”

- “Practice”: What I’ve learned over the years, with examples
- → Seven Guidelines (or *Rules*) for Good Practices in R Programming.

Rule 4: Do learn from the masters

An art is learned from the master artists:

Picasso, Van Gogh, Gauguin, Manet, Klimt ...

John Chambers, **Bill Venables**, Bill Dunlap, Luke Tierney, Brian Ripley, R-core in general :-), Dirk Eddelbuettel, Hadley Wickham ...

one of the *S* masters is an Aussie and has been here in Brisbane !! ...

Read others’ source — Learn by examples

Outline

- 1 Preamble – Kudos to an Aussie Artist
- 2 **Introduction**
- 3 FAQ 7.31 — generalized: Loss of Accuracy
- 4 Accurately computing $\log 1\text{mexp}(x) := \log(1 - \exp(-x))$
- 5 R pkg Rmpfr

R is accurate to 16 digits

All current R implementations use IEC 60559 floating-point arithmetic: A number is stored in 64 bits, 53 for the significand (“mantissa”), 11 for the exponent (both with a sign):

```
> str(.Machine[grep("^double", names(.Machine))], digits = 6)
```

List of 13

```
$ double.eps          : num 2.22045e-16
$ double.neg.eps      : num 1.11022e-16
$ double.xmin         : num 2.22507e-308
$ double.xmax         : num 1.79769e+308
$ double.base         : int 2
$ double.digits       : int 53
$ double.rounding     : int 5
$ double.guard        : int 0
$ double.ulp.digits   : int -52
$ double.neg.ulp.digits : int -53
$ double.exponent     : int 11
$ double.min.exp      : int -1022
$ double.max.exp      : int 1024
```

R is accurate ...

R is accurate to 16 digits, but not more:

```
> 1.000000000000000123456 - 1
```

```
[1] 2.220446e-16
```

```
> # 1 3 5 7 9 1 3 5^^^^^^ ==> correct difference would be 1.23456e-16
```

The first and most important accuracy loss in computer computations happens with **cancellation**:

Cancellation

Subtracting two almost equal numbers may lose almost all precision

Outline

- 1 Preamble – Kudos to an Aussie Artist
- 2 Introduction
- 3 **FAQ 7.31 — generalized: Loss of Accuracy**
- 4 Accurately computing $\log 1\text{mexp}(x) := \log(1 - \exp(-x))$
- 5 R pkg Rmpfr

FAQ 7.31 — Floating Point Numbers are Limited

R FAQ 7.31

Why doesn't R think these numbers are equal?

The only numbers that can be represented exactly in R's numeric type are integers and fractions whose denominator is a power of 2. Other numbers have to be rounded to (typically) 53 binary digits accuracy. As a result, two floating point numbers will not reliably be equal unless they have been computed by the same algorithm, and not always even then. For example

```
> a <- sqrt(2)
> a * a == 2 # mathematically, yes, ...
[1] FALSE
> a * a - 2
[1] 4.440892e-16
```

For more, ... David Goldberg (1991), “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, *ACM Computing Surveys*, **23/1**, 5–48...



FAQ 7.31 — Floating Point – 2 –

R FAQ 7.31

Why doesn't R think these numbers are equal?

A quote from “*The Elements of Programming Style*” by Kernighan and Plauger:

10.0 times 0.1 is hardly ever 1.0.

Actually, it is in R, (always / typically (?)), nowadays.

Then, it seems 0.1 exists in *several different* kinds:

```
> (1:10)/10 - (0:9)/10
```

```
[1] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
```

```
> (1:10)/10 - (0:9)/10 == 1/10
```

```
[1] TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
> unique((1:10)/10 - (0:9)/10)
```

```
[1] 0.1 0.1 0.1 0.1
```

