Arbitrarily Accurate Computation with R: The Rmpfr Package

Martin Mächler
ETH Zurich

Abstract

The R package **Rmpfr** allows to use arbitrary high precision 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 "***M****ultiple* ***P****recision* ***F****loating-Point* ***R****eliably*". 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 stable distributions.

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, Abitrary 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> 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 occasionaly want to work with *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 R's double precision, and standard printing precision

```
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
[11] 3.991680e+07 4.790016e+08 6.227021e+09 8.717829e+10 1.307674e+12
[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
```

the full precision of 24! is clearly not printed. However, if you display it with more than its full internal precision,

```
R> noquote(sprintf("%-30.0f", factorial(24)))
```

```
[1] 620448401733239409999872
```

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 **Rmpfr**, by replacing "simple" numbers by mpfr-numbers:

```
R> ns <- mpfr(1:24, 120) ; factorial(ns)
```

```
24 'mpfr' numbers of precision  120   bits
 [1] 1                         2
 [3] 6                         24
 [5] 120                       720
 [7] 5040                      40320
 [9] 362880                    3628800
[11] 39916800                  479001600
[13] 6227020800                87178291200
[15] 1307674368000             20922789888000
[17] 355687428096000           6402373705728000
[19] 121645100408832000        2432902008176640000
[21] 51090942171709440000      1124000727777607680000
[23] 25852016738884976640000   620448401733239439360000
```

Or for the 80-th Pascal triangle row, $\binom{n}{k}$ for $n = 80$ and $k = 0, \ldots, n$,

```
R> chooseMpfr.all(n = 80)
```

```
80 'mpfr' numbers of precision  77 .. 128  bits
 [1] 80                        3160
 [3] 82160                     1581580
 [5] 24040016                  300500200
 [7] 3176716400                28987537150
 [9] 231900297200              1646492110120
[11] 10477677064400            60246643120300
[13] 315136287090800           1508152231077400
[15] 6635869816740560          26958221130508525
[17] 101489773667796800        355214207837288800
[19] 1159120046626942400       3535316142212174320
```

```
[21] 10100903263463355200      27088786024742634400
[23] 68310851714568382400      162238272822099908200
[25] 363413731121503794368     768759815833950334240
[27] 1537519631667900668480    2910305017085669122480
[29] 5218477961670854978240    8871412534840453463008
[31] 14308729894903957198400   21910242651571684460050
[33] 31869443856831541032800   44054819449149483192400
[35] 57900619847453606481440   72375774809317008101800
[37] 86068488962431036661600   97393290141698278327600
[39] 104885081691059684352800  107507208733336176461620
[41] 104885081691059684352800  97393290141698278327600
[43] 86068488962431036661600   72375774809317008101800
[45] 57900619847453606481440   44054819449149483192400
[47] 31869443856831541032800   21910242651571684460050
[49] 14308729894903957198400   8871412534840453463008
[51] 5218477961670854978240    2910305017085669122480
[53] 1537519631667900668480    768759815833950334240
[55] 363413731121503794368     162238272822099908200
[57] 68310851714568382400      27088786024742634400
[59] 10100903263463355200      3535316142212174320
[61] 1159120046626942400       355214207837288800
[63] 101489773667796800        26958221130508525
[65] 6635869816740560          1508152231077400
[67] 315136287090800           60246643120300
[69] 10477677064400            1646492110120
[71] 231900297200              28987537150
[73] 3176716400                300500200
[75] 24040016                  1581580
[77] 82160                     3160
[79] 80                        1
```

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 "***M**ultiple **P**recision **F**loating-Point **R**eliably*".

MPFR is Free Software, available under the LGPL license, see `http://mpfr.org/`, and MPFR itself is built on and requires the GNU Multiple Precision arithmetic library (GMP), see `http://gmplib.org/`. 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
```

The standard reference to MPFR is Fousse, Hanrot, Lefèvre, Pélissier, and Zimmermann (2011).

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
[8] 1.0000000
```

```
R> options(digits= 16)
R> (0:7) / 7  #  in full  double precision accuracy
```

```
[1] 0.0000000000000000 0.1428571428571428 0.2857142857142857
[4] 0.4285714285714285 0.5714285714285714 0.7142857142857143
[7] 0.8571428571428571 1.0000000000000000
```

```
R> options(digits=  7) # back to default
R> str(.Machine[c("double.digits","double.eps", "double.neg.eps")], digits=10)
```

```
List of 3
 $ double.digits : int 53
 $ double.eps    : num 2.220446049e-16
 $ double.neg.eps: num 1.110223025e-16
```

```
R> 2^-(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] 0.85714285714285714285714250  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]

[1] 64-bit Linux, Fedora 13 on a "AMD Phenom 925" processor

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).

