Midpoint-radius interval arithmetic, FLINT, and **flint**

Mikael Jagan

Department of Mathematics and Statistics
McMaster University
Hamilton, Ontario, Canada

Research Seminar @ WU
October 1, 2025

Outline

Outline

Some things that I do

- Scientific software (especially R package) development
  - Collaboration with Ben Bolker, Jonathan Dushoff, and David Earn
- Statistical inference for ecological models
  - Variation in initial epidemic growth rates across time series
  - Temporal patterns in unobserved infectious disease transmission rates
- Statistical computing
  - Mixed models (**lme4**, **glmmTMB**)
  - Linear algebra (**Matrix**)
  - Reliable, arbitrary precision arithmetic (**flint**)
- More "interfaces" than "algorithms"

Outline

Arithmetic types in C: Fixed precision

- The built-in types
  - 2^k-bit signed and unsigned integers, $k = 3, 4, 5, 6$, e.g., `long int`
  - real floating types conforming to IEEE 754 standard, e.g., `double` using binary64 format
  - complex floating types (corresponding to real), e.g., `double _Complex`
- Arithmetic is fast, implemented in hardware
- Precision is insufficient in many applications
  - Crytography, number theory
  - Scientific computing: $\mathrm{fl}(2^{100} + 2^{10}) = 2^{100}$ in binary64

Arithmetic types in C: Arbitrary precision

- Precision is limited only by available memory
- Tradeoff: arithmetic is slower, implemented in software
- The "main" libraries and types
  - GNU MP: integer `mpz_t`, rational `mpq_t`, real floating `mpf_t`
  - GNU MPFR: real floating `mpfr_t`
  - GNU MPC: complex floating `mpc_t`
  - FLINT: integer `fmpz_t`, rational `fmpq_t`, real floating `arf_t`, complex floating `acf_t`
- Computation is not *reliable* without formal error analysis

Arithmetic types in C: Intervals, balls, boxes

- Move from floating-point approximations to *enclosures* of images:
  $F(X) = Y \implies f(x) \in Y \quad \text{for all} \quad x \in X$
  - $X, Y \in \mathcal{I} = \{[a, b] : a, b \in \mathbb{R}, a \leq b\}$
  - $X, Y \in \mathcal{I} = \{[m - r, m + r] : m \in \mathbb{R}, r \in \mathbb{R}_{\geq 0}\}$
  - $X, Y \in \mathcal{B} = \{\overline{B_r(m)} : m \in V, r \in \mathbb{R}_{\geq 0}\}$
  - $X, Y \in \mathcal{B}^d$
- Rounding and other errors are propagated rigorously
- Tradeoff: no standard choice of F
- The "main" libraries and types
  - MPFI: real intervals `mpfi_t`
  - MPC: complex balls `mpcb_t`
  - FLINT: real intervals `arb_t`, complex boxes `acb_t`

Outline

The FLINT C library

- FLINT: Fast Library for Number Theory
- Current maintainers: Fredrik Johansson (Inria Bordeaux), Albin Ahlbäck (École polytechnique)
- Efficient arithmetic in rings: $\mathbb{Z}$, $\mathbb{Z}/m\mathbb{Z}$, $\mathbb{F}_q$, $\mathbb{Q}$, $\mathbb{Q}_p$, $\mathbb{R}$, $\mathbb{C}$, $R[x]$, $R[[x]]$, $R^{n\times n}$, ...
- Built on top of GNU MP
- >16000 entry points, >3600 test programs, >600000 lines of code, ...
- Research software

The FLINT C library

- 2023: Arb library is merged into FLINT, adding ...
  - Arbitrary precision floating types `arf_t`, `acf_t`; corresponding midpoint-radius interval types `arb_t`, `acb_t`
  - Analytic functions on real and complex domains (at least the principal branches)
  - Numerical linear algebra: matrix-matrix products, (some) matrix factorizations, linear system solution
  - Polynomial evaluation, differentiation, rootfinding, interpolation, composition, ...

