Arbitrarily Accurate Computation with R: The Rmpfr Package

Martin Mächler
ETH Zurich

Abstract

The R package Rmpfr allows to use arbitrarily precise numbers instead of R’s double precision numbers in many R computations and functions. This is achieved by defining S4 classes of such numbers and vectors, matrices, and arrays thereof, where all arithmetic and mathematical functions work via the (GNU) MPFR C library, where MPFR is acronym for “Multiple Precision Floating-Point Reliably”. MPFR is Free Software, available under the LGPL license, and itself is built on the free GNU Multiple Precision arithmetic library (GMP).

Consequently, by using Rmpfr, you can often call your R function or numerical code with mpfr–numbers instead of simple numbers, and all results will automatically be much more accurate.

Applications by the package author include testing of Bessel or polylog functions and distribution computations, e.g. for (α-)stable distributions and Archimedean Copulas. In addition, the Rmpfr has been used on the R-help or R-devel mailing list for high-accuracy computations, e.g., in comparison with results from other software, and also in improving existing R functionality, e.g., fixing R bug PR#14491.

Keywords: MPFR, Arbitrary Precision, Multiple Precision Floating-Point, R.

1. Introduction

There are situations, notably in researching better numerical algorithms for non-trivial mathematical functions, say the $F$-distribution function, where it is interesting and very useful to be able to rerun computations in R in (potentially much) higher precision.

For example, if you are interested in Euler’s $e$, the base of natural logarithms, and given, e.g., by $e^x = \exp(x)$, you will look into

R> exp(1)
[1] 2.718282

which typically uses 7 digits for printing, as getOption("digits") is 7. To see R’s internal accuracy fully, you can use

R> print(exp(1), digits = 17)
[1] 2.7182818284590451

With Rmpfr you can now simply use “mpfr – numbers” and get more accurate results automatically, here using a vector of numbers as is customary in R:

R> print(c(exp(1), exp(2)), digits = 17)
[1] 2.7182818284590451 7.3890560989306504
Arbitrarily Accurate Computation with \textit{R}: The \texttt{Rmpfr} Package

\begin{verbatim}
R> require("Rmpfr") # after having installed the package ...
R> (one <- mpfr(1, 120))

1 'mpfr' number of precision 120 bits
[1] 1

R> exp(one)

1 'mpfr' number of precision 120 bits
[1] 2.7182818284590452353602874713526624979

In combinatorics, number theory or when computing series, you may occasionally want to work with \textit{exact} factorials or binomial coefficients, where e.g. you may need all factorials \(k!\), for \(k = 1, 2, \ldots, 24\) or a full row of Pascal’s triangle, i.e., want all \(\binom{n}{k}\) for \(n = 80\).

With \textit{R}’s double precision, and standard printing precision
\begin{verbatim}
R> ns <- 1:24 ; factorial(ns)

[1] 1.000000e+00 2.000000e+00 6.000000e+00 2.400000e+01 1.200000e+02
[6] 7.200000e+02 5.040000e+03 4.032000e+04 3.628800e+05 3.628800e+06
[16] 2.092279e+13 3.556874e+14 6.402374e+15 1.216451e+17 2.432902e+18
[21] 5.109094e+19 1.124001e+21 2.585202e+22 6.204484e+23
\end{verbatim}

the full precision of \(24!\) is clearly not printed. However, if you display it with more than its full internal precision,
\begin{verbatim}
R> noquote(sprintf("%-30.0f", factorial(24)))

[1] 620448401733239409999872
\end{verbatim}

it is obviously wrong in the last couple of digits as they are known to be 0. However, you can easily get full precision results with \texttt{Rmpfr}, by replacing “simple” numbers by \texttt{mpfr}-numbers:
\begin{verbatim}
R> ns <- mpfr(1:24, 120) ; factorial(ns)