```
R> Const("pi")
```

```
1 'mpfr' number of precision  120   bits
[1] 3.1415926535897932384626433832795028847
```

```
R> Const("log2")
```

```
1 'mpfr' number of precision  120   bits
[1] 0.69314718055994530941723212145817656831
```

where you may note a default precision of 120 digits, a bit more than quadruple precision, but also that 1000 digits of π are available instantaneously,

```
R> system.time(Pi <- Const("pi", 1000 *log2(10)))
```

```
   user  system elapsed
  0.001   0.000   0.002
```

```
R> Pi
```

```
1 'mpfr' number of precision  3321   bits
[1] 3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253
```

TODO — an example of a user written function, computing something relevant ...

...

`seqMpfr()`

...

...

3. "All" mathematical functions, arbitrarily precise

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

```
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"    "gamma"    "lgamma"
[31] "digamma"  "trigamma"
```

where currently , `digamma`, and `trigamma` are not provided by the MPFR library, and hence the methods 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 $\Gamma(n+1)$ =`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"

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

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

6 'mpfr' numbers of precision  64   bits
[1] 0                       0.142857142857142857141 0.285714285714285714282
[4] 0.428571428571428571436 0.571428571428571428564 0.714285714285714285691

R> dim(mx) <- c(4,2)
```

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

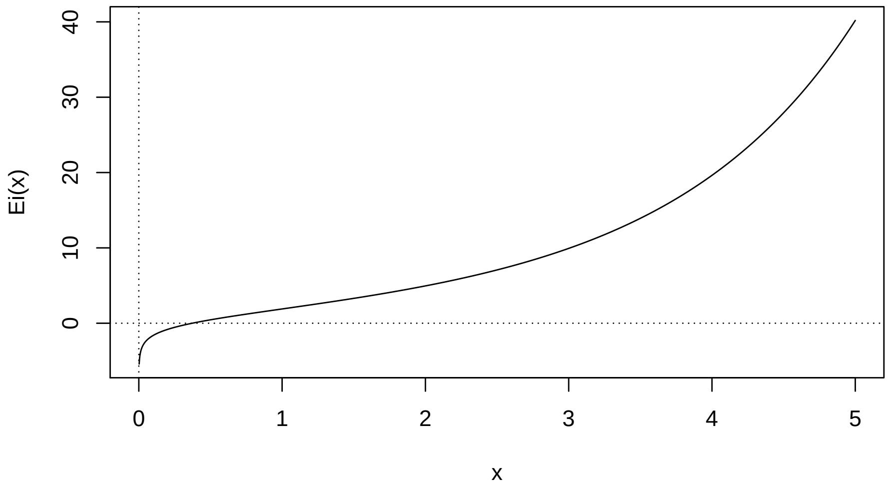
`zeta(x)` computes Riemann's Zeta function $\zeta(x)$ important in analytical number theory and related fields. The traditional definition is

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

`Ei(x)` computes the **e**xponential integral,

$$\int_{-\infty}^{x} \frac{e^t}{t}\, dt.$$

```
R> curve(Ei,  0, 5, n=2001);  abline(h=0,v=0, lty=3)
```



`Li2(x)`, part of the MPFR C library since version 2.4.0, computes the dilogarithm,

$$\mathtt{Li2(x)} = \mathrm{Li}_2(x) := \int_0^x \frac{-log(1-t)}{t}\, dt,$$

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

$$\mathrm{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s},\ \forall s \in \mathbb{C}\ \ \forall |z| < 1, z \in \mathbb{C},$$

see http://en.wikipedia.org/wiki/Polylogarithm#Dilogarithm.

