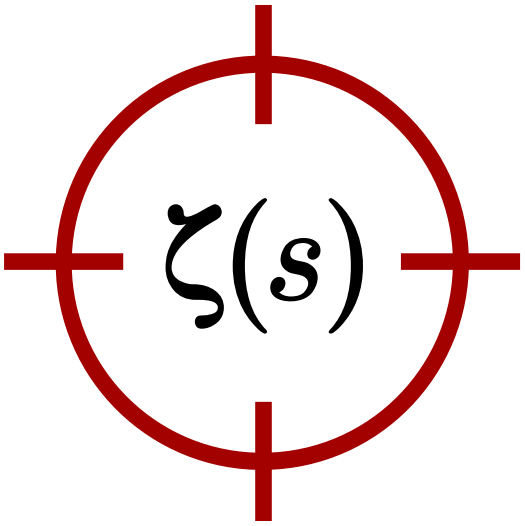

Arb

Arb Documentation

Release 2.9.0-git

Fredrik Johansson

Aug 31, 2016

CHAPTER

ONE

INTRODUCTION

Arb is a C library for arbitrary-precision floating-point ball arithmetic, developed by Fredrik Johansson (fredrik.johansson@gmail.com). It supports real and complex numbers, polynomials, power series, matrices, and evaluation of many transcendental functions. All is done with automatic, rigorous error bounds.

Arb is free software distributed under the GNU Lesser General Public License (LGPL), version 2.1 or later (see *License* for details).

The git repository is `https://github.com/fredrik-johansson/arb/`

The documentation website is `http://fredrikj.net/arb/`

CHAPTER

TWO

GENERAL INFORMATION

2.1 Feature overview

Ball arithmetic, also known as mid-rad interval arithmetic, is an extension of floating-point arithmetic in which an error bound is attached to each variable. This allows computing rigorously with real and complex numbers.

With plain floating-point arithmetic, the user must do an error analysis to guarantee that results are correct. Manual error analysis is time-consuming and bug-prone. Ball arithmetic effectively makes error analysis automatic.

In traditional (inf-sup) interval arithmetic, both endpoints of an interval $[a, b]$ are full-precision numbers, which makes interval arithmetic twice as expensive as floating-point arithmetic. In ball arithmetic, only the midpoint m of an interval $[m \pm r]$ is a full-precision number, and a few bits suffice for the radius r. At high precision, ball arithmetic is therefore not more expensive than plain floating-point arithmetic.

Joris van der Hoeven's paper *[Hoe2009]* is a good introduction to the subject.

Other implementations of ball arithmetic include iRRAM and Mathemagix. Arb differs from earlier implementations in technical aspects of the implementation, which makes certain computations more efficient. It also provides a more comprehensive low-level interface, giving the user full access to the internals. Finally, it implements a wider range of transcendental functions, covering a large portion of the special functions in standard reference works such as *[NIST2012]*.

Arb contains:

- A module (*arf*) for correctly rounded arbitrary-precision floating-point arithmetic. Arb's floating-point numbers have a few special features, such as arbitrary-size exponents (useful for combinatorics and asymptotics) and dynamic allocation (facilitating implementation of hybrid integer/floating-point and mixed-precision algorithms).
- A module (*mag*) for representing magnitudes (error bounds) more efficiently than with an arbitrary-precision floating-point type.
- A module (*arb*) for real ball arithmetic, where a ball is implemented as an *arf* midpoint and a *mag* radius.
- A module (*acb*) for complex numbers in rectangular form, represented as pairs of real balls.
- Modules (*arb_poly*, *acb_poly*) for polynomials or power series over the real and complex numbers, implemented using balls as coefficients, with asymptotically fast polynomial multiplication and many other operations.
- Modules (*arb_mat*, *acb_mat*) for matrices over the real and complex numbers, implemented using balls as coefficients. At the moment, only rudimentary linear algebra operations are provided.
- Functions for high-precision evaluation of various mathematical constants and special functions, implemented using ball arithmetic with rigorous error bounds.

Arb 1.x used a different set of numerical base types (*fmpr*, *fmprb* and *fmpcb*). These types had a slightly simpler internal representation, but generally had worse performance. All methods for the Arb 1.x types have now been ported to faster equivalents for the Arb 2.x types. The last version to include both the

Arb 1.x and Arb 2.x types and methods was Arb 2.2. As of Arb 2.9, only a small set of *fmpr* methods are left for fallback and testing purposes.

Arb uses GMP / MPIR and FLINT for the underlying integer arithmetic and various utility functions. Arb also uses MPFR for testing purposes and internally to evaluate some functions.

2.2 Setup

2.2.1 Download

Tarballs of released versions can be downloaded from https://github.com/fredrik-johansson/arb/releases

Alternatively, you can simply install Arb from a git checkout of https://github.com/fredrik-johansson/arb/. The master branch is generally safe to use (the test suite should pass at all times), and recommended for keeping up with the latest changes.

2.2.2 Dependencies

Arb has the following dependencies:

- Either MPIR (http://www.mpir.org) 2.6.0 or later, or GMP (http://www.gmplib.org) 5.1.0 or later. If MPIR is used instead of GMP, it must be compiled with the `--enable-gmpcompat` option.
- MPFR (http://www.mpfr.org) 3.0.0 or later.
- FLINT (http://www.flintlib.org) version 2.4 or later. You may also use a git checkout of https://github.com/fredrik-johansson/flint2

2.2.3 Installation as part of FLINT

With a sufficiently new version of FLINT, Arb can be compiled as a FLINT extension package.

Simply put the Arb source directory somewhere, say `/path/to/arb`. Then go to the FLINT source directory and build FLINT using:

```
./configure --extensions=/path/to/arb <other options>
make
make check       (optional)
make install
```

This is convenient, as Arb does not need to be configured or linked separately. Arb becomes part of the compiled FLINT library, and the Arb header files will be installed along with the other FLINT header files.

2.2.4 Standalone installation

To compile, test and install Arb from source as a standalone library, first install FLINT. Then go to the Arb source directory and run:

```
./configure <options>
make
make check       (optional)
make install
```

If GMP/MPIR, MPFR or FLINT is installed in some other location than the default path `/usr/local`, pass `--with-gmp=...`, `--with-mpfr=...` or `--with-flint=...` with the correct path to configure (type `./configure --help` to show more options).

2.2.5 Running code

Here is an example program to get started using Arb:

```
#include "arb.h"

int main()
{
    arb_t x;
    arb_init(x);
    arb_const_pi(x, 50 * 3.33);
    arb_printn(x, 50, 0); flint_printf("\n");
    flint_printf("Computed with arb-%s\n", arb_version);
    arb_clear(x);
}
```

Compile it with:

```
gcc -larb test.c
```

or (if Arb is built as part of FLINT):

```
gcc -lflint test.c
```

If the Arb/FLINT header and library files are not in a standard location (`/usr/local` on most systems), you may also have to pass options such as:

```
-I/path/to/arb -I/path/to/flint -L/path/to/flint -L/path/to/arb
```

to `gcc`. Finally, to run the program, make sure that the linker can find the FLINT (and Arb) libraries. If they are installed in a nonstandard location, you can for example add this path to the `LD_LIBRARY_PATH` environment variable.

The output of the example program should be something like the following:

```
[3.14159265358979323846264338327950288419716939937751 +/- 6.28e-50]
Computed with arb-2.4.0
```

2.3 Using ball arithmetic

This section gives an introduction to working with real numbers in Arb (see *arb.h – real numbers* for the API and technical documentation). The general principles carry over to complex numbers, polynomials and matrices.

2.3.1 Ball semantics

Let $f : A \to B$ be a function. A ball implementation of f is a function F that maps sets $X \subseteq A$ to sets $F(X) \subseteq B$ subject to the following rule:

> For all $x \in X$, we have $f(x) \in F(X)$.

In other words, $F(X)$ is an *enclosure* for the set $\{f(x) : x \in X\}$. This rule is sometimes called the *inclusion principle*.

Throughout the documentation (except where otherwise noted), we will simply write $f(x)$ instead of $F(X)$ when describing ball implementations of pointwise-defined mathematical functions, understanding that the input is a set of point values and that the output is an enclosure.

General subsets of $\mathbb{R}$ are not possible to represent on a computer. Instead, we work with subsets of the form $[m \pm r] = [m - r, m + r]$ where the midpoint m and radius r are binary floating-point numbers, i.e.

numbers of the form $u2^v$ with $u, v \in \mathbb{Z}$ (to make this scheme complete, we also need to adjoin the special floating-point values $-\infty$, $+\infty$ and NaN).

Given a ball $[m \pm r]$ with $m \in \mathbb{R}$ (not necessarily a floating-point number), we can always round m to a nearby floating-point number that has at most most *prec* bits in the component u, and add an upper bound for the rounding error to r. In Arb, ball functions that take a *prec* argument as input (e.g. *arb_add()*) always round their output to *prec* bits. Some functions are always exact (e.g. *arb_neg()*), and thus do not take a *prec* argument.

The programming interface resembles that of GMP. Each *arb_t* variable must be initialized with *arb_init()* before use (this also sets its value to zero), and deallocated with *arb_clear()* after use. Variables have pass-by-reference semantics. In the list of arguments to a function, output variables come first, followed by input variables, and finally the precision:

```
#include "arb.h"

int main()
{
    arb_t x, y;
    arb_init(x); arb_init(y);
    arb_set_ui(x, 3);       /* x = 3 */
    arb_const_pi(y, 128);   /* y = pi, to 128 bits */
    arb_sub(y, y, x, 53);   /* y = y - x, to 53 bits */
    arb_clear(x); arb_clear(y);
}
```

2.3.2 Binary and decimal

While the internal representation uses binary floating-point numbers, it is usually preferable to print numbers in decimal. The binary-to-decimal conversion generally requires rounding. Three different methods are available for printing a number to standard output:

- *arb_print()* shows the exact internal representation of a ball, with binary exponents.
- *arb_printd()* shows an inexact view of the internal representation, approximated by decimal floating-point numbers.
- *arb_printn()* shows a *decimal ball* that is guaranteed to be an enclosure of the binary floating-point ball. By default, it only prints digits in the midpoint that are certain to be correct, up to an error of at most one unit in the last place. Converting from binary to decimal is generally inexact, and the output of this method takes this rounding into account when printing the radius.

This snippet computes a 53-bit enclosure of π and prints it in three ways:

```
arb_const_pi(x, 53);
arb_print(x); printf("\n");
arb_printd(x, 20); printf("\n");
arb_printn(x, 20, 0); printf("\n");
```

The output is:

```
(884279719003555 * 2^-48) +/- (536870913 * 2^-80)
3.141592653589793116 +/- 4.4409e-16
[3.141592653589793 +/- 5.61e-16]
```

The *arb_get_str()* and *arb_set_str()* methods are useful for converting rigorously between decimal strings and binary balls (*arb_get_str()* produces the same string as *arb_printn()*, and *arb_set_str()* can parse such strings back).

A potential mistake is to create a ball from a `double` constant such as `2.3`, when this actually represents `2.29999999999999982236431605997495353221893310546875`. To produce a ball containing the rational number 23/10, one of the following can be used:

```
arb_set_str(x, "2.3", prec)

arb_set_ui(x, 23);
arb_div_ui(x, x, 10, prec)

fmpq_set_si(q, 23, 10);   /* q is a FLINT fmpq_t */
arb_set_fmpq(x, q, prec);
```

2.3.3 Quality of enclosures

The main problem when working with ball arithmetic (or interval arithmetic) is *overestimation*. In general, the enclosure of a value or set of values as computed with ball arithmetic will be larger than the smallest possible enclosure.

Overestimation results naturally from rounding errors and cancellations in the individual steps of a calculation. As a general principle, formula rewriting techniques that make floating-point code more numerically stable also make ball arithmetic code more numerically stable, in the sense of producing tighter enclosures.

As a result of the *dependency problem*, ball or interval arithmetic can produce error bounds that are much larger than the actual numerical errors resulting from doing floating-point arithmetic. Consider the expression $(x + 1) - x$ as an example. When evaluated in floating-point arithmetic, x may have a large initial error. However, that error will cancel itself out in the subtraction, so that the result equals 1 (except perhaps for a small rounding error left from the operation $x + 1$). In ball arithmetic, dependent errors add up instead of cancelling out. If $x = [3 \pm 0.1]$, the result will be $[1 \pm 0.2]$, where the error bound has doubled. In unfavorable circumstances, error bounds can grow exponentially with the number of steps.

If all inputs to a calculation are "point values", i.e. exact numbers and known mathematical constants that can be approximated arbitrarily closely (such as π), then an error of order 2^n can typically be overcome by working with n extra bits of precision, increasing the computation time by an amount that is polynomial in n. In certain situations, however, overestimation leads to exponential slowdown or even failure of an algorithm to converge. For example, root-finding algorithms that refine the result iteratively may fail to converge in ball arithmetic, even if they do converge in plain floating-point arithmetic.

Therefore, ball arithmetic is not a silver bullet: there will always be situations where some amount of numerical or mathematical analysis is required. Some experimentation may be required to find whether (and how) it can be used effectively for a given problem.

2.3.4 Predicates

A ball implementation of a predicate $f : \mathbb{R} \to \{\text{True}, \text{False}\}$ would need to be able to return a third logical value indicating that the result could be either True or False. In most cases, predicates in Arb are implemented as functions that return the *int* value 1 to indicate that the result certainly is True, and the *int* value 0 to indicate that the result could be either True or False. To test whether a predicate certainly is False, the user must test whether the negated predicate certainly is True.

For example, the following code would *not* be correct in general:

```
if (arb_is_positive(x))
{
    ...  /* do things assuming that x > 0 */
}
else
{
    ...  /* do things assuming that x <= 0 */
}
```

Instead, the following can be used:

```
if (arb_is_positive(x))
{
    ...  /* do things assuming that x > 0 */
}
else if (arb_is_nonpositive(x))
{
    ...  /* do things assuming that x <= 0 */
}
else
{
    ... /* do things assuming that the sign of x is unknown */
}
```

Likewise, we will write $x \le y$ in mathematical notation with the meaning that $x \le y$ holds for all $x \in X, y \in Y$ where X and Y are balls.

Note that some predicates such as *arb_overlaps()* and *arb_contains()* actually are predicates on balls viewed as sets, and not ball implementations of pointwise predicates.

Some predicates are also complementary. For example *arb_contains_zero()* tests whether the input ball contains the point zero. Negated, it is equivalent to *arb_is_nonzero()*, and complementary to *arb_is_zero()* as a pointwise predicate:

```
if (arb_is_zero(x))
{
    ...  /* do things assuming that x = 0 */
}
#if 1
else if (arb_is_nonzero(x))
#else
else if (!arb_contains_zero(x))      /* equivalent */
#endif
{
    ...  /* do things assuming that x != 0 */
}
else
{
    ... /* do things assuming that the sign of x is unknown */
}
```

2.3.5 A worked example: the sine function

We implement the function $\sin(x)$ naively using the Taylor series $\sum_{k=0}^{\infty} (-1)^k x^{2k+1}/(2k+1)!$ and *arb_t* arithmetic. Since there are infinitely many terms, we need to split the series in two parts: a finite sum that can be evaluated directly, and a tail that has to be bounded.

We stop as soon as we reach a term t bounded by $|t| \le 2^{-prec} < 1$. The terms are alternating and must have decreasing magnitude from that point, so the tail of the series is bounded by $|t|$. We add this magnitude to the radius of the output. Since ball arithmetic automatically bounds the numerical errors resulting from all arithmetic operations, the output *res* is a ball guaranteed to contain $\sin(x)$.

```
#include "arb.h"

void arb_sin_naive(arb_t res, const arb_t x, slong prec)
{
    arb_t s, t, u, tol;
    slong k;
    arb_init(s); arb_init(t); arb_init(u); arb_init(tol);

    arb_one(tol);
    arb_mul_2exp_si(tol, tol, -prec);  /* tol = 2^-prec */
```

```
    for (k = 0; ; k++)
    {
        arb_pow_ui(t, x, 2 * k + 1, prec);
        arb_fac_ui(u, 2 * k + 1, prec);
        arb_div(t, t, u, prec);  /* t = x^(2k+1) / (2k+1)! */

        arb_abs(u, t);
        if (arb_le(u, tol))   /* if |t| <= 2^-prec */
        {
            arb_add_error(s, u);    /* add |t| to the radius and stop */
            break;
        }

        if (k % 2 == 0)
            arb_add(s, s, t, prec);
        else
            arb_sub(s, s, t, prec);

    }

    arb_set(res, s);
    arb_clear(s); arb_clear(t); arb_clear(u); arb_clear(tol);
}
```

This algorithm is naive, because the Taylor series is slow to converge and suffers from catastrophic cancellation when $|x|$ is large (we could also improve the efficiency of the code slightly by computing the terms using recurrence relations instead of computing x^k and $k!$ from scratch each iteration).

As a test, we compute $\sin(2016.1)$. The largest term in the Taylor series for $\sin(x)$ reaches a magnitude of about $x^x/x!$, or about 10^{873} in this case. Therefore, we need over 873 digits (about 3000 bits) of precision to overcome the catastrophic cancellation and determine the result with sufficient accuracy to tell whether it is positive or negative.

```
int main()
{
    arb_t x, y;
    slong prec;
    arb_init(x); arb_init(y);

    for (prec = 64; ; prec *= 2)
    {
        arb_set_str(x, "2016.1", prec);
        arb_sin_naive(y, x, prec);
        printf("Using %5ld bits, sin(x) = ", prec);
        arb_printn(y, 10, 0); printf("\n");
        if (!arb_contains_zero(y))  /* stopping condition */
            break;
    }

    arb_clear(x); arb_clear(y);
}
```

The program produces the following output:

```
Using    64 bits, sin(x) = [+/- 2.67e+859]
Using   128 bits, sin(x) = [+/- 1.30e+840]
Using   256 bits, sin(x) = [+/- 3.60e+801]
Using   512 bits, sin(x) = [+/- 3.01e+724]
Using  1024 bits, sin(x) = [+/- 2.18e+570]
Using  2048 bits, sin(x) = [+/- 1.22e+262]
Using  4096 bits, sin(x) = [-0.7190842207 +/- 1.20e-11]
```

As an exercise, the reader may improve the naive algorithm by making it subtract a well-chosen mul-

tiple of 2π from x before invoking the Taylor series (hint: use *arb_const_pi()*, *arb_div()* and *arf_get_fmpz()*). If done correctly, 64 bits of precision should be more than enough to compute sin(2016.1), and with minor adjustments to the code, the user should be able to compute sin(exp(2016.1)) quite easily as well.

This example illustrates how ball arithmetic can be used to perform nontrivial calculations. To evaluate an infinite series, the user needs to know how to bound the tail of the series, but everything else is automatic. When evaluating a finite formula that can be expressed completely using built-in functions, all error bounding is automatic from the point of view of the user. In particular, the *arb_sin()* method should be used to compute the sine of a real number; it uses a much more efficient algorithm than the naive code above.

This example also illustrates the "guess-and-verify" paradigm: instead of determining *a priori* the floating-point precision necessary to get a correct result, we *guess* some initial precision, use ball arithmetic to *verify* that the result is accurate enough, and restart with higher precision (or signal failure) if it is not.

If we think of rounding errors as essentially random processes, then a floating-point computation is analogous to a *Monte Carlo algorithm*. Using ball arithmetic to get a verified result effectively turns it into the analog of a *Las Vegas algorithm*, which is a randomized algorithm that always gives a correct result if it terminates, but may fail to terminate (alternatively, instead of actually looping forever, it might signal failure after a certain number of iterations).

The loop will fail to terminate if we attempt to determine the sign of $\sin(\pi)$:

```
Using    64 bits, sin(x) = [+/- 3.96e-18]
Using   128 bits, sin(x) = [+/- 2.17e-37]
Using   256 bits, sin(x) = [+/- 6.10e-76]
Using   512 bits, sin(x) = [+/- 5.13e-153]
Using  1024 bits, sin(x) = [+/- 4.01e-307]
Using  2048 bits, sin(x) = [+/- 2.13e-615]
Using  4096 bits, sin(x) = [+/- 6.85e-1232]
Using  8192 bits, sin(x) = [+/- 6.46e-2465]
Using 16384 bits, sin(x) = [+/- 5.09e-4931]
Using 32768 bits, sin(x) = [+/- 5.41e-9863]
...
```

The sign of a nonzero real number can be decided by computing it to sufficiently high accuracy, but the sign of an expression that is exactly equal to zero cannot be decided by a numerical computation unless the entire computation happens to be exact (in this example, we could use the *arb_sin_pi()* function which computes $\sin(\pi x)$ in one step, with the input $x = 1$).

It is up to the user to implement a stopping criterion appropriate for the circumstances of a given application. For example, breaking when it is clear that $|\sin(x)| < 10^{-10000}$ would allow the program to terminate and convey some meaningful information about the input $x = \pi$, though this would not constitute a mathematical proof that $\sin(\pi) = 0$.

2.3.6 More on precision and accuracy

The relation between the working precision and the accuracy of the output is not always easy predict. The following remarks might help to choose *prec* optimally.

For a ball $[m \pm r]$ it is convenient to define the following notions:

- Absolute error: $e_{abs} = |r|$
- Relative error: $e_{rel} = |r| / \max(0, |m| - |r|)$ (or $e_{rel} = 0$ if $r = m = 0$)
- Absolute accuracy: $a_{abs} = 1/e_{abs}$
- Relative accuracy: $a_{rel} = 1/e_{rel}$

Expressed in bits, one takes the corresponding $\log_2$ values.

Of course, if x is the exact value being approximated, then the "absolute error" so defined is an upper bound for the actual absolute error $|x - m|$ and "absolute accuracy" a lower bound for $1/|x - m|$, etc.

The *prec* argument in Arb should be thought of as controlling the working precision. Generically, when evaluating a fixed expression (that is, when the sequence of operations does not depend on the precision), the absolute or relative error will be bounded by

$$2^{O(1)-prec}$$

where the $O(1)$ term depends on the expression and implementation details of the ball functions used to evaluate it. Accordingly, for an accuracy of p bits, we need to use a working precision $O(1) + p$. If the expression is numerically well-behaved, then the $O(1)$ term will be small, which leads to the heuristic of "adding a few guard bits" (for most basic calculations, 10 or 20 guard bits is enough). If the $O(1)$ term is unknown, then increasing the number of guard bits in exponential steps until the result is accurate enough is generally a good heuristic.

Sometimes, a partially accurate result can be used to estimate the $O(1)$ term. For example, if the goal is to achieve 100 bits of accuracy and a precision of 120 bits yields 80 bits of accuracy, then it is plausible that a precision of just over 140 bits yields 100 bits of accuracy.

Built-in functions in Arb can roughly be characterized as belonging to one of two extremes (though there is actually a spectrum):

- Simple operations, including basic arithmetic operations and many elementary functions. In most cases, for an input $x = [m \pm r]$, $f(x)$ is evaluated by computing $f(m)$ and then separately bounding the *propagated error* $|f(m) - f(m + \varepsilon)|, |\varepsilon| \le r$. The working precision is automatically increased internally so that $f(m)$ is computed to *prec* bits of relative accuracy with an error of at most a few units in the last place (perhaps with rare exceptions). The propagated error can generally be bounded quite tightly as well (see *General formulas and bounds*). As a result, the enclosure will be close to the best possible at the given precision, and the user can estimate the precision to use accordingly.
- Complex operations, such as certain higher transcendental functions (for example, the Riemann zeta function). The function is evaluated by performing a sequence of simpler operations, each using ball arithmetic with a working precision of roughly *prec* bits. The sequence of operations might depend on *prec*; for example, an infinite series might be truncated so that the remainder is smaller than 2^{-prec}. The final result can be far from tight, and it is not guaranteed that the error converges to zero as $prec \to \infty$, though in practice, it should do so in most cases.

In short, the *inclusion principle* is the fundamental contract in Arb. Enclosures computed by built-in functions may or may not be tight enough to be useful, but the hope is that they will be sufficient for most purposes. Tightening the error bounds for more complex operations is a long term optimization goal, which in many cases will require a fair amount of research. A tradeoff also has to be made for efficiency: tighter error bounds allow the user to work with a lower precision, but they may also be much more expensive to compute.

2.3.7 Polynomial time guarantee

Arb provides a soft guarantee that the time used to evaluate a ball function will depend polynomially on *prec* and the bit size of the input, uniformly regardless of the numerical value of the input.

The idea behind this soft guarantee is to allow Arb to be used as a black box to evaluate expressions numerically without potentially slowing down, hanging indefinitely or crashing because of "bad" input such as nested exponentials. By controlling the precision, the user can cancel a computation before it uses up an unreasonable amount of resources, without having to rely on other timeout or exception mechanisms. A result that is feasible but very expensive to compute can still be forced by setting the precision high enough.

As motivation, consider evaluating $\sin(x)$ or $\exp(x)$ with the exact floating-point number $x = 2^{2^n}$ as input. The time and space required to compute an accurate floating-point approximation of $\sin(x)$ or $\exp(x)$ increases as 2^n, in the first case because because of the need to subtract an accurate multiple of 2π and in the second case due to the size of the output exponent and the internal subtraction of

an accurate multiple of log(2). This is despite the fact that the size of x as an object in memory only increases linearly with n. Already $n = 33$ would require at least 1 GB of memory, and $n = 100$ would be physically impossible to process. For functions that are computed by direct use of power series expansions, e.g. $f(x) = \sum_{k=0}^{\infty} c_k x^k$, without having fast argument-reduction techniques like those for elementary functions, the time would be exponential in n already when $x = 2^n$.

Therefore, Arb caps internal work parameters (the internal working precision, the number terms of an infinite series to add, etc.) by polynomial, usually linear, functions of *prec*. When the limit is exceeded, the output is set to a crude bound. For example, if x is too large, `arb_sin()` will simply return $[\pm 1]$, and `arb_exp()` will simply return $[\pm\infty]$ if x is positive or $[\pm 2^{-m}]$ if x is negative.

This is not just a failsafe, but occasionally a useful optimization. It is not entirely uncommon to have formulas where one term is modest and another term decreases exponentially, such as:

$$\log(x) + \sin(x)\exp(-x).$$

For example, the reflection formula of the digamma function has a similar structure. When x is large, the right term would be expensive to compute to high relative accuracy. Doing so is unnecessary, however, since a crude bound of $[\pm 1] \cdot [\pm 2^{-m}]$ is enough to evaluate the expression as a whole accurately.

The polynomial time guarantee is "soft" in that there are a few exceptions. For example, the complexity of computing the Riemann zeta function $\zeta(\sigma + it)$ increases linearly with the imaginary height $|t|$ in the current implementation, and all known algorithms have a complexity of $|t|^{\alpha}$ where the best known value for α is about 0.3. Input with large $|t|$ is most likely to be given deliberately by users with the explicit intent of evaluating the zeta function itself, so the evaluation is not cut off automatically.

2.4 Technical conventions and potential issues

2.4.1 Integer types

Arb generally uses the *int* type for boolean values and status flags.

The *char*, *short* and *int* types are assumed to be two's complement types with exactly 8, 16 and 32 bits. This is not technically guaranteed by the C standard, but there are no mainstream platforms where this assumption does not hold, and new ones are unlikely to appear in the near future (ignoring certain low-power DSPs and the like, which are out of scope for this software).

Since the C types *long* and *unsigned long* do not have a standardized size in practice, FLINT defines *slong* and *ulong* types which are guaranteed to be 32 bits on a 32-bit system and 64 bits on a 64-bit system. They are also guaranteed to have the same size as GMP's `mp_limb_t`. GMP builds with a different limb size configuration are not supported at all. For convenience, the macro *FLINT_BITS* specifies the word length (32 or 64) of the system.

`slong`
: The *slong* type is used for precisions, bit counts, loop indices, array sizes, and the like, even when those values are known to be nonnegative. It is also used for small integer-valued coefficients. In method names, an *slong* parameter is denoted by *si*, for example `arb_add_si()`.

  The constants *WORD_MIN* and *WORD_MAX* give the range of this type. This type can be printed with *flint_printf* using the format string `%wd`.

`ulong`
: The *ulong* type is used for integer-valued coefficients that are known to be unsigned, and for values that require the full 32-bit or 64-bit range. In method names, a *ulong* parameter is denoted by *ui*, for example `arb_add_ui()`.

  The constant *UWORD_MAX* gives the range of this type. This type can be printed with *flint_printf* using the format string `%wu`.

The following GMP-defined types are used in methods that manipulate the internal representation of numbers (using limb arrays).

mp_limb_t
A single limb.

mp_ptr
Pointer to a writable array of limbs.

mp_srcptr
Pointer to a read-only array of limbs.

mp_size_t
A limb count (always nonnegative).

mp_bitcnt_t
A bit offset within an array of limbs (always nonnegative).

Arb uses the following FLINT types for exact (integral and rational) arbitrary-size values. For details, refer to the FLINT documentation.

fmpz_t
The FLINT multi-precision integer type uses an inline representation for small integers, specifically when the absolute value is at most $2^{62} - 1$ (on 64-bit machines) or $2^{30} - 1$ (on 32-bit machines). It switches automatically to a GMP integer for larger values. The *fmpz_t* type is functionally identical to the GMP *mpz_t* type, but faster for small values.

fmpq_t
FLINT multi-precision rational number.

fmpz_poly_t

fmpq_poly_t

fmpz_mat_t

fmpq_mat_t
FLINT polynomials and matrices with integer and rational coefficients.

2.4.2 Integer overflow

When machine-size integers are used for precisions, sizes of integers in bits, lengths of polynomials, and similar quantities that relate to sizes in memory, very few internal checks are performed to verify that such quantities do not overflow.

Precisions and lengths exceeding a small fraction of *LONG_MAX*, say $2^{24} \approx 10^7$ on 32-bit systems, should be regarded as resulting in undefined behavior. On 64-bit systems this should generally not be an issue, since most calculations will exhaust the available memory (or the user's patience waiting for the computation to complete) long before running into integer overflows. However, the user needs to be wary of unintentionally passing input parameters of order *LONG_MAX* or negative parameters where positive parameters are expected, for example due to a runaway loop that repeatedly increases the precision.

Currently, no hard upper limit on the precision is defined, but $2^{24} \approx 10^7$ bits on 32-bit system and $2^{36} \approx 10^{11}$ bits on a 64-bit system can be considered safe for most purposes. The relatively low limit on 64-bit systems is due to the fact that GMP integers are used internally in some algorithms, and GMP integers are limited to 2^{37} bits. The minimum allowed precision is 2 bits.

This caveat does not apply to exponents of floating-point numbers, which are represented as arbitrary-precision integers, nor to integers used as numerical scalars (e.g. *arb_mul_si()*). However, it still applies to conversions and operations where the result is requested exactly and sizes become an issue. For example, trying to convert the floating-point number $2^{2^{100}}$ to an integer could result in anything from a silent wrong value to thrashing followed by a crash, and it is the user's responsibility not to attempt such a thing.

2.4.3 Aliasing

As a rule, Arb allows aliasing of operands. For example, in the function call `arb_add(z, x, y, prec)`, which performs $z \leftarrow x + y$, any two (or all three) of the variables x, y and z are allowed to be the same. Exceptions to this rule are documented explicitly.

The general rule that input and output variables can be aliased with each other only applies to variables *of the same type* (ignoring *const* qualifiers on input variables – a special case is that *arb_srcptr* is considered the *const* version of *arb_ptr*). This is a natural extension of the so-called *strict aliasing rule* in C.

For example, in *arb_poly_evaluate()* which evaluates $y = f(x)$ for a polynomial f, the output variable y is not allowed to be a pointer to one of the coefficients of f (but aliasing between x and y or between x and the coefficients of f is allowed). This also applies to *_arb_poly_evaluate()*: for the purposes of aliasing, *arb_srcptr* (the type of the coefficient array within f) and *arb_t* (the type of x) are *not* considered to be the same type, and therefore must not be aliased with each other, even though an *arb_ptr*/*arb_srcptr* variable pointing to a length 1 array would otherwise be interchangeable with an *arb_t*/*const arb_t*.

Moreover, in functions that allow aliasing between an input array and an output array, the arrays must either be identical or completely disjoint, never partially overlapping.

There are natural exceptions to these aliasing restrictions, which may used internally without being documented explicitly. However, third party code should avoid relying on such exceptions.

An important caveat applies to **aliasing of input variables**. Identical pointers are understood to give permission for **algebraic simplification**. This assumption is made to improve performance. For example, the call `arb_mul(z, x, x, prec)` sets z to a ball enclosing the set

$$\{t^2 : t \in x\}$$

and not the (generally larger) set

$$\{tu : t \in x, u \in x\}.$$

If the user knows that two values x and y both lie in the interval $[-1, 1]$ and wants to compute an enclosure for $f(x, y)$, then it would be a mistake to create an *arb_t* variable x enclosing $[-1, 1]$ and reusing the same variable for y, calling $f(x, x)$. Instead, the user has to create a distinct variable y also enclosing $[-1, 1]$.

Algebraic simplification is not guaranteed to occur. For example, `arb_add(z, x, x, prec)` and `arb_sub(z, x, x, prec)` currently do not implement this optimization. It is better to use `arb_mul_2exp_si(z, x, 1)` and `arb_zero(z)`, respectively.

2.4.4 Thread safety and caches

Arb should be fully threadsafe, provided that both MPFR and FLINT have been built in threadsafe mode. Use `flint_set_num_threads()` to set the number of threads that Arb is allowed to use internally for single computations (this is currently only exploited by a handful of operations). Please note that thread safety is only tested minimally, and extra caution when developing multithreaded code is therefore recommended.

Arb may cache some data (such as the value of π and Bernoulli numbers) to speed up various computations. In threadsafe mode, caches use thread-local storage. There is currently no way to save memory and avoid recomputation by having several threads share the same cache. Caches can be freed by calling the `flint_cleanup()` function. To avoid memory leaks, the user should call `flint_cleanup()` when exiting a thread. It is also recommended to call `flint_cleanup()` when exiting the main program (this should result in a clean output when running Valgrind, and can help catching memory issues).

There does not seem to be an obvious way to make sure that `flint_cleanup()` is called when exiting a thread using OpenMP. A possible solution to this problem is to use OpenMP sections, or to use C++ and create a thread-local object whose destructor invokes `flint_cleanup()`.

2.4.5 Use of hardware floating-point arithmetic

Arb uses hardware floating-point arithmetic (the `double` type in C) in two different ways.

Firstly, `double` arithmetic as well as transcendental `libm` functions (such as `exp`, `log`) are used to select parameters heuristically in various algorithms. Such heuristic use of approximate arithmetic does not affect correctness: when any error bounds depend on the parameters, the error bounds are evaluated separately using rigorous methods. At worst, flaws in the floating-point arithmetic on a particular machine could cause an algorithm to become inefficient due to inefficient parameters being selected.

Secondly, `double` arithmetic is used internally for some rigorous error bound calculations. To guarantee correctness, we make the following assumptions. With the stated exceptions, these should hold on all commonly used platforms.

- A `double` uses the standard IEEE 754 format (with a 53-bit significand, 11-bit exponent, encoding of infinities and NaNs, etc.)
- We assume that the compiler does not perform "unsafe" floating-point optimizations, such as reordering of operations. Unsafe optimizations are disabled by default in most modern C compilers, including GCC and Clang. The exception appears to be the Intel C++ compiler, which does some unsafe optimizations by default. These must be disabled by the user.
- We do not assume that floating-point operations are correctly rounded (a counterexample is the x87 FPU), or that rounding is done in any particular direction (the rounding mode may have been changed by the user). We assume that any floating-point operation is done with at most 1.1 ulp error.
- We do not assume that underflow or overflow behaves in a particular way (we only use doubles that fit in the regular exponent range, or explicit infinities).
- We do not use transcendental `libm` functions, since these can have errors of several ulps, and there is unfortunately no way to get guaranteed bounds. However, we do use functions such as `ldexp` and `sqrt`, which we assume to be correctly implemented.

2.4.6 Interface changes

Most of the core API should be stable at this point, and significant compatibility-breaking changes will be specified in the release notes.

In general, Arb does not distinguish between "private" and "public" parts of the API. The implementation is meant to be transparent by design. All methods are intended to be fully documented and tested (exceptions to this are mainly due to lack of time on part of the author). The user should use common sense to determine whether a function is concerned with implementation details, making it likely to change as the implementation changes in the future. The interface of *arb_add()* is probably not going to change in the next version, but *_arb_get_mpn_fixed_mod_pi4()* just might.

2.4.7 General note on correctness

Except where otherwise specified, Arb is designed to produce provably correct error bounds. The code has been written carefully, and the library is extensively tested. However, like any complex mathematical software, Arb is virtually certain to contain bugs, so the usual precautions are advised:

- Do sanity checks. For example, check that the result satisfies an expected mathematical relation, or compute the same result in two different ways, with different settings, and with different levels of precision. Arb's unit tests already do such checks, but they are not guaranteed to catch every possible bug, and they provide no protection against the user accidentally using the interface incorrectly.
- Compare results with other mathematical software.
- Read the source code to verify that it really does what it is supposed to do.

All bug reports are highly appreciated.

2.5 Example programs

The *examples* directory (https://github.com/fredrik-johansson/arb/tree/master/examples) contains several complete C programs, which are documented below. Running:

```
make examples
```

will compile the programs and place the binaries in `build/examples`.

2.5.1 pi.c

This program computes π to an accuracy of roughly n decimal digits by calling the *arb_const_pi()* function with a working precision of roughly $n \log_2(10)$ bits.

Sample output, computing π to one million digits:

```
> build/examples/pi 1000000
computing pi with a precision of 3321933 bits... cpu/wall(s): 0.58 0.586
virt/peak/res/peak(MB): 28.24 36.84 8.86 15.56
[3.14159265358979323846{...999959 digits...}42209010610577945815 +/- 3e-1000000]
```

The program prints an interval guaranteed to contain π, and where all displayed digits are correct up to an error of plus or minus one unit in the last place (see *arb_printn()*). By default, only the first and last few digits are printed. Pass 0 as a second argument to print all digits (or pass m to print $m + 1$ leading and m trailing digits, as above with the default $m = 20$).

2.5.2 hilbert_matrix.c

Given an input integer n, this program accurately computes the determinant of the n by n Hilbert matrix. Hilbert matrices are notoriously ill-conditioned: although the entries are close to unit magnitude, the determinant h_n decreases superexponentially (nearly as $1/4^{n^2}$) as a function of n. This program automatically doubles the working precision until the ball computed for h_n by *arb_mat_det()* does not contain zero.

Sample output:

```
> build/examples/hilbert_matrix 200
prec=20: 0 +/- 5.5777e-330
prec=40: 0 +/- 2.5785e-542
prec=80: 0 +/- 8.1169e-926
prec=160: 0 +/- 2.8538e-1924
prec=320: 0 +/- 6.3868e-4129
prec=640: 0 +/- 1.7529e-8826
prec=1280: 0 +/- 1.8545e-17758
prec=2560: 2.955454297e-23924 +/- 6.4586e-24044
success!
cpu/wall(s): 9.06 9.095
virt/peak/res/peak(MB): 55.52 55.52 35.50 35.50
```

2.5.3 keiper_li.c

Given an input integer n, this program rigorously computes numerical values of the Keiper-Li coefficients $\lambda_0, \ldots, \lambda_n$. The Keiper-Li coefficients have the property that $\lambda_n > 0$ for all $n > 0$ if and only if the Riemann hypothesis is true. This program was used for the record computations described in *[Joh2013]* (the paper describes the algorithm in some more detail).

The program takes the following parameters:

```
keiper_li n [-prec prec] [-threads num_threads] [-out out_file]
```

The program prints the first and last few coefficients. It can optionally write all the computed data to a file. The working precision defaults to a value that should give all the coefficients to a few digits of accuracy, but can optionally be set higher (or lower). On a multicore system, using several threads results in faster execution.

Sample output:

```
> build/examples/keiper_li 1000 -threads 2
zeta: cpu/wall(s): 0.4 0.244
virt/peak/res/peak(MB): 167.98 294.69 5.09 7.43
log: cpu/wall(s): 0.03 0.038
gamma: cpu/wall(s): 0.02 0.016
binomial transform: cpu/wall(s): 0.01 0.018
0: -0.693147180559945309417232121458176568075500134360260 +/- 6.5389e-347
1: 0.023095708966121033814310247906495291621932127152051 +/- 2.0924e-345
2: 0.046172867614023335192864243096033943387066108314123 +/- 1.674e-344
3: 0.069212973518108267930497348872601068994212026393 2 +/- 5.0219e-344
4: 0.092197619873060409647627872409439018065541673490213 +/- 2.0089e-343
5: 0.115108542892235490486221281098572766713491323035 96 +/- 1.0044e-342
6: 0.137927668713729882904167137003416663561389660786 54 +/- 6.0264e-342
7: 0.160637159652994212940402872573853662922824420461 63 +/- 2.1092e-341
8: 0.183219459643382579081939317747218598489980982734 32 +/- 8.4368e-341
9: 0.205657338709170461702893874213433047412365534100 44 +/- 7.5931e-340
10: 0.227933936319315774369030405736844533807483859427 38 +/- 7.5931e-339
991: 2.319661796161336792837389965699468256210143081334 1 +/- 2.461e-11
992: 2.320376623925488403534989651833255023316290971728 8 +/- 9.5363e-11
993: 2.321092061239733282811659116333262802034375592414 +/- 1.8495e-10
994: 2.321807354018846211025882612150387011274718888889 3 +/- 3.5907e-10
995: 2.322521739281518572692870295122531402377335815253 3 +/- 6.978e-10
996: 2.323234448581462387333322360941370391235828307128 1 +/- 1.3574e-09
997: 2.323944711488601452288954266758038203452650923247 5 +/- 2.6433e-09
998: 2.324651759103270080834414324035260514885686932220 9 +/- 5.1524e-09
999: 2.325354827586138211981257605206052698854499316210 1 +/- 1.0053e-08
1000: 2.326053161686466457406504694083223815804498204187 2 +/- 3.927e-08
virt/peak/res/peak(MB): 170.18 294.69 7.51 7.51
```

2.5.4 logistic.c

This program computes the n-th iterate of the logistic map defined by $x_{n+1} = rx_n(1 - x_n)$ where r and x_0 are given. It takes the following parameters:

```
logistic n [x_0] [r] [digits]
```

The inputs x_0, r and *digits* default to 0.5, 3.75 and 10 respectively. The computation is automatically restarted with doubled precision until the result is accurate to *digits* decimal digits.

Sample output:

```
> build/examples/logistic 10
Trying prec=64 bits...success!
cpu/wall(s): 0 0.001
x_10 = [0.6453672908 +/- 3.10e-11]

> build/examples/logistic 100
Trying prec=64 bits...ran out of accuracy at step 18
Trying prec=128 bits...ran out of accuracy at step 53
Trying prec=256 bits...success!
cpu/wall(s): 0 0
x_100 = [0.8882939923 +/- 1.60e-11]
```

```
> build/examples/logistic 10000
Trying prec=64 bits...ran out of accuracy at step 18
Trying prec=128 bits...ran out of accuracy at step 53
Trying prec=256 bits...ran out of accuracy at step 121
Trying prec=512 bits...ran out of accuracy at step 256
Trying prec=1024 bits...ran out of accuracy at step 525
Trying prec=2048 bits...ran out of accuracy at step 1063
Trying prec=4096 bits...ran out of accuracy at step 2139
Trying prec=8192 bits...ran out of accuracy at step 4288
Trying prec=16384 bits...ran out of accuracy at step 8584
Trying prec=32768 bits...success!
cpu/wall(s): 0.859 0.858
x_10000 = [0.8242048008 +/- 4.35e-11]

> build/examples/logistic 1234 0.1 3.99 30
Trying prec=64 bits...ran out of accuracy at step 0
Trying prec=128 bits...ran out of accuracy at step 10
Trying prec=256 bits...ran out of accuracy at step 76
Trying prec=512 bits...ran out of accuracy at step 205
Trying prec=1024 bits...ran out of accuracy at step 461
Trying prec=2048 bits...ran out of accuracy at step 974
Trying prec=4096 bits...success!
cpu/wall(s): 0.009 0.009
x_1234 = [0.256445391958651410579677945635 +/- 3.92e-31]
```

2.5.5 real_roots.c

This program isolates the roots of a function on the interval (a, b) (where a and b are input as double-precision literals) using the routines in the *arb_calc* module. The program takes the following arguments:

```
real_roots function a b [-refine d] [-verbose] [-maxdepth n] [-maxeval n] [-maxfound n] [-prec n]
```

The following functions (specified by an integer code) are implemented:

- 0 - $Z(x)$ (Riemann-Siegel Z-function)
- 1 - $\sin(x)$
- 2 - $\sin(x^2)$
- 3 - $\sin(1/x)$
- 4 - $\mathrm{Ai}(x)$ (Airy function)
- 5 - $\mathrm{Ai}'(x)$ (Airy function)
- 6 - $\mathrm{Bi}(x)$ (Airy function)
- 7 - $\mathrm{Bi}'(x)$ (Airy function)

The following options are available:

- `-refine d`: If provided, after isolating the roots, attempt to refine the roots to d digits of accuracy using a few bisection steps followed by Newton's method with adaptive precision, and then print them.
- `-verbose`: Print more information.
- `-maxdepth n`: Stop searching after n recursive subdivisions.
- `-maxeval n`: Stop searching after approximately n function evaluations (the actual number evaluations will be a small multiple of this).
- `-maxfound n`: Stop searching after having found n isolated roots.
- `-prec n`: Working precision to use for the root isolation.

With *function* 0, the program isolates roots of the Riemann zeta function on the critical line, and guarantees that no roots are missed (there are more efficient ways to do this, but it is a nice example):

```
> build/examples/real_roots 0 0.0 50.0 -verbose
interval: [0, 50]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
found isolated root in: [14.111328125, 14.16015625]
found isolated root in: [20.99609375, 21.044921875]
found isolated root in: [25, 25.048828125]
found isolated root in: [30.419921875, 30.4443359375]
found isolated root in: [32.91015625, 32.958984375]
found isolated root in: [37.548828125, 37.59765625]
found isolated root in: [40.91796875, 40.966796875]
found isolated root in: [43.310546875, 43.3349609375]
found isolated root in: [47.998046875, 48.0224609375]
found isolated root in: [49.755859375, 49.7802734375]
---------------------------------------------------------------
Found roots: 10
Subintervals possibly containing undetected roots: 0
Function evaluations: 3058
cpu/wall(s): 0.202 0.202
virt/peak/res/peak(MB): 26.12 26.14 2.76 2.76
```

Find just one root and refine it to approximately 75 digits:

```
> build/examples/real_roots 0 0.0 50.0 -maxfound 1 -refine 75
interval: [0, 50]
maxdepth = 30, maxeval = 100000, maxfound = 1, low_prec = 30
refined root (0/8):
[14.134725141734693790457251983562470270784257115699243175685567460149963429809 +/- 2.57e-76]

---------------------------------------------------------------
Found roots: 1
Subintervals possibly containing undetected roots: 7
Function evaluations: 761
cpu/wall(s): 0.055 0.056
virt/peak/res/peak(MB): 26.12 26.14 2.75 2.75
```

Find the first few roots of an Airy function and refine them to 50 digits each:

```
> build/examples/real_roots 4 -10 0 -refine 50
interval: [-10, 0]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
refined root (0/6):
[-9.022650853340980380158190839880089256524677535156083 +/- 4.85e-52]

refined root (1/6):
[-7.944133587120853123138280555798268532140674396972215 +/- 1.92e-52]

refined root (2/6):
[-6.786708090071758998780246384496176966053882477393494 +/- 3.84e-52]

refined root (3/6):
[-5.520559828095551059129855512931293573797214280617525 +/- 1.05e-52]

refined root (4/6):
[-4.087949444130970616636988701457391060224764699108530 +/- 2.46e-52]

refined root (5/6):
[-2.338107410459767038489197252446735440638540145672388 +/- 1.48e-52]

---------------------------------------------------------------
Found roots: 6
Subintervals possibly containing undetected roots: 0
```

```
Function evaluations: 200
cpu/wall(s): 0.003 0.003
virt/peak/res/peak(MB): 26.12 26.14 2.24 2.24
```

Find roots of $\sin(x^2)$ on $(0, 100)$. The algorithm cannot isolate the root at $x = 0$ (it is at the endpoint of the interval, and in any case a root of multiplicity higher than one). The failure is reported:

```
> build/examples/real_roots 2 0 100
interval: [0, 100]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
---------------------------------------------------------------
Found roots: 3183
Subintervals possibly containing undetected roots: 1
Function evaluations: 34058
cpu/wall(s): 0.032 0.032
virt/peak/res/peak(MB): 26.32 26.37 2.04 2.04
```

This does not miss any roots:

```
> build/examples/real_roots 2 1 100
interval: [1, 100]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
---------------------------------------------------------------
Found roots: 3183
Subintervals possibly containing undetected roots: 0
Function evaluations: 34039
cpu/wall(s): 0.023 0.023
virt/peak/res/peak(MB): 26.32 26.37 2.01 2.01
```

Looking for roots of $\sin(1/x)$ on $(0, 1)$, the algorithm finds many roots, but will never find all of them since there are infinitely many:

```
> build/examples/real_roots 3 0.0 1.0
interval: [0, 1]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
---------------------------------------------------------------
Found roots: 10198
Subintervals possibly containing undetected roots: 24695
Function evaluations: 202587
cpu/wall(s): 0.171 0.171
virt/peak/res/peak(MB): 28.39 30.38 4.05 4.05
```

Remark: the program always computes rigorous containing intervals for the roots, but the accuracy after refinement could be less than d digits.

2.5.6 poly_roots.c

This program finds the complex roots of an integer polynomial by calling *acb_poly_find_roots()* with increasing precision until the roots certainly have been isolated. The program takes the following arguments:

```
poly_roots [-refine d] [-print d] <poly>

Isolates all the complex roots of a polynomial with
integer coefficients. For convergence, the input polynomial
is required to be squarefree.

If -refine d is passed, the roots are refined to an absolute
tolerance better than 10^(-d). By default, the roots are only
computed to sufficient accuracy to isolate them.
The refinement is not currently done efficiently.
```

```
If -print d is passed, the computed roots are printed to
d decimals. By default, the roots are not printed.

The polynomial can be specified by passing the following as <poly>:

a <n>          Easy polynomial 1 + 2x + ... + (n+1)x^n
t <n>          Chebyshev polynomial T_n
u <n>          Chebyshev polynomial U_n
p <n>          Legendre polynomial P_n
c <n>          Cyclotomic polynomial Phi_n
s <n>          Swinnerton-Dyer polynomial S_n
b <n>          Bernoulli polynomial B_n
w <n>          Wilkinson polynomial W_n
e <n>          Taylor series of exp(x) truncated to degree n
m <n> <m>      The Mignotte-like polynomial x^n + (100x+1)^m, n > m
c0 c1 ... cn   c0 + c1 x + ... + cn x^n where all c:s are specified integers
```

This finds the roots of the Wilkinson polynomial with roots at the positive integers 1, 2, ..., 100:

```
> build/examples/poly_roots -print 15 w 100
prec=53: 0 isolated roots | cpu/wall(s): 0.42 0.426
prec=106: 0 isolated roots | cpu/wall(s): 1.37 1.368
prec=212: 0 isolated roots | cpu/wall(s): 1.48 1.485
prec=424: 100 isolated roots | cpu/wall(s): 0.61 0.611
done!
(1 + 1.7285178043492e-125j)  +/-  (7.2e-122, 7.2e-122j)
(2 + 5.1605530263601e-122j)  +/-  (3.77e-118, 3.77e-118j)
(3 + -2.58115555871665e-118j)  +/-  (5.72e-115, 5.72e-115j)
(4 + 1.02141628524271e-115j)  +/-  (4.38e-112, 4.38e-112j)
(5 + 1.61326834094948e-113j)  +/-  (2.6e-109, 2.6e-109j)
    ...
(95 + 4.15294196875447e-62j)  +/-  (6.66e-59, 6.66e-59j)
(96 + 3.54502401922667e-64j)  +/-  (7.37e-60, 7.37e-60j)
(97 + -1.67755595325625e-65j)  +/-  (6.4e-61, 6.4e-61j)
(98 + 2.04638822325299e-65j)  +/-  (4e-62, 4e-62j)
(99 + -2.73425468028238e-66j)  +/-  (1.71e-63, 1.71e-63j)
(100 + -1.00950111302288e-68j)  +/-  (3.24e-65, 3.24e-65j)
cpu/wall(s): 3.88 3.893
```

This finds the roots of a Bernoulli polynomial which has both real and complex roots. Note that the program does not attempt to determine that the imaginary parts of the real roots really are zero (this could be done by verifying sign changes):

```
> build/examples/poly_roots -refine 100 -print 20 b 16
prec=53: 16 isolated roots | cpu/wall(s): 0 0.007
prec=106: 16 isolated roots | cpu/wall(s): 0 0.004
prec=212: 16 isolated roots | cpu/wall(s): 0 0.004
prec=424: 16 isolated roots | cpu/wall(s): 0 0.004
done!
(-0.94308706466055783383 + -5.512272663168484603e-128j)  +/-  (2.2e-125, 2.2e-125j)
(-0.75534059252067985752 + 1.937401283040249068e-128j)  +/-  (1.09e-125, 1.09e-125j)
(-0.24999757119077421009 + -4.534792442246038692e-130j)  +/-  (3.6e-127, 3.6e-127j)
(0.24999757152512726002 + 4.2191300761823281708e-129j)  +/-  (4.98e-127, 4.98e-127j)
(0.75000242847487273998 + 9.0360649917413170142e-128j)  +/-  (8.88e-126, 8.88e-126j)
(1.2499975711907742101 + 7.8804123808107088267e-127j)  +/-  (2.66e-124, 2.66e-124j)
(1.7553405925206798575 + 5.432465269253967768e-126j)  +/-  (6.23e-123, 6.23e-123j)
(1.9430870646605578338 + 3.3035377342500953239e-125j)  +/-  (7.05e-123, 7.05e-123j)
(-0.99509334829256233279 + 0.44547958157103608805j)  +/-  (5.5e-125, 5.5e-125j)
(-0.99509334829256233279 + -0.44547958157103608805j)  +/-  (5.46e-125, 5.46e-125j)
(1.9950933482925623328 + 0.44547958157103608805j)  +/-  (1.44e-122, 1.44e-122j)
(1.9950933482925623328 + -0.44547958157103608805j)  +/-  (1.43e-122, 1.43e-122j)
(-0.92177327714429290564 + -1.0954360955079385542j)  +/-  (9.31e-125, 9.31e-125j)
(-0.92177327714429290564 + 1.0954360955079385542j)  +/-  (1.02e-124, 1.02e-124j)
```

```
(1.9217732771442929056 + 1.0954360955079385542j)  +/-  (9.15e-123, 9.15e-123j)
(1.9217732771442929056 + -1.0954360955079385542j)  +/-  (8.12e-123, 8.12e-123j)
cpu/wall(s): 0.02 0.02
```

2.5.7 complex_plot.c

This program plots one of the predefined functions over a complex interval $[x_a, x_b]+[y_a, y_b]i$ using domain coloring, at a resolution of xn times yn pixels.

The program takes the parameters:

```
complex_plot [-range xa xb ya yb] [-size xn yn] <func>
```

Defaults parameters are $[-10, 10] + [-10, 10]i$ and $xn = yn = 512$.

The output is written to `arbplot.ppm`. If you have ImageMagick, run `convert arbplot.ppm arbplot.png` to get a PNG.

Function codes `<func>` are:

- `gamma` - Gamma function
- `digamma` - Digamma function
- `lgamma` - Logarithmic gamma function
- `zeta` - Riemann zeta function
- `erf` - Error function
- `ai` - Airy function Ai
- `bi` - Airy function Bi
- `besselj` - Bessel function J_0
- `bessely` - Bessel function Y_0
- `besseli` - Bessel function I_0
- `besselk` - Bessel function K_0
- `modj` - Modular j-function
- `modeta` - Dedekind eta function
- `barnesg` - Barnes G-function
- `agm` - Arithmetic geometric mean

The function is just sampled at point values; no attempt is made to resolve small features by adaptive subsampling.

For example, the following plots the Riemann zeta function around a portion of the critical strip with imaginary part between 100 and 140:

```
> build/examples/complex_plot zeta -range -10 10 100 140 -size 256 512
```

CHAPTER

THREE

FLOATING-POINT NUMBERS

The radius and midpoint of a ball are represented using two specialized floating-point types.

3.1 mag.h – fixed-precision unsigned floating-point numbers for bounds

The *mag_t* type is an unsigned floating-point type with a fixed-precision mantissa (30 bits) and an arbitrary-precision exponent (represented as an *fmpz_t*), suited for representing and rigorously manipulating magnitude bounds efficiently. Operations always produce a strict upper or lower bound, but for performance reasons, no attempt is made to compute the best possible bound (in general, a result may a few ulps larger/smaller than the optimal value). The special values zero and positive infinity are supported (but not NaN). Applications requiring more flexibility (such as correct rounding, or higher precision) should use the *arf_t* type instead.

3.1.1 Types, macros and constants

mag_struct
A *mag_struct* holds a mantissa and an exponent. Special values are encoded by the mantissa being set to zero.

mag_t
A *mag_t* is defined as an array of length one of type *mag_struct*, permitting a *mag_t* to be passed by reference.

3.1.2 Memory management

void **mag_init**(*mag_t* *x*)
Initializes the variable *x* for use. Its value is set to zero.

void **mag_clear**(*mag_t* *x*)
Clears the variable *x*, freeing or recycling its allocated memory.

void **mag_init_set**(*mag_t* *x*, const *mag_t* *y*)
Initializes *x* and sets it to the value of *y*.

void **mag_swap**(*mag_t* *x*, *mag_t* *y*)
Swaps *x* and *y* efficiently.

void **mag_set**(*mag_t* *x*, const *mag_t* *y*)
Sets *x* to the value of *y*.

mag_ptr **_mag_vec_init**(*slong* *n*)
Allocates a vector of length *n*. All entries are set to zero.

void **_mag_vec_clear**(mag_ptr *v*, *slong* *n*)
Clears a vector of length *n*.

3.1.3 Special values

void **mag_zero**(*mag_t* *x*)
Sets x to zero.

void **mag_one**(*mag_t* *x*)
Sets x to one.

void **mag_inf**(*mag_t* *x*)
Sets x to positive infinity.

int **mag_is_special**(const *mag_t* *x*)
Returns nonzero iff x is zero or positive infinity.

int **mag_is_zero**(const *mag_t* *x*)
Returns nonzero iff x is zero.

int **mag_is_inf**(const *mag_t* *x*)
Returns nonzero iff x is positive infinity.

int **mag_is_finite**(const *mag_t* *x*)
Returns nonzero iff x is not positive infinity (since there is no NaN value, this function is exactly the negation of *mag_is_inf()*).

3.1.4 Comparisons

int **mag_equal**(const *mag_t* *x*, const *mag_t* *y*)
Returns nonzero iff x and y have the same value.

int **mag_cmp**(const *mag_t* *x*, const *mag_t* *y*)
Returns negative, zero, or positive, depending on whether x is smaller, equal, or larger than y.

int **mag_cmp_2exp_si**(const *mag_t* *x*, *slong* *y*)
Returns negative, zero, or positive, depending on whether x is smaller, equal, or larger than 2^y.

void **mag_min**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)

void **mag_max**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)
Sets z respectively to the smaller or the larger of x and y.

3.1.5 Input and output

void **mag_print**(const *mag_t* *x*)
Prints x to standard output.

void **mag_fprint**(FILE * *file*, const *mag_t* *x*)
Prints x to the stream *file*.

3.1.6 Random generation

void **mag_randtest**(*mag_t* *x*, flint_rand_t *state*, *slong* *expbits*)
Sets x to a random finite value, with an exponent up to *expbits* bits large.

void **mag_randtest_special**(*mag_t* *x*, flint_rand_t *state*, *slong* *expbits*)
Like *mag_randtest()*, but also sometimes sets x to infinity.

3.1.7 Conversions

void **mag_set_d**(*mag_t* *y*, double *x*)

void **mag_set_fmpr**(*mag_t* *y*, const *fmpr_t* *x*)

void **mag_set_ui**(*mag_t* *y*, *ulong* *x*)

void **mag_set_fmpz**(*mag_t* *y*, const *fmpz_t* *x*)
Sets y to an upper bound for $|x|$.

void **mag_set_d_2exp_fmpz**(*mag_t* *z*, double *x*, const *fmpz_t* *y*)

void **mag_set_fmpz_2exp_fmpz**(*mag_t* *z*, const *fmpz_t* *x*, const *fmpz_t* *y*)

void **mag_set_ui_2exp_si**(*mag_t* *z*, *ulong* *x*, *slong* *y*)
Sets z to an upper bound for $|x| \times 2^y$.

void **mag_get_fmpr**(*fmpr_t* *y*, const *mag_t* *x*)
Sets y exactly to x.

void **mag_get_fmpq**(*fmpq_t* *y*, const *mag_t* *x*)
Sets y exactly to x. Assumes that no overflow occurs.

double **mag_get_d**(const *mag_t* *x*)
Returns a *double* giving an upper bound for x.

void **mag_set_ui_lower**(*mag_t* *z*, *ulong* *x*)

void **mag_set_fmpz_lower**(*mag_t* *z*, const *fmpz_t* *x*)
Sets y to a lower bound for $|x|$.

void **mag_set_fmpz_2exp_fmpz_lower**(*mag_t* *z*, const *fmpz_t* *x*, const *fmpz_t* *y*)
Sets z to a lower bound for $|x| \times 2^y$.

3.1.8 Arithmetic

void **mag_mul_2exp_si**(*mag_t* *z*, const *mag_t* *x*, *slong* *y*)

void **mag_mul_2exp_fmpz**(*mag_t* *z*, const *mag_t* *x*, const *fmpz_t* *y*)
Sets z to $x \times 2^y$. This operation is exact.

void **mag_mul**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)

void **mag_mul_ui**(*mag_t* *z*, const *mag_t* *x*, *ulong* *y*)

void **mag_mul_fmpz**(*mag_t* *z*, const *mag_t* *x*, const *fmpz_t* *y*)
Sets z to an upper bound for xy.

void **mag_add**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)

void **mag_add_ui**(*mag_t* *z*, const *mag_t* *x*, *ulong* *y*)
Sets z to an upper bound for $x + y$.

void **mag_addmul**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)
Sets z to an upper bound for $z + xy$.

void **mag_add_2exp_fmpz**(*mag_t* *z*, const *mag_t* *x*, const *fmpz_t* *e*)
Sets z to an upper bound for $x + 2^e$.

void **mag_add_ui_2exp_si**(*mag_t* *z*, const *mag_t* *x*, *ulong* *y*, *slong* *e*)
Sets z to an upper bound for $x + y2^e$.

void **mag_div**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)

void **mag_div_ui**(*mag_t* *z*, const *mag_t* *x*, *ulong* *y*)

void **mag_div_fmpz**(*mag_t* *z*, const *mag_t* *x*, const *fmpz_t* *y*)
Sets z to an upper bound for x/y.

void **mag_mul_lower**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)

void **mag_mul_ui_lower**(*mag_t* *z*, const *mag_t* *x*, *ulong* *y*)

void **mag_mul_fmpz_lower**(*mag_t* *z*, const *mag_t* *x*, const *fmpz_t* *y*)
Sets z to a lower bound for xy.

void **mag_add_lower**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)
Sets z to a lower bound for $x + y$.

void **mag_sub**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)
Sets z to an upper bound for $\max(x - y, 0)$.

void **mag_sub_lower**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)
Sets z to a lower bound for $\max(x - y, 0)$.

3.1.9 Fast, unsafe arithmetic

The following methods assume that all inputs are finite and that all exponents (in all inputs as well as the final result) fit as *fmpz* inline values. They also assume that the output variables do not have promoted exponents, as they will be overwritten directly (thus leaking memory).

void **mag_fast_init_set**(*mag_t* *x*, const *mag_t* *y*)
Initialises x and sets it to the value of y.

void **mag_fast_zero**(*mag_t* *x*)
Sets x to zero.

int **mag_fast_is_zero**(const *mag_t* *x*)
Returns nonzero iff x to zero.

void **mag_fast_mul**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)
Sets z to an upper bound for xy.

void **mag_fast_addmul**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)
Sets z to an upper bound for $z + xy$.

void **mag_fast_add_2exp_si**(*mag_t* *z*, const *mag_t* *x*, *slong* *e*)
Sets z to an upper bound for $x + 2^e$.

void **mag_fast_mul_2exp_si**(*mag_t* *z*, const *mag_t* *x*, *slong* *e*)
Sets z to an upper bound for $x2^e$.

3.1.10 Powers and logarithms

void **mag_pow_ui**(*mag_t* *z*, const *mag_t* *x*, *ulong* *e*)

void **mag_pow_fmpz**(*mag_t* *z*, const *mag_t* *x*, const *fmpz_t* *e*)
Sets z to an upper bound for x^e. Requires $e \geq 0$.

void **mag_pow_ui_lower**(*mag_t* *z*, const *mag_t* *x*, *ulong* *e*)
Sets z to a lower bound for x^e.

void **mag_sqrt**(*mag_t* *z*, const *mag_t* *x*)
Sets z to an upper bound for $\sqrt{x}$.

void **mag_rsqrt**(*mag_t* *z*, const *mag_t* *x*)
Sets z to an upper bound for $1/\sqrt{x}$.

void **mag_hypot**(*mag_t* *z*, const *mag_t* *x*, const *mag_t* *y*)
Sets z to an upper bound for $\sqrt{x^2 + y^2}$.

void **mag_root**(*mag_t* *z*, const *mag_t* *x*, *ulong* *n*)
Sets z to an upper bound for $x^{1/n}$. We evaluate $\exp(\log(1 + 2^{kn}x)/n)2^{-k}$, where k is chosen so that $2^{kn}x \approx 2^{30}$.

void **mag_log1p**(*mag_t* *z*, const *mag_t* *x*)
Sets z to an upper bound for $\log(1 + x)$. The bound is computed accurately for small x.

void **mag_log_ui**(*mag_t* *z*, *ulong* *n*)
Sets z to an upper bound for $\log(n)$.

void **mag_exp**(*mag_t z*, const *mag_t x*)
Sets z to an upper bound for $\exp(x)$.

void **mag_expinv**(*mag_t z*, const *mag_t x*)
Sets z to an upper bound for $\exp(-x)$. As currently implemented, the bound is computed crudely by rounding x down to an integer before approximating the exponential.

void **mag_expm1**(*mag_t z*, const *mag_t x*)
Sets z to an upper bound for $\exp(x) - 1$. The bound is computed accurately for small x.

void **mag_exp_tail**(*mag_t z*, const *mag_t x*, *ulong N*)
Sets z to an upper bound for $\sum_{k=N}^{\infty} x^k/k!$.

void **mag_binpow_uiui**(*mag_t z*, *ulong m*, *ulong n*)
Sets z to an upper bound for $(1 + 1/m)^n$.

void **mag_geom_series**(*mag_t res*, const *mag_t x*, *ulong N*)
Sets *res* to an upper bound for $\sum_{k=N}^{\infty} x^k$.