Outline

FLINT types: recursive definition

`ulong`	fixed precision (32- or 64-bit) unsigned integers.
`slong`	fixed precision (32- or 64-bit) signed integers.
`fmpz_t`	arbitrary precision signed integers.
`fmpq_t`	rational numbers with `fmpz_t` numerator and (positive, coprime) denominator.
`mag_t`	unsigned floating-point real numbers with fixed precision (30-bit) significand and `fmpz_t` exponent.
`arf_t`	floating-point real numbers with arbitrary precision significand and `fmpz_t` exponent.
`acf_t`	floating-point complex numbers with `arf_t` real and imaginary parts.
`arb_t`	real intervals with `arf_t` midpoint and `mag_t` radius, of the form $m \pm r$.
`acb_t`	complex boxes with `arb_t` real and imaginary parts, of the form $(m_1 \pm r_1) + (m_2 \pm r_2)i$.

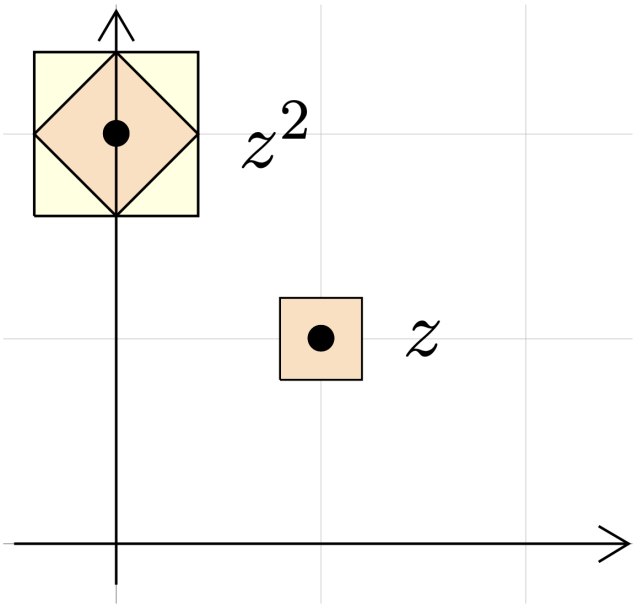
FLINT types: comparison with GNU analogues

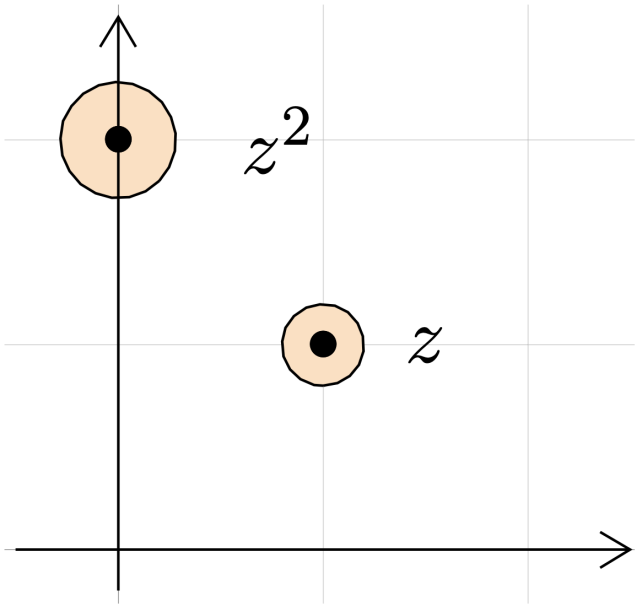
- Integers
  - An `mpz_t` is a C struct
  - An `fmpz_t` is a `slong`
    - If the top two bits are 01, then the `slong` is an index into a heap-allocated array of `mpz_t`
    - Otherwise, it is "as is"
- Real numbers
  - `mpfr_t` exponent is fixed precision
  - `arf_t` exponent is arbitrary precision