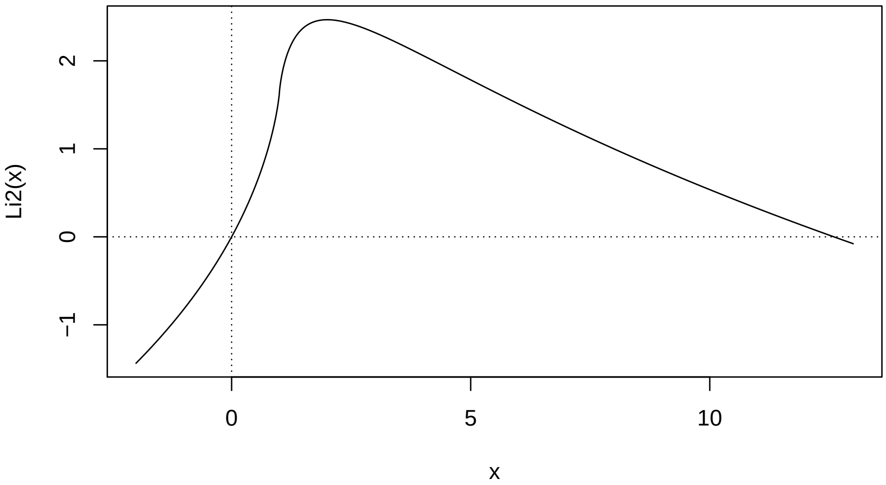
Note that the integral definition is valid for all $x \in \mathbb{C}$, and also, $Li_2(1) = \zeta(2) = \pi^2/6$.

```
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)

[1] TRUE
```

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

```
R> if(mpfrVersion() >= "2.4.0")
    curve(Li2, -2, 13,   n=2000); abline(h=0,v=0, lty=3)
```

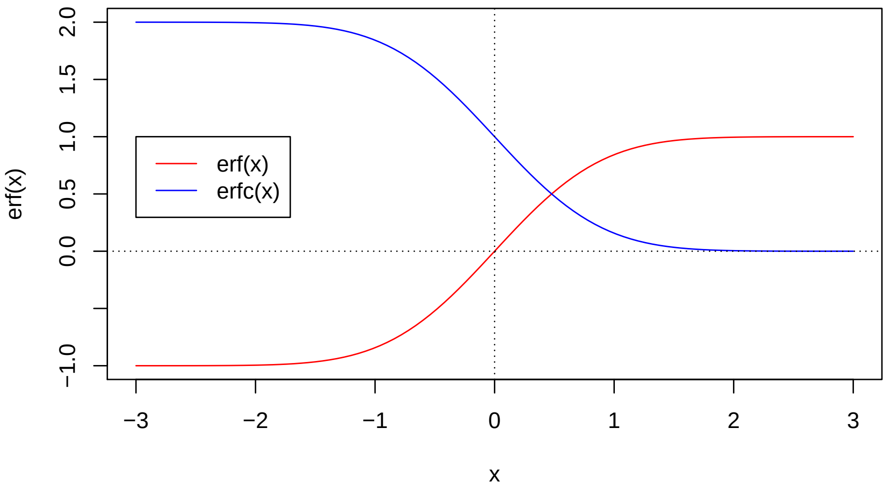


`erf(x)` is the "error[2] function" and `erfc(x)` its **c**omplement, `erfc(x) := 1 - erf(x)`, defined es

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt,$$

and consequently, both functions simply are reparametrizations of `pnorm`, `erf(x) = 2*pnorm(sqrt(2)*x)` and `erfc(x) = 1 - erf(x) = 2* pnorm(sqrt(2)*x, lower=FALSE)`.

```
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)
R> legend(-3,1, c("erf(x)", "erfc(x)"), col = c("red","blue"), lty=1)
```



6. Integration highly precisely

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

[2]named exactly because of its relation to the normal / Gaussian distribution

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> integrateR(dnorm,0,2000)

0.5000413 with absolute error < 4.4e-05

R> integrateR(dnorm,0,2000, rel.tol=1e-15)

0.5 with absolute error < 0

R> integrateR(dnorm,0,2000, rel.tol=1e-15, verbose=TRUE)