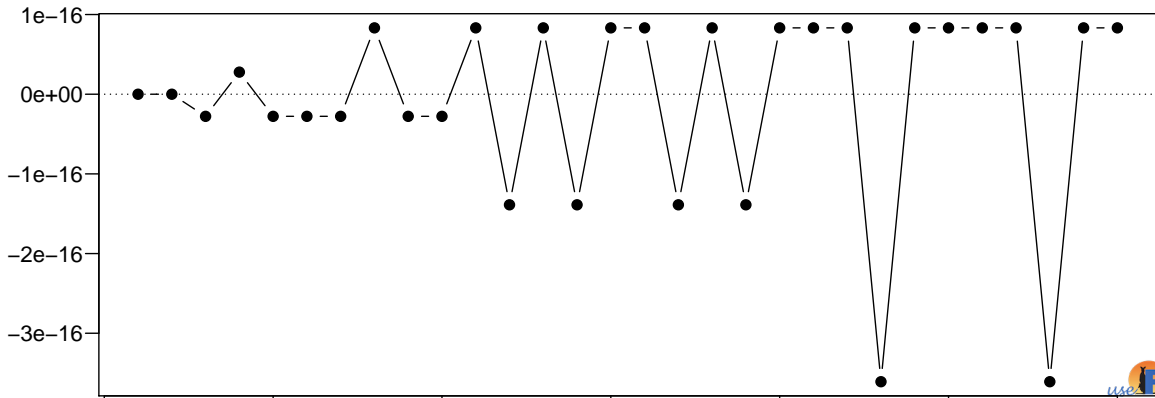
FAQ 7.31 — Different 0.1 's

Different kinds of 0.1 :

```
> diff((0:10)/10)
```

```
[1] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
```

```
> plot(diff((0:30)/10) - 1/10, ylab="", type = "b") ; abline(h=0, lty=3)
```



FAQ 7.31 ++ : The "log" in the dpq-functions

All "dpq" distribution functions in R, i.e. density cumulative probability and quantile functions, have a `log` or `log.p` argument (`FALSE` / `TRUE`).

Why ?

- 1 Compute Likelihoods: X_i independent \rightarrow Likelihood is *product*
 $L(\theta) = f(\theta, x_1)f(\theta, x_2) \cdots f(\theta, x_n)$ easily overflows or underflows.
- 2 Probabilistic Networks, MC(MC): $P = P_1 \cdot P_2 \cdots P_n$ quickly underflows to zero.

Solution: Work in "log space":

- 1 **Log** likelihood $\log L(\theta) = \sum_{i=1}^n \log f(\theta, x_i)$
- 2 $\log P = \sum_j \log P_j$ (for networks, MC, .. probability computations),

where $\log P_j$ are computed via R's `d<foo>(*, log=TRUE)` or `p<foo>(*, log.p=TRUE)`,
e.g., `sum(dnorm(x, *, log=TRUE))` instead of `prod(dnorm(x, *))`
or `sum(pnorm(*, log.p=TRUE))` instead of `prod(..)`, rather than taking logs.



FAQ 7.31 ... Why R has needed even more functions

- 1 `log1p()` (since R 1.0.0), mathematically, $\log1p(x) := \log(1 + x)$
- 2 `expm1()` (since R 1.5.0), mathematically, $\expm1(x) := \exp(x) - 1$

Computationally, they are defined to *not* suffer from cancellation when $|x|$ is small ($|x| \ll 1$), see below for $\log1p(x)$.

Why $\log(1+x)$ is not good enough, but $\log1p(x)$ is

$1+x$ cannot be numerically accurate when $|x| \ll 1$. In double precision (53 bits \approx 16 digits) accuracy, $1+x$ “sees” only 2–3 digits of x when $x = 4/9 \cdot 10^{-13}$,

```
> u <- 1 + (x <- 4e-13/9); cbind("u-1" = u-1, x) # u - 1 = x    mathematically, but:
```

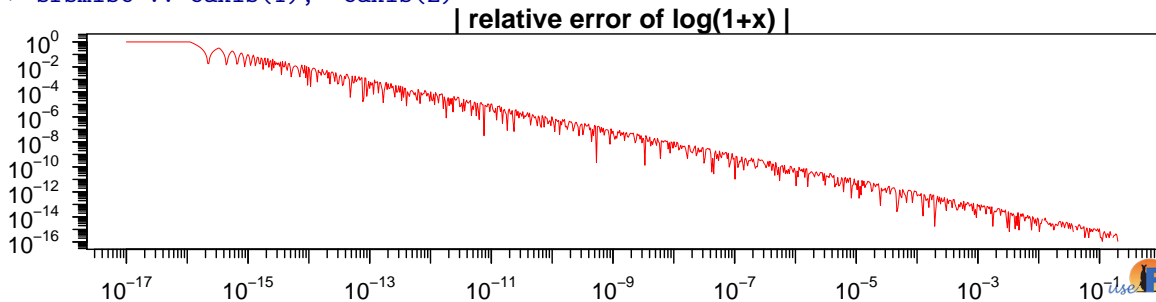
```
      u-1      x
```

```
[1,] 4.440892e-14 4.444444e-14
```