FLINT types: comparison with GNU analogues

- Real intervals
  - `mpfi_t` is inf-sup
  - `arb_t` is mid-rad
  - Radius uses floating type with fixed precision significand, halving overhead compared to inf-sup
  - $(-\infty, b]$, $[a, \infty)$ not representable in mid-rad
- Complex enclosures
  - `mpcb_t` is mid-rad (ball)
  - `acb_t` is mid-rad *componentwise* (box)

van der Hoeven (2009): $z \in \mathcal{B}_r(1+i)$





Outline

Motivation

- We needed to compute the analytic continuation of Lerch's transcendent

$$\Phi(z, s, a) = \sum_{k=0}^{\infty} \frac{z^k}{(a+k)^s}, \ |z| < 1 \vee (|z| = 1 \wedge \Re s > 1)$$

- We did not want to leave R
  - **VGAM**'s `lerch` was insufficient
- It seemed to be a common situation
  - CRAN Task View: Numerical Mathematics
- An R interface to FLINT types and functions would solve many problems (in one place)

Outline

Scope

- Provide data types most relevant to R users (applied mathematicians, statisticians)
  - Focus on $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$
- Provide functions "as needed" or "by request"
  - Start with Φ
- Leave nontrivial algorithms to experts
  - Request unimplemented ones upstream
- Avoid complexity until some use case demands it
  - Default to simple, perhaps suboptimal implementations

Outline

Design

- Implement S4 classes representing vectors (including arrays) of FLINT-type numbers and intervals

```
> library(flint)
> cl <- getClass("flint")
> names(cl@subclasses)
[1] "ulong" "slong" "fmpz"  "fmpq"  "mag"
[6] "arf"   "acf"   "arb"   "acb"
```