3.1.11 Special functions

void **mag_const_pi**(*mag_t z*)
Sets z to an upper bound for π.

void **mag_fac_ui**(*mag_t z*, *ulong n*)
Sets z to an upper bound for $n!$.

void **mag_rfac_ui**(*mag_t z*, *ulong n*)
Sets z to an upper bound for $1/n!$.

void **mag_bernoulli_div_fac_ui**(*mag_t z*, *ulong n*)
Sets z to an upper bound for $|B_n|/n!$ where B_n denotes a Bernoulli number.

void **mag_polylog_tail**(*mag_t u*, const *mag_t z*, *slong s*, *ulong d*, *ulong N*)
Sets u to an upper bound for

$$\sum_{k=N}^{\infty} \frac{z^k \log^d(k)}{k^s}.$$

Note: in applications where s in this formula may be real or complex, the user can simply substitute any convenient integer s' such that $s' \le \operatorname{Re}(s)$.

Denote the terms by $T(k)$. We pick a nonincreasing function $U(k)$ such that

$$\frac{T(k+1)}{T(k)} = z \left(\frac{k}{k+1}\right)^s \left(\frac{\log(k+1)}{\log(k)}\right)^d \le U(k).$$

Then, as soon as $U(N) < 1$,

$$\sum_{k=N}^{\infty} T(k) \le T(N) \sum_{k=0}^{\infty} U(N)^k = \frac{T(N)}{1 - U(N)}.$$

In particular, we take

$$U(k) = z\, B(k, \max(0, -s))\, B(k \log(k), d)$$

where $B(m, n) = (1 + 1/m)^n$. This follows from the bounds

$$\left(\frac{k}{k+1}\right)^s \le \begin{cases} 1 & \text{if } s \ge 0 \\ (1 + 1/k)^{-s} & \text{if } s < 0. \end{cases}$$

and

$$\left(\frac{\log(k+1)}{\log(k)}\right)^d \le \left(1 + \frac{1}{k \log(k)}\right)^d.$$

void **mag_hurwitz_zeta_uiui**(*mag_t* *res*, *ulong* *s*, *ulong* *a*)
Sets *res* to an upper bound for $\zeta(s,a) = \sum_{k=0}^{\infty}(k+a)^{-s}$. We use the formula

$$\zeta(s,a) \le \frac{1}{a^s} + \frac{1}{(s-1)a^{s-1}}$$

which is obtained by estimating the sum by an integral. If $s \le 1$ or $a = 0$, the bound is infinite.

3.2 arf.h – arbitrary-precision floating-point numbers

A variable of type *arf_t* holds an arbitrary-precision binary floating-point number, i.e. a rational number of the form $x \times 2^y$ where $x, y \in \mathbb{Z}$ and x is odd; or one of the special values zero, plus infinity, minus infinity, or NaN (not-a-number).

The *exponent* of a finite and nonzero floating-point number can be defined in different ways: for example, as the component y above, or as the unique integer e such that $x \times 2^y = m \times 2^e$ where $1/2 \le |m| < 1$. The internal representation of an *arf_t* stores the exponent in the latter format.

The conventions for special values largely follow those of the IEEE floating-point standard. At the moment, there is no support for negative zero, unsigned infinity, or a NaN with a payload, though some of these might be added in the future.

Except where otherwise noted, the output of an operation is the floating-point number obtained by taking the inputs as exact numbers, in principle carrying out the operation exactly, and rounding the resulting real number to the nearest representable floating-point number whose mantissa has at most the specified number of bits, in the specified direction of rounding. Some operations are always or optionally done exactly.

The *arf_t* type is almost identical semantically to the legacy *fmpr_t* type, but uses a more efficient internal representation. The most significant differences that the user has to be aware of are:

- The mantissa is no longer represented as a FLINT `fmpz`, and the internal exponent points to the top of the binary expansion of the mantissa instead of of the bottom. Code designed to manipulate components of an *fmpr_t* directly can be ported to the *arf_t* type by making use of *arf_get_fmpz_2exp()* and *arf_set_fmpz_2exp()*.
- Some *arf_t* functions return an `int` indicating whether a result is inexact, whereas the corresponding *fmpr_t* functions return an *slong* encoding the relative exponent of the error.

3.2.1 Types, macros and constants

arf_struct

arf_t
An *arf_struct* contains four words: an `fmpz` exponent (*exp*), a *size* field tracking the number of limbs used (one bit of this field is also used for the sign of the number), and two more words. The last two words hold the value directly if there are at most two limbs, and otherwise contain one *alloc* field (tracking the total number of allocated limbs, not all of which might be used) and a pointer to the actual limbs. Thus, up to 128 bits on a 64-bit machine and 64 bits on a 32-bit machine, no space outside of the *arf_struct* is used.

An *arf_t* is defined as an array of length one of type *arf_struct*, permitting an *arf_t* to be passed by reference.

arf_rnd_t
Specifies the rounding mode for the result of an approximate operation.

ARF_RND_DOWN
Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards zero.

`ARF_RND_UP`
: Specifies that the result of an operation should be rounded to the nearest representable number in the direction away from zero.

`ARF_RND_FLOOR`
: Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards minus infinity.

`ARF_RND_CEIL`
: Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards plus infinity.

`ARF_RND_NEAR`
: Specifies that the result of an operation should be rounded to the nearest representable number, rounding to even if there is a tie between two values.

`ARF_PREC_EXACT`
: If passed as the precision parameter to a function, indicates that no rounding is to be performed. **Warning**: use of this value is unsafe in general. It must only be passed as input under the following two conditions:

  - The operation in question can inherently be viewed as an exact operation in $\mathbb{Z}[\frac{1}{2}]$ for all possible inputs, provided that the precision is large enough. Examples include addition, multiplication, conversion from integer types to arbitrary-precision floating-point types, and evaluation of some integer-valued functions.
  - The exact result of the operation will certainly fit in memory. Note that, for example, adding two numbers whose exponents are far apart can easily produce an exact result that is far too large to store in memory.

  The typical use case is to work with small integer values, double precision constants, and the like. It is also useful when writing test code. If in doubt, simply try with some convenient high precision instead of using this special value, and check that the result is exact.

3.2.2 Memory management

void **arf_init**(*arf_t* *x*)
: Initializes the variable x for use. Its value is set to zero.

void **arf_clear**(*arf_t* *x*)
: Clears the variable x, freeing or recycling its allocated memory.

3.2.3 Special values

void **arf_zero**(*arf_t* *x*)

void **arf_one**(*arf_t* *x*)

void **arf_pos_inf**(*arf_t* *x*)

void **arf_neg_inf**(*arf_t* *x*)

void **arf_nan**(*arf_t* *x*)
: Sets x respectively to 0, 1, $+\infty$, $-\infty$, NaN.

int **arf_is_zero**(const *arf_t* *x*)

int **arf_is_one**(const *arf_t* *x*)

int **arf_is_pos_inf**(const *arf_t* *x*)

int **arf_is_neg_inf**(const *arf_t* *x*)

int **arf_is_nan**(const *arf_t* *x*)
: Returns nonzero iff x respectively equals 0, 1, $+\infty$, $-\infty$, NaN.

int **arf_is_inf**(const *arf_t x*)
Returns nonzero iff x equals either $+\infty$ or $-\infty$.

int **arf_is_normal**(const *arf_t x*)
Returns nonzero iff x is a finite, nonzero floating-point value, i.e. not one of the special values 0, $+\infty$, $-\infty$, NaN.

int **arf_is_special**(const *arf_t x*)
Returns nonzero iff x is one of the special values 0, $+\infty$, $-\infty$, NaN, i.e. not a finite, nonzero floating-point value.

int **arf_is_finite**(*arf_t x*)
Returns nonzero iff x is a finite floating-point value, i.e. not one of the values $+\infty$, $-\infty$, NaN. (Note that this is not equivalent to the negation of *arf_is_inf()*.)

3.2.4 Assignment, rounding and conversions

void **arf_set**(*arf_t y*, const *arf_t x*)

void **arf_set_mpz**(*arf_t y*, const mpz_t *x*)

void **arf_set_fmpz**(*arf_t y*, const *fmpz_t x*)

void **arf_set_ui**(*arf_t y*, *ulong x*)

void **arf_set_si**(*arf_t y*, *slong x*)

void **arf_set_mpfr**(*arf_t y*, const mpfr_t *x*)

void **arf_set_fmpr**(*arf_t y*, const *fmpr_t x*)

void **arf_set_d**(*arf_t y*, double *x*)
Sets y exactly to x.

void **arf_swap**(*arf_t y*, *arf_t x*)
Swaps y and x efficiently.

void **arf_init_set_ui**(*arf_t y*, *ulong x*)

void **arf_init_set_si**(*arf_t y*, *slong x*)
Initialises y and sets it to x in a single operation.

int **arf_set_round**(*arf_t y*, const *arf_t x*, *slong prec*, *arf_rnd_t rnd*)

int **arf_set_round_si**(*arf_t x*, *slong v*, *slong prec*, *arf_rnd_t rnd*)

int **arf_set_round_ui**(*arf_t x*, *ulong v*, *slong prec*, *arf_rnd_t rnd*)

int **arf_set_round_mpz**(*arf_t y*, const mpz_t *x*, *slong prec*, *arf_rnd_t rnd*)

int **arf_set_round_fmpz**(*arf_t y*, const *fmpz_t x*, *slong prec*, *arf_rnd_t rnd*)
Sets y to x, rounded to *prec* bits in the direction specified by *rnd*.

void **arf_set_si_2exp_si**(*arf_t y*, *slong m*, *slong e*)

void **arf_set_ui_2exp_si**(*arf_t y*, *ulong m*, *slong e*)

void **arf_set_fmpz_2exp**(*arf_t y*, const *fmpz_t m*, const *fmpz_t e*)
Sets y to $m \times 2^e$.

int **arf_set_round_fmpz_2exp**(*arf_t y*, const *fmpz_t x*, const *fmpz_t e*, *slong prec*, *arf_rnd_t rnd*)
Sets y to $x \times 2^e$, rounded to *prec* bits in the direction specified by *rnd*.

void **arf_get_fmpz_2exp**(*fmpz_t m*, *fmpz_t e*, const *arf_t x*)
Sets m and e to the unique integers such that $x = m \times 2^e$ and m is odd, provided that x is a nonzero finite fraction. If x is zero, both m and e are set to zero. If x is infinite or NaN, the result is undefined.

void **arf_frexp**(*arf_t* *m*, *fmpz_t* *e*, const *arf_t* *x*)
Writes x as $m \times 2^e$, where $1/2 \leq |m| < 1$ if x is a normal value. If x is a special value, copies this to m and sets e to zero. Note: for the inverse operation (*ldexp*), use *arf_mul_2exp_fmpz()*.

double **arf_get_d**(const *arf_t* *x*, *arf_rnd_t* *rnd*)
Returns x rounded to a double in the direction specified by *rnd*. This method rounds correctly when overflowing or underflowing the double exponent range (this was not the case in an earlier version).

void **arf_get_fmpr**(*fmpr_t* *y*, const *arf_t* *x*)
Sets y exactly to x.

int **arf_get_mpfr**(mpfr_t *y*, const *arf_t* *x*, mpfr_rnd_t *rnd*)
Sets the MPFR variable y to the value of x. If the precision of x is too small to allow y to be represented exactly, it is rounded in the specified MPFR rounding mode. The return value (-1, 0 or 1) indicates the direction of rounding, following the convention of the MPFR library.

int **arf_get_fmpz**(*fmpz_t* *z*, const *arf_t* *x*, *arf_rnd_t* *rnd*)
Sets z to x rounded to the nearest integer in the direction specified by *rnd*. If rnd is *ARF_RND_NEAR*, rounds to the nearest even integer in case of a tie. Returns inexact (beware: accordingly returns whether x is *not* an integer).

This method aborts if x is infinite or NaN, or if the exponent of x is so large that allocating memory for the result fails.

Warning: this method will allocate a huge amount of memory to store the result if the exponent of x is huge. Memory allocation could succeed even if the required space is far larger than the physical memory available on the machine, resulting in swapping. It is recommended to check that x is within a reasonable range before calling this method.

slong **arf_get_si**(const *arf_t* *x*, *arf_rnd_t* *rnd*)
Returns x rounded to the nearest integer in the direction specified by *rnd*. If *rnd* is *ARF_RND_NEAR*, rounds to the nearest even integer in case of a tie. Aborts if x is infinite, NaN, or the value is too large to fit in a slong.

int **arf_get_fmpz_fixed_fmpz**(*fmpz_t* *y*, const *arf_t* *x*, const *fmpz_t* *e*)

int **arf_get_fmpz_fixed_si**(*fmpz_t* *y*, const *arf_t* *x*, *slong* *e*)
Converts x to a mantissa with predetermined exponent, i.e. computes an integer y such that $y \times 2^e \approx x$, truncating if necessary. Returns 0 if exact and 1 if truncation occurred.

The warnings for *arf_get_fmpz()* apply.

void **arf_floor**(*arf_t* *y*, const *arf_t* *x*)

void **arf_ceil**(*arf_t* *y*, const *arf_t* *x*)
Sets y to $\lfloor x \rfloor$ and $\lceil x \rceil$ respectively. The result is always represented exactly, requiring no more bits to store than the input. To round the result to a floating-point number with a lower precision, call *arf_set_round()* afterwards.

3.2.5 Comparisons and bounds

int **arf_equal**(const *arf_t* *x*, const *arf_t* *y*)

int **arf_equal_si**(const *arf_t* *x*, *slong* *y*)
Returns nonzero iff x and y are exactly equal. This function does not treat NaN specially, i.e. NaN compares as equal to itself.

int **arf_cmp**(const *arf_t* *x*, const *arf_t* *y*)
Returns negative, zero, or positive, depending on whether x is respectively smaller, equal, or greater compared to y. Comparison with NaN is undefined.

int **arf_cmpabs**(const *arf_t* *x*, const *arf_t* *y*)

int **arf_cmpabs_ui**(const *arf_t* *x*, *ulong* *y*)

int **arf_cmpabs_mag**(const *arf_t* x, const *mag_t* y)
Compares the absolute values of x and y.

int **arf_cmp_2exp_si**(const *arf_t* x, *slong* e)

int **arf_cmpabs_2exp_si**(const *arf_t* x, *slong* e)
Compares x (respectively its absolute value) with 2^e.

int **arf_sgn**(const *arf_t* x)
Returns -1, 0 or $+1$ according to the sign of x. The sign of NaN is undefined.

void **arf_min**(*arf_t* z, const *arf_t* a, const *arf_t* b)

void **arf_max**(*arf_t* z, const *arf_t* a, const *arf_t* b)
Sets z respectively to the minimum and the maximum of a and b.

slong **arf_bits**(const *arf_t* x)
Returns the number of bits needed to represent the absolute value of the mantissa of x, i.e. the minimum precision sufficient to represent x exactly. Returns 0 if x is a special value.

int **arf_is_int**(const *arf_t* x)
Returns nonzero iff x is integer-valued.

int **arf_is_int_2exp_si**(const *arf_t* x, *slong* e)
Returns nonzero iff x equals $n2^e$ for some integer n.

void **arf_abs_bound_lt_2exp_fmpz**(*fmpz_t* b, const *arf_t* x)
Sets b to the smallest integer such that $|x| < 2^b$. If x is zero, infinity or NaN, the result is undefined.

void **arf_abs_bound_le_2exp_fmpz**(*fmpz_t* b, const *arf_t* x)
Sets b to the smallest integer such that $|x| \le 2^b$. If x is zero, infinity or NaN, the result is undefined.

slong **arf_abs_bound_lt_2exp_si**(const *arf_t* x)
Returns the smallest integer b such that $|x| < 2^b$, clamping the result to lie between -*ARF_PREC_EXACT* and *ARF_PREC_EXACT* inclusive. If x is zero, -*ARF_PREC_EXACT* is returned, and if x is infinity or NaN, *ARF_PREC_EXACT* is returned.

3.2.6 Magnitude functions

void **arf_get_mag**(*mag_t* y, const *arf_t* x)
Sets y to an upper bound for the absolute value of x.

void **arf_get_mag_lower**(*mag_t* y, const *arf_t* x)
Sets y to a lower bound for the absolute value of x.

void **arf_set_mag**(*arf_t* y, const *mag_t* x)
Sets y to x.

void **mag_init_set_arf**(*mag_t* y, const *arf_t* x)
Initializes y and sets it to an upper bound for x.

void **mag_fast_init_set_arf**(*mag_t* y, const *arf_t* x)
Initializes y and sets it to an upper bound for x. Assumes that the exponent of y is small.

void **arf_mag_set_ulp**(*mag_t* z, const *arf_t* y, *slong* *prec*)
Sets z to the magnitude of the unit in the last place (ulp) of y at precision *prec*.

void **arf_mag_add_ulp**(*mag_t* z, const *mag_t* x, const *arf_t* y, *slong* *prec*)
Sets z to an upper bound for the sum of x and the magnitude of the unit in the last place (ulp) of y at precision *prec*.

void **arf_mag_fast_add_ulp**(*mag_t* z, const *mag_t* x, const *arf_t* y, *slong* *prec*)
Sets z to an upper bound for the sum of x and the magnitude of the unit in the last place (ulp) of y at precision *prec*. Assumes that all exponents are small.

3.2.7 Shallow assignment

void **arf_init_set_shallow**(*arf_t* *z*, const *arf_t* *x*)

void **arf_init_set_mag_shallow**(*arf_t* *z*, const *mag_t* *x*)
Initializes z to a shallow copy of x. A shallow copy just involves copying struct data (no heap allocation is performed).

The target variable z may not be cleared or modified in any way (it can only be used as constant input to functions), and may not be used after x has been cleared. Moreover, after x has been assigned shallowly to z, no modification of x is permitted as slong as z is in use.

void **arf_init_neg_shallow**(*arf_t* *z*, const *arf_t* *x*)

void **arf_init_neg_mag_shallow**(*arf_t* *z*, const *mag_t* *x*)
Initializes z shallowly to the negation of x.

3.2.8 Random number generation

void **arf_randtest**(*arf_t* *x*, flint_rand_t *state*, *slong* *bits*, *slong* *mag_bits*)
Generates a finite random number whose mantissa has precision at most *bits* and whose exponent has at most *mag_bits* bits. The values are distributed non-uniformly: special bit patterns are generated with high probability in order to allow the test code to exercise corner cases.

void **arf_randtest_not_zero**(*arf_t* *x*, flint_rand_t *state*, *slong* *bits*, *slong* *mag_bits*)
Identical to *arf_randtest()*, except that zero is never produced as an output.

void **arf_randtest_special**(*arf_t* *x*, flint_rand_t *state*, *slong* *bits*, *slong* *mag_bits*)
Identical to *arf_randtest()*, except that the output occasionally is set to an infinity or NaN.

3.2.9 Input and output

void **arf_debug**(const *arf_t* *x*)
Prints information about the internal representation of x.

void **arf_print**(const *arf_t* *x*)
Prints x as an integer mantissa and exponent.

void **arf_printd**(const *arf_t* *y*, *slong* *d*)
Prints x as a decimal floating-point number, rounding to d digits. This function is currently implemented using MPFR, and does not support large exponents.

void **arf_fprint**(FILE * *file*, const *arf_t* *x*)
Prints x as an integer mantissa and exponent to the stream *file*.

void **arf_fprintd**(FILE * *file*, const *arf_t* *y*, *slong* *d*)
Prints x as a decimal floating-point number to the stream *file*, rounding to d digits. This function is currently implemented using MPFR, and does not support large exponents.

3.2.10 Addition and multiplication

void **arf_abs**(*arf_t* *y*, const *arf_t* *x*)
Sets y to the absolute value of x.

void **arf_neg**(*arf_t* *y*, const *arf_t* *x*)
Sets $y = -x$ exactly.

int **arf_neg_round**(*arf_t* *y*, const *arf_t* *x*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $y = -x$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact.

void **arf_mul_2exp_si**(*arf_t* *y*, const *arf_t* *x*, *slong* *e*)

void **arf_mul_2exp_fmpz**(*arf_t* *y*, const *arf_t* *x*, const *fmpz_t* *e*)
Sets $y = x2^e$ exactly.

int **arf_mul**(*arf_t* *z*, const *arf_t* *x*, const *arf_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_mul_ui**(*arf_t* *z*, const *arf_t* *x*, *ulong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_mul_si**(*arf_t* *z*, const *arf_t* *x*, *slong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_mul_mpz**(*arf_t* *z*, const *arf_t* *x*, const mpz_t *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_mul_fmpz**(*arf_t* *z*, const *arf_t* *x*, const *fmpz_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $z = x \times y$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact.

int **arf_add**(*arf_t* *z*, const *arf_t* *x*, const *arf_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_add_si**(*arf_t* *z*, const *arf_t* *x*, *slong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_add_ui**(*arf_t* *z*, const *arf_t* *x*, *ulong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_add_fmpz**(*arf_t* *z*, const *arf_t* *x*, const *fmpz_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $z = x + y$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact.

int **arf_add_fmpz_2exp**(*arf_t* *z*, const *arf_t* *x*, const *fmpz_t* *y*, const *fmpz_t* *e*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $z = x + y2^e$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact.

int **arf_sub**(*arf_t* *z*, const *arf_t* *x*, const *arf_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_sub_si**(*arf_t* *z*, const *arf_t* *x*, *slong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_sub_ui**(*arf_t* *z*, const *arf_t* *x*, *ulong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_sub_fmpz**(*arf_t* *z*, const *arf_t* *x*, const *fmpz_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $z = x - y$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact.

int **arf_addmul**(*arf_t* *z*, const *arf_t* *x*, const *arf_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_addmul_ui**(*arf_t* *z*, const *arf_t* *x*, *ulong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_addmul_si**(*arf_t* *z*, const *arf_t* *x*, *slong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_addmul_mpz**(*arf_t* *z*, const *arf_t* *x*, const mpz_t *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_addmul_fmpz**(*arf_t* *z*, const *arf_t* *x*, const *fmpz_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $z = z + x \times y$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact.

int **arf_submul**(*arf_t* *z*, const *arf_t* *x*, const *arf_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_submul_ui**(*arf_t* *z*, const *arf_t* *x*, *ulong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_submul_si**(*arf_t* *z*, const *arf_t* *x*, *slong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_submul_mpz**(*arf_t* *z*, const *arf_t* *x*, const mpz_t *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_submul_fmpz**(*arf_t* *z*, const *arf_t* *x*, const *fmpz_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $z = z - x \times y$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact.

3.2.11 Summation

int **arf_sum**(*arf_t* *s*, arf_srcptr *terms*, *slong* *len*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets *s* to the sum of the array *terms* of length *len*, rounded to *prec* bits in the direction specified by *rnd*. The sum is computed as if done without any intermediate rounding error, with only a single

rounding applied to the final result. Unlike repeated calls to *arf_add()* with infinite precision, this function does not overflow if the magnitudes of the terms are far apart. Warning: this function is implemented naively, and the running time is quadratic with respect to *len* in the worst case.

3.2.12 Division

int **arf_div**(*arf_t* *z*, const *arf_t* *x*, const *arf_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_div_ui**(*arf_t* *z*, const *arf_t* *x*, *ulong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_ui_div**(*arf_t* *z*, *ulong* *x*, const *arf_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_div_si**(*arf_t* *z*, const *arf_t* *x*, *slong* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_si_div**(*arf_t* *z*, *slong* *x*, const *arf_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_div_fmpz**(*arf_t* *z*, const *arf_t* *x*, const *fmpz_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_fmpz_div**(*arf_t* *z*, const *fmpz_t* *x*, const *arf_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_fmpz_div_fmpz**(*arf_t* *z*, const *fmpz_t* *x*, const *fmpz_t* *y*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $z = x/y$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact. The result is NaN if y is zero.

3.2.13 Square roots

int **arf_sqrt**(*arf_t* *z*, const *arf_t* *x*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_sqrt_ui**(*arf_t* *z*, *ulong* *x*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_sqrt_fmpz**(*arf_t* *z*, const *fmpz_t* *x*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $z = \sqrt{x}$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact. The result is NaN if x is negative.

int **arf_rsqrt**(*arf_t* *z*, const *arf_t* *x*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $z = 1/\sqrt{x}$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact. The result is NaN if x is negative, and $+\infty$ if x is zero.

int **arf_root**(*arf_t* *z*, const *arf_t* *x*, *ulong* *k*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets $z = x^{1/k}$, rounded to *prec* bits in the direction specified by *rnd*, returning nonzero iff the operation is inexact. The result is NaN if x is negative. Warning: this function is a wrapper around the MPFR root function. It gets slow and uses much memory for large k.

3.2.14 Complex arithmetic

int **arf_complex_mul**(*arf_t* *e*, *arf_t* *f*, const *arf_t* *a*, const *arf_t* *b*, const *arf_t* *c*, const *arf_t* *d*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arf_complex_mul_fallback**(*arf_t* *e*, *arf_t* *f*, const *arf_t* *a*, const *arf_t* *b*, const *arf_t* *c*, const *arf_t* *d*, *slong* *prec*, *arf_rnd_t* *rnd*)
Computes the complex product $e + fi = (a + bi)(c + di)$, rounding both e and f correctly to *prec* bits in the direction specified by *rnd*. The first bit in the return code indicates inexactness of e, and the second bit indicates inexactness of f.

If any of the components a, b, c, d is zero, two real multiplications and no additions are done. This convention is used even if any other part contains an infinity or NaN, and the behavior with infinite/NaN input is defined accordingly.

The *fallback* version is implemented naively, for testing purposes. No squaring optimization is implemented.

int **arf_complex_sqr**(*arf_t* *e*, *arf_t* *f*, const *arf_t* *a*, const *arf_t* *b*, *slong* *prec*, *arf_rnd_t* *rnd*)
Computes the complex square $e + fi = (a + bi)^2$. This function has identical semantics to *arf_complex_mul()* (with $c = a, b = d$), but is faster.

3.2.15 Low-level methods

int **_arf_get_integer_mpn**(*mp_ptr* *y*, *mp_srcptr* *xp*, *mp_size_t* *xn*, *slong* *exp*)
Given a floating-point number x represented by *xn* limbs at *xp* and an exponent *exp*, writes the integer part of x to y, returning whether the result is inexact. The correct number of limbs is written (no limbs are written if the integer part of x is zero). Assumes that `xp[0]` is nonzero and that the top bit of `xp[xn-1]` is set.

int **_arf_set_mpn_fixed**(*arf_t* *z*, *mp_srcptr* *xp*, *mp_size_t* *xn*, *mp_size_t* *fixn*, int *negative*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets z to the fixed-point number having *xn* total limbs and *fixn* fractional limbs, negated if *negative* is set, rounding z to *prec* bits in the direction *rnd* and returning whether the result is inexact. Both *xn* and *fixn* must be nonnegative and not so large that the bit shift would overflow an *slong*, but otherwise no assumptions are made about the input.

int **_arf_set_round_ui**(*arf_t* *z*, *ulong* *x*, int *sgnbit*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets z to the integer x, negated if *sgnbit* is 1, rounded to *prec* bits in the direction specified by *rnd*. There are no assumptions on x.

int **_arf_set_round_uiui**(*arf_t* *z*, *slong* * *fix*, *mp_limb_t* *hi*, *mp_limb_t* *lo*, int *sgnbit*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets the mantissa of z to the two-limb mantissa given by *hi* and *lo*, negated if *sgnbit* is 1, rounded to *prec* bits in the direction specified by *rnd*. Requires that not both *hi* and *lo* are zero. Writes the exponent shift to *fix* without writing the exponent of z directly.

int **_arf_set_round_mpn**(*arf_t* *z*, *slong* * *exp_shift*, *mp_srcptr* *x*, *mp_size_t* *xn*, int *sgnbit*, *slong* *prec*, *arf_rnd_t* *rnd*)
Sets the mantissa of z to the mantissa given by the *xn* limbs in x, negated if *sgnbit* is 1, rounded to *prec* bits in the direction specified by *rnd*. Returns the inexact flag. Requires that *xn* is positive and that the top limb of x is nonzero. If x has leading zero bits, writes the shift to *exp_shift*. This method does not write the exponent of z directly. Requires that x does not point to the limbs of z.

CHAPTER

FOUR

REAL AND COMPLEX NUMBERS

Real numbers (*arb_t*) are represented as midpoint-radius intervals, also known as balls. Complex numbers (*acb_t*) are represented in rectangular form, with balls for the real and imaginary parts.

4.1 arb.h – real numbers

An *arb_t* represents a ball over the real numbers, that is, an interval $[m \pm r] \equiv [m - r, m + r]$ where the midpoint m and the radius r are (extended) real numbers and r is nonnegative (possibly infinite). The result of an (approximate) operation done on *arb_t* variables is a ball which contains the result of the (mathematically exact) operation applied to any choice of points in the input balls. In general, the output ball is not the smallest possible.

The precision parameter passed to each function roughly indicates the precision to which calculations on the midpoint are carried out (operations on the radius are always done using a fixed, small precision.)

For arithmetic operations, the precision parameter currently simply specifies the precision of the corresponding *arf_t* operation. In the future, the arithmetic might be made faster by incorporating sloppy rounding (typically equivalent to a loss of 1-2 bits of effective working precision) when the result is known to be inexact (while still propagating errors rigorously, of course). Arithmetic operations done on exact input with exactly representable output are always guaranteed to produce exact output.

For more complex operations, the precision parameter indicates a minimum working precision (algorithms might allocate extra internal precision to attempt to produce an output accurate to the requested number of bits, especially when the required precision can be estimated easily, but this is not generally required).

If the precision is increased and the inputs either are exact or are computed with increased accuracy as well, the output should converge proportionally, absent any bugs. The general intended strategy for using ball arithmetic is to add a few guard bits, and then repeat the calculation as necessary with an exponentially increasing number of guard bits (Ziv's strategy) until the result is exact enough for one's purposes (typically the first attempt will be successful).

The following balls with an infinite or NaN component are permitted, and may be returned as output from functions.

- The ball $[+\infty \pm c]$, where c is finite, represents the point at positive infinity. Such a ball can always be replaced by $[+\infty \pm 0]$ while preserving mathematical correctness (this is currently not done automatically by the library).
- The ball $[-\infty \pm c]$, where c is finite, represents the point at negative infinity. Such a ball can always be replaced by $[-\infty \pm 0]$ while preserving mathematical correctness (this is currently not done automatically by the library).
- The ball $[c \pm \infty]$, where c is finite or infinite, represents the whole extended real line $[-\infty, +\infty]$. Such a ball can always be replaced by $[0 \pm \infty]$ while preserving mathematical correctness (this is currently not done automatically by the library). Note that there is no way to represent a half-infinite interval such as $[0, \infty]$.
- The ball $[\mathrm{NaN} \pm c]$, where c is finite or infinite, represents an indeterminate value (the value could be any extended real number, or it could represent a function being evaluated outside its domain

of definition, for example where the result would be complex). Such an indeterminate ball can always be replaced by [NaN $\pm\infty$] while preserving mathematical correctness (this is currently not done automatically by the library).

4.1.1 Types, macros and constants

arb_struct

arb_t
An *arb_struct* consists of an *arf_struct* (the midpoint) and a *mag_struct* (the radius). An *arb_t* is defined as an array of length one of type *arb_struct*, permitting an *arb_t* to be passed by reference.

arb_ptr
Alias for `arb_struct *`, used for vectors of numbers.

arb_srcptr
Alias for `const arb_struct *`, used for vectors of numbers when passed as constant input to functions.

arb_midref(x)
Macro returning a pointer to the midpoint of x as an *arf_t*.

arb_radref(x)
Macro returning a pointer to the radius of x as a *mag_t*.

4.1.2 Memory management

void **arb_init**(*arb_t* x)
Initializes the variable x for use. Its midpoint and radius are both set to zero.

void **arb_clear**(*arb_t* x)
Clears the variable x, freeing or recycling its allocated memory.

arb_ptr **_arb_vec_init**(*slong* n)
Returns a pointer to an array of n initialized *arb_struct* entries.

void **_arb_vec_clear**(*arb_ptr* v, *slong* n)
Clears an array of n initialized *arb_struct* entries.

void **arb_swap**(*arb_t* x, *arb_t* y)
Swaps x and y efficiently.

4.1.3 Assignment and rounding

void **arb_set**(*arb_t* y, const *arb_t* x)

void **arb_set_arf**(*arb_t* y, const *arf_t* x)

void **arb_set_si**(*arb_t* y, *slong* x)

void **arb_set_ui**(*arb_t* y, *ulong* x)

void **arb_set_d**(*arb_t* y, double x)

void **arb_set_fmpz**(*arb_t* y, const *fmpz_t* x)
Sets y to the value of x without rounding.

void **arb_set_fmpz_2exp**(*arb_t* y, const *fmpz_t* x, const *fmpz_t* e)
Sets y to $x \cdot 2^e$.

void **arb_set_round**(*arb_t* y, const *arb_t* x, *slong* *prec*)

void **arb_set_round_fmpz**(*arb_t* *y*, const *fmpz_t* *x*, *slong* *prec*)
Sets y to the value of x, rounded to *prec* bits.

void **arb_set_round_fmpz_2exp**(*arb_t* *y*, const *fmpz_t* *x*, const *fmpz_t* *e*, *slong* *prec*)
Sets y to $x \cdot 2^e$, rounded to *prec* bits.

void **arb_set_fmpq**(*arb_t* *y*, const *fmpq_t* *x*, *slong* *prec*)
Sets y to the rational number x, rounded to *prec* bits.

int **arb_set_str**(*arb_t* *res*, const char * *inp*, *slong* *prec*)
Sets *res* to the value specified by the human-readable string *inp*. The input may be a decimal floating-point literal, such as "25", "0.001", "7e+141" or "-31.4159e-1", and may also consist of two such literals separated by the symbol "+/-" and optionally enclosed in brackets, e.g. "[3.25 +/- 0.0001]", or simply "[+/- 10]" with an implicit zero midpoint. The output is rounded to *prec* bits, and if the binary-to-decimal conversion is inexact, the resulting error is added to the radius.

The symbols "inf" and "nan" are recognized (a nan midpoint results in an indeterminate interval, with infinite radius).

Returns 0 if successful and nonzero if unsuccessful. If unsuccessful, the result is set to an indeterminate interval.

char * **arb_get_str**(const *arb_t* *x*, *slong* *n*, *ulong* *flags*)
Returns a nice human-readable representation of x, with at most n digits of the midpoint printed.

With default flags, the output can be parsed back with *arb_set_str()*, and this is guaranteed to produce an interval containing the original interval x.

By default, the output is rounded so that the value given for the midpoint is correct up to 1 ulp (unit in the last decimal place).

If *ARB_STR_MORE* is added to *flags*, more (possibly incorrect) digits may be printed.

If *ARB_STR_NO_RADIUS* is added to *flags*, the radius is not included in the output if at least 1 digit of the midpoint can be printed.

By adding a multiple m of *ARB_STR_CONDENSE* to *flags*, strings of more than three times m consecutive digits are condensed, only printing the leading and trailing m digits along with brackets indicating the number of digits omitted (useful when computing values to extremely high precision).

4.1.4 Assignment of special values

void **arb_zero**(*arb_t* *x*)
Sets x to zero.

void **arb_one**(*arb_t* *f*)
Sets x to the exact integer 1.

void **arb_pos_inf**(*arb_t* *x*)
Sets x to positive infinity, with a zero radius.

void **arb_neg_inf**(*arb_t* *x*)
Sets x to negative infinity, with a zero radius.

void **arb_zero_pm_inf**(*arb_t* *x*)
Sets x to $[0 \pm \infty]$, representing the whole extended real line.

void **arb_indeterminate**(*arb_t* *x*)
Sets x to $[\mathrm{NaN} \pm \infty]$, representing an indeterminate result.

4.1.5 Input and output

void **arb_print**(const *arb_t* *x*)
Prints the internal representation of *x*.

void **arb_printd**(const *arb_t* *x*, *slong* *digits*)
Prints *x* in decimal. The printed value of the radius is not adjusted to compensate for the fact that the binary-to-decimal conversion of both the midpoint and the radius introduces additional error.

void **arb_printn**(const *arb_t* *x*, *slong* *digits*, *ulong* *flags*)
Prints a nice decimal representation of *x*. By default, the output is guaranteed to be correct to within one unit in the last digit. An error bound is also printed explicitly. See *arb_get_str()* for details.

void **arb_fprint**(FILE * *file*, const *arb_t* *x*)
Prints the internal representation of *x* to the stream *file*.

void **arb_fprintd**(FILE * *file*, const *arb_t* *x*, *slong* *digits*)
Prints *x* in decimal to the stream *file*. The printed value of the radius is not adjusted to compensate for the fact that the binary-to-decimal conversion of both the midpoint and the radius introduces additional error.

void **arb_fprintn**(FILE * *file*, const *arb_t* *x*, *slong* *digits*, *ulong* *flags*)
Prints a nice decimal representation of *x* to the stream *file*. By default, the output is guaranteed to be correct to within one unit in the last digit. An error bound is also printed explicitly. See *arb_get_str()* for details.

4.1.6 Random number generation

void **arb_randtest**(*arb_t* *x*, flint_rand_t *state*, *slong* *prec*, *slong* *mag_bits*)
Generates a random ball. The midpoint and radius will both be finite.

void **arb_randtest_exact**(*arb_t* *x*, flint_rand_t *state*, *slong* *prec*, *slong* *mag_bits*)
Generates a random number with zero radius.

void **arb_randtest_precise**(*arb_t* *x*, flint_rand_t *state*, *slong* *prec*, *slong* *mag_bits*)
Generates a random number with radius around $2^{-\text{prec}}$ the magnitude of the midpoint.

void **arb_randtest_wide**(*arb_t* *x*, flint_rand_t *state*, *slong* *prec*, *slong* *mag_bits*)
Generates a random number with midpoint and radius chosen independently, possibly giving a very large interval.

void **arb_randtest_special**(*arb_t* *x*, flint_rand_t *state*, *slong* *prec*, *slong* *mag_bits*)
Generates a random interval, possibly having NaN or an infinity as the midpoint and possibly having an infinite radius.

void **arb_get_rand_fmpq**(*fmpq_t* *q*, flint_rand_t *state*, const *arb_t* *x*, *slong* *bits*)
Sets *q* to a random rational number from the interval represented by *x*. A denominator is chosen by multiplying the binary denominator of *x* by a random integer up to *bits* bits.

The outcome is undefined if the midpoint or radius of *x* is non-finite, or if the exponent of the midpoint or radius is so large or small that representing the endpoints as exact rational numbers would cause overflows.

4.1.7 Radius and interval operations

void **arb_get_mid_arb**(*arb_t* *m*, const *arb_t* *x*)
Sets *m* to the midpoint of *x*.

void **arb_get_rad_arb**(*arb_t* *r*, const *arb_t* *x*)
Sets *m* to the radius of *x*.

void **arb_add_error_arf**(*arb_t* *x*, const *arf_t* *err*)

void **arb_add_error_mag**(*arb_t* *x*, const *mag_t* *err*)

void **arb_add_error**(*arb_t* *x*, const *arb_t* *err*)
Adds the absolute value of *err* to the radius of *x* (the operation is done in-place).

void **arb_add_error_2exp_si**(*arb_t* *x*, *slong* *e*)

void **arb_add_error_2exp_fmpz**(*arb_t* *x*, const *fmpz_t* *e*)
Adds 2^e to the radius of *x*.

void **arb_union**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)
Sets *z* to a ball containing both *x* and *y*.

int **arb_intersection**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)
If *x* and *y* overlap according to ***arb_overlaps()***, then *z* is set to a ball containing the intersection of *x* and *y* and a nonzero value is returned. Otherwise zero is returned and the value of *z* is undefined. If *x* or *y* contains NaN, the result is NaN.

void **arb_get_abs_ubound_arf**(*arf_t* *u*, const *arb_t* *x*, *slong* *prec*)
Sets *u* to the upper bound for the absolute value of *x*, rounded up to *prec* bits. If *x* contains NaN, the result is NaN.

void **arb_get_abs_lbound_arf**(*arf_t* *u*, const *arb_t* *x*, *slong* *prec*)
Sets *u* to the lower bound for the absolute value of *x*, rounded down to *prec* bits. If *x* contains NaN, the result is NaN.

void **arb_get_ubound_arf**(*arf_t* *u*, const *arb_t* *x*, long *prec*)
Sets *u* to the upper bound for the value of *x*, rounded up to *prec* bits. If *x* contains NaN, the result is NaN.

void **arb_get_lbound_arf**(*arf_t* *u*, const *arb_t* *x*, long *prec*)
Sets *u* to the lower bound for the value of *x*, rounded down to *prec* bits. If *x* contains NaN, the result is NaN.

void **arb_get_mag**(*mag_t* *z*, const *arb_t* *x*)
Sets *z* to an upper bound for the absolute value of *x*. If *x* contains NaN, the result is positive infinity.

void **arb_get_mag_lower**(*mag_t* *z*, const *arb_t* *x*)
Sets *z* to a lower bound for the absolute value of *x*. If *x* contains NaN, the result is zero.

void **arb_get_mag_lower_nonnegative**(*mag_t* *z*, const *arb_t* *x*)
Sets *z* to a lower bound for the signed value of *x*, or zero if *x* overlaps with the negative half-axis. If *x* contains NaN, the result is zero.

void **arb_get_interval_fmpz_2exp**(*fmpz_t* *a*, *fmpz_t* *b*, *fmpz_t* *exp*, const *arb_t* *x*)
Computes the exact interval represented by *x*, in the form of an integer interval multiplied by a power of two, i.e. $x = [a, b] \times 2^{\text{exp}}$. The result is normalized by removing common trailing zeros from *a* and *b*.

This method aborts if *x* is infinite or NaN, or if the difference between the exponents of the midpoint and the radius is so large that allocating memory for the result fails.

Warning: this method will allocate a huge amount of memory to store the result if the exponent difference is huge. Memory allocation could succeed even if the required space is far larger than the physical memory available on the machine, resulting in swapping. It is recommended to check that the midpoint and radius of *x* both are within a reasonable range before calling this method.

void **arb_set_interval_arf**(*arb_t* *x*, const *arf_t* *a*, const *arf_t* *b*, *slong* *prec*)

void **arb_set_interval_mpfr**(*arb_t* *x*, const mpfr_t *a*, const mpfr_t *b*, *slong* *prec*)
Sets *x* to a ball containing the interval $[a, b]$. We require that $a \le b$.

void **arb_get_interval_arf**(*arf_t* *a*, *arf_t* *b*, const *arb_t* *x*, *slong* *prec*)

void **arb_get_interval_mpfr**(mpfr_t *a*, mpfr_t *b*, const *arb_t* *x*)
Constructs an interval $[a, b]$ containing the ball *x*. The MPFR version uses the precision of the output variables.

slong **arb_rel_error_bits**(const *arb_t* *x*)
Returns the effective relative error of x measured in bits, defined as the difference between the position of the top bit in the radius and the top bit in the midpoint, plus one. The result is clamped between plus/minus *ARF_PREC_EXACT*.

slong **arb_rel_accuracy_bits**(const *arb_t* *x*)
Returns the effective relative accuracy of x measured in bits, equal to the negative of the return value from *arb_rel_error_bits()*.

slong **arb_bits**(const *arb_t* *x*)
Returns the number of bits needed to represent the absolute value of the mantissa of the midpoint of x, i.e. the minimum precision sufficient to represent x exactly. Returns 0 if the midpoint of x is a special value.

void **arb_trim**(*arb_t* *y*, const *arb_t* *x*)
Sets y to a trimmed copy of x: rounds x to a number of bits equal to the accuracy of x (as indicated by its radius), plus a few guard bits. The resulting ball is guaranteed to contain x, but is more economical if x has less than full accuracy.

int **arb_get_unique_fmpz**(*fmpz_t* *z*, const *arb_t* *x*)
If x contains a unique integer, sets z to that value and returns nonzero. Otherwise (if x represents no integers or more than one integer), returns zero.

This method aborts if there is a unique integer but that integer is so large that allocating memory for the result fails.

Warning: this method will allocate a huge amount of memory to store the result if there is a unique integer and that integer is huge. Memory allocation could succeed even if the required space is far larger than the physical memory available on the machine, resulting in swapping. It is recommended to check that the midpoint of x is within a reasonable range before calling this method.

void **arb_floor**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)

void **arb_ceil**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)
Sets y to a ball containing $\lfloor x \rfloor$ and $\lceil x \rceil$ respectively, with the midpoint of y rounded to at most *prec* bits.

void **arb_get_fmpz_mid_rad_10exp**(*fmpz_t* *mid*, *fmpz_t* *rad*, *fmpz_t* *exp*, const *arb_t* *x*, *slong* *n*)
Assuming that x is finite and not exactly zero, computes integers *mid*, *rad*, *exp* such that $x \in [m - r, m + r] \times 10^e$ and such that the larger out of *mid* and *rad* has at least n digits plus a few guard digits. If x is infinite or exactly zero, the outputs are all set to zero.

int **arb_can_round_arf**(const *arb_t* *x*, *slong* *prec*, *arf_rnd_t* *rnd*)

int **arb_can_round_mpfr**(const *arb_t* *x*, *slong* *prec*, mpfr_rnd_t *rnd*)
Returns nonzero if rounding the midpoint of x to *prec* bits in the direction *rnd* is guaranteed to give the unique correctly rounded floating-point approximation for the real number represented by x.

In other words, if this function returns nonzero, applying *arf_set_round()*, or *arf_get_mpfr()*, or *arf_get_d()* to the midpoint of x is guaranteed to return a correctly rounded *arf_t*, *mpfr_t* (provided that *prec* is the precision of the output variable), or *double* (provided that *prec* is 53). Moreover, *arf_get_mpfr()* is guaranteed to return the correct ternary value according to MPFR semantics.

Note that the *mpfr* version of this function takes an MPFR rounding mode symbol as input, while the *arf* version takes an *arf* rounding mode symbol. Otherwise, the functions are identical.

This function may perform a fast, inexact test; that is, it may return zero in some cases even when correct rounding actually is possible.

To be conservative, zero is returned when x is non-finite, even if it is an "exact" infinity.

4.1.8 Comparisons

int **arb_is_zero**(const *arb_t* *x*)
Returns nonzero iff the midpoint and radius of x are both zero.

int **arb_is_nonzero**(const *arb_t* *x*)
Returns nonzero iff zero is not contained in the interval represented by x.

int **arb_is_one**(const *arb_t* *f*)
Returns nonzero iff x is exactly 1.

int **arb_is_finite**(const *arb_t* *x*)
Returns nonzero iff the midpoint and radius of x are both finite floating-point numbers, i.e. not infinities or NaN.

int **arb_is_exact**(const *arb_t* *x*)
Returns nonzero iff the radius of x is zero.

int **arb_is_int**(const *arb_t* *x*)
Returns nonzero iff x is an exact integer.

int **arb_equal**(const *arb_t* *x*, const *arb_t* *y*)
Returns nonzero iff x and y are equal as balls, i.e. have both the same midpoint and radius.

Note that this is not the same thing as testing whether both x and y certainly represent the same real number, unless either x or y is exact (and neither contains NaN). To test whether both operands *might* represent the same mathematical quantity, use *arb_overlaps()* or *arb_contains()*, depending on the circumstance.

int **arb_equal_si**(const *arb_t* *x*, *slong* *y*)
Returns nonzero iff x is equal to the integer y.

int **arb_is_positive**(const *arb_t* *x*)

int **arb_is_nonnegative**(const *arb_t* *x*)

int **arb_is_negative**(const *arb_t* *x*)

int **arb_is_nonpositive**(const *arb_t* *x*)
Returns nonzero iff all points p in the interval represented by x satisfy, respectively, $p > 0$, $p \geq 0$, $p < 0$, $p \leq 0$. If x contains NaN, returns zero.

int **arb_overlaps**(const *arb_t* *x*, const *arb_t* *y*)
Returns nonzero iff x and y have some point in common. If either x or y contains NaN, this function always returns nonzero (as a NaN could be anything, it could in particular contain any number that is included in the other operand).

int **arb_contains_arf**(const *arb_t* *x*, const *arf_t* *y*)

int **arb_contains_fmpq**(const *arb_t* *x*, const *fmpq_t* *y*)

int **arb_contains_fmpz**(const *arb_t* *x*, const *fmpz_t* *y*)

int **arb_contains_si**(const *arb_t* *x*, *slong* *y*)

int **arb_contains_mpfr**(const *arb_t* *x*, const mpfr_t *y*)

int **arb_contains**(const *arb_t* *x*, const *arb_t* *y*)
Returns nonzero iff the given number (or ball) y is contained in the interval represented by x.

If x is contains NaN, this function always returns nonzero (as it could represent anything, and in particular could represent all the points included in y). If y contains NaN and x does not, it always returns zero.

int **arb_contains_int**(const *arb_t* *x*)
Returns nonzero iff the interval represented by x contains an integer.

int **arb_contains_zero**(const *arb_t* *x*)

int **arb_contains_negative**(const *arb_t* *x*)

int **arb_contains_nonpositive**(const *arb_t* *x*)

int **arb_contains_positive**(const *arb_t* *x*)

int **arb_contains_nonnegative**(const *arb_t* *x*)
Returns nonzero iff there is any point p in the interval represented by x satisfying, respectively, $p = 0$, $p < 0$, $p \leq 0$, $p > 0$, $p \geq 0$. If x contains NaN, returns nonzero.

int **arb_eq**(const *arb_t* *x*, const *arb_t* *y*)

int **arb_ne**(const *arb_t* *x*, const *arb_t* *y*)

int **arb_lt**(const *arb_t* *x*, const *arb_t* *y*)

int **arb_le**(const *arb_t* *x*, const *arb_t* *y*)

int **arb_gt**(const *arb_t* *x*, const *arb_t* *y*)

int **arb_ge**(const *arb_t* *x*, const *arb_t* *y*)
Respectively performs the comparison $x = y$, $x \neq y$, $x < y$, $x \leq y$, $x > y$, $x \geq y$ in a mathematically meaningful way. If the comparison $t \,(\mathrm{op})\, u$ holds for all $t \in x$ and all $u \in y$, returns 1. Otherwise, returns 0.

The balls x and y are viewed as subintervals of the extended real line. Note that balls that are formally different can compare as equal under this definition: for example, $[-\infty \pm 3] = [-\infty \pm 0]$. Also $[-\infty] \leq [\infty \pm \infty]$.

The output is always 0 if either input has NaN as midpoint.

4.1.9 Arithmetic

void **arb_neg**(*arb_t* *y*, const *arb_t* *x*)

void **arb_neg_round**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)
Sets y to the negation of x.

void **arb_abs**(*arb_t* *y*, const *arb_t* *x*)
Sets y to the absolute value of x. No attempt is made to improve the interval represented by x if it contains zero.

void **arb_sgn**(*arb_t* *y*, const *arb_t* *x*)
Sets y to the sign function of x. The result is $[0 \pm 1]$ if x contains both zero and nonzero numbers.

void **arb_min**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)

void **arb_max**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)
Sets z respectively to the minimum and the maximum of x and y.

void **arb_add**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)

void **arb_add_arf**(*arb_t* *z*, const *arb_t* *x*, const *arf_t* *y*, *slong* *prec*)

void **arb_add_ui**(*arb_t* *z*, const *arb_t* *x*, *ulong* *y*, *slong* *prec*)

void **arb_add_si**(*arb_t* *z*, const *arb_t* *x*, *slong* *y*, *slong* *prec*)

void **arb_add_fmpz**(*arb_t* *z*, const *arb_t* *x*, const *fmpz_t* *y*, *slong* *prec*)
Sets $z = x + y$, rounded to *prec* bits. The precision can be *ARF_PREC_EXACT* provided that the result fits in memory.

void **arb_add_fmpz_2exp**(*arb_t* *z*, const *arb_t* *x*, const *fmpz_t* *m*, const *fmpz_t* *e*, *slong* *prec*)
Sets $z = x + m \cdot 2^e$, rounded to *prec* bits. The precision can be *ARF_PREC_EXACT* provided that the result fits in memory.

void **arb_sub**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)

void **arb_sub_arf**(*arb_t* *z*, const *arb_t* *x*, const *arf_t* *y*, *slong* *prec*)

void **arb_sub_ui**(*arb_t* *z*, const *arb_t* *x*, *ulong* *y*, *slong* *prec*)

void **arb_sub_si**(*arb_t* *z*, const *arb_t* *x*, *slong* *y*, *slong* *prec*)

void **arb_sub_fmpz**(*arb_t* *z*, const *arb_t* *x*, const *fmpz_t* *y*, *slong* *prec*)
Sets $z = x - y$, rounded to *prec* bits. The precision can be *ARF_PREC_EXACT* provided that the result fits in memory.

void **arb_mul**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)

void **arb_mul_arf**(*arb_t* *z*, const *arb_t* *x*, const *arf_t* *y*, *slong* *prec*)

void **arb_mul_si**(*arb_t* *z*, const *arb_t* *x*, *slong* *y*, *slong* *prec*)

void **arb_mul_ui**(*arb_t* *z*, const *arb_t* *x*, *ulong* *y*, *slong* *prec*)

void **arb_mul_fmpz**(*arb_t* *z*, const *arb_t* *x*, const *fmpz_t* *y*, *slong* *prec*)
Sets $z = x \cdot y$, rounded to *prec* bits. The precision can be *ARF_PREC_EXACT* provided that the result fits in memory.

void **arb_mul_2exp_si**(*arb_t* *y*, const *arb_t* *x*, *slong* *e*)

void **arb_mul_2exp_fmpz**(*arb_t* *y*, const *arb_t* *x*, const *fmpz_t* *e*)
Sets y to x multiplied by 2^e.

void **arb_addmul**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)

void **arb_addmul_arf**(*arb_t* *z*, const *arb_t* *x*, const *arf_t* *y*, *slong* *prec*)

void **arb_addmul_si**(*arb_t* *z*, const *arb_t* *x*, *slong* *y*, *slong* *prec*)

void **arb_addmul_ui**(*arb_t* *z*, const *arb_t* *x*, *ulong* *y*, *slong* *prec*)

void **arb_addmul_fmpz**(*arb_t* *z*, const *arb_t* *x*, const *fmpz_t* *y*, *slong* *prec*)
Sets $z = z + x \cdot y$, rounded to prec bits. The precision can be *ARF_PREC_EXACT* provided that the result fits in memory.

void **arb_submul**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)

void **arb_submul_arf**(*arb_t* *z*, const *arb_t* *x*, const *arf_t* *y*, *slong* *prec*)

void **arb_submul_si**(*arb_t* *z*, const *arb_t* *x*, *slong* *y*, *slong* *prec*)

void **arb_submul_ui**(*arb_t* *z*, const *arb_t* *x*, *ulong* *y*, *slong* *prec*)

void **arb_submul_fmpz**(*arb_t* *z*, const *arb_t* *x*, const *fmpz_t* *y*, *slong* *prec*)
Sets $z = z - x \cdot y$, rounded to prec bits. The precision can be *ARF_PREC_EXACT* provided that the result fits in memory.

void **arb_inv**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)
Sets z to $1/x$.

void **arb_div**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)

void **arb_div_arf**(*arb_t* *z*, const *arb_t* *x*, const *arf_t* *y*, *slong* *prec*)

void **arb_div_si**(*arb_t* *z*, const *arb_t* *x*, *slong* *y*, *slong* *prec*)

void **arb_div_ui**(*arb_t* *z*, const *arb_t* *x*, *ulong* *y*, *slong* *prec*)

void **arb_div_fmpz**(*arb_t* *z*, const *arb_t* *x*, const *fmpz_t* *y*, *slong* *prec*)

void **arb_fmpz_div_fmpz**(*arb_t* *z*, const *fmpz_t* *x*, const *fmpz_t* *y*, *slong* *prec*)

void **arb_ui_div**(*arb_t* *z*, *ulong* *x*, const *arb_t* *y*, *slong* *prec*)
Sets $z = x/y$, rounded to *prec* bits. If y contains zero, z is set to $0 \pm \infty$. Otherwise, error propagation uses the rule

$$\left|\frac{x}{y} - \frac{x+\xi_1 a}{y+\xi_2 b}\right| = \left|\frac{x\xi_2 b - y\xi_1 a}{y(y+\xi_2 b)}\right| \le \frac{|xb|+|ya|}{|y|(|y|-b)}$$

where $-1 \le \xi_1, \xi_2 \le 1$, and where the triangle inequality has been applied to the numerator and the reverse triangle inequality has been applied to the denominator.

void **arb_div_2expm1_ui**(*arb_t* *z*, const *arb_t* *x*, *ulong* *n*, *slong* *prec*)
Sets $z = x/(2^n - 1)$, rounded to *prec* bits.

4.1.10 Powers and roots

void **arb_sqrt**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)

void **arb_sqrt_arf**(*arb_t* *z*, const *arf_t* *x*, *slong* *prec*)

void **arb_sqrt_fmpz**(*arb_t* *z*, const *fmpz_t* *x*, *slong* *prec*)

void **arb_sqrt_ui**(*arb_t* *z*, *ulong* *x*, *slong* *prec*)
Sets z to the square root of x, rounded to *prec* bits.

If $x = m \pm x$ where $m \geq r \geq 0$, the propagated error is bounded by $\sqrt{m} - \sqrt{m-r} = \sqrt{m}(1 - \sqrt{1 - r/m}) \leq \sqrt{m}(r/m + (r/m)^2)/2$.

void **arb_sqrtpos**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets z to the square root of x, assuming that x represents a nonnegative number (i.e. discarding any negative numbers in the input interval).

void **arb_hypot**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)
Sets z to $\sqrt{x^2 + y^2}$.

void **arb_rsqrt**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)

void **arb_rsqrt_ui**(*arb_t* *z*, *ulong* *x*, *slong* *prec*)
Sets z to the reciprocal square root of x, rounded to *prec* bits. At high precision, this is faster than computing a square root.

void **arb_sqrt1pm1**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \sqrt{1+x} - 1$, computed accurately when $x \approx 0$.

void **arb_root_ui**(*arb_t* *z*, const *arb_t* *x*, *ulong* *k*, *slong* *prec*)
Sets z to the k-th root of x, rounded to *prec* bits. This function selects between different algorithms. For large k, it evaluates $\exp(\log(x)/k)$. For small k, it uses *arf_root()* at the midpoint and computes a propagated error bound as follows: if input interval is $[m-r, m+r]$ with $r \leq m$, the error is largest at $m - r$ where it satisfies

$$\begin{aligned} & m^{1/k} - (m-r)^{1/k} = m^{1/k}[1 - (1 - r/m)^{1/k}] \\ & = m^{1/k}[1 - \exp(\log(1 - r/m)/k)] \\ & \leq m^{1/k} \min(1, -\log(1 - r/m)/k) \\ & = m^{1/k} \min(1, \log(1 + r/(m-r))/k). \end{aligned}$$

This is evaluated using *mag_log1p()*.

void **arb_root**(*arb_t* *z*, const *arb_t* *x*, *ulong* *k*, *slong* *prec*)
Alias for *arb_root_ui()*, provided for backwards compatibility.

void **arb_sqr**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)
Sets y to be the square of x.

void **arb_pow_fmpz_binexp**(*arb_t* *y*, const *arb_t* *b*, const *fmpz_t* *e*, *slong* *prec*)

void **arb_pow_fmpz**(*arb_t* *y*, const *arb_t* *b*, const *fmpz_t* *e*, *slong* *prec*)

void **arb_pow_ui**(*arb_t* *y*, const *arb_t* *b*, *ulong* *e*, *slong* *prec*)

void **arb_ui_pow_ui**(*arb_t* *y*, *ulong* *b*, *ulong* *e*, *slong* *prec*)

void **arb_si_pow_ui**(*arb_t* *y*, *slong* *b*, *ulong* *e*, *slong* *prec*)
Sets $y = b^e$ using binary exponentiation (with an initial division if $e < 0$). Provided that b and e are small enough and the exponent is positive, the exact power can be computed by setting the precision to *ARF_PREC_EXACT*.

Note that these functions can get slow if the exponent is extremely large (in such cases *arb_pow()* may be superior).

void **arb_pow_fmpq**(*arb_t* *y*, const *arb_t* *x*, const *fmpq_t* *a*, *slong* *prec*)
Sets $y = b^e$, computed as $y = (b^{1/q})^p$ if the denominator of $e = p/q$ is small, and generally as $y = \exp(e \log b)$.

Note that this function can get slow if the exponent is extremely large (in such cases *arb_pow()* may be superior).

void **arb_pow**(*arb_t* *z*, const *arb_t* *x*, const *arb_t* *y*, *slong* *prec*)
Sets $z = x^y$, computed using binary exponentiation if y if a small exact integer, as $z = (x^{1/2})^{2y}$ if y is a small exact half-integer, and generally as $z = \exp(y \log x)$.

4.1.11 Exponentials and logarithms

void **arb_log_ui**(*arb_t* *z*, *ulong* *x*, *slong* *prec*)

void **arb_log_fmpz**(*arb_t* *z*, const *fmpz_t* *x*, *slong* *prec*)

void **arb_log_arf**(*arb_t* *z*, const *arf_t* *x*, *slong* *prec*)

void **arb_log**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \log(x)$.

At low to medium precision (up to about 4096 bits), *arb_log_arf()* uses table-based argument reduction and fast Taylor series evaluation via *_arb_atan_taylor_rs()*. At high precision, it falls back to MPFR. The function *arb_log()* simply calls *arb_log_arf()* with the midpoint as input, and separately adds the propagated error.

void **arb_log_ui_from_prev**(*arb_t* *log_k1*, *ulong* *k1*, *arb_t* *log_k0*, *ulong* *k0*, *slong* *prec*)
Computes $\log(k_1)$, given $\log(k_0)$ where $k_0 < k_1$. At high precision, this function uses the formula $\log(k_1) = \log(k_0) + 2\,\mathrm{atanh}((k_1 - k_0)/(k_1 + k_0))$, evaluating the inverse hyperbolic tangent using binary splitting (for best efficiency, k_0 should be large and $k_1 - k_0$ should be small). Otherwise, it ignores $\log(k_0)$ and evaluates the logarithm the usual way.

void **arb_log1p**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \log(1 + x)$, computed accurately when $x \approx 0$.

void **arb_log_base_ui**(*arb_t* *res*, const *arb_t* *x*, *ulong* *b*, *slong* *prec*)
Sets *res* to $\log_b(x)$. The result is computed exactly when possible.

void **arb_exp**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \exp(x)$. Error propagation is done using the following rule: assuming $x = m \pm r$, the error is largest at $m + r$, and we have $\exp(m + r) - \exp(m) = \exp(m)(\exp(r) - 1) \le r \exp(m + r)$.

void **arb_expm1**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \exp(x) - 1$, computed accurately when $x \approx 0$.

void **arb_exp_invexp**(*arb_t* *z*, *arb_t* *w*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \exp(x)$ and $w = \exp(-x)$. The second exponential is computed from the first using a division, but propagated error bounds are computed separately.

4.1.12 Trigonometric functions

void **arb_sin**(*arb_t* *s*, const *arb_t* *x*, *slong* *prec*)

void **arb_cos**(*arb_t* *c*, const *arb_t* *x*, *slong* *prec*)

void **arb_sin_cos**(*arb_t* *s*, *arb_t* *c*, const *arb_t* *x*, *slong* *prec*)
Sets $s = \sin(x)$, $c = \cos(x)$. Error propagation uses the rule $|\sin(m \pm r) - \sin(m)| \le \min(r, 2)$.

void **arb_sin_pi**(*arb_t* *s*, const *arb_t* *x*, *slong* *prec*)

void **arb_cos_pi**(*arb_t* *c*, const *arb_t* *x*, *slong* *prec*)

void **arb_sin_cos_pi**(*arb_t* *s*, *arb_t* *c*, const *arb_t* *x*, *slong* *prec*)
Sets $s = \sin(\pi x)$, $c = \cos(\pi x)$.

void **arb_tan**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)
Sets $y = \tan(x) = \sin(x)/\cos(y)$.

void **arb_cot**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)
Sets $y = \cot(x) = \cos(x)/\sin(y)$.

void **arb_sin_cos_pi_fmpq**(*arb_t* *s*, *arb_t* *c*, const *fmpq_t* *x*, *slong* *prec*)

void **arb_sin_pi_fmpq**(*arb_t* *s*, const *fmpq_t* *x*, *slong* *prec*)

void **arb_cos_pi_fmpq**(*arb_t* *c*, const *fmpq_t* *x*, *slong* *prec*)
Sets $s = \sin(\pi x)$, $c = \cos(\pi x)$ where x is a rational number (whose numerator and denominator are assumed to be reduced). We first use trigonometric symmetries to reduce the argument to the octant $[0, 1/4]$. Then we either multiply by a numerical approximation of π and evaluate the trigonometric function the usual way, or we use algebraic methods, depending on which is estimated to be faster. Since the argument has been reduced to the first octant, the first of these two methods gives full accuracy even if the original argument is close to some root other the origin.

void **arb_tan_pi**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)
Sets $y = \tan(\pi x)$.

void **arb_cot_pi**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)
Sets $y = \cot(\pi x)$.

void **arb_sinc**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \operatorname{sinc}(x) = \sin(x)/x$.