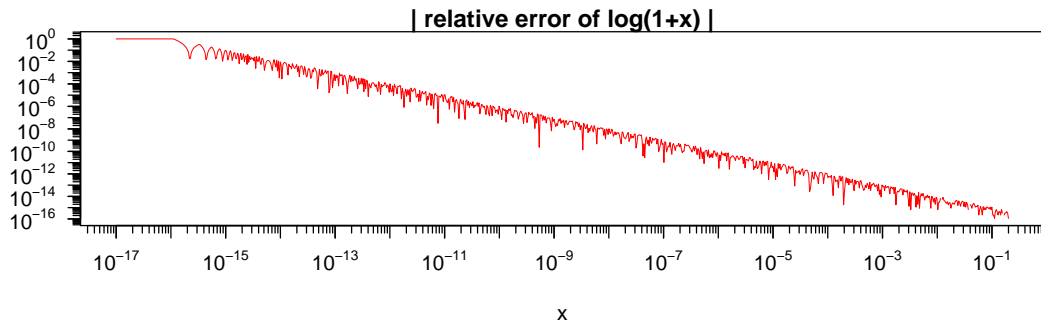
Consequence for $\log(1+x)$: Increasingly inaccurate for $|x| \rightarrow 0$.

```
> curve(abs(1 - log(1+x) / log1p(x)), 1e-17, .2, log = 'xy', main = "..", ..)
```

```
> sfsmisc :: eaxis(1); eaxis(2)
```



Why $\log1p(x)$ beats $\log(1+x)$



Solution: Expand $\log(1+x)$ around $x=0$. Well known

$$\log(1+x) = x - x^2/2 + x^3/3 \pm \dots = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{x^n}{n},$$

for $|x| < 1$. Fast version of this expansion: typically used in $\log1p()$'s implementation, and analogously for $\expm1()$

FAQ 7.31 ... Why R got even more functions –2–

- ❶ `cospi()`, `sinpi()`, `tanpi()` (from R 3.2.0), e.g.,

`cospi(x) := $\cos(\pi \cdot x)$` , accurately, e.g., for $x = \frac{1}{2}$:

```
> cos(pi/2) ## mathematically == 0
```

```
[1] 6.123234e-17
```

```
> cospi(1/2)
```

```
[1] 0
```

- ❷ `log1mexp(x) := $\log(1 - \exp(-x))$` , $x \geq 0$, accurately ...

my research → in R's Rmathlib C code, for years; named `R_Log1_Exp()` and used for accurate computations of the beta, gamma, exponential, Weibull, t, logistic, geometric and hypergeometric distributions.

→ quite important for (extreme cases of) many computations.

Outline

- 1 Preamble – Kudos to an Aussie Artist
- 2 Introduction
- 3 FAQ 7.31 — generalized: Loss of Accuracy
- 4 Accurately computing $\log 1\text{mexp}(x) := \log(1 - \exp(-x))$
- 5 R pkg Rmpfr

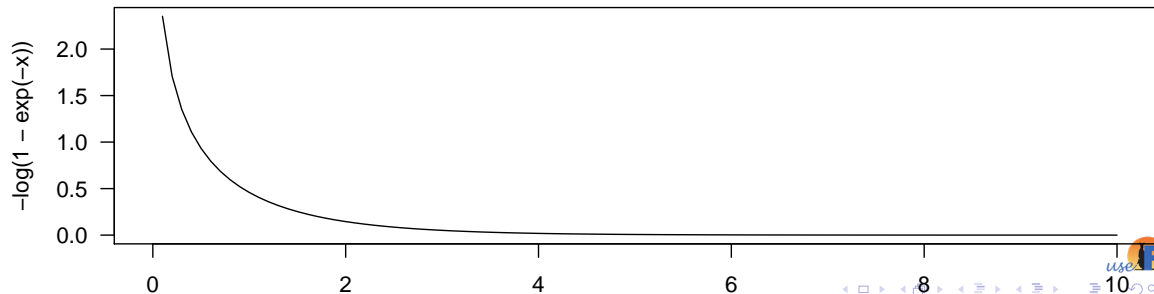
Simple (semi-artificial!) Example: `logit(exp(-L))`

Logistic regression: Computing “logit()”s, $\log \frac{p}{1-p}$ accurately for very small p , i.e., $p = \exp(-L)$, or