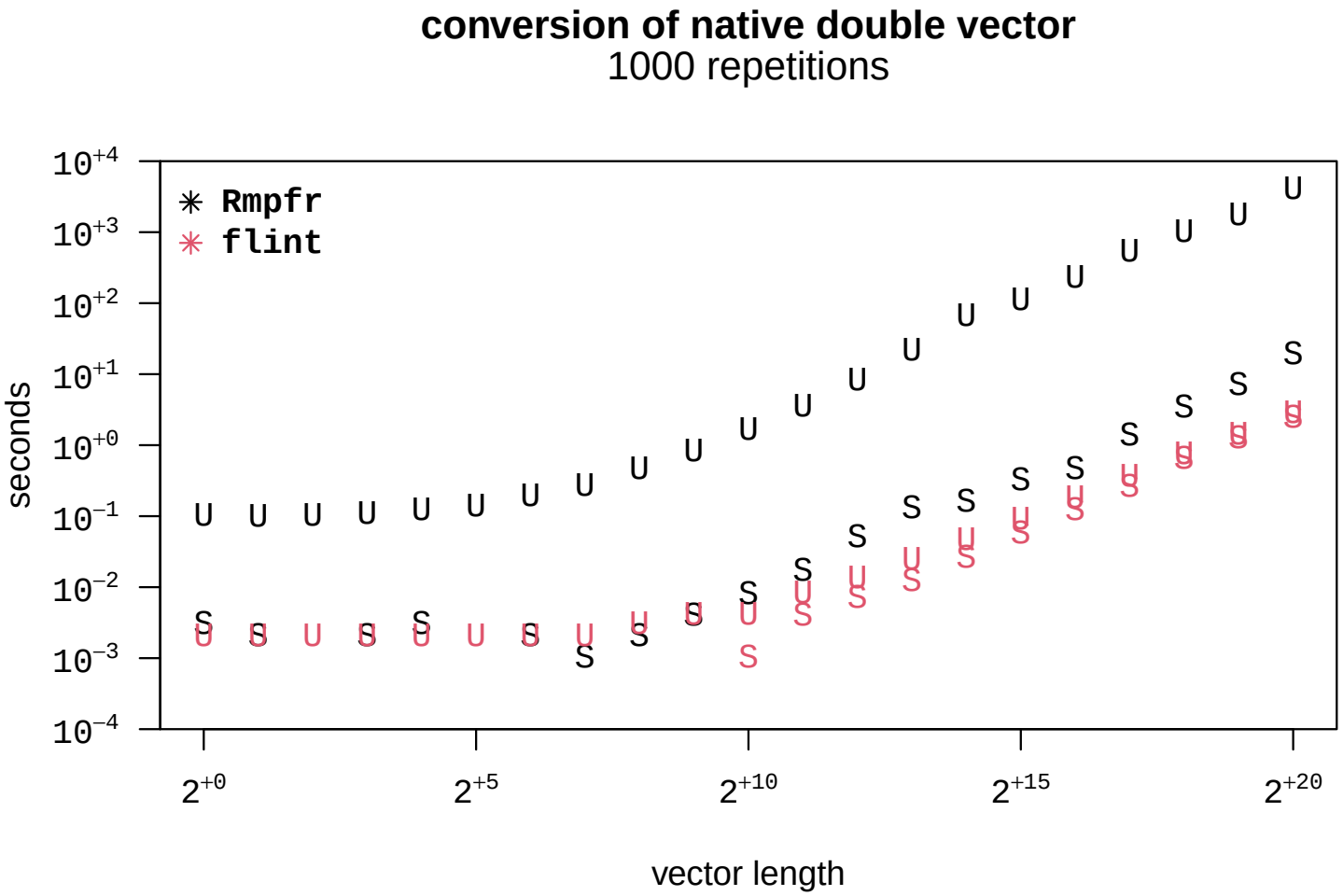
Design

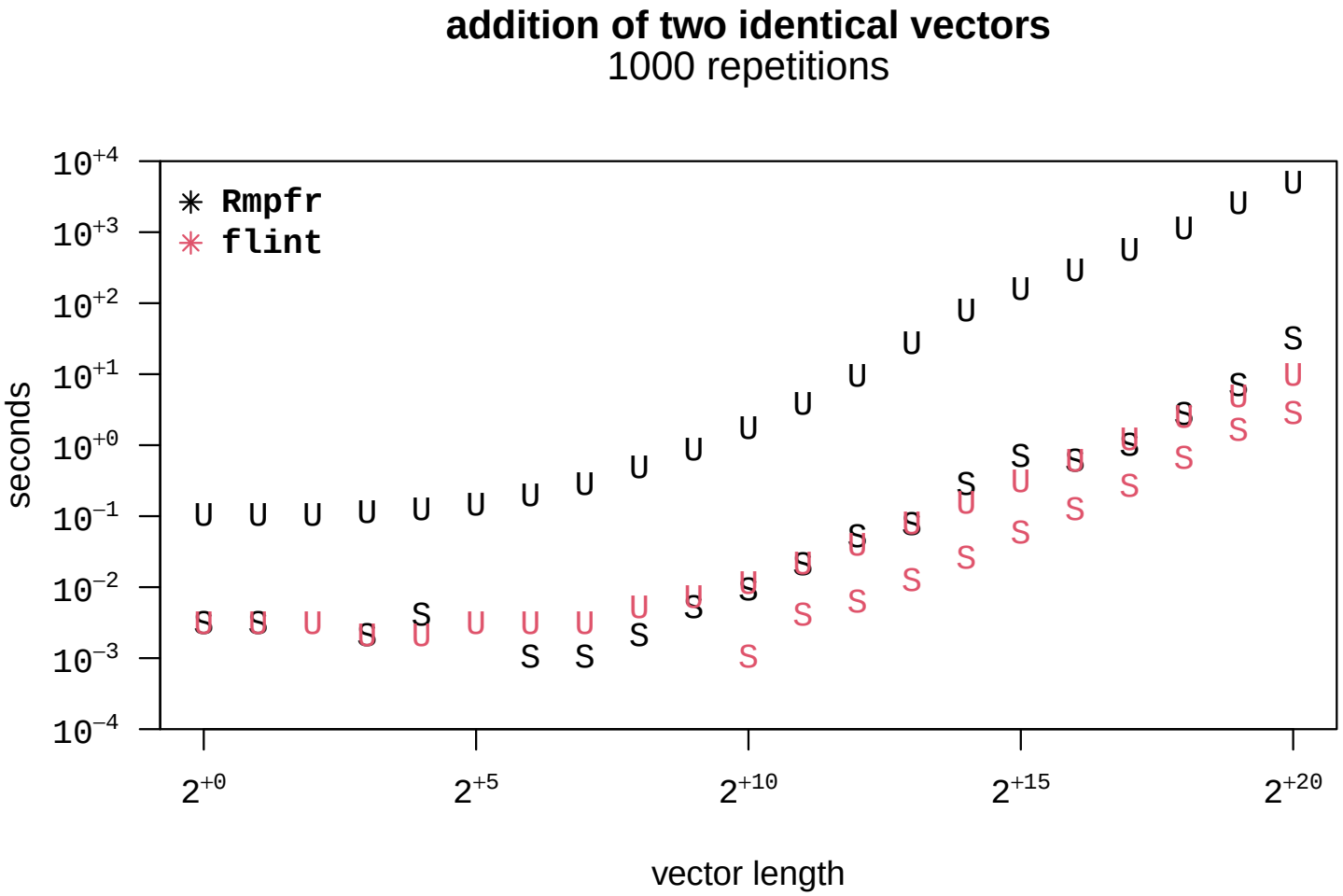
- Have objects store pointers to heap-allocated arrays of the underlying C type

```
> x <- arf(c(0, 1))
> str(x)
Formal class 'arf' [package "flint"]..
  ..@ dim     : NULL
  ..@ dimnames: NULL
  ..@ names   : NULL
  ..@ .xData  :<pointer: 0x600001cec000>
```