4.1.13 Inverse trigonometric functions

void **arb_atan_arf**(*arb_t* *z*, const *arf_t* *x*, *slong* *prec*)

void **arb_atan**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \operatorname{atan}(x)$.

At low to medium precision (up to about 4096 bits), *arb_atan_arf()* uses table-based argument reduction and fast Taylor series evaluation via *_arb_atan_taylor_rs()*. At high precision, it falls back to MPFR. The function *arb_atan()* simply calls *arb_atan_arf()* with the midpoint as input, and separately adds the propagated error.

The function *arb_atan_arf()* uses lookup tables if possible, and otherwise falls back to *arb_atan_arf_bb()*.

void **arb_atan2**(*arb_t* *z*, const *arb_t* *b*, const *arb_t* *a*, *slong* *prec*)
Sets r to an the argument (phase) of the complex number $a + bi$, with the branch cut discontinuity on $(-\infty, 0]$. We define $\operatorname{atan2}(0, 0) = 0$, and for $a < 0$, $\operatorname{atan2}(0, a) = \pi$.

void **arb_asin**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \operatorname{asin}(x) = \operatorname{atan}(x/\sqrt{1 - x^2})$. If x is not contained in the domain $[-1, 1]$, the result is an indeterminate interval.

void **arb_acos**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \operatorname{acos}(x) = \pi/2 - \operatorname{asin}(x)$. If x is not contained in the domain $[-1, 1]$, the result is an indeterminate interval.

4.1.14 Hyperbolic functions

void **arb_sinh**(*arb_t* *s*, const *arb_t* *x*, *slong* *prec*)

void **arb_cosh**(*arb_t* *c*, const *arb_t* *x*, *slong* *prec*)

void **arb_sinh_cosh**(*arb_t* *s*, *arb_t* *c*, const *arb_t* *x*, *slong* *prec*)
Sets $s = \sinh(x)$, $c = \cosh(x)$. If the midpoint of x is close to zero and the hyperbolic sine is to be computed, evaluates $(e^{2x} \pm 1)/(2e^x)$ via *arb_expm1()* to avoid loss of accuracy. Otherwise evaluates $(e^x \pm e^{-x})/2$.

void **arb_tanh**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)
Sets $y = \tanh(x) = \sinh(x)/\cosh(x)$, evaluated via *arb_expm1()* as $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$ if $|x|$ is small, and as $\tanh(\pm x) = 1 - 2e^{\mp 2x}/(1 + e^{\mp 2x})$ if $|x|$ is large.

void **arb_coth**(*arb_t* *y*, const *arb_t* *x*, *slong* *prec*)
Sets $y = \coth(x) = \cosh(x)/\sinh(x)$, evaluated using the same strategy as *arb_tanh()*.

4.1.15 Inverse hyperbolic functions

void **arb_atanh**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \operatorname{atanh}(x)$.

void **arb_asinh**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \operatorname{asinh}(x)$.

void **arb_acosh**(*arb_t* *z*, const *arb_t* *x*, *slong* *prec*)
Sets $z = \operatorname{acosh}(x)$. If $x < 1$, the result is an indeterminate interval.

4.1.16 Constants

The following functions cache the computed values to speed up repeated calls at the same or lower precision. For further implementation details, see *Algorithms for mathematical constants*.

void **arb_const_pi**(*arb_t* *z*, *slong* *prec*)
Computes π.

void **arb_const_sqrt_pi**(*arb_t* *z*, *slong* *prec*)
Computes $\sqrt{\pi}$.

void **arb_const_log_sqrt2pi**(*arb_t* *z*, *slong* *prec*)
Computes $\log \sqrt{2\pi}$.

void **arb_const_log2**(*arb_t* *z*, *slong* *prec*)
Computes $\log(2)$.

void **arb_const_log10**(*arb_t* *z*, *slong* *prec*)
Computes $\log(10)$.

void **arb_const_euler**(*arb_t* *z*, *slong* *prec*)
Computes Euler's constant $\gamma = \lim_{k\to\infty}(H_k - \log k)$ where $H_k = 1 + 1/2 + \ldots + 1/k$.

void **arb_const_catalan**(*arb_t* *z*, *slong* *prec*)
Computes Catalan's constant $C = \sum_{n=0}^{\infty}(-1)^n/(2n+1)^2$.

void **arb_const_e**(*arb_t* *z*, *slong* *prec*)
Computes $e = \exp(1)$.

void **arb_const_khinchin**(*arb_t* *z*, *slong* *prec*)
Computes Khinchin's constant K_0.

void **arb_const_glaisher**(*arb_t* *z*, *slong* *prec*)
Computes the Glaisher-Kinkelin constant $A = \exp(1/12 - \zeta'(-1))$.

void **arb_const_apery**(*arb_t* *z*, *slong* *prec*)
Computes Apery's constant $\zeta(3)$.

4.1.17 Gamma function and factorials

void **arb_rising_ui_bs**(*arb_t z*, const *arb_t x*, *ulong n*, *slong prec*)

void **arb_rising_ui_rs**(*arb_t z*, const *arb_t x*, *ulong n*, *ulong step*, *slong prec*)

void **arb_rising_ui_rec**(*arb_t z*, const *arb_t x*, *ulong n*, *slong prec*)

void **arb_rising_ui**(*arb_t z*, const *arb_t x*, *ulong n*, *slong prec*)

void **arb_rising**(*arb_t z*, const *arb_t x*, const *arb_t n*, *slong prec*)
Computes the rising factorial $z = x(x+1)(x+2)\cdots(x+n-1)$.

The *bs* version uses binary splitting. The *rs* version uses rectangular splitting. The *rec* version uses either *bs* or *rs* depending on the input. The default version uses the gamma function unless n is a small integer.

The *rs* version takes an optional *step* parameter for tuning purposes (to use the default step length, pass zero).

void **arb_rising_fmpq_ui**(*arb_t z*, const *fmpq_t x*, *ulong n*, *slong prec*)
Computes the rising factorial $z = x(x+1)(x+2)\cdots(x+n-1)$ using binary splitting. If the denominator or numerator of x is large compared to *prec*, it is more efficient to convert x to an approximation and use *arb_rising_ui()*.

void **arb_rising2_ui_bs**(*arb_t u*, *arb_t v*, const *arb_t x*, *ulong n*, *slong prec*)

void **arb_rising2_ui_rs**(*arb_t u*, *arb_t v*, const *arb_t x*, *ulong n*, *ulong step*, *slong prec*)

void **arb_rising2_ui**(*arb_t u*, *arb_t v*, const *arb_t x*, *ulong n*, *slong prec*)
Letting $u(x) = x(x+1)(x+2)\cdots(x+n-1)$, simultaneously compute $u(x)$ and $v(x) = u'(x)$, respectively using binary splitting, rectangular splitting (with optional nonzero step length *step* to override the default choice), and an automatic algorithm choice.

void **arb_fac_ui**(*arb_t z*, *ulong n*, *slong prec*)
Computes the factorial $z = n!$ via the gamma function.

void **arb_doublefac_ui**(*arb_t z*, *ulong n*, *slong prec*)
Computes the double factorial $z = n!!$ via the gamma function.

void **arb_bin_ui**(*arb_t z*, const *arb_t n*, *ulong k*, *slong prec*)

void **arb_bin_uiui**(*arb_t z*, *ulong n*, *ulong k*, *slong prec*)
Computes the binomial coefficient $z = \binom{n}{k}$, via the rising factorial as $\binom{n}{k} = (n-k+1)_k/k!$.

void **arb_gamma**(*arb_t z*, const *arb_t x*, *slong prec*)

void **arb_gamma_fmpq**(*arb_t z*, const *fmpq_t x*, *slong prec*)

void **arb_gamma_fmpz**(*arb_t z*, const *fmpz_t x*, *slong prec*)
Computes the gamma function $z = \Gamma(x)$.

void **arb_lgamma**(*arb_t z*, const *arb_t x*, *slong prec*)
Computes the logarithmic gamma function $z = \log\Gamma(x)$. The complex branch structure is assumed, so if $x \le 0$, the result is an indeterminate interval.

void **arb_rgamma**(*arb_t z*, const *arb_t x*, *slong prec*)
Computes the reciprocal gamma function $z = 1/\Gamma(x)$, avoiding division by zero at the poles of the gamma function.

void **arb_digamma**(*arb_t y*, const *arb_t x*, *slong prec*)
Computes the digamma function $z = \psi(x) = (\log\Gamma(x))' = \Gamma'(x)/\Gamma(x)$.

4.1.18 Zeta function

void **arb_zeta_ui_vec_borwein**(*arb_ptr z*, *ulong start*, *slong num*, *ulong step*, *slong prec*)
Evaluates $\zeta(s)$ at num consecutive integers s beginning with *start* and proceeding in increments of

step. Uses Borwein's formula (*[Bor2000]*, *[GS2003]*), implemented to support fast multi-evaluation (but also works well for a single s).

Requires start ≥ 2. For efficiency, the largest s should be at most about as large as *prec*. Arguments approaching *LONG_MAX* will cause overflows. One should therefore only use this function for s up to about *prec*, and then switch to the Euler product.

The algorithm for single s is basically identical to the one used in MPFR (see *[MPFR2012]* for a detailed description). In particular, we evaluate the sum backwards to avoid storing more than one d_k coefficient, and use integer arithmetic throughout since it is convenient and the terms turn out to be slightly larger than 2^{prec}. The only numerical error in the main loop comes from the division by k^s, which adds less than 1 unit of error per term. For fast multi-evaluation, we repeatedly divide by k^{step}. Each division reduces the input error and adds at most 1 unit of additional rounding error, so by induction, the error per term is always smaller than 2 units.

void **arb_zeta_ui_asymp**(*arb_t* *x*, *ulong* *s*, *slong* *prec*)

void **arb_zeta_ui_euler_product**(*arb_t* *z*, *ulong* *s*, *slong* *prec*)
Computes $\zeta(s)$ using the Euler product. This is fast only if s is large compared to the precision. Both methods are trivial wrappers for *_acb_dirichlet_euler_product_real_ui()*.

void **arb_zeta_ui_bernoulli**(*arb_t* *x*, *ulong* *s*, *slong* *prec*)
Computes $\zeta(s)$ for even s via the corresponding Bernoulli number.

void **arb_zeta_ui_borwein_bsplit**(*arb_t* *x*, *ulong* *s*, *slong* *prec*)
Computes $\zeta(s)$ for arbitrary $s \geq 2$ using a binary splitting implementation of Borwein's algorithm. This has quasilinear complexity with respect to the precision (assuming that s is fixed).

void **arb_zeta_ui_vec**(*arb_ptr* *x*, *ulong* *start*, *slong* *num*, *slong* *prec*)

void **arb_zeta_ui_vec_even**(*arb_ptr* *x*, *ulong* *start*, *slong* *num*, *slong* *prec*)

void **arb_zeta_ui_vec_odd**(*arb_ptr* *x*, *ulong* *start*, *slong* *num*, *slong* *prec*)
Computes $\zeta(s)$ at *num* consecutive integers (respectively *num* even or *num* odd integers) beginning with $s = \text{start} \geq 2$, automatically choosing an appropriate algorithm.

void **arb_zeta_ui**(*arb_t* *x*, *ulong* *s*, *slong* *prec*)
Computes $\zeta(s)$ for nonnegative integer $s \neq 1$, automatically choosing an appropriate algorithm. This function is intended for numerical evaluation of isolated zeta values; for multi-evaluation, the vector versions are more efficient.

void **arb_zeta**(*arb_t* *z*, const *arb_t* *s*, *slong* *prec*)
Sets z to the value of the Riemann zeta function $\zeta(s)$.

For computing derivatives with respect to s, use *arb_poly_zeta_series()*.

void **arb_hurwitz_zeta**(*arb_t* *z*, const *arb_t* *s*, const *arb_t* *a*, *slong* *prec*)
Sets z to the value of the Hurwitz zeta function $\zeta(s, a)$.

For computing derivatives with respect to s, use *arb_poly_zeta_series()*.

4.1.19 Bernoulli numbers and polynomials

void **arb_bernoulli_ui**(*arb_t* *b*, *ulong* *n*, *slong* *prec*)

void **arb_bernoulli_fmpz**(*arb_t* *b*, const *fmpz_t* *n*, *slong* *prec*)
Sets b to the numerical value of the Bernoulli number B_n approximated to *prec* bits.

The internal precision is increased automatically to give an accurate result. Note that, with huge *fmpz* input, the output will have a huge exponent and evaluation will accordingly be slower.

A single division from the exact fraction of B_n is used if this value is in the global cache or the exact numerator roughly is larger than *prec* bits. Otherwise, the Riemann zeta function is used (see *arb_bernoulli_ui_zeta()*).

This function reads B_n from the global cache if the number is already cached, but does not automatically extend the cache by itself.

void **arb_bernoulli_ui_zeta**(*arb_t b*, *ulong n*, *slong prec*)
Sets b to the numerical value of B_n accurate to *prec* bits, computed using the formula $B_{2n} = (-1)^{n+1} 2(2n)! \zeta(2n) / (2\pi)^n$.

To avoid potential infinite recursion, we explicitly call the Euler product implementation of the zeta function. This method will only give high accuracy if the precision is small enough compared to n for the Euler product to converge rapidly.

void **arb_bernoulli_poly_ui**(*arb_t res*, *ulong n*, const *arb_t x*, *slong prec*)
Sets *res* to the value of the Bernoulli polynomial $B_n(x)$.

Warning: this function is only fast if either n or x is a small integer.

This function reads Bernoulli numbers from the global cache if they are already cached, but does not automatically extend the cache by itself.

void **arb_power_sum_vec**(*arb_ptr res*, const *arb_t a*, const *arb_t b*, *slong len*, *slong prec*)
For n from 0 to *len* - 1, sets entry n in the output vector *res* to

$$S_n(a, b) = \frac{1}{n+1} \left(B_{n+1}(b) - B_{n+1}(a) \right)$$

where $B_n(x)$ is a Bernoulli polynomial. If a and b are integers and $b \geq a$, this is equivalent to

$$S_n(a, b) = \sum_{k=a}^{b-1} k^n.$$

The computation uses the generating function for Bernoulli polynomials.

4.1.20 Polylogarithms

void **arb_polylog**(*arb_t w*, const *arb_t s*, const *arb_t z*, *slong prec*)

void **arb_polylog_si**(*arb_t w*, *slong s*, const *arb_t z*, *slong prec*)
Sets w to the polylogarithm $\mathrm{Li}_s(z)$.

4.1.21 Other special functions

void **arb_fib_fmpz**(*arb_t z*, const *fmpz_t n*, *slong prec*)

void **arb_fib_ui**(*arb_t z*, *ulong n*, *slong prec*)
Computes the Fibonacci number F_n. Uses the binary squaring algorithm described in *[Tak2000]*. Provided that n is small enough, an exact Fibonacci number can be computed by setting the precision to *ARF_PREC_EXACT*.

void **arb_agm**(*arb_t z*, const *arb_t x*, const *arb_t y*, *slong prec*)
Sets z to the arithmetic-geometric mean of x and y.

void **arb_chebyshev_t_ui**(*arb_t a*, *ulong n*, const *arb_t x*, *slong prec*)

void **arb_chebyshev_u_ui**(*arb_t a*, *ulong n*, const *arb_t x*, *slong prec*)
Evaluates the Chebyshev polynomial of the first kind $a = T_n(x)$ or the Chebyshev polynomial of the second kind $a = U_n(x)$.

void **arb_chebyshev_t2_ui**(*arb_t a*, *arb_t b*, *ulong n*, const *arb_t x*, *slong prec*)

void **arb_chebyshev_u2_ui**(*arb_t a*, *arb_t b*, *ulong n*, const *arb_t x*, *slong prec*)
Simultaneously evaluates $a = T_n(x), b = T_{n-1}(x)$ or $a = U_n(x), b = U_{n-1}(x)$. Aliasing between a, b and x is not permitted.

void **arb_bell_sum_bsplit**(*arb_t res*, const *fmpz_t n*, const *fmpz_t a*, const *fmpz_t b*, const *fmpz_t mmag*, *slong prec*)

void **arb_bell_sum_taylor**(*arb_t res*, const *fmpz_t n*, const *fmpz_t a*, const *fmpz_t b*, const *fmpz_t mmag*, *slong prec*)

Helper functions for Bell numbers, evaluating the sum $\sum_{k=a}^{b-1} k^n/k!$. If *mmag* is non-NULL, it may be used to indicate that the target error tolerance should be $2^{mmag-prec}$.

void **arb_bell_fmpz**(*arb_t res*, const *fmpz_t n*, *slong prec*)

void **arb_bell_ui**(*arb_t res*, *ulong n*, *slong prec*)

Sets *res* to the Bell number B_n. If the number is too large to fit exactly in *prec* bits, a numerical approximation is computed efficiently.

The algorithm to compute Bell numbers, including error analysis, is described in detail in *[Joh2015]*.

void **arb_euler_number_fmpz**(*arb_t res*, const *fmpz_t n*, *slong prec*)

void **arb_euler_number_ui**(*arb_t res*, *ulong n*, *slong prec*)

Sets *res* to the Euler number E_n, which is defined by having the exponential generating function $1/\cosh(x)$.

The Euler product for the Dirichlet beta function (*_acb_dirichlet_euler_product_real_ui()*) is used to compute a numerical approximation. If *prec* is more than enough to represent the result exactly, the exact value is automatically determined from a lower-precision approximation.

void **arb_partitions_fmpz**(*arb_t res*, const *fmpz_t n*, *slong prec*)

void **arb_partitions_ui**(*arb_t res*, *ulong n*, *slong prec*)

Sets *res* to the partition function $p(n)$. When n is large and $\log_2 p(n)$ is more than twice *prec*, the leading term in the Hardy-Ramanujan asymptotic series is used together with an error bound. Otherwise, the exact value is computed and rounded.

4.1.22 Internals for computing elementary functions

void **_arb_atan_taylor_naive**(*mp_ptr y*, *mp_limb_t* * *error*, *mp_srcptr x*, *mp_size_t xn*, *ulong N*, int *alternating*)

void **_arb_atan_taylor_rs**(*mp_ptr y*, *mp_limb_t* * *error*, *mp_srcptr x*, *mp_size_t xn*, *ulong N*, int *alternating*)

Computes an approximation of $y = \sum_{k=0}^{N-1} x^{2k+1}/(2k+1)$ (if *alternating* is 0) or $y = \sum_{k=0}^{N-1} (-1)^k x^{2k+1}/(2k+1)$ (if *alternating* is 1). Used internally for computing arctangents and logarithms. The *naive* version uses the forward recurrence, and the *rs* version uses a division-avoiding rectangular splitting scheme.

Requires $N \le 255$, $0 \le x \le 1/16$, and *xn* positive. The input x and output y are fixed-point numbers with *xn* fractional limbs. A bound for the ulp error is written to *error*.

void **_arb_exp_taylor_naive**(*mp_ptr y*, *mp_limb_t* * *error*, *mp_srcptr x*, *mp_size_t xn*, *ulong N*)

void **_arb_exp_taylor_rs**(*mp_ptr y*, *mp_limb_t* * *error*, *mp_srcptr x*, *mp_size_t xn*, *ulong N*)

Computes an approximation of $y = \sum_{k=0}^{N-1} x^k/k!$. Used internally for computing exponentials. The *naive* version uses the forward recurrence, and the *rs* version uses a division-avoiding rectangular splitting scheme.

Requires $N \le 287$, $0 \le x \le 1/16$, and *xn* positive. The input x is a fixed-point number with *xn* fractional limbs, and the output y is a fixed-point number with *xn* fractional limbs plus one extra limb for the integer part of the result.

A bound for the ulp error is written to *error*.

void **_arb_sin_cos_taylor_naive**(*mp_ptr ysin*, *mp_ptr ycos*, *mp_limb_t* * *error*, *mp_srcptr x*, *mp_size_t xn*, *ulong N*)

void **_arb_sin_cos_taylor_rs**(*mp_ptr ysin*, *mp_ptr ycos*, *mp_limb_t* * *error*, *mp_srcptr x*, *mp_size_t xn*, *ulong N*, int *sinonly*, int *alternating*)

Computes approximations of $y_s = \sum_{k=0}^{N-1} (-1)^k x^{2k+1}/(2k+1)!$ and $y_c = \sum_{k=0}^{N-1} (-1)^k x^{2k}/(2k)!$.

Used internally for computing sines and cosines. The *naive* version uses the forward recurrence, and the *rs* version uses a division-avoiding rectangular splitting scheme.

Requires $N \le 143$, $0 \le x \le 1/16$, and *xn* positive. The input x and outputs *ysin*, *ycos* are fixed-point numbers with *xn* fractional limbs. A bound for the ulp error is written to *error*.

If *sinonly* is 1, only the sine is computed; if *sinonly* is 0 both the sine and cosine are computed. To compute sin and cos, *alternating* should be 1. If *alternating* is 0, the hyperbolic sine is computed (this is currently only intended to be used together with *sinonly*).

int **_arb_get_mpn_fixed_mod_log2**(*mp_ptr w*, *fmpz_t q*, *mp_limb_t * error*, const *arf_t x*, *mp_size_t wn*)

Attempts to write $w = x - q\log(2)$ with $0 \le w < \log(2)$, where w is a fixed-point number with *wn* limbs and ulp error *error*. Returns success.

int **_arb_get_mpn_fixed_mod_pi4**(*mp_ptr w*, *fmpz_t q*, int * *octant*, *mp_limb_t * error*, const *arf_t x*, *mp_size_t wn*)

Attempts to write $w = |x| - q\pi/4$ with $0 \le w < \pi/4$, where w is a fixed-point number with *wn* limbs and ulp error *error*. Returns success.

The value of q mod 8 is written to *octant*. The output variable q can be NULL, in which case the full value of q is not stored.

slong **_arb_exp_taylor_bound**(*slong mag*, *slong prec*)

Returns n such that $\left|\sum_{k=n}^{\infty} x^k/k!\right| \le 2^{-\text{prec}}$, assuming $|x| \le 2^{\text{mag}} \le 1/4$.

void **arb_exp_arf_bb**(*arb_t z*, const *arf_t x*, *slong prec*, int *m1*)

Computes the exponential function using the bit-burst algorithm. If *m1* is nonzero, the exponential function minus one is computed accurately.

Aborts if x is extremely small or large (where another algorithm should be used).

For large x, repeated halving is used. In fact, we always do argument reduction until $|x|$ is smaller than about 2^{-d} where $d \approx 16$ to speed up convergence. If $|x| \approx 2^m$, we thus need about $m + d$ squarings.

Computing log(2) costs roughly 100-200 multiplications, so is not usually worth the effort at very high precision. However, this function could be improved by using log(2) based reduction at precision low enough that the value can be assumed to be cached.

void **_arb_exp_sum_bs_simple**(*fmpz_t T*, *fmpz_t Q*, *mp_bitcnt_t * Qexp*, const *fmpz_t x*, *mp_bitcnt_t r*, *slong N*)

void **_arb_exp_sum_bs_powtab**(*fmpz_t T*, *fmpz_t Q*, *mp_bitcnt_t * Qexp*, const *fmpz_t x*, *mp_bitcnt_t r*, *slong N*)

Computes T, Q and *Qexp* such that $T/(Q2^{\text{Qexp}}) = \sum_{k=1}^{N}(x/2^r)^k/k!$ using binary splitting. Note that the sum is taken to N inclusive and omits the constant term.

The *powtab* version precomputes a table of powers of x, resulting in slightly higher memory usage but better speed. For best efficiency, N should have many trailing zero bits.

void **_arb_atan_sum_bs_simple**(*fmpz_t T*, *fmpz_t Q*, *mp_bitcnt_t * Qexp*, const *fmpz_t x*, *mp_bitcnt_t r*, *slong N*)

void **_arb_atan_sum_bs_powtab**(*fmpz_t T*, *fmpz_t Q*, *mp_bitcnt_t * Qexp*, const *fmpz_t x*, *mp_bitcnt_t r*, *slong N*)

Computes T, Q and *Qexp* such that $T/(Q2^{\text{Qexp}}) = \sum_{k=1}^{N}(-1)^k(x/2^r)^{2k}/(2k+1)$ using binary splitting. Note that the sum is taken to N inclusive, omits the linear term, and requires a final multiplication by $(x/2^r)$ to give the true series for atan.

The *powtab* version precomputes a table of powers of x, resulting in slightly higher memory usage but better speed. For best efficiency, N should have many trailing zero bits.

void **arb_atan_arf_bb**(*arb_t z*, const *arf_t x*, *slong prec*)

Computes the arctangent of x. Initially, the argument-halving formula

$$\operatorname{atan}(x) = 2\operatorname{atan}\left(\frac{x}{1+\sqrt{1+x^2}}\right)$$

is applied up to 8 times to get a small argument. Then a version of the bit-burst algorithm is used. The functional equation

$$\operatorname{atan}(x) = \operatorname{atan}(p/q) + \operatorname{atan}(w), \quad w = \frac{qx-p}{px+q}, \quad p = \lfloor qx \rfloor$$

is applied repeatedly instead of integrating a differential equation for the arctangent, as this appears to be more efficient.

4.1.23 Vector functions

void **_arb_vec_zero**(*arb_ptr vec*, *slong n*)
Sets all entries in *vec* to zero.

int **_arb_vec_is_zero**(*arb_srcptr vec*, *slong len*)
Returns nonzero iff all entries in x are zero.

int **_arb_vec_is_finite**(*arb_srcptr x*, *slong len*)
Returns nonzero iff all entries in x certainly are finite.

void **_arb_vec_set**(*arb_ptr res*, *arb_srcptr vec*, *slong len*)
Sets *res* to a copy of *vec*.

void **_arb_vec_set_round**(*arb_ptr res*, *arb_srcptr vec*, *slong len*, *slong prec*)
Sets *res* to a copy of *vec*, rounding each entry to *prec* bits.

void **_arb_vec_swap**(*arb_ptr vec1*, *arb_ptr vec2*, *slong len*)
Swaps the entries of *vec1* and *vec2*.

void **_arb_vec_neg**(*arb_ptr B*, *arb_srcptr A*, *slong n*)

void **_arb_vec_sub**(*arb_ptr C*, *arb_srcptr A*, *arb_srcptr B*, *slong n*, *slong prec*)

void **_arb_vec_add**(*arb_ptr C*, *arb_srcptr A*, *arb_srcptr B*, *slong n*, *slong prec*)

void **_arb_vec_scalar_mul**(*arb_ptr res*, *arb_srcptr vec*, *slong len*, const *arb_t c*, *slong prec*)

void **_arb_vec_scalar_div**(*arb_ptr res*, *arb_srcptr vec*, *slong len*, const *arb_t c*, *slong prec*)

void **_arb_vec_scalar_mul_fmpz**(*arb_ptr res*, *arb_srcptr vec*, *slong len*, const *fmpz_t c*, *slong prec*)

void **_arb_vec_scalar_mul_2exp_si**(*arb_ptr res*, *arb_srcptr src*, *slong len*, *slong c*)

void **_arb_vec_scalar_addmul**(*arb_ptr res*, *arb_srcptr vec*, *slong len*, const *arb_t c*, *slong prec*)
Performs the respective scalar operation elementwise.

void **_arb_vec_dot**(*arb_t res*, *arb_srcptr vec1*, *arb_srcptr vec2*, *slong len2*, *slong prec*)
Sets *res* to the dot product of *vec1* and *vec2*.

void **_arb_vec_get_mag**(*mag_t bound*, *arb_srcptr vec*, *slong len*, *slong prec*)
Sets *bound* to an upper bound for the entries in *vec*.

slong **_arb_vec_bits**(*arb_srcptr x*, *slong len*)
Returns the maximum of *arb_bits()* for all entries in *vec*.

void **_arb_vec_set_powers**(*arb_ptr xs*, const *arb_t x*, *slong len*, *slong prec*)
Sets *xs* to the powers $1, x, x^2, \ldots, x^{len-1}$.

void **_arb_vec_add_error_arf_vec**(*arb_ptr res*, arf_srcptr *err*, *slong len*)

void **_arb_vec_add_error_mag_vec**(*arb_ptr res*, mag_srcptr *err*, *slong len*)
Adds the magnitude of each entry in *err* to the radius of the corresponding entry in *res*.

void **_arb_vec_indeterminate**(*arb_ptr vec*, *slong len*)
Applies *arb_indeterminate()* elementwise.

void **_arb_vec_trim**(*arb_ptr res*, *arb_srcptr vec*, *slong len*)
Applies *arb_trim()* elementwise.

int **_arb_vec_get_unique_fmpz_vec**(fmpz * *res*, *arb_srcptr vec*, *slong len*)
Calls *arb_get_unique_fmpz()* elementwise and returns nonzero if all entries can be rounded uniquely to integers. If any entry in *vec* cannot be rounded uniquely to an integer, returns zero.

4.2 acb.h – complex numbers

An *acb_t* represents a complex number with error bounds. An *acb_t* consists of a pair of real number balls of type *arb_struct*, representing the real and imaginary part with separate error bounds.

An *acb_t* thus represents a rectangle $[m_1 - r_1, m_1 + r_1] + [m_2 - r_2, m_2 + r_2]i$ in the complex plane. This is used instead of a disk or square representation (consisting of a complex floating-point midpoint with a single radius), since it allows implementing many operations more conveniently by splitting into ball operations on the real and imaginary parts. It also allows tracking when complex numbers have an exact (for example exactly zero) real part and an inexact imaginary part, or vice versa.

The interface for the *acb_t* type is slightly less developed than that for the *arb_t* type. In many cases, the user can easily perform missing operations by directly manipulating the real and imaginary parts.

4.2.1 Types, macros and constants

acb_struct

acb_t
An *acb_struct* consists of a pair of *arb_struct*:s. An *acb_t* is defined as an array of length one of type *acb_struct*, permitting an *acb_t* to be passed by reference.

acb_ptr
Alias for `acb_struct *`, used for vectors of numbers.

acb_srcptr
Alias for `const acb_struct *`, used for vectors of numbers when passed as constant input to functions.

acb_realref(x)
Macro returning a pointer to the real part of x as an *arb_t*.

acb_imagref(x)
Macro returning a pointer to the imaginary part of x as an *arb_t*.

4.2.2 Memory management

void **acb_init**(*acb_t x*)
Initializes the variable x for use, and sets its value to zero.

void **acb_clear**(*acb_t x*)
Clears the variable x, freeing or recycling its allocated memory.

acb_ptr **_acb_vec_init**(*slong n*)
Returns a pointer to an array of n initialized *acb_struct*:s.

void **_acb_vec_clear**(*acb_ptr v*, *slong n*)
Clears an array of n initialized *acb_struct*:s.

4.2.3 Basic manipulation

int **acb_is_zero**(const *acb_t z*)
Returns nonzero iff z is zero.

int **acb_is_one**(const *acb_t z*)
Returns nonzero iff z is exactly 1.

int **acb_is_finite**(const *acb_t z*)
Returns nonzero iff z certainly is finite.

int **acb_is_exact**(const *acb_t z*)
Returns nonzero iff z is exact.

int **acb_is_int**(const *acb_t z*)
Returns nonzero iff z is an exact integer.

void **acb_zero**(*acb_t z*)

void **acb_one**(*acb_t z*)

void **acb_onei**(*acb_t z*)
Sets z respectively to 0, 1, $i = \sqrt{-1}$.

void **acb_set**(*acb_t z*, const *acb_t x*)

void **acb_set_ui**(*acb_t z*, *slong x*)

void **acb_set_si**(*acb_t z*, *slong x*)

void **acb_set_d**(*acb_t z*, double *x*)

void **acb_set_fmpz**(*acb_t z*, const *fmpz_t x*)

void **acb_set_arb**(*acb_t z*, const *arb_t c*)
Sets z to the value of x.

void **acb_set_si_si**(*acb_t z*, *slong x*, *slong y*)

void **acb_set_d_d**(*acb_t z*, double *x*, double *y*)

void **acb_set_fmpz_fmpz**(*acb_t z*, const *fmpz_t x*, const *fmpz_t y*)

void **acb_set_arb_arb**(*acb_t z*, const *arb_t x*, const *arb_t y*)
Sets the real and imaginary part of z to the values x and y respectively

void **acb_set_fmpq**(*acb_t z*, const *fmpq_t x*, *slong prec*)

void **acb_set_round**(*acb_t z*, const *acb_t x*, *slong prec*)

void **acb_set_round_fmpz**(*acb_t z*, const *fmpz_t x*, *slong prec*)

void **acb_set_round_arb**(*acb_t z*, const *arb_t x*, *slong prec*)
Sets z to x, rounded to *prec* bits.

void **acb_swap**(*acb_t z*, *acb_t x*)
Swaps z and x efficiently.

void **acb_add_error_mag**(*acb_t x*, const *mag_t err*)
Adds *err* to the error bounds of both the real and imaginary parts of x, modifying x in-place.

4.2.4 Input and output

void **acb_print**(const *acb_t x*)
Prints the internal representation of x.

void **acb_printd**(const *acb_t z*, *slong digits*)
Prints x in decimal. The printed value of the radius is not adjusted to compensate for the fact that the binary-to-decimal conversion of both the midpoint and the radius introduces additional error.

void **acb_fprint**(FILE * *file*, const *acb_t* *x*)
Prints the internal representation of x to the stream *file*.

void **acb_fprintd**(FILE * *file*, const *acb_t* *z*, *slong* *digits*)
Prints x in decimal to the stream *file*. The printed value of the radius is not adjusted to compensate for the fact that the binary-to-decimal conversion of both the midpoint and the radius introduces additional error.

4.2.5 Random number generation

void **acb_randtest**(*acb_t* *z*, flint_rand_t *state*, *slong* *prec*, *slong* *mag_bits*)
Generates a random complex number by generating separate random real and imaginary parts.

void **acb_randtest_special**(*acb_t* *z*, flint_rand_t *state*, *slong* *prec*, *slong* *mag_bits*)
Generates a random complex number by generating separate random real and imaginary parts. Also generates NaNs and infinities.

void **acb_randtest_precise**(*acb_t* *z*, flint_rand_t *state*, *slong* *prec*, *slong* *mag_bits*)
Generates a random complex number with precise real and imaginary parts.

void **acb_randtest_param**(*acb_t* *z*, flint_rand_t *state*, *slong* *prec*, *slong* *mag_bits*)
Generates a random complex number, with very high probability of generating integers and half-integers.

4.2.6 Precision and comparisons

int **acb_equal**(const *acb_t* *x*, const *acb_t* *y*)
Returns nonzero iff x and y are identical as sets, i.e. if the real and imaginary parts are equal as balls.

Note that this is not the same thing as testing whether both x and y certainly represent the same complex number, unless either x or y is exact (and neither contains NaN). To test whether both operands *might* represent the same mathematical quantity, use *acb_overlaps()* or *acb_contains()*, depending on the circumstance.

int **acb_equal_si**(const *acb_t* *x*, *slong* *y*)
Returns nonzero iff x is equal to the integer y.

int **acb_eq**(const *acb_t* *x*, const *acb_t* *y*)
Returns nonzero iff x and y are certainly equal, as determined by testing that *arb_eq()* holds for both the real and imaginary parts.

int **acb_ne**(const *acb_t* *x*, const *acb_t* *y*)
Returns nonzero iff x and y are certainly not equal, as determined by testing that *arb_ne()* holds for either the real or imaginary parts.

int **acb_overlaps**(const *acb_t* *x*, const *acb_t* *y*)
Returns nonzero iff x and y have some point in common.

void **acb_get_abs_ubound_arf**(*arf_t* *u*, const *acb_t* *z*, *slong* *prec*)
Sets u to an upper bound for the absolute value of z, computed using a working precision of *prec* bits.

void **acb_get_abs_lbound_arf**(*arf_t* *u*, const *acb_t* *z*, *slong* *prec*)
Sets u to a lower bound for the absolute value of z, computed using a working precision of *prec* bits.

void **acb_get_rad_ubound_arf**(*arf_t* *u*, const *acb_t* *z*, *slong* *prec*)
Sets u to an upper bound for the error radius of z (the value is currently not computed tightly).

void **acb_get_mag**(*mag_t* *u*, const *acb_t* *x*)
Sets u to an upper bound for the absolute value of x.

void **acb_get_mag_lower**(*mag_t* *u*, const *acb_t* *x*)
Sets u to a lower bound for the absolute value of x.

int **acb_contains_fmpq**(const *acb_t* *x*, const *fmpq_t* *y*)

int **acb_contains_fmpz**(const *acb_t* *x*, const *fmpz_t* *y*)

int **acb_contains**(const *acb_t* *x*, const *acb_t* *y*)
Returns nonzero iff y is contained in x.

int **acb_contains_zero**(const *acb_t* *x*)
Returns nonzero iff zero is contained in x.

int **acb_contains_int**(const *acb_t* *x*)
Returns nonzero iff the complex interval represented by x contains an integer.

slong **acb_rel_error_bits**(const *acb_t* *x*)
Returns the effective relative error of x measured in bits. This is computed as if calling *arb_rel_error_bits()* on the real ball whose midpoint is the larger out of the real and imaginary midpoints of x, and whose radius is the larger out of the real and imaginary radiuses of x.

slong **acb_rel_accuracy_bits**(const *acb_t* *x*)
Returns the effective relative accuracy of x measured in bits, equal to the negative of the return value from *acb_rel_error_bits()*.

slong **acb_bits**(const *acb_t* *x*)
Returns the maximum of *arb_bits* applied to the real and imaginary parts of x, i.e. the minimum precision sufficient to represent x exactly.

void **acb_indeterminate**(*acb_t* *x*)
Sets x to $[\text{NaN} \pm\infty] + [\text{NaN} \pm\infty]i$, representing an indeterminate result.

void **acb_trim**(*acb_t* *y*, const *acb_t* *x*)
Sets y to a a copy of x with both the real and imaginary parts trimmed (see *arb_trim()*).

int **acb_is_real**(const *acb_t* *x*)
Returns nonzero iff the imaginary part of x is zero. It does not test whether the real part of x also is finite.

int **acb_get_unique_fmpz**(*fmpz_t* *z*, const *acb_t* *x*)
If x contains a unique integer, sets z to that value and returns nonzero. Otherwise (if x represents no integers or more than one integer), returns zero.

4.2.7 Complex parts

void **acb_get_real**(*arb_t* *re*, const *acb_t* *z*)
Sets *re* to the real part of z.

void **acb_get_imag**(*arb_t* *im*, const *acb_t* *z*)
Sets *im* to the imaginary part of z.

void **acb_arg**(*arb_t* *r*, const *acb_t* *z*, *slong* *prec*)
Sets r to a real interval containing the complex argument (phase) of z. We define the complex argument have a discontinuity on $(-\infty, 0]$, with the special value $\arg(0) = 0$, and $\arg(a + 0i) = \pi$ for $a < 0$. Equivalently, if $z = a + bi$, the argument is given by $\operatorname{atan2}(b, a)$ (see *arb_atan2()*).

void **acb_abs**(*arb_t* *r*, const *acb_t* *z*, *slong* *prec*)
Sets r to the absolute value of z.

void **acb_sgn**(*acb_t* *r*, const *acb_t* *z*, *slong* *prec*)
Sets r to the complex sign of z, defined as 0 if z is exactly zero and the projection onto the unit circle $z/|z| = \exp(i \arg(z))$ otherwise.

void **acb_csgn**(*arb_t* *r*, const *acb_t* *z*)
Sets r to the extension of the real sign function taking the value 1 for z strictly in the right half

plane, -1 for z strictly in the left half plane, and the sign of the imaginary part when z is on the imaginary axis. Equivalently, $\operatorname{csgn}(z) = z/\sqrt{z^2}$ except that the value is 0 when z is exactly zero.

4.2.8 Arithmetic

void **acb_neg**(*acb_t* z, const *acb_t* x)
Sets z to the negation of x.

void **acb_conj**(*acb_t* z, const *acb_t* x)
Sets z to the complex conjugate of x.

void **acb_add_ui**(*acb_t* z, const *acb_t* x, *ulong* y, *slong* *prec*)

void **acb_add_si**(*acb_t* z, const *acb_t* x, *slong* y, *slong* *prec*)

void **acb_add_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* y, *slong* *prec*)

void **acb_add_arb**(*acb_t* z, const *acb_t* x, const *arb_t* y, *slong* *prec*)

void **acb_add**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* *prec*)
Sets z to the sum of x and y.

void **acb_sub_ui**(*acb_t* z, const *acb_t* x, *ulong* y, *slong* *prec*)

void **acb_sub_si**(*acb_t* z, const *acb_t* x, *slong* y, *slong* *prec*)

void **acb_sub_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* y, *slong* *prec*)

void **acb_sub_arb**(*acb_t* z, const *acb_t* x, const *arb_t* y, *slong* *prec*)

void **acb_sub**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* *prec*)
Sets z to the difference of x and y.

void **acb_mul_onei**(*acb_t* z, const *acb_t* x)
Sets z to x multiplied by the imaginary unit.

void **acb_div_onei**(*acb_t* z, const *acb_t* x)
Sets z to x divided by the imaginary unit.

void **acb_mul_ui**(*acb_t* z, const *acb_t* x, *ulong* y, *slong* *prec*)

void **acb_mul_si**(*acb_t* z, const *acb_t* x, *slong* y, *slong* *prec*)

void **acb_mul_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* y, *slong* *prec*)

void **acb_mul_arb**(*acb_t* z, const *acb_t* x, const *arb_t* y, *slong* *prec*)
Sets z to the product of x and y.

void **acb_mul**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* *prec*)
Sets z to the product of x and y. If at least one part of x or y is zero, the operations is reduced to two real multiplications. If x and y are the same pointers, they are assumed to represent the same mathematical quantity and the squaring formula is used.

void **acb_mul_2exp_si**(*acb_t* z, const *acb_t* x, *slong* e)

void **acb_mul_2exp_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* e)
Sets z to x multiplied by 2^e, without rounding.

void **acb_sqr**(*acb_t* z, const *acb_t* x, *slong* *prec*)
Sets z to x squared.

void **acb_cube**(*acb_t* z, const *acb_t* x, *slong* *prec*)
Sets z to x cubed, computed efficiently using two real squarings, two real multiplications, and scalar operations.

void **acb_addmul**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* *prec*)

void **acb_addmul_ui**(*acb_t* z, const *acb_t* x, *ulong* y, *slong* *prec*)

void **acb_addmul_si**(*acb_t* z, const *acb_t* x, *slong* y, *slong* *prec*)

void **acb_addmul_fmpz**(*acb_t z*, const *acb_t x*, const *fmpz_t y*, *slong prec*)

void **acb_addmul_arb**(*acb_t z*, const *acb_t x*, const *arb_t y*, *slong prec*)
Sets z to z plus the product of x and y.

void **acb_submul**(*acb_t z*, const *acb_t x*, const *acb_t y*, *slong prec*)

void **acb_submul_ui**(*acb_t z*, const *acb_t x*, *ulong y*, *slong prec*)

void **acb_submul_si**(*acb_t z*, const *acb_t x*, *slong y*, *slong prec*)

void **acb_submul_fmpz**(*acb_t z*, const *acb_t x*, const *fmpz_t y*, *slong prec*)

void **acb_submul_arb**(*acb_t z*, const *acb_t x*, const *arb_t y*, *slong prec*)
Sets z to z minus the product of x and y.

void **acb_inv**(*acb_t z*, const *acb_t x*, *slong prec*)
Sets z to the multiplicative inverse of x.

void **acb_div_ui**(*acb_t z*, const *acb_t x*, *ulong y*, *slong prec*)

void **acb_div_si**(*acb_t z*, const *acb_t x*, *slong y*, *slong prec*)

void **acb_div_fmpz**(*acb_t z*, const *acb_t x*, const *fmpz_t y*, *slong prec*)

void **acb_div_arb**(*acb_t z*, const *acb_t x*, const *arb_t y*, *slong prec*)

void **acb_div**(*acb_t z*, const *acb_t x*, const *acb_t y*, *slong prec*)
Sets z to the quotient of x and y.

4.2.9 Mathematical constants

void **acb_const_pi**(*acb_t y*, *slong prec*)
Sets y to the constant π.

4.2.10 Powers and roots

void **acb_sqrt**(*acb_t r*, const *acb_t z*, *slong prec*)
Sets r to the square root of z. If either the real or imaginary part is exactly zero, only a single real square root is needed. Generally, we use the formula $\sqrt{a+bi} = u/2 + ib/u, u = \sqrt{2(|a+bi|+a)}$, requiring two real square root extractions.

void **acb_rsqrt**(*acb_t r*, const *acb_t z*, *slong prec*)
Sets r to the reciprocal square root of z. If either the real or imaginary part is exactly zero, only a single real reciprocal square root is needed. Generally, we use the formula $1/\sqrt{a+bi} = ((a+r)-bi)/v, r = |a+bi|, v = \sqrt{r|a+bi+r|^2}$, requiring one real square root and one real reciprocal square root.

void **acb_quadratic_roots_fmpz**(*acb_t r1*, *acb_t r2*, const *fmpz_t a*, const *fmpz_t b*, const *fmpz_t c*, *slong prec*)
Sets *r1* and *r2* to the roots of the quadratic polynomial $ax^2 + bx + c$. Requires that a is nonzero. This function is implemented so that both roots are computed accurately even when direct use of the quadratic formula would lose accuracy.

void **acb_root_ui**(*acb_t r*, const *acb_t z*, *ulong k*, *slong prec*)
Sets r to the principal k-th root of z.

void **acb_pow_fmpz**(*acb_t y*, const *acb_t b*, const *fmpz_t e*, *slong prec*)

void **acb_pow_ui**(*acb_t y*, const *acb_t b*, *ulong e*, *slong prec*)

void **acb_pow_si**(*acb_t y*, const *acb_t b*, *slong e*, *slong prec*)
Sets $y = b^e$ using binary exponentiation (with an initial division if $e < 0$). Note that these functions can get slow if the exponent is extremely large (in such cases *acb_pow()* may be superior).

void **acb_pow_arb**(*acb_t z*, const *acb_t x*, const *arb_t y*, *slong prec*)

void **acb_pow**(*acb_t* *z*, const *acb_t* *x*, const *acb_t* *y*, *slong* *prec*)
Sets $z = x^y$, computed using binary exponentiation if y if a small exact integer, as $z = (x^{1/2})^{2y}$ if y is a small exact half-integer, and generally as $z = \exp(y \log x)$.

4.2.11 Exponentials and logarithms

void **acb_exp**(*acb_t* *y*, const *acb_t* *z*, *slong* *prec*)
Sets y to the exponential function of z, computed as $\exp(a+bi) = \exp(a)\,(\cos(b) + \sin(b)i)$.

void **acb_exp_pi_i**(*acb_t* *y*, const *acb_t* *z*, *slong* *prec*)
Sets y to $\exp(\pi i z)$.

void **acb_exp_invexp**(*acb_t* *s*, *acb_t* *t*, const *acb_t* *z*, *slong* *prec*)
Sets $v = \exp(z)$ and $w = \exp(-z)$.

void **acb_log**(*acb_t* *y*, const *acb_t* *z*, *slong* *prec*)
Sets y to the principal branch of the natural logarithm of z, computed as $\log(a+bi) = \frac{1}{2}\log(a^2 + b^2) + i \arg(a+bi)$.

void **acb_log1p**(*acb_t* *z*, const *acb_t* *x*, *slong* *prec*)
Sets $z = \log(1+x)$, computed accurately when $x \approx 0$.

4.2.12 Trigonometric functions

void **acb_sin**(*acb_t* *s*, const *acb_t* *z*, *slong* *prec*)

void **acb_cos**(*acb_t* *c*, const *acb_t* *z*, *slong* *prec*)

void **acb_sin_cos**(*acb_t* *s*, *acb_t* *c*, const *acb_t* *z*, *slong* *prec*)
Sets $s = \sin(z)$, $c = \cos(z)$, evaluated as $\sin(a+bi) = \sin(a)\cosh(b) + i\cos(a)\sinh(b)$, $\cos(a+bi) = \cos(a)\cosh(b) - i\sin(a)\sinh(b)$.

void **acb_tan**(*acb_t* *s*, const *acb_t* *z*, *slong* *prec*)
Sets $s = \tan(z) = \sin(z)/\cos(z)$. For large imaginary parts, the function is evaluated in a numerically stable way as $\pm i$ plus a decreasing exponential factor.

void **acb_cot**(*acb_t* *s*, const *acb_t* *z*, *slong* *prec*)
Sets $s = \cot(z) = \cos(z)/\sin(z)$. For large imaginary parts, the function is evaluated in a numerically stable way as $\pm i$ plus a decreasing exponential factor.

void **acb_sin_pi**(*acb_t* *s*, const *acb_t* *z*, *slong* *prec*)

void **acb_cos_pi**(*acb_t* *s*, const *acb_t* *z*, *slong* *prec*)

void **acb_sin_cos_pi**(*acb_t* *s*, *acb_t* *c*, const *acb_t* *z*, *slong* *prec*)
Sets $s = \sin(\pi z)$, $c = \cos(\pi z)$, evaluating the trigonometric factors of the real and imaginary part accurately via *arb_sin_cos_pi()*.

void **acb_tan_pi**(*acb_t* *s*, const *acb_t* *z*, *slong* *prec*)
Sets $s = \tan(\pi z)$. Uses the same algorithm as *acb_tan()*, but evaluates the sine and cosine accurately via *arb_sin_cos_pi()*.

void **acb_cot_pi**(*acb_t* *s*, const *acb_t* *z*, *slong* *prec*)
Sets $s = \cot(\pi z)$. Uses the same algorithm as *acb_cot()*, but evaluates the sine and cosine accurately via *arb_sin_cos_pi()*.

void **acb_sinc**(*acb_t* *s*, const *acb_t* *z*, *slong* *prec*)
Sets $s = \operatorname{sinc}(x) = \sin(z)/z$.

4.2.13 Inverse trigonometric functions

void **acb_asin**(*acb_t* *res*, const *acb_t* *z*, *slong* *prec*)
Sets *res* to $\operatorname{asin}(z) = -i \log(iz + \sqrt{1-z^2})$.

void **acb_acos**(*acb_t res*, const *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{acos}(z) = \frac{1}{2}\pi - \operatorname{asin}(z)$.

void **acb_atan**(*acb_t res*, const *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{atan}(z) = \frac{1}{2}i(\log(1 - iz) - \log(1 + iz))$.

4.2.14 Hyperbolic functions

void **acb_sinh**(*acb_t s*, const *acb_t z*, *slong prec*)

void **acb_cosh**(*acb_t c*, const *acb_t z*, *slong prec*)

void **acb_sinh_cosh**(*acb_t s*, *acb_t c*, const *acb_t z*, *slong prec*)

void **acb_tanh**(*acb_t s*, const *acb_t z*, *slong prec*)

void **acb_coth**(*acb_t s*, const *acb_t z*, *slong prec*)
Respectively computes $\sinh(z) = -i\sin(iz)$, $\cosh(z) = \cos(iz)$, $\tanh(z) = -i\tan(iz)$, $\coth(z) = i\cot(iz)$.

4.2.15 Inverse hyperbolic functions

void **acb_asinh**(*acb_t res*, const *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{asinh}(z) = -i\operatorname{asin}(iz)$.

void **acb_acosh**(*acb_t res*, const *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{acosh}(z) = \log(z + \sqrt{z+1}\sqrt{z-1})$.

void **acb_atanh**(*acb_t res*, const *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{atanh}(z) = -i\operatorname{atan}(iz)$.

4.2.16 Rising factorials

void **acb_rising_ui_bs**(*acb_t z*, const *acb_t x*, *ulong n*, *slong prec*)

void **acb_rising_ui_rs**(*acb_t z*, const *acb_t x*, *ulong n*, *ulong step*, *slong prec*)

void **acb_rising_ui_rec**(*acb_t z*, const *acb_t x*, *ulong n*, *slong prec*)

void **acb_rising_ui**(*acb_t z*, const *acb_t x*, *ulong n*, *slong prec*)

void **acb_rising**(*acb_t z*, const *acb_t x*, const *acb_t n*, *slong prec*)
Computes the rising factorial $z = x(x+1)(x+2)\cdots(x+n-1)$.

The *bs* version uses binary splitting. The *rs* version uses rectangular splitting. The *rec* version uses either *bs* or *rs* depending on the input. The default version uses the gamma function unless n is a small integer.

The *rs* version takes an optional *step* parameter for tuning purposes (to use the default step length, pass zero).

void **acb_rising2_ui_bs**(*acb_t u*, *acb_t v*, const *acb_t x*, *ulong n*, *slong prec*)

void **acb_rising2_ui_rs**(*acb_t u*, *acb_t v*, const *acb_t x*, *ulong n*, *ulong step*, *slong prec*)

void **acb_rising2_ui**(*acb_t u*, *acb_t v*, const *acb_t x*, *ulong n*, *slong prec*)
Letting $u(x) = x(x+1)(x+2)\cdots(x+n-1)$, simultaneously compute $u(x)$ and $v(x) = u'(x)$, respectively using binary splitting, rectangular splitting (with optional nonzero step length *step* to override the default choice), and an automatic algorithm choice.

void **acb_rising_ui_get_mag**(*mag_t bound*, const *acb_t x*, *ulong n*)
Computes an upper bound for the absolute value of the rising factorial $z = x(x+1)(x+2)\cdots(x+n-1)$. Not currently optimized for large n.

4.2.17 Gamma function

void **acb_gamma**(*acb_t* *y*, const *acb_t* *x*, *slong* *prec*)
Computes the gamma function $y = \Gamma(x)$.

void **acb_rgamma**(*acb_t* *y*, const *acb_t* *x*, *slong* *prec*)
Computes the reciprocal gamma function $y = 1/\Gamma(x)$, avoiding division by zero at the poles of the gamma function.

void **acb_lgamma**(*acb_t* *y*, const *acb_t* *x*, *slong* *prec*)
Computes the logarithmic gamma function $y = \log \Gamma(x)$.

The branch cut of the logarithmic gamma function is placed on the negative half-axis, which means that $\log \Gamma(z)+\log z = \log \Gamma(z+1)$ holds for all z, whereas $\log \Gamma(z) \neq \log(\Gamma(z))$ in general. In the left half plane, the reflection formula with correct branch structure is evaluated via *acb_log_sin_pi()*.

void **acb_digamma**(*acb_t* *y*, const *acb_t* *x*, *slong* *prec*)
Computes the digamma function $y = \psi(x) = (\log \Gamma(x))' = \Gamma'(x)/\Gamma(x)$.

void **acb_log_sin_pi**(*acb_t* *res*, const *acb_t* *z*, *slong* *prec*)
Computes the logarithmic sine function defined by

$$S(z) = \log(\pi) - \log \Gamma(z) + \log \Gamma(1 - z)$$

which is equal to

$$S(z) = \int_{1/2}^{z} \pi \cot(\pi t) dt$$

where the path of integration goes through the upper half plane if $0 < \arg(z) \leq \pi$ and through the lower half plane if $-\pi < \arg(z) \leq 0$. Equivalently,

$$S(z) = \log(\sin(\pi(z - n))) \mp n\pi i, \quad n = \lfloor \operatorname{re}(z) \rfloor$$

where the negative sign is taken if $0 < \arg(z) \leq \pi$ and the positive sign is taken otherwise (if the interval $\arg(z)$ does not certainly satisfy either condition, the union of both cases is computed). After subtracting n, we have $0 \leq \operatorname{re}(z) < 1$. In this strip, we use use $S(z) = \log(\sin(\pi(z)))$ if the imaginary part of z is small. Otherwise, we use $S(z) = i\pi(z - 1/2) + \log((1 + e^{-2i\pi z})/2)$ in the lower half-plane and the conjugated expression in the upper half-plane to avoid exponent overflow.

The function is evaluated at the midpoint and the propagated error is computed from $S'(z)$ to get a continuous change when z is non-real and n spans more than one possible integer value.

void **acb_polygamma**(*acb_t* *z*, const *acb_t* *s*, const *acb_t* *z*, *slong* *prec*)
Sets *res* to the value of the generalized polygamma function $\psi(s, z)$.

If s is a nonnegative order, this is simply the s-order derivative of the digamma function. If $s = 0$, this function simply calls the digamma function internally. For integers $s \geq 1$, it calls the Hurwitz zeta function. Note that for small integers $s \geq 1$, it can be faster to use *acb_poly_digamma_series()* and read off the coefficients.

The generalization to other values of s is due to Espinosa and Moll *[EM2004]*:

$$\psi(s, z) = \frac{\zeta'(s + 1, z) + (\gamma + \psi(-s))\zeta(s + 1, z)}{\Gamma(-s)}$$

void **acb_barnes_g**(*acb_t* *res*, const *acb_t* *z*, *slong* *prec*)

void **acb_log_barnes_g**(*acb_t* *res*, const *acb_t* *z*, *slong* *prec*)
Computes Barnes G-function or the logarithmic Barnes G-function, respectively. The logarithmic version has branch cuts on the negative real axis and is continuous elsewhere in the complex plane, in analogy with the logarithmic gamma function. The functional equation

$$\log G(z + 1) = \log \Gamma(z) + \log G(z).$$

holds for all z.

For small integers, we directly use the recurrence relation $G(z+1) = \Gamma(z)G(z)$ together with the initial value $G(1) = 1$. For general z, we use the formula

$$\log G(z) = (z-1)\log\Gamma(z) - \zeta'(-1,z) + \zeta'(-1).$$

4.2.18 Zeta function

void **acb_zeta**(*acb_t* *z*, const *acb_t* *s*, *slong* *prec*)
Sets z to the value of the Riemann zeta function $\zeta(s)$. Note: for computing derivatives with respect to s, use *acb_poly_zeta_series()* or related methods.

void **acb_hurwitz_zeta**(*acb_t* *z*, const *acb_t* *s*, const *acb_t* *a*, *slong* *prec*)
Sets z to the value of the Hurwitz zeta function $\zeta(s,a)$. Note: for computing derivatives with respect to s, use *acb_poly_zeta_series()* or related methods.

void **acb_bernoulli_poly_ui**(*acb_t* *res*, *ulong* *n*, const *acb_t* *x*, *slong* *prec*)
Sets *res* to the value of the Bernoulli polynomial $B_n(x)$.

Warning: this function is only fast if either n or x is a small integer.

This function reads Bernoulli numbers from the global cache if they are already cached, but does not automatically extend the cache by itself.

4.2.19 Polylogarithms

void **acb_polylog**(*acb_t* *w*, const *acb_t* *s*, const *acb_t* *z*, *slong* *prec*)

void **acb_polylog_si**(*acb_t* *w*, *slong* *s*, const *acb_t* *z*, *slong* *prec*)
Sets w to the polylogarithm $\mathrm{Li}_s(z)$.

4.2.20 Arithmetic-geometric mean

See *Algorithms for the arithmetic-geometric mean* for implementation details.

void **acb_agm1**(*acb_t* *m*, const *acb_t* *z*, *slong* *prec*)
Sets m to the arithmetic-geometric mean $M(z) = \mathrm{agm}(1,z)$, defined such that the function is continuous in the complex plane except for a branch cut along the negative half axis (where it is continuous from above). This corresponds to always choosing an "optimal" branch for the square root in the arithmetic-geometric mean iteration.

void **acb_agm1_cpx**(*acb_ptr* *m*, const *acb_t* *z*, *slong* *len*, *slong* *prec*)
Sets the coefficients in the array m to the power series expansion of the arithmetic-geometric mean at the point z truncated to length *len*, i.e. $M(z+x) \in \mathbb{C}[[x]]$.

4.2.21 Other special functions

void **acb_chebyshev_t_ui**(*acb_t* *a*, *ulong* *n*, const *acb_t* *x*, *slong* *prec*)

void **acb_chebyshev_u_ui**(*acb_t* *a*, *ulong* *n*, const *acb_t* *x*, *slong* *prec*)
Evaluates the Chebyshev polynomial of the first kind $a = T_n(x)$ or the Chebyshev polynomial of the second kind $a = U_n(x)$.

void **acb_chebyshev_t2_ui**(*acb_t* *a*, *acb_t* *b*, *ulong* *n*, const *acb_t* *x*, *slong* *prec*)

void **acb_chebyshev_u2_ui**(*acb_t* *a*, *acb_t* *b*, *ulong* *n*, const *acb_t* *x*, *slong* *prec*)
Simultaneously evaluates $a = T_n(x), b = T_{n-1}(x)$ or $a = U_n(x), b = U_{n-1}(x)$. Aliasing between a, b and x is not permitted.

4.2.22 Vector functions

void **_acb_vec_zero**(*acb_ptr* *A*, *slong* *n*)
Sets all entries in *vec* to zero.

int **_acb_vec_is_zero**(*acb_srcptr* *vec*, *slong* *len*)
Returns nonzero iff all entries in x are zero.

int **_acb_vec_is_real**(*acb_srcptr* *v*, *slong* *len*)
Returns nonzero iff all entries in x have zero imaginary part.

void **_acb_vec_set**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*)
Sets *res* to a copy of *vec*.

void **_acb_vec_set_round**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, *slong* *prec*)
Sets *res* to a copy of *vec*, rounding each entry to *prec* bits.

void **_acb_vec_neg**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*)

void **_acb_vec_add**(*acb_ptr* *res*, *acb_srcptr* *vec1*, *acb_srcptr* *vec2*, *slong* *len*, *slong* *prec*)

void **_acb_vec_sub**(*acb_ptr* *res*, *acb_srcptr* *vec1*, *acb_srcptr* *vec2*, *slong* *len*, *slong* *prec*)

void **_acb_vec_scalar_submul**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, const *acb_t* *c*, *slong* *prec*)

void **_acb_vec_scalar_addmul**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, const *acb_t* *c*, *slong* *prec*)

void **_acb_vec_scalar_mul**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, const *acb_t* *c*, *slong* *prec*)

void **_acb_vec_scalar_mul_ui**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, *ulong* *c*, *slong* *prec*)

void **_acb_vec_scalar_mul_2exp_si**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, *slong* *c*)

void **_acb_vec_scalar_mul_onei**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*)

void **_acb_vec_scalar_div_ui**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, *ulong* *c*, *slong* *prec*)

void **_acb_vec_scalar_div**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, const *acb_t* *c*, *slong* *prec*)

void **_acb_vec_scalar_mul_arb**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, const *arb_t* *c*, *slong* *prec*)

void **_acb_vec_scalar_div_arb**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, const *arb_t* *c*, *slong* *prec*)

void **_acb_vec_scalar_mul_fmpz**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, const *fmpz_t* *c*, *slong* *prec*)

void **_acb_vec_scalar_div_fmpz**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*, const *fmpz_t* *c*, *slong* *prec*)
Performs the respective scalar operation elementwise.

slong **_acb_vec_bits**(*acb_srcptr* *vec*, *slong* *len*)
Returns the maximum of *arb_bits()* for all entries in *vec*.

void **_acb_vec_set_powers**(*acb_ptr* *xs*, const *acb_t* *x*, *slong* *len*, *slong* *prec*)
Sets *xs* to the powers $1, x, x^2, \ldots, x^{len-1}$.

void **_acb_vec_add_error_arf_vec**(*acb_ptr* *res*, arf_srcptr *err*, *slong* *len*)

void **_acb_vec_add_error_mag_vec**(*acb_ptr* *res*, mag_srcptr *err*, *slong* *len*)
Adds the magnitude of each entry in *err* to the radius of the corresponding entry in *res*.

void **_acb_vec_indeterminate**(*acb_ptr* *vec*, *slong* *len*)
Applies *acb_indeterminate()* elementwise.

void **_acb_vec_trim**(*acb_ptr* *res*, *acb_srcptr* *vec*, *slong* *len*)
Applies *acb_trim()* elementwise.

int **_acb_vec_get_unique_fmpz_vec**(fmpz * *res*, *acb_srcptr* *vec*, *slong* *len*)
Calls *acb_get_unique_fmpz()* elementwise and returns nonzero if all entries can be rounded uniquely to integers. If any entry in *vec* cannot be rounded uniquely to an integer, returns zero.

void **_acb_vec_sort_pretty**(*acb_ptr vec*, *slong len*)

Sorts the vector of complex numbers based on the real and imaginary parts. This is intended to reveal structure when printing a set of complex numbers, not to apply an order relation in a rigorous way.

CHAPTER

FIVE

POLYNOMIALS AND POWER SERIES

These modules implement dense univariate polynomials with real and complex coefficients. Truncated power series are supported via methods acting on polynomials, without introducing a separate power series type.

5.1 arb_poly.h – polynomials over the real numbers

An *arb_poly_t* represents a polynomial over the real numbers, implemented as an array of coefficients of type *arb_struct*.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

5.1.1 Types, macros and constants

arb_poly_struct

arb_poly_t
Contains a pointer to an array of coefficients (coeffs), the used length (length), and the allocated size of the array (alloc).

An *arb_poly_t* is defined as an array of length one of type *arb_poly_struct*, permitting an *arb_poly_t* to be passed by reference.

5.1.2 Memory management

void **arb_poly_init**(*arb_poly_t poly*)
Initializes the polynomial for use, setting it to the zero polynomial.

void **arb_poly_clear**(*arb_poly_t poly*)
Clears the polynomial, deallocating all coefficients and the coefficient array.

void **arb_poly_fit_length**(*arb_poly_t poly*, *slong len*)
Makes sure that the coefficient array of the polynomial contains at least *len* initialized coefficients.

void **_arb_poly_set_length**(*arb_poly_t poly*, *slong len*)
Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void **_arb_poly_normalise**(*arb_poly_t poly*)
Strips any trailing coefficients which are identical to zero.