24 'mpfr' numbers of precision 120 bits
[1]  1  2
[5] 120 720
[9] 362880 3628800
[13] 6227020800 87178291200
[17] 355687428096000 640237305728000
[21] 5109094217170944000 1124000727777607680000
[23] 25852016738884976640000 620448401733239439360000
\end{verbatim}

Or for the 80-th Pascal triangle row, \(\binom{n}{k}\) for \(n = 80\) and \(k = 1, \ldots, n\),
\begin{verbatim}
R> chooseMpfr.all(n = 80)

80 'mpfr' numbers of precision 77 bits
[1] 80 3160
[5] 24040016 300500200
[7] 3176716400 28987537150
\end{verbatim}

\end{verbatim}
S4 classes and methods: S4 allows “multiple dispatch” which means that the method that is called for a generic function may not just depend on the first argument of the function (as in S3 or in traditional class-based OOP), but on a “signature” of multiple arguments. For example, a + b is the same as ‘+’(a,b), i.e., calling a function with two arguments.

1.1. The engine behind: MPFR and GMP

The package Rmpfr interfaces R to the C (GNU) library

MPFR, acronym for “Multiple Precision Floating-Point Reliably”

MPFR is Free Software, available under the LGPL license, see http://mpfr.org/ and Fousse, Hanrot, Lefèvre, Péllissier, and Zimmermann (2007) and the standard reference to MPFR, Fousse, Hanrot, Lefèvre, Péllissier, and Zimmermann (2011). MPFR itself is built on and requires the GNU Multiple Precision arithmetic library (GMP), see http://gmplib.org/ and Granlund and the GMP development team (2011). It can be obtained from there, or from your operating system vendor.

On some platforms, it is very simple, to install MPFR and GMP, something necessary before Rmpfr can be used. E.g., in Linux distributions Debian, Ubuntu and other Debian derivatives, it is sufficient (for both libraries) to simply issue

```
sudo apt-get install libmpfr-dev
```

2. Arithmetic with mpfr-numbers

```
R> (0:7) / 7 # k/7, for k= 0..7 printed with R's default precision
[1] 0.0000000 0.1428571 0.2857143 0.4285714 0.5714286 0.7142857 0.8571429
R> options(digits= 16)
R> (0:7) / 7 # in full double precision accuracy
[1] 0.0000000000000000 0.1428571428571428 0.2857142857142857 0.4285714285714285 0.5714285714285714 0.7142857142857143 0.8571428571428571
R> options(digits= 7) # back to default
R> str(.Machine["double.digits","double.eps", "double.neg.eps"], digits=10)
```
List of 3  
$\text{double.digits} : \text{int } 53$  
$\text{double.eps} : \text{num } 2.220446049e-16$  
$\text{double.neg.eps} : \text{num } 1.110223025e-16$

R> 2^\text{-(52:53)}
[1] 2.220446e-16 1.110223e-16

In other words, the double precision numbers R uses have a 53-bit mantissa, and the two “computer epsilons” are $2^{-52}$ and $2^{-53}$, respectively.

Less technically, how many decimal digits can double precision numbers work with, $2^{-53} = 10^{-x} \iff x = 53 \log_{10}(2)$,

R> 53 * log10(2)
[1] 15.95459

i.e., almost 16 digits.

If we want to compute some arithmetic expression with higher precision, this can now easily be achieved, using the Rmpfr package, by defining “mpfr–numbers” and then work with these.

Starting with simple examples, a more precise version of $k/7$, $k = 0, \ldots, 7$ from above:

R> x <- mpfr(0:7, 80)/7 # using 80 bits precision
R> x

8 ‘mpfr’ numbers of precision 80 bits  
[1] 0 0.14285714285714285714285708
[3] 0.28571428571428571428571417 0.42857142857142857142857125
[5] 0.57142857142857142857142834 0.71428571428571428571428583
[7] 0.8571428571428571428571425 1