- Hence avoid repeated copying as done by **gmp**, **Rmpfr**

```
> y <- Rmpfr::mpfr(c(0, 1), precBits = 53L)
> utils:::str.default(y)
Formal class ‘mpfr’ [package "Rmpfr"] with 1 slot
  ..@ .Data:List of 2
  .. ..$ :Formal class ‘mpfr1’ [package "Rmpfr"]..
  .. .. .. ..@ prec: int 53
  .. .. .. ..@ exp : int [1:2] 1 NA
  .. .. .. ..@ sign: int 1
  .. .. .. ..@ d   : int(0)
  .. ..$ :Formal class ‘mpfr1’ [package "Rmpfr"]..
  .. .. .. ..@ prec: int 53
  .. .. .. ..@ exp : int [1:2] 1 0
  .. .. .. ..@ sign: int 1
  .. .. .. ..@ d   : int [1:2] 0 NA
```


conversion of native double vector
1000 repetitions
Rmpfr
flint
seconds
vector length
10^{+4}
10^{+3}
10^{+2}
10^{+1}
10^{+0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
2^{+0}
2^{+5}
2^{+10}
2^{+15}
2^{+20}


addition of two identical vectors
1000 repetitions
Rmpfr
flint
seconds
vector length
10^{+4}
10^{+3}
10^{+2}
10^{+1}
10^{+0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
2^{+0}
2^{+5}
2^{+10}
2^{+15}
2^{+20}

Design

- Emulate native atomic vectors as closely as possible
- Write S4 methods for "almost all" S4 generic functions available in **base** and **methods**
- Export S4 generic versions of S3 generic functions and critical non-generic functions

```
> G <- getGenerics(getNamespace("flint"))
> length(G)
[1] 100
> G[startsWith(G, "d")]
 [1] "det"          "determinant" "diag"
 [4] "diag<-"       "dim"         "dim<-"
 [7] "dimnames"     "dimnames<-"  "drop"
[10] "duplicated"
```