5.1.3 Basic manipulation

slong **arb_poly_length**(const *arb_poly_t poly*)
Returns the length of *poly*, i.e. zero if *poly* is identically zero, and otherwise one more than the index of the highest term that is not identically zero.

slong **arb_poly_degree**(const *arb_poly_t poly*)
Returns the degree of *poly*, defined as one less than its length. Note that if one or several leading coefficients are balls containing zero, this value can be larger than the true degree of the exact polynomial represented by *poly*, so the return value of this function is effectively an upper bound.

int **arb_poly_is_zero**(const *arb_poly_t poly*)

int **arb_poly_is_one**(const *arb_poly_t poly*)

int **arb_poly_is_x**(const *arb_poly_t poly*)
Returns 1 if *poly* is exactly the polynomial 0, 1 or x respectively. Returns 0 otherwise.

void **arb_poly_zero**(*arb_poly_t poly*)

void **arb_poly_one**(*arb_poly_t poly*)
Sets *poly* to the constant 0 respectively 1.

void **arb_poly_set**(*arb_poly_t dest*, const *arb_poly_t src*)
Sets *dest* to a copy of *src*.

void **arb_poly_set_round**(*arb_poly_t dest*, const *arb_poly_t src*, *slong prec*)
Sets *dest* to a copy of *src*, rounded to *prec* bits.

void **arb_poly_set_coeff_si**(*arb_poly_t poly*, *slong n*, *slong c*)

void **arb_poly_set_coeff_arb**(*arb_poly_t poly*, *slong n*, const *arb_t c*)
Sets the coefficient with index n in *poly* to the value c. We require that n is nonnegative.

void **arb_poly_get_coeff_arb**(*arb_t v*, const *arb_poly_t poly*, *slong n*)
Sets v to the value of the coefficient with index n in *poly*. We require that n is nonnegative.

arb_poly_get_coeff_ptr(poly, n)
Given $n \geq 0$, returns a pointer to coefficient n of *poly*, or *NULL* if n exceeds the length of *poly*.

void **_arb_poly_shift_right**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong n*)

void **arb_poly_shift_right**(*arb_poly_t res*, const *arb_poly_t poly*, *slong n*)
Sets *res* to *poly* divided by x^n, throwing away the lower coefficients. We require that n is nonnegative.

void **_arb_poly_shift_left**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong n*)

void **arb_poly_shift_left**(*arb_poly_t res*, const *arb_poly_t poly*, *slong n*)
Sets *res* to *poly* multiplied by x^n. We require that n is nonnegative.

void **arb_poly_truncate**(*arb_poly_t poly*, *slong n*)
Truncates *poly* to have length at most n, i.e. degree strictly smaller than n.

5.1.4 Conversions

void **arb_poly_set_fmpz_poly**(*arb_poly_t poly*, const *fmpz_poly_t src*, *slong prec*)

void **arb_poly_set_fmpq_poly**(*arb_poly_t poly*, const *fmpq_poly_t src*, *slong prec*)

void **arb_poly_set_si**(*arb_poly_t poly*, *slong src*)
Sets *poly* to *src*, rounding the coefficients to *prec* bits.

5.1.5 Input and output

void **arb_poly_printd**(const *arb_poly_t poly*, *slong digits*)
Prints the polynomial as an array of coefficients, printing each coefficient using *arb_printd*.

void **arb_poly_fprintd**(FILE * *file*, const *arb_poly_t poly*, *slong digits*)
Prints the polynomial as an array of coefficients to the stream *file*, printing each coefficient using *arb_fprintd*.

5.1.6 Random generation

void **arb_poly_randtest**(*arb_poly_t poly*, flint_rand_t *state*, *slong len*, *slong prec*, *slong mag_bits*)
Creates a random polynomial with length at most *len*.

5.1.7 Comparisons

int **arb_poly_contains**(const *arb_poly_t poly1*, const *arb_poly_t poly2*)

int **arb_poly_contains_fmpz_poly**(const *arb_poly_t poly1*, const *fmpz_poly_t poly2*)

int **arb_poly_contains_fmpq_poly**(const *arb_poly_t poly1*, const *fmpq_poly_t poly2*)
Returns nonzero iff *poly1* contains *poly2*.

int **arb_poly_equal**(const *arb_poly_t A*, const *arb_poly_t B*)
Returns nonzero iff A and B are equal as polynomial balls, i.e. all coefficients have equal midpoint and radius.

int **_arb_poly_overlaps**(*arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*)

int **arb_poly_overlaps**(const *arb_poly_t poly1*, const *arb_poly_t poly2*)
Returns nonzero iff *poly1* overlaps with *poly2*. The underscore function requires that *len1* ist at least as large as *len2*.

int **arb_poly_get_unique_fmpz_poly**(*fmpz_poly_t z*, const *arb_poly_t x*)
If x contains a unique integer polynomial, sets z to that value and returns nonzero. Otherwise (if x represents no integers or more than one integer), returns zero, possibly partially modifying z.

5.1.8 Bounds

void **_arb_poly_majorant**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong prec*)

void **arb_poly_majorant**(*arb_poly_t res*, const *arb_poly_t poly*, *slong prec*)
Sets *res* to an exact real polynomial whose coefficients are upper bounds for the absolute values of the coefficients in *poly*, rounded to *prec* bits.

5.1.9 Arithmetic

void **_arb_poly_add**(*arb_ptr C*, *arb_srcptr A*, *slong lenA*, *arb_srcptr B*, *slong lenB*, *slong prec*)
Sets *{C, max(lenA, lenB)}* to the sum of *{A, lenA}* and *{B, lenB}*. Allows aliasing of the input and output operands.

void **arb_poly_add**(*arb_poly_t C*, const *arb_poly_t A*, const *arb_poly_t B*, *slong prec*)

void **arb_poly_add_si**(*arb_poly_t C*, const *arb_poly_t A*, *slong B*, *slong prec*)
Sets C to the sum of A and B.

void **_arb_poly_sub**(*arb_ptr C*, *arb_srcptr A*, *slong lenA*, *arb_srcptr B*, *slong lenB*, *slong prec*)
Sets *{C, max(lenA, lenB)}* to the difference of *{A, lenA}* and *{B, lenB}*. Allows aliasing of the input and output operands.

void **arb_poly_sub**(*arb_poly_t* *C*, const *arb_poly_t* *A*, const *arb_poly_t* *B*, *slong* *prec*)
Sets C to the difference of A and B.

void **arb_poly_neg**(*arb_poly_t* *C*, const *arb_poly_t* *A*)
Sets C to the negation of A.

void **arb_poly_scalar_mul_2exp_si**(*arb_poly_t* *C*, const *arb_poly_t* *A*, *slong* *c*)
Sets C to A multiplied by 2^c.

void **arb_poly_scalar_mul**(*arb_poly_t* *C*, const *arb_poly_t* *A*, const *arb_t* *c*, *slong* *prec*)
Sets C to A multiplied by c.

void **arb_poly_scalar_div**(*arb_poly_t* *C*, const *arb_poly_t* *A*, const *arb_t* *c*, *slong* *prec*)
Sets C to A divided by c.

void **_arb_poly_mullow_classical**(*arb_ptr* *C*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *n*, *slong* *prec*)

void **_arb_poly_mullow_block**(*arb_ptr* *C*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *n*, *slong* *prec*)

void **_arb_poly_mullow**(*arb_ptr* *C*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *n*, *slong* *prec*)
Sets *{C, n}* to the product of *{A, lenA}* and *{B, lenB}*, truncated to length n. The output is not allowed to be aliased with either of the inputs. We require $\text{lenA} \geq \text{lenB} > 0$, $n > 0$, $\text{lenA} + \text{lenB} - 1 \geq n$.

The *classical* version uses a plain loop. This has good numerical stability but gets slow for large n.

The *block* version decomposes the product into several subproducts which are computed exactly over the integers.

It first attempts to find an integer c such that $A(2^c x)$ and $B(2^c x)$ have slowly varying coefficients, to reduce the number of blocks.

The scaling factor c is chosen in a quick, heuristic way by picking the first and last nonzero terms in each polynomial. If the indices in A are a_2, a_1 and the log-2 magnitudes are e_2, e_1, and the indices in B are b_2, b_1 with corresponding magnitudes f_2, f_1, then we compute c as the weighted arithmetic mean of the slopes, rounded to the nearest integer:

$$c = \left\lfloor \frac{(e_2 - e_1) + (f_2 + f_1)}{(a_2 - a_1) + (b_2 - b_1)} + \frac{1}{2} \right\rfloor.$$

This strategy is used because it is simple. It is not optimal in all cases, but will typically give good performance when multiplying two power series with a similar decay rate.

The default algorithm chooses the *classical* algorithm for short polynomials and the *block* algorithm for long polynomials.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void **arb_poly_mullow_classical**(*arb_poly_t* *C*, const *arb_poly_t* *A*, const *arb_poly_t* *B*, *slong* *n*, *slong* *prec*)

void **arb_poly_mullow_ztrunc**(*arb_poly_t* *C*, const *arb_poly_t* *A*, const *arb_poly_t* *B*, *slong* *n*, *slong* *prec*)

void **arb_poly_mullow_block**(*arb_poly_t* *C*, const *arb_poly_t* *A*, const *arb_poly_t* *B*, *slong* *n*, *slong* *prec*)

void **arb_poly_mullow**(*arb_poly_t* *C*, const *arb_poly_t* *A*, const *arb_poly_t* *B*, *slong* *n*, *slong* *prec*)
Sets C to the product of A and B, truncated to length n. If the same variable is passed for A and B, sets C to the square of A truncated to length n.

void **_arb_poly_mul**(*arb_ptr* *C*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *prec*)
Sets *{C, lenA + lenB - 1}* to the product of *{A, lenA}* and *{B, lenB}*. The output is not allowed

to be aliased with either of the inputs. We require lenA $\geq$ lenB > 0. This function is implemented as a simple wrapper for *_arb_poly_mullow()*.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void **arb_poly_mul**(*arb_poly_t* *C*, const *arb_poly_t* *A*, const *arb_poly_t* *B*, *slong* *prec*)
Sets C to the product of A and B. If the same variable is passed for A and B, sets C to the square of A.

void **_arb_poly_inv_series**(*arb_ptr* *Q*, *arb_srcptr* *A*, *slong* *Alen*, *slong* *len*, *slong* *prec*)
Sets *{Q, len}* to the power series inverse of *{A, Alen}*. Uses Newton iteration.

void **arb_poly_inv_series**(*arb_poly_t* *Q*, const *arb_poly_t* *A*, *slong* *n*, *slong* *prec*)
Sets Q to the power series inverse of A, truncated to length n.

void **_arb_poly_div_series**(*arb_ptr* *Q*, *arb_srcptr* *A*, *slong* *Alen*, *arb_srcptr* *B*, *slong* *Blen*, *slong* *n*, *slong* *prec*)
Sets *{Q, n}* to the power series quotient of *{A, Alen}* by *{B, Blen}*. Uses Newton iteration followed by multiplication.

void **arb_poly_div_series**(*arb_poly_t* *Q*, const *arb_poly_t* *A*, const *arb_poly_t* *B*, *slong* *n*, *slong* *prec*)
Sets Q to the power series quotient A divided by B, truncated to length n.

void **_arb_poly_div**(*arb_ptr* *Q*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *prec*)

void **_arb_poly_rem**(*arb_ptr* *R*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *prec*)

void **_arb_poly_divrem**(*arb_ptr* *Q*, *arb_ptr* *R*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *prec*)

int **arb_poly_divrem**(*arb_poly_t* *Q*, *arb_poly_t* *R*, const *arb_poly_t* *A*, const *arb_poly_t* *B*, *slong* *prec*)
Performs polynomial division with remainder, computing a quotient Q and a remainder R such that $A = BQ + R$. The implementation reverses the inputs and performs power series division.

If the leading coefficient of B contains zero (or if B is identically zero), returns 0 indicating failure without modifying the outputs. Otherwise returns nonzero.

void **_arb_poly_div_root**(*arb_ptr* *Q*, *arb_t* *R*, *arb_srcptr* *A*, *slong* *len*, const *arb_t* *c*, *slong* *prec*)
Divides A by the polynomial $x - c$, computing the quotient Q as well as the remainder $R = f(c)$.

5.1.10 Composition

void **_arb_poly_taylor_shift_horner**(*arb_ptr* *g*, const *arb_t* *c*, *slong* *n*, *slong* *prec*)

void **arb_poly_taylor_shift_horner**(*arb_poly_t* *g*, const *arb_poly_t* *f*, const *arb_t* *c*, *slong* *prec*)

void **_arb_poly_taylor_shift_divconquer**(*arb_ptr* *g*, const *arb_t* *c*, *slong* *n*, *slong* *prec*)

void **arb_poly_taylor_shift_divconquer**(*arb_poly_t* *g*, const *arb_poly_t* *f*, const *arb_t* *c*, *slong* *prec*)

void **_arb_poly_taylor_shift_convolution**(*arb_ptr* *g*, const *arb_t* *c*, *slong* *n*, *slong* *prec*)

void **arb_poly_taylor_shift_convolution**(*arb_poly_t* *g*, const *arb_poly_t* *f*, const *arb_t* *c*, *slong* *prec*)

void **_arb_poly_taylor_shift**(*arb_ptr* *g*, const *arb_t* *c*, *slong* *n*, *slong* *prec*)

void **arb_poly_taylor_shift**(*arb_poly_t* *g*, const *arb_poly_t* *f*, const *arb_t* *c*, *slong* *prec*)
Sets g to the Taylor shift $f(x+c)$, computed respectively using an optimized form of Horner's rule, divide-and-conquer, a single convolution, and an automatic choice between the three algorithms.

The underscore methods act in-place on $g = f$ which has length n.

void **_arb_poly_compose_horner**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong prec*)

void **arb_poly_compose_horner**(*arb_poly_t res*, const *arb_poly_t poly1*, const *arb_poly_t poly2*, *slong prec*)

void **_arb_poly_compose_divconquer**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong prec*)

void **arb_poly_compose_divconquer**(*arb_poly_t res*, const *arb_poly_t poly1*, const *arb_poly_t poly2*, *slong prec*)

void **_arb_poly_compose**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong prec*)

void **arb_poly_compose**(*arb_poly_t res*, const *arb_poly_t poly1*, const *arb_poly_t poly2*, *slong prec*)

Sets *res* to the composition $h(x) = f(g(x))$ where f is given by *poly1* and g is given by *poly2*, respectively using Horner's rule, divide-and-conquer, and an automatic choice between the two algorithms.

The default algorithm also handles special-form input $g = ax^n + c$ efficiently by performing a Taylor shift followed by a rescaling.

The underscore methods do not support aliasing of the output with either input polynomial.

void **_arb_poly_compose_series_horner**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong n*, *slong prec*)

void **arb_poly_compose_series_horner**(*arb_poly_t res*, const *arb_poly_t poly1*, const *arb_poly_t poly2*, *slong n*, *slong prec*)

void **_arb_poly_compose_series_brent_kung**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong n*, *slong prec*)

void **arb_poly_compose_series_brent_kung**(*arb_poly_t res*, const *arb_poly_t poly1*, const *arb_poly_t poly2*, *slong n*, *slong prec*)

void **_arb_poly_compose_series**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong n*, *slong prec*)

void **arb_poly_compose_series**(*arb_poly_t res*, const *arb_poly_t poly1*, const *arb_poly_t poly2*, *slong n*, *slong prec*)

Sets *res* to the power series composition $h(x) = f(g(x))$ truncated to order $O(x^n)$ where f is given by *poly1* and g is given by *poly2*, respectively using Horner's rule, the Brent-Kung baby step-giant step algorithm, and an automatic choice between the two algorithms.

The default algorithm also handles special-form input $g = ax^n$ efficiently.

We require that the constant term in $g(x)$ is exactly zero. The underscore methods do not support aliasing of the output with either input polynomial.

void **_arb_poly_revert_series_lagrange**(*arb_ptr h*, *arb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **arb_poly_revert_series_lagrange**(*arb_poly_t h*, const *arb_poly_t f*, *slong n*, *slong prec*)

void **_arb_poly_revert_series_newton**(*arb_ptr h*, *arb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **arb_poly_revert_series_newton**(*arb_poly_t h*, const *arb_poly_t f*, *slong n*, *slong prec*)

void **_arb_poly_revert_series_lagrange_fast**(*arb_ptr h*, *arb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **arb_poly_revert_series_lagrange_fast**(*arb_poly_t h*, const *arb_poly_t f*, *slong n*, *slong prec*)

void **_arb_poly_revert_series**(*arb_ptr h*, *arb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **arb_poly_revert_series**(*arb_poly_t h*, const *arb_poly_t f*, *slong n*, *slong prec*)

Sets h to the power series reversion of f, i.e. the expansion of the compositional inverse function

$f^{-1}(x)$, truncated to order $O(x^n)$, using respectively Lagrange inversion, Newton iteration, fast Lagrange inversion, and a default algorithm choice.

We require that the constant term in f is exactly zero and that the linear term is nonzero. The underscore methods assume that *flen* is at least 2, and do not support aliasing.

5.1.11 Evaluation

void **_arb_poly_evaluate_horner**(*arb_t y*, *arb_srcptr f*, *slong len*, const *arb_t x*, *slong prec*)

void **arb_poly_evaluate_horner**(*arb_t y*, const *arb_poly_t f*, const *arb_t x*, *slong prec*)

void **_arb_poly_evaluate_rectangular**(*arb_t y*, *arb_srcptr f*, *slong len*, const *arb_t x*, *slong prec*)

void **arb_poly_evaluate_rectangular**(*arb_t y*, const *arb_poly_t f*, const *arb_t x*, *slong prec*)

void **_arb_poly_evaluate**(*arb_t y*, *arb_srcptr f*, *slong len*, const *arb_t x*, *slong prec*)

void **arb_poly_evaluate**(*arb_t y*, const *arb_poly_t f*, const *arb_t x*, *slong prec*)
Sets $y = f(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

void **_arb_poly_evaluate_acb_horner**(*acb_t y*, *arb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **arb_poly_evaluate_acb_horner**(*acb_t y*, const *arb_poly_t f*, const *acb_t x*, *slong prec*)

void **_arb_poly_evaluate_acb_rectangular**(*acb_t y*, *arb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **arb_poly_evaluate_acb_rectangular**(*acb_t y*, const *arb_poly_t f*, const *acb_t x*, *slong prec*)

void **_arb_poly_evaluate_acb**(*acb_t y*, *arb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **arb_poly_evaluate_acb**(*acb_t y*, const *arb_poly_t f*, const *acb_t x*, *slong prec*)
Sets $y = f(x)$ where x is a complex number, evaluating the polynomial respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

void **_arb_poly_evaluate2_horner**(*arb_t y*, *arb_t z*, *arb_srcptr f*, *slong len*, const *arb_t x*, *slong prec*)

void **arb_poly_evaluate2_horner**(*arb_t y*, *arb_t z*, const *arb_poly_t f*, const *arb_t x*, *slong prec*)

void **_arb_poly_evaluate2_rectangular**(*arb_t y*, *arb_t z*, *arb_srcptr f*, *slong len*, const *arb_t x*, *slong prec*)

void **arb_poly_evaluate2_rectangular**(*arb_t y*, *arb_t z*, const *arb_poly_t f*, const *arb_t x*, *slong prec*)

void **_arb_poly_evaluate2**(*arb_t y*, *arb_t z*, *arb_srcptr f*, *slong len*, const *arb_t x*, *slong prec*)

void **arb_poly_evaluate2**(*arb_t y*, *arb_t z*, const *arb_poly_t f*, const *arb_t x*, *slong prec*)
Sets $y = f(x), z = f'(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

When Horner's rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.

void **_arb_poly_evaluate2_acb_horner**(*acb_t y*, *acb_t z*, *arb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **arb_poly_evaluate2_acb_horner**(*acb_t y*, *acb_t z*, const *arb_poly_t f*, const *acb_t x*, *slong prec*)

void **_arb_poly_evaluate2_acb_rectangular**(*acb_t y*, *acb_t z*, *arb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **arb_poly_evaluate2_acb_rectangular**(*acb_t y*, *acb_t z*, const *arb_poly_t f*, const *acb_t x*, *slong prec*)

void **_arb_poly_evaluate2_acb**(*acb_t y*, *acb_t z*, *arb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **arb_poly_evaluate2_acb**(*acb_t y*, *acb_t z*, const *arb_poly_t f*, const *acb_t x*, *slong prec*)
Sets $y = f(x), z = f'(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

5.1.12 Product trees

void **_arb_poly_product_roots**(*arb_ptr poly*, *arb_srcptr xs*, *slong n*, *slong prec*)

void **arb_poly_product_roots**(*arb_poly_t poly*, *arb_srcptr xs*, *slong n*, *slong prec*)
Generates the polynomial $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$.

arb_ptr * **_arb_poly_tree_alloc**(*slong len*)
Returns an initialized data structured capable of representing a remainder tree (product tree) of *len* roots.

void **_arb_poly_tree_free**(*arb_ptr* * *tree*, *slong len*)
Deallocates a tree structure as allocated using *_arb_poly_tree_alloc*.

void **_arb_poly_tree_build**(*arb_ptr* * *tree*, *arb_srcptr roots*, *slong len*, *slong prec*)
Constructs a product tree from a given array of *len* roots. The tree structure must be pre-allocated to the specified length using *_arb_poly_tree_alloc()*.

5.1.13 Multipoint evaluation

void **_arb_poly_evaluate_vec_iter**(*arb_ptr ys*, *arb_srcptr poly*, *slong plen*, *arb_srcptr xs*, *slong n*, *slong prec*)

void **arb_poly_evaluate_vec_iter**(*arb_ptr ys*, const *arb_poly_t poly*, *arb_srcptr xs*, *slong n*, *slong prec*)
Evaluates the polynomial simultaneously at n given points, calling *_arb_poly_evaluate()* repeatedly.

void **_arb_poly_evaluate_vec_fast_precomp**(*arb_ptr vs*, *arb_srcptr poly*, *slong plen*, *arb_ptr* * *tree*, *slong len*, *slong prec*)

void **_arb_poly_evaluate_vec_fast**(*arb_ptr ys*, *arb_srcptr poly*, *slong plen*, *arb_srcptr xs*, *slong n*, *slong prec*)

void **arb_poly_evaluate_vec_fast**(*arb_ptr ys*, const *arb_poly_t poly*, *arb_srcptr xs*, *slong n*, *slong prec*)
Evaluates the polynomial simultaneously at n given points, using fast multipoint evaluation.

5.1.14 Interpolation

void **_arb_poly_interpolate_newton**(*arb_ptr poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong n*, *slong prec*)

void **arb_poly_interpolate_newton**(*arb_poly_t poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong n*, *slong prec*)
Recovers the unique polynomial of length at most n that interpolates the given x and y values. This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

void **_arb_poly_interpolate_barycentric**(*arb_ptr poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong n*, *slong prec*)

void **arb_poly_interpolate_barycentric**(*arb_poly_t poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong n*, *slong prec*)
Recovers the unique polynomial of length at most n that interpolates the given x and y values. This implementation uses the barycentric form of Lagrange interpolation.

void **_arb_poly_interpolation_weights**(*arb_ptr w*, *arb_ptr* * *tree*, *slong len*, *slong prec*)

void **_arb_poly_interpolate_fast_precomp**(*arb_ptr poly*, *arb_srcptr ys*, *arb_ptr* * *tree*, *arb_srcptr weights*, *slong len*, *slong prec*)

void **_arb_poly_interpolate_fast**(*arb_ptr poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong len*, *slong prec*)

void **arb_poly_interpolate_fast**(*arb_poly_t poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong n*, *slong prec*)
Recovers the unique polynomial of length at most n that interpolates the given x and y values, using fast Lagrange interpolation. The precomp function takes a precomputed product tree over the x values and a vector of interpolation weights as additional inputs.

5.1.15 Differentiation

void **_arb_poly_derivative**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong prec*)
Sets *{res, len - 1}* to the derivative of *{poly, len}*. Allows aliasing of the input and output.

void **arb_poly_derivative**(*arb_poly_t res*, const *arb_poly_t poly*, *slong prec*)
Sets *res* to the derivative of *poly*.

void **_arb_poly_integral**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong prec*)
Sets *{res, len}* to the integral of *{poly, len - 1}*. Allows aliasing of the input and output.

void **arb_poly_integral**(*arb_poly_t res*, const *arb_poly_t poly*, *slong prec*)
Sets *res* to the integral of *poly*.

5.1.16 Transforms

void **_arb_poly_borel_transform**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong prec*)

void **arb_poly_borel_transform**(*arb_poly_t res*, const *arb_poly_t poly*, *slong prec*)
Computes the Borel transform of the input polynomial, mapping $\sum_k a_k x^k$ to $\sum_k (a_k/k!) x^k$. The underscore method allows aliasing.

void **_arb_poly_inv_borel_transform**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong prec*)

void **arb_poly_inv_borel_transform**(*arb_poly_t res*, const *arb_poly_t poly*, *slong prec*)
Computes the inverse Borel transform of the input polynomial, mapping $\sum_k a_k x^k$ to $\sum_k a_k k! x^k$. The underscore method allows aliasing.

void **_arb_poly_binomial_transform_basecase**(*arb_ptr b*, *arb_srcptr a*, *slong alen*, *slong len*, *slong prec*)

void **arb_poly_binomial_transform_basecase**(*arb_poly_t b*, const *arb_poly_t a*, *slong len*, *slong prec*)

void **_arb_poly_binomial_transform_convolution**(*arb_ptr b*, *arb_srcptr a*, *slong alen*, *slong len*, *slong prec*)

void **arb_poly_binomial_transform_convolution**(*arb_poly_t b*, const *arb_poly_t a*, *slong len*, *slong prec*)

void **_arb_poly_binomial_transform**(*arb_ptr b*, *arb_srcptr a*, *slong alen*, *slong len*, *slong prec*)

void **arb_poly_binomial_transform**(*arb_poly_t b*, const *arb_poly_t a*, *slong len*, *slong prec*)
Computes the binomial transform of the input polynomial, truncating the output to length *len*. The binomial transform maps the coefficients a_k in the input polynomial to the coefficients b_k in the output polynomial via $b_n = \sum_{k=0}^{n} (-1)^k \binom{n}{k} a_k$. The binomial transform is equivalent to the power series composition $f(x) \to (1-x)^{-1} f(x/(x-1))$, and is its own inverse.

The *basecase* version evaluates coefficients one by one from the definition, generating the binomial coefficients by a recurrence relation.

The *convolution* version uses the identity $T(f(x)) = B^{-1}(e^x B(f(-x)))$ where T denotes the binomial transform operator and B denotes the Borel transform operator. This only costs a single polynomial multiplication, plus some scalar operations.

The default version automatically chooses an algorithm.

The underscore methods do not support aliasing, and assume that the lengths are nonzero.

5.1.17 Powers and elementary functions

void **_arb_poly_pow_ui_trunc_binexp**(*arb_ptr res*, *arb_srcptr f*, *slong flen*, *ulong exp*, *slong len*, *slong prec*)
Sets *{res, len}* to *{f, flen}* raised to the power *exp*, truncated to length *len*. Requires that *len* is no longer than the length of the power as computed without truncation (i.e. no zero-padding is performed). Does not support aliasing of the input and output, and requires that *flen* and *len* are positive. Uses binary expontiation.

void **arb_poly_pow_ui_trunc_binexp**(*arb_poly_t res*, const *arb_poly_t poly*, *ulong exp*, *slong len*, *slong prec*)
Sets *res* to *poly* raised to the power *exp*, truncated to length *len*. Uses binary exponentiation.

void **_arb_poly_pow_ui**(*arb_ptr res*, *arb_srcptr f*, *slong flen*, *ulong exp*, *slong prec*)
Sets *res* to *{f, flen}* raised to the power *exp*. Does not support aliasing of the input and output, and requires that *flen* is positive.

void **arb_poly_pow_ui**(*arb_poly_t res*, const *arb_poly_t poly*, *ulong exp*, *slong prec*)
Sets *res* to *poly* raised to the power *exp*.

void **_arb_poly_pow_series**(*arb_ptr h*, *arb_srcptr f*, *slong flen*, *arb_srcptr g*, *slong glen*, *slong len*, *slong prec*)
Sets *{h, len}* to the power series $f(x)^{g(x)} = \exp(g(x)\log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that *flen* and *glen* do not exceed *len*.

void **arb_poly_pow_series**(*arb_poly_t h*, const *arb_poly_t f*, const *arb_poly_t g*, *slong len*, *slong prec*)
Sets *h* to the power series $f(x)^{g(x)} = \exp(g(x)\log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently.

void **_arb_poly_pow_arb_series**(*arb_ptr h*, *arb_srcptr f*, *slong flen*, const *arb_t g*, *slong len*, *slong prec*)
Sets *{h, len}* to the power series $f(x)^g = \exp(g\log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that *flen* does not exceed *len*.

void **arb_poly_pow_arb_series**(*arb_poly_t h*, const *arb_poly_t f*, const *arb_t g*, *slong len*, *slong prec*)
Sets *h* to the power series $f(x)^g = \exp(g\log f(x))$ truncated to length *len*.

void **_arb_poly_sqrt_series**(*arb_ptr g*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_sqrt_series**(*arb_poly_t g*, const *arb_poly_t h*, *slong n*, *slong prec*)
Sets g to the power series square root of h, truncated to length n. Uses division-free Newton iteration for the reciprocal square root, followed by a multiplication.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and n are greater than zero.

void **_arb_poly_rsqrt_series**(*arb_ptr* *g*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_rsqrt_series**(*arb_poly_t* *g*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)
Sets g to the reciprocal power series square root of h, truncated to length n. Uses division-free Newton iteration.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and n are greater than zero.

void **_arb_poly_log_series**(*arb_ptr* *res*, *arb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **arb_poly_log_series**(*arb_poly_t* *res*, const *arb_poly_t* *f*, *slong* *n*, *slong* *prec*)
Sets *res* to the power series logarithm of f, truncated to length n. Uses the formula $\log(f(x)) = \int f'(x)/f(x)dx$, adding the logarithm of the constant term in f as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that *flen* and n are greater than zero.

void **_arb_poly_atan_series**(*arb_ptr* *res*, *arb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **arb_poly_atan_series**(*arb_poly_t* *res*, const *arb_poly_t* *f*, *slong* *n*, *slong* *prec*)

void **_arb_poly_asin_series**(*arb_ptr* *res*, *arb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **arb_poly_asin_series**(*arb_poly_t* *res*, const *arb_poly_t* *f*, *slong* *n*, *slong* *prec*)

void **_arb_poly_acos_series**(*arb_ptr* *res*, *arb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **arb_poly_acos_series**(*arb_poly_t* *res*, const *arb_poly_t* *f*, *slong* *n*, *slong* *prec*)
Sets *res* respectively to the power series inverse tangent, inverse sine and inverse cosine of f, truncated to length n.

Uses the formulas

$$\tan^{-1}(f(x)) = \int f'(x)/(1+f(x)^2)dx,$$
$$\sin^{-1}(f(x)) = \int f'(x)/(1-f(x)^2)^{1/2}dx,$$
$$\cos^{-1}(f(x)) = -\int f'(x)/(1-f(x)^2)^{1/2}dx,$$

adding the inverse function of the constant term in f as the constant of integration.

The underscore methods supports aliasing of the input and output arrays. They require that *flen* and n are greater than zero.

void **_arb_poly_exp_series_basecase**(*arb_ptr* *f*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_exp_series_basecase**(*arb_poly_t* *f*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_arb_poly_exp_series**(*arb_ptr* *f*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_exp_series**(*arb_poly_t* *f*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)
Sets f to the power series exponential of h, truncated to length n.

The basecase version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where m is the length of h.

The main implementation uses Newton iteration, starting from a small number of terms given by the basecase algorithm. The complexity is $O(M(n))$. Redundant operations in the Newton iteration are avoided by using the scheme described in *[HZ2004]*.

The underscore methods support aliasing and allow the input to be shorter than the output, but require the lengths to be nonzero.

void **_arb_poly_sin_cos_series_basecase**(*arb_ptr* *s*, *arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*, int *times_pi*)

void **arb_poly_sin_cos_series_basecase**(*arb_poly_t* *s*, *arb_poly_t* *c*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*, int *times_pi*)

void **_arb_poly_sin_cos_series_tangent**(*arb_ptr* *s*, *arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*, int *times_pi*)

void **arb_poly_sin_cos_series_tangent**(*arb_poly_t* *s*, *arb_poly_t* *c*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*, int *times_pi*)

void **_arb_poly_sin_cos_series**(*arb_ptr* *s*, *arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sin_cos_series**(*arb_poly_t* *s*, *arb_poly_t* *c*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Sets s and c to the power series sine and cosine of h, computed simultaneously.

The *basecase* version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where m is the length of h.

The *tangent* version uses the tangent half-angle formulas to compute the sine and cosine via *_arb_poly_tan_series()*. This requires $O(M(n))$ operations. When $h = h_0 + h_1$ where the constant term h_0 is nonzero, the evaluation is done as $\sin(h_0 + h_1) = \cos(h_0)\sin(h_1) + \sin(h_0)\cos(h_1)$, $\cos(h_0 + h_1) = \cos(h_0)\cos(h_1) - \sin(h_0)\sin(h_1)$, to improve accuracy and avoid dividing by zero at the poles of the tangent function.

The default version automatically selects between the *basecase* and *tangent* algorithms depending on the input.

The *basecase* and *tangent* versions take a flag *times_pi* specifying that the input is to be multiplied by π.

The underscore methods support aliasing and require the lengths to be nonzero.

void **_arb_poly_sin_series**(*arb_ptr* *s*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sin_series**(*arb_poly_t* *s*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_arb_poly_cos_series**(*arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_cos_series**(*arb_poly_t* *c*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Respectively evaluates the power series sine or cosine. These functions simply wrap *_arb_poly_sin_cos_series()*. The underscore methods support aliasing and require the lengths to be nonzero.

void **_arb_poly_tan_series**(*arb_ptr* *g*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *len*, *slong* *prec*)

void **arb_poly_tan_series**(*arb_poly_t* *g*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Sets g to the power series tangent of h.

For small n takes the quotient of the sine and cosine as computed using the basecase algorithm. For large n, uses Newton iteration to invert the inverse tangent series. The complexity is $O(M(n))$.

The underscore version does not support aliasing, and requires the lengths to be nonzero.

void **_arb_poly_sin_cos_pi_series**(*arb_ptr* *s*, *arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sin_cos_pi_series**(*arb_poly_t* *s*, *arb_poly_t* *c*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_arb_poly_sin_pi_series**(*arb_ptr* *s*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sin_pi_series**(*arb_poly_t* *s*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_arb_poly_cos_pi_series**(*arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_cos_pi_series**(*arb_poly_t* *c*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_arb_poly_cot_pi_series**(*arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_cot_pi_series**(*arb_poly_t* *c*, const *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Compute the respective trigonometric functions of the input multiplied by π.

void **_arb_poly_sinh_cosh_series_basecase**(*arb_ptr* *s*, *arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sinh_cosh_series_basecase**(*arb_poly_t s*, *arb_poly_t c*, const *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_sinh_cosh_series_exponential**(*arb_ptr s*, *arb_ptr c*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_sinh_cosh_series_exponential**(*arb_poly_t s*, *arb_poly_t c*, const *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_sinh_cosh_series**(*arb_ptr s*, *arb_ptr c*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_sinh_cosh_series**(*arb_poly_t s*, *arb_poly_t c*, const *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_sinh_series**(*arb_ptr s*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_sinh_series**(*arb_poly_t s*, const *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_cosh_series**(*arb_ptr c*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_cosh_series**(*arb_poly_t c*, const *arb_poly_t h*, *slong n*, *slong prec*)
Sets s and c respectively to the hyperbolic sine and cosine of the power series h, truncated to length n.

The implementations mirror those for sine and cosine, except that the *exponential* version computes both functions using the exponential function instead of the hyperbolic tangent.

void **_arb_poly_sinc_series**(*arb_ptr s*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_sinc_series**(*arb_poly_t s*, const *arb_poly_t h*, *slong n*, *slong prec*)
Sets c to the sinc function of the power series h, truncated to length n.

5.1.18 Gamma function and factorials

void **_arb_poly_gamma_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_gamma_series**(*arb_poly_t res*, const *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_rgamma_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_rgamma_series**(*arb_poly_t res*, const *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_lgamma_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_lgamma_series**(*arb_poly_t res*, const *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_digamma_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_digamma_series**(*arb_poly_t res*, const *arb_poly_t h*, *slong n*, *slong prec*)
Sets *res* to the series expansion of $\Gamma(h(x))$, $1/\Gamma(h(x))$, or $\log\Gamma(h(x))$, $\psi(h(x))$, truncated to length n.

These functions first generate the Taylor series at the constant term of h, and then call *_arb_poly_compose_series()*. The Taylor coefficients are generated using the Riemann zeta function if the constant term of h is a small integer, and with Stirling's series otherwise.

The underscore methods support aliasing of the input and output arrays, and require that *hlen* and n are greater than zero.

void **_arb_poly_rising_ui_series**(*arb_ptr res*, *arb_srcptr f*, *slong flen*, *ulong r*, *slong trunc*, *slong prec*)

void **arb_poly_rising_ui_series**(*arb_poly_t res*, const *arb_poly_t f*, *ulong r*, *slong trunc*, *slong prec*)
Sets *res* to the rising factorial $(f)(f+1)(f+2)\cdots(f+r-1)$, truncated to length *trunc*. The underscore method assumes that *flen*, r and *trunc* are at least 1, and does not support aliasing. Uses binary splitting.

5.1.19 Zeta function

void **arb_poly_zeta_series**(*arb_poly_t res*, const *arb_poly_t s*, const *arb_t a*, int *deflate*, *slong n*, *slong prec*)

Sets *res* to the Hurwitz zeta function $\zeta(s,a)$ where s a power series and a is a constant, truncated to length n. To evaluate the usual Riemann zeta function, set $a = 1$.

If *deflate* is nonzero, evaluates $\zeta(s,a)+1/(1-s)$, which is well-defined as a limit when the constant term of s is 1. In particular, expanding $\zeta(s,a)+1/(1-s)$ with $s = 1+x$ gives the Stieltjes constants

$$\sum_{k=0}^{n-1} \frac{(-1)^k}{k!} \gamma_k(a) x^k.$$

If $a = 1$, this implementation uses the reflection formula if the midpoint of the constant term of s is negative.

void **_arb_poly_riemann_siegel_theta_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_riemann_siegel_theta_series**(*arb_poly_t res*, const *arb_poly_t h*, *slong n*, *slong prec*)

Sets *res* to the series expansion of the Riemann-Siegel theta function

$$\theta(h) = \arg\left(\Gamma\left(\frac{2ih+1}{4}\right)\right) - \frac{\log \pi}{2} h$$

where the argument of the gamma function is chosen continuously as the imaginary part of the log gamma function.

The underscore method does not support aliasing of the input and output arrays, and requires that the lengths are greater than zero.

void **_arb_poly_riemann_siegel_z_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_riemann_siegel_z_series**(*arb_poly_t res*, const *arb_poly_t h*, *slong n*, *slong prec*)

Sets *res* to the series expansion of the Riemann-Siegel Z-function

$$Z(h) = e^{i\theta(h)} \zeta(1/2 + ih).$$

The zeros of the Z-function on the real line precisely correspond to the imaginary parts of the zeros of the Riemann zeta function on the critical line.

The underscore method supports aliasing of the input and output arrays, and requires that the lengths are greater than zero.

5.1.20 Root-finding

void **_arb_poly_root_bound_fujiwara**(*mag_t bound*, *arb_srcptr poly*, *slong len*)

void **arb_poly_root_bound_fujiwara**(*mag_t bound*, *arb_poly_t poly*)

Sets *bound* to an upper bound for the magnitude of all the complex roots of *poly*. Uses Fujiwara's bound

$$2 \max\left\{ \left|\frac{a_{n-1}}{a_n}\right|, \left|\frac{a_{n-2}}{a_n}\right|^{1/2}, \cdots, \left|\frac{a_1}{a_n}\right|^{1/(n-1)}, \left|\frac{a_0}{2a_n}\right|^{1/n} \right\}$$

where $a_0, \ldots, a_n$ are the coefficients of *poly*.

void **_arb_poly_newton_convergence_factor**(*arf_t convergence_factor*, *arb_srcptr poly*, *slong len*, const *arb_t convergence_interval*, *slong prec*)

Given an interval I specified by *convergence_interval*, evaluates a bound for C =

$\sup_{t,u \in I} \frac{1}{2}|f''(t)|/|f'(u)|$, where f is the polynomial defined by the coefficients *{poly, len}*. The bound is obtained by evaluating $f'(I)$ and $f''(I)$ directly. If f has large coefficients, I must be extremely precise in order to get a finite factor.

int **_arb_poly_newton_step**(*arb_t xnew*, *arb_srcptr poly*, *slong len*, const *arb_t x*, const *arb_t convergence_interval*, const *arf_t convergence_factor*, *slong prec*)

Performs a single step with Newton's method.

The input consists of the polynomial f specified by the coefficients *{poly, len}*, an interval $x = [m - r, m + r]$ known to contain a single root of f, an interval I (*convergence_interval*) containing x with an associated bound (*convergence_factor*) for $C = \sup_{t,u \in I} \frac{1}{2}|f''(t)|/|f'(u)|$, and a working precision *prec*.

The Newton update consists of setting $x' = [m' - r', m' + r']$ where $m' = m - f(m)/f'(m)$ and $r' = Cr^2$. The expression $m - f(m)/f'(m)$ is evaluated using ball arithmetic at a working precision of *prec* bits, and the rounding error during this evaluation is accounted for in the output. We now check that $x' \in I$ and $m' < m$. If both conditions are satisfied, we set *xnew* to x' and return nonzero. If either condition fails, we set *xnew* to x and return zero, indicating that no progress was made.

void **_arb_poly_newton_refine_root**(*arb_t r*, *arb_srcptr poly*, *slong len*, const *arb_t start*, const *arb_t convergence_interval*, const *arf_t convergence_factor*, *slong eval_extra_prec*, *slong prec*)

Refines a precise estimate of a polynomial root to high precision by performing several Newton steps, using nearly optimally chosen doubling precision steps.

The inputs are defined as for *_arb_poly_newton_step*, except for the precision parameters: *prec* is the target accuracy and *eval_extra_prec* is the estimated number of guard bits that need to be added to evaluate the polynomial accurately close to the root (typically, if the polynomial has large coefficients of alternating signs, this needs to be approximately the bit size of the coefficients).

5.1.21 Other special polynomials

void **_arb_poly_swinnerton_dyer_ui**(*arb_ptr poly*, *ulong n*, *slong trunc*, *slong prec*)

void **arb_poly_swinnerton_dyer_ui**(*arb_poly_t poly*, *ulong n*, *slong prec*)

Computes the Swinnerton-Dyer polynomial S_n, which has degree 2^n and is the rational minimal polynomial of the sum of the square roots of the first n prime numbers.

If *prec* is set to zero, a precision is chosen automatically such that *arb_poly_get_unique_fmpz_poly()* should be successful. Otherwise a working precision of *prec* bits is used.

The underscore version accepts an additional *trunc* parameter. Even when computing a truncated polynomial, the array *poly* must have room for $2^n + 1$ coefficients, used as temporary space.

5.2 acb_poly.h – polynomials over the complex numbers

An *acb_poly_t* represents a polynomial over the complex numbers, implemented as an array of coefficients of type *acb_struct*.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

5.2.1 Types, macros and constants

acb_poly_struct

acb_poly_t
Contains a pointer to an array of coefficients (coeffs), the used length (length), and the allocated size of the array (alloc).

An *acb_poly_t* is defined as an array of length one of type *acb_poly_struct*, permitting an *acb_poly_t* to be passed by reference.

5.2.2 Memory management

void **acb_poly_init**(*acb_poly_t poly*)
Initializes the polynomial for use, setting it to the zero polynomial.

void **acb_poly_clear**(*acb_poly_t poly*)
Clears the polynomial, deallocating all coefficients and the coefficient array.

void **acb_poly_fit_length**(*acb_poly_t poly*, *slong len*)
Makes sure that the coefficient array of the polynomial contains at least *len* initialized coefficients.

void **_acb_poly_set_length**(*acb_poly_t poly*, *slong len*)
Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void **_acb_poly_normalise**(*acb_poly_t poly*)
Strips any trailing coefficients which are identical to zero.

void **acb_poly_swap**(*acb_poly_t poly1*, *acb_poly_t poly2*)
Swaps *poly1* and *poly2* efficiently.

5.2.3 Basic properties and manipulation

slong **acb_poly_length**(const *acb_poly_t poly*)
Returns the length of *poly*, i.e. zero if *poly* is identically zero, and otherwise one more than the index of the highest term that is not identically zero.

slong **acb_poly_degree**(const *acb_poly_t poly*)
Returns the degree of *poly*, defined as one less than its length. Note that if one or several leading coefficients are balls containing zero, this value can be larger than the true degree of the exact polynomial represented by *poly*, so the return value of this function is effectively an upper bound.

int **acb_poly_is_zero**(const *acb_poly_t poly*)

int **acb_poly_is_one**(const *acb_poly_t poly*)

int **acb_poly_is_x**(const *acb_poly_t poly*)
Returns 1 if *poly* is exactly the polynomial 0, 1 or x respectively. Returns 0 otherwise.

void **acb_poly_zero**(*acb_poly_t poly*)
Sets *poly* to the zero polynomial.

void **acb_poly_one**(*acb_poly_t poly*)
Sets *poly* to the constant polynomial 1.

void **acb_poly_set**(*acb_poly_t dest*, const *acb_poly_t src*)
Sets *dest* to a copy of *src*.

void **acb_poly_set_round**(*acb_poly_t dest*, const *acb_poly_t src*, *slong prec*)
Sets *dest* to a copy of *src*, rounded to *prec* bits.

void **acb_poly_set_coeff_si**(*acb_poly_t poly*, *slong n*, *slong c*)

void **acb_poly_set_coeff_acb**(*acb_poly_t poly*, *slong n*, const *acb_t c*)
Sets the coefficient with index n in *poly* to the value c. We require that n is nonnegative.

void **acb_poly_get_coeff_acb**(*acb_t v*, const *acb_poly_t poly*, *slong n*)
Sets v to the value of the coefficient with index n in *poly*. We require that n is nonnegative.

acb_poly_get_coeff_ptr(poly, n)
Given $n \geq 0$, returns a pointer to coefficient n of *poly*, or *NULL* if n exceeds the length of *poly*.

void **_acb_poly_shift_right**(*acb_ptr* *res*, *acb_srcptr* *poly*, *slong* *len*, *slong* *n*)

void **acb_poly_shift_right**(*acb_poly_t* *res*, const *acb_poly_t* *poly*, *slong* *n*)
Sets *res* to *poly* divided by x^n, throwing away the lower coefficients. We require that n is nonnegative.

void **_acb_poly_shift_left**(*acb_ptr* *res*, *acb_srcptr* *poly*, *slong* *len*, *slong* *n*)

void **acb_poly_shift_left**(*acb_poly_t* *res*, const *acb_poly_t* *poly*, *slong* *n*)
Sets *res* to *poly* multiplied by x^n. We require that n is nonnegative.

void **acb_poly_truncate**(*acb_poly_t* *poly*, *slong* *n*)
Truncates *poly* to have length at most n, i.e. degree strictly smaller than n.

5.2.4 Input and output

void **acb_poly_printd**(const *acb_poly_t* *poly*, *slong* *digits*)
Prints the polynomial as an array of coefficients, printing each coefficient using *acb_printd*.

void **acb_poly_fprintd**(FILE * *file*, const *acb_poly_t* *poly*, *slong* *digits*)
Prints the polynomial as an array of coefficients to the stream *file*, printing each coefficient using *acb_fprintd*.

5.2.5 Random generation

void **acb_poly_randtest**(*acb_poly_t* *poly*, flint_rand_t *state*, *slong* *len*, *slong* *prec*, *slong* *mag_bits*)
Creates a random polynomial with length at most *len*.

5.2.6 Comparisons

int **acb_poly_equal**(const *acb_poly_t* *A*, const *acb_poly_t* *B*)
Returns nonzero iff A and B are identical as interval polynomials.

int **acb_poly_contains**(const *acb_poly_t* *poly1*, const *acb_poly_t* *poly2*)

int **acb_poly_contains_fmpz_poly**(const *acb_poly_t* *poly1*, const *fmpz_poly_t* *poly2*)

int **acb_poly_contains_fmpq_poly**(const *acb_poly_t* *poly1*, const *fmpq_poly_t* *poly2*)
Returns nonzero iff *poly2* is contained in *poly1*.

int **_acb_poly_overlaps**(*acb_srcptr* *poly1*, *slong* *len1*, *acb_srcptr* *poly2*, *slong* *len2*)

int **acb_poly_overlaps**(const *acb_poly_t* *poly1*, const *acb_poly_t* *poly2*)
Returns nonzero iff *poly1* overlaps with *poly2*. The underscore function requires that *len1* ist at least as large as *len2*.

int **acb_poly_get_unique_fmpz_poly**(*fmpz_poly_t* *z*, const *acb_poly_t* *x*)
If x contains a unique integer polynomial, sets z to that value and returns nonzero. Otherwise (if x represents no integers or more than one integer), returns zero, possibly partially modifying z.

int **acb_poly_is_real**(const *acb_poly_t* *poly*)
Returns nonzero iff all coefficients in *poly* have zero imaginary part.

5.2.7 Conversions

void **acb_poly_set_fmpz_poly**(*acb_poly_t* *poly*, const *fmpz_poly_t* *re*, *slong* *prec*)

void **acb_poly_set2_fmpz_poly**(*acb_poly_t poly*, const *fmpz_poly_t re*, const *fmpz_poly_t im*, *slong prec*)

void **acb_poly_set_arb_poly**(*acb_poly_t poly*, const *arb_poly_t re*)

void **acb_poly_set2_arb_poly**(*acb_poly_t poly*, const *arb_poly_t re*, const *arb_poly_t im*)

void **acb_poly_set_fmpq_poly**(*acb_poly_t poly*, const *fmpq_poly_t re*, *slong prec*)

void **acb_poly_set2_fmpq_poly**(*acb_poly_t poly*, const *fmpq_poly_t re*, const *fmpq_poly_t im*, *slong prec*)
Sets *poly* to the given real part *re* plus the imaginary part *im*, both rounded to *prec* bits.

void **acb_poly_set_acb**(*acb_poly_t poly*, *slong src*)

void **acb_poly_set_si**(*acb_poly_t poly*, *slong src*)
Sets *poly* to *src*.

5.2.8 Bounds

void **_acb_poly_majorant**(*arb_ptr res*, *acb_srcptr poly*, *slong len*, *slong prec*)

void **acb_poly_majorant**(*arb_poly_t res*, const *acb_poly_t poly*, *slong prec*)
Sets *res* to an exact real polynomial whose coefficients are upper bounds for the absolute values of the coefficients in *poly*, rounded to *prec* bits.

5.2.9 Arithmetic

void **_acb_poly_add**(*acb_ptr C*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)
Sets *{C, max(lenA, lenB)}* to the sum of *{A, lenA}* and *{B, lenB}*. Allows aliasing of the input and output operands.

void **acb_poly_add**(*acb_poly_t C*, const *acb_poly_t A*, const *acb_poly_t B*, *slong prec*)

void **acb_poly_add_si**(*acb_poly_t C*, const *acb_poly_t A*, *slong B*, *slong prec*)
Sets C to the sum of A and B.

void **_acb_poly_sub**(*acb_ptr C*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)
Sets *{C, max(lenA, lenB)}* to the difference of *{A, lenA}* and *{B, lenB}*. Allows aliasing of the input and output operands.

void **acb_poly_sub**(*acb_poly_t C*, const *acb_poly_t A*, const *acb_poly_t B*, *slong prec*)
Sets C to the difference of A and B.

void **acb_poly_neg**(*acb_poly_t C*, const *acb_poly_t A*)
Sets C to the negation of A.

void **acb_poly_scalar_mul_2exp_si**(*acb_poly_t C*, const *acb_poly_t A*, *slong c*)
Sets C to A multiplied by 2^c.

void **acb_poly_scalar_mul**(*acb_poly_t C*, const *acb_poly_t A*, const *acb_t c*, *slong prec*)
Sets C to A multiplied by c.

void **acb_poly_scalar_div**(*acb_poly_t C*, const *acb_poly_t A*, const *acb_t c*, *slong prec*)
Sets C to A divided by c.

void **_acb_poly_mullow_classical**(*acb_ptr C*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong n*, *slong prec*)

void **_acb_poly_mullow_transpose**(*acb_ptr C*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong n*, *slong prec*)

void **_acb_poly_mullow_transpose_gauss**(*acb_ptr C*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong n*, *slong prec*)

void **_acb_poly_mullow**(*acb_ptr C*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong n*, *slong prec*)

Sets *{C, n}* to the product of *{A, lenA}* and *{B, lenB}*, truncated to length n. The output is not allowed to be aliased with either of the inputs. We require $\text{lenA} \geq \text{lenB} > 0$, $n > 0$, $\text{lenA} + \text{lenB} - 1 \geq n$.

The *classical* version uses a plain loop.

The *transpose* version evaluates the product using four real polynomial multiplications (via *_arb_poly_mullow()*).

The *transpose_gauss* version evaluates the product using three real polynomial multiplications. This is almost always faster than *transpose*, but has worse numerical stability when the coefficients vary in magnitude.

The default function *_acb_poly_mullow()* automatically switches been *classical* and *transpose* multiplication.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void **acb_poly_mullow_classical**(*acb_poly_t C*, const *acb_poly_t A*, const *acb_poly_t B*, *slong n*, *slong prec*)

void **acb_poly_mullow_transpose**(*acb_poly_t C*, const *acb_poly_t A*, const *acb_poly_t B*, *slong n*, *slong prec*)

void **acb_poly_mullow_transpose_gauss**(*acb_poly_t C*, const *acb_poly_t A*, const *acb_poly_t B*, *slong n*, *slong prec*)

void **acb_poly_mullow**(*acb_poly_t C*, const *acb_poly_t A*, const *acb_poly_t B*, *slong n*, *slong prec*)

Sets C to the product of A and B, truncated to length n. If the same variable is passed for A and B, sets C to the square of A truncated to length n.

void **_acb_poly_mul**(*acb_ptr C*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)

Sets *{C, lenA + lenB - 1}* to the product of *{A, lenA}* and *{B, lenB}*. The output is not allowed to be aliased with either of the inputs. We require $\text{lenA} \geq \text{lenB} > 0$. This function is implemented as a simple wrapper for *_acb_poly_mullow()*.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void **acb_poly_mul**(*acb_poly_t C*, const *acb_poly_t A1*, const *acb_poly_t B2*, *slong prec*)

Sets C to the product of A and B. If the same variable is passed for A and B, sets C to the square of A.

void **_acb_poly_inv_series**(*acb_ptr Qinv*, *acb_srcptr Q*, *slong Qlen*, *slong len*, *slong prec*)

Sets *{Qinv, len}* to the power series inverse of *{Q, Qlen}*. Uses Newton iteration.

void **acb_poly_inv_series**(*acb_poly_t Qinv*, const *acb_poly_t Q*, *slong n*, *slong prec*)

Sets *Qinv* to the power series inverse of *Q*.

void **_acb_poly_div_series**(*acb_ptr Q*, *acb_srcptr A*, *slong Alen*, *acb_srcptr B*, *slong Blen*, *slong n*, *slong prec*)

Sets *{Q, n}* to the power series quotient of *{A, Alen}* by *{B, Blen}*. Uses Newton iteration followed by multiplication.

void **acb_poly_div_series**(*acb_poly_t Q*, const *acb_poly_t A*, const *acb_poly_t B*, *slong n*, *slong prec*)

Sets Q to the power series quotient A divided by B, truncated to length n.

void **_acb_poly_div**(*acb_ptr Q*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)

void **_acb_poly_rem**(*acb_ptr R*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)

void **_acb_poly_divrem**(*acb_ptr Q*, *acb_ptr R*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)

void **acb_poly_divrem**(*acb_poly_t Q*, *acb_poly_t R*, const *acb_poly_t A*, const *acb_poly_t B*, *slong prec*)

Performs polynomial division with remainder, computing a quotient Q and a remainder R such that $A = BQ + R$. The implementation reverses the inputs and performs power series division.

If the leading coefficient of B contains zero (or if B is identically zero), returns 0 indicating failure without modifying the outputs. Otherwise returns nonzero.

void **_acb_poly_div_root**(*acb_ptr Q*, *acb_t R*, *acb_srcptr A*, *slong len*, const *acb_t c*, *slong prec*)

Divides A by the polynomial $x - c$, computing the quotient Q as well as the remainder $R = f(c)$.

5.2.10 Composition

void **_acb_poly_taylor_shift_horner**(*acb_ptr g*, const *acb_t c*, *slong n*, *slong prec*)

void **acb_poly_taylor_shift_horner**(*acb_poly_t g*, const *acb_poly_t f*, const *acb_t c*, *slong prec*)

void **_acb_poly_taylor_shift_divconquer**(*acb_ptr g*, const *acb_t c*, *slong n*, *slong prec*)

void **acb_poly_taylor_shift_divconquer**(*acb_poly_t g*, const *acb_poly_t f*, const *acb_t c*, *slong prec*)

void **_acb_poly_taylor_shift_convolution**(*acb_ptr g*, const *acb_t c*, *slong n*, *slong prec*)

void **acb_poly_taylor_shift_convolution**(*acb_poly_t g*, const *acb_poly_t f*, const *acb_t c*, *slong prec*)

void **_acb_poly_taylor_shift**(*acb_ptr g*, const *acb_t c*, *slong n*, *slong prec*)

void **acb_poly_taylor_shift**(*acb_poly_t g*, const *acb_poly_t f*, const *acb_t c*, *slong prec*)

Sets g to the Taylor shift $f(x+c)$, computed respectively using an optimized form of Horner's rule, divide-and-conquer, a single convolution, and an automatic choice between the three algorithms.

The underscore methods act in-place on $g = f$ which has length n.

void **_acb_poly_compose_horner**(*acb_ptr res*, *acb_srcptr poly1*, *slong len1*, *acb_srcptr poly2*, *slong len2*, *slong prec*)

void **acb_poly_compose_horner**(*acb_poly_t res*, const *acb_poly_t poly1*, const *acb_poly_t poly2*, *slong prec*)

void **_acb_poly_compose_divconquer**(*acb_ptr res*, *acb_srcptr poly1*, *slong len1*, *acb_srcptr poly2*, *slong len2*, *slong prec*)

void **acb_poly_compose_divconquer**(*acb_poly_t res*, const *acb_poly_t poly1*, const *acb_poly_t poly2*, *slong prec*)

void **_acb_poly_compose**(*acb_ptr res*, *acb_srcptr poly1*, *slong len1*, *acb_srcptr poly2*, *slong len2*, *slong prec*)

void **acb_poly_compose**(*acb_poly_t res*, const *acb_poly_t poly1*, const *acb_poly_t poly2*, *slong prec*)

Sets *res* to the composition $h(x) = f(g(x))$ where f is given by *poly1* and g is given by *poly2*, respectively using Horner's rule, divide-and-conquer, and an automatic choice between the two algorithms.

The default algorithm also handles special-form input $g = ax^n + c$ efficiently by performing a Taylor shift followed by a rescaling.

The underscore methods do not support aliasing of the output with either input polynomial.

void **_acb_poly_compose_series_horner**(*acb_ptr res*, *acb_srcptr poly1*, *slong len1*, *acb_srcptr poly2*, *slong len2*, *slong n*, *slong prec*)

void **acb_poly_compose_series_horner**(*acb_poly_t res*, const *acb_poly_t poly1*, const *acb_poly_t poly2*, *slong n*, *slong prec*)

void **_acb_poly_compose_series_brent_kung**(*acb_ptr res*, *acb_srcptr poly1*, *slong len1*, *acb_srcptr poly2*, *slong len2*, *slong n*, *slong prec*)

void **acb_poly_compose_series_brent_kung**(*acb_poly_t res*, const *acb_poly_t poly1*, const *acb_poly_t poly2*, *slong n*, *slong prec*)

void **_acb_poly_compose_series**(*acb_ptr res*, *acb_srcptr poly1*, *slong len1*, *acb_srcptr poly2*, *slong len2*, *slong n*, *slong prec*)

void **acb_poly_compose_series**(*acb_poly_t res*, const *acb_poly_t poly1*, const *acb_poly_t poly2*, *slong n*, *slong prec*)

Sets *res* to the power series composition $h(x) = f(g(x))$ truncated to order $O(x^n)$ where f is given by *poly1* and g is given by *poly2*, respectively using Horner's rule, the Brent-Kung baby step-giant step algorithm, and an automatic choice between the two algorithms.

The default algorithm also handles special-form input $g = ax^n$ efficiently.

We require that the constant term in $g(x)$ is exactly zero. The underscore methods do not support aliasing of the output with either input polynomial.

void **_acb_poly_revert_series_lagrange**(*acb_ptr h*, *acb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **acb_poly_revert_series_lagrange**(*acb_poly_t h*, const *acb_poly_t f*, *slong n*, *slong prec*)

void **_acb_poly_revert_series_newton**(*acb_ptr h*, *acb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **acb_poly_revert_series_newton**(*acb_poly_t h*, const *acb_poly_t f*, *slong n*, *slong prec*)

void **_acb_poly_revert_series_lagrange_fast**(*acb_ptr h*, *acb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **acb_poly_revert_series_lagrange_fast**(*acb_poly_t h*, const *acb_poly_t f*, *slong n*, *slong prec*)

void **_acb_poly_revert_series**(*acb_ptr h*, *acb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **acb_poly_revert_series**(*acb_poly_t h*, const *acb_poly_t f*, *slong n*, *slong prec*)

Sets h to the power series reversion of f, i.e. the expansion of the compositional inverse function $f^{-1}(x)$, truncated to order $O(x^n)$, using respectively Lagrange inversion, Newton iteration, fast Lagrange inversion, and a default algorithm choice.

We require that the constant term in f is exactly zero and that the linear term is nonzero. The underscore methods assume that *flen* is at least 2, and do not support aliasing.

5.2.11 Evaluation

void **_acb_poly_evaluate_horner**(*acb_t y*, *acb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **acb_poly_evaluate_horner**(*acb_t y*, const *acb_poly_t f*, const *acb_t x*, *slong prec*)

void **_acb_poly_evaluate_rectangular**(*acb_t y*, *acb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **acb_poly_evaluate_rectangular**(*acb_t y*, const *acb_poly_t f*, const *acb_t x*, *slong prec*)

void **_acb_poly_evaluate**(*acb_t y*, *acb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **acb_poly_evaluate**(*acb_t y*, const *acb_poly_t f*, const *acb_t x*, *slong prec*)

Sets $y = f(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

void **_acb_poly_evaluate2_horner**(*acb_t y*, *acb_t z*, *acb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **acb_poly_evaluate2_horner**(*acb_t y*, *acb_t z*, const *acb_poly_t f*, const *acb_t x*, *slong prec*)

void **_acb_poly_evaluate2_rectangular**(*acb_t y*, *acb_t z*, *acb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **acb_poly_evaluate2_rectangular**(*acb_t y*, *acb_t z*, const *acb_poly_t f*, const *acb_t x*, *slong prec*)

void **_acb_poly_evaluate2**(*acb_t y*, *acb_t z*, *acb_srcptr f*, *slong len*, const *acb_t x*, *slong prec*)

void **acb_poly_evaluate2**(*acb_t y*, *acb_t z*, const *acb_poly_t f*, const *acb_t x*, *slong prec*)
Sets $y = f(x), z = f'(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

When Horner's rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.

5.2.12 Product trees

void **_acb_poly_product_roots**(*acb_ptr poly*, *acb_srcptr xs*, *slong n*, *slong prec*)

void **acb_poly_product_roots**(*acb_poly_t poly*, *acb_srcptr xs*, *slong n*, *slong prec*)
Generates the polynomial $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$.

acb_ptr * **_acb_poly_tree_alloc**(*slong len*)
Returns an initialized data structured capable of representing a remainder tree (product tree) of *len* roots.

void **_acb_poly_tree_free**(*acb_ptr* * *tree*, *slong len*)
Deallocates a tree structure as allocated using *_acb_poly_tree_alloc*.

void **_acb_poly_tree_build**(*acb_ptr* * *tree*, *acb_srcptr roots*, *slong len*, *slong prec*)
Constructs a product tree from a given array of *len* roots. The tree structure must be pre-allocated to the specified length using *_acb_poly_tree_alloc()*.

5.2.13 Multipoint evaluation

void **_acb_poly_evaluate_vec_iter**(*acb_ptr ys*, *acb_srcptr poly*, *slong plen*, *acb_srcptr xs*, *slong n*, *slong prec*)

void **acb_poly_evaluate_vec_iter**(*acb_ptr ys*, const *acb_poly_t poly*, *acb_srcptr xs*, *slong n*, *slong prec*)
Evaluates the polynomial simultaneously at n given points, calling *_acb_poly_evaluate()* repeatedly.

void **_acb_poly_evaluate_vec_fast_precomp**(*acb_ptr vs*, *acb_srcptr poly*, *slong plen*, *acb_ptr* * *tree*, *slong len*, *slong prec*)

void **_acb_poly_evaluate_vec_fast**(*acb_ptr ys*, *acb_srcptr poly*, *slong plen*, *acb_srcptr xs*, *slong n*, *slong prec*)

void **acb_poly_evaluate_vec_fast**(*acb_ptr ys*, const *acb_poly_t poly*, *acb_srcptr xs*, *slong n*, *slong prec*)
Evaluates the polynomial simultaneously at n given points, using fast multipoint evaluation.

5.2.14 Interpolation

void **_acb_poly_interpolate_newton**(*acb_ptr poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong n*, *slong prec*)

void **acb_poly_interpolate_newton**(*acb_poly_t poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong n*, *slong prec*)
Recovers the unique polynomial of length at most n that interpolates the given x and y values. This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

void **_acb_poly_interpolate_barycentric**(*acb_ptr poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong n*, *slong prec*)

void **acb_poly_interpolate_barycentric**(*acb_poly_t poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong n*, *slong prec*)

Recovers the unique polynomial of length at most n that interpolates the given x and y values. This implementation uses the barycentric form of Lagrange interpolation.

void **_acb_poly_interpolation_weights**(*acb_ptr w*, *acb_ptr* * *tree*, *slong len*, *slong prec*)

void **_acb_poly_interpolate_fast_precomp**(*acb_ptr poly*, *acb_srcptr ys*, *acb_ptr* * *tree*, *acb_srcptr weights*, *slong len*, *slong prec*)

void **_acb_poly_interpolate_fast**(*acb_ptr poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong len*, *slong prec*)

void **acb_poly_interpolate_fast**(*acb_poly_t poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong n*, *slong prec*)

Recovers the unique polynomial of length at most n that interpolates the given x and y values, using fast Lagrange interpolation. The precomp function takes a precomputed product tree over the x values and a vector of interpolation weights as additional inputs.