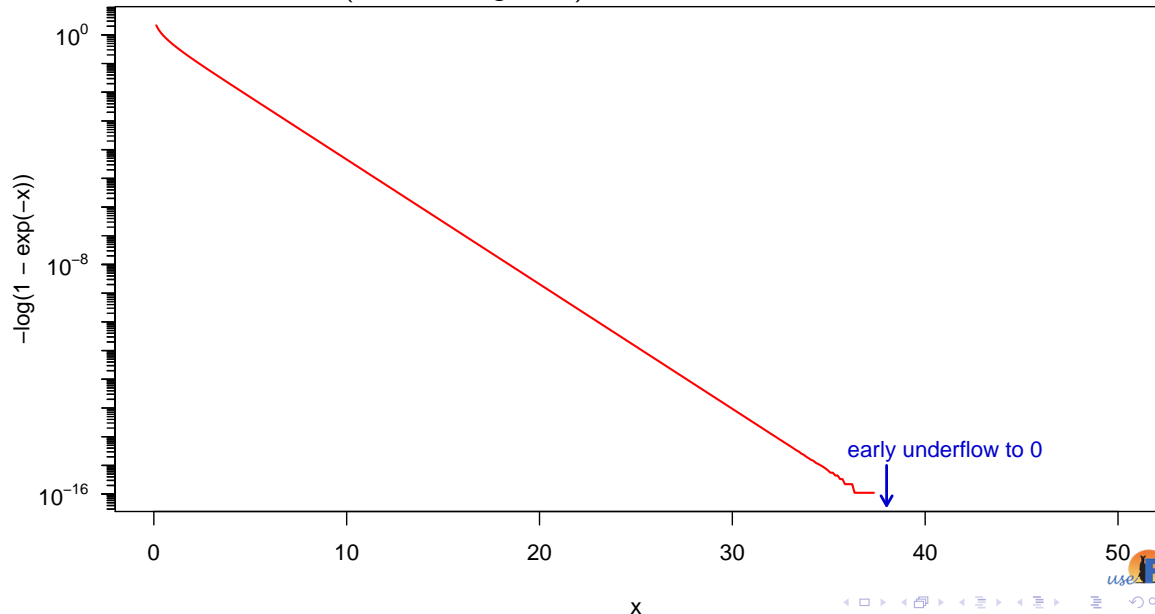
$$\log \frac{p}{1-p} = \log p - \log(1-p) = -L - \log(1 - \exp(-L)),$$

and hence $-\log(1 - \exp(-L))$ is needed, e.g., when p is really really close to 0, say $p = 10^{-1000}$, as then we can only compute `logit(p)`, if we specify $L := -\log(p) \leftrightarrow p = \exp(-L)$.

```
> curve(-log(1 - exp(-x)), 0, 10)
```



However, further out to 50 (and on a log scale), we observe



Explanation:

What did happen? Look at `log(.)`,

```
> x <- -40:-35
>      -log(1 - exp(x))
[1] 0.000000e+00 0.000000e+00 0.000000e+00 1.110223e-16 2.220446e-16
[6] 6.661338e-16

> log(-log(1 - exp(x)))# --> -Inf ..
[1]      -Inf      -Inf      -Inf -36.73680 -36.04365 -34.94504

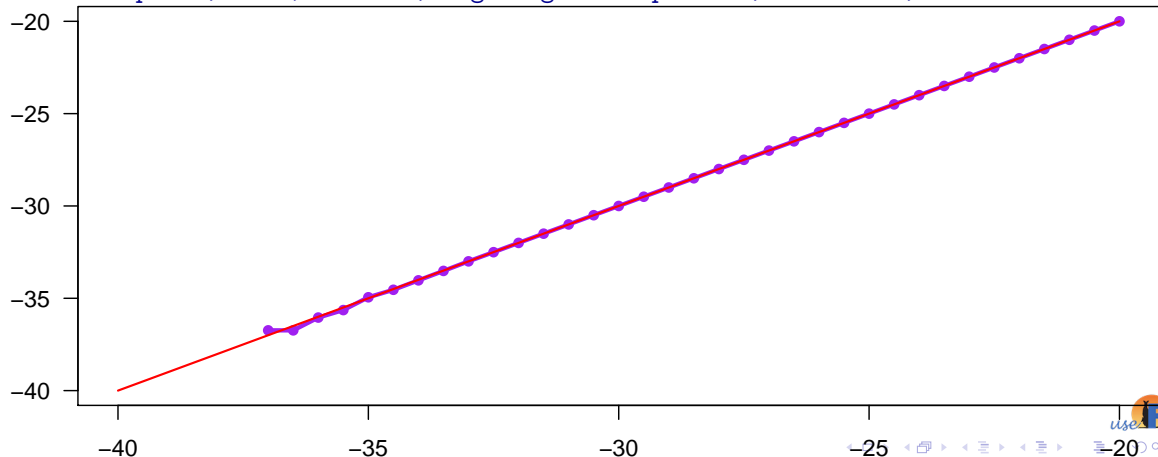
> ## ok, how about more accuracy
> x. <- mpfr(x, 120)
> log(-log(1 - exp(x.)))# aha... looks perfect now

6 'mpfr' numbers of precision 120 bits
[1] -39.999999999999999793290487753824173
[2] -38.999999999999999423372196756935807
[3] -37.9999999999999998430451715981029611
[4] -36.9999999999999995733184857961316543
[5] -35.99999999999999988402406183055208724
[6] -34.99999999999999968474421401530753269
```