R> 7*x

8 ‘mpfr’ numbers of precision 80 bits  
[1] 0 1 2 3 4 5 6 7
R> 7*x - 0:7

8 ‘mpfr’ numbers of precision 80 bits  
[1] 0 0 0 0 0 0 0 0

which here is even “perfect” – but that’s “luck” only, and also the case here for “simple” double precision numbers, at least on our current platform.¹

2.1. Mathematical Constants, Pi ($\pi$), gamma, etc

Our Rmpfr package also provides the mathematical constants which MPFR provides, via Const(. , ⟨prec⟩), currently the 4 constants

R> formals(Const)$name
c("pi", "gamma", "catalan", "log2")

are available, where "gamma" is for Euler’s gamma, $\gamma := \lim_{n \to \infty} \sum_{k=1}^{n} \frac{1}{k} - \log(n) \approx 0.5777$, and "catalan" for Catalan’s constant (see http://en.wikipedia.org/wiki/Catalan%27s_constant).

¹64-bit Linux, Fedora 13 on a “AMD Phenom 925” processor
As nice example of using Mpfr arithmetic: On a wintry Sunday, Hans Borchers desired to have an exact π constant in Rmpfr, and realized that of course mpfr(pi, 256) could not be the solution, as pi is the double precision version of π and hence only about 53 bit accurate (and the mpfr() cannot do magic, recognizing “symbolic” π). As he overlooked the Const("pi", .) solution above, he implemented the following function that computes pi applying Gauss’ spectacular AGM-based (AGM := Arithmetic-Geometric Mean) approach [Borwein and Borwein (1987), Pi and the AGM]; I have added a verbose argument, explicit iteration counting and slightly adapted the style to my own:

```r
R> piMpfr <- function(prec=256, itermax = 100, verbose=TRUE) {
  m2 <- mpfr(2, prec) # 2 as mpfr number
  ## -> all derived numbers are mpfr (with precision 'prec')
  p <- m2 + sqrt(m2) # 2 + sqrt(2) = 3.414..
  y <- sqrt(sqrt(m2)) # 2^1/4
  x <- (y+1/y) / m2
  it <- 0L
  repeat {
    p.old <- p
    it <- it+1L
    p <- p * (1+x) / (1+y)
    if(verbose) cat(sprintf("it=%2d, pi^ = %s, |.-.|/|.|=%e\n",
      it, formatMpfr(p, min(50, prec/log2(10))), 1-p.old/p))
    if (abs(p-p.old) <= m2^(-prec))
      break
    if(it > itermax) {
      warning("not converged in", it, "iterations") ; break
    }
    ## else
    s <- sqrt(x)
    y <- (y*s + 1/s) / (1+y)
    x <- (s+1/s)/2
  }
  p
}
R> piMpfr()# indeed converges *quadratically* fast
Arbitrarily Accurate Computation with R: The `Rmpfr` Package

```r
it = 1, pi = 3.142606753941622600790198236183018919713562462772, |.-.|/|.-.| = -8.642723e-02
it = 2, pi = 3.14159266096604423049775223512033969067928425686845, |.-.|/|.-.| = -3.227958e-04
it = 3, pi = 3.141592653897932386457739917571417940347896238675, |.-.|/|.-.| = -2.347934e-09
it = 4, pi = 3.14159265389793238462643383279502884197224204666, |.-.|/|.-.| = -5.829228e-20
it = 5, pi = 3.141592653897932384626433832795028841971693993751, |.-.|/|.-.| = -1.741826e-41
it = 6, pi = 3.141592653897932384626433832795028841971693993751, |.-.|/|.-.| = 0.000000e+00
```

1 'mpfr' number of precision 256 bits

R> ## with relative error
R> relErr <- 1 - piMpfr(256, verbose=FALSE) / Const("pi",260)
R> ## in bits :
R> asNumeric(-log2(abs(relErr)))

[1] 255.2451

2.2. seqMpfr() for sequences:

In R, arithmetic sequences are constructed by `seq()`, the “sequence” function, which is not generic, and with its many ways and possible arguments is convenient, but straightforward to automatically generalize for mpfr numbers. Instead, we provide the `seqMpfr` function...

2.3. Rounding, `roundMpfr()`, `asNumeric()` etc:

In R, the `round()` and `signif()` functions belong to the `Math2` group, and we provide "mpfr"-class methods for them:

```r
R> getGroupMembers("Math2")
[1] "round" "signif"
R> showMethods("Math2", classes=c("mpfr", "mpfrArray"))
```

For consistency reasons, however the resulting numbers keep the same number of precision bits, `precBits`

```r
R> i7 <- 1/mpfr(700, 100)
R> c(i7, round(i7, digits = 6), signif(i7, digits = 6))
```

3 'mpfr' numbers of precision 100 bits

```r
table
```

If you really want to “truncate” the precision to less digits or bits, you call `roundMpfr()`,

```r
R> roundMpfr(i7, precBits = 30)
1 'mpfr' number of precision 30 bits

[1] 0.001428571428571428571428571428571428571
R> roundMpfr(i7, precBits = 15)
1 'mpfr' number of precision 15 bits

[1] 0.00142854
```

Note that 15 bits correspond to approximately \(15 \cdot 0.3\), i.e., 4.5 digits, because \(1 / \log_2(10) \approx 0.30103\).
asNumeric(): Often used, e.g., to return to fast (R-internal) arithmetic, also as alternative to roundMpfr() is to “round to double precision” producing standard Rnumbers from “mpfr” numbers. We provide the function asNumeric(), a generic function with methods also for "mpfrArray" see below and the big integers and big rationals from package gmp.

R> showMethods(asNumeric)

Function: asNumeric (package gmp)
x="ANY"
x="bigq"
x="bigz"
x="mpfr"
x="mpfrArray"
x="numeric"

(herited from: x="ANY")

see, e.g., its use above.

Formatting: For explicit printing or plotting purposes, we provide an "mpfr" method for R’s format() function, also as explicit utility function formatMpfr(x, digits) which provides results to digits significant digits,

R> cbind( sapply(1:7, function(d) format(i7, digits=d)) )

[1,] "0.001" "0.0014" "0.00143" "0.001429" "0.0014286" "0.00142857" "0.001428571"

There, digits = NULL is the default where the help has (“always”) promised The default, NULL, uses enough digits to represent the full precision, often one or two digits more than you would expect. However, for large numbers, say 10^20000, e.g., new("mpfr1", prec = 80, exp = c(66439, 0), sign = 1, d = c(0, -1008336896, 1315775171, -1124830946)), all of formatMpfr(x), format(x), and print(x) (including “auto-printing” of x), have shown all digits before the decimal point and not at all taken into account the 80-bit precision of x (which corresponds to only 80 / log2(10) ≈ 24 decimal digits). This has finally changed in the (typically default) case formatMpfr(*, maybe.full = FALSE):

R> x <- mpfr(2, 80) ^ ((1:4)*10000)
R> cbind(x) # -> show() -> print.mpfr() -> formatMpfr(*, digits = NULL, maybe.full = FALSE)

'mpfrMatrix' of dim(.) = (4, 1) of precision 80 bits

x

[,1] 1.9950631168807583848837422e+3010
[2,] 3.9802768403379665923543072e+6020
[3,] 7.9409035191329603241325178e+9030
[4,] 1.5842603725730738680597362e+12041

R> nchar(formatMpfr(x))

[1] 33 33 33 34

R> nchar(formatMpfr(x, maybe.full = TRUE))
3. “All” mathematical functions, arbitrarily precise

All the S4 “Math” group functions are defined, using multiple precision (MPFR) arithmetic, i.e.,

```r
R> getGroupMembers("Math")