5.2.15 Differentiation

void **_acb_poly_derivative**(*acb_ptr res*, *acb_srcptr poly*, *slong len*, *slong prec*)

Sets *{res, len - 1}* to the derivative of *{poly, len}*. Allows aliasing of the input and output.

void **acb_poly_derivative**(*acb_poly_t res*, const *acb_poly_t poly*, *slong prec*)

Sets *res* to the derivative of *poly*.

void **_acb_poly_integral**(*acb_ptr res*, *acb_srcptr poly*, *slong len*, *slong prec*)

Sets *{res, len}* to the integral of *{poly, len - 1}*. Allows aliasing of the input and output.

void **acb_poly_integral**(*acb_poly_t res*, const *acb_poly_t poly*, *slong prec*)

Sets *res* to the integral of *poly*.

5.2.16 Elementary functions

void **_acb_poly_pow_ui_trunc_binexp**(*acb_ptr res*, *acb_srcptr f*, *slong flen*, *ulong exp*, *slong len*, *slong prec*)

Sets *{res, len}* to *{f, flen}* raised to the power *exp*, truncated to length *len*. Requires that *len* is no longer than the length of the power as computed without truncation (i.e. no zero-padding is performed). Does not support aliasing of the input and output, and requires that *flen* and *len* are positive. Uses binary expontiation.

void **acb_poly_pow_ui_trunc_binexp**(*acb_poly_t res*, const *acb_poly_t poly*, *ulong exp*, *slong len*, *slong prec*)

Sets *res* to *poly* raised to the power *exp*, truncated to length *len*. Uses binary exponentiation.

void **_acb_poly_pow_ui**(*acb_ptr res*, *acb_srcptr f*, *slong flen*, *ulong exp*, *slong prec*)

Sets *res* to *{f, flen}* raised to the power *exp*. Does not support aliasing of the input and output, and requires that *flen* is positive.

void **acb_poly_pow_ui**(*acb_poly_t res*, const *acb_poly_t poly*, *ulong exp*, *slong prec*)

Sets *res* to *poly* raised to the power *exp*.

void **_acb_poly_pow_series**(*acb_ptr h*, *acb_srcptr f*, *slong flen*, *acb_srcptr g*, *slong glen*, *slong len*, *slong prec*)

Sets *{h, len}* to the power series $f(x)^{g(x)} = \exp(g(x) \log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that *flen* and *glen* do not exceed *len*.

void **acb_poly_pow_series**(*acb_poly_t h*, const *acb_poly_t f*, const *acb_poly_t g*, *slong len*, *slong prec*)
Sets h to the power series $f(x)^{g(x)} = \exp(g(x) \log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently.

void **_acb_poly_pow_acb_series**(*acb_ptr h*, *acb_srcptr f*, *slong flen*, const *acb_t g*, *slong len*, *slong prec*)
Sets *{h, len}* to the power series $f(x)^g = \exp(g \log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that *flen* does not exceed *len*.

void **acb_poly_pow_acb_series**(*acb_poly_t h*, const *acb_poly_t f*, const *acb_t g*, *slong len*, *slong prec*)
Sets h to the power series $f(x)^g = \exp(g \log f(x))$ truncated to length *len*.

void **_acb_poly_sqrt_series**(*acb_ptr g*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_sqrt_series**(*acb_poly_t g*, const *acb_poly_t h*, *slong n*, *slong prec*)
Sets g to the power series square root of h, truncated to length n. Uses division-free Newton iteration for the reciprocal square root, followed by a multiplication.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and n are greater than zero.

void **_acb_poly_rsqrt_series**(*acb_ptr g*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_rsqrt_series**(*acb_poly_t g*, const *acb_poly_t h*, *slong n*, *slong prec*)
Sets g to the reciprocal power series square root of h, truncated to length n. Uses division-free Newton iteration.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and n are greater than zero.

void **_acb_poly_log_series**(*acb_ptr res*, *acb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **acb_poly_log_series**(*acb_poly_t res*, const *acb_poly_t f*, *slong n*, *slong prec*)
Sets *res* to the power series logarithm of f, truncated to length n. Uses the formula $\log(f(x)) = \int f'(x)/f(x)dx$, adding the logarithm of the constant term in f as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that *flen* and n are greater than zero.

void **_acb_poly_atan_series**(*acb_ptr res*, *acb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **acb_poly_atan_series**(*acb_poly_t res*, const *acb_poly_t f*, *slong n*, *slong prec*)
Sets *res* the power series inverse tangent of f, truncated to length n.

Uses the formula

$$\tan^{-1}(f(x)) = \int f'(x)/(1 + f(x)^2)dx,$$

adding the function of the constant term in f as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that *flen* and n are greater than zero.

void **_acb_poly_exp_series_basecase**(*acb_ptr f*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_exp_series_basecase**(*acb_poly_t f*, const *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_exp_series**(*acb_ptr f*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_exp_series**(*acb_poly_t f*, const *acb_poly_t h*, *slong n*, *slong prec*)
Sets f to the power series exponential of h, truncated to length n.

The basecase version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where m is the length of h.

The main implementation uses Newton iteration, starting from a small number of terms given by the basecase algorithm. The complexity is $O(M(n))$. Redundant operations in the Newton iteration are avoided by using the scheme described in *[HZ2004]*.

The underscore methods support aliasing and allow the input to be shorter than the output, but require the lengths to be nonzero.

void **_acb_poly_sin_cos_series_basecase**(*acb_ptr* *s*, *acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*, int *times_pi*)

void **acb_poly_sin_cos_series_basecase**(*acb_poly_t* *s*, *acb_poly_t* *c*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*, int *times_pi*)

void **_acb_poly_sin_cos_series_tangent**(*acb_ptr* *s*, *acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*, int *times_pi*)

void **acb_poly_sin_cos_series_tangent**(*acb_poly_t* *s*, *acb_poly_t* *c*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*, int *times_pi*)

void **_acb_poly_sin_cos_series**(*acb_ptr* *s*, *acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sin_cos_series**(*acb_poly_t* *s*, *acb_poly_t* *c*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Sets s and c to the power series sine and cosine of h, computed simultaneously.

The *basecase* version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where m is the length of h.

The *tangent* version uses the tangent half-angle formulas to compute the sine and cosine via *_acb_poly_tan_series()*. This requires $O(M(n))$ operations. When $h = h_0 + h_1$ where the constant term h_0 is nonzero, the evaluation is done as $\sin(h_0 + h_1) = \cos(h_0)\sin(h_1) + \sin(h_0)\cos(h_1)$, $\cos(h_0 + h_1) = \cos(h_0)\cos(h_1) - \sin(h_0)\sin(h_1)$, to improve accuracy and avoid dividing by zero at the poles of the tangent function.

The default version automatically selects between the *basecase* and *tangent* algorithms depending on the input.

The *basecase* and *tangent* versions take a flag *times_pi* specifying that the input is to be multiplied by π.

The underscore methods support aliasing and require the lengths to be nonzero.

void **_acb_poly_sin_series**(*acb_ptr* *s*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sin_series**(*acb_poly_t* *s*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_acb_poly_cos_series**(*acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_cos_series**(*acb_poly_t* *c*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Respectively evaluates the power series sine or cosine. These functions simply wrap *_acb_poly_sin_cos_series()*. The underscore methods support aliasing and require the lengths to be nonzero.

void **_acb_poly_tan_series**(*acb_ptr* *g*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *len*, *slong* *prec*)

void **acb_poly_tan_series**(*acb_poly_t* *g*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Sets g to the power series tangent of h.

For small n takes the quotient of the sine and cosine as computed using the basecase algorithm. For large n, uses Newton iteration to invert the inverse tangent series. The complexity is $O(M(n))$.

The underscore version does not support aliasing, and requires the lengths to be nonzero.

void **_acb_poly_sin_cos_pi_series**(*acb_ptr* *s*, *acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sin_cos_pi_series**(*acb_poly_t* *s*, *acb_poly_t* *c*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_acb_poly_sin_pi_series**(*acb_ptr* *s*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sin_pi_series**(*acb_poly_t s*, const *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_cos_pi_series**(*acb_ptr c*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_cos_pi_series**(*acb_poly_t c*, const *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_cot_pi_series**(*acb_ptr c*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_cot_pi_series**(*acb_poly_t c*, const *acb_poly_t h*, *slong n*, *slong prec*)
Compute the respective trigonometric functions of the input multiplied by π.

void **_acb_poly_sinh_cosh_series_basecase**(*acb_ptr s*, *acb_ptr c*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_sinh_cosh_series_basecase**(*acb_poly_t s*, *acb_poly_t c*, const *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_sinh_cosh_series_exponential**(*acb_ptr s*, *acb_ptr c*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_sinh_cosh_series_exponential**(*acb_poly_t s*, *acb_poly_t c*, const *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_sinh_cosh_series**(*acb_ptr s*, *acb_ptr c*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_sinh_cosh_series**(*acb_poly_t s*, *acb_poly_t c*, const *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_sinh_series**(*acb_ptr s*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_sinh_series**(*acb_poly_t s*, const *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_cosh_series**(*acb_ptr c*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_cosh_series**(*acb_poly_t c*, const *acb_poly_t h*, *slong n*, *slong prec*)
Sets s and c respectively to the hyperbolic sine and cosine of the power series h, truncated to length n.

The implementations mirror those for sine and cosine, except that the *exponential* version computes both functions using the exponential function instead of the hyperbolic tangent.

void **_acb_poly_sinc_series**(*acb_ptr s*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_sinc_series**(*acb_poly_t s*, const *acb_poly_t h*, *slong n*, *slong prec*)
Sets s to the sinc function of the power series h, truncated to length n.

5.2.17 Gamma function

void **_acb_poly_gamma_series**(*acb_ptr res*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_gamma_series**(*acb_poly_t res*, const *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_rgamma_series**(*acb_ptr res*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_rgamma_series**(*acb_poly_t res*, const *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_lgamma_series**(*acb_ptr res*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_lgamma_series**(*acb_poly_t res*, const *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_digamma_series**(*acb_ptr res*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_digamma_series**(*acb_poly_t res*, const *acb_poly_t h*, *slong n*, *slong prec*)
Sets *res* to the series expansion of $\Gamma(h(x))$, $1/\Gamma(h(x))$, or $\log\Gamma(h(x))$, $\psi(h(x))$, truncated to length n.

These functions first generate the Taylor series at the constant term of h, and then call *_acb_poly_compose_series()*. The Taylor coefficients are generated using Stirling's series.

The underscore methods support aliasing of the input and output arrays, and require that *hlen* and n are greater than zero.

void **_acb_poly_rising_ui_series**(*acb_ptr* *res*, *acb_srcptr* *f*, *slong* *flen*, *ulong* *r*, *slong* *trunc*, *slong* *prec*)

void **acb_poly_rising_ui_series**(*acb_poly_t* *res*, const *acb_poly_t* *f*, *ulong* *r*, *slong* *trunc*, *slong* *prec*)

Sets *res* to the rising factorial $(f)(f+1)(f+2)\cdots(f+r-1)$, truncated to length *trunc*. The underscore method assumes that *flen*, *r* and *trunc* are at least 1, and does not support aliasing. Uses binary splitting.

5.2.18 Power sums

void **_acb_poly_powsum_series_naive**(*acb_ptr* *z*, const *acb_t* *s*, const *acb_t* *a*, const *acb_t* *q*, *slong* *n*, *slong* *len*, *slong* *prec*)

void **_acb_poly_powsum_series_naive_threaded**(*acb_ptr* *z*, const *acb_t* *s*, const *acb_t* *a*, const *acb_t* *q*, *slong* *n*, *slong* *len*, *slong* *prec*)

Computes

$$z = S(s,a,n) = \sum_{k=0}^{n-1} \frac{q^k}{(k+a)^{s+t}}$$

as a power series in t truncated to length *len*. This function evaluates the sum naively term by term. The *threaded* version splits the computation over the number of threads returned by *flint_get_num_threads()*.

void **_acb_poly_powsum_one_series_sieved**(*acb_ptr* *z*, const *acb_t* *s*, *slong* *n*, *slong* *len*, *slong* *prec*)

Computes

$$z = S(s,1,n) \sum_{k=1}^{n} \frac{1}{k^{s+t}}$$

as a power series in t truncated to length *len*. This function stores a table of powers that have already been calculated, computing $(ij)^r$ as $i^r j^r$ whenever $k = ij$ is composite. As a further optimization, it groups all even k and evaluates the sum as a polynomial in $2^{-(s+t)}$. This scheme requires about $n/\log n$ powers, $n/2$ multiplications, and temporary storage of $n/6$ power series. Due to the extra power series multiplications, it is only faster than the naive algorithm when *len* is small.

5.2.19 Zeta function

void **_acb_poly_zeta_em_choose_param**(*mag_t* *bound*, *ulong* * *N*, *ulong* * *M*, const *acb_t* *s*, const *acb_t* *a*, *slong* *d*, *slong* *target*, *slong* *prec*)

Chooses N and M for Euler-Maclaurin summation of the Hurwitz zeta function, using a default algorithm.

void **_acb_poly_zeta_em_bound1**(*mag_t* *bound*, const *acb_t* *s*, const *acb_t* *a*, *slong* *N*, *slong* *M*, *slong* *d*, *slong* *wp*)

void **_acb_poly_zeta_em_bound**(*arb_ptr* *vec*, const *acb_t* *s*, const *acb_t* *a*, *ulong* *N*, *ulong* *M*, *slong* *d*, *slong* *wp*)

Compute bounds for Euler-Maclaurin evaluation of the Hurwitz zeta function or its power series, using the formulas in *[Joh2013]*.

void **_acb_poly_zeta_em_tail_naive**(*acb_ptr* *z*, const *acb_t* *s*, const *acb_t* *Na*, *acb_srcptr* *Nasx*, *slong* *M*, *slong* *len*, *slong* *prec*)

void **_acb_poly_zeta_em_tail_bsplit**(*acb_ptr z*, const *acb_t s*, const *acb_t Na*, *acb_srcptr Nasx*, *slong M*, *slong len*, *slong prec*)
Evaluates the tail in the Euler-Maclaurin sum for the Hurwitz zeta function, respectively using the naive recurrence and binary splitting.

void **_acb_poly_zeta_em_sum**(*acb_ptr z*, const *acb_t s*, const *acb_t a*, int *deflate*, *ulong N*, *ulong M*, *slong d*, *slong prec*)
Evaluates the truncated Euler-Maclaurin sum of order N, M for the length-d truncated Taylor series of the Hurwitz zeta function $\zeta(s, a)$ at s, using a working precision of *prec* bits. With $a = 1$, this gives the usual Riemann zeta function.

If *deflate* is nonzero, $\zeta(s, a) - 1/(s-1)$ is evaluated (which permits series expansion at $s = 1$).

void **_acb_poly_zeta_cpx_series**(*acb_ptr z*, const *acb_t s*, const *acb_t a*, int *deflate*, *slong d*, *slong prec*)
Computes the series expansion of $\zeta(s+x, a)$ (or $\zeta(s+x, a) - 1/(s+x-1)$ if *deflate* is nonzero) to order d.

This function wraps *_acb_poly_zeta_em_sum()*, automatically choosing default values for N, M using *_acb_poly_zeta_em_choose_param()* to target an absolute truncation error of $2^{-\text{prec}}$.

void **_acb_poly_zeta_series**(*acb_ptr res*, *acb_srcptr h*, *slong hlen*, const *acb_t a*, int *deflate*, *slong len*, *slong prec*)

void **acb_poly_zeta_series**(*acb_poly_t res*, const *acb_poly_t f*, const *acb_t a*, int *deflate*, *slong n*, *slong prec*)
Sets *res* to the Hurwitz zeta function $\zeta(s, a)$ where s a power series and a is a constant, truncated to length n. To evaluate the usual Riemann zeta function, set $a = 1$.

If *deflate* is nonzero, evaluates $\zeta(s, a) + 1/(1-s)$, which is well-defined as a limit when the constant term of s is 1. In particular, expanding $\zeta(s, a) + 1/(1-s)$ with $s = 1+x$ gives the Stieltjes constants

$$\sum_{k=0}^{n-1} \frac{(-1)^k}{k!} \gamma_k(a) x^k.$$

If $a = 1$, this implementation uses the reflection formula if the midpoint of the constant term of s is negative.

5.2.20 Other special functions

void **_acb_poly_polylog_cpx_small**(*acb_ptr w*, const *acb_t s*, const *acb_t z*, *slong len*, *slong prec*)

void **_acb_poly_polylog_cpx_zeta**(*acb_ptr w*, const *acb_t s*, const *acb_t z*, *slong len*, *slong prec*)

void **_acb_poly_polylog_cpx**(*acb_ptr w*, const *acb_t s*, const *acb_t z*, *slong len*, *slong prec*)
Sets w to the Taylor series with respect to x of the polylogarithm $\mathrm{Li}_{s+x}(z)$, where s and z are given complex constants. The output is computed to length *len* which must be positive. Aliasing between w and s or z is not permitted.

The *small* version uses the standard power series expansion with respect to z, convergent when $|z| < 1$. The *zeta* version evaluates the polylogarithm as a sum of two Hurwitz zeta functions. The default version automatically delegates to the *small* version when z is close to zero, and the *zeta* version otherwise. For further details, see *Algorithms for polylogarithms*.

void **_acb_poly_polylog_series**(*acb_ptr w*, *acb_srcptr s*, *slong slen*, const *acb_t z*, *slong len*, *slong prec*)

void **acb_poly_polylog_series**(*acb_poly_t w*, const *acb_poly_t s*, const *acb_t z*, *slong len*, *slong prec*)
Sets w to the polylogarithm $\mathrm{Li}_s(z)$ where s is a given power series, truncating the output to length *len*. The underscore method requires all lengths to be positive and supports aliasing between all inputs and outputs.

void **_acb_poly_erf_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong n*, *slong prec*)

void **acb_poly_erf_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong n*, *slong prec*)
Sets *res* to the error function of the power series *z*, truncated to length *n*. These methods are provided for backwards compatibility. See *acb_hypgeom_erf_series()*, *acb_hypgeom_erfc_series()*, *acb_hypgeom_erfi_series()*.

void **_acb_poly_agm1_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_poly_agm1_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong n*, *slong prec*)
Sets *res* to the arithmetic-geometric mean of 1 and the power series *z*, truncated to length *n*.

void **_acb_poly_elliptic_k_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_poly_elliptic_k_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong n*, *slong prec*)
Sets *res* to the complete elliptic integral of the first kind of the power series *z*, truncated to length *n*.

void **_acb_poly_elliptic_p_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, const *acb_t tau*, *slong len*, *slong prec*)

void **acb_poly_elliptic_p_series**(*acb_poly_t res*, const *acb_poly_t z*, const *acb_t tau*, *slong n*, *slong prec*)
Sets *res* to the Weierstrass elliptic function of the power series *z*, with periods 1 and *tau*, truncated to length *n*.

5.2.21 Root-finding

void **_acb_poly_root_bound_fujiwara**(*mag_t bound*, *acb_srcptr poly*, *slong len*)

void **acb_poly_root_bound_fujiwara**(*mag_t bound*, *acb_poly_t poly*)
Sets *bound* to an upper bound for the magnitude of all the complex roots of *poly*. Uses Fujiwara's bound

$$2 \max \left\{ \left| \frac{a_{n-1}}{a_n} \right|, \left| \frac{a_{n-2}}{a_n} \right|^{1/2}, \cdots, \left| \frac{a_1}{a_n} \right|^{1/(n-1)}, \left| \frac{a_0}{2a_n} \right|^{1/n} \right\}$$

where $a_0, \ldots, a_n$ are the coefficients of *poly*.

void **_acb_poly_root_inclusion**(*acb_t r*, const *acb_t m*, *acb_srcptr poly*, *acb_srcptr polyder*, *slong len*, *slong prec*)
Given any complex number m, and a nonconstant polynomial f and its derivative f', sets r to a complex interval centered on m that is guaranteed to contain at least one root of f. Such an interval is obtained by taking a ball of radius $|f(m)/f'(m)|n$ where n is the degree of f. Proof: assume that the distance to the nearest root exceeds $r = |f(m)/f'(m)|n$. Then

$$\left| \frac{f'(m)}{f(m)} \right| = \left| \sum_i \frac{1}{m - \zeta_i} \right| \le \sum_i \frac{1}{|m - \zeta_i|} < \frac{n}{r} = \left| \frac{f'(m)}{f(m)} \right|$$

which is a contradiction (see *[Kob2010]*).

slong **_acb_poly_validate_roots**(*acb_ptr roots*, *acb_srcptr poly*, *slong len*, *slong prec*)
Given a list of approximate roots of the input polynomial, this function sets a rigorous bounding interval for each root, and determines which roots are isolated from all the other roots. It then rearranges the list of roots so that the isolated roots are at the front of the list, and returns the count of isolated roots.

If the return value equals the degree of the polynomial, then all roots have been found. If the return value is smaller, all the remaining output intervals are guaranteed to contain roots, but it is possible that not all of the polynomial's roots are contained among them.

void **_acb_poly_refine_roots_durand_kerner**(*acb_ptr roots*, *acb_srcptr poly*, *slong len*, *slong prec*)
Refines the given roots simultaneously using a single iteration of the Durand-Kerner method. The

radius of each root is set to an approximation of the correction, giving a rough estimate of its error (not a rigorous bound).

slong **_acb_poly_find_roots**(*acb_ptr roots*, *acb_srcptr poly*, *acb_srcptr initial*, *slong len*, *slong maxiter*, *slong prec*)

slong **acb_poly_find_roots**(*acb_ptr roots*, const *acb_poly_t poly*, *acb_srcptr initial*, *slong maxiter*, *slong prec*)

Attempts to compute all the roots of the given nonzero polynomial *poly* using a working precision of *prec* bits. If n denotes the degree of *poly*, the function writes n approximate roots with rigorous error bounds to the preallocated array *roots*, and returns the number of roots that are isolated.

If the return value equals the degree of the polynomial, then all roots have been found. If the return value is smaller, all the output intervals are guaranteed to contain roots, but it is possible that not all of the polynomial's roots are contained among them.

The roots are computed numerically by performing several steps with the Durand-Kerner method and terminating if the estimated accuracy of the roots approaches the working precision or if the number of steps exceeds *maxiter*, which can be set to zero in order to use a default value. Finally, the approximate roots are validated rigorously.

Initial values for the iteration can be provided as the array *initial*. If *initial* is set to *NULL*, default values $(0.4 + 0.9i)^k$ are used.

The polynomial is assumed to be squarefree. If there are repeated roots, the iteration is likely to find them (with low numerical accuracy), but the error bounds will not converge as the precision increases.

int **_acb_poly_validate_real_roots**(*acb_srcptr roots*, *acb_srcptr poly*, *slong len*, *slong prec*)

int **acb_poly_validate_real_roots**(*acb_srcptr roots*, const *acb_poly_t poly*, *slong prec*)

Given a strictly real polynomial *poly* (of length *len*) and isolating intervals for all its complex roots, determines if all the real roots are separated from the non-real roots. If this function returns nonzero, every root enclosure that touches the real axis (as tested by applying *arb_contains_zero()* to the imaginary part) corresponds to a real root (its imaginary part can be set to zero), and every other root enclosure corresponds to a non-real root (with known sign for the imaginary part).

If this function returns zero, then the signs of the imaginary parts are not known for certain, based on the accuracy of the inputs and the working precision *prec*.

CHAPTER

SIX

MATRICES

These modules implement dense matrices with real and complex coefficients. Rudimentary linear algebra is supported.

6.1 arb_mat.h – matrices over the real numbers

An *arb_mat_t* represents a dense matrix over the real numbers, implemented as an array of entries of type *arb_struct*.

The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

6.1.1 Types, macros and constants

arb_mat_struct

arb_mat_t
Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An *arb_mat_t* is defined as an array of length one of type *arb_mat_struct*, permitting an *arb_mat_t* to be passed by reference.

arb_mat_entry(mat, i, j)
Macro giving a pointer to the entry at row i and column j.

arb_mat_nrows(mat)
Returns the number of rows of the matrix.

arb_mat_ncols(mat)
Returns the number of columns of the matrix.

6.1.2 Memory management

void **arb_mat_init**(*arb_mat_t mat*, *slong r*, *slong c*)
Initializes the matrix, setting it to the zero matrix with r rows and c columns.

void **arb_mat_clear**(*arb_mat_t mat*)
Clears the matrix, deallocating all entries.

6.1.3 Conversions

void **arb_mat_set**(*arb_mat_t dest*, const *arb_mat_t src*)

void **arb_mat_set_fmpz_mat**(*arb_mat_t dest*, const *fmpz_mat_t src*)

void **arb_mat_set_round_fmpz_mat**(*arb_mat_t dest*, const *fmpz_mat_t src*, *slong prec*)

void **arb_mat_set_fmpq_mat**(*arb_mat_t dest*, const *fmpq_mat_t src*, *slong prec*)
Sets *dest* to *src*. The operands must have identical dimensions.

6.1.4 Random generation

void **arb_mat_randtest**(*arb_mat_t mat*, flint_rand_t *state*, *slong prec*, *slong mag_bits*)
Sets *mat* to a random matrix with up to *prec* bits of precision and with exponents of width up to *mag_bits*.

6.1.5 Input and output

void **arb_mat_printd**(const *arb_mat_t mat*, *slong digits*)
Prints each entry in the matrix with the specified number of decimal digits.

void **arb_mat_fprintd**(FILE * *file*, const *arb_mat_t mat*, *slong digits*)
Prints each entry in the matrix with the specified number of decimal digits to the stream *file*.

6.1.6 Comparisons

int **arb_mat_equal**(const *arb_mat_t mat1*, const *arb_mat_t mat2*)
Returns nonzero iff the matrices have the same dimensions and identical entries.

int **arb_mat_overlaps**(const *arb_mat_t mat1*, const *arb_mat_t mat2*)
Returns nonzero iff the matrices have the same dimensions and each entry in *mat1* overlaps with the corresponding entry in *mat2*.

int **arb_mat_contains**(const *arb_mat_t mat1*, const *arb_mat_t mat2*)

int **arb_mat_contains_fmpz_mat**(const *arb_mat_t mat1*, const *fmpz_mat_t mat2*)

int **arb_mat_contains_fmpq_mat**(const *arb_mat_t mat1*, const *fmpq_mat_t mat2*)
Returns nonzero iff the matrices have the same dimensions and each entry in *mat2* is contained in the corresponding entry in *mat1*.

int **arb_mat_eq**(const *arb_mat_t mat1*, const *arb_mat_t mat2*)
Returns nonzero iff *mat1* and *mat2* certainly represent the same matrix.

int **arb_mat_ne**(const *arb_mat_t mat1*, const *arb_mat_t mat2*)
Returns nonzero iff *mat1* and *mat2* certainly do not represent the same matrix.

int **arb_mat_is_empty**(const *arb_mat_t mat*)
Returns nonzero iff the number of rows or the number of columns in *mat* is zero.

int **arb_mat_is_square**(const *arb_mat_t mat*)
Returns nonzero iff the number of rows is equal to the number of columns in *mat*.

6.1.7 Special matrices

void **arb_mat_zero**(*arb_mat_t mat*)
Sets all entries in mat to zero.

void **arb_mat_one**(*arb_mat_t mat*)
Sets the entries on the main diagonal to ones, and all other entries to zero.

6.1.8 Transpose

void **arb_mat_transpose**(*arb_mat_t dest*, const *arb_mat_t src*)
Sets *dest* to the exact transpose *src*. The operands must have compatible dimensions. Aliasing is allowed.

6.1.9 Norms

void **arb_mat_bound_inf_norm**(*mag_t b*, const *arb_mat_t A*)
Sets b to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of A.

void **arb_mat_frobenius_norm**(*arb_t res*, const *arb_mat_t A*, *slong prec*)
Sets *res* to the Frobenius norm (i.e. the square root of the sum of squares of entries) of A.

void **arb_mat_bound_frobenius_norm**(*mag_t res*, const *arb_mat_t A*)
Sets *res* to an upper bound for the Frobenius norm of A.

6.1.10 Arithmetic

void **arb_mat_neg**(*arb_mat_t dest*, const *arb_mat_t src*)
Sets *dest* to the exact negation of *src*. The operands must have the same dimensions.

void **arb_mat_add**(*arb_mat_t res*, const *arb_mat_t mat1*, const *arb_mat_t mat2*, *slong prec*)
Sets res to the sum of *mat1* and *mat2*. The operands must have the same dimensions.

void **arb_mat_sub**(*arb_mat_t res*, const *arb_mat_t mat1*, const *arb_mat_t mat2*, *slong prec*)
Sets *res* to the difference of *mat1* and *mat2*. The operands must have the same dimensions.

void **arb_mat_mul_classical**(*arb_mat_t C*, const *arb_mat_t A*, const *arb_mat_t B*, *slong prec*)

void **arb_mat_mul_threaded**(*arb_mat_t C*, const *arb_mat_t A*, const *arb_mat_t B*, *slong prec*)

void **arb_mat_mul**(*arb_mat_t res*, const *arb_mat_t mat1*, const *arb_mat_t mat2*, *slong prec*)
Sets *res* to the matrix product of *mat1* and *mat2*. The operands must have compatible dimensions for matrix multiplication.

The *threaded* version splits the computation over the number of threads returned by *flint_get_num_threads()*. The default version automatically calls the *threaded* version if the matrices are sufficiently large and more than one thread can be used.

void **arb_mat_mul_entrywise**(*arb_mat_t C*, const *arb_mat_t A*, const *arb_mat_t B*, *slong prec*)
Sets C to the entrywise product of A and B. The operands must have the same dimensions.

void **arb_mat_sqr_classical**(*arb_mat_t B*, const *arb_mat_t A*, *slong prec*)

void **arb_mat_sqr**(*arb_mat_t res*, const *arb_mat_t mat*, *slong prec*)
Sets *res* to the matrix square of *mat*. The operands must both be square with the same dimensions.

void **arb_mat_pow_ui**(*arb_mat_t res*, const *arb_mat_t mat*, *ulong exp*, *slong prec*)
Sets *res* to *mat* raised to the power *exp*. Requires that *mat* is a square matrix.

6.1.11 Scalar arithmetic

void **arb_mat_scalar_mul_2exp_si**(*arb_mat_t B*, const *arb_mat_t A*, *slong c*)
Sets B to A multiplied by 2^c.

void **arb_mat_scalar_addmul_si**(*arb_mat_t B*, const *arb_mat_t A*, *slong c*, *slong prec*)

void **arb_mat_scalar_addmul_fmpz**(*arb_mat_t B*, const *arb_mat_t A*, const *fmpz_t c*, *slong prec*)

void **arb_mat_scalar_addmul_arb**(*arb_mat_t* B, const *arb_mat_t* A, const *arb_t* c, *slong* *prec*)
Sets B to $B + A \times c$.

void **arb_mat_scalar_mul_si**(*arb_mat_t* B, const *arb_mat_t* A, *slong* c, *slong* *prec*)

void **arb_mat_scalar_mul_fmpz**(*arb_mat_t* B, const *arb_mat_t* A, const *fmpz_t* c, *slong* *prec*)

void **arb_mat_scalar_mul_arb**(*arb_mat_t* B, const *arb_mat_t* A, const *arb_t* c, *slong* *prec*)
Sets B to $A \times c$.

void **arb_mat_scalar_div_si**(*arb_mat_t* B, const *arb_mat_t* A, *slong* c, *slong* *prec*)

void **arb_mat_scalar_div_fmpz**(*arb_mat_t* B, const *arb_mat_t* A, const *fmpz_t* c, *slong* *prec*)

void **arb_mat_scalar_div_arb**(*arb_mat_t* B, const *arb_mat_t* A, const *arb_t* c, *slong* *prec*)
Sets B to A/c.

6.1.12 Gaussian elimination and solving

int **arb_mat_lu**(*slong* * *perm*, *arb_mat_t* LU, const *arb_mat_t* A, *slong* *prec*)
Given an $n \times n$ matrix A, computes an LU decomposition $PLU = A$ using Gaussian elimination with partial pivoting. The input and output matrices can be the same, performing the decomposition in-place.

Entry i in the permutation vector perm is set to the row index in the input matrix corresponding to row i in the output matrix.

The algorithm succeeds and returns nonzero if it can find n invertible (i.e. not containing zero) pivot entries. This guarantees that the matrix is invertible.

The algorithm fails and returns zero, leaving the entries in P and LU undefined, if it cannot find n invertible pivot elements. In this case, either the matrix is singular, the input matrix was computed to insufficient precision, or the LU decomposition was attempted at insufficient precision.

void **arb_mat_solve_lu_precomp**(*arb_mat_t* X, const *slong* * *perm*, const *arb_mat_t* LU, const *arb_mat_t* B, *slong* *prec*)
Solves $AX = B$ given the precomputed nonsingular LU decomposition $A = PLU$. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with LU.

int **arb_mat_solve**(*arb_mat_t* X, const *arb_mat_t* A, const *arb_mat_t* B, *slong* *prec*)
Solves $AX = B$ where A is a nonsingular $n \times n$ matrix and X and B are $n \times m$ matrices, using LU decomposition.

If $m > 0$ and A cannot be inverted numerically (indicating either that A is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that A is invertible and that the exact solution matrix is contained in the output.

int **arb_mat_inv**(*arb_mat_t* X, const *arb_mat_t* A, *slong* *prec*)
Sets $X = A^{-1}$ where A is a square matrix, computed by solving the system $AX = I$.

If A cannot be inverted numerically (indicating either that A is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the matrix is invertible and that the exact inverse is contained in the output.

void **arb_mat_det**(*arb_t* *det*, const *arb_mat_t* A, *slong* *prec*)
Computes the determinant of the matrix, using Gaussian elimination with partial pivoting. If at some point an invertible pivot element cannot be found, the elimination is stopped and the magnitude of the determinant of the remaining submatrix is bounded using Hadamard's inequality.

6.1.13 Cholesky decomposition and solving

int **_arb_mat_cholesky_banachiewicz**(*arb_mat_t* A, *slong* *prec*)

int **arb_mat_cho**(*arb_mat_t* *L*, const *arb_mat_t* *A*, *slong* *prec*)
Computes the Cholesky decomposition of A, returning nonzero iff the symmetric matrix defined by the lower triangular part of A is certainly positive definite.

If a nonzero value is returned, then L is set to the lower triangular matrix such that $A = L * L^T$.

If zero is returned, then either the matrix is not symmetric positive definite, the input matrix was computed to insufficient precision, or the decomposition was attempted at insufficient precision.

The underscore method computes L from A in-place, leaving the strict upper triangular region undefined.

void **arb_mat_solve_cho_precomp**(*arb_mat_t* *X*, const *arb_mat_t* *L*, const *arb_mat_t* *B*, *slong* *prec*)
Solves $AX = B$ given the precomputed Cholesky decomposition $A = LL^T$. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with L.

int **arb_mat_spd_solve**(*arb_mat_t* *X*, const *arb_mat_t* *A*, const *arb_mat_t* *B*, *slong* *prec*)
Solves $AX = B$ where A is a symmetric positive definite matrix and X and B are $n \times m$ matrices, using Cholesky decomposition.

If $m > 0$ and A cannot be factored using Cholesky decomposition (indicating either that A is not symmetric positive definite or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the symmetric matrix defined through the lower triangular part of A is invertible and that the exact solution matrix is contained in the output.

void **arb_mat_inv_cho_precomp**(*arb_mat_t* *X*, const *arb_mat_t* *L*, *slong* *prec*)
Sets $X = A^{-1}$ where A is a symmetric positive definite matrix whose Cholesky decomposition L has been computed with *arb_mat_cho()*. The inverse is calculated using the method of *[Kri2013]* which is more efficient than solving $AX = I$ with *arb_mat_solve_cho_precomp()*.

int **arb_mat_spd_inv**(*arb_mat_t* *X*, const *arb_mat_t* *A*, *slong* *prec*)
Sets $X = A^{-1}$ where A is a symmetric positive definite matrix. It is calculated using the method of *[Kri2013]* which computes fewer intermediate results than solving $AX = I$ with *arb_mat_spd_solve()*.

If A cannot be factored using Cholesky decomposition (indicating either that A is not symmetric positive definite or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the symmetric matrix defined through the lower triangular part of A is invertible and that the exact inverse is contained in the output.

int **_arb_mat_ldl_inplace**(*arb_mat_t* *A*, *slong* *prec*)

int **_arb_mat_ldl_golub_and_van_loan**(*arb_mat_t* *A*, *slong* *prec*)

int **arb_mat_ldl**(*arb_mat_t* *res*, const *arb_mat_t* *A*, *slong* *prec*)
Computes the LDL^T decomposition of A, returning nonzero iff the symmetric matrix defined by the lower triangular part of A is certainly positive definite.

If a nonzero value is returned, then *res* is set to a lower triangular matrix that encodes the $L*D*L^T$ decomposition of A. In particular, L is a lower triangular matrix with ones on its diagonal and whose strictly lower triangular region is the same as that of *res*. D is a diagonal matrix with the same diagonal as that of *res*.

If zero is returned, then either the matrix is not symmetric positive definite, the input matrix was computed to insufficient precision, or the decomposition was attempted at insufficient precision.

The underscore methods compute *res* from A in-place, leaving the strict upper triangular region undefined. The default method uses algorithm 4.1.2 from *[GVL1996]*.

void **arb_mat_solve_ldl_precomp**(*arb_mat_t* *X*, const *arb_mat_t* *L*, const *arb_mat_t* *B*, *slong* *prec*)
Solves $AX = B$ given the precomputed $A = LDL^T$ decomposition encoded by L. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with L.

void **arb_mat_inv_ldl_precomp**(*arb_mat_t* *X*, const *arb_mat_t* *L*, *slong* *prec*)
Sets $X = A^{-1}$ where A is a symmetric positive definite matrix whose LDL^T decomposition encoded by L has been computed with *arb_mat_ldl()*. The inverse is calculated using the method of *[Kri2013]* which is more efficient than solving $AX = I$ with *arb_mat_solve_ldl_precomp()*.

6.1.14 Characteristic polynomial

void **_arb_mat_charpoly**(*arb_ptr* *cp*, const *arb_mat_t* *mat*, *slong* *prec*)

void **arb_mat_charpoly**(*arb_poly_t* *cp*, const *arb_mat_t* *mat*, *slong* *prec*)
Sets *cp* to the characteristic polynomial of *mat* which must be a square matrix. If the matrix has n rows, the underscore method requires space for $n+1$ output coefficients. Employs a division-free algorithm using $O(n^4)$ operations.

6.1.15 Special functions

void **arb_mat_exp_taylor_sum**(*arb_mat_t* *S*, const *arb_mat_t* *A*, *slong* *N*, *slong* *prec*)
Sets S to the truncated exponential Taylor series $S = \sum_{k=0}^{N-1} A^k/k!$. Uses rectangular splitting to compute the sum using $O(\sqrt{N})$ matrix multiplications. The recurrence relation for factorials is used to get scalars that are small integers instead of full factorials. As in *[Joh2014b]*, all divisions are postponed to the end by computing partial factorials of length $O(\sqrt{N})$. The scalars could be reduced by doing more divisions, but this appears to be slower in most cases.

void **arb_mat_exp**(*arb_mat_t* *B*, const *arb_mat_t* *A*, *slong* *prec*)
Sets B to the exponential of the matrix A, defined by the Taylor series

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

The function is evaluated as $\exp(A/2^r)^{2^r}$, where r is chosen to give rapid convergence.

The elementwise error when truncating the Taylor series after N terms is bounded by the error in the infinity norm, for which we have

$$\left\| \exp(2^{-r}A) - \sum_{k=0}^{N-1} \frac{(2^{-r}A)^k}{k!} \right\|_\infty = \left\| \sum_{k=N}^{\infty} \frac{(2^{-r}A)^k}{k!} \right\|_\infty \le \sum_{k=N}^{\infty} \frac{(2^{-r}\|A\|_\infty)^k}{k!}.$$

We bound the sum on the right using *mag_exp_tail()*. Truncation error is not added to entries whose values are determined by the sparsity structure of A.

void **arb_mat_trace**(*arb_t* *trace*, const *arb_mat_t* *mat*, *slong* *prec*)
Sets *trace* to the trace of the matrix, i.e. the sum of entries on the main diagonal of *mat*. The matrix is required to be square.

6.1.16 Sparsity structure

void **arb_mat_entrywise_is_zero**(*fmpz_mat_t* *dest*, const *arb_mat_t* *src*)
Sets each entry of *dest* to indicate whether the corresponding entry of *src* is certainly zero. If the entry of *src* at row i and column j is zero according to *arb_is_zero()* then the entry of *dest* at that row and column is set to one, otherwise that entry of *dest* is set to zero.

void **arb_mat_entrywise_not_is_zero**(*fmpz_mat_t* *dest*, const *arb_mat_t* *src*)
Sets each entry of *dest* to indicate whether the corresponding entry of *src* is not certainly zero. This the complement of *arb_mat_entrywise_is_zero()*.

slong **arb_mat_count_is_zero**(const *arb_mat_t* *mat*)
Returns the number of entries of *mat* that are certainly zero according to *arb_is_zero()*.

slong **arb_mat_count_not_is_zero**(const *arb_mat_t* *mat*)
Returns the number of entries of *mat* that are not certainly zero.

6.2 acb_mat.h – matrices over the complex numbers

An *acb_mat_t* represents a dense matrix over the complex numbers, implemented as an array of entries of type *acb_struct*.

The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

6.2.1 Types, macros and constants

acb_mat_struct

acb_mat_t
Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An *acb_mat_t* is defined as an array of length one of type *acb_mat_struct*, permitting an *acb_mat_t* to be passed by reference.

acb_mat_entry(mat, i, j)
Macro giving a pointer to the entry at row i and column j.

acb_mat_nrows(mat)
Returns the number of rows of the matrix.

acb_mat_ncols(mat)
Returns the number of columns of the matrix.

6.2.2 Memory management

void **acb_mat_init**(*acb_mat_t mat*, *slong r*, *slong c*)
Initializes the matrix, setting it to the zero matrix with r rows and c columns.

void **acb_mat_clear**(*acb_mat_t mat*)
Clears the matrix, deallocating all entries.

6.2.3 Conversions

void **acb_mat_set**(*acb_mat_t dest*, const *acb_mat_t src*)

void **acb_mat_set_fmpz_mat**(*acb_mat_t dest*, const *fmpz_mat_t src*)

void **acb_mat_set_round_fmpz_mat**(*acb_mat_t dest*, const *fmpz_mat_t src*, *slong prec*)

void **acb_mat_set_fmpq_mat**(*acb_mat_t dest*, const *fmpq_mat_t src*, *slong prec*)

void **acb_mat_set_arb_mat**(*acb_mat_t dest*, const *arb_mat_t src*)

void **acb_mat_set_round_arb_mat**(*acb_mat_t dest*, const *arb_mat_t src*, *slong prec*)
Sets *dest* to *src*. The operands must have identical dimensions.

6.2.4 Random generation

void **acb_mat_randtest**(*acb_mat_t mat*, flint_rand_t *state*, *slong prec*, *slong mag_bits*)
Sets *mat* to a random matrix with up to *prec* bits of precision and with exponents of width up to *mag_bits*.

6.2.5 Input and output

void **acb_mat_printd**(const *acb_mat_t mat*, *slong digits*)
Prints each entry in the matrix with the specified number of decimal digits.

void **acb_mat_fprintd**(FILE * *file*, const *acb_mat_t mat*, *slong digits*)
Prints each entry in the matrix with the specified number of decimal digits to the stream *file*.

6.2.6 Comparisons

int **acb_mat_equal**(const *acb_mat_t mat1*, const *acb_mat_t mat2*)
Returns nonzero iff the matrices have the same dimensions and identical entries.

int **acb_mat_overlaps**(const *acb_mat_t mat1*, const *acb_mat_t mat2*)
Returns nonzero iff the matrices have the same dimensions and each entry in *mat1* overlaps with the corresponding entry in *mat2*.

int **acb_mat_contains**(const *acb_mat_t mat1*, const *acb_mat_t mat2*)

int **acb_mat_contains_fmpz_mat**(const *acb_mat_t mat1*, const *fmpz_mat_t mat2*)

int **acb_mat_contains_fmpq_mat**(const *acb_mat_t mat1*, const *fmpq_mat_t mat2*)
Returns nonzero iff the matrices have the same dimensions and each entry in *mat2* is contained in the corresponding entry in *mat1*.

int **acb_mat_eq**(const *acb_mat_t mat1*, const *acb_mat_t mat2*)
Returns nonzero iff *mat1* and *mat2* certainly represent the same matrix.

int **acb_mat_ne**(const *acb_mat_t mat1*, const *acb_mat_t mat2*)
Returns nonzero iff *mat1* and *mat2* certainly do not represent the same matrix.

int **acb_mat_is_real**(const *acb_mat_t mat*)
Returns nonzero iff all entries in *mat* have zero imaginary part.

int **acb_mat_is_empty**(const *acb_mat_t mat*)
Returns nonzero iff the number of rows or the number of columns in *mat* is zero.

int **acb_mat_is_square**(const *acb_mat_t mat*)
Returns nonzero iff the number of rows is equal to the number of columns in *mat*.

6.2.7 Special matrices

void **acb_mat_zero**(*acb_mat_t mat*)
Sets all entries in mat to zero.

void **acb_mat_one**(*acb_mat_t mat*)
Sets the entries on the main diagonal to ones, and all other entries to zero.

6.2.8 Transpose

void **acb_mat_transpose**(*acb_mat_t dest*, const *acb_mat_t src*)
Sets *dest* to the exact transpose *src*. The operands must have compatible dimensions. Aliasing is allowed.

6.2.9 Norms

void **acb_mat_bound_inf_norm**(*mag_t b*, const *acb_mat_t A*)
Sets b to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of A.

void **acb_mat_frobenius_norm**(*acb_t res*, const *acb_mat_t A*, *slong prec*)
Sets *res* to the Frobenius norm (i.e. the square root of the sum of squares of entries) of A.

void **acb_mat_bound_frobenius_norm**(*mag_t res*, const *acb_mat_t A*)
Sets *res* to an upper bound for the Frobenius norm of A.

6.2.10 Arithmetic

void **acb_mat_neg**(*acb_mat_t dest*, const *acb_mat_t src*)
Sets *dest* to the exact negation of *src*. The operands must have the same dimensions.

void **acb_mat_add**(*acb_mat_t res*, const *acb_mat_t mat1*, const *acb_mat_t mat2*, *slong prec*)
Sets res to the sum of *mat1* and *mat2*. The operands must have the same dimensions.

void **acb_mat_sub**(*acb_mat_t res*, const *acb_mat_t mat1*, const *acb_mat_t mat2*, *slong prec*)
Sets *res* to the difference of *mat1* and *mat2*. The operands must have the same dimensions.

void **acb_mat_mul**(*acb_mat_t res*, const *acb_mat_t mat1*, const *acb_mat_t mat2*, *slong prec*)
Sets *res* to the matrix product of *mat1* and *mat2*. The operands must have compatible dimensions for matrix multiplication.

void **acb_mat_mul_entrywise**(*acb_mat_t res*, const *acb_mat_t mat1*, const *acb_mat_t mat2*, *slong prec*)
Sets *res* to the entrywise product of *mat1* and *mat2*. The operands must have the same dimensions.

void **acb_mat_sqr**(*acb_mat_t res*, const *acb_mat_t mat*, *slong prec*)
Sets *res* to the matrix square of *mat*. The operands must both be square with the same dimensions.

void **acb_mat_pow_ui**(*acb_mat_t res*, const *acb_mat_t mat*, *ulong exp*, *slong prec*)
Sets *res* to *mat* raised to the power *exp*. Requires that *mat* is a square matrix.

6.2.11 Scalar arithmetic

void **acb_mat_scalar_mul_2exp_si**(*acb_mat_t B*, const *acb_mat_t A*, *slong c*)
Sets B to A multiplied by 2^c.

void **acb_mat_scalar_addmul_si**(*acb_mat_t B*, const *acb_mat_t A*, *slong c*, *slong prec*)

void **acb_mat_scalar_addmul_fmpz**(*acb_mat_t B*, const *acb_mat_t A*, const *fmpz_t c*, *slong prec*)

void **acb_mat_scalar_addmul_arb**(*acb_mat_t B*, const *acb_mat_t A*, const *arb_t c*, *slong prec*)

void **acb_mat_scalar_addmul_acb**(*acb_mat_t B*, const *acb_mat_t A*, const *acb_t c*, *slong prec*)
Sets B to $B + A \times c$.

void **acb_mat_scalar_mul_si**(*acb_mat_t B*, const *acb_mat_t A*, *slong c*, *slong prec*)

void **acb_mat_scalar_mul_fmpz**(*acb_mat_t B*, const *acb_mat_t A*, const *fmpz_t c*, *slong prec*)

void **acb_mat_scalar_mul_arb**(*acb_mat_t B*, const *acb_mat_t A*, const *arb_t c*, *slong prec*)

void **acb_mat_scalar_mul_acb**(*acb_mat_t B*, const *acb_mat_t A*, const *acb_t c*, *slong prec*)
Sets B to $A \times c$.

void **acb_mat_scalar_div_si**(*acb_mat_t B*, const *acb_mat_t A*, *slong c*, *slong prec*)

void **acb_mat_scalar_div_fmpz**(*acb_mat_t B*, const *acb_mat_t A*, const *fmpz_t c*, *slong prec*)

void **acb_mat_scalar_div_arb**(*acb_mat_t B*, const *acb_mat_t A*, const *arb_t c*, *slong prec*)

void **acb_mat_scalar_div_acb**(*acb_mat_t B*, const *acb_mat_t A*, const *acb_t c*, *slong prec*)
Sets B to A/c.

6.2.12 Gaussian elimination and solving

int **acb_mat_lu**(*slong* * *perm*, *acb_mat_t* *LU*, const *acb_mat_t* *A*, *slong* *prec*)
Given an $n \times n$ matrix A, computes an LU decomposition $PLU = A$ using Gaussian elimination with partial pivoting. The input and output matrices can be the same, performing the decomposition in-place.

Entry i in the permutation vector perm is set to the row index in the input matrix corresponding to row i in the output matrix.

The algorithm succeeds and returns nonzero if it can find n invertible (i.e. not containing zero) pivot entries. This guarantees that the matrix is invertible.

The algorithm fails and returns zero, leaving the entries in P and LU undefined, if it cannot find n invertible pivot elements. In this case, either the matrix is singular, the input matrix was computed to insufficient precision, or the LU decomposition was attempted at insufficient precision.

void **acb_mat_solve_lu_precomp**(*acb_mat_t* *X*, const *slong* * *perm*, const *acb_mat_t* *LU*, const *acb_mat_t* *B*, *slong* *prec*)
Solves $AX = B$ given the precomputed nonsingular LU decomposition $A = PLU$. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with LU.

int **acb_mat_solve**(*acb_mat_t* *X*, const *acb_mat_t* *A*, const *acb_mat_t* *B*, *slong* *prec*)
Solves $AX = B$ where A is a nonsingular $n \times n$ matrix and X and B are $n \times m$ matrices, using LU decomposition.

If $m > 0$ and A cannot be inverted numerically (indicating either that A is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that A is invertible and that the exact solution matrix is contained in the output.

int **acb_mat_inv**(*acb_mat_t* *X*, const *acb_mat_t* *A*, *slong* *prec*)
Sets $X = A^{-1}$ where A is a square matrix, computed by solving the system $AX = I$.

If A cannot be inverted numerically (indicating either that A is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the matrix is invertible and that the exact inverse is contained in the output.

void **acb_mat_det**(*acb_t* *det*, const *acb_mat_t* *A*, *slong* *prec*)
Computes the determinant of the matrix, using Gaussian elimination with partial pivoting. If at some point an invertible pivot element cannot be found, the elimination is stopped and the magnitude of the determinant of the remaining submatrix is bounded using Hadamard's inequality.

6.2.13 Characteristic polynomial

void **_acb_mat_charpoly**(*acb_ptr* *cp*, const *acb_mat_t* *mat*, *slong* *prec*)

void **acb_mat_charpoly**(*acb_poly_t* *cp*, const *acb_mat_t* *mat*, *slong* *prec*)
Sets *cp* to the characteristic polynomial of *mat* which must be a square matrix. If the matrix has n rows, the underscore method requires space for $n+1$ output coefficients. Employs a division-free algorithm using $O(n^4)$ operations.

6.2.14 Special functions

void **acb_mat_exp_taylor_sum**(*acb_mat_t* *S*, const *acb_mat_t* *A*, *slong* *N*, *slong* *prec*)
Sets S to the truncated exponential Taylor series $S = \sum_{k=0}^{N-1} A^k/k!$. See *arb_mat_exp_taylor_sum()* for implementation notes.

void **acb_mat_exp**(*acb_mat_t* *B*, const *acb_mat_t* *A*, *slong* *prec*)
Sets B to the exponential of the matrix A, defined by the Taylor series

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

The function is evaluated as $\exp(A/2^r)^{2^r}$, where r is chosen to give rapid convergence of the Taylor series. Error bounds are computed as for *arb_mat_exp()*.

void **acb_mat_trace**(*acb_t* *trace*, const *acb_mat_t* *mat*, *slong* *prec*)
Sets *trace* to the trace of the matrix, i.e. the sum of entries on the main diagonal of *mat*. The matrix is required to be square.

CHAPTER

SEVEN

HIGHER MATHEMATICAL FUNCTIONS

These modules implement mathematical functions with complexity that goes beyond the basics covered directly in the *arb* and *acb* modules.

7.1 acb_hypgeom.h – hypergeometric functions of complex variables

The generalized hypergeometric function is formally defined by

$$ {}_pF_q(a_1,\ldots,a_p;b_1,\ldots,b_q;z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \ldots (a_p)_k}{(b_1)_k \ldots (b_q)_k} \frac{z^k}{k!}. $$

It can be interpreted using analytic continuation or regularization when the sum does not converge. In a looser sense, we understand "hypergeometric functions" to be linear combinations of generalized hypergeometric functions with prefactors that are products of exponentials, powers, and gamma functions.

7.1.1 Convergent series

In this section, we define

$$ T(k) = \frac{\prod_{i=0}^{p-1}(a_i)_k}{\prod_{i=0}^{q-1}(b_i)_k} z^k $$

and

$$ {}_pf_q(a_0,\ldots,a_{p-1};b_0\ldots b_{q-1};z) = {}_{p+1}F_q(a_0,\ldots,a_{p-1},1;b_0\ldots b_{q-1};z) = \sum_{k=0}^{\infty} T(k) $$

For the conventional generalized hypergeometric function ${}_pF_q$, compute ${}_pf_{q+1}$ with the explicit parameter $b_q = 1$, or remove a 1 from the a_i parameters if there is one.

void **acb_hypgeom_pfq_bound_factor**(*mag_t* *C*, *acb_srcptr* *a*, *slong* *p*, *acb_srcptr* *b*, *slong* *q*, const *acb_t* *z*, *ulong* *n*)

Computes a factor C such that $|\sum_{k=n}^{\infty} T(k)| \le C|T(n)|$. See *Convergent series*. As currently implemented, the bound becomes infinite when n is too small, even if the series converges.

slong **acb_hypgeom_pfq_choose_n**(*acb_srcptr* *a*, *slong* *p*, *acb_srcptr* *b*, *slong* *q*, const *acb_t* *z*, *slong* *prec*)

Heuristically attempts to choose a number of terms n to sum of a hypergeometric series at a working precision of *prec* bits.

Uses double precision arithmetic internally. As currently implemented, it can fail to produce a good result if the parameters are extremely large or extremely close to nonpositive integers.

Numerical cancellation is assumed to be significant, so truncation is done when the current term is *prec* bits smaller than the largest encountered term.

This function will also attempt to pick a reasonable truncation point for divergent series.

void **acb_hypgeom_pfq_sum_forward**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, const *acb_t z*, *slong n*, *slong prec*)

void **acb_hypgeom_pfq_sum_rs**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, const *acb_t z*, *slong n*, *slong prec*)

void **acb_hypgeom_pfq_sum_bs**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, const *acb_t z*, *slong n*, *slong prec*)

void **acb_hypgeom_pfq_sum_fme**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, const *acb_t z*, *slong n*, *slong prec*)

void **acb_hypgeom_pfq_sum**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, const *acb_t z*, *slong n*, *slong prec*)

Computes $s = \sum_{k=0}^{n-1} T(k)$ and $t = T(n)$. Does not allow aliasing between input and output variables. We require $n \geq 0$.

The *forward* version computes the sum using forward recurrence.

The *bs* version computes the sum using binary splitting.

The *rs* version computes the sum in reverse order using rectangular splitting. It only computes a magnitude bound for the value of *t*.

The *fme* version uses fast multipoint evaluation.

The default version automatically chooses an algorithm depending on the inputs.

void **acb_hypgeom_pfq_sum_bs_invz**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, const *acb_t w*, *slong n*, *slong prec*)

void **acb_hypgeom_pfq_sum_invz**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, const *acb_t z*, const *acb_t w*, *slong n*, *slong prec*)

Like *acb_hypgeom_pfq_sum()*, but taking advantage of $w = 1/z$ possibly having few bits.

void **acb_hypgeom_pfq_direct**(*acb_t res*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, const *acb_t z*, *slong n*, *slong prec*)

Computes

$$ {}_pf_q(z) = \sum_{k=0}^{\infty} T(k) = \sum_{k=0}^{n-1} T(k) + \varepsilon$$

directly from the defining series, including a rigorous bound for the truncation error ε in the output.

If $n < 0$, this function chooses a number of terms automatically using *acb_hypgeom_pfq_choose_n()*.

void **acb_hypgeom_pfq_series_sum_forward**(*acb_poly_t s*, *acb_poly_t t*, const *acb_poly_struct* * *a*, *slong p*, const *acb_poly_struct* * *b*, *slong q*, const *acb_poly_t z*, int *regularized*, *slong n*, *slong len*, *slong prec*)

void **acb_hypgeom_pfq_series_sum_bs**(*acb_poly_t s*, *acb_poly_t t*, const *acb_poly_struct* * *a*, *slong p*, const *acb_poly_struct* * *b*, *slong q*, const *acb_poly_t z*, int *regularized*, *slong n*, *slong len*, *slong prec*)

void **acb_hypgeom_pfq_series_sum_rs**(*acb_poly_t s*, *acb_poly_t t*, const *acb_poly_struct* * *a*, *slong p*, const *acb_poly_struct* * *b*, *slong q*, const *acb_poly_t z*, int *regularized*, *slong n*, *slong len*, *slong prec*)

void **acb_hypgeom_pfq_series_sum**(*acb_poly_t s*, *acb_poly_t t*, const *acb_poly_struct* * *a*, *slong p*, const *acb_poly_struct* * *b*, *slong q*, const *acb_poly_t z*, int *regularized*, *slong n*, *slong len*, *slong prec*)

Computes $s = \sum_{k=0}^{n-1} T(k)$ and $t = T(n)$ given parameters and argument that are power series. Does not allow aliasing between input and output variables. We require $n \geq 0$ and that *len* is positive.

If *regularized* is set, the regularized sum is computed, avoiding division by zero at the poles of the gamma function.

The *forward*, *bs*, *rs* and default versions use forward recurrence, binary splitting, rectangular splitting, and an automatic algorithm choice.

void **acb_hypgeom_pfq_series_direct**(*acb_poly_t res*, const *acb_poly_struct* * *a*, *slong p*, const *acb_poly_struct* * *b*, *slong q*, const *acb_poly_t z*, int *regularized*, *slong n*, *slong len*, *slong prec*)

Computes ${}_pf_q(z)$ directly using the defining series, given parameters and argument that are power series. The result is a power series of length *len*. We require that *len* is positive.

An error bound is computed automatically as a function of the number of terms n. If $n < 0$, the number of terms is chosen automatically.

If *regularized* is set, the regularized hypergeometric function is computed instead.

7.1.2 Asymptotic series

$U(a, b, z)$ is the confluent hypergeometric function of the second kind with the principal branch cut, and $U^* = z^a U(a, b, z)$. For details about how error bounds are computed, see *Asymptotic series for the confluent hypergeometric function*.

void **acb_hypgeom_u_asymp**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t z*, *slong n*, *slong prec*)

Sets *res* to $U^*(a, b, z)$ computed using n terms of the asymptotic series, with a rigorous bound for the error included in the output. We require $n \geq 0$.

int **acb_hypgeom_u_use_asymp**(const *acb_t z*, *slong prec*)

Heuristically determines whether the asymptotic series can be used to evaluate $U(a, b, z)$ to *prec* accurate bits (assuming that a and b are small).

7.1.3 Generalized hypergeometric function

void **acb_hypgeom_pfq**(*acb_poly_t res*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, const *acb_t z*, int *regularized*, *slong prec*)

Computes the generalized hypergeometric function ${}_pF_q(z)$, or the regularized version if *regularized* is set.

This function automatically delegates to a specialized implementation when the order (p, q) is one of (0,0), (1,0), (0,1), (1,1), (2,1). Otherwise, it falls back to direct summation.

While this is a top-level function meant to take care of special cases automatically, it does not generally perform the optimization of deleting parameters that appear in both a and b. This can be done ahead of time by the user in applications where duplicate parameters are likely to occur.

7.1.4 Confluent hypergeometric functions

void **acb_hypgeom_u_1f1_series**(*acb_poly_t res*, const *acb_poly_t a*, const *acb_poly_t b*, const *acb_poly_t z*, *slong len*, *slong prec*)

Computes $U(a, b, z)$ as a power series truncated to length *len*, given $a, b, z \in \mathbb{C}[[x]]$. If $b[0] \in \mathbb{Z}$, it computes one extra derivative and removes the singularity (it is then assumed that $b[1] \neq 0$). As currently implemented, the output is indeterminate if b is nonexact and contains an integer.

void **acb_hypgeom_u_1f1**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t z*, *slong prec*)

Computes $U(a, b, z)$ as a sum of two convergent hypergeometric series. If $b \in \mathbb{Z}$, it computes the limit value via *acb_hypgeom_u_1f1_series()*. As currently implemented, the output is indeterminate if b is nonexact and contains an integer.

void **acb_hypgeom_u**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t z*, *slong prec*)
Computes $U(a,b,z)$ using an automatic algorithm choice. The function *acb_hypgeom_u_asymp()* is used if a or $a-b+1$ is a nonpositive integer (in which case the asymptotic series terminates), or if z is sufficiently large. Otherwise *acb_hypgeom_u_1f1()* is used.

void **acb_hypgeom_m_asymp**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_m_1f1**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_m**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t z*, int *regularized*, *slong prec*)
Computes the confluent hypergeometric function $M(a,b,z) = {}_1F_1(a,b,z)$, or $\mathbf{M}(a,b,z) = \frac{1}{\Gamma(b)}{}_1F_1(a,b,z)$ if *regularized* is set.

void **acb_hypgeom_1f1**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t z*, int *regularized*, *slong prec*)
Alias for *acb_hypgeom_m()*.

void **acb_hypgeom_0f1_asymp**(*acb_t res*, const *acb_t a*, const *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_0f1_direct**(*acb_t res*, const *acb_t a*, const *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_0f1**(*acb_t res*, const *acb_t a*, const *acb_t z*, int *regularized*, *slong prec*)
Computes the confluent hypergeometric function ${}_0F_1(a,z)$, or $\frac{1}{\Gamma(a)}{}_0F_1(a,z)$ if *regularized* is set, using asymptotic expansions, direct summation, or an automatic algorithm choice. The *asymp* version uses the asymptotic expansions of Bessel functions, together with the connection formulas

$$\frac{{}_0F_1(a,z)}{\Gamma(a)} = (-z)^{(1-a)/2} J_{a-1}(2\sqrt{-z}) = z^{(1-a)/2} I_{a-1}(2\sqrt{z}).$$

The Bessel-J function is used in the left half-plane and the Bessel-I function is used in the right half-plane, to avoid loss of accuracy due to evaluating the square root on the branch cut.