Using `Rmpfr::mpfr(.)`

Visually, and with “high accuracy” `mpfr`-numbers:

```
> x <- seq(-40, -20, by = .5) ; plot(x,x, type="n", ylab="", ann=FALSE)
> lines(x, log(-log(1 - exp(x))), type = "o", col = "purple", lwd=3, cex = .6)
> x. <- mpfr(x, 120); lines(x, log(-log(1 - exp(x.))), col="red", lwd=1.5)
```



Optimal log1mexp() “implementation”

The “real” solution uses a piecewise implementation of $\mathbf{log1mexp}(x) = \log(1 - \exp(-x))$ for $x > 0$:

$$f(x) = \mathbf{log1mexp}(x) := \begin{cases} \log(-\expm1(-x)) & 0 < x \leq x_0 \quad (:= \log 2 \approx 0.693) \\ \log1p(-\exp(-x)) & x > x_0. \end{cases}$$

where it is *beautiful* that $x_0 = \log 2$ is optimal; see the Rmpfr package vignette *Accurately Computing $\log(1 - \exp(.))$ – Assessed by Rmpfr*.

Outline

- 1 Preamble – Kudos to an Aussie Artist
- 2 Introduction
- 3 FAQ 7.31 — generalized: Loss of Accuracy
- 4 Accurately computing $\log 1\text{mexp}(x) := \log(1 - \exp(-x))$
- 5 R pkg **Rmpfr**

Arithmetic (via S4 classes and methods) for arbitrary precision floating point numbers, including transcendental ("special") functions. To this end, the package interfaces to the 'LGPL' licensed 'MPFR' (Multiple Precision Floating-Point Reliable) Library which itself is based on the 'GMP' (GNU Multiple Precision) Library.

Version: 0.7-0
 Depends: [gmp](#) ($\geq 0.5-8$), R ($\geq 3.1.0$)
 Imports: stats, utils, methods
 Suggests: [MASS](#), [polynom](#), [sfsmisc](#) ($\geq 1.0-20$), [Matrix](#)
 Published: 2018-01-13
 Author: Martin Maechler
 Maintainer: Martin Maechler <maechler at stat.math.ethz.ch>
 License: [GPL-2](#) | [GPL-3](#) [expanded from: GPL (≥ 2)]
 URL: <http://rmpfr.r-forge.r-project.org/>
 NeedsCompilation: yes
 SystemRequirements: gmp ($\geq 4.2.3$), mpfr ($\geq 3.0.0$)
 Materials: [README](#) [NEWS](#) [ChangeLog](#)
 In views: [NumericalMathematics](#)
 CRAN checks: [Rmpfr results](#)

Downloads:

Reference manual: [Rmpfr.pdf](#)
 Vignettes: [useR-2011-abstract](#)
[Arbitrarily Accurate Computation with R Package Rmpfr](#)
[Accurately Computing \$\log\(1 - \exp\(.\)\)\$ - Assessed by Rmpfr](#)
 Package source: [Rmpfr_0.7-0.tar.gz](#)
 Windows binaries: r-devel: [Rmpfr_0.7-0.zip](#), r-release: [Rmpfr_0.7-0.zip](#), r-oldrel: [Rmpfr_0.7-0.zip](#)
 OS X binaries: r-release: [Rmpfr_0.7-0.tgz](#), r-oldrel: [Rmpfr_0.7-0.tgz](#)
 Old sources: [Rmpfr archive](#)

Reverse dependencies:

Reverse depends: [Bessel](#), [correlbinom](#), [CryptRndTest](#), [frmqa](#), [GeDS](#), [LDOD](#), [LIHNPSD](#), [MixedPoisson](#)
 Reverse imports: [corHMM](#), [CVXR](#), [downscale](#), [ecd](#), [ether](#), [GJRM](#), [HH](#), [ldatuning](#), [NetworkChange](#), [OUwie](#), [PMCMRplus](#), [SNscan](#), [WeMix](#)
 Reverse suggests: [alphastable](#), [blearn](#), [copula](#), [expm](#), [gmp](#), [IMIFA](#), [QuACN](#), [selectiveInference](#), [stabledist](#), [updog](#)

Example where we *need* Rmpfr: Alternating Binomial Sums

Alternating binomial sums ... are typically challenging, i.e., currently *impossible* to evaluate reliably as soon as n is larger than around 50 – 70.

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

$$sB(f, n) := \sum_{k=0}^n (-1)^{n-k} \binom{n}{k} \cdot f(k) = \Delta^n f, \quad (1)$$

where $\Delta^n f$ is the n -fold iterated forward difference $\Delta f(x) = f(x+1) - f(x)$ (here for $x = n0 = 0$).

computing alternating binomial sums in R

An obvious R implementation of $sB(f, n) = \sum_{k=0}^n (-1)^{n-k} \binom{n}{k} \cdot f(k)$,

```
> sumBinom <- function(n, f, n0=0, ...) {  
  k <- n0:n  
  sum( choose(n, k) * (-1)^(n-k) * f(k, ...))  
}  
  
> ## and the same for a whole *SET* of n values:  
> sumBin.all.R <- function(n, f, n0=0, ...)  
  sapply(n, sumBinom, f=f, n0=n0, ...)
```

Will see: gets numerical problems, for relatively small n even for well behaved functions $f(\cdot)$.

Comparison “double” vs “mpfr”:

For comparison, computing the alternating binomial sum,

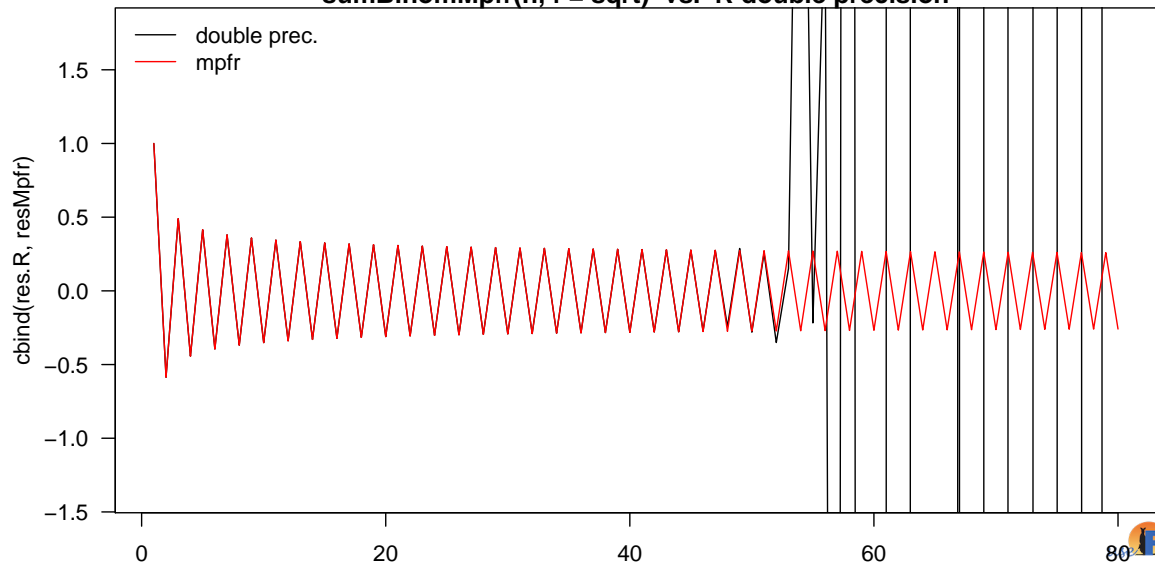
$$sB(f, n) := \sum_{k=0}^n (-1)^k \binom{n}{k} \cdot f(k),$$

now try the simple $f(x) = \sqrt{x}$, i.e., in R, `sqrt(x)`:

```
> nn <- 1:80
> system.time(res.R    <- sumBin.all.R(nn, f = sqrt)) ## 1/50 sec
  user  system elapsed
0.017   0.001   0.018
> system.time(resMpfr <- sumBin.all  (nn, f = sqrt)) ## 1/5  sec
  user  system elapsed
0.191   0.000   0.192
```