- Register S3 methods calling S4 generic functions so that S3 generic functions dispatch S4 methods

```
> base::as.matrix
function (x, ...)
UseMethod("as.matrix")
<bytecode: 0x123ede4c8>
<environment: namespace:base>
```

```
> getS3method("as.matrix", "flint")
function (x, ...)
as.matrix(x, ...)
<bytecode: 0x13348c7c0>
<environment: namespace:flint>
```

```
> getDataPart(flint::as.matrix)
function (x, ...)
standardGeneric("as.matrix")
<environment: 0x123f035b8>
```

```
> getDataPart(getMethod("as.matrix", "flint"))
function (x, ...)
{
    ## Redacted.
}
<bytecode: 0x123f3d288>
<environment: namespace:flint>
```

Design

- Enable interoperability with native atomic vectors by generalizing the type hierarchy, from

```
logical < integer < double < complex <
      character
```

to

```
{logical} < {integer, ulong, slong, fmpz} <
      {fmpq} < {double, mag, arf, arb} <
          {complex, acf, acb} < {character}
```

Design

- Make the interface easy to extend
  - Use templates so that data types and functions with common prototypes can share code

```
#define TEMPLATE(name, ptr_t) \
do { \
    ptr_t x = R_flint_get_pointer(s_x); \
    ptr_t y = R_flint_get_pointer(s_y); \
    for (j = 0; j < n; ++j) \
        if (!name##_equal(x__++, y__++)) \
            return 0; \
} while (0)

R_FLINT_SWITCH(class, TEMPLATE);
```

```
SEXP R_flint_arb_hypgeom_bessel_j(
        SEXP s_res, SEXP s_nu, SEXP s_x, SEXP s_prec)
{
    return
    R_flint_arb_2ary(&arb_hypgeom_bessel_j,
                     s_res, s_nu, s_x, s_prec);
}

SEXP R_flint_arb_hypgeom_bessel_y(
        SEXP s_res, SEXP s_nu, SEXP s_x, SEXP s_prec)
{
    return
    R_flint_arb_2ary(&arb_hypgeom_bessel_y,
                     s_res, s_nu, s_x, s_prec);
}
```

Outline

Construction: big integers

```
> fmpz(-2:2)
class "fmpz", length 5, address 0x6000007f0000
[1] -2 -1  0  1  2
> (M <- 0x1p+53) # == 2^53
[1] 9.007199e+15
> M + -2:2 == M
[1] FALSE FALSE  TRUE  TRUE FALSE
> (fM <- fmpz(M))
class "fmpz", length 1, address 0x600000be8070
[1] 9007199254740992
> fM + -2:2
class "fmpz", length 5, address 0x6000007f4060
[1] 9007199254740990 9007199254740991
[3] 9007199254740992 9007199254740993
[5] 9007199254740994
> fmpz("0xFFFFFFFFFFFFFFFFFFFFFFFF") # == 16^24 - 1
class "fmpz", length 1, address 0x600000be80d0
[1] 79228162514264337593543950335
```

Construction: big rationals

```
> fmpq(0.5)
class "fmpq", length 1, address 0x600000bec030
[1] 1/2
> fmpq(0.4)
class "fmpq", length 1, address 0x600000be0000
[1] 3602879701896397/9007199254740992
> fmpq(num = 4L, den = 10L)
class "fmpq", length 1, address 0x600000be00d0
[1] 2/5
```

Construction: real floating intervals

```
> (oprec <- flintPrec(0x1p+20L)) # == 2^20
[1] 53
> (pi.wrong <- arb(pi))
class "arb", length 1, address 0x6000007f4090
[1] (3.141593e+0 +/- 0.000e+0)
> (pi.right <- arb_const_pi())
class "arb", length 1, address 0x6000007f40c0
[1] (3.141593e+0 +/- 5.934e-315653)
> 0x1p+20 * log(2)/log(10)
[1] 315652.8
> print(pi.right, digits = 12L)
class "arb", length 1, address 0x6000007f40c0
[1] (3.14159265359e+0 +/- 5.934e-315653)
> print(pi.right, digits = 12L, base = 2L)
class "arb", length 1, address 0x6000007f40c0
[1] (1.10010010001@+1 +/- 1.001@-1111111111111111110)
> arb(mid = Mid(pi.right), rad = Rad(pi.right) * 2)
class "arb", length 1, address 0x6000007f8030
[1] (3.141593e+0 +/- 1.187e-315652)
```