7.1.5 Error functions and Fresnel integrals

void **acb_hypgeom_erf_propagated_error**(*mag_t re*, *mag_t im*, const *acb_t z*)
Sets *re* and *im* to upper bounds for the error in the real and imaginary part resulting from approximating the error function of z by the error function evaluated at the midpoint of z. Uses the first derivative.

void **acb_hypgeom_erf_1f1a**(*acb_t res*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_erf_1f1b**(*acb_t res*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_erf_asymp**(*acb_t res*, const *acb_t z*, int *complementary*, *slong prec*, *slong prec2*)
Computes the error function respectively using

$$\operatorname{erf}(z) = \frac{2z}{\sqrt{\pi}} {}_1F_1(\tfrac{1}{2}, \tfrac{3}{2}, -z^2)$$

$$\operatorname{erf}(z) = \frac{2ze^{-z^2}}{\sqrt{\pi}} {}_1F_1(1, \tfrac{3}{2}, z^2)$$

$$\operatorname{erf}(z) = \frac{z}{\sqrt{z^2}} \left(1 - \frac{e^{-z^2}}{\sqrt{\pi}} U(\tfrac{1}{2}, \tfrac{1}{2}, z^2)\right) = \frac{z}{\sqrt{z^2}} - \frac{e^{-z^2}}{z\sqrt{\pi}} U^*(\tfrac{1}{2}, \tfrac{1}{2}, z^2).$$

The *asymp* version takes a second precision to use for the U term. It also takes an extra flag *complementary*, computing the complementary error function if set.

void **acb_hypgeom_erf**(*acb_t res*, const *acb_t z*, *slong prec*)
Computes the error function using an automatic algorithm choice. If z is too small to use the

asymptotic expansion, a working precision sufficient to circumvent cancellation in the hypergeometric series is determined automatically, and a bound for the propagated error is computed with *acb_hypgeom_erf_propagated_error()*.

void **_acb_hypgeom_erf_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_erf_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong len*, *slong prec*)
Computes the error function of the power series z, truncated to length *len*.

void **acb_hypgeom_erfc**(*acb_t res*, const *acb_t z*, *slong prec*)
Computes the complementary error function $\operatorname{erfc}(z) = 1 - \operatorname{erf}(z)$. This function avoids catastrophic cancellation for large positive z.

void **_acb_hypgeom_erfc_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_erfc_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong len*, *slong prec*)
Computes the complementary error function of the power series z, truncated to length *len*.

void **acb_hypgeom_erfi**(*acb_t res*, const *acb_t z*, *slong prec*)
Computes the imaginary error function $\operatorname{erfi}(z) = -i \operatorname{erf}(iz)$. This is a trivial wrapper of *acb_hypgeom_erf()*.

void **_acb_hypgeom_erfi_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_erfi_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong len*, *slong prec*)
Computes the imaginary error function of the power series z, truncated to length *len*.

void **acb_hypgeom_fresnel**(*acb_t res1*, *acb_t res2*, const *acb_t z*, int *normalized*, *slong prec*)
Sets *res1* to the Fresnel sine integral $S(z)$ and *res2* to the Fresnel cosine integral $C(z)$. Optionally, just a single function can be computed by passing *NULL* as the other output variable. The definition $S(z) = \int_0^z \sin(t^2) dt$ is used if *normalized* is 0, and $S(z) = \int_0^z \sin(\frac{1}{2}\pi t^2) dt$ is used if *normalized* is 1 (the latter is the Abramowitz & Stegun convention). $C(z)$ is defined analogously.

void **_acb_hypgeom_fresnel_series**(*acb_ptr res1*, *acb_ptr res2*, *acb_srcptr z*, *slong zlen*, int *normalized*, *slong len*, *slong prec*)

void **acb_hypgeom_fresnel_series**(*acb_poly_t res1*, *acb_poly_t res2*, const *acb_poly_t z*, int *normalized*, *slong len*, *slong prec*)
Sets *res1* to the Fresnel sine integral and *res2* to the Fresnel cosine integral of the power series z, truncated to length *len*. Optionally, just a single function can be computed by passing *NULL* as the other output variable.

7.1.6 Bessel functions

void **acb_hypgeom_bessel_j_asymp**(*acb_t res*, const *acb_t nu*, const *acb_t z*, *slong prec*)
Computes the Bessel function of the first kind via *acb_hypgeom_u_asymp()*. For all complex ν, z, we have

$$J_\nu(z) = \frac{z^\nu}{2^\nu e^{iz} \Gamma(\nu+1)} {}_1F_1(\nu + \tfrac{1}{2}, 2\nu+1, 2iz) = A_+ B_+ + A_- B_-$$

where

$$A_\pm = z^\nu (z^2)^{-\frac{1}{2}-\nu} (\mp iz)^{\frac{1}{2}+\nu} (2\pi)^{-1/2} = (\pm iz)^{-1/2-\nu} z^\nu (2\pi)^{-1/2}$$

$$B_\pm = e^{\pm iz} U^*(\nu + \tfrac{1}{2}, 2\nu+1, \mp 2iz).$$

Nicer representations of the factors $A_\pm$ can be given depending conditionally on the parameters. If $\nu + \frac{1}{2} = n \in \mathbb{Z}$, we have $A_\pm = (\pm i)^n (2\pi z)^{-1/2}$. And if $\operatorname{Re}(z) > 0$, we have $A_\pm = \exp(\mp i[(2\nu+1)/4]\pi)(2\pi z)^{-1/2}$.

void **acb_hypgeom_bessel_j_0f1**(*acb_t res*, const *acb_t nu*, const *acb_t z*, *slong prec*)
Computes the Bessel function of the first kind from

$$J_\nu(z) = \frac{1}{\Gamma(\nu+1)} \left(\frac{z}{2}\right)^\nu {}_0F_1\left(\nu+1, -\frac{z^2}{4}\right).$$

void **acb_hypgeom_bessel_j**(*acb_t res*, const *acb_t nu*, const *acb_t z*, *slong prec*)
Computes the Bessel function of the first kind $J_\nu(z)$ using an automatic algorithm choice.

void **acb_hypgeom_bessel_y**(*acb_t res*, const *acb_t nu*, const *acb_t z*, *slong prec*)
Computes the Bessel function of the second kind $Y_\nu(z)$ from the formula

$$Y_\nu(z) = \frac{\cos(\nu\pi)J_\nu(z) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

unless $\nu = n$ is an integer in which case the limit value

$$Y_n(z) = -\frac{2}{\pi}\left(i^n K_n(iz) + \left[\log(iz) - \log(z)\right] J_n(z)\right)$$

is computed. As currently implemented, the output is indeterminate if ν is nonexact and contains an integer.

void **acb_hypgeom_bessel_jy**(*acb_t res1*, *acb_t res2*, const *acb_t nu*, const *acb_t z*, *slong prec*)
Sets *res1* to $J_\nu(z)$ and *res2* to $Y_\nu(z)$, computed simultaneously. From these values, the user can easily construct the Bessel functions of the third kind (Hankel functions) $H_\nu^{(1)}(z), H_\nu^{(2)}(z) = J_\nu(z) \pm iY_\nu(z)$.

void **acb_hypgeom_bessel_i_asymp**(*acb_t res*, const *acb_t nu*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_bessel_i_0f1**(*acb_t res*, const *acb_t nu*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_bessel_i**(*acb_t res*, const *acb_t nu*, const *acb_t z*, *slong prec*)
Computes the modified Bessel function of the first kind $I_\nu(z) = z^\nu(iz)^{-\nu}J_\nu(iz)$ respectively using asymptotic series (see *acb_hypgeom_bessel_j_asymp()*), the convergent series

$$I_\nu(z) = \frac{1}{\Gamma(\nu+1)}\left(\frac{z}{2}\right)^\nu {}_0F_1\left(\nu+1, \frac{z^2}{4}\right),$$

or an automatic algorithm choice.

void **acb_hypgeom_bessel_k_asymp**(*acb_t res*, const *acb_t nu*, const *acb_t z*, *slong prec*)
Computes the modified Bessel function of the second kind via via *acb_hypgeom_u_asymp()*. For all ν and all $z \neq 0$, we have

$$K_\nu(z) = \left(\frac{2z}{\pi}\right)^{-1/2} e^{-z} U^*(\nu + \tfrac{1}{2}, 2\nu + 1, 2z).$$

void **acb_hypgeom_bessel_k_0f1_series**(*acb_poly_t res*, const *acb_poly_t nu*, const *acb_poly_t z*, *slong len*, *slong prec*)
Computes the modified Bessel function of the second kind $K_\nu(z)$ as a power series truncated to length *len*, given $\nu, z \in \mathbb{C}[[x]]$. Uses the formula

$$K_\nu(z) = \frac{1}{2}\frac{\pi}{\sin(\pi\nu)}\left[\left(\frac{z}{2}\right)^{-\nu} {}_0\widetilde{F}_1\left(1-\nu, \frac{z^2}{4}\right) - \left(\frac{z}{2}\right)^{\nu} {}_0\widetilde{F}_1\left(1+\nu, \frac{z^2}{4}\right)\right].$$

If $\nu[0] \in \mathbb{Z}$, it computes one extra derivative and removes the singularity (it is then assumed that $\nu[1] \neq 0$). As currently implemented, the output is indeterminate if $\nu[0]$ is nonexact and contains an integer.

void **acb_hypgeom_bessel_k_0f1**(*acb_t res*, const *acb_t nu*, const *acb_t z*, *slong prec*)
Computes the modified Bessel function of the second kind from

$$K_\nu(z) = \frac{1}{2}\left[\left(\frac{z}{2}\right)^{-\nu}\Gamma(\nu){}_0F_1\left(1-\nu, \frac{z^2}{4}\right) - \left(\frac{z}{2}\right)^{\nu}\frac{\pi}{\nu\sin(\pi\nu)\Gamma(\nu)}{}_0F_1\left(\nu+1, \frac{z^2}{4}\right)\right]$$

if $\nu \notin \mathbb{Z}$. If $\nu \in \mathbb{Z}$, it computes the limit value via *acb_hypgeom_bessel_k_0f1_series()*. As currently implemented, the output is indeterminate if ν is nonexact and contains an integer.

void **acb_hypgeom_bessel_k**(*acb_t res*, const *acb_t nu*, const *acb_t z*, *slong prec*)
Computes the modified Bessel function of the second kind $K_\nu(z)$ using an automatic algorithm choice.

7.1.7 Airy functions

The Airy functions are linearly independent solutions of the differential equation $y'' - zy = 0$. All solutions are entire functions. The standard solutions are denoted $\mathrm{Ai}(z), \mathrm{Bi}(z)$. For negative z, both functions are oscillatory. For positive z, the first function decreases exponentially while the second increases exponentially.

The Airy functions can be expressed in terms of Bessel functions of fractional order, but this is inconvenient since such formulas only hold piecewise (due to the Stokes phenomenon). Computation of the Airy functions can also be optimized more than Bessel functions in general. We therefore provide a dedicated interface for evaluating Airy functions.

The following methods optionally compute $(\mathrm{Ai}(z), \mathrm{Ai}'(z), \mathrm{Bi}(z), \mathrm{Bi}'(z))$ simultaneously. Any of the four function values can be omitted by passing *NULL* for the unwanted output variables, speeding up the evaluation.

void **acb_hypgeom_airy_direct**(*acb_t ai*, *acb_t ai_prime*, *acb_t bi*, *acb_t bi_prime*, const *acb_t z*, *slong n*, *slong prec*)
Computes the Airy functions using direct series expansions truncated at n terms. Error bounds are included in the output.

void **acb_hypgeom_airy_asymp**(*acb_t ai*, *acb_t ai_prime*, *acb_t bi*, *acb_t bi_prime*, const *acb_t z*, *slong n*, *slong prec*)
Computes the Airy functions using asymptotic expansions truncated at n terms. Error bounds are included in the output. For details about how the error bounds are computed, see *Asymptotic series for Airy functions*.

void **acb_hypgeom_airy_bound**(*mag_t ai*, *mag_t ai_prime*, *mag_t bi*, *mag_t bi_prime*, const *acb_t z*)
Computes bounds for the Airy functions using first-order asymptotic expansions together with error bounds. This function uses some shortcuts to make it slightly faster than calling *acb_hypgeom_airy_asymp()* with $n = 1$.

void **acb_hypgeom_airy**(*acb_t ai*, *acb_t ai_prime*, *acb_t bi*, *acb_t bi_prime*, const *acb_t z*, *slong prec*)
Computes Airy functions using an automatic algorithm choice.

We use *acb_hypgeom_airy_asymp()* whenever this gives full accuracy and *acb_hypgeom_airy_direct()* otherwise. In the latter case, we first use hardware double precision arithmetic to determine an accurate estimate of the working precision needed to compute the Airy functions accurately for given z. This estimate is obtained by comparing the leading-order asymptotic estimate of the Airy functions with the magnitude of the largest term in the power series. The estimate is generic in the sense that it does not take into account vanishing near the roots of the functions. We subsequently evaluate the power series at the midpoint of z and bound the propagated error using derivatives. Derivatives are bounded using *acb_hypgeom_airy_bound()*.

void **acb_hypgeom_airy_jet**(*acb_ptr ai*, *acb_ptr bi*, const *acb_t z*, *slong len*, *slong prec*)
Writes to *ai* and *bi* the respective Taylor expansions of the Airy functions at the point z, truncated to length *len*. Either of the outputs can be *NULL* to avoid computing that function. The variable z is not allowed to be aliased with the outputs. To simplify the implementation, this method does not compute the series expansions of the primed versions directly; these are easily obtained by computing one extra coefficient and differentiating the output with *_acb_poly_derivative()*.

void **_acb_hypgeom_airy_series**(*acb_ptr ai*, *acb_ptr ai_prime*, *acb_ptr bi*, *acb_ptr bi_prime*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_airy_series**(*acb_poly_t ai*, *acb_poly_t ai_prime*, *acb_poly_t bi*, *acb_poly_t bi_prime*, const *acb_poly_t z*, *slong len*, *slong prec*)
Computes the Airy functions evaluated at the power series z, truncated to length *len*. As with the other Airy methods, any of the outputs can be *NULL*.

7.1.8 Incomplete gamma and beta functions

void **acb_hypgeom_gamma_upper_asymp**(*acb_t res*, const *acb_t s*, const *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_gamma_upper_1f1a**(*acb_t res*, const *acb_t s*, const *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_gamma_upper_1f1b**(*acb_t res*, const *acb_t s*, const *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_gamma_upper_singular**(*acb_t res*, *slong s*, const *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_gamma_upper**(*acb_t res*, const *acb_t s*, const *acb_t z*, int *regularized*, *slong prec*)

If *regularized* is 0, computes the upper incomplete gamma function $\Gamma(s,z)$.

If *regularized* is 1, computes the regularized upper incomplete gamma function $Q(s,z) = \Gamma(s,z)/\Gamma(s)$.

If *regularized* is 2, computes the generalized exponential integral $z^{-s}\Gamma(s,z) = E_{1-s}(z)$ instead (this option is mainly intended for internal use; *acb_hypgeom_expint()* is the intended interface for computing the exponential integral).

The different methods respectively implement the formulas

$$\Gamma(s,z) = e^{-z} U(1-s, 1-s, z)$$

$$\Gamma(s,z) = \Gamma(s) - \frac{z^s}{s} {}_1F_1(s, s+1, -z)$$

$$\Gamma(s,z) = \Gamma(s) - \frac{z^s e^{-z}}{s} {}_1F_1(1, s+1, z)$$

$$\Gamma(s,z) = \frac{(-1)^n}{n!}(\psi(n+1) - \log(z)) + \frac{(-1)^n}{(n+1)!} z\, {}_2F_2(1,1,2,2+n,-z) - z^{-n} \sum_{k=0}^{n-1} \frac{(-z)^k}{(k-n)k!}, \quad n = -s \in \mathbb{Z}_{\geq 0}$$

and an automatic algorithm choice. The automatic version also handles other special input such as $z = 0$ and $s = 1, 2, 3$. The *singular* version evaluates the finite sum directly and therefore assumes that s is not too large.

void **_acb_hypgeom_gamma_upper_series**(*acb_ptr res*, const *acb_t s*, *acb_srcptr z*, *slong zlen*, int *regularized*, *slong n*, *slong prec*)

void **acb_hypgeom_gamma_upper_series**(*acb_poly_t res*, const *acb_t s*, const *acb_poly_t z*, int *regularized*, *slong n*, *slong prec*)

Sets *res* to an upper incomplete gamma function where s is a constant and z is a power series, truncated to length n. The *regularized* argument has the same interpretation as in *acb_hypgeom_gamma_upper()*.

void **acb_hypgeom_gamma_lower**(*acb_t res*, const *acb_t s*, const *acb_t z*, int *regularized*, *slong prec*)

If *regularized* is 0, computes the lower incomplete gamma function $\gamma(s,z) = \frac{z^s}{s} {}_1F_1(s, s+1, -z)$.

If *regularized* is 1, computes the regularized lower incomplete gamma function $P(s,z) = \gamma(s,z)/\Gamma(s)$.

If *regularized* is 2, computes a further regularized lower incomplete gamma function $\gamma^*(s,z) = z^{-s}P(s,z)$.

void **_acb_hypgeom_gamma_lower_series**(*acb_ptr res*, const *acb_t s*, *acb_srcptr z*, *slong zlen*, int *regularized*, *slong n*, *slong prec*)

void **acb_hypgeom_gamma_lower_series**(*acb_poly_t res*, const *acb_t s*, const *acb_poly_t z*, int *regularized*, *slong n*, *slong prec*)
Sets *res* to an lower incomplete gamma function where s is a constant and z is a power series, truncated to length n. The *regularized* argument has the same interpretation as in *acb_hypgeom_gamma_lower()*.

void **acb_hypgeom_beta_lower**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t z*, int *regularized*, *slong prec*)
Computes the (lower) incomplete beta function, defined by $B(a,b;z) = \int_0^z t^{a-1}(1-t)^{b-1}$, optionally the regularized incomplete beta function $I(a,b;z) = B(a,b;z)/B(a,b;1)$.

In general, the integral must be interpreted using analytic continuation. The precise definitions for all parameter values are

$$B(a,b;z) = \frac{z^a}{a} {}_2F_1(a, 1-b, a+1, z)$$

$$I(a,b;z) = \frac{\Gamma(a+b)}{\Gamma(b)} z^a {}_2\widetilde{F}_1(a, 1-b, a+1, z).$$

Note that both functions with this definition are undefined for nonpositive integer a, and I is undefined for nonpositive integer $a+b$.

void **_acb_hypgeom_beta_lower_series**(*acb_ptr res*, const *acb_t a*, const *acb_t b*, *acb_srcptr z*, *slong zlen*, int *regularized*, *slong n*, *slong prec*)

void **acb_hypgeom_beta_lower_series**(*acb_poly_t res*, const *acb_t a*, const *acb_t b*, const *acb_poly_t z*, int *regularized*, *slong n*, *slong prec*)
Sets *res* to the lower incomplete beta function $B(a,b;z)$ (optionally the regularized version $I(a,b;z)$) where a and b are constants and z is a power series, truncating the result to length n. The underscore method requires positive lengths and does not support aliasing.

7.1.9 Exponential and trigonometric integrals

The branch cut conventions of the following functions match Mathematica.

void **acb_hypgeom_expint**(*acb_t res*, const *acb_t s*, const *acb_t z*, *slong prec*)
Computes the generalized exponential integral $E_s(z)$. This is a trivial wrapper of *acb_hypgeom_gamma_upper()*.

void **acb_hypgeom_ei_asymp**(*acb_t res*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_ei_2f2**(*acb_t res*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_ei**(*acb_t res*, const *acb_t z*, *slong prec*)
Computes the exponential integral $\mathrm{Ei}(z)$, respectively using

$$\mathrm{Ei}(z) = -e^z U(1,1,-z) - \log(-z) + \frac{1}{2}\left(\log(z) - \log\left(\frac{1}{z}\right)\right)$$

$$\mathrm{Ei}(z) = z\,{}_2F_2(1,1;2,2;z) + \gamma + \frac{1}{2}\left(\log(z) - \log\left(\frac{1}{z}\right)\right)$$

and an automatic algorithm choice.

void **_acb_hypgeom_ei_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_ei_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong len*, *slong prec*)
Computes the exponential integral of the power series z, truncated to length *len*.

void **acb_hypgeom_si_asymp**(*acb_t res*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_si_1f2**(*acb_t res*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_si**(*acb_t res*, const *acb_t z*, *slong prec*)
Computes the sine integral $\operatorname{Si}(z)$, respectively using

$$\operatorname{Si}(z) = \frac{i}{2}\left[e^{iz}U(1,1,-iz) - e^{-iz}U(1,1,iz) + \log(-iz) - \log(iz)\right]$$

$$\operatorname{Si}(z) = z\,{}_1F_2(\tfrac{1}{2}; \tfrac{3}{2}, \tfrac{3}{2}; -\tfrac{z^2}{4})$$

and an automatic algorithm choice.

void **_acb_hypgeom_si_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_si_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong len*, *slong prec*)
Computes the sine integral of the power series z, truncated to length *len*.

void **acb_hypgeom_ci_asymp**(*acb_t res*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_ci_2f3**(*acb_t res*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_ci**(*acb_t res*, const *acb_t z*, *slong prec*)
Computes the cosine integral $\operatorname{Ci}(z)$, respectively using

$$\operatorname{Ci}(z) = \log(z) - \frac{1}{2}\left[e^{iz}U(1,1,-iz) + e^{-iz}U(1,1,iz) + \log(-iz) + \log(iz)\right]$$

$$\operatorname{Ci}(z) = -\tfrac{z^2}{4}\,{}_2F_3(1,1;2,2,\tfrac{3}{2};-\tfrac{z^2}{4}) + \log(z) + \gamma$$

and an automatic algorithm choice.

void **_acb_hypgeom_ci_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_ci_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong len*, *slong prec*)
Computes the cosine integral of the power series z, truncated to length *len*.

void **acb_hypgeom_shi**(*acb_t res*, const *acb_t z*, *slong prec*)
Computes the hyperbolic sine integral $\operatorname{Shi}(z) = -i\operatorname{Si}(iz)$. This is a trivial wrapper of *acb_hypgeom_si()*.

void **_acb_hypgeom_shi_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_shi_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong len*, *slong prec*)
Computes the hyperbolic sine integral of the power series z, truncated to length *len*.

void **acb_hypgeom_chi_asymp**(*acb_t res*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_chi_2f3**(*acb_t res*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_chi**(*acb_t res*, const *acb_t z*, *slong prec*)
Computes the hyperbolic cosine integral $\operatorname{Chi}(z)$, respectively using

$$\operatorname{Chi}(z) = -\frac{1}{2}\left[e^{z}U(1,1,-z) + e^{-z}U(1,1,z) + \log(-z) - \log(z)\right]$$

$$\operatorname{Chi}(z) = \tfrac{z^2}{4}\,{}_2F_3(1,1;2,2,\tfrac{3}{2};\tfrac{z^2}{4}) + \log(z) + \gamma$$

and an automatic algorithm choice.

void **_acb_hypgeom_chi_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_chi_series**(*acb_poly_t res*, const *acb_poly_t z*, *slong len*, *slong prec*)
Computes the hyperbolic cosine integral of the power series z, truncated to length *len*.

void **acb_hypgeom_li**(*acb_t res*, const *acb_t z*, int *offset*, *slong prec*)
If *offset* is zero, computes the logarithmic integral $\operatorname{li}(z) = \operatorname{Ei}(\log(z))$.

If *offset* is nonzero, computes the offset logarithmic integral $\operatorname{Li}(z) = \operatorname{li}(z) - \operatorname{li}(2)$.

void **_acb_hypgeom_li_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, int *offset*, *slong len*, *slong prec*)

void **acb_hypgeom_li_series**(*acb_poly_t res*, const *acb_poly_t z*, int *offset*, *slong len*, *slong prec*)

Computes the logarithmic integral (optionally the offset version) of the power series z, truncated to length *len*.

7.1.10 Gauss hypergeometric function

The following methods compute the Gauss hypergeometric function

$$F(z) = {}_2F_1(a,b,c,z) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \frac{z^k}{k!}$$

or the regularized version $\mathbf{F}(z) = \mathbf{F}(a,b,c,z) = {}_2F_1(a,b,c,z)/\Gamma(c)$ if the flag *regularized* is set.

void **acb_hypgeom_2f1_continuation**(*acb_t res0*, *acb_t res1*, const *acb_t a*, const *acb_t b*, const *acb_t c*, const *acb_t z0*, const *acb_t z1*, const *acb_t f0*, const *acb_t f1*, *slong prec*)

Given $F(z_0), F'(z_0)$ in *f0*, *f1*, sets *res0* and *res1* to $F(z_1), F'(z_1)$ by integrating the hypergeometric differential equation along a straight-line path. The evaluation points should be well-isolated from the singular points 0 and 1.

void **acb_hypgeom_2f1_series_direct**(*acb_poly_t res*, const *acb_poly_t a*, const *acb_poly_t b*, const *acb_poly_t c*, const *acb_poly_t z*, int *regularized*, *slong len*, *slong prec*)

Computes $F(z)$ of the given power series truncated to length *len*, using direct summation of the hypergeometric series.

void **acb_hypgeom_2f1_direct**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t c*, const *acb_t z*, int *regularized*, *slong prec*)

Computes $F(z)$ using direct summation of the hypergeometric series.

void **acb_hypgeom_2f1_transform**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t c*, const *acb_t z*, int *flags*, int *which*, *slong prec*)

void **acb_hypgeom_2f1_transform_limit**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t c*, const *acb_t z*, int *regularized*, int *which*, *slong prec*)

Computes $F(z)$ using an argument transformation determined by the flag *which*. Legal values are 1 for $z/(z-1)$, 2 for $1/z$, 3 for $1/(1-z)$, 4 for $1-z$, and 5 for $1-1/z$.

The *transform_limit* version assumes that *which* is not 1. If *which* is 2 or 3, it assumes that $b-a$ represents an exact integer. If *which* is 4 or 5, it assumes that $c-a-b$ represents an exact integer. In these cases, it computes the correct limit value.

See *acb_hypgeom_2f1()* for the meaning of *flags*.

void **acb_hypgeom_2f1_corner**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t c*, const *acb_t z*, int *regularized*, *slong prec*)

Computes $F(z)$ near the corner cases $\exp(\pm\pi i\sqrt{3})$ by analytic continuation.

int **acb_hypgeom_2f1_choose**(const *acb_t z*)

Chooses a method to compute the function based on the location of z in the complex plane. If the return value is 0, direct summation should be used. If the return value is 1 to 5, the transformation with this index in *acb_hypgeom_2f1_transform()* should be used. If the return value is 6, the corner case algorithm should be used.

void **acb_hypgeom_2f1**(*acb_t res*, const *acb_t a*, const *acb_t b*, const *acb_t c*, const *acb_t z*, int *flags*, *slong prec*)

Computes $F(z)$ or $\mathbf{F}(z)$ using an automatic algorithm choice.

The following bit fields can be set in *flags*:

- *ACB_HYPGEOM_2F1_REGULARIZED* - computes the regularized hypergeometric function $\mathbf{F}(z)$. Setting *flags* to 1 is the same as just toggling this option.
- *ACB_HYPGEOM_2F1_AB* - $a - b$ is an integer.
- *ACB_HYPGEOM_2F1_ABC* - $a + b - c$ is an integer.
- *ACB_HYPGEOM_2F1_AC* - $a - c$ is an integer.
- *ACB_HYPGEOM_2F1_BC* - $b - c$ is an integer.

The last four flags can be set to indicate that the respective parameter differences are known to represent exact integers, even if the input intervals are inexact. This allows the correct limits to be evaluated when applying transformation formulas. For example, to evaluate ${}_2F_1(\sqrt{2}, 1/2, \sqrt{2} + 3/2, 9/10)$, the *ABC* flag should be set. If not set, the result will be an indeterminate interval due to internally dividing by an interval containing zero. If the parameters are exact floating-point numbers (including exact integers or half-integers), then the limits are computed automatically, and setting these flags is unnecessary.

Currently, only the *AB* and *ABC* flags are used this way; the *AC* and *BC* flags might be used in the future.

7.1.11 Orthogonal polynomials and functions

void **acb_hypgeom_chebyshev_t**(*acb_t res*, const *acb_t n*, const *acb_t z*, *slong prec*)

void **acb_hypgeom_chebyshev_u**(*acb_t res*, const *acb_t n*, const *acb_t z*, *slong prec*)

Computes the Chebyshev polynomial (or Chebyshev function) of first or second kind

$$T_n(z) = {}_2F_1\left(-n, n, \frac{1}{2}, \frac{1-z}{2}\right)$$

$$U_n(z) = (n+1){}_2F_1\left(-n, n+2, \frac{3}{2}, \frac{1-z}{2}\right).$$

The hypergeometric series definitions are only used for computation near the point 1. In general, trigonometric representations are used. For word-size integer n, `acb_chebyshev_t_ui()` and `acb_chebyshev_u_ui()` are called.

void **acb_hypgeom_jacobi_p**(*acb_t res*, const *acb_t n*, const *acb_t a*, const *acb_t b*, const *acb_t z*, *slong prec*)

Computes the Jacobi polynomial (or Jacobi function)

$$P_n^{(a,b)}(z) = \frac{(a+1)_n}{\Gamma(n+1)} {}_2F_1\left(-n, n+a+b+1, a+1, \frac{1-z}{2}\right).$$

For nonnegative integer n, this is a polynomial in a, b and z, even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

void **acb_hypgeom_gegenbauer_c**(*acb_t res*, const *acb_t n*, const *acb_t m*, const *acb_t z*, *slong prec*)

Computes the Gegenbauer polynomial (or Gegenbauer function)

$$C_n^m(z) = \frac{(2m)_n}{\Gamma(n+1)} {}_2F_1\left(-n, 2m+n, m+\frac{1}{2}, \frac{1-z}{2}\right).$$

For nonnegative integer n, this is a polynomial in m and z, even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

void **acb_hypgeom_laguerre_l**(*acb_t res*, const *acb_t n*, const *acb_t m*, const *acb_t z*, *slong prec*)

Computes the Laguerre polynomial (or Laguerre function)

$$L_n^m(z) = \frac{(m+1)_n}{\Gamma(n+1)} {}_1F_1\left(-n, m+1, z\right).$$

For nonnegative integer n, this is a polynomial in m and z, even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

There are at least two incompatible ways to define the Laguerre function when n is a negative integer. One possibility when $m = 0$ is to define $L_{-n}^0(z) = e^z L_{n-1}^0(-z)$. Another possibility is to cover this case with the recurrence relation $L_{n-1}^m(z) + L_n^{m-1}(z) = L_n^m(z)$. Currently, we leave this case undefined (returning indeterminate).

void **acb_hypgeom_hermite_h**(*acb_t res*, const *acb_t n*, const *acb_t z*, *slong prec*)

Computes the Hermite polynomial (or Hermite function)

$$H_n(z) = 2^n \sqrt{\pi} \left(\frac{1}{\Gamma((1-n)/2)} {}_1F_1\left(-\frac{n}{2}, \frac{1}{2}, z^2\right) - \frac{2z}{\Gamma(-n/2)} {}_1F_1\left(\frac{1-n}{2}, \frac{3}{2}, z^2\right)\right).$$

void **acb_hypgeom_legendre_p**(*acb_t res*, const *acb_t n*, const *acb_t m*, const *acb_t z*, int *type*, *slong prec*)

Sets *res* to the associated Legendre function of the first kind evaluated for degree n, order m, and argument z. When m is zero, this reduces to the Legendre polynomial $P_n(z)$.

Many different branch cut conventions appear in the literature. If *type* is 0, the version

$$P_n^m(z) = \frac{(1+z)^{m/2}}{(1-z)^{m/2}} \mathbf{F}\left(-n, n+1, 1-m, \frac{1-z}{2}\right)$$

is computed, and if *type* is 1, the alternative version

$$\mathcal{P}_n^m(z) = \frac{(z+1)^{m/2}}{(z-1)^{m/2}} \mathbf{F}\left(-n, n+1, 1-m, \frac{1-z}{2}\right).$$

is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in *Mathematica* and *mpmath*.

void **acb_hypgeom_legendre_q**(*acb_t res*, const *acb_t n*, const *acb_t m*, const *acb_t z*, int *type*, *slong prec*)

Sets *res* to the associated Legendre function of the second kind evaluated for degree n, order m, and argument z. When m is zero, this reduces to the Legendre function $Q_n(z)$.

Many different branch cut conventions appear in the literature. If *type* is 0, the version

$$Q_n^m(z) = \frac{\pi}{2\sin(\pi m)} \left(\cos(\pi m) P_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} P_n^{-m}(z)\right)$$

is computed, and if *type* is 1, the alternative version

$$\mathcal{Q}_n^m(z) = \frac{\pi}{2\sin(\pi m)} e^{\pi i m} \left(\mathcal{P}_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} \mathcal{P}_n^{-m}(z)\right)$$

is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in *Mathematica* and *mpmath*.

When m is an integer, either expression is interpreted as a limit. We make use of the connection formulas *[WQ3a]*, *[WQ3b]* and *[WQ3c]* to allow computing the function even in the limiting case. (The formula *[WQ3d]* would be useful, but is incorrect in the lower half plane.)

void **acb_hypgeom_legendre_p_uiui_rec**(*acb_t res*, *ulong n*, *ulong m*, const *acb_t z*, *slong prec*)

For nonnegative integer n and m, uses recurrence relations to evaluate $(1-z^2)^{-m/2} P_n^m(z)$ which is a polynomial in z.

void **acb_hypgeom_spherical_y**(*acb_t res*, *slong n*, *slong m*, const *acb_t theta*, const *acb_t phi*, *slong prec*)

Computes the spherical harmonic of degree n, order m, latitude angle *theta*, and longitude angle *phi*, normalized such that

$$Y_n^m(\theta,\phi) = \sqrt{\frac{2n+1}{4\pi}\frac{(n-m)!}{(n+m)!}} e^{im\phi} P_n^m(\cos(\theta)).$$

The definition is extended to negative m and n by symmetry. This function is a polynomial in $\cos(\theta)$ and $\sin(\theta)$. We evaluate it using *acb_hypgeom_legendre_p_uiui_rec()*.

7.2 arb_hypgeom.h – hypergeometric functions of real variables

See *acb_hypgeom.h – hypergeometric functions of complex variables* for the implementation of hypergeometric functions.

For convenience, this module provides corresponding functions for direct use with the real types *arb_t* and *arb_poly_t*. Most methods are simple wrappers around the complex versions, with a tiny amount of extra overhead for conversions, but in some cases the functions in this module will be faster. In the future, code that is further optimized specifically for real variables might be added to this module.

7.2.1 Generalized hypergeometric function

void **arb_hypgeom_pfq**(*arb_t res*, *arb_srcptr a*, *slong p*, *arb_srcptr b*, *slong q*, const *arb_t z*, int *regularized*, *slong prec*)

Computes the generalized hypergeometric function ${}_pF_q(z)$, or the regularized version if *regularized* is set.

7.2.2 Confluent hypergeometric functions

void **arb_hypgeom_0f1**(*arb_t res*, const *arb_t a*, const *arb_t z*, int *regularized*, *slong prec*)

Computes the confluent hypergeometric limit function ${}_0F_1(a,z)$, or $\frac{1}{\Gamma(a)}{}_0F_1(a,z)$ if *regularized* is set.

void **arb_hypgeom_m**(*arb_t res*, const *arb_t a*, const *arb_t b*, const *arb_t z*, int *regularized*, *slong prec*)

Computes the confluent hypergeometric function $M(a,b,z) = {}_1F_1(a,b,z)$, or $\mathbf{M}(a,b,z) = \frac{1}{\Gamma(b)}{}_1F_1(a,b,z)$ if *regularized* is set.

void **arb_hypgeom_1f1**(*arb_t res*, const *arb_t a*, const *arb_t b*, const *arb_t z*, int *regularized*, *slong prec*)

Alias for *arb_hypgeom_m()*.

void **arb_hypgeom_u**(*arb_t res*, const *arb_t a*, const *arb_t b*, const *arb_t z*, *slong prec*)

Computes the confluent hypergeometric function $U(a,b,z)$.

7.2.3 Gauss hypergeometric function

void **arb_hypgeom_2f1**(*arb_t res*, const *arb_t a*, const *arb_t b*, const *arb_t c*, const *arb_t z*, int *regularized*, *slong prec*)

Computes the Gauss hypergeometric function ${}_2F_1(a,b,c,z)$, or $\mathbf{F}(a,b,c,z) = \frac{1}{\Gamma(c)}{}_2F_1(a,b,c,z)$ if *regularized* is set.

Additional evaluation flags can be passed via the *regularized* argument; see *acb_hypgeom_2f1()* for documentation.

7.2.4 Error functions and Fresnel integrals

void **arb_hypgeom_erf**(*arb_t res*, const *arb_t z*, *slong prec*)
Computes the error function $\operatorname{erf}(z)$.

void **_arb_hypgeom_erf_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_erf_series**(*arb_poly_t res*, const *arb_poly_t z*, *slong len*, *slong prec*)
Computes the error function of the power series z, truncated to length *len*.

void **arb_hypgeom_erfc**(*arb_t res*, const *arb_t z*, *slong prec*)
Computes the complementary error function $\operatorname{erfc}(z) = 1 - \operatorname{erf}(z)$. This function avoids catastrophic cancellation for large positive z.

void **_arb_hypgeom_erfc_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_erfc_series**(*arb_poly_t res*, const *arb_poly_t z*, *slong len*, *slong prec*)
Computes the complementary error function of the power series z, truncated to length *len*.

void **arb_hypgeom_erfi**(*arb_t res*, const *arb_t z*, *slong prec*)
Computes the imaginary error function $\operatorname{erfi}(z) = -i \operatorname{erf}(iz)$.

void **_arb_hypgeom_erfi_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_erfi_series**(*arb_poly_t res*, const *arb_poly_t z*, *slong len*, *slong prec*)
Computes the imaginary error function of the power series z, truncated to length *len*.

void **arb_hypgeom_fresnel**(*arb_t res1*, *arb_t res2*, const *arb_t z*, int *normalized*, *slong prec*)
Sets *res1* to the Fresnel sine integral $S(z)$ and *res2* to the Fresnel cosine integral $C(z)$. Optionally, just a single function can be computed by passing *NULL* as the other output variable. The definition $S(z) = \int_0^z \sin(t^2) dt$ is used if *normalized* is 0, and $S(z) = \int_0^z \sin(\frac{1}{2}\pi t^2) dt$ is used if *normalized* is 1 (the latter is the Abramowitz & Stegun convention). $C(z)$ is defined analogously.

void **_arb_hypgeom_fresnel_series**(*arb_ptr res1*, *arb_ptr res2*, *arb_srcptr z*, *slong zlen*, int *normalized*, *slong len*, *slong prec*)

void **arb_hypgeom_fresnel_series**(*arb_poly_t res1*, *arb_poly_t res2*, const *arb_poly_t z*, int *normalized*, *slong len*, *slong prec*)
Sets *res1* to the Fresnel sine integral and *res2* to the Fresnel cosine integral of the power series z, truncated to length *len*. Optionally, just a single function can be computed by passing *NULL* as the other output variable.

7.2.5 Exponential and trigonometric integrals

void **arb_hypgeom_ei**(*arb_t res*, const *arb_t z*, *slong prec*)
Computes the exponential integral $\operatorname{Ei}(z)$.

void **_arb_hypgeom_ei_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_ei_series**(*arb_poly_t res*, const *arb_poly_t z*, *slong len*, *slong prec*)
Computes the exponential integral of the power series z, truncated to length *len*.

void **arb_hypgeom_si**(*arb_t res*, const *arb_t z*, *slong prec*)
Computes the sine integral $\operatorname{Si}(z)$.

void **_arb_hypgeom_si_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_si_series**(*arb_poly_t res*, const *arb_poly_t z*, *slong len*, *slong prec*)
Computes the sine integral of the power series z, truncated to length *len*.

void **arb_hypgeom_ci**(*arb_t res*, const *arb_t z*, *slong prec*)
Computes the cosine integral $\operatorname{Ci}(z)$. The result is indeterminate if $z < 0$ since the value of the function would be complex.

void **_arb_hypgeom_ci_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_ci_series**(*arb_poly_t* *res*, const *arb_poly_t* *z*, *slong* *len*, *slong* *prec*)
Computes the cosine integral of the power series z, truncated to length *len*.

void **arb_hypgeom_shi**(*arb_t* *res*, const *arb_t* *z*, *slong* *prec*)
Computes the hyperbolic sine integral $\operatorname{Shi}(z) = -i \operatorname{Si}(iz)$.

void **_arb_hypgeom_shi_series**(*arb_ptr* *res*, *arb_srcptr* *z*, *slong* *zlen*, *slong* *len*, *slong* *prec*)

void **arb_hypgeom_shi_series**(*arb_poly_t* *res*, const *arb_poly_t* *z*, *slong* *len*, *slong* *prec*)
Computes the hyperbolic sine integral of the power series z, truncated to length *len*.

void **arb_hypgeom_chi**(*arb_t* *res*, const *arb_t* *z*, *slong* *prec*)
Computes the hyperbolic cosine integral $\operatorname{Chi}(z)$. The result is indeterminate if $z < 0$ since the value of the function would be complex.

void **_arb_hypgeom_chi_series**(*arb_ptr* *res*, *arb_srcptr* *z*, *slong* *zlen*, *slong* *len*, *slong* *prec*)

void **arb_hypgeom_chi_series**(*arb_poly_t* *res*, const *arb_poly_t* *z*, *slong* *len*, *slong* *prec*)
Computes the hyperbolic cosine integral of the power series z, truncated to length *len*.

void **arb_hypgeom_li**(*arb_t* *res*, const *arb_t* *z*, int *offset*, *slong* *prec*)
If *offset* is zero, computes the logarithmic integral $\operatorname{li}(z) = \operatorname{Ei}(\log(z))$.

If *offset* is nonzero, computes the offset logarithmic integral $\operatorname{Li}(z) = \operatorname{li}(z) - \operatorname{li}(2)$.

The result is indeterminate if $z < 0$ since the value of the function would be complex.

void **_arb_hypgeom_li_series**(*arb_ptr* *res*, *arb_srcptr* *z*, *slong* *zlen*, int *offset*, *slong* *len*, *slong* *prec*)

void **arb_hypgeom_li_series**(*arb_poly_t* *res*, const *arb_poly_t* *z*, int *offset*, *slong* *len*, *slong* *prec*)
Computes the logarithmic integral (optionally the offset version) of the power series z, truncated to length *len*.

7.3 acb_modular.h – modular forms of complex variables

This module provides methods for numerical evaluation of modular forms, Jacobi theta functions, and elliptic functions.

In the context of this module, *tau* or τ always denotes an element of the complex upper half-plane $\mathbb{H} = \{z \in \mathbb{C} : \operatorname{Im}(z) > 0\}$. We also often use the variable q, variously defined as $q = e^{2\pi i \tau}$ (usually in relation to modular forms) or $q = e^{\pi i \tau}$ (usually in relation to theta functions) and satisfying $|q| < 1$. We will clarify the local meaning of q every time such a quantity appears as a function of τ.

As usual, the numerical functions in this module compute strict error bounds: if *tau* is represented by an *acb_t* whose content overlaps with the real line (or lies in the lower half-plane), and *tau* is passed to a function defined only on $\mathbb{H}$, then the output will have an infinite radius. The analogous behavior holds for functions requiring $|q| < 1$.

7.3.1 The modular group

psl2z_struct

psl2z_t
Represents an element of the modular group $\mathrm{PSL}(2, \mathbb{Z})$, namely an integer matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

with $ad - bc = 1$, and with signs canonicalized such that $c \geq 0$, and $d > 0$ if $c = 0$. The struct members a, b, c, d are of type *fmpz*.

void **psl2z_init**(*psl2z_t g*)
Initializes g and set it to the identity element.

void **psl2z_clear**(*psl2z_t g*)
Clears g.

void **psl2z_swap**(*psl2z_t f*, *psl2z_t g*)
Swaps f and g efficiently.

void **psl2z_set**(*psl2z_t f*, const *psl2z_t g*)
Sets f to a copy of g.

void **psl2z_one**(*psl2z_t g*)
Sets g to the identity element.

int **psl2z_is_one**(const *psl2z_t g*)
Returns nonzero iff g is the identity element.

void **psl2z_print**(const *psl2z_t g*)
Prints g to standard output.

void **psl2z_fprint**(FILE * *file*, const *psl2z_t g*)
Prints g to the stream *file*.

int **psl2z_equal**(const *psl2z_t f*, const *psl2z_t g*)
Returns nonzero iff f and g are equal.

void **psl2z_mul**(*psl2z_t h*, const *psl2z_t f*, const *psl2z_t g*)
Sets h to the product of f and g, namely the matrix product with the signs canonicalized.

void **psl2z_inv**(*psl2z_t h*, const *psl2z_t g*)
Sets h to the inverse of g.

int **psl2z_is_correct**(const *psl2z_t g*)
Returns nonzero iff g contains correct data, i.e. satisfying $ad - bc = 1$, $c \geq 0$, and $d > 0$ if $c = 0$.

void **psl2z_randtest**(*psl2z_t g*, flint_rand_t *state*, *slong bits*)
Sets g to a random element of $\mathrm{PSL}(2, \mathbb{Z})$ with entries of bit length at most *bits* (or 1, if *bits* is not positive). We first generate a and d, compute their Bezout coefficients, divide by the GCD, and then correct the signs.

7.3.2 Modular transformations

void **acb_modular_transform**(*acb_t w*, const *psl2z_t g*, const *acb_t z*, *slong prec*)
Applies the modular transformation g to the complex number z, evaluating

$$w = gz = \frac{az + b}{cz + d}.$$

void **acb_modular_fundamental_domain_approx_d**(*psl2z_t g*, double *x*, double *y*, double *one_minus_eps*)

void **acb_modular_fundamental_domain_approx_arf**(*psl2z_t g*, const *arf_t x*, const *arf_t y*, const *arf_t one_minus_eps*, *slong prec*)
Attempts to determine a modular transformation g that maps the complex number $x + yi$ to the fundamental domain or just slightly outside the fundamental domain, where the target tolerance (not a strict bound) is specified by *one_minus_eps*.

The inputs are assumed to be finite numbers, with y positive.

Uses floating-point iteration, repeatedly applying either the transformation $z \leftarrow z+b$ or $z \leftarrow -1/z$. The iteration is terminated if $|x| \leq 1/2$ and $x^2+y^2 \geq 1-\varepsilon$ where $1-\varepsilon$ is passed as *one_minus_eps*. It is also terminated if too many steps have been taken without convergence, or if the numbers end up too large or too small for the working precision.

The algorithm can fail to produce a satisfactory transformation. The output g is always set to *some* correct modular transformation, but it is up to the user to verify a posteriori that g maps $x + yi$ close enough to the fundamental domain.

void **acb_modular_fundamental_domain_approx**(*acb_t w*, *psl2z_t g*, const *acb_t z*, const *arf_t one_minus_eps*, *slong prec*)

Attempts to determine a modular transformation g that maps the complex number z to the fundamental domain or just slightly outside the fundamental domain, where the target tolerance (not a strict bound) is specified by *one_minus_eps*. It also computes the transformed value $w = gz$.

This function first tries to use *acb_modular_fundamental_domain_approx_d()* and checks if the result is acceptable. If this fails, it calls *acb_modular_fundamental_domain_approx_arf()* with higher precision. Finally, $w = gz$ is evaluated by a single application of g.

The algorithm can fail to produce a satisfactory transformation. The output g is always set to *some* correct modular transformation, but it is up to the user to verify a posteriori that w is close enough to the fundamental domain.

int **acb_modular_is_in_fundamental_domain**(const *acb_t z*, const *arf_t tol*, *slong prec*)

Returns nonzero if it is certainly true that $|z| \geq 1 - \varepsilon$ and $|\operatorname{Re}(z)| \leq 1/2 + \varepsilon$ where ε is specified by *tol*. Returns zero if this is false or cannot be determined.

7.3.3 Addition sequences

void **acb_modular_fill_addseq**(*slong* * *tab*, *slong len*)

Builds a near-optimal addition sequence for a sequence of integers which is assumed to be reasonably dense.

As input, the caller should set each entry in *tab* to -1 if that index is to be part of the addition sequence, and to 0 otherwise. On output, entry i in *tab* will either be zero (if the number is not part of the sequence), or a value j such that both j and $i - j$ are also marked. The first two entries in *tab* are ignored (the number 1 is always assumed to be part of the sequence).

7.3.4 Jacobi theta functions

Unfortunately, there are many inconsistent notational variations for Jacobi theta functions in the literature. Unless otherwise noted, we use the functions

$$\theta_1(z,\tau) = -i \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i[(n+1/2)^2\tau + (2n+1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} (-1)^n q^{n(n+1)} \sin((2n+1)\pi z)$$

$$\theta_2(z,\tau) = \sum_{n=-\infty}^{\infty} \exp(\pi i[(n+1/2)^2\tau + (2n+1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos((2n+1)\pi z)$$

$$\theta_3(z,\tau) = \sum_{n=-\infty}^{\infty} \exp(\pi i[n^2\tau + 2nz]) = 1 + 2\sum_{n=1}^{\infty} q^{n^2} \cos(2n\pi z)$$

$$\theta_4(z,\tau) = \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i[n^2\tau + 2nz]) = 1 + 2\sum_{n=1}^{\infty} (-1)^n q^{n^2} \cos(2n\pi z)$$

where $q = \exp(\pi i \tau)$ and $q_{1/4} = \exp(\pi i \tau/4)$. Note that many authors write $q_{1/4}$ as $q^{1/4}$, but the principal fourth root $(q)^{1/4} = \exp(\frac{1}{4}\log q)$ differs from $q_{1/4}$ in general and some formulas are only correct if one reads "$q^{1/4} = \exp(\pi i \tau/4)$". To avoid confusion, we only write q^k when k is an integer.

void **acb_modular_theta_transform**(int * *R*, int * *S*, int * *C*, const *psl2z_t* *g*)

We wish to write a theta function with quasiperiod τ in terms of a theta function with quasiperiod $\tau' = g\tau$, given some $g = (a, b; c, d) \in \mathrm{PSL}(2, \mathbb{Z})$. For $i = 0, 1, 2, 3$, this function computes integers R_i and S_i (R and S should be arrays of length 4) and $C \in \{0, 1\}$ such that

$$\theta_{1+i}(z, \tau) = \exp(\pi i R_i/4) \cdot A \cdot B \cdot \theta_{1+S_i}(z', \tau')$$

where $z' = z, A = B = 1$ if $C = 0$, and

$$z' = \frac{-z}{c\tau + d}, \quad A = \sqrt{\frac{i}{c\tau + d}}, \quad B = \exp\left(-\pi i c \frac{z^2}{c\tau + d}\right)$$

if $C = 1$. Note that A is well-defined with the principal branch of the square root since $A^2 = i/(c\tau + d)$ lies in the right half-plane.

Firstly, if $c = 0$, we have $\theta_i(z, \tau) = \exp(-\pi i b/4)\theta_i(z, \tau + b)$ for $i = 1, 2$, whereas θ_3 and θ_4 remain unchanged when b is even and swap places with each other when b is odd. In this case we set $C = 0$.

For an arbitrary g with $c > 0$, we set $C = 1$. The general transformations are given by Rademacher *[Rad1973]*. We need the function $\theta_{m,n}(z, \tau)$ defined for $m, n \in \mathbb{Z}$ by (beware of the typos in *[Rad1973]*)

$$\theta_{0,0}(z, \tau) = \theta_3(z, \tau), \quad \theta_{0,1}(z, \tau) = \theta_4(z, \tau)$$

$$\theta_{1,0}(z, \tau) = \theta_2(z, \tau), \quad \theta_{1,1}(z, \tau) = i\theta_1(z, \tau)$$

$$\theta_{m+2,n}(z, \tau) = (-1)^n \theta_{m,n}(z, \tau)$$

$$\theta_{m,n+2}(z, \tau) = \theta_{m,n}(z, \tau).$$

Then we may write

$$\begin{aligned}
\theta_1(z, \tau) &= \varepsilon_1 AB\theta_1(z', \tau') \\
\theta_2(z, \tau) &= \varepsilon_2 AB\theta_{1-c,1+a}(z', \tau') \\
\theta_3(z, \tau) &= \varepsilon_3 AB\theta_{1+d-c,1-b+a}(z', \tau') \\
\theta_4(z, \tau) &= \varepsilon_4 AB\theta_{1+d,1-b}(z', \tau')
\end{aligned}$$

where ε_i is an 8th root of unity. Specifically, if we denote the 24th root of unity in the transformation formula of the Dedekind eta function by $\varepsilon(a, b, c, d) = \exp(\pi i R(a, b, c, d)/12)$ (see *acb_modular_epsilon_arg()*), then:

$$\begin{aligned}
\varepsilon_1(a, b, c, d) &= \exp(\pi i[R(-d, b, c, -a) + 1]/4) \\
\varepsilon_2(a, b, c, d) &= \exp(\pi i[-R(a, b, c, d) + (5 + (2 - c)a)]/4) \\
\varepsilon_3(a, b, c, d) &= \exp(\pi i[-R(a, b, c, d) + (4 + (c - d - 2)(b - a))]/4) \\
\varepsilon_4(a, b, c, d) &= \exp(\pi i[-R(a, b, c, d) + (3 - (2 + d)b)]/4)
\end{aligned}$$

These formulas are easily derived from the formulas in *[Rad1973]* (Rademacher has the transformed/untransformed variables exchanged, and his "ε" differs from ours by a constant offset in the phase).

void **acb_modular_addseq_theta**(*slong* * *exponents*, *slong* * *aindex*, *slong* * *bindex*, *slong* *num*)

Constructs an addition sequence for the first *num* squares and triangular numbers interleaved (excluding zero), i.e. 1, 2, 4, 6, 9, 12, 16, 20, 25, 30 etc.

void **acb_modular_theta_sum**(*acb_ptr theta1*, *acb_ptr theta2*, *acb_ptr theta3*, *acb_ptr theta4*, const *acb_t w*, int *w_is_unit*, const *acb_t q*, *slong len*, *slong prec*)

Simultaneously computes the first *len* coefficients of each of the formal power series

$$\theta_1(z+x,\tau)/q_{1/4} \in \mathbb{C}[[x]]$$
$$\theta_2(z+x,\tau)/q_{1/4} \in \mathbb{C}[[x]]$$
$$\theta_3(z+x,\tau) \in \mathbb{C}[[x]]$$
$$\theta_4(z+x,\tau) \in \mathbb{C}[[x]]$$

given $w = \exp(\pi i z)$ and $q = \exp(\pi i \tau)$, by summing a finite truncation of the respective theta function series. In particular, with *len* equal to 1, computes the respective value of the theta function at the point z. We require *len* to be positive. If *w_is_unit* is nonzero, w is assumed to lie on the unit circle, i.e. z is assumed to be real.

Note that the factor $q_{1/4}$ is removed from θ_1 and θ_2. To get the true theta function values, the user has to multiply this factor back. This convention avoids unnecessary computations, since the user can compute $q_{1/4} = \exp(\pi i \tau/4)$ followed by $q = (q_{1/4})^4$, and in many cases when computing products or quotients of theta functions, the factor $q_{1/4}$ can be eliminated entirely.

This function is intended for $|q| \ll 1$. It can be called with any q, but will return useless intervals if convergence is not rapid. For general evaluation of theta functions, the user should only call this function after applying a suitable modular transformation.

We consider the sums together, alternatingly updating (θ_1, θ_2) or (θ_3, θ_4). For $k = 0, 1, 2, \ldots$, the powers of q are $\lfloor (k+2)^2/4 \rfloor = 1, 2, 4, 6, 9$ etc. and the powers of w are $\pm(k+2) = \pm 2, \pm 3, \pm 4, \ldots$ etc. The scheme is illustrated by the following table:

	θ_1, θ_2	q^0	$(w^1 \pm w^{-1})$
$k=0$	θ_3, θ_4	q^1	$(w^2 \pm w^{-2})$
$k=1$	θ_1, θ_2	q^2	$(w^3 \pm w^{-3})$
$k=2$	θ_3, θ_4	q^4	$(w^4 \pm w^{-4})$
$k=3$	θ_1, θ_2	q^6	$(w^5 \pm w^{-5})$
$k=4$	θ_3, θ_4	q^9	$(w^6 \pm w^{-6})$
$k=5$	θ_1, θ_2	q^{12}	$(w^7 \pm w^{-7})$

For some integer $N \geq 1$, the summation is stopped just before term $k = N$. Let $Q = |q|$, $W = \max(|w|, |w^{-1}|)$, $E = \lfloor (N+2)^2/4 \rfloor$ and $F = \lfloor (N+1)/2 \rfloor + 1$. The error of the zeroth derivative can be bounded as

$$2Q^E W^{N+2} \left[1 + Q^F W + Q^{2F} W^2 + \ldots\right] = \frac{2Q^E W^{N+2}}{1 - Q^F W}$$

provided that the denominator is positive (otherwise we set the error bound to infinity). When *len* is greater than 1, consider the derivative of order r. The term of index k and order r picks up a factor of magnitude $(k+2)^r$ from differentiation of w^{k+2} (it also picks up a factor π^r, but we omit this until we rescale the coefficients at the end of the computation). Thus we have the error bound

$$2Q^E W^{N+2} (N+2)^r \left[1 + Q^F W \frac{(N+3)^r}{(N+2)^r} + Q^{2F} W^2 \frac{(N+4)^r}{(N+2)^r} + \ldots\right]$$

which by the inequality $(1 + m/(N+2))^r \leq \exp(mr/(N+2))$ can be bounded as

$$\frac{2Q^E W^{N+2} (N+2)^r}{1 - Q^F W \exp(r/(N+2))},$$

again valid when the denominator is positive.

To actually evaluate the series, we write the even cosine terms as $w^{2n} + w^{-2n}$, the odd cosine terms as $w(w^{2n} + w^{-2n-2})$, and the sine terms as $w(w^{2n} - w^{-2n-2})$. This way we only need even powers of w and w^{-1}. The implementation is not yet optimized for real z, in which case further work can be saved.

This function does not permit aliasing between input and output arguments.

void **acb_modular_theta_const_sum_basecase**(*acb_t theta2*, *acb_t theta3*, *acb_t theta4*, const *acb_t q*, *slong N*, *slong prec*)

void **acb_modular_theta_const_sum_rs**(*acb_t theta2*, *acb_t theta3*, *acb_t theta4*, const *acb_t q*, *slong N*, *slong prec*)

Computes the truncated theta constant sums $\theta_2 = \sum_{k(k+1)<N} q^{k(k+1)}$, $\theta_3 = \sum_{k^2<N} q^{k^2}$, $\theta_4 = \sum_{k^2<N} (-1)^k q^{k^2}$. The *basecase* version uses a minimal addition sequence. The *rs* version uses rectangular splitting.

void **acb_modular_theta_const_sum**(*acb_t theta2*, *acb_t theta3*, *acb_t theta4*, const *acb_t q*, *slong prec*)

Computes the respective theta constants by direct summation (without applying modular transformations). This function selects an appropriate N, calls either *acb_modular_theta_const_sum_basecase()* or *acb_modular_theta_const_sum_rs()* or depending on N, and adds a bound for the truncation error.

void **acb_modular_theta_notransform**(*acb_t theta1*, *acb_t theta2*, *acb_t theta3*, *acb_t theta4*, const *acb_t z*, const *acb_t tau*, *slong prec*)

Evaluates the Jacobi theta functions $\theta_i(z, \tau)$, $i = 1, 2, 3, 4$ simultaneously. This function does not move τ to the fundamental domain. This is generally worse than *acb_modular_theta()*, but can be slightly better for moderate input.

void **acb_modular_theta**(*acb_t theta1*, *acb_t theta2*, *acb_t theta3*, *acb_t theta4*, const *acb_t z*, const *acb_t tau*, *slong prec*)

Evaluates the Jacobi theta functions $\theta_i(z, \tau)$, $i = 1, 2, 3, 4$ simultaneously. This function moves τ to the fundamental domain before calling *acb_modular_theta_sum()*.

7.3.5 The Dedekind eta function

void **acb_modular_addseq_eta**(*slong* * *exponents*, *slong* * *aindex*, *slong* * *bindex*, *slong num*)

Constructs an addition sequence for the first *num* generalized pentagonal numbers (excluding zero), i.e. 1, 2, 5, 7, 12, 15, 22, 26, 35, 40 etc.

void **acb_modular_eta_sum**(*acb_t eta*, const *acb_t q*, *slong prec*)

Evaluates the Dedekind eta function without the leading 24th root, i.e.

$$\exp(-\pi i \tau/12)\eta(\tau) = \sum_{n=-\infty}^{\infty} (-1)^n q^{(3n^2-n)/2}$$

given $q = \exp(2\pi i \tau)$, by summing the defining series.

This function is intended for $|q| \ll 1$. It can be called with any q, but will return useless intervals if convergence is not rapid. For general evaluation of the eta function, the user should only call this function after applying a suitable modular transformation.

int **acb_modular_epsilon_arg**(const *psl2z_t g*)

Given $g = (a, b; c, d)$, computes an integer R such that $\varepsilon(a, b, c, d) = \exp(\pi i R/12)$ is the 24th root of unity in the transformation formula for the Dedekind eta function,

$$\eta\left(\frac{a\tau + b}{c\tau + d}\right) = \varepsilon(a, b, c, d)\sqrt{c\tau + d}\eta(\tau).$$

void **acb_modular_eta**(*acb_t r*, const *acb_t tau*, *slong prec*)

Computes the Dedekind eta function $\eta(\tau)$ given τ in the upper half-plane. This function applies the functional equation to move τ to the fundamental domain before calling *acb_modular_eta_sum()*.

7.3.6 Modular forms

void **acb_modular_j**(*acb_t r*, const *acb_t tau*, *slong prec*)

Computes Klein's j-invariant $j(\tau)$ given τ in the upper half-plane. The function is normalized so

that $j(i) = 1728$. We first move τ to the fundamental domain, which does not change the value of the function. Then we use the formula $j(\tau) = 32(\theta_2^8 + \theta_3^8 + \theta_4^8)^3/(\theta_2\theta_3\theta_4)^8$ where $\theta_i = \theta_i(0, \tau)$.

void **acb_modular_lambda**(*acb_t r*, const *acb_t tau*, *slong prec*)
Computes the lambda function $\lambda(\tau) = \theta_2^4(0,\tau)/\theta_3^4(0,\tau)$, which is invariant under modular transformations $(a, b; c, d)$ where a, d are odd and b, c are even.

void **acb_modular_delta**(*acb_t r*, const *acb_t tau*, *slong prec*)
Computes the modular discriminant $\Delta(\tau) = \eta(\tau)^{24}$, which transforms as

$$\Delta\left(\frac{a\tau+b}{c\tau+d}\right) = (c\tau+d)^{12}\Delta(\tau).$$

The modular discriminant is sometimes defined with an extra factor $(2\pi)^{12}$, which we omit in this implementation.

void **acb_modular_eisenstein**(*acb_ptr r*, const *acb_t tau*, *slong len*, *slong prec*)
Computes simultaneously the first *len* entries in the sequence of Eisenstein series $G_4(\tau), G_6(\tau), G_8(\tau), \ldots$, defined by

$$G_{2k}(\tau) = \sum_{m^2+n^2\neq 0} \frac{1}{(m+n\tau)^{2k}}$$

and satisfying

$$G_{2k}\left(\frac{a\tau+b}{c\tau+d}\right) = (c\tau+d)^{2k}G_{2k}(\tau).$$

We first evaluate $G_4(\tau)$ and $G_6(\tau)$ on the fundamental domain using theta functions, and then compute the Eisenstein series of higher index using a recurrence relation.

7.3.7 Elliptic functions

void **acb_modular_elliptic_p**(*acb_t wp*, const *acb_t z*, const *acb_t tau*, *slong prec*)
Computes Weierstrass's elliptic function

$$\wp(z,\tau) = \frac{1}{z^2} + \sum_{n^2+m^2\neq 0}\left[\frac{1}{(z+m+n\tau)^2} - \frac{1}{(m+n\tau)^2}\right]$$

which satisfies $\wp(z,\tau) = \wp(z+1,\tau) = \wp(z+\tau,\tau)$. To evaluate the function efficiently, we use the formula

$$\wp(z,\tau) = \pi^2\theta_2^2(0,\tau)\theta_3^2(0,\tau)\frac{\theta_4^2(z,\tau)}{\theta_1^2(z,\tau)} - \frac{\pi^2}{3}\left[\theta_3^4(0,\tau) + \theta_3^4(0,\tau)\right].$$

void **acb_modular_elliptic_p_zpx**(*acb_ptr wp*, const *acb_t z*, const *acb_t tau*, *slong len*, *slong prec*)
Computes the formal power series $\wp(z + x, \tau) \in \mathbb{C}[[x]]$, truncated to length *len*. In particular, with *len* $= 2$, simultaneously computes $\wp(z,\tau), \wp'(z,\tau)$ which together generate the field of elliptic functions with periods 1 and τ.

7.3.8 Elliptic integrals

void **acb_modular_elliptic_k**(*acb_t w*, const *acb_t m*, *slong prec*)
Computes the complete elliptic integral of the first kind $K(m)$, using the arithmetic-geometric mean: $K(m) = \pi/(2M(\sqrt{1-m}))$.

void **acb_modular_elliptic_k_cpx**(*acb_ptr w*, const *acb_t m*, *slong len*, *slong prec*)
Sets the coefficients in the array w to the power series expansion of the complete elliptic integral of the first kind at the point m truncated to length *len*, i.e. $K(m + x) \in \mathbb{C}[[x]]$.

void **acb_modular_elliptic_e**(*acb_t w*, const *acb_t m*, *slong prec*)
Computes the complete elliptic integral of the second kind $E(m)$, which is given by $E(m) = (1-m)(2mK'(m) + K(m))$ (where the prime denotes a derivative, not a complementary integral).

7.3.9 Class polynomials

void **acb_modular_hilbert_class_poly**(*fmpz_poly_t* *res*, *slong* *D*)
Sets *res* to the Hilbert class polynomial of discriminant D, defined as

$$H_D(x) = \prod_{(a,b,c)} \left(x - j\left(\frac{-b+\sqrt{D}}{2a}\right)\right)$$

where (a, b, c) ranges over the primitive reduced positive definite binary quadratic forms of discriminant $b^2 - 4ac = D$.

The Hilbert class polynomial is only defined if $D < 0$ and D is congruent to 0 or 1 mod 4. If some other value of D is passed as input, *res* is set to the zero polynomial.

7.4 acb_dirichlet.h – Dirichlet L-functions, zeta functions, and related functions

Warning: the interfaces in this module are experimental and may change without notice.

This module will eventually allow working with Dirichlet L-functions and possibly slightly more general Dirichlet series. At the moment, it contains nothing interesting.

The code in other modules for computing the Riemann zeta function, Hurwitz zeta function and polylogarithm will possibly be migrated to this module in the future.

A Dirichlet L-function is the analytic continuation of an L-series

$$L(s, \chi) = \sum_{k=1}^{\infty} \frac{\chi(k)}{k^s}$$

where $\chi(k)$ is a Dirichlet character.

7.4.1 Dirichlet characters

acb_dirichlet_group_struct

acb_dirichlet_group_t
Represents the group of Dirichlet characters mod q.

An *acb_dirichlet_group_t* is defined as an array of *acb_dirichlet_struct* of length 1, permitting it to be passed by reference.

void **acb_dirichlet_group_init**(*acb_dirichlet_group_t* *G*, *ulong* *q*)
Initializes G to the group of Dirichlet characters mod q.

This method computes the prime factorization of q and other useful invariants. It does *not* automatically precompute lookup tables of discrete logarithms or numerical roots of unity, and can therefore safely be called even with large q. For implementation reasons, the largest prime factor of q must not exceed 10^{12} (an abort will be raised). This restriction could be removed in the future.

void **acb_dirichlet_group_clear**(*acb_dirichlet_group_t* *G*)
Clears G.

void **acb_dirichlet_chi**(*acb_t* *res*, const *acb_dirichlet_group_t* *G*, *ulong* *m*, *ulong* *n*, *slong* *prec*)
Sets *res* to $\chi_m(n)$, the value of the Dirichlet character of index m evaluated at the integer n.