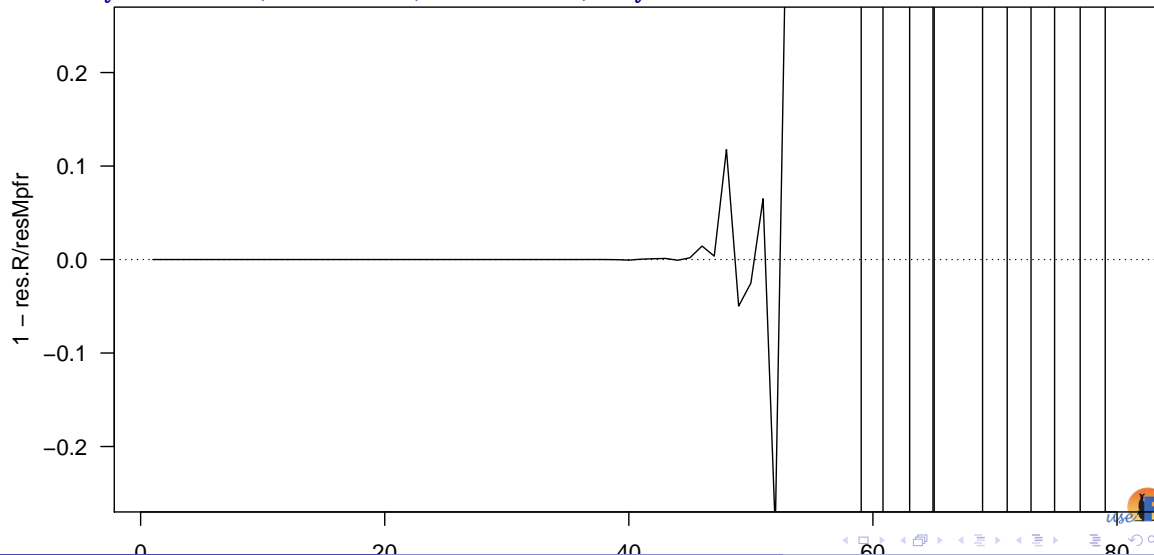
```
> matplot(nn, cbind(res.R, resMpfr), type = "l", lty=1, ylim = ..., main = ...) ; leg
```

sumBinomMpfr(n, f = sqrt) vs. R double precision

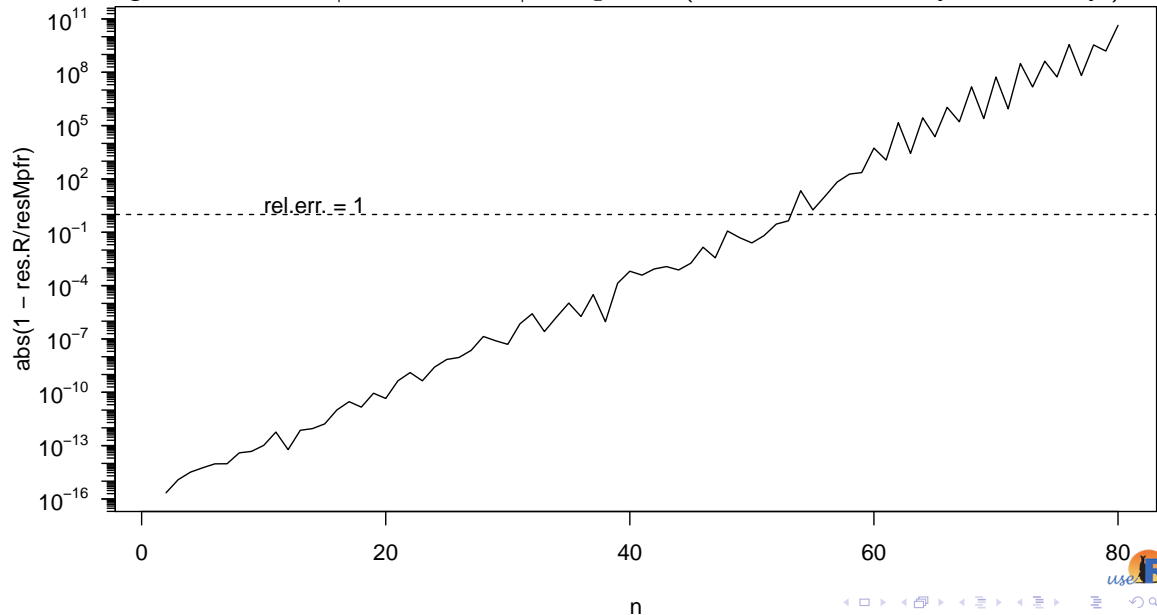


Relative error (alternating binomial sum):

```
> plot(nn, 1 - res.R/ resMpfr, type = "l", xlab=quote(n),  
      ylim = c(-1,1) * .25) ; abline(h=0, lty=3)
```