[1] "abs" "sign" "sqrt" "ceiling" "floor" "trunc"
[7] "cummax" "cummin" "cumprod" "cumsum" "exp" "expm1"
[13] "log" "log10" "log2" "log1p" "cos" "cosh"
[19] "sin" "sinh" "tan" "tanh" "acos" "acosh"
[25] "asin" "asinh" "atan" "atanh" "acospi" "asincospi"
[31] "tanpi" "gamma" "lgamma" "digamma" "trigamma"
```

where currently, `trigamma` is not provided by the MPFR library, and hence not implemented yet.

`factorial()` has a "mpfr" method; and in addition, `factorialMpfr()` computes $n!$ efficiently in arbitrary precision, using the MPFR-internal implementation. This is mathematically (but not numerically) the same as $\Gamma(n+1) = \text{gamma}(n+1)$.

Similarly to `factorialMpfr()`, but more generally useful, the functions `chooseMpfr(a, n)` and `pochMpfr(a, n)` compute (generalized!) binomial coefficients \( \binom{a}{n} \) and “the” Pochhammer symbol or “rising factorial"

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

Note that with this definition,

\[ \binom{a}{n} \equiv \frac{a^{(n)}}{n!}. \]

4. Arbitrarily precise matrices and arrays

The classes "mpfrMatrix" and "mpfrArray" correspond to the classical numerical R "matrix" and "array" objects, which basically are arrays or vectors of numbers with a dimension `dim`, possibly named by `dimnames`. As there, they can be constructed by `dim(.) <- ..` setting, e.g.,

```r
R> head(x <- mpfr(0:7, 64)/7) ; mx <- x

6 'mpfr' numbers of precision 64 bits
[1] 0.142857142857142857141 0.285714285714285714282
[4] 0.428571428571428571436 0.571428571428571428564 0.714285714285714285691
R> dim(mx) <- c(4,2)
```

or by the `mpfrArray()` constructor,