n= 1, 2^n=        2 | I = 132.9807601338109    , abs.err =      265.962
n= 2, 2^n=        4 | I = 62.05768806244507    , abs.err =      70.9231
n= 3, 2^n=        8 | I = 30.53632269739361    , abs.err =      31.5214
n= 4, 2^n=       16 | I = 15.20828620615289    , abs.err =       15.328
n= 5, 2^n=       32 | I = 7.596709923112541    , abs.err =      7.61158
n= 6, 2^n=       64 | I = 3.797427402347099    , abs.err =      3.79928
n= 7, 2^n=      128 | I = 1.898597805812433    , abs.err =      1.89883
n= 8, 2^n=      256 | I = 0.9492844175337234   , abs.err =     0.949313
n= 9, 2^n=      512 | I = 0.4757402595960551   , abs.err =     0.473544
n=10, 2^n=     1024 | I = 0.4055234695749382   , abs.err =    0.0702168
n=11, 2^n=     2048 | I = 0.5057584163511011   , abs.err =     0.100235
n=12, 2^n=     4096 | I = 0.5000413486855022   , abs.err =   0.00571707
n=13, 2^n=     8192 | I = 0.4999976613053521   , abs.err =  4.36874e-05
n=14, 2^n=    16384 | I = 0.5000000108190541   , abs.err =  2.34951e-06
n=15, 2^n=    32768 | I = 0.4999999999890231   , abs.err =    1.083e-08
n=16, 2^n=    65536 | I = 0.5000000000000028   , abs.err =  1.09797e-11
n=17, 2^n=   131072 | I = 0.5                  , abs.err =  2.77556e-15
n=18, 2^n=   262144 | I = 0.5                  , abs.err =            0
0.5 with absolute error < 0
```

Now, for situtations 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 $\exp(x) = e^x$ with its well-known $\int \exp(t)\, dt = \exp(x)$, both with standard (double precision) floats,

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

n= 1, 2^n=        2 | I = 1.718861151876593    , abs.err =      0.14028
n= 2, 2^n=        4 | I = 1.718282687924757    , abs.err =  0.000578464
n= 3, 2^n=        8 | I = 1.71828182879453     , abs.err =   8.5913e-07
n= 4, 2^n=       16 | I = 1.718281828459078    , abs.err =  3.35452e-10
n= 5, 2^n=       32 | I = 1.718281828459045    , abs.err =  3.30846e-14
n= 6, 2^n=       64 | I = 1.718281828459045    , abs.err =            0
1.718282 with absolute error < 0
```

and then the same, using 200-bit accurate mpfr-numbers:

```
R> (Ie.m <- integrateR(exp, mpfr(0,200), 1, rel.tol=1e-25, verbose=TRUE))

n= 1, 2^n=        2 | I = 1.718861151876593    , abs.err =      0.14028
n= 2, 2^n=        4 | I = 1.718282687924757    , abs.err =  0.000578464
n= 3, 2^n=        8 | I = 1.71828182879453     , abs.err =   8.5913e-07
n= 4, 2^n=       16 | I = 1.718281828459078    , abs.err =  3.35452e-10
```

```
n= 5, 2^n=        32 | I = 1.718281828459045    , abs.err =  3.30865e-14
n= 6, 2^n=        64 | I = 1.718281828459045    , abs.err =  8.17815e-19
n= 7, 2^n=       128 | I = 1.718281828459045    , abs.err =  5.05653e-24
n= 8, 2^n=       256 | I = 1.718281828459045    , abs.err =  7.81722e-30
1.718282 with absolute error < 7.8e-30
```

```
R> (I.true <- exp(mpfr(1, 200)) - 1)
```

```
1 'mpfr' number of precision  200   bits
[1] 1.7182818284590452353602874713526624977572470936999595749669679
```

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

```
[1] -7.747992e-17 -7.817219e-30
```

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

```
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> (Id <- integrateR(Fp, 0,      5))
```

```
20 with absolute error < 7.1e-15
```

```
R> (Im <- integrateR(Fp, 0, mpfr(5, 256),
                   rel.tol = 1e-70, verbose=TRUE))
```

```
n= 1, 2^n=         2 | I = 46.04166666666666    , abs.err =      98.9583
n= 2, 2^n=         4 | I = 20                   , abs.err =      26.0417
n= 3, 2^n=         8 | I = 20                   , abs.err =  2.76357e-76
20.00000 with absolute error < 2.8e-76
```

```
R> ## and the numerical errors, are indeed of the expected size:
R> 256 * log10(2) # - expect ~ 77 digit accuracy for mpfr(*., 256)
```

```
[1] 77.06368
```

```
R> as.numeric(Itrue - c(Im$value, Id$value))
```

```
[1]  2.763574e-76 -3.552714e-15
```

7. Conclusion

References

Bauer FL (1961). "Algorithm 60: Romberg integration." *Commun. ACM*, **4**, 255. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/366573.366594. URL http://doi.acm.org/10.1145/366573.366594.

Fousse L, Hanrot G, Lefèvre V, Pélissier P, Zimmermann P (2007). "MPFR: A multiple-precision binary floating-point library with correct rounding." *ACM Trans. Math. Softw.*, **33**(2), 13. ISSN 0098-3500. URL http://doi.acm.org/10.1145/1236463.1236468.

Fousse L, Hanrot G, Lefèvre V, Pélissier P, Zimmermann P (2011). *MPFR: A multiple-precision binary floating-point library with correct rounding*. URL http://mpfr.org/.

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

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