Construction: with attributes

```
> ulong(c(a = 1L, b = 4L))
class "ulong", length 2, address 0x600000bd00f0
a b
1 4
> (M <- provideDimnames(diag(c(1L, 4L))))
  A B
A 1 0
B 0 4
> ulong(M)
class "ulong", dim (2,2), address 0x6000009f4180
  A B
A 1 0
B 0 4
> as.vector(ulong(M))
[1] 1 0 0 4
> asVector(ulong(M))
class "ulong", length 4, address 0x6000009e8340
[1] 1 0 0 4
```

Construction: arrays

```
> fmpz.array(1:6,
+            dim = c(2L, 3L),
+            dimnames = list(c("otf", "tfs"), NULL))
class "fmpz", dim (2,3), address 0x6000031a0000
    [,1] [,2] [,3]
otf    1    3    5
tfs    2    4    6
> fmpq.array(num = -6:5, den = 12L, dim = c(3L, 4L))
class "fmpq", dim (3,4), address 0x600000a8c000
      [,1]  [,2]  [,3]  [,4]
[1,] -1/ 2 -1/ 4  0/ 1  1/ 4
[2,] -5/12 -1/ 6  1/12  1/ 3
[3,] -1/ 3 -1/12  1/ 6  5/12
```

Construction: missing values

```
> slong(NA)
Error in slong(NA) :
  NaN is not representable by "slong"
> slong(0x1p+63)
Error in slong(9223372036854775808) :
  floating-point number not in range of "slong"
> arb(mid = NA, rad = 0)
class "arb", length 1, address 0x6000031bc090
[1] (NaN +/- 0.000e+0)
> arb(mid = 0, rad = NA)
Error in asMethod(object) :
  NaN is not representable by "mag"
> arb(mid = 0, rad = Inf)
class "arb", length 1, address 0x6000031a4000
[1] (0.000000e+0 +/- Inf)
```

Methods: high coverage

```
> (x <- cumsum(slong(-4:4)))
class "slong", length 9, address 0x6000023ec050
[1]  -4  -7  -9 -10 -10  -9  -7  -4   0
> (D <- as.data.frame(diag(x[5:7])))
   V1 V2 V3
1 -10  0  0
2   0 -9  0
3   0  0 -7
> summary(D)
       V1                 V2                 V3
 Min.   :-10/1   Min.   :-9/1   Min.   :-7/1
 1st Qu.: -5/1   1st Qu.:-9/2   1st Qu.:-7/2
 Median :  0/1   Median : 0/1   Median : 0/1
 Mean   :-10/3   Mean   :-3/1   Mean   :-7/3
 3rd Qu.:  0/1   3rd Qu.: 0/1   3rd Qu.: 0/1
 Max.   :  0/1   Max.   : 0/1   Max.   : 0/1
```

Methods: relations on intervals

```
> x <- arb(mid =             0, rad = 2)
> y <- arb(mid = c(4, 5, NaN), rad = 2)
> z <- arb(mid =             0, rad = 4)
> x < y
[1] FALSE  TRUE    NA
> sort(y)
Error in (function (cond)  :
  error in evaluating the argument 'i' in selecting a
  method for function '[': '<=' is not a total order
  on the range of "arb"
> quantile(arb())
Error in stop(.error.notTotalOrder()) :
  '<=' is not a total order on the range of "arb"
> x <= z
[1] FALSE
> z <= x
[1] FALSE
```

Example: so-called black box algorithm

Goal: Compute $\sin(\pi + \exp(-10000))$ to 53 accurate bits.

```
> x <- sin(pi + exp(-10000))
> x == sin(pi)
[1] TRUE
> x == 0
[1] FALSE
> x
[1] 1.224647e-16
```