```r
R> dim(aa <- mpfrArray(1:24, precBits = 80, dim = 2:4))
```
[1] 2 3 4
R> aa
'mpfrArray' of dim(.) = (2, 3, 4) of precision 80 bits
 , , 1
   [,1]    [,2]
[1,] 1.00000000000000000000000000000000 3.00000000000000000000000000000000
[2,] 2.00000000000000000000000000000000 4.00000000000000000000000000000000
 [,3]
[1,] 5.00000000000000000000000000000000
[2,] 6.00000000000000000000000000000000
 , , 2
 .......
 .......
 [,3]
[1,] 23.00000000000000000000000000000000
[2,] 24.00000000000000000000000000000000
and we can index and multiply such matrices, e.g.,
R> mx[1:3,] + c(1,10,100)
'mpfrMatrix' of dim(.) = (3, 2) of precision 64 bits
 [,1]     [,2]
[1,] 1.00000000000000000000 1.57142857142857142851
[2,] 10.1428571428571428570 10.7142857142857142860
[3,] 100.285714285714285712 100.857142857142857144
R> crossprod(mx)
'mpfrMatrix' of dim(.) = (2, 2) of precision 64 bits
 [,1]     [,2]
[1,] 0.285714285714285714282 0.775510204081632653086
[2,] 0.775510204081632653086 2.57142857142857142851
and also apply functions,
R> apply(7 * mx, 2, sum)
 2 'mpfr' numbers of precision 64 bits
[1] 6 22

5. Special mathematical functions

\texttt{zeta(x)} computes Riemann’s Zeta function $\zeta(x)$ important in analytical number theory and related fields. The traditional definition is

$$\zeta(x) = \sum_{n=1}^{\infty} \frac{1}{n^x}.$$ 

\texttt{Ei(x)} computes the exponential integral,

$$\int_{-\infty}^{x} \frac{e^t}{t} dt.$$
Arbitrarily Accurate Computation with \textit{R}: The \texttt{Rmpfr} Package

\begin{verbatim}
R> curve(Ei, 0, 5, n=2001); abline(h=0,v=0, lty=3)
\end{verbatim}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{plot.png}
\end{figure}

\texttt{Ei}, part of the MPFR C library since version 2.4.0, computes the dilogarithm,

\[
\text{Li}_2(x) = \text{Li}_2(x) := \int_0^x \frac{-\log(1-t)}{t} \, dt,
\]

which is the most prominent “polylogarithm” function, where the general polylogarithm is (initially) defined as

\[
\text{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}, \quad \forall s \in \mathbb{C} \quad \forall |z| < 1, \ z \in \mathbb{C},
\]

see \url{http://en.wikipedia.org/wiki/Polylogarithm#Dilogarithm}.

Note that the integral definition is valid for all \(x \in \mathbb{C}\), and also, \(\text{Li}_2(1) = \zeta(2) = \pi^2/6\).

\begin{verbatim}
R> if(mpfrVersion() >= "2.4.0") ## Li2() is not available in older MPFR versions
    all.equal(Li2(1), Const("pi", 128)^2/6, tol = 1e-30)
\end{verbatim}

\[1\] TRUE

where we also see that \texttt{Rmpfr} provides \texttt{all.equal()} methods for mpfr-numbers which naturally allow very small tolerances \texttt{tol}.

\begin{verbatim}
R> if(mpfrVersion() >= "2.4.0")
    curve(Li2, -2, 13, n=2000); abline(h=0,v=0, lty=3)
\end{verbatim}
erf(x) is the “error function” and \texttt{erfc(x)} its complement, \( \texttt{erfc(x) := 1 - erf(x)} \), defined as

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt,
\]

and consequently, both functions simply are reparametrizations of the cumulative normal, \( \Phi(x) = \int_{-\infty}^x \phi(t) \, dt = \text{pnorm}(x) \) where \( \phi \) is the normal density function \( \phi(t) := \frac{1}{\sqrt{2\pi}} e^{-t^2} = \text{dnorm}(x) \).

Namely, \( \text{erf}(x) = 2 \times \text{pnorm}(\sqrt{2} \times x) \) and \( \text{erfc}(x) = 1 - \text{erf}(x) = 2 \times \text{pnorm}(\sqrt{2} \times x, \text{lower}=\text{FALSE}) \).

\begin{verbatim}
R> curve(erf, -3,3, col = "red", ylim = c(-1,2))
R> curve(erfc, add = TRUE, col = "blue")
R> abline(h=0, v=0, lty=3); abline(v=\text{-1,1}), lty=3, lwd=0.8, col="gray")
R> legend(-3,1, c("erf(x)", "erfc(x)"), col = c("red","blue"), lty=1)
\end{verbatim}

\[2\text{named exactly because of its relation to the normal / Gaussian distribution}\]
5.1. Applications

The CRAN package `Bessel` provides asymptotic formulas for Bessel functions also of fractional order which do work for `mpfr`-vector arguments as well.

6. Integration highly precisely

Sometimes, important functions are defined as integrals of other known functions, e.g., the dilogarithm \( \text{Li}_2() \) above. Consequently, we found it desirable to allow numerical integration, using `mpfr`-numbers, and hence—conventionally—arbitrarily precisely.

R’s `integrate()` uses a relatively smart adaptive integration scheme, but based on C code which is not very simply translatable to pure R, to be used with `mpfr` numbers. For this reason, our `integrateR()` function uses classical Romberg integration (Bauer 1961).

We demonstrate its use, first by looking at a situation where R’s `integrate()` can get problems:

```r
R> integrateR(dnorm, 0, 2000)
0.5 with absolute error < 4.3687e-05
```

```r
R> integrateR(dnorm, 0, 2000, rel.tol=1e-15)
0.5 with absolute error < 0
```