Requires that m is a valid index, that is, $1 \le m \le q$ and m is coprime to q. There are no restrictions on n.

7.4.2 Euler products

void **_acb_dirichlet_euler_product_real_ui**(*arb_t* *res*, *ulong* *s*, const signed char * *chi*, int *mod*, int *reciprocal*, *slong* *prec*)

Sets *res* to $L(s,\chi)$ where χ is a real Dirichlet character given by the explicit list *chi* of character values at 0, 1, ..., *mod* - 1. If *reciprocal* is set, computes $1/L(s,\chi)$ (this is faster if the reciprocal can be used directly).

This function uses the Euler product, and is only intended for use when s is large. An error bound is computed via *mag_hurwitz_zeta_uiui()*. Since

$$\frac{1}{L(s,\chi)} = \prod_p \left(1 - \frac{\chi(p)}{p^s}\right) = \sum_{k=1}^{\infty} \frac{\mu(k)\chi(k)}{k^s}$$

and the truncated product gives all smooth-index terms in the series, we have

$$\left| \prod_{p<N} \left(1 - \frac{\chi(p)}{p^s}\right) - \frac{1}{L(s,\chi)} \right| \le \sum_{k=N}^{\infty} \frac{1}{k^s} = \zeta(s,N).$$

7.4.3 Simple functions

void **acb_dirichlet_eta**(*acb_t* *res*, const *acb_t* *s*, *slong* *prec*)

Sets *res* to the Dirichlet eta function $\eta(s) = \sum_{k=1}^{\infty}(-1)^k/k^s = (1-2^{1-s})\zeta(s)$, also known as the alternating zeta function. Note that the alternating character $\{1,-1\}$ is not itself a Dirichlet character.

7.5 bernoulli.h – support for Bernoulli numbers

This module provides helper functions for exact or approximate calculation of the Bernoulli numbers, which are defined by the exponential generating function

$$\frac{x}{e^x - 1} = \sum_{n=0}^{\infty} B_n \frac{x^n}{n!}.$$

Efficient algorithms are implemented for both multi-evaluation and calculation of isolated Bernoulli numbers. A global (or thread-local) cache is also provided, to support fast repeated evaluation of various special functions that depend on the Bernoulli numbers (including the gamma function and the Riemann zeta function).

7.5.1 Generation of Bernoulli numbers

bernoulli_rev_t

An iterator object for generating a range of even-indexed Bernoulli numbers exactly in reverse order, i.e. computing the exact fractions $B_n, B_{n-2}, B_{n-4}, \ldots, B_0$. The Bernoulli numbers are generated from scratch, i.e. no caching is performed.

The Bernoulli numbers are computed by direct summation of the zeta series. This is made fast by storing a table of powers (as done by *[Blo2009]*). As an optimization, we only include the odd powers, and use fixed-point arithmetic.

The reverse iteration order is preferred for performance reasons, as the powers can be updated using multiplications instead of divisions, and we avoid having to periodically recompute terms to higher precision. To generate Bernoulli numbers in the forward direction without having to store all of them, one can split the desired range into smaller blocks and compute each block with a single reverse pass.

void **bernoulli_rev_init**(*bernoulli_rev_t iter*, *ulong n*)
Initializes the iterator *iter*. The first Bernoulli number to be generated by calling *bernoulli_rev_next()* is B_n. It is assumed that n is even.

void **bernoulli_rev_next**(*fmpz_t numer*, *fmpz_t denom*, *bernoulli_rev_t iter*)
Sets *numer* and *denom* to the exact, reduced numerator and denominator of the Bernoulli number B_k and advances the state of *iter* so that the next invocation generates B_{k-2}.

void **bernoulli_rev_clear**(*bernoulli_rev_t iter*)
Frees all memory allocated internally by *iter*.

7.5.2 Caching

slong **bernoulli_cache_num**

fmpq * **bernoulli_cache**
Cache of Bernoulli numbers. Uses thread-local storage if enabled in FLINT.

void **bernoulli_cache_compute**(*slong n*)
Makes sure that the Bernoulli numbers up to at least B_{n-1} are cached. Calling `flint_cleanup()` frees the cache.

7.5.3 Bounding

slong **bernoulli_bound_2exp_si**(*ulong n*)
Returns an integer b such that $|B_n| \le 2^b$. Uses a lookup table for small n, and for larger n uses the inequality $|B_n| < 4n!/(2\pi)^n < 4(n+1)^{n+1}e^{-n}/(2\pi)^n$. Uses integer arithmetic throughout, with the bound for the logarithm being looked up from a table. If $|B_n| = 0$, returns *LONG_MIN*. Otherwise, the returned exponent b is never more than one percent larger than the true magnitude.

This function is intended for use when n small enough that one might comfortably compute B_n exactly. It aborts if n is so large that internal overflow occurs.

void **_bernoulli_fmpq_ui_zeta**(*fmpz_t num*, *fmpz_t den*, *ulong n*)
Sets *num* and *den* to the reduced numerator and denominator of the Bernoulli number B_n.

This function computes the denominator d using von Staudt-Clausen theorem, numerically approximates B_n using *arb_bernoulli_ui_zeta()*, and then rounds dB_n to the correct numerator. If the working precision is insufficient to determine the numerator, the function prints a warning message and retries with increased precision (this should not be expected to happen).

void **_bernoulli_fmpq_ui**(*fmpz_t num*, *fmpz_t den*, *ulong n*)

void **bernoulli_fmpq_ui**(*fmpq_t b*, *ulong n*)
Computes the Bernoulli number B_n as an exact fraction, for an isolated integer n. This function reads B_n from the global cache if the number is already cached, but does not automatically extend the cache by itself.

7.6 hypgeom.h – support for hypergeometric series

This module provides functions for high-precision evaluation of series of the form

$$\sum_{k=0}^{n-1} \frac{A(k)}{B(k)} \prod_{j=1}^{k} \frac{P(k)}{Q(k)} z^k$$

where A, B, P, Q are polynomials. The present version only supports $A, B, P, Q \in \mathbb{Z}[k]$ (represented using the FLINT *fmpz_poly_t* type). This module also provides functions for high-precision evaluation of infinite series ($n \to \infty$), with automatic, rigorous error bounding.

Note that we can standardize to $A = B = 1$ by setting $\tilde{P}(k) = P(k)A(k)B(k-1), \tilde{Q}(k) = Q(k)A(k-1)B(k)$. However, separating out A and B is convenient and improves efficiency during evaluation.

7.6.1 Strategy for error bounding

We wish to evaluate $S(z) = \sum_{k=0}^{\infty} T(k)z^k$ where $T(k)$ satisfies $T(0) = 1$ and

$$T(k) = R(k)T(k-1) = \left(\frac{P(k)}{Q(k)}\right) T(k-1)$$

for given polynomials

$$P(k) = a_p k^p + a_{p-1}k^{p-1} + \ldots a_0$$
$$Q(k) = b_q k^q + b_{q-1}k^{q-1} + \ldots b_0.$$

For convergence, we require $p < q$, or $p = q$ with $|z||a_p| < |b_q|$. We also assume that $P(k)$ and $Q(k)$ have no roots among the positive integers (if there are positive integer roots, the sum is either finite or undefined). With these conditions satisfied, our goal is to find a parameter $n \geq 0$ such that

$$\left|\sum_{k=n}^{\infty} T(k)z^k\right| \leq 2^{-d}.$$

We can rewrite the hypergeometric term ratio as

$$zR(k) = z\frac{P(k)}{Q(k)} = z\left(\frac{a_p}{b_q}\right)\frac{1}{k^{q-p}}F(k)$$

where

$$F(k) = \frac{1 + \tilde{a}_1/k + \tilde{a}_2/k^2 + \ldots + \tilde{a}_q/k^p}{1 + \tilde{b}_1/k + \tilde{b}_2/k^2 + \ldots + \tilde{b}_q/k^q} = 1 + O(1/k)$$

and where $\tilde{a}_i = a_{p-i}/a_p$, $\tilde{b}_i = b_{q-i}/b_q$. Next, we define

$$C = \max_{1\leq i\leq p} |\tilde{a}_i|^{(1/i)}, \quad D = \max_{1\leq i\leq q} |\tilde{b}_i|^{(1/i)}.$$

Now, if $k > C$, the magnitude of the numerator of $F(k)$ is bounded from above by

$$1 + \sum_{i=1}^{p}\left(\frac{C}{k}\right)^i \leq 1 + \frac{C}{k-C}$$

and if $k > 2D$, the magnitude of the denominator of $F(k)$ is bounded from below by

$$1 - \sum_{i=1}^{q}\left(\frac{D}{k}\right)^i \geq 1 + \frac{D}{D-k}.$$

Putting the inequalities together gives the following bound, valid for $k > K = \max(C, 2D)$:

$$|F(k)| \leq \frac{k(k-D)}{(k-C)(k-2D)} = \left(1 + \frac{C}{k-C}\right)\left(1 + \frac{D}{k-2D}\right).$$

Let $r = q - p$ and $\tilde{z} = |za_p/b_q|$. Assuming $k > \max(C, 2D, \tilde{z}^{1/r})$, we have

$$|zR(k)| \leq G(k) = \frac{\tilde{z}F(k)}{k^r}$$

where $G(k)$ is monotonically decreasing. Now we just need to find an n such that $G(n) < 1$ and for which $|T(n)|/(1 - G(n)) \leq 2^{-d}$. This can be done by computing a floating-point guess for n then trying successively larger values.

This strategy leaves room for some improvement. For example, if $\tilde{b}_1$ is positive and large, the bound B becomes very pessimistic (a larger positive $\tilde{b}_1$ causes faster convergence, not slower convergence).

7.6.2 Types, macros and constants

hypgeom_struct

hypgeom_t
Stores polynomials A, B, P, Q and precomputed bounds, representing a fixed hypergeometric series.

7.6.3 Memory management

void **hypgeom_init**(*hypgeom_t hyp*)

void **hypgeom_clear**(*hypgeom_t hyp*)

7.6.4 Error bounding

slong **hypgeom_estimate_terms**(const *mag_t z*, int *r*, *slong d*)
Computes an approximation of the largest n such that $|z|^n/(n!)^r = 2^{-d}$, giving a first-order estimate of the number of terms needed to approximate the sum of a hypergeometric series of weight $r \geq 0$ and argument z to an absolute precision of $d \geq 0$ bits. If $r = 0$, the direct solution of the equation is given by $n = (\log(1-z) - d\log 2)/\log z$. If $r > 0$, using $\log n! \approx n\log n - n$ gives an equation that can be solved in terms of the Lambert W-function as $n = (d\log 2)/(r\,W(t))$ where $t = (d\log 2)/(erz^{1/r})$.

The evaluation is done using double precision arithmetic. The function aborts if the computed value of n is greater than or equal to LONG_MAX / 2.

slong **hypgeom_bound**(*mag_t error*, int *r*, *slong C*, *slong D*, *slong K*, const *mag_t TK*, const *mag_t z*, *slong prec*)
Computes a truncation parameter sufficient to achieve *prec* bits of absolute accuracy, according to the strategy described above. The input consists of r, C, D, K, precomputed bound for $T(K)$, and $\tilde{z} = z(a_p/b_q)$, such that for $k > K$, the hypergeometric term ratio is bounded by

$$\frac{\tilde{z}}{k^r}\frac{k(k-D)}{(k-C)(k-2D)}.$$

Given this information, we compute a ε and an integer n such that $|\sum_{k=n}^{\infty} T(k)| \leq \varepsilon \leq 2^{-\text{prec}}$. The output variable *error* is set to the value of ε, and n is returned.

void **hypgeom_precompute**(*hypgeom_t hyp*)
Precomputes the bounds data C, D, K and an upper bound for $T(K)$.

7.6.5 Summation

void **arb_hypgeom_sum**(*arb_t P*, *arb_t Q*, const *hypgeom_t hyp*, const *slong n*, *slong prec*)
Computes P, Q such that $P/Q = \sum_{k=0}^{n-1} T(k)$ where $T(k)$ is defined by *hyp*, using binary splitting and a working precision of *prec* bits.

void **arb_hypgeom_infsum**(*arb_t P*, *arb_t Q*, *hypgeom_t hyp*, *slong tol*, *slong prec*)
Computes P, Q such that $P/Q = \sum_{k=0}^{\infty} T(k)$ where $T(k)$ is defined by *hyp*, using binary splitting and working precision of *prec* bits. The number of terms is chosen automatically to bound the truncation error by at most $2^{-\text{tol}}$. The bound for the truncation error is included in the output as part of P.

7.7 partitions.h – computation of the partition function

This module implements the asymptotically fast algorithm for evaluating the integer partition function $p(n)$ described in *[Joh2012]*. The idea is to evaluate a truncation of the Hardy-Ramanujan-Rademacher series using tight precision estimates, and symbolically factoring the occurring exponential sums.

An implementation based on floating-point arithmetic can also be found in FLINT. That version relies on some numerical subroutines that have not been proved correct.

The implementation provided here uses ball arithmetic throughout to guarantee a correct error bound for the numerical approximation of $p(n)$. Optionally, hardware double arithmetic can be used for low-precision terms. This gives a significant speedup for small (e.g. $n < 10^6$).

void **partitions_rademacher_bound**(*arf_t* *b*, const *fmpz_t* *n*, *ulong* *N*)
Sets b to an upper bound for

$$M(n,N) = \frac{44\pi^2}{225\sqrt{3}} N^{-1/2} + \frac{\pi\sqrt{2}}{75}\left(\frac{N}{n-1}\right)^{1/2} \sinh\left(\frac{\pi}{N}\sqrt{\frac{2n}{3}}\right).$$

This formula gives an upper bound for the truncation error in the Hardy-Ramanujan-Rademacher formula when the series is taken up to the term $t(n,N)$ inclusive.

partitions_hrr_sum_arb(*arb_t* *x*, const *fmpz_t* *n*, *slong* *N0*, *slong* *N*, int *use_doubles*)
Evaluates the partial sum $\sum_{k=N_0}^{N} t(n,k)$ of the Hardy-Ramanujan-Rademacher series.

If *use_doubles* is nonzero, doubles and the system's standard library math functions are used to evaluate the smallest terms. This significantly speeds up evaluation for small n (e.g. $n < 10^6$), and gives a small speed improvement for larger n, but the result is not guaranteed to be correct. In practice, the error is estimated very conservatively, and unless the system's standard library is broken, use of doubles can be considered safe. Setting *use_doubles* to zero gives a fully guaranteed bound.

void **partitions_fmpz_fmpz**(*fmpz_t* *p*, const *fmpz_t* *n*, int *use_doubles*)
Computes the partition function $p(n)$ using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing $p(n)$ and verifies that the ball contains a unique integer.

If n is sufficiently large and a number of threads greater than 1 has been selected with `flint_set_num_threads()`, the computation time will be reduced by using two threads.

See *partitions_hrr_sum_arb()* for an explanation of the *use_doubles* option.

void **partitions_fmpz_ui**(*fmpz_t* *p*, *ulong* *n*)
Computes the partition function $p(n)$ using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing $p(n)$ and verifies that the ball contains a unique integer.

void **partitions_fmpz_ui_using_doubles**(*fmpz_t* *p*, *ulong* *n*)
Computes the partition function $p(n)$, enabling the use of doubles internally. This significantly speeds up evaluation for small n (e.g. $n < 10^6$), but the error bounds are not certified (see remarks for *partitions_hrr_sum_arb()*).

void **partitions_leading_fmpz**(*arb_t* *res*, const *fmpz_t* *n*, *slong* *prec*)
Sets *res* to the leading term in the Hardy-Ramanujan series for $p(n)$ (without Rademacher's correction of this term, which is vanishingly small when n is large), that is, $\sqrt{12}(1-1/t)e^t/(24n-1)$ where $t = \pi\sqrt{24n-1}/6$.

CHAPTER

EIGHT

CALCULUS

Using ball arithmetic, it is possible to do rigorous root-finding and integration (among other operations) with generic functions. This code should be considered experimental.

8.1 arb_calc.h – calculus with real-valued functions

This module provides functions for operations of calculus over the real numbers (intended to include root-finding, optimization, integration, and so on). It is planned that the module will include two types of algorithms:

- Interval algorithms that give provably correct results. An example would be numerical integration on an interval by dividing the interval into small balls and evaluating the function on each ball, giving rigorous upper and lower bounds.
- Conventional numerical algorithms that use heuristics to estimate the accuracy of a result, without guaranteeing that it is correct. An example would be numerical integration based on pointwise evaluation, where the error is estimated by comparing the results with two different sets of evaluation points. Ball arithmetic then still tracks the accuracy of the function evaluations.

Any algorithms of the second kind will be clearly marked as such.

8.1.1 Types, macros and constants

arb_calc_func_t

Typedef for a pointer to a function with signature:

```
int func(arb_ptr out, const arb_t inp, void * param, slong order, slong prec)
```

implementing a univariate real function $f(x)$. When called, *func* should write to *out* the first *order* coefficients in the Taylor series expansion of $f(x)$ at the point *inp*, evaluated at a precision of *prec* bits. The *param* argument may be used to pass through additional parameters to the function. The return value is reserved for future use as an error code. It can be assumed that *out* and *inp* are not aliased and that *order* is positive.

ARB_CALC_SUCCESS

Return value indicating that an operation is successful.

ARB_CALC_IMPRECISE_INPUT

Return value indicating that the input to a function probably needs to be computed more accurately.

ARB_CALC_NO_CONVERGENCE

Return value indicating that an algorithm has failed to convergence, possibly due to the problem not having a solution, the algorithm not being applicable, or the precision being insufficient

8.1.2 Debugging

int **arb_calc_verbose**
If set, enables printing information about the calculation to standard output.

8.1.3 Subdivision-based root finding

arf_interval_struct

arf_interval_t
An *arf_interval_struct* consists of a pair of *arf_struct*, representing an interval used for subdivision-based root-finding. An *arf_interval_t* is defined as an array of length one of type *arf_interval_struct*, permitting an *arf_interval_t* to be passed by reference.

arf_interval_ptr
Alias for `arf_interval_struct *`, used for vectors of intervals.

arf_interval_srcptr
Alias for `const arf_interval_struct *`, used for vectors of intervals.

void **arf_interval_init**(*arf_interval_t* *v*)

void **arf_interval_clear**(*arf_interval_t* *v*)

arf_interval_ptr **_arf_interval_vec_init**(*slong* *n*)

void **_arf_interval_vec_clear**(*arf_interval_ptr* *v*, *slong* *n*)

void **arf_interval_set**(*arf_interval_t* *v*, const *arf_interval_t* *u*)

void **arf_interval_swap**(*arf_interval_t* *v*, *arf_interval_t* *u*)

void **arf_interval_get_arb**(*arb_t* *x*, const *arf_interval_t* *v*, *slong* *prec*)

void **arf_interval_printd**(const *arf_interval_t* *v*, *slong* *n*)
Helper functions for endpoint-based intervals.

void **arf_interval_fprintd**(FILE * *file*, const *arf_interval_t* *v*, *slong* *n*)
Helper functions for endpoint-based intervals.

slong **arb_calc_isolate_roots**(*arf_interval_ptr* * *found*, int ** *flags*, *arb_calc_func_t* *func*, void * *param*, const *arf_interval_t* *interval*, *slong* *maxdepth*, *slong* *maxeval*, *slong* *maxfound*, *slong* *prec*)
Rigorously isolates single roots of a real analytic function on the interior of an interval.

This routine writes an array of n interesting subintervals of *interval* to *found* and corresponding flags to *flags*, returning the integer n. The output has the following properties:

- The function has no roots on *interval* outside of the output subintervals.
- Subintervals are sorted in increasing order (with no overlap except possibly starting and ending with the same point).
- Subintervals with a flag of 1 contain exactly one (single) root.
- Subintervals with any other flag may or may not contain roots.

If no flags other than 1 occur, all roots of the function on *interval* have been isolated. If there are output subintervals on which the existence or nonexistence of roots could not be determined, the user may attempt further searches on those subintervals (possibly with increased precision and/or increased bounds for the breaking criteria). Note that roots of multiplicity higher than one and roots located exactly at endpoints cannot be isolated by the algorithm.

The following breaking criteria are implemented:

- At most *maxdepth* recursive subdivisions are attempted. The smallest details that can be distinguished are therefore about $2^{-\text{maxdepth}}$ times the width of *interval*. A typical, reasonable value might be between 20 and 50.

- If the total number of tested subintervals exceeds *maxeval*, the algorithm is terminated and any untested subintervals are added to the output. The total number of calls to *func* is thereby restricted to a small multiple of *maxeval* (the actual count can be slightly higher depending on implementation details). A typical, reasonable value might be between 100 and 100000.
- The algorithm terminates if *maxfound* roots have been isolated. In particular, setting *maxfound* to 1 can be used to locate just one root of the function even if there are numerous roots. To try to find all roots, *LONG_MAX* may be passed.

The argument *prec* denotes the precision used to evaluate the function. It is possibly also used for some other arithmetic operations performed internally by the algorithm. Note that it probably does not make sense for *maxdepth* to exceed *prec*.

Warning: it is assumed that subdivision points of *interval* can be represented exactly as floating-point numbers in memory. Do not pass $1 \pm 2^{-10^{100}}$ as input.

int **arb_calc_refine_root_bisect**(*arf_interval_t* *r*, *arb_calc_func_t* *func*, void * *param*, const *arf_interval_t* *start*, *slong* *iter*, *slong* *prec*)

Given an interval *start* known to contain a single root of *func*, refines it using *iter* bisection steps. The algorithm can return a failure code if the sign of the function at an evaluation point is ambiguous. The output r is set to a valid isolating interval (possibly just *start*) even if the algorithm fails.

8.1.4 Newton-based root finding

void **arb_calc_newton_conv_factor**(*arf_t* *conv_factor*, *arb_calc_func_t* *func*, void * *param*, const *arb_t* *conv_region*, *slong* *prec*)

Given an interval I specified by *conv_region*, evaluates a bound for $C = \sup_{t,u \in I} \frac{1}{2}|f''(t)|/|f'(u)|$, where f is the function specified by *func* and *param*. The bound is obtained by evaluating $f'(I)$ and $f''(I)$ directly. If f is ill-conditioned, I may need to be extremely precise in order to get an effective, finite bound for C.

int **arb_calc_newton_step**(*arb_t* *xnew*, *arb_calc_func_t* *func*, void * *param*, const *arb_t* *x*, const *arb_t* *conv_region*, const *arf_t* *conv_factor*, *slong* *prec*)

Performs a single step with an interval version of Newton's method. The input consists of the function f specified by *func* and *param*, a ball $x = [m - r, m + r]$ known to contain a single root of f, a ball I (*conv_region*) containing x with an associated bound (*conv_factor*) for $C = \sup_{t,u \in I} \frac{1}{2}|f''(t)|/|f'(u)|$, and a working precision *prec*.

The Newton update consists of setting $x' = [m' - r', m' + r']$ where $m' = m - f(m)/f'(m)$ and $r' = Cr^2$. The expression $m - f(m)/f'(m)$ is evaluated using ball arithmetic at a working precision of *prec* bits, and the rounding error during this evaluation is accounted for in the output. We now check that $x' \in I$ and $r' < r$. If both conditions are satisfied, we set *xnew* to x' and return *ARB_CALC_SUCCESS*. If either condition fails, we set *xnew* to x and return *ARB_CALC_NO_CONVERGENCE*, indicating that no progress is made.

int **arb_calc_refine_root_newton**(*arb_t* *r*, *arb_calc_func_t* *func*, void * *param*, const *arb_t* *start*, const *arb_t* *conv_region*, const *arf_t* *conv_factor*, *slong* *eval_extra_prec*, *slong* *prec*)

Refines a precise estimate of a single root of a function to high precision by performing several Newton steps, using nearly optimally chosen doubling precision steps.

The inputs are defined as for *arb_calc_newton_step*, except for the precision parameters: *prec* is the target accuracy and *eval_extra_prec* is the estimated number of guard bits that need to be added to evaluate the function accurately close to the root (for example, if the function is a polynomial with large coefficients of alternating signs and Horner's rule is used to evaluate it, the extra precision should typically be approximately the bit size of the coefficients).

This function returns *ARB_CALC_SUCCESS* if all attempted Newton steps are successful (note that this does not guarantee that the computed root is accurate to *prec* bits, which has to be verified by the user), only that it is more accurate than the starting ball.

On failure, *ARB_CALC_IMPRECISE_INPUT* or *ARB_CALC_NO_CONVERGENCE* may be returned. In this case, r is set to a ball for the root which is valid but likely does have full accuracy (it can possibly just be equal to the starting ball).

8.2 acb_calc.h – calculus with complex-valued functions

This module provides functions for operations of calculus over the complex numbers (intended to include root-finding, integration, and so on).

8.2.1 Types, macros and constants

acb_calc_func_t

Typedef for a pointer to a function with signature:

```
int func(acb_ptr out, const acb_t inp, void * param, slong order, slong prec)
```

implementing a univariate complex function $f(x)$. When called, *func* should write to *out* the first *order* coefficients in the Taylor series expansion of $f(x)$ at the point *inp*, evaluated at a precision of *prec* bits. The *param* argument may be used to pass through additional parameters to the function. The return value is reserved for future use as an error code. It can be assumed that *out* and *inp* are not aliased and that *order* is positive.

8.2.2 Bounds

void **acb_calc_cauchy_bound**(*arb_t bound*, *acb_calc_func_t func*, void * *param*, const *acb_t x*, const *arb_t radius*, *slong maxdepth*, *slong prec*)

Sets *bound* to a ball containing the value of the integral

$$C(x,r) = \frac{1}{2\pi r} \oint_{|z-x|=r} |f(z)|dz = \int_0^1 |f(x + re^{2\pi it})|dt$$

where f is specified by (*func*, *param*) and r is given by *radius*. The integral is computed using a simple step sum. The integration range is subdivided until the order of magnitude of b can be determined (i.e. its error bound is smaller than its midpoint), or until the step length has been cut in half *maxdepth* times. This function is currently implemented completely naively, and repeatedly subdivides the whole integration range instead of performing adaptive subdivisions.

8.2.3 Integration

int **acb_calc_integrate_taylor**(*acb_t res*, *acb_calc_func_t func*, void * *param*, const *acb_t a*, const *acb_t b*, const *arf_t inner_radius*, const *arf_t outer_radius*, *slong accuracy_goal*, *slong prec*)

Computes the integral

$$I = \int_a^b f(t)dt$$

where f is specified by (*func*, *param*), following a straight-line path between the complex numbers a and b which both must be finite.

The integral is approximated by piecewise centered Taylor polynomials. Rigorous truncation error bounds are calculated using the Cauchy integral formula. More precisely, if the Taylor series of f centered at the point m is $f(m+x) = \sum_{n=0}^{\infty} a_n x^n$, then

$$\int f(m+x) = \left(\sum_{n=0}^{N-1} a_n \frac{x^{n+1}}{n+1}\right) + \left(\sum_{n=N}^{\infty} a_n \frac{x^{n+1}}{n+1}\right).$$

For sufficiently small x, the second series converges and its absolute value is bounded by

$$\sum_{n=N}^{\infty} \frac{C(m,R)}{R^n} \frac{|x|^{n+1}}{N+1} = \frac{C(m,R)Rx}{(R-x)(N+1)} \left(\frac{x}{R}\right)^N.$$

It is required that any singularities of f are isolated from the path of integration by a distance strictly greater than the positive value *outer_radius* (which is the integration radius used for the Cauchy bound). Taylor series step lengths are chosen so as not to exceed *inner_radius*, which must be strictly smaller than *outer_radius* for convergence. A smaller *inner_radius* gives more rapid convergence of each Taylor series but means that more series might have to be used. A reasonable choice might be to set *inner_radius* to half the value of *outer_radius*, giving roughly one accurate bit per term.

The truncation point of each Taylor series is chosen so that the absolute truncation error is roughly 2^{-p} where p is given by *accuracy_goal* (in the future, this might change to a relative accuracy). Arithmetic operations and function evaluations are performed at a precision of *prec* bits. Note that due to accumulation of numerical errors, both values may have to be set higher (and the endpoints may have to be computed more accurately) to achieve a desired accuracy.

This function chooses the evaluation points uniformly rather than implementing adaptive subdivision.

CHAPTER

NINE

EXTRA UTILITY MODULES

Mainly for internal use.

9.1 fmpz_extras.h – extra methods for FLINT integers

This module implements a few utility methods for the FLINT multiprecision integer type (*fmpz_t*). It is mainly intended for internal use.

9.1.1 Convenience methods

void **fmpz_add_si**(*fmpz_t* *z*, const *fmpz_t* *x*, *slong* *y*)

void **fmpz_sub_si**(*fmpz_t* *z*, const *fmpz_t* *x*, *slong* *y*)
Sets z to the sum (respectively difference) of x and y.

void **fmpz_adiv_q_2exp**(*fmpz_t* *z*, const *fmpz_t* *x*, *mp_bitcnt_t* *exp*)
Sets z to $x/2^{exp}$, rounded away from zero.

void **fmpz_ui_pow_ui**(*fmpz_t* *x*, *ulong* *b*, *ulong* *e*)
Sets x to b raised to the power e.

void **fmpz_max**(*fmpz_t* *z*, const *fmpz_t* *x*, const *fmpz_t* *y*)

void **fmpz_min**(*fmpz_t* *z*, const *fmpz_t* *x*, const *fmpz_t* *y*)
Sets z to the maximum (respectively minimum) of x and y.

9.1.2 Inlined arithmetic

The *fmpz_t* bignum type uses an immediate representation for small integers, specifically when the absolute value is at most $2^{62} - 1$ (on 64-bit machines) or $2^{30} - 1$ (on 32-bit machines). The following methods completely inline the case where all operands (and possibly some intermediate values in the calculation) are known to be small. This is faster in code where all values *almost certainly will be much smaller than a full word*. In particular, these methods are used within Arb for manipulating exponents of floating-point numbers. Inlining slows down the general case, and increases code size, so these methods should not be used gratuitously.

void **fmpz_add_inline**(*fmpz_t* *z*, const *fmpz_t* *x*, const *fmpz_t* *y*)

void **fmpz_add_si_inline**(*fmpz_t* *z*, const *fmpz_t* *x*, *slong* *y*)

void **fmpz_add_ui_inline**(*fmpz_t* *z*, const *fmpz_t* *x*, *ulong* *y*)
Sets z to the sum of x and y.

void **fmpz_sub_si_inline**(*fmpz_t* *z*, const *fmpz_t* *x*, *slong* *y*)
Sets z to the difference of x and y.

void **fmpz_add2_fmpz_si_inline**(*fmpz_t* *z*, const *fmpz_t* *x*, const *fmpz_t* *y*, *slong* *c*)
Sets z to the sum of x, y, and c.

mp_size_t **_fmpz_size**(const *fmpz_t x*)
Returns the number of limbs required to represent x.

slong **_fmpz_sub_small**(const *fmpz_t x*, const *fmpz_t y*)
Computes the difference of x and y and returns the result as an *slong*. The result is clamped between -*WORD_MAX* and *WORD_MAX*, i.e. between $\pm(2^{63}-1)$ inclusive on a 64-bit machine.

9.1.3 Low-level conversions

void **fmpz_set_mpn_large**(*fmpz_t z*, *mp_srcptr src*, *mp_size_t n*, int *negative*)
Sets z to the integer represented by the n limbs in the array *src*, or minus this value if *negative* is 1. Requires $n \geq 2$ and that the top limb of *src* is nonzero. Note that *fmpz_set_ui*, *fmpz_neg_ui* can be used for single-limb integers.

void **FMPZ_GET_MPN_READONLY**(zsign, zn, zptr, ztmp, zv)
Given an *fmpz_t zv*, this macro sets *zptr* to a pointer to the limbs of *zv*, *zn* to the number of limbs, and *zsign* to a sign bit (0 if nonnegative, 1 if negative). The variable *ztmp* must be a single *mp_limb_t*, which is used as a buffer. If *zv* is a small value, *zv* itself contains no limb array that *zptr* could point to, so the single limb is copied to *ztmp* and *zptr* is set to point to *ztmp*. The case where *zv* is zero is not handled specially, and *zn* is set to 1.

void **fmpz_lshift_mpn**(*fmpz_t z*, *mp_srcptr src*, *mp_size_t n*, int *negative*, *mp_bitcnt_t shift*)
Sets z to the integer represented by the n limbs in the array *src*, or minus this value if *negative* is 1, shifted left by *shift* bits. Requires $n \geq 1$ and that the top limb of *src* is nonzero.

9.2 bool_mat.h – matrices over booleans

A *bool_mat_t* represents a dense matrix over the boolean semiring $\langle \{0,1\}, \vee, \wedge \rangle$, implemented as an array of entries of type **int**.

The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

9.2.1 Types, macros and constants

bool_mat_struct

bool_mat_t
Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An *bool_mat_t* is defined as an array of length one of type *bool_mat_struct*, permitting an *bool_mat_t* to be passed by reference.

int **bool_mat_get_entry**(const *bool_mat_t mat*, *slong i*, *slong j*)
Returns the entry of matrix *mat* at row i and column j.

void **bool_mat_set_entry**(*bool_mat_t mat*, *slong i*, *slong j*, int *x*)
Sets the entry of matrix *mat* at row i and column j to x.

bool_mat_nrows(mat)
Returns the number of rows of the matrix.

bool_mat_ncols(mat)
Returns the number of columns of the matrix.

9.2.2 Memory management

void **bool_mat_init**(*bool_mat_t mat*, *slong r*, *slong c*)
Initializes the matrix, setting it to the zero matrix with r rows and c columns.

void **bool_mat_clear**(*bool_mat_t mat*)
Clears the matrix, deallocating all entries.

int **bool_mat_is_empty**(const *bool_mat_t mat*)
Returns nonzero iff the number of rows or the number of columns in *mat* is zero. Note that this does not depend on the entry values of *mat*.

int **bool_mat_is_square**(const *bool_mat_t mat*)
Returns nonzero iff the number of rows is equal to the number of columns in *mat*.

9.2.3 Conversions

void **bool_mat_set**(*bool_mat_t dest*, const *bool_mat_t src*)
Sets *dest* to *src*. The operands must have identical dimensions.

9.2.4 Input and output

void **bool_mat_print**(const *bool_mat_t mat*)
Prints each entry in the matrix.

void **bool_mat_fprint**(FILE * *file*, const *bool_mat_t mat*)
Prints each entry in the matrix to the stream *file*.

9.2.5 Value comparisons

int **bool_mat_equal**(const *bool_mat_t mat1*, const *bool_mat_t mat2*)
Returns nonzero iff the matrices have the same dimensions and identical entries.

int **bool_mat_any**(const *bool_mat_t mat*)
Returns nonzero iff *mat* has a nonzero entry.

int **bool_mat_all**(const *bool_mat_t mat*)
Returns nonzero iff all entries of *mat* are nonzero.

int **bool_mat_is_diagonal**(const *bool_mat_t A*)
Returns nonzero iff $i \neq j \implies \bar{A_{ij}}$.

int **bool_mat_is_lower_triangular**(const *bool_mat_t A*)
Returns nonzero iff $i < j \implies \bar{A_{ij}}$.

int **bool_mat_is_transitive**(const *bool_mat_t mat*)
Returns nonzero iff $A_{ij} \wedge A_{jk} \implies A_{ik}$.

int **bool_mat_is_nilpotent**(const *bool_mat_t A*)
Returns nonzero iff some positive matrix power of A is zero.

9.2.6 Random generation

void **bool_mat_randtest**(*bool_mat_t mat*, flint_rand_t *state*)
Sets *mat* to a random matrix.

void **bool_mat_randtest_diagonal**(*bool_mat_t mat*, flint_rand_t *state*)
Sets *mat* to a random diagonal matrix.

void **bool_mat_randtest_nilpotent**(*bool_mat_t mat*, flint_rand_t *state*)
Sets *mat* to a random nilpotent matrix.

9.2.7 Special matrices

void **bool_mat_zero**(*bool_mat_t mat*)
Sets all entries in mat to zero.

void **bool_mat_one**(*bool_mat_t mat*)
Sets the entries on the main diagonal to ones, and all other entries to zero.

void **bool_mat_directed_path**(*bool_mat_t A*)
Sets A_{ij} to $j = i + 1$. Requires that A is a square matrix.

void **bool_mat_directed_cycle**(*bool_mat_t A*)
Sets A_{ij} to $j = (i + 1) \mod n$ where n is the order of the square matrix A.

9.2.8 Transpose

void **bool_mat_transpose**(*bool_mat_t dest*, const *bool_mat_t src*)
Sets *dest* to the transpose of *src*. The operands must have compatible dimensions. Aliasing is allowed.

9.2.9 Arithmetic

void **bool_mat_complement**(*bool_mat_t B*, const *bool_mat_t A*)
Sets B to the logical complement of A. That is B_{ij} is set to $\bar{A}_{ij}$. The operands must have the same dimensions.

void **bool_mat_add**(*bool_mat_t res*, const *bool_mat_t mat1*, const *bool_mat_t mat2*)
Sets *res* to the sum of *mat1* and *mat2*. The operands must have the same dimensions.

void **bool_mat_mul**(*bool_mat_t res*, const *bool_mat_t mat1*, const *bool_mat_t mat2*)
Sets *res* to the matrix product of *mat1* and *mat2*. The operands must have compatible dimensions for matrix multiplication.

void **bool_mat_mul_entrywise**(*bool_mat_t res*, const *bool_mat_t mat1*, const *bool_mat_t mat2*)
Sets *res* to the entrywise product of *mat1* and *mat2*. The operands must have the same dimensions.

void **bool_mat_sqr**(*bool_mat_t B*, const *bool_mat_t A*)
Sets B to the matrix square of A. The operands must both be square with the same dimensions.

void **bool_mat_pow_ui**(*bool_mat_t B*, const *bool_mat_t A*, *ulong exp*)
Sets B to A raised to the power *exp*. Requires that A is a square matrix.

9.2.10 Special functions

int **bool_mat_trace**(const *bool_mat_t mat*)
Returns the trace of the matrix, i.e. the sum of entries on the main diagonal of *mat*. The matrix is required to be square. The sum is in the boolean semiring, so this function returns nonzero iff any entry on the diagonal of *mat* is nonzero.

slong **bool_mat_nilpotency_degree**(const *bool_mat_t A*)
Returns the nilpotency degree of the $n \times n$ matrix A. It returns the smallest positive k such that $A^k = 0$. If no such k exists then the function returns -1 if n is positive, and otherwise it returns 0.

void **bool_mat_transitive_closure**(*bool_mat_t B*, const *bool_mat_t A*)
Sets B to the transitive closure $\sum_{k=1}^{\infty} A^k$. The matrix A is required to be square.

slong **bool_mat_get_strongly_connected_components**(*slong * p*, const *bool_mat_t A*)
Partitions the n row and column indices of the $n \times n$ matrix A according to the strongly connected components (SCC) of the graph for which A is the adjacency matrix. If the graph has k SCCs

then the function returns k, and for each vertex $i \in [0, n-1]$, p_i is set to the index of the SCC to which the vertex belongs. The SCCs themselves can be considered as nodes in a directed acyclic graph (DAG), and the SCCs are indexed in postorder with respect to that DAG.

slong **bool_mat_all_pairs_longest_walk**(*fmpz_mat_t* B, const *bool_mat_t* A)
Sets B_{ij} to the length of the longest walk with endpoint vertices i and j in the graph whose adjacency matrix is A. The matrix A must be square. Empty walks with zero length which begin and end at the same vertex are allowed. If j is not reachable from i then no walk from i to j exists and B_{ij} is set to the special value -1. If arbitrarily long walks from i to j exist then B_{ij} is set to the special value -2.

The function returns -2 if any entry of B_{ij} is -2, and otherwise it returns the maximum entry in B, except if A is empty in which case -1 is returned. Note that the returned value is one less than that of `nilpotency_degree()`.

This function can help quantify entrywise errors in a truncated evaluation of a matrix power series. If A is an indictor matrix with the same sparsity pattern as a matrix M over the real or complex numbers, and if B_{ij} does not take the special value -2, then the tail $\left[\sum_{k=N}^{\infty} a_k M^k\right]_{ij}$ vanishes when $N > B_{ij}$.

9.3 fmpr.h – Arb 1.x floating-point numbers (deprecated)

This module is deprecated, and any methods contained herein could disappear in the future. This module is mainly kept for testing the faster *arf_t* type that was introduced in Arb 2.0. Please use *arf_t* instead of the *fmpr_t* type.

A variable of type *fmpr_t* holds an arbitrary-precision binary floating-point number, i.e. a rational number of the form $x \times 2^y$ where $x, y \in \mathbb{Z}$ and x is odd; or one of the special values zero, plus infinity, minus infinity, or NaN (not-a-number).

The component x is called the *mantissa*, and y is called the *exponent*. Note that this is just one among many possible conventions: the mantissa (alternatively *significand*) is sometimes viewed as a fraction in the interval $[1/2, 1)$, with the exponent pointing to the position above the top bit rather than the position of the bottom bit, and with a separate sign.

The conventions for special values largely follow those of the IEEE floating-point standard. At the moment, there is no support for negative zero, unsigned infinity, or a NaN with a payload, though some these might be added in the future.

An *fmpr* number is exact and has no inherent "accuracy". We use the term *precision* to denote either the target precision of an operation, or the bit size of a mantissa (which in general is unrelated to the "accuracy" of the number: for example, the floating-point value 1 has a precision of 1 bit in this sense and is simultaneously an infinitely accurate approximation of the integer 1 and a 2-bit accurate approximation of $\sqrt{2} = 1.011010100\ldots_2$).

Except where otherwise noted, the output of an operation is the floating-point number obtained by taking the inputs as exact numbers, in principle carrying out the operation exactly, and rounding the resulting real number to the nearest representable floating-point number whose mantissa has at most the specified number of bits, in the specified direction of rounding. Some operations are always or optionally done exactly.

9.3.1 Types, macros and constants

fmpr_struct
An *fmpr_struct* holds a mantissa and an exponent. If the mantissa and exponent are sufficiently small, their values are stored as immediate values in the *fmpr_struct*; large values are represented by pointers to heap-allocated arbitrary-precision integers. Currently, both the mantissa and exponent are implemented using the FLINT *fmpz* type. Special values are currently encoded by the mantissa being set to zero.

fmpr_t
An *fmpr_t* is defined as an array of length one of type *fmpr_struct*, permitting an *fmpr_t* to be passed by reference.

fmpr_rnd_t
Specifies the rounding mode for the result of an approximate operation.

FMPR_RND_DOWN
Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards zero.

FMPR_RND_UP
Specifies that the result of an operation should be rounded to the nearest representable number in the direction away from zero.

FMPR_RND_FLOOR
Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards minus infinity.

FMPR_RND_CEIL
Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards plus infinity.

FMPR_RND_NEAR
Specifies that the result of an operation should be rounded to the nearest representable number, rounding to an odd mantissa if there is a tie between two values. *Warning*: this rounding mode is currently not implemented (except for a few conversions functions where this stated explicitly).

FMPR_PREC_EXACT
If passed as the precision parameter to a function, indicates that no rounding is to be performed. This must only be used when it is known that the result of the operation can be represented exactly and fits in memory (the typical use case is working small integer values). Note that, for example, adding two numbers whose exponents are far apart can easily produce an exact result that is far too large to store in memory.

9.3.2 Memory management

void **fmpr_init**(*fmpr_t* x)
Initializes the variable x for use. Its value is set to zero.

void **fmpr_clear**(*fmpr_t* x)
Clears the variable x, freeing or recycling its allocated memory.

9.3.3 Special values

void **fmpr_zero**(*fmpr_t* x)

void **fmpr_one**(*fmpr_t* x)

void **fmpr_pos_inf**(*fmpr_t* x)

void **fmpr_neg_inf**(*fmpr_t* x)

void **fmpr_nan**(*fmpr_t* x)
Sets x respectively to 0, 1, $+\infty$, $-\infty$, NaN.

int **fmpr_is_zero**(const *fmpr_t* x)

int **fmpr_is_one**(const *fmpr_t* x)

int **fmpr_is_pos_inf**(const *fmpr_t* x)

int **fmpr_is_neg_inf**(const *fmpr_t* x)

int **fmpr_is_nan**(const *fmpr_t x*)
Returns nonzero iff x respectively equals 0, 1, $+\infty$, $-\infty$, NaN.

int **fmpr_is_inf**(const *fmpr_t x*)
Returns nonzero iff x equals either $+\infty$ or $-\infty$.

int **fmpr_is_normal**(const *fmpr_t x*)
Returns nonzero iff x is a finite, nonzero floating-point value, i.e. not one of the special values 0, $+\infty$, $-\infty$, NaN.

int **fmpr_is_special**(const *fmpr_t x*)
Returns nonzero iff x is one of the special values 0, $+\infty$, $-\infty$, NaN, i.e. not a finite, nonzero floating-point value.

int **fmpr_is_finite**(*fmpr_t x*)
Returns nonzero iff x is a finite floating-point value, i.e. not one of the values $+\infty$, $-\infty$, NaN. (Note that this is not equivalent to the negation of *fmpr_is_inf()*.)

9.3.4 Assignment, rounding and conversions

slong **_fmpr_normalise**(*fmpz_t man*, *fmpz_t exp*, *slong prec*, *fmpr_rnd_t rnd*)
Rounds the mantissa and exponent in-place.

void **fmpr_set**(*fmpr_t y*, const *fmpr_t x*)
Sets y to a copy of x.

void **fmpr_swap**(*fmpr_t x*, *fmpr_t y*)
Swaps x and y efficiently.

slong **fmpr_set_round**(*fmpr_t y*, const *fmpr_t x*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_set_round_fmpz**(*fmpr_t x*, const *fmpz_t x*, *slong prec*, *fmpr_rnd_t rnd*)
Sets y to a copy of x rounded in the direction specified by rnd to the number of bits specified by prec.

slong **_fmpr_set_round_mpn**(*slong* * *shift*, *fmpz_t man*, *mp_srcptr x*, *mp_size_t xn*, int *negative*, *slong prec*, *fmpr_rnd_t rnd*)
Given an integer represented by a pointer x to a raw array of xn limbs (negated if *negative* is nonzero), sets *man* to the corresponding floating-point mantissa rounded to *prec* bits in direction *rnd*, sets *shift* to the exponent, and returns the error bound. We require that xn is positive and that the leading limb of x is nonzero.

slong **fmpr_set_round_ui_2exp_fmpz**(*fmpr_t z*, *mp_limb_t lo*, const *fmpz_t exp*, int *negative*, *slong prec*, *fmpr_rnd_t rnd*)
Sets z to the unsigned integer *lo* times two to the power *exp*, negating the value if *negative* is nonzero, and rounding the result to *prec* bits in direction *rnd*.

slong **fmpr_set_round_uiui_2exp_fmpz**(*fmpr_t z*, *mp_limb_t hi*, *mp_limb_t lo*, const *fmpz_t exp*, int *negative*, *slong prec*, *fmpr_rnd_t rnd*)
Sets z to the unsigned two-limb integer *{hi, lo}* times two to the power *exp*, negating the value if *negative* is nonzero, and rounding the result to *prec* bits in direction *rnd*.

void **fmpr_set_error_result**(*fmpr_t err*, const *fmpr_t result*, *slong rret*)
Given the return value *rret* and output variable *result* from a function performing a rounding (e.g. *fmpr_set_round* or *fmpr_add*), sets *err* to a bound for the absolute error.

void **fmpr_add_error_result**(*fmpr_t err*, const *fmpr_t err_in*, const *fmpr_t result*, *slong rret*, *slong prec*, *fmpr_rnd_t rnd*)
Like *fmpr_set_error_result*, but adds *err_in* to the error.

void **fmpr_ulp**(*fmpr_t u*, const *fmpr_t x*, *slong prec*)
Sets u to the floating-point unit in the last place (ulp) of x. The ulp is defined as in the MPFR documentation and satisfies $2^{-n}|x| < u \leq 2^{-n+1}|x|$ for any finite nonzero x. If x is a special value, u is set to the absolute value of x.

int **fmpr_check_ulp**(const *fmpr_t* *x*, *slong* *r*, *slong* *prec*)
Assume that r is the return code and x is the floating-point result from a single floating-point rounding. Then this function returns nonzero iff x and r define an error of exactly 0 or 1 ulp. In other words, this function checks that *fmpr_set_error_result()* gives exactly 0 or 1 ulp as expected.

int **fmpr_get_mpfr**(mpfr_t *x*, const *fmpr_t* *y*, mpfr_rnd_t *rnd*)
Sets the MPFR variable x to the value of y. If the precision of x is too small to allow y to be represented exactly, it is rounded in the specified MPFR rounding mode. The return value indicates the direction of rounding, following the standard convention of the MPFR library.

void **fmpr_set_mpfr**(*fmpr_t* *x*, const mpfr_t *y*)
Sets x to the exact value of the MPFR variable y.

double **fmpr_get_d**(const *fmpr_t* *x*, *fmpr_rnd_t* *rnd*)
Returns x rounded to a *double* in the direction specified by *rnd*.

void **fmpr_set_d**(*fmpr_t* *x*, double *v*)
Sets x the the exact value of the argument v of type *double*.

void **fmpr_set_ui**(*fmpr_t* *x*, *ulong* *c*)

void **fmpr_set_si**(*fmpr_t* *x*, *slong* *c*)

void **fmpr_set_fmpz**(*fmpr_t* *x*, const *fmpz_t* *c*)
Sets x exactly to the integer c.

void **fmpr_get_fmpz**(*fmpz_t* *z*, const *fmpr_t* *x*, *fmpr_rnd_t* *rnd*)
Sets z to x rounded to the nearest integer in the direction specified by *rnd*. If *rnd* is *FMPR_RND_NEAR*, rounds to the nearest even integer in case of a tie. Aborts if x is infinite, NaN or if the exponent is unreasonably large.

slong **fmpr_get_si**(const *fmpr_t* *x*, *fmpr_rnd_t* *rnd*)
Returns x rounded to the nearest integer in the direction specified by *rnd*. If *rnd* is *FMPR_RND_NEAR*, rounds to the nearest even integer in case of a tie. Aborts if x is infinite, NaN, or the value is too large to fit in an *slong*.

void **fmpr_get_fmpq**(*fmpq_t* *y*, const *fmpr_t* *x*)
Sets y to the exact value of x. The result is undefined if x is not a finite fraction.

slong **fmpr_set_fmpq**(*fmpr_t* *x*, const *fmpq_t* *y*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets x to the value of y, rounded according to *prec* and *rnd*.

void **fmpr_set_fmpz_2exp**(*fmpr_t* *x*, const *fmpz_t* *man*, const *fmpz_t* *exp*)

void **fmpr_set_si_2exp_si**(*fmpr_t* *x*, *slong* *man*, *slong* *exp*)

void **fmpr_set_ui_2exp_si**(*fmpr_t* *x*, *ulong* *man*, *slong* *exp*)
Sets x to $\text{man} \times 2^{\text{exp}}$.

slong **fmpr_set_round_fmpz_2exp**(*fmpr_t* *x*, const *fmpz_t* *man*, const *fmpz_t* *exp*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets x to $\text{man} \times 2^{\text{exp}}$, rounded according to *prec* and *rnd*.

void **fmpr_get_fmpz_2exp**(*fmpz_t* *man*, *fmpz_t* *exp*, const *fmpr_t* *x*)
Sets *man* and *exp* to the unique integers such that $x = \text{man} \times 2^{\text{exp}}$ and *man* is odd, provided that x is a nonzero finite fraction. If x is zero, both *man* and *exp* are set to zero. If x is infinite or NaN, the result is undefined.

int **fmpr_get_fmpz_fixed_fmpz**(*fmpz_t* *y*, const *fmpr_t* *x*, const *fmpz_t* *e*)

int **fmpr_get_fmpz_fixed_si**(*fmpz_t* *y*, const *fmpr_t* *x*, *slong* *e*)
Converts x to a mantissa with predetermined exponent, i.e. computes an integer y such that $y \times 2^e \approx x$, truncating if necessary. Returns 0 if exact and 1 if truncation occurred.

9.3.5 Comparisons

int **fmpr_equal**(const *fmpr_t* *x*, const *fmpr_t* *y*)
Returns nonzero iff x and y are exactly equal. This function does not treat NaN specially, i.e. NaN compares as equal to itself.

int **fmpr_cmp**(const *fmpr_t* *x*, const *fmpr_t* *y*)
Returns negative, zero, or positive, depending on whether x is respectively smaller, equal, or greater compared to y. Comparison with NaN is undefined.

int **fmpr_cmpabs**(const *fmpr_t* *x*, const *fmpr_t* *y*)

int **fmpr_cmpabs_ui**(const *fmpr_t* *x*, *ulong* *y*)
Compares the absolute values of x and y.

int **fmpr_cmp_2exp_si**(const *fmpr_t* *x*, *slong* *e*)

int **fmpr_cmpabs_2exp_si**(const *fmpr_t* *x*, *slong* *e*)
Compares x (respectively its absolute value) with 2^e.

int **fmpr_sgn**(const *fmpr_t* *x*)
Returns -1, 0 or $+1$ according to the sign of x. The sign of NaN is undefined.

void **fmpr_min**(*fmpr_t* *z*, const *fmpr_t* *a*, const *fmpr_t* *b*)

void **fmpr_max**(*fmpr_t* *z*, const *fmpr_t* *a*, const *fmpr_t* *b*)
Sets z respectively to the minimum and the maximum of a and b.

slong **fmpr_bits**(const *fmpr_t* *x*)
Returns the number of bits needed to represent the absolute value of the mantissa of x, i.e. the minimum precision sufficient to represent x exactly. Returns 0 if x is a special value.

int **fmpr_is_int**(const *fmpr_t* *x*)
Returns nonzero iff x is integer-valued.

int **fmpr_is_int_2exp_si**(const *fmpr_t* *x*, *slong* *e*)
Returns nonzero iff x equals $n2^e$ for some integer n.

void **fmpr_abs_bound_le_2exp_fmpz**(*fmpz_t* *b*, const *fmpr_t* *x*)
Sets b to the smallest integer such that $|x| \le 2^b$. If x is zero, infinity or NaN, the result is undefined.

void **fmpr_abs_bound_lt_2exp_fmpz**(*fmpz_t* *b*, const *fmpr_t* *x*)
Sets b to the smallest integer such that $|x| < 2^b$. If x is zero, infinity or NaN, the result is undefined.

slong **fmpr_abs_bound_lt_2exp_si**(const *fmpr_t* *x*)
Returns the smallest integer b such that $|x| < 2^b$, clamping the result to lie between *-FMPR_PREC_EXACT* and *FMPR_PREC_EXACT* inclusive. If x is zero, *-FMPR_PREC_EXACT* is returned, and if x is infinity or NaN, *FMPR_PREC_EXACT* is returned.

9.3.6 Random number generation

void **fmpr_randtest**(*fmpr_t* *x*, flint_rand_t *state*, *slong* *bits*, *slong* *mag_bits*)
Generates a finite random number whose mantissa has precision at most *bits* and whose exponent has at most *mag_bits* bits. The values are distributed non-uniformly: special bit patterns are generated with high probability in order to allow the test code to exercise corner cases.

void **fmpr_randtest_not_zero**(*fmpr_t* *x*, flint_rand_t *state*, *slong* *bits*, *slong* *mag_bits*)
Identical to *fmpr_randtest*, except that zero is never produced as an output.

void **fmpr_randtest_special**(*fmpr_t* *x*, flint_rand_t *state*, *slong* *bits*, *slong* *mag_bits*)
Identical to *fmpr_randtest*, except that the output occasionally is set to an infinity or NaN.

9.3.7 Input and output

void **fmpr_print**(const *fmpr_t x*)
Prints the mantissa and exponent of x as integers, precisely showing the internal representation.

void **fmpr_printd**(const *fmpr_t x*, *slong digits*)
Prints x as a decimal floating-point number, rounding to the specified number of digits. This function is currently implemented using MPFR, and does not support large exponents.

9.3.8 Arithmetic

void **fmpr_neg**(*fmpr_t y*, const *fmpr_t x*)
Sets y to the negation of x.

slong **fmpr_neg_round**(*fmpr_t y*, const *fmpr_t x*, *slong prec*, *fmpr_rnd_t rnd*)
Sets y to the negation of x, rounding the result.

void **fmpr_abs**(*fmpr_t y*, const *fmpr_t x*)
Sets y to the absolute value of x.

slong **fmpr_add**(*fmpr_t z*, const *fmpr_t x*, const *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_add_ui**(*fmpr_t z*, const *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_add_si**(*fmpr_t z*, const *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_add_fmpz**(*fmpr_t z*, const *fmpr_t x*, const *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = x + y$, rounded according to *prec* and *rnd*. The precision can be *FMPR_PREC_EXACT* to perform an exact addition, provided that the result fits in memory.

slong **_fmpr_add_eps**(*fmpr_t z*, const *fmpr_t x*, int *sign*, *slong prec*, *fmpr_rnd_t rnd*)
Sets z to the value that results by adding an infinitesimal quantity of the given sign to x, and rounding. The result is undefined if x is zero.

slong **fmpr_sub**(*fmpr_t z*, const *fmpr_t x*, const *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_sub_ui**(*fmpr_t z*, const *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_sub_si**(*fmpr_t z*, const *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_sub_fmpz**(*fmpr_t z*, const *fmpr_t x*, const *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = x - y$, rounded according to *prec* and *rnd*. The precision can be *FMPR_PREC_EXACT* to perform an exact addition, provided that the result fits in memory.

slong **fmpr_sum**(*fmpr_t s*, const *fmpr_struct* * *terms*, *slong len*, *slong prec*, *fmpr_rnd_t rnd*)
Sets s to the sum of the array *terms* of length *len*, rounded to *prec* bits in the direction *rnd*. The sum is computed as if done without any intermediate rounding error, with only a single rounding applied to the final result. Unlike repeated calls to *fmpr_add*, this function does not overflow if the magnitudes of the terms are far apart. Warning: this function is implemented naively, and the running time is quadratic with respect to *len* in the worst case.

slong **fmpr_mul**(*fmpr_t z*, const *fmpr_t x*, const *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_mul_ui**(*fmpr_t z*, const *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_mul_si**(*fmpr_t z*, const *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_mul_fmpz**(*fmpr_t z*, const *fmpr_t x*, const *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = x \times y$, rounded according to prec and rnd. The precision can be *FMPR_PREC_EXACT* to perform an exact multiplication, provided that the result fits in memory.

void **fmpr_mul_2exp_si**(*fmpr_t y*, const *fmpr_t x*, *slong e*)

void **fmpr_mul_2exp_fmpz**(*fmpr_t y*, const *fmpr_t x*, const *fmpz_t e*)
Sets y to x multiplied by 2^e without rounding.

slong **fmpr_div**(*fmpr_t z*, const *fmpr_t x*, const *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_div_ui**(*fmpr_t z*, const *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_ui_div**(*fmpr_t z*, *ulong x*, const *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_div_si**(*fmpr_t z*, const *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_si_div**(*fmpr_t z*, *slong x*, const *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_div_fmpz**(*fmpr_t z*, const *fmpr_t x*, const *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_fmpz_div**(*fmpr_t z*, const *fmpz_t x*, const *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_fmpz_div_fmpz**(*fmpr_t z*, const *fmpz_t x*, const *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = x/y$, rounded according to *prec* and *rnd*. If y is zero, z is set to NaN.

void **fmpr_divappr_abs_ubound**(*fmpr_t z*, const *fmpr_t x*, const *fmpr_t y*, *slong prec*)
Sets z to an upper bound for $|x|/|y|$, computed to a precision of approximately *prec* bits. The error can be a few ulp.

slong **fmpr_addmul**(*fmpr_t z*, const *fmpr_t x*, const *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_addmul_ui**(*fmpr_t z*, const *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_addmul_si**(*fmpr_t z*, const *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_addmul_fmpz**(*fmpr_t z*, const *fmpr_t x*, const *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = z + x \times y$, rounded according to *prec* and *rnd*. The intermediate multiplication is always performed without roundoff. The precision can be *FMPR_PREC_EXACT* to perform an exact addition, provided that the result fits in memory.

slong **fmpr_submul**(*fmpr_t z*, const *fmpr_t x*, const *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_submul_ui**(*fmpr_t z*, const *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_submul_si**(*fmpr_t z*, const *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_submul_fmpz**(*fmpr_t z*, const *fmpr_t x*, const *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = z - x \times y$, rounded according to *prec* and *rnd*. The intermediate multiplication is always performed without roundoff. The precision can be *FMPR_PREC_EXACT* to perform an exact subtraction, provided that the result fits in memory.

slong **fmpr_sqrt**(*fmpr_t y*, const *fmpr_t x*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_sqrt_ui**(*fmpr_t z*, *ulong x*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_sqrt_fmpz**(*fmpr_t z*, const *fmpz_t x*, *slong prec*, *fmpr_rnd_t rnd*)
Sets z to the square root of x, rounded according to *prec* and *rnd*. The result is NaN if x is negative.

slong **fmpr_rsqrt**(*fmpr_t z*, const *fmpr_t x*, *slong prec*, *fmpr_rnd_t rnd*)
Sets z to the reciprocal square root of x, rounded according to *prec* and *rnd*. The result is NaN if x is negative. At high precision, this is faster than computing a square root.

slong **fmpr_root**(*fmpr_t z*, const *fmpr_t x*, *ulong k*, *slong prec*, *fmpr_rnd_t rnd*)
Sets z to the k-th root of x, rounded to *prec* bits in the direction *rnd*. Warning: this function wraps MPFR, and is currently only fast for small k.

void **fmpr_pow_sloppy_fmpz**(*fmpr_t y*, const *fmpr_t b*, const *fmpz_t e*, *slong prec*, *fmpr_rnd_t rnd*)

void **fmpr_pow_sloppy_ui**(*fmpr_t y*, const *fmpr_t b*, *ulong e*, *slong prec*, *fmpr_rnd_t rnd*)

void **fmpr_pow_sloppy_si**(*fmpr_t y*, const *fmpr_t b*, *slong e*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $y = b^e$, computed using without guaranteeing correct (optimal) rounding, but guaranteeing that the result is a correct upper or lower bound if the rounding is directional. Currently requires $b \geq 0$.

9.3.9 Special functions

slong **fmpr_log**(*fmpr_t* *y*, const *fmpr_t* *x*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets y to $\log(x)$, rounded according to *prec* and *rnd*. The result is NaN if x is negative. This function is currently implemented using MPFR and does not support large exponents.

slong **fmpr_log1p**(*fmpr_t* *y*, const *fmpr_t* *x*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets y to $\log(1+x)$, rounded according to *prec* and *rnd*. This function computes an accurate value when x is small. The result is NaN if $1+x$ is negative. This function is currently implemented using MPFR and does not support large exponents.

slong **fmpr_exp**(*fmpr_t* *y*, const *fmpr_t* *x*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets y to $\exp(x)$, rounded according to *prec* and *rnd*. This function is currently implemented using MPFR and does not support large exponents.

slong **fmpr_expm1**(*fmpr_t* *y*, const *fmpr_t* *x*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets y to $\exp(x)-1$, rounded according to *prec* and *rnd*. This function computes an accurate value when x is small. This function is currently implemented using MPFR and does not support large exponents.

CHAPTER

TEN

SUPPLEMENTARY ALGORITHM NOTES

Here, we give extra proofs, error bounds, and formulas that would be too lengthy to reproduce in the documentation for each module.

10.1 General formulas and bounds

This section collects some results from real and complex analysis that are useful when deriving error bounds. Beware of typos.

10.1.1 Error propagation

We want to bound the error when $f(x+a)$ is approximated by $f(x)$. Specifically, the goal is to bound $f(x+a)-f(x)$ in terms of r for the set of values a with $|a| \le r$. Most bounds will be monotone increasing with $|a|$ (assuming that x is fixed), so for brevity we simply express the bounds in terms of $|a|$.

Theorem (generic first-order bound):

$$|f(x+a) - f(x)| \le \min(2C_0, C_1|a|)$$

where

$$C_0 = \sup_{|t| \le |a|} |f(x+t)|, \quad C_1 = \sup_{|t| \le |a|} |f'(x+t)|.$$

The statement is valid with either $a, t \in \mathbb{R}$ or $a, t \in \mathbb{C}$.

Theorem (product): For $x, y \in \mathbb{C}$ and $a, b \in \mathbb{C}$,

$$|(x+a)(y+b) - xy| \le |xb| + |ya| + |ab|.$$

Theorem (quotient): For $x, y \in \mathbb{C}$ and $a, b \in \mathbb{C}$ with $|b| < |y|$,

$$\left|\frac{x}{y} - \frac{x+a}{y+b}\right| \le \frac{|xb| + |ya|}{|y|(|y| - |b|)}.$$

Theorem (square root): For $x, a \in \mathbb{R}$ with $0 \le |a| \le x$,

$$\left|\sqrt{x+a} - \sqrt{x}\right| \le \sqrt{x}\left(1 - \sqrt{1 - \frac{|a|}{x}}\right) \le \frac{\sqrt{x}}{2}\left(\frac{|a|}{x} + \frac{|a|^2}{x^2}\right)$$

where the first inequality is an equality if $a \le 0$. (When $x = a = 0$, the limiting value is 0.)

Theorem (reciprocal square root): For $x, a \in \mathbb{R}$ with $0 \le |a| < x$,

$$\left|\frac{1}{\sqrt{x+a}} - \frac{1}{\sqrt{x}}\right| \le \frac{|a|}{2(x-|a|)^{3/2}}.$$

Theorem (k-th root): For $k > 1$ and $x, a \in \mathbb{R}$ with $0 \le |a| \le x$,

$$\left|(x+a)^{1/k} - x^{1/k}\right| \le x^{1/k} \min\left(1, \frac{1}{k} \log\left(1 + \frac{|a|}{x - |a|}\right)\right).$$

Proof: The error is largest when $a = -r$ is negative, and

$$\begin{aligned} x^{1/k} - (x-r)^{1/k} &= x^{1/k}[1 - (1 - r/x)^{1/k}] \\ &= x^{1/k}[1 - \exp(\log(1 - r/x)/k)] \le x^{1/k} \min(1, -\log(1 - r/x)/k) \\ &= x^{1/k} \min(1, \log(1 + r/(x-r))/k). \end{aligned}$$

Theorem (sine, cosine): For $x, a \in \mathbb{R}$, $|\sin(x+a) - \sin(x)| \le \min(2, |a|)$.