Example: so-called black box algorithm

Goal: Compute $\sin(\pi + \exp(-10000))$ to 53 accurate bits.

```
#include <stdio.h>
#include <flint/arb.h>
int main()
{
    slong prec, a;
    arb_t x, y;
    arb_init(x);
    arb_init(y);
    for (prec = 64; ; prec *= 2) {
        arb_const_pi(x, prec);
        arb_set_si(y, -10000);
        arb_exp(y, y, prec);
        arb_add(x, x, y, prec);
        arb_sin(y, x, prec);
        a = arb_rel_accuracy_bits(y);
        printf("%lld\n", (long long int) a);
        if (a >= 53)
            break;
    }
```

```
-4
-4
-3
-5
-3
-5
-4
-3
1953
```

Example: so-called black box algorithm

Goal: Compute $\sin(\pi + \exp(-10000))$ to 53 accurate bits.

```
> prec <- 64
> repeat {
+     flintPrec(prec)
+     y <- sin(arb_const_pi() + exp(arb(-10000)))
+     a <- flintBitsAccurate(y)
+     cat(sprintf("%d\n", as.integer(a)))
+     if (a >= 53L)
+         break
+     prec <- prec * 2
+ }
```

```
-4
-4
-3
-5
-3
-5
-4
-3
1953
```

Example: so-called black box algorithm

```
> y
class "arb", length 1, address 0x60000110d320
[1] (-1.135484e-4343 +/- 6.725e-4932)
> prec
[1] 16384
> prec * log(2)/log(10)
[1] 4932.075
> (s <- format(Mid(y), digits = 54L, base = 2L))
[1] "-1.000010001111001111100101001111111001010001010100001110@-11100
> strsplit(s, "(?<=.{40})", perl = TRUE)[[1L]]
[1] "-1.00001000111100111110010100111111100101"
[2] "0001010100001011@-11100001011011"
```

Example: validated unit tests

Problem: Bad (typically platform-dependent) unit tests in R packages.

```
> stopifnot(log(exp(10)) == 10)
> stopifnot(abs(log(exp(10)) - 10)/10 < 1e-16)
```

Example: validated unit tests

Goal: Determine a valid tolerance, supposing that arithmetic operators and mathematical functions guarantee only 48 accurate bits.

```
> flintPrec(48L)
[1] 53
> log(exp(arb(10)))
class "arb", length 1, address 0x600001760060
[1] (1.000000e+1 +/- 6.216e-14)
> stopifnot(abs(log(exp(10)) - 10)/10 < 7e-14)
```

Outline

To do

- More documentation, examples, "marketing"
- Using **flint** to perform error analysis in R
  - Numerical accuracy of mathematical functions
  - Validity of unit tests in packages
- Numerical linear algebra: QR, SVD, EVD
- Numerical integration: solution of ODE systems
- Random number generation
- Serialization
- Format specifiers (generalizing `sprintf`)
- Classes, methods for representing, operating on polynomials

To thank

- David Earn, McMaster University (initial use case)
- Martin Mächler, ETH Zurich and R Core Team (testing, comments)
- CRAN Team (testing, binary distribution: R package *and* C libraries)
- Many, many contributors to FLINT

To read

- Rump, S. M (2010). Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, *19*, 287–449. https://doi.org/10.1017/S096249291000005X
- van der Hoeven, J. (2009). Ball arithmetic. https://hal.science/hal-00432152
- Johansson, F. (2017). Arb: efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, *66*(8), 1281-1292. https://doi.org/10.1109/TC.2017.2690633
- Johansson, F. (2018). Numerical integration in arbitrary-precision ball arithmetic. https://doi.org/10.48550/arXiv.1802.07942
- Johansson, F. (2019). Faster arbitrary-precision dot product and matrix multiplication. https://doi.org/10.48550/arXiv.1901.04289