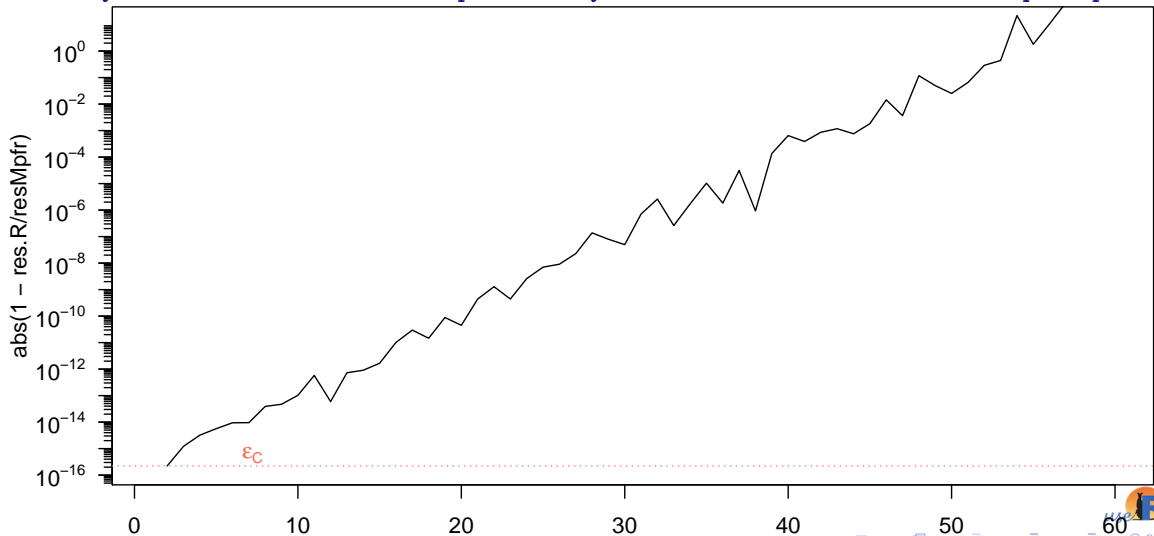


Alternating binomial sum: |Relative error| in log scale (Rel. error of 1 is very bad already!):



“Zoom in” of $|\text{Relative error}|$ (alternating binomial sum):

```
> plot(abs(1 - res.R/resMpfr) ~ nn, subset = nn <= 60, type="l", log="y",  
      ylim= c(2e-16, 10), xlab=quote(n), yaxt='n'); sfsmisc::eaxis(2); pl.Ceps()
```



Summary (aka 'TLDR')

- Double precision accuracy (almost 16 digits) is not always sufficient
- Use R's extras: e.g., `[dpq]norm(*, log=TRUE)`;
- `log1p()`, `exp1m()`;
- possibly `copula::log1mexp()` [which R uses in its C code for many `[dpq]<distr>()`].
- R package `Rmpfr` for arbitrarily precise computations in R .
- Many R functions — **when** `source()d` — will work with “mpfr”-numbers automatically

Handouts will become available possibly the useR! 2018 web page ...
and https://stat.ethz.ch/Teaching/maechler/R/useR_2018/

That's all Folks!

Summary (aka 'TLDR')

- Double precision accuracy (almost 16 digits) is not always sufficient
- Use R's extras: e.g., `[dpq]norm(*, log=TRUE)`;
- `log1p()`, `exp1m()`;
- possibly `copula::log1mexp()` [which R uses in its C code for many `[dpq]<distr>()`].
- R package `Rmpfr` for arbitrarily precise computations in R .
- Many R functions — **when** `source()d` — will work with “mpfr”-numbers automatically

Handouts will become available possibly the useR! 2018 web page ...
and https://stat.ethz.ch/Teaching/maechler/R/useR_2018/

That's all Folks!

Martin.Maechler@R-project.org 