Theorem (logarithm): For $x, a \in \mathbb{R}$ with $0 \le |a| < x$,

$$|\log(x+a) - \log(x)| \le \log\left(1 + \frac{|a|}{x - |a|}\right),$$

with equality if $a \le 0$.

Theorem (exponential): For $x, a \in \mathbb{R}$, $|e^{x+a} - e^x| = e^x(e^a - 1) \le e^x(e^{|a|} - 1)$, with equality if $a \ge 0$.

Theorem (inverse tangent): For $x, a \in \mathbb{R}$,

$$|\operatorname{atan}(x+a) - \operatorname{atan}(x)| \le \min(\pi, C_1|a|).$$

where

$$C_1 = \sup_{|t| \le |a|} \frac{1}{1 + (x+t)^2}.$$

If $|a| < |x|$, then $C_1 = (1 + (|x| - |a|)^2)^{-1}$ gives a monotone bound.

An exact bound: if $|a| < |x|$ or $|x(x+a)| < 1$, then

$$|\operatorname{atan}(x+a) - \operatorname{atan}(x)| = \operatorname{atan}\left(\frac{|a|}{1 + x(x+a)}\right).$$

In the last formula, a case distinction has to be made depending on the signs of x and a.

10.1.2 Sums and series

Theorem (geometric bound): If $|c_k| \le C$ and $|z| \le D < 1$, then

$$\left|\sum_{k=N}^{\infty} c_k z^k\right| \le \frac{CD^N}{1 - D}.$$

Theorem (integral bound): If $f(x)$ is nonnegative and monotone decreasing, then

$$\int_N^{\infty} f(x) \le \sum_{k=N}^{\infty} f(k) \le f(N) + \int_N^{\infty} f(x)dx.$$

10.1.3 Complex analytic functions

Theorem (Cauchy's integral formula): If $f(z) = \sum_{k=0}^{\infty} c_k z^k$ is analytic (on an open subset of $\mathbb{C}$ containing the disk $D = \{z : |z| \le R\}$ in its interior, where $R > 0$), then

$$c_k = \frac{1}{2\pi i} \int_{|z|=R} \frac{f(z)}{z^{k+1}}\, dz.$$

Corollary (derivative bound):

$$|c_k| \le \frac{C}{R^k}, \quad C = \max_{|z|=R} |f(z)|.$$

Corollary (Taylor series tail): If $0 \le r < R$ and $|z| \le r$, then

$$\left| \sum_{k=N}^{\infty} c_k z^k \right| \le \frac{CD^N}{1-D}, \quad D = \left|\frac{r}{R}\right|.$$

10.1.4 Euler-Maclaurin formula

Theorem (Euler-Maclaurin): If $f(t)$ is $2M$-times differentiable, then

$$\sum_{k=L}^{U} f(k) = S + I + T + R$$

$$S = \sum_{k=L}^{N-1} f(k), \quad I = \int_N^U f(t)dt,$$

$$T = \frac{1}{2}\left(f(N) + f(U)\right) + \sum_{k=1}^{M} \frac{B_{2k}}{(2k)!}\left(f^{(2k-1)}(U) - f^{(2k-1)}(N)\right),$$

$$R = -\int_N^U \frac{B_{2M}(t - \lfloor t \rfloor)}{(2M)!} f^{(2M)}(t)dt.$$

Lemma (Bernoulli polynomials): $|B_n(t - \lfloor t \rfloor)| \le 4n!/(2\pi)^n$.

Theorem (remainder bound):

$$|R| \le \frac{4}{(2\pi)^{2M}} \int_N^U \left| f^{(2M)}(t) \right| dt.$$

Theorem (parameter derivatives): If $f(t) = f(t,x) = \sum_{k=0}^{\infty} a_k(t)x^k$ and $R = R(x) = \sum_{k=0}^{\infty} c_k x^k$ are analytic functions of x, then

$$|c_k| \le \frac{4}{(2\pi)^{2M}} \int_N^U |a_k^{(2M)}(t)|dt.$$

10.2 Algorithms for mathematical constants

Most mathematical constants are evaluated using the generic hypergeometric summation code.

10.2.1 Pi

π is computed using the Chudnovsky series

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}}$$

which is hypergeometric and adds roughly 14 digits per term. Methods based on the arithmetic-geometric mean seem to be slower by a factor three in practice.

A small trick is to compute $1/\sqrt{640320}$ instead of $\sqrt{640320}$ at the end.

10.2.2 Logarithms of integers

We use the formulas

$$\log(2) = \frac{3}{4} \sum_{k=0}^{\infty} \frac{(-1)^k (k!)^2}{2^k (2k+1)!}$$

$$\log(10) = 46\,\operatorname{atanh}(1/31) + 34\,\operatorname{atanh}(1/49) + 20\,\operatorname{atanh}(1/161)$$

10.2.3 Euler's constant

Euler's constant γ is computed using the Brent-McMillan formula (*[BM1980]*, *[MPFR2012]*)

$$\gamma = \frac{S_0(2n) - K_0(2n)}{I_0(2n)} - \log(n)$$

in which n is a free parameter and

$$S_0(x) = \sum_{k=0}^{\infty} \frac{H_k}{(k!)^2} \left(\frac{x}{2}\right)^{2k}, \quad I_0(x) = \sum_{k=0}^{\infty} \frac{1}{(k!)^2} \left(\frac{x}{2}\right)^{2k}$$

$$2x I_0(x) K_0(x) \sim \sum_{k=0}^{\infty} \frac{[(2k)!]^3}{(k!)^4 8^{2k} x^{2k}}.$$

All series are evaluated using binary splitting. The first two series are evaluated simultaneously, with the summation taken up to $k = N - 1$ inclusive where $N \geq \alpha n + 1$ and $\alpha \approx 4.9706257595442318644$ satisfies $\alpha(\log \alpha - 1) = 3$. The third series is taken up to $k = 2n - 1$ inclusive. With these parameters, it is shown in *[BJ2013]* that the error is bounded by $24e^{-8n}$.

10.2.4 Catalan's constant

Catalan's constant is computed using the hypergeometric series

$$C = \sum_{k=0}^{\infty} \frac{(-1)^k 4^{4k+1} \left(40k^2 + 56k + 19\right) [(k+1)!]^2 [(2k+2)!]^3}{(k+1)^3 (2k+1) [(4k+4)!]^2}$$

10.2.5 Khinchin's constant

Khinchin's constant K_0 is computed using the formula

$$\log K_0 = \frac{1}{\log 2} \left[\sum_{k=2}^{N-1} \log\left(\frac{k-1}{k}\right) \log\left(\frac{k+1}{k}\right) + \sum_{n=1}^{\infty} \frac{\zeta(2n, N)}{n} \sum_{k=1}^{2n-1} \frac{(-1)^{k+1}}{k} \right]$$

where $N \geq 2$ is a free parameter that can be used for tuning *[BBC1997]*. If the infinite series is truncated after $n = M$, the remainder is smaller in absolute value than

$$\sum_{n=M+1}^{\infty} \zeta(2n, N) = \sum_{n=M+1}^{\infty} \sum_{k=0}^{\infty} (k+N)^{-2n} \leq \sum_{n=M+1}^{\infty} \left(N^{-2n} + \int_0^{\infty} (t+N)^{-2n} dt \right)$$
$$= \sum_{n=M+1}^{\infty} \frac{1}{N^{2n}} \left(1 + \frac{N}{2n-1}\right) \leq \sum_{n=M+1}^{\infty} \frac{N+1}{N^{2n}} = \frac{1}{N^{2M}(N-1)} \leq \frac{1}{N^{2M}}.$$

Thus, for an error of at most 2^{-p} in the series, it is sufficient to choose $M \geq p/(2 \log_2 N)$.

10.2.6 Glaisher's constant

Glaisher's constant $A = \exp(1/12 - \zeta'(-1))$ is computed directly from this formula. We don't use the reflection formula for the zeta function, as the arithmetic in Euler-Maclaurin summation is faster at $s = -1$ than at $s = 2$.

10.2.7 Apery's constant

Apery's constant $\zeta(3)$ is computed using the hypergeometric series

$$\zeta(3) = \frac{1}{64} \sum_{k=0}^{\infty} (-1)^k (205k^2 + 250k + 77) \frac{(k!)^{10}}{[(2k+1)!]^5}.$$

10.3 Algorithms for gamma functions

10.3.1 The Stirling series

In general, the gamma function is computed via the Stirling series

$$\log \Gamma(z) = \left(z - \frac{1}{2}\right) \log z - z + \frac{\ln 2\pi}{2} + \sum_{k=1}^{n-1} \frac{B_{2k}}{2k(2k-1)z^{2k-1}} + R(n, z)$$

where (*[Olv1997]* pp. 293-295) the remainder term is exactly

$$R_n(z) = \int_0^{\infty} \frac{B_{2n} - \tilde{B}_{2n}(x)}{2n(x+z)^{2n}} dx.$$

To evaluate the gamma function of a power series argument, we substitute $z \to z + t \in \mathbb{C}[[t]]$.

Using the bound for $|x + z|$ given by *[Olv1997]* and the fact that the numerator of the integrand is bounded in absolute value by $2|B_{2n}|$, the remainder can be shown to satisfy the bound

$$|[t^k] R_n(z+t)| \le 2|B_{2n}| \frac{\Gamma(2n+k-1)}{\Gamma(k+1)\Gamma(2n+1)} \, |z| \, \left(\frac{b}{|z|}\right)^{2n+k}$$

where $b = 1/\cos(\arg(z)/2)$. Note that by trigonometric identities, assuming that $z = x + yi$, we have $b = \sqrt{1+u^2}$ where

$$u = \frac{y}{\sqrt{x^2+y^2} + x} = \frac{\sqrt{x^2+y^2} - x}{y}.$$

To use the Stirling series at p-bit precision, we select parameters r, n such that the remainder $R(n, z)$ approximately is bounded by 2^{-p}. If $|z|$ is too small for the Stirling series to give sufficient accuracy directly, we first translate to $z + r$ using the formula $\Gamma(z) = \Gamma(z+r)/(z(z+1)(z+2)\cdots(z+r-1))$.

To obtain a remainder smaller than 2^{-p}, we must choose an r such that, in the real case, $z + r > \beta p$, where $\beta > \log(2)/(2\pi) \approx 0.11$. In practice, a slightly larger factor $\beta \approx 0.2$ more closely balances n and r. A much larger β (e.g. $\beta = 1$) could be used to reduce the number of Bernoulli numbers that have to be precomputed, at the expense of slower repeated evaluation.

10.3.2 Rational arguments

We use efficient methods to compute $y = \Gamma(p/q)$ where q is one of $1, 2, 3, 4, 6$ and p is a small integer.

The cases $\Gamma(1) = 1$ and $\Gamma(1/2) = \sqrt{\pi}$ are trivial. We reduce all remaining cases to $\Gamma(1/3)$ or $\Gamma(1/4)$ using the following relations:

$$\Gamma(2/3) = \frac{2\pi}{3^{1/2}\Gamma(1/3)}, \qquad \Gamma(3/4) = \frac{2^{1/2}\pi}{\Gamma(1/4)},$$

$$\Gamma(1/6) = \frac{\Gamma(1/3)^2}{(\pi/3)^{1/2}2^{1/3}}, \qquad \Gamma(5/6) = \frac{2\pi(\pi/3)^{1/2}2^{1/3}}{\Gamma(1/3)^2}.$$

We compute $\Gamma(1/3)$ and $\Gamma(1/4)$ rapidly to high precision using

$$\Gamma(1/3) = \left(\frac{12\pi^4}{\sqrt{10}} \sum_{k=0}^{\infty} \frac{(6k)!(-1)^k}{(k!)^3(3k)!3^k 160^{3k}}\right)^{1/6}, \qquad \Gamma(1/4) = \sqrt{\frac{(2\pi)^{3/2}}{\operatorname{agm}(1,\sqrt{2})}}.$$

An alternative formula which could be used for $\Gamma(1/3)$ is

$$\Gamma(1/3) = \frac{2^{4/9}\pi^{2/3}}{3^{1/12}\left(\operatorname{agm}\left(1, \frac{1}{2}\sqrt{2+\sqrt{3}}\right)\right)^{1/3}},$$

but this appears to be slightly slower in practice.

10.4 Algorithms for polylogarithms

The polylogarithm is defined for $s, z \in \mathbb{C}$ with $|z| < 1$ by

$$\operatorname{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}$$

and for $|z| \geq 1$ by analytic continuation, except for the singular point $z = 1$.

10.4.1 Computation for small z

The power sum converges rapidly when $|z| \ll 1$. To compute the series expansion with respect to s, we substitute $s \to s + x \in \mathbb{C}[[x]]$ and obtain

$$\operatorname{Li}_{s+x}(z) = \sum_{d=0}^{\infty} x^d \frac{(-1)^d}{d!} \sum_{k=1}^{\infty} T(k)$$

where

$$T(k) = \frac{z^k \log^d(k)}{k^s}.$$

The remainder term $|\sum_{k=N}^{\infty} T(k)|$ is bounded via *mag_polylog_tail()*.

10.4.2 Expansion for general z

For general complex s, z, we write the polylogarithm as a sum of two Hurwitz zeta functions

$$\operatorname{Li}_s(z) = \frac{\Gamma(v)}{(2\pi)^v}\left[i^v \zeta\left(v, \frac{1}{2} + \frac{\log(-z)}{2\pi i}\right) + i^{-v}\zeta\left(v, \frac{1}{2} - \frac{\log(-z)}{2\pi i}\right)\right]$$

in which $s = 1 - v$. With the principal branch of $\log(-z)$, we obtain the conventional analytic continuation of the polylogarithm with a branch cut on $z \in (1, +\infty)$.

To compute the series expansion with respect to v, we substitute $v \to v + x \in \mathbb{C}[[x]]$ in this formula (at the end of the computation, we map $x \to -x$ to obtain the power series for $\mathrm{Li}_{s+x}(z)$). The right hand side becomes

$$\Gamma(v+x)[E_1 Z_1 + E_2 Z_2]$$

where $E_1 = (i/(2\pi))^{v+x}$, $Z_1 = \zeta(v+x,\ldots)$, $E_2 = (1/(2\pi i))^{v+x}$, $Z_2 = \zeta(v+x,\ldots)$.

When $v = 1$, the Z_1 and Z_2 terms become Laurent series with a leading $1/x$ term. In this case, we compute the deflated series $\tilde{Z}_1, \tilde{Z}_2 = \zeta(x,\ldots) - 1/x$. Then

$$E_1 Z_1 + E_2 Z_2 = (E_1 + E_2)/x + E_1 \tilde{Z}_1 + E_2 \tilde{Z}_2.$$

Note that $(E_1 + E_2)/x$ is a power series, since the constant term in $E_1 + E_2$ is zero when $v = 1$. So we simply compute one extra derivative of both E_1 and E_2, and shift them one step. When $v = 0, -1, -2, \ldots$, the $\Gamma(v+x)$ prefactor has a pole. In this case, we proceed analogously and formally multiply $x\,\Gamma(v+x)$ with $[E_1 Z_1 + E_2 Z_2]/x$.

Note that the formal cancellation only works when the order s (or v) is an exact integer: it is not currently possible to use this method when s is a small ball containing any of $0, 1, 2, \ldots$ (then the result becomes indeterminate).

The Hurwitz zeta method becomes inefficient when $|z| \to 0$ (it gives an indeterminate result when $z = 0$). This is not a problem since we just use the defining series for the polylogarithm in that region. It also becomes inefficient when $|z| \to \infty$, for which an asymptotic expansion would better.

10.5 Algorithms for hypergeometric functions

10.5.1 Convergent series

Let

$$T(k) = \frac{\prod_{i=0}^{p-1} (a_i)_k}{\prod_{i=0}^{q-1} (b_i)_k} z^k.$$

We compute a factor C such that

$$\left| \sum_{k=n}^{\infty} T(k) \right| \le C |T(n)|.$$

We check that $\mathrm{Re}(b+n) > 0$ for all lower parameters b. If this does not hold, C is set to infinity. Otherwise, we cancel out pairs of parameters a and b against each other. We have

$$\left| \frac{a+k}{b+k} \right| = \left| 1 + \frac{a-b}{b+k} \right| \le 1 + \frac{|a-b|}{|b+n|}$$

and

$$\left| \frac{1}{b+k} \right| \le \frac{1}{|b+n|}$$

for all $k \ge n$. This gives us a constant D such that $T(k+1) \le DT(k)$ for all $k \ge n$. If $D \ge 1$, we set C to infinity. Otherwise, we take $C = \sum_{k=0}^{\infty} D^k = (1-D)^{-1}$.

10.5.2 Convergent series of power series

The same principle is used to get tail bounds for with $a_i, b_i, z \in \mathbb{C}[[x]]$, or more precisely, bounds for each coefficient in $\sum_{k=N}^{\infty} T(k) \in \mathbb{C}[[x]]/\langle x^n \rangle$ given $a_i, b_i, z \in \mathbb{C}[[x]]/\langle x^n \rangle$. First, we fix some notation, assuming that A and B are power series:

- $A_{[k]}$ denotes the coefficient of x^k in A, and $A_{[m:n]}$ denotes the power series $\sum_{k=m}^{n-1} A_{[k]} x^k$.
- $|A|$ denotes $\sum_{k=0}^{\infty} |A_{[k]}| x^k$ (this can be viewed as an element of $\mathbb{R}_{\geq 0}[[x]]$).
- $A \leq B$ signifies that $|A|_{[k]} \leq |B|_{[k]}$ holds for all k.
- We define $\mathcal{R}(B) = |B_{[0]}| - |B_{[1:\infty]}|$.

Using the formulas

$$(AB)_{[k]} = \sum_{j=0}^{k} A_{[j]} B_{[k-j]}, \quad (1/B)_{[k]} = \frac{1}{B_{[0]}} \sum_{j=1}^{k} -B_{[j]} (1/B)_{[k-j]},$$

it is easy prove the following bounds for the coefficients of sums, products and quotients of formal power series:

$$|A + B| \leq |A| + |B|, \quad |AB| \leq |A||B|, \quad |A/B| \leq |A|/\mathcal{R}(B).$$

If $p \leq q$ and $\operatorname{Re}(b_{i\,[0]} + N) > 0$ for all b_i, then we may take

$$D = |z| \prod_{i=1}^{p} \left(1 + \frac{|a_i - b_i|}{\mathcal{R}(b_i + N)}\right) \prod_{i=p+1}^{q} \frac{1}{\mathcal{R}(b_i + N)}.$$

If $D_{[0]} < 1$,then $(1 - D)^{-1} |T(n)|$ gives the error bound.

Note when adding and multiplying power series with (complex) interval coefficients, we can use point-valued upper bounds for the absolute values instead of performing interval arithmetic throughout. For $\mathcal{R}(B)$, we must then pick a lower bound for $|B_{[0]}|$ and upper bounds for the coefficients of $|B_{[1:\infty]}|$.

10.5.3 Asymptotic series for the confluent hypergeometric function

Let $U(a, b, z)$ denote the confluent hypergeometric function of the second kind with the principal branch cut, and let $U^* = z^a U(a, b, z)$. For all $z \neq 0$ and $b \notin \mathbb{Z}$ (but valid for all b as a limit), we have (DLMF 13.2.42)

$$U(a, b, z) = \frac{\Gamma(1 - b)}{\Gamma(a - b + 1)} M(a, b, z) + \frac{\Gamma(b - 1)}{\Gamma(a)} z^{1-b} M(a - b + 1, 2 - b, z).$$

Moreover, for all $z \neq 0$ we have

$$\frac{{}_1F_1(a, b, z)}{\Gamma(b)} = \frac{(-z)^{-a}}{\Gamma(b - a)} U^*(a, b, z) + \frac{z^{a-b} e^z}{\Gamma(a)} U^*(b - a, b, -z)$$

which is equivalent to DLMF 13.2.41 (but simpler in form).

We have the asymptotic expansion

$$U^*(a, b, z) \sim {}_2F_0(a, a - b + 1, -1/z)$$

where ${}_2F_0(a, b, z)$ denotes a formal hypergeometric series, i.e.

$$U^*(a, b, z) = \sum_{k=0}^{n-1} \frac{(a)_k (a - b + 1)_k}{k! (-z)^k} + \varepsilon_n(z).$$

The error term $\varepsilon_n(z)$ is bounded according to DLMF 13.7. A case distinction is made depending on whether z lies in one of three regions which we index by R. Our formula for the error bound increases with the value of R, so we can always choose the larger out of two indices if z lies in the union of two regions.

Let $r = |b - 2a|$. If $\operatorname{Re}(z) \geq r$, set $R = 1$. Otherwise, if $\operatorname{Im}(z) \geq r$ or $\operatorname{Re}(z) \geq 0 \wedge |z| \geq r$, set $R = 2$. Otherwise, if $|z| \geq 2r$, set $R = 3$. Otherwise, the bound is infinite. If the bound is finite, we have

$$|\varepsilon_n(z)| \leq 2\alpha C_n \left| \frac{(a)_n (a - b + 1)_n}{n! z^n} \right| \exp(2\alpha\rho C_1 / |z|)$$

in terms of the following auxiliary quantities

$$\sigma = |(b-2a)/z|$$

$$C_n = \begin{cases} 1 & \text{if } R = 1 \\ \chi(n) & \text{if } R = 2 \\ (\chi(n) + \sigma\nu^2 n)\nu^n & \text{if } R = 3 \end{cases}$$

$$\nu = \left(\tfrac{1}{2} + \tfrac{1}{2}\sqrt{1-4\sigma^2}\right)^{-1/2} \le 1 + 2\sigma^2$$

$$\chi(n) = \sqrt{\pi}\Gamma(\tfrac{1}{2}n+1)/\Gamma(\tfrac{1}{2}n+\tfrac{1}{2})$$

$$\sigma' = \begin{cases} \sigma & \text{if } R \ne 3 \\ \nu\sigma & \text{if } R = 3 \end{cases}$$

$$\alpha = (1-\sigma')^{-1}$$

$$\rho = \tfrac{1}{2}|2a^2 - 2ab + b| + \sigma'(1 + \tfrac{1}{4}\sigma')(1-\sigma')^{-2}$$

10.5.4 Asymptotic series for Airy functions

Error bounds are based on Olver (DLMF section 9.7). For $\arg(z) < \pi$ and $\zeta = (2/3)z^{3/2}$, we have

$$\operatorname{Ai}(z) = \frac{e^{-\zeta}}{2\sqrt{\pi}z^{1/4}}\left[S_n(\zeta) + R_n(z)\right], \quad \operatorname{Ai}'(z) = -\frac{z^{1/4}e^{-\zeta}}{2\sqrt{\pi}}\left[(S_n'(\zeta) + R_n'(z)\right]$$

$$S_n(\zeta) = \sum_{k=0}^{n-1}(-1)^k\frac{u(k)}{\zeta^k}, \quad S_n'(\zeta) = \sum_{k=0}^{n-1}(-1)^k\frac{v(k)}{\zeta^k}$$

$$u(k) = \frac{(1/6)_k(5/6)_k}{2^k k!}, \quad v(k) = \frac{6k+1}{1-6k}u(k).$$

Assuming that n is positive, the error terms are bounded by

$$|R_n(z)| \le C|u(n)||\zeta|^{-n}, \quad |R_n'(z)| \le C|v(n)||\zeta|^{-n}$$

where

$$C = \begin{cases} 2\exp(7/(36|\zeta|)) & |\arg(z)| \le \pi/3 \\ 2\chi(n)\exp(7\pi/(72|\zeta|)) & \pi/3 \le |\arg(z)| \le 2\pi/3 \\ 4\chi(n)\exp(7\pi/(36|\operatorname{re}(\zeta)|))|\cos(\arg(\zeta))|^{-n} & 2\pi/3 \le |\arg(z)| < \pi. \end{cases}$$

For computing Bi when z is roughly in the positive half-plane, we use the connection formulas

$$\operatorname{Bi}(z) = -i(2w^{+1}\operatorname{Ai}(zw^{-2}) - \operatorname{Ai}(z))$$
$$\operatorname{Bi}(z) = +i(2w^{-1}\operatorname{Ai}(zw^{+2}) - \operatorname{Ai}(z))$$

where $w = \exp(\pi i/3)$. Combining roots of unity gives

$$\operatorname{Bi}(z) = \frac{1}{2\sqrt{\pi}z^{1/4}}[2X + iY]$$

$$\operatorname{Bi}(z) = \frac{1}{2\sqrt{\pi} z^{1/4}} [2X - iY]$$

$$X = \exp(+\zeta)[S_n(-\zeta) + R_n(z w^{\mp 2})], \quad Y = \exp(-\zeta)[S_n(\zeta) + R_n(z)]$$

where the upper formula is valid for $-\pi/3 < \arg(z) < \pi$ and the lower formula is valid for $-\pi < \arg(z) < \pi/3$. We proceed analogously for the derivative of Bi.

In the negative half-plane, we use the connection formulas

$$\operatorname{Ai}(z) = e^{+\pi i/3} \operatorname{Ai}(z_1) + e^{-\pi i/3} \operatorname{Ai}(z_2)$$

$$\operatorname{Bi}(z) = e^{-\pi i/6} \operatorname{Ai}(z_1) + e^{+\pi i/6} \operatorname{Ai}(z_2)$$

where $z_1 = -z e^{+\pi i/3}$, $z_2 = -z e^{-\pi i/3}$. Provided that $|\arg(-z)| < 2\pi/3$, we have $|\arg(z_1)|, |\arg(z_2)| < \pi$, and thus the asymptotic expansion for Ai can be used. As before, we collect roots of unity to obtain

$$\operatorname{Ai}(z) = A_1[S_n(i\zeta) + R_n(z_1)] + A_2[S_n(-i\zeta) + R_n(z_2)]$$

$$\operatorname{Bi}(z) = A_3[S_n(i\zeta) + R_n(z_1)] + A_4[S_n(-i\zeta) + R_n(z_2)]$$

where $\zeta = (2/3)(-z)^{3/2}$ and

$$A_1 = \frac{\exp(-i(\zeta - \pi/4))}{2\sqrt{\pi}(-z)^{1/4}}, \quad A_2 = \frac{\exp(+i(\zeta - \pi/4))}{2\sqrt{\pi}(-z)^{1/4}}, \quad A_3 = -iA_1, \quad A_4 = +iA_2.$$

The differentiated formulas are analogous.

10.5.5 Corner case of the Gauss hypergeometric function

In the corner case where z is near $\exp(\pm \pi i/3)$, none of the linear fractional transformations is effective. In this case, we use Taylor series to analytically continue the solution of the hypergeometric differential equation from the origin. The function $f(z) = {}_2F_1(a, b, c, z_0 + z)$ satisfies

$$f''(z) = -\frac{((z_0 + z)(a + b + 1) - c)}{(z_0 + z)(z_0 - 1 + z)} f'(z) - \frac{ab}{(z_0 + z)(z_0 - 1 + z)} f(z).$$

Knowing $f(0), f'(0)$, we can compute the consecutive derivatives recursively, and evaluating the truncated Taylor series allows us to compute $f(z), f'(z)$ to high accuracy for sufficiently small z. Some experimentation showed that two continuation steps

$$0 \quad \to \quad 0.375 \pm 0.625i \quad \to \quad 0.5 \pm 0.8125i \quad \to \quad z$$

gives good performance. Error bounds for the truncated Taylor series are obtained using the Cauchy-Kovalevskaya majorant method, following the outline in *[Hoe2001]*. The differential equation is majorized by

$$g''(z) = \frac{N + 1}{2} \left(\frac{\nu}{1 - \nu z} \right) g'(z) + \frac{(N + 1)N}{2} \left(\frac{\nu}{1 - \nu z} \right)^2 g(z)$$

provided that N and $\nu \geq \max(1/|z_0|, 1/|z_0 - 1|)$ are chosen sufficiently large. It follows that we can compute explicit numbers A, N, ν such that the simple solution $g(z) = A(1 - \nu z)^{-N}$ of the differential equation provides the bound

$$|f_{[k]}| \leq g_{[k]} = A \binom{N + k}{k} \nu^k.$$

10.6 Algorithms for the arithmetic-geometric mean

With complex variables, it is convenient to work with the univariate function $M(z) = \operatorname{agm}(1, z)$. The general case is given by $\operatorname{agm}(a, b) = aM(1, b/a)$.

10.6.1 Functional equation

If the real part of z initially is not completely nonnegative, we apply the functional equation $M(z) = (z+1)M(u)/2$ where $u = \sqrt{z}/(z+1)$.

Note that u has nonnegative real part, absent rounding error. It is not a problem for correctness if rounding makes the interval contain negative points, as this just inflates the final result.

For the derivative, the functional equation becomes $M'(z) = [M(u) - (z-1)M'(u)/((1+z)\sqrt{z})]/2$.

10.6.2 AGM iteration

Once z is in the right half plane, we can apply the AGM iteration ($2a_{n+1} = a_n + b_n, b_{n+1}^2 = a_n b_n$) directly. The correct square root is given by $\sqrt{a}\sqrt{b}$, which is computed as $\sqrt{ab}, i\sqrt{-ab}, -i\sqrt{-ab}, \sqrt{a}\sqrt{b}$ respectively if both a and b have positive real part, nonnegative imaginary part, nonpositive imaginary part, or otherwise.

It is shown in *[Dup2006]*, p. 87 that, for z with nonnegative real part, $|M(z) - a_n| \le |a_n - b_n|$.

A small improvement would be to switch to a series expansion for the last one or two steps.

10.6.3 First derivative

Assuming that z is exact and that $|\arg(z)| \le 3\pi/4$, we compute $(M(z), M'(z))$ simultaneously using a finite difference.

The basic inequality we need is $|M(z)| \le \max(1, |z|)$, which is an immediate consequence of the AGM iteration.

By Cauchy's integral formula, $|M^{(k)}(z)/k!| \le CD^k$ where $C = \max(1, |z| + r)$ and $D = 1/r$, for any $0 < r < |z|$ (we choose r to be of the order $|z|/4$). Taylor expansion now gives

$$\left| \frac{M(z+h) - M(z)}{h} - M'(z) \right| \le \frac{CD^2 h}{1 - Dh}$$

assuming that h is chosen so that it satisfies $hD < 1$.

The forward finite difference requires two function evaluations at doubled precision. It would be more efficient to use a central difference (this could be implemented in the future).

When z is not exact, we evaluate at the midpoint as above and bound the propagated error using derivatives. Again by Cauchy's integral formula, we have

$$|M'(z+\varepsilon)| \le \frac{\max(1, |z| + |\varepsilon| + r)}{r}$$
$$|M''(z+\varepsilon)| \le \frac{2\max(1, |z| + |\varepsilon| + r)}{r^2}$$

assuming that the circle centered on z with radius $|\varepsilon| + r$ does not cross the negative half axis. We choose r of order $|z|/2$ and verify that all assumptions hold.

10.6.4 Higher derivatives

The function $W(z) = 1/M(z)$ is D-finite. The coefficients of $W(z+x) = \sum_{k=0}^{\infty} c_k x^k$ satisfy

$$-2z(z^2-1)c_2 = (3z^2-1)c_1 + zc_0,$$

$$-(k+2)(k+3)z(z^2-1)c_{k+3} = (k+2)^2(3z^2-1)c_{k+2} + (3k(k+3)+7)zc_{k+1} + (k+1)^2 c_k$$

in general, and

$$-(k+2)^2 c_{k+2} = (3k(k+3)+7)c_{k+1} + (k+1)^2 c_k$$

when $z = 1$.

CHAPTER

ELEVEN

HISTORY, CREDITS AND REFERENCES

11.1 Credits and references

11.1.1 License

Arb is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License (LGPL) as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

Arb is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Arb (see the LICENSE file in the root of the Arb source directory). If not, see http://www.gnu.org/licenses/.

Versions of Arb up to and including 2.8 were distributed under the GNU General Public License (GPL), not the LGPL. The switch to the LGPL applies retroactively; i.e. users may redistribute older versions of Arb under the LGPL if they prefer.

11.1.2 Authors

Fredrik Johansson is the main author. The project was started in 2012 as a numerical extension of FLINT, and the initial design was heavily based on FLINT 2.0 (with particular credit to Bill Hart and Sebastian Pancratz).

Several people have contributed patches, bug reports, or substantial feedback. This list is probably incomplete.

- Bill Hart - build system, Windows 64 support, design of FLINT
- Sebastian Pancratz - divide-and-conquer polynomial composition algorithm (taken from FLINT)
- The MPFR development team - Arb includes two-limb multiplication code taken from MPFR
- Jonathan Bober - Dirichlet characters (the code in Arb is based on his Cython implementation), C++ compatibility fixes
- Yuri Matiyasevich - feedback about the zeta function and root-finding code
- Abhinav Baid - dot product and norm functions
- Ondřej Čertík - bug reports, feedback
- Andrew Booker - bug reports, feedback
- Francesco Biscani - C++ compatibility fixes, feedback
- Clemens Heuberger - work on Arb interface in Sage, feedback
- Marc Mezzarobba - work on Arb interface in Sage, bug reports, feedback

- Pascal Molin - feedback
- Ricky Farr - convenience functions, feedback
- Marcello Seri - fix for static builds on OS X
- Tommy Hofmann - matrix transpose, comparison, other utility methods, Julia interface
- Alexander Kobel - documentation and code cleanup patches
- Hrvoje Abraham - patches for MinGW compatibility
- Julien Puydt - soname versioning support
- Alex Griffing - sinc function, matrix trace, improved matrix squaring, boolean matrices, improved structured matrix exponentials, Cholesky decomposition, miscellaneous patches
- Jeroen Demeyer - patch for major bug on PPC64

11.1.3 Funding

From 2012 to July 2014, Fredrik's work on Arb was supported by Austrian Science Fund FWF Grant Y464-N18 (Fast Computer Algebra for Special Functions). During that period, he was a PhD student (and briefly a postdoc) at RISC, Johannes Kepler University, Linz, supervised by Manuel Kauers.

From September 2014 to October 2015, Fredrik was a postdoc at INRIA Bordeaux and Institut de Mathématiques de Bordeaux, in the LFANT project-team headed by Andreas Enge. During that period, Fredrik's work on Arb was supported by ERC Starting Grant ANTICS 278537 (Algorithmic Number Theory in Computer Science) http://cordis.europa.eu/project/rcn/101288_en.html Since October 2015, Fredrik is a CR2 researcher in the LFANT team, funded by INRIA.

11.1.4 Software

The following software has been helpful in the development of Arb.

- GMP (Torbjörn Granlund and others), http://gmplib.org
- MPIR (Brian Gladman, Jason Moxham, William Hart and others), http://mpir.org
- MPFR (Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Philippe Théveny, Paul Zimmermann and others), http://mpfr.org
- FLINT (William Hart, Sebastian Pancratz, Andy Novocin, Fredrik Johansson, David Harvey and others), http://flintlib.org
- Sage (William Stein and others), http://sagemath.org
- Pari/GP (The Pari group), http://pari.math.u-bordeaux.fr/
- SymPy (Ondřej Čertík, Aaron Meurer and others), http://sympy.org
- mpmath (Fredrik Johansson and others), http://mpmath.org
- Mathematica (Wolfram Research), http://www.wolfram.com/mathematica
- HolonomicFunctions (Christoph Koutschan), http://www.risc.jku.at/research/combinat/software/HolonomicFunctions/
- Sphinx (George Brandl and others), http://sphinx.pocoo.org
- CM (Andreas Enge), http://www.multiprecision.org/index.php?prog=cm

11.1.5 Citing Arb

If you wish to cite Arb in a scientific paper, the following reference can be used (you may also cite the manual or the website directly):

F. Johansson. "Arb: a C library for ball arithmetic", *ACM Communications in Computer Algebra*, 47(4):166-169, 2013.

In BibTeX format:

```
@article{Johansson2013arb,
  title={{A}rb: a {C} library for ball arithmetic},
  author={F. Johansson},
  journal={ACM Communications in Computer Algebra},
  volume={47},
  number={4},
  pages={166--169},
  year={2013},
  publisher={ACM}
}
```

11.1.6 Bibliography

(In the PDF edition, this section is empty. See the bibliography listing at the end of the document.)

11.2 History and changes

For more details, view the commit log in the git repository https://github.com/fredrik-johansson/arb

Old releases of the code can be accessed from https://github.com/fredrik-johansson/arb/releases

11.2.1 Old versions of the documentation

- http://fredrikj.net/arb/arb-2.8.1.pdf
- http://fredrikj.net/arb/arb-2.8.0.pdf
- http://fredrikj.net/arb/arb-2.7.0.pdf
- http://fredrikj.net/arb/arb-2.6.0.pdf
- http://fredrikj.net/arb/arb-2.5.0.pdf
- http://fredrikj.net/arb/arb-2.4.0.pdf
- http://fredrikj.net/arb/arb-2.3.0.pdf

11.2.2 2015-12-31 - version 2.8.1

- Fixed 32-bit test failure for the Laguerre function.
- Made the Laguerre function indeterminate at negative integer orders, to be consistent with the test code.

11.2.3 2015-12-29 - version 2.8.0

- Compatibility and build system
  - Windows64 support (contributed by Bill Hart).
  - Fixed a bug that broke basic arithmetic on targets where FLINT uses fallback code instead of assembly code, such as PPC64 (contributed by Jeroen Demeyer).
  - Fixed configure to use EXTRA_SHARED_FLAGS/LDFLAGS, and other build system fixes (contributed by Tommy Hofmann, Bill Hart).
  - Added soname versioning (contributed by Julien Puydt).
  - Fixed test code on MinGW (contributed by Hrvoje Abraham).
  - Miscellaneous fixes to simplify interfacing Arb from Julia.
- Arithmetic and elementary functions
  - Fixed arf_get_d to handle underflow/overflow correctly and to support round-to-nearest.
  - Added more complex inverse hyperbolic functions (acb_asin, acb_acos, acb_asinh, acb_acosh, acb_atanh).
  - Added arb_contains_int and acb_contains_int for testing whether an interval contains any integer.
  - Added acb_quadratic_roots_fmpz.
  - Improved arb_sinh to use a more accurate formula for $x < 0$.
  - Added sinc function (arb_sinc) (contributed by Alex Griffing).
  - Fixed bug in arb_exp affecting convergence for huge input.
  - Faster implementation of arb_div_2expm1_ui.
  - Added mag_root, mag_geom_series.
  - Improved and added test code for arb_add_error functions.
  - Changed arb_pow and acb_pow to make pow(0,positive) = 0 instead of nan.
  - Improved acb_sqrt to return finite output for finite input straddling the branch cut.
  - Improved arb_set_interval_arf so that [inf,inf] = inf instead of an infinite interval.
  - Added computation of Bell numbers (arb_bell_fmpz).
  - Added arb_power_sum_vec for computing power sums using Bernoulli numbers.
  - Added computation of the Fujiwara root bound for acb_poly.
  - Added code to identify all the real roots of a real polynomial (acb_poly_validate_real_roots).
  - Added several convenient assignment functions, including arb_set_d, acb_set_d, acb_set_d_d, acb_set_fmpz_fmpz (contributed by Ricky Farr).
  - Added many accessor functions (_arb/acb_vec_entry_ptr, arb_get_mid/rad_arb, acb_real/imag_ptr, arb_mid/rad_ptr, acb_get_real/imag).
  - Added missing functions acb_add_si, acb_sub_si.
  - Renamed arb_root to arb_root_ui (keeping alias) and added acb_root_ui.
- Special functions
  - Implemented the Gauss hypergeometric function 2F1 and its regularized version.
  - Fixed two bugs in acb_hypgeom_pfq_series_direct discovered while implementing 2F1. In rare cases, these could lead to incorrect values for functions depending on parameter derivatives of hypergeometric series.

    * The first bug involved incorrect handling of negative integer parameters. The bug only affected 2F1 and higher functions; it did not affect correctness of any previously implemented functions that relied on acb_hypgeom_pfq_series_direct (such as Bessel Y and K functions of integer order).
    * The second bug involved a too small bound being computed for the sum of a geometric series. The geometric series bound is nearly tight for 2F1, and the incorrect version caused immediate test failures for that function. Theoretically, this bug affected correctness of some previously-implemented functions that relied on acb_hypgeom_pfq_series_direct (such as Bessel Y and K functions of integer order), but since the geometric bound is not as tight in those cases, those functions were still reliable in practice (no failing test case has been found).
  - Implemented Airy functions and their derivatives (acb_hypgeom_airy).
  - Implemented the confluent hypergeometric function 0F1 (acb_hypgeom_0f1).
  - Implemented associated Legendre functions P and Q.
  - Implemented Chebyshev, Jacobi, Gegenbauer, Laguerre, Hermite functions.
  - Implemented spherical harmonics.
  - Added function for computing Bessel J and Y functions simultaneously.
  - Added rising factorials for non-integer n (arb_rising, acb_rising).
  - Made rising factorials use gamma function for large integer n.
  - Faster algorithm for theta constants and Dedekind eta function at very high precision.
  - Fixed erf to give finite values instead of +/-inf for big imaginary input.
  - Improved acb_zeta (and arb_zeta) to automatically use fast code for integer zeta values.
  - Added double factorial (arb_doublefac_ui).
  - Added code for generating Hilbert class polynomials (acb_modular_hilbert_class_poly).
- Matrices
  - Added faster matrix squaring (arb/acb_mat_sqr) (contributed by Alex Griffing).
  - Added matrix trace (arb/acb_mat_trace) (contributed by Alex Griffing).
  - Added arb/acb_mat_set_round_fmpz_mat, acb_mat_set(_round)_arb_mat (contributed by Tommy Hofmann).
  - Added arb/acb_mat_transpose (contributed by Tommy Hofmann).
  - Added comparison methods arb/acb_mat_eq/ne (contributed by Tommy Hofmann).
- Other
  - Added complex_plot example program.
  - Added Airy functions to real_roots example program.
  - Other minor patches were contributed by Alexander Kobel, Marc Mezzarobba, Julien Puydt.
  - Removed obsolete file config.h.

11.2.4 2015-07-14 - version 2.7.0

- Hypergeometric functions
  - Implemented Bessel I and Y functions (acb_hypgeom_bessel_i, acb_hypgeom_bessel_y).
  - Fixed bug in Bessel K function giving the wrong branch for negative real arguments.
  - Added code for evaluating complex hypergeometric series binary splitting.

  - Added code for evaluating complex hypergeometric series using fast multipoint evaluation.

- Gamma related functions
  - Implemented the Barnes G-function and its continuous logarithm (acb_barnes_g, acb_log_barnes_g).
  - Implemented the generalized polygamma function (acb_polygamma).
  - Implemented the reflection formula for the logarithmic gamma function (acb_lgamma, acb_poly_lgamma_series).
  - Implemented the digamma function of power series (arb_poly_digamma_series, acb_poly_digamma_series).
  - Improved acb_poly_zeta_series to produce exact zero imaginary parts in most cases when the result should be real-valued.
  - Made the real logarithmic gamma function (arb_lgamma, arb_poly_lgamma_series) abort more quickly for negative input.
- Elementary functions
  - Added arb_exp_expinv and acb_exp_expinv functions for simultaneously computing exp(x), exp(-x).
  - Improved acb_tan, acb_tan_pi, acb_cot and acb_cot_pi for input with large imaginary parts.
  - Added complex hyperbolic functions (acb_sinh, acb_cosh, acb_sinh_cosh, acb_tanh, acb_coth).
  - Added acb_log_sin_pi for computing the logarithmic sine function without branch cuts away from the real line.
  - Added arb_poly_cot_pi_series, acb_poly_cot_pi_series.
  - Added arf_root and improved speed of arb_root.
  - Tuned algorithm selection in arb_pow_fmpq.
  - Other
    - Added documentation for arb and acb vector functions.

11.2.5 2015-04-19 - version 2.6.0

- Special functions
  - Added the Bessel K function.
  - Added the confluent hypergeometric functions M and U.
  - Added exponential, trigonometric and logarithmic integrals ei, si, shi, ci, chi, li.
  - Added the complete elliptic integral of the second kind E.
  - Added support for computing hypergeometric functions with power series as parameters.
  - Fixed special cases in Bessel J function returning useless output.
  - Fixed precision of zeta function accidentally being capped at 7000 digits (bug in 2.5).
  - Special-cased real input in the gamma functions for complex types.
  - Fixed exp of huge numbers outputting unnecessarily useless intervals.
  - Fixed broken code in erf that sometimes gave useless output.
  - Made selection of number of terms in hypergeometric series more robust.
- Polynomials and power series.

  - Added sin_pi, cos_pi and sin_cos_pi for real and complex power series.
  - Speeded up series reciprocal and division for length = 2.
  - Added add_si methods for polynomials.
  - Made inv_series and div_series with zero input produce indeterminates instead of aborting.
  - Added arb_poly_majorant, acb_poly_majorant.
- Basic functions
  - Added comparison methods arb_eq, arb_ne, arb_lt, arb_le, arb_gt, arb_ge, acb_eq, acb_ne.
  - Added acb_rel_accuracy_bits and improved the real version.
  - Fixed precision of constants like pi behaving more nondeterministically than necessary.
  - Fixed arf_get_mag_lower(nan) to output 0 instead of inf.
- Other
  - Removed call to fmpq_dedekind_sum which only exists in the git version of flint.
  - Fixed a test code bug that could cause crashes on some systems.
  - Added fix for static build on OS X (thanks Marcello Seri).
  - Miscellaneous corrections to the documentation.

11.2.6 2015-01-28 - version 2.5.0

- String conversion
  - Added arb_set_str.
  - Added arb_get_str and arb_printn for pretty-printed rigorous decimal output.
  - Added helper functions for binary to decimal conversion.
- Core arithmetic
  - Improved speed of division when using GMP instead of MPIR.
  - Improved complex division with a small denominator.
  - Removed a little bit of overhead for complex squaring.
- Special functions
  - Faster code for atan at very high precision, used instead of mpfr_atan.
  - Optimized elementary functions slightly for small input.
  - Added modified error functions erfc and erfi.
  - Added the generalized exponential integral.
  - Added the upper incomplete gamma function.
  - Implemented the complete elliptic integral of the first kind.
  - Implemented the arithmetic-geometric mean of complex numbers.
  - Optimized arb_digamma for small integers.
  - Made mag_log_ui, mag_binpow_uiui and mag_polylog_tail proper functions.
  - Added pow, agm, erf, elliptic_k, elliptic_p as functions of complex power series.
  - Added incomplete gamma function of complex power series.

  - Improved code for bounding complex rising factorials (the old code could potentially have given wrong results in degenerate cases).
  - Added arb_sqrt1pm1, arb_atanh, arb_asinh, arb_atanh.
  - Added arb_log1p, acb_log1p, acb_atan.
  - Added arb_hurwitz_zeta.
  - Improved parameter selection in the Hurwitz zeta function to try to avoid stalling when given enormous input.
  - Optimized sqrt and rsqrt of power series when given a binomial as input.
  - Made arb_bernoulli_ui(2^64-2) not crash.
  - Fixed rgamma of negative integers returning indeterminate.
- Polynomials and matrices
  - Added characteristic polynomial computation for real and complex matrices.
  - Added polynomial set_round methods.
  - Added is_real methods for more types.
  - Added more get_unique_fmpz methods.
  - Added code for generating Swinnerton-Dyer polynomials.
  - Improved error bounding in det() and exp() of complex matrices to recognize when the result is real-valued.
  - Changed polynomial divrem to return success/fail instead of aborting on divide by zero.
- Miscellaneous
  - Added logo to documentation.
  - Made inlined functions build as part of the library.
  - Silenced a clang warning.
  - Made _acb_vec_sort_pretty a library function.

11.2.7 2014-11-15 - version 2.4.0

- Arithmetic and core functions
  - Made evaluation of sin, cos and exp at medium precision faster using the sqrt trick.
  - Optimized arb_sinh and arb_sinh_cosh.
  - Optimized complex division with a small denominator.
  - Optimized cubing of complex numbers.
  - Added floor and ceil functions for the arf and arb types.
  - Added acb_poly powering functions.
  - Added acb_exp_pi_i.
  - Added functions for evaluation of Chebyshev polynomials.
  - Fixed arb_div to output nan for input containing nan.
- Added a module acb_hypgeom for hypergeometric functions
  - Evaluation of the generalized hypergeometric function in convergent cases.
  - Evaluation of confluent hypergeometric functions using asymptotic expansions.
  - The Bessel function of the first kind for complex input.

  - The error function for complex input.
- Added a module acb_modular for modular forms and elliptic functions
  - Support for working with modular transformations.
  - Mapping a point to the fundamental domain.
  - Evaluation of Jacobi theta functions and their series expansions.
  - The Dedekind eta function.
  - The j-invariant and the modular lambda and delta function.
  - Eisenstein series.
  - The Weierstrass elliptic function and its series expansion.
- Miscellaneous
  - Fixed mag_print printing a too large exponent.
  - Fixed printd methods to use a fallback instead of aborting when printing numbers too large for MPFR.
  - Added version number string (arb_version).
  - Various additions to the documentation.

11.2.8 2014-09-25 - version 2.3.0

- Removed most of the legacy (Arb 1.x) modules.
- Updated build scripts, hopefully fixing various issues.
- New implementations of arb_sin, arb_cos, arb_sin_cos, arb_atan, arb_log, arb_exp, arb_expm1, much faster up to a few thousand bits.
- Ported the bit-burst code for high-precision exponentials to the arb type.
- Speeded up arb_log_ui_from_prev.
- Added mag_exp, mag_expm1, mag_exp_tail, mag_pow_fmpz.
- Improved various mag functions.
- Added arb_get/set_interval_mpfr, arb_get_interval_arf, and improved arb_set_interval_arf.
- Improved arf_get_fmpz.
- Prettier printing of complex numbers with negative imaginary part.
- Changed some frequently-used functions from inline to non-inline to reduce code size.

11.2.9 2014-08-01 - version 2.2.0

- Added functions for computing polylogarithms and order expansions of polylogarithms, with support for real and complex s, z.
- Added a missing cast affecting C++ compatibility.
- Generalized powsum functions to allow a geometric factor.
- Improved powsum functions slightly when the exponent is an integer.
- Faster arb_log_ui_from_prev.
- Added mag_sqrt and mag_rsqrt functions.
- Fixed various minor bugs and added missing tests and documentation entries.

11.2.10 2014-06-20 - version 2.1.0

- Ported most of the remaining functions to the new arb/acb types, including:
  - Elementary functions (log, atan, etc.).
  - Hypergeometric series summation.
  - The gamma function.
  - The Riemann zeta function and related functions.
  - Bernoulli numbers.
  - The partition function.
  - The calculus modules (rigorous real root isolation, rigorous numerical integration of complex-valued functions).
  - Example programs.
- Added several missing utility functions to the arf and mag modules.

11.2.11 2014-05-27 - version 2.0.0

- New modules mag, arf, arb, arb_poly, arb_mat, acb, acb_poly, acb_mat for higher-performance ball arithmetic.
- Poly_roots2 and hilbert_matrix2 example programs.
- Vector dot product and norm functions (contributed by Abhinav Baid).

11.2.12 2014-05-03 - version 1.1.0

- Faster and more accurate error bounds for polynomial multiplication (error bounds are now always as good as with classical multiplication, and multiplying high-degree polynomials with approximately equal coefficients now has proper quasilinear complexity).
- Faster and much less memory-hungry exponentials at very high precision.
- Improved the partition function to support n bigger than a single word, and enabled the possibility to use two threads for the computation.
- Fixed a bug in floating-point arithmetic that caused a too small bound for the rounding error to be reported when the result of an inexact operation was rounded up to a power of two (this bug did not affect the correctness of ball arithmetic, because operations on ball midpoints always round down).
- Minor optimizations to floating-point arithmetic.
- Improved argument reduction of the digamma function and short series expansions of the rising factorial.
- Removed the holonomic module for now, as it did not really do anything very useful.

11.2.13 2013-12-21 - version 1.0.0

- New example programs directory
  - poly_roots example program.
  - real_roots example program.
  - pi_digits example program.
  - hilbert_matrix example program.

  - keiper_li example program.
- New fmprb_calc module for calculus with real functions
  - Bisection-based root isolation.
  - Asymptotically fast Newton root refinement.
- New fmpcb_calc module for calculus with complex functions
  - Numerical integration using Taylor series.
- Scalar functions
  - Simplified fmprb_const_euler using published error bound.
  - Added fmprb_inv.
  - Added fmprb_trim, fmpcb_trim.
  - Added fmpcb_rsqrt (complex reciprocal square root).
  - Fixed bug in fmprb_sqrtpos with nonfinite input.
  - Slightly improved fmprb powering code.
  - Added various functions for bounding fmprs by powers of two.
  - Added fmpr_is_int.
- Polynomials and power series
  - Implemented scaling to speed up blockwise multiplication.
  - Slightly faster basecase power series exponentials.
  - Improved sin/cos/tan/exp for short power series.
  - Added complex sqrt_series, rsqrt_series.
  - Implemented the Riemann-Siegel Z and theta functions for real power series.
  - Added fmprb_poly_pow_series, fmprb_poly_pow_ui and related methods.
  - Added fmprb/fmpcb_poly_contains_fmpz_poly.
  - Faster composition by monomials.
  - Implemented Borel transform and binomial transform for real power series.
- Matrices
  - Implemented matrix exponentials.
  - Multithreaded fmprb_mat_mul.
  - Added matrix infinity norm functions.
  - Added some more matrix-scalar functions.
  - Added matrix contains and overlaps methods.
- Zeta function evaluation
  - Multithreaded power sum evaluation.
  - Faster parameter selection when computing many derivatives.
  - Implemented binary splitting to speed up computing many derivatives.
- Miscellaneous
  - Corrections for C++ compatibility (contributed by Jonathan Bober).
  - Several minor bugfixes and test code enhancements.

11.2.14 2013-08-07 - version 0.7

- Floating-point and ball functions
  - Documented, added test code, and fixed bugs for various operations involving a ball containing an infinity or NaN.
  - Added reciprocal square root functions (fmpr_rsqrt, fmprb_rsqrt) based on mpfr_rec_sqrt.
  - Faster high-precision division by not computing an explicit remainder.
  - Slightly faster computation of pi by using new reciprocal square root and division code.
  - Added an fmpr function for approximate division to speed up certain radius operations.
  - Added fmpr_set_d for conversion from double.
  - Allow use of doubles to optionally compute the partition function faster but without an error bound.
  - Bypass mpfr overflow when computing the exponential function to extremely high precision (approximately 1 billion digits).
  - Made fmprb_exp faster for large numbers at extremely high precision by skipping the log(2) removal.
  - Made fmpcb_lgamma faster at high precision by speeding up the argument reduction branch computation.
  - Added fmprb_asin, fmprb_acos.
  - Added various other utility functions to the fmprb module.
  - Added a function for computing the Glaisher constant.
  - Optimized evaluation of the Riemann zeta function at high precision.
- Polynomials and power series
  - Made squaring of polynomials faster than generic multiplication.
  - Implemented power series reversion (various algorithms) for the fmprb_poly type.
  - Added many fmprb_poly utility functions (shifting, truncating, setting/getting coefficients, etc.).
  - Improved power series division when either operand is short
  - Improved power series logarithm when the input is short.
  - Improved power series exponential to use the basecase algorithm for short input regardless of the output size.
  - Added power series square root and reciprocal square root.
  - Added atan, tan, sin, cos, sin_cos, asin, acos fmprb_poly power series functions.
  - Added Newton iteration macros to simplify various functions.
  - Added gamma functions of real and complex power series ([fmprb/fmpcb]_poly_[gamma/rgamma/lgamma]_series).
  - Added wrappers for computing the Hurwitz zeta function of a power series ([fmprb/fmpcb]_poly_zeta_series).
  - Implemented sieving and other optimizations to improve performance for evaluating the zeta function of a short power series.
  - Improved power series composition when the inner series is linear.
  - Added many fmpcb_poly versions of nearly all fmprb_poly functions.

  - Improved speed and stability of series composition/reversion by balancing the power table exponents.

- Other
  - Added support for freeing all cached data by calling flint_cleanup().
  - Introduced fmprb_ptr, fmprb_srcptr, fmpcb_ptr, fmpcb_srcptr typedefs for cleaner function signatures.
  - Various bug fixes and general cleanup.

11.2.15 2013-05-31 - version 0.6

- Made fast polynomial multiplication over the reals numerically stable by using a blockwise algorithm.
- Disabled default use of the Gauss formula for multiplication of complex polynomials, to improve numerical stability.
- Added division and remainder for complex polynomials.
- Added fast multipoint evaluation and interpolation for complex polynomials.
- Added missing fmprb_poly_sub and fmpcb_poly_sub functions.
- Faster exponentials (fmprb_exp and dependent functions) at low precision, using precomputation.
- Rewrote fmpr_add and fmpr_sub using mpn level code, improving efficiency at low precision.
- Ported the partition function implementation from flint (using ball arithmetic in all steps of the calculation to guarantee correctness).
- Ported algorithm for computing the cosine minimal polynomial from flint (using ball arithmetic to guarantee correctness).
- Support using GMP instead of MPIR.
- Only use thread-local storage when enabled in flint.
- Slightly faster error bounding for the zeta function.
- Added some other helper functions.

11.2.16 2013-03-28 - version 0.5

- Arithmetic and elementary functions
  - Added fmpr_get_fmpz, fmpr_get_si.
  - Fixed accuracy problem with fmprb_div_2expm1.
  - Special-cased squaring of complex numbers.
  - Added various fmpcb convenience functions (addmul_ui, etc).
  - Optimized fmpr_cmp_2exp_si and fmpr_cmpabs_2exp_si, and added test code for comparison functions.
  - Added fmprb_atan2, also fixing a bug in fmpcb_arg.
  - Added fmprb_sin_pi, cos_pi, sin_cos_pi, etc.
  - Added fmprb_sin_pi_fmpq (etc.) using algebraic methods for fast evaluation of roots of unity.
  - Faster fmprb_poly_evaluate and evaluate_fmpcb using rectangular splitting.
  - Added fmprb_poly_evaluate2, evaluate2_fmpcb for simultaneously evaluating the derivative.

  - Added fmprb_poly root polishing code using near-optimal Newton steps (experimental).
  - Added fmpr_root, fmprb_root (currently based on MPFR).
  - Added fmpr_min, fmpr_max.
  - Added fmprb_set_interval_fmpr, fmprb_union.
  - Added fmpr_bits, fmprb_bits, fmpcb_bits for obtaining the mantissa width.
  - Added fmprb_hypot.
  - Added complex square roots.
  - Improved fmprb_log to slightly improve speed, and properly support huge arguments.
  - Fixed exp, cosh, sinh to work with huge arguments.
  - Added fmprb_expm1.
  - Fixed sin, cos, atan to work with huge arguments.
  - Improved fmprb_pow and fmpcb_pow, including automatic detection of small integer and half-integer exponents.
  - Added many more elementary functions: fmprb_tan/cot/tanh/coth, fmpcb_tan/cot, and pi versions.
  - Added fmprb const_e, const_log2, const_log10, const_catalan.
  - Fixed ball containment/overlap checking to work operate efficiently and correctly with huge exponents.
  - Strengthened test code for many core operations.
- Special functions
  - Reorganized zeta function related code.
  - Faster evaluation of the Riemann zeta function via sieving.
  - Documented and improved efficiency of the zeta constant binary splitting code.
  - Calculate error bound in Borwein's algorithm with fmprs instead of using doubles.
  - Optimized divisions in zeta evaluation via the Euler product.
  - Use functional equation for Riemann zeta function of a negative argument.
  - Compute single Bernoulli numbers using ball arithmetic instead of relying on the floating-point code in flint.
  - Initial code for evaluating the gamma function using its Taylor series.
  - Much faster rising factorials at high precision, using difference polynomials.
  - Much faster gamma function at high precision.
  - Added complex gamma function, log gamma function, and other versions.
  - Added fmprb_agm (real arithmetic-geometric mean).
  - Added fmprb_gamma_fmpq, supporting rapid computation of gamma(p/q) for q = 1,2,3,4,6.
  - Added real and complex digamma function.
  - Fixed unnecessary recomputation of Bernoulli numbers.
  - Optimized computation of Euler's constant, and added proper error bounds.
  - Avoid reliance on doubles in the hypergeometric series tail bound.
  - Cleaned up factorials and binomials, computing factorials via gamma.
- Other

  - Added an fmpz_extras module to collect various internal fmpz helper functions.
  - Fixed detection of flint header files.
  - Fixed various other small bugs.

11.2.17 2013-01-26 - version 0.4

- Much faster fmpr_mul, fmprb_mul and set_round, resulting in general speed improvements.
- Code for computing the complex Hurwitz zeta function with derivatives.
- Fixed and documented error bounds for hypergeometric series.
- Better algorithm for series evaluation of the gamma function at a rational point.
- Much faster generation of Bernoulli numbers.
- Complex log, exp, pow, trigonometric functions (currently based on MPFR).
- Complex nth roots via Newton iteration.
- Added code for arithmetic on fmpcb_polys.
- Code for computing Khinchin's constant.
- Code for rising factorials of polynomials or power series
- Faster sin_cos.
- Better div_2expm1.
- Many other new helper functions.
- Improved thread safety.
- More test code for core operations.

11.2.18 2012-11-07 - version 0.3

- Converted documentation to Sphinx.
- New module fmpcb for ball interval arithmetic over the complex numbers
  - Conversions, utility functions and arithmetic operations.
- New module fmpcb_mat for matrices over the complex numbers
  - Conversions, utility functions and arithmetic operations.
  - Multiplication, LU decomposition, solving, inverse and determinant.
- New module fmpcb_poly for polynomials over the complex numbers
  - Root isolation for complex polynomials.
- New module fmpz_holonomic for functions/sequences defined by linear differential/difference equations with polynomial coefficients
  - Functions for creating various special sequences and functions.
  - Some closure properties for sequences.
  - Taylor series expansion for differential equations.
  - Computing the nth entry of a sequence using binary splitting.
  - Computing the nth entry mod p using fast multipoint evaluation.
- Generic binary splitting code with automatic error bounding is now used for evaluating hypergeometric series.

- Matrix powering.
- Various other helper functions.

11.2.19 2012-09-29 - version 0.2

- Code for computing the gamma function (Karatsuba, Stirling's series).
- Rising factorials.
- Fast exp_series using Newton iteration.
- Improved multiplication of small polynomials by using classical multiplication.
- Implemented error propagation for square roots.
- Polynomial division (Newton-based).
- Polynomial evaluation (Horner) and composition (divide-and-conquer).
- Product trees, fast multipoint evaluation and interpolation (various algorithms).
- Power series composition (Horner, Brent-Kung).
- Added the fmprb_mat module for matrices of balls of real numbers.
- Matrix multiplication.
- Interval-aware LU decomposition, solving, inverse and determinant.
- Many helper functions and small bugfixes.

11.2.20 2012-09-14 - version 0.1

- 2012-08-05 - Began simplified rewrite.
- 2012-04-05 - Experimental ball and polynomial code (first commit).

[WQ3a] http://functions.wolfram.com/07.11.26.0033.01

[WQ3b] http://functions.wolfram.com/07.12.27.0014.01

[WQ3c] http://functions.wolfram.com/07.12.26.0003.01

[WQ3d] http://functions.wolfram.com/07.12.26.0088.01

[Arn2010] J. Arndt, *Matters Computational*, Springer (2010), http://www.jjj.de/fxt/#fxtbook

[BBC1997] D. H. Bailey, J. M. Borwein and R. E. Crandall, "On the Khintchine constant", Mathematics of Computation 66 (1997) 417-431

[Blo2009] R. Bloemen, "Even faster zeta(2n) calculation!", https://web.archive.org/web/20141101133659/http://xn--2-umb.com/09/11/even-faster-zeta-calculation

[BBC2000] J. Borwein, D. M. Bradley and R. E. Crandall, "Computational strategies for the Riemann zeta function", Journal of Computational and Applied Mathematics 121 (2000) 247-296

[Bor2000] P. Borwein, "An Efficient Algorithm for the Riemann Zeta Function", Constructive experimental and nonlinear analysis, CMS Conference Proc. 27 (2000) 29-34, http://www.cecm.sfu.ca/personal/pborwein/PAPERS/P155.pdf

[BM1980] R. P. Brent and E. M. McMillan, "Some new algorithms for high-precision computation of Euler's constant", Mathematics of Computation 34 (1980) 305-312.

[Bre1978] R. P. Brent, "A Fortran multiple-precision arithmetic package", ACM Transactions on Mathematical Software, 4(1):57–70, March 1978.

[Bre2010] R. P. Brent, "Ramanujan and Euler's Constant", http://wwwmaths.anu.edu.au/~brent/pd/Euler_CARMA_10.pdf

[BJ2013] R. P. Brent and F. Johansson, "A bound for the error term in the Brent-McMillan algorithm", preprint (2013), http://arxiv.org/abs/1312.0039

[BZ2011] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*, Cambridge University Press (2011), http://www.loria.fr/~zimmerma/mca/pub226.html

[CP2005] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*, second edition, Springer (2005).

[Dup2006] R. Dupont. "Moyenne arithmético-géométrique, suites de Borchardt et applications." These de doctorat, École polytechnique, Palaiseau (2006). http://http://www.lix.polytechnique.fr/Labo/Regis.Dupont/these_soutenance.pdf

[EM2004] O. Espinosa and V. Moll, "A generalized polygamma function", Integral Transforms and Special Functions (2004), 101-115.

[Fil1992] S. Fillebrown, "Faster Computation of Bernoulli Numbers", Journal of Algorithms 13 (1992) 431-445

[GG2003] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, second edition, Cambridge University Press (2003)

[GVL1996] G. H. Golub and C. F. Van Loan, *Matrix Computations*, third edition, Johns Hopkins University Press (1996