```r
R> integrateR(dnorm, 0, 2000, rel.tol=1e-15, verbose=TRUE)
n= 1, 2^n= 2 | I = 132.98076013381089, abs.err = 265.9615
n= 2, 2^n= 4 | I = 62.05688062445074, abs.err = 70.92307
n= 3, 2^n= 8 | I = 30.536322697393608, abs.err = 31.52137
n= 4, 2^n= 16 | I = 15.208286206152895, abs.err = 15.32804
n= 5, 2^n= 32 | I = 7.5967099231125408, abs.err = 7.611576
n= 6, 2^n= 64 | I = 3.7974274023470991, abs.err = 3.799283
n= 7, 2^n= 128 | I = 1.8985978058124329, abs.err = 1.899134
n= 8, 2^n= 256 | I = 0.94928441753372339, abs.err = 0.9493134
n= 9, 2^n= 512 | I = 0.4757402595965515, abs.err = 0.4756442
n=10, 2^n= 1024 | I = 0.20552346957493818, abs.err = 0.07021679
n=11, 2^n= 2048 | I = 0.1002349
n=12, 2^n= 4096 | I = 0.005717068
n=13, 2^n= 8192 | I = 0.00017068
n=14, 2^n= 16384 | I = 0.000039584652330221, abs.err = 0.0000717068
n=15, 2^n= 32768 | I = 0.49999999999999999999, abs.err = 2.349514e-06
n=16, 2^n= 65536 | I = 0.49999999999999999999, abs.err = 1.083003e-08
n=17, 2^n= 131072 | I = 0.49999999999999999999, abs.err = 1.097966e-11
n=18, 2^n= 262144 | I = 0.50000000000000000000, abs.err = 2.775558e-15
n=19, 2^n= 524288 | I = 0.50000000000000000000, abs.err = 0
0.5 with absolute error < 0
```

Now, for situations where numerical integration would not be necessary, as the solution is known analytically, but hence are useful for exploration of high accuracy numerical integration:

First, the exponential function \( e^x \) with its well-known \( \int e^t \, dt = e^x \), both with standard (double precision) floats,

```r
R> (Ie.d <- integrateR(exp, 0, 1, rel.tol=1e-15, verbose=TRUE))
```

```r
0.5 with absolute error < 4.3687e-05
```

```r
R> integrateR(exp, 0, 1, rel.tol=1e-15, verbose=TRUE)
n= 1, 2^n= 2 | I = 1.8985978058124329, abs.err = 1.899134
n= 2, 2^n= 4 | I = 0.94928441753372339, abs.err = 0.9493134
n= 3, 2^n= 8 | I = 0.4757402595965515, abs.err = 0.4756442
n= 4, 2^n= 16 | I = 0.20552346957493818, abs.err = 0.07021679
n= 5, 2^n= 32 | I = 0.1002349
n= 6, 2^n= 64 | I = 0.005717068
n= 7, 2^n= 128 | I = 0.00017068
n= 8, 2^n= 256 | I = 0.000039584652330221, abs.err = 0.0000717068
n= 9, 2^n= 512 | I = 0.49999999999999999999, abs.err = 2.349514e-06
n=10, 2^n= 1024 | I = 0.49999999999999999999, abs.err = 1.083003e-08
n=11, 2^n= 2048 | I = 0.49999999999999999999, abs.err = 1.097966e-11
n=12, 2^n= 4096 | I = 0.50000000000000000000, abs.err = 2.775558e-15
n=13, 2^n= 8192 | I = 0.50000000000000000000, abs.err = 0
0.5 with absolute error < 0
```

```r
R> integrateR(exp, 0, 1, rel.tol=1e-15, verbose=TRUE)
n= 1, 2^n= 2 | I = 1.8985978058124329, abs.err = 1.899134
n= 2, 2^n= 4 | I = 0.94928441753372339, abs.err = 0.9493134
n= 3, 2^n= 8 | I = 0.4757402595965515, abs.err = 0.4756442
n= 4, 2^n= 16 | I = 0.20552346957493818, abs.err = 0.07021679
n= 5, 2^n= 32 | I = 0.1002349
n= 6, 2^n= 64 | I = 0.005717068
n= 7, 2^n= 128 | I = 0.00017068
n= 8, 2^n= 256 | I = 0.000039584652330221, abs.err = 0.0000717068
n= 9, 2^n= 512 | I = 0.49999999999999999999, abs.err = 2.349514e-06
n=10, 2^n= 1024 | I = 0.49999999999999999999, abs.err = 1.083003e-08
n=11, 2^n= 2048 | I = 0.49999999999999999999, abs.err = 1.097966e-11
n=12, 2^n= 4096 | I = 0.50000000000000000000, abs.err = 2.775558e-15
n=13, 2^n= 8192 | I = 0.50000000000000000000, abs.err = 0
0.5 with absolute error < 0
```
and then the same, using 200-bit accurate mpfr-numbers:

```r
R> (Ie.m <- integrateR(exp, mpfr(0,200), 1, rel.tol=1e-25, verbose=TRUE))
n= 1, 2^n= 2 | I = 1.71886115187659297045914844, abs.err = 0.1402798
n= 2, 2^n= 4 | I = 1.71828268792475745881674571, abs.err = 0.0005784640
n= 3, 2^n= 8 | I = 1.71828182879453042315257873, abs.err = 8.591302e-76
n= 4, 2^n= 16 | I = 1.718281828459045236135029253, abs.err = 3.354521e-10
n= 5, 2^n= 32 | I = 1.71828182845904523536028747, abs.err = 5.056528e-24
n= 6, 2^n= 64 | I = 1.71828182845904523536028747, abs.err = 7.817216e-30
```

1.7183 with absolute error < 7.8172e-30

```r
R> (I.true <- exp(mpfr(1, 200)) - 1)
1 'mpfr' number of precision 200 bits
[1] 1.7182818284590452353602874713526624977572470936996595749669679
```

```r
R> ## with absolute errors
R> as.numeric(c(I.true - Ie.d$value,
               I.true - Ie.m$value))
```

Now, for polynomials, where Romberg integration of the appropriate order is exact, mathematically,

```r
R> if(require("polynom")) {
  x <- polynomial(0:1)
p <- (x-2)^4 - 3*(x-3)^2
Fp <- as.function(p)
print(pI <- integral(p)) # formally
print(Itrue <- predict(pI, 5) - predict(pI, 0)) ## == 20
} else {
  Fp <- function(x) (x-2)^4 - 3*(x-3)^2
Itrue <- 20
}
```

-11*x - 7*x^2 + 7*x^3 - 2*x^4 + 0.2*x^5

[1] 20

```r
R> (Id <- integrateR(Fp, 0, 5))
20 with absolute error < 7.1054e-15
```

```r
R> (Im <- integrateR(Fp, 0, mpfr(5, 256),
                     rel.tol = 1e-70, verbose=TRUE))
n= 1, 2^n= 2 | I = 46.041666666666666666666666666666666666666666666667, abs.err = 9.377777
n= 2, 2^n= 4 | I = 20.000000000000000000000000000000000000000000000000, abs.err = 2.7658e-19
n= 3, 2^n= 8 | I = 20.000000000000000000000000000000000000000000000000, abs.err = 2.7636e-38
```

20.000 with absolute error < 2.7636e-76
7. Conclusion

The R package `Rmpfr`, available from CRAN since August 2009, provides the possibility to run many computations in R with (arbitrarily) high accuracy, though typically with substantial speed penalty.

This is particularly important and useful for checking and exploring the numerical stability and appropriateness of mathematical formulae that are translated to a computer language like R, often without very careful consideration of the limits of computer arithmetic.

References


Granlund T, the GMP development team (2011). *GNU MP - The GNU Multiple Precision Arithmetic Library*. URL http://gmplib.org/.

FIXME: Index of all functions mentioned ...

Affiliation:

Martin Mächler  
Seminar für Statistik, HG G 16  
ETH Zurich  
8092 Zurich, Switzerland  
E-mail: maechler@stat.math.ethz.ch  
URL: http://stat.ethz.ch/people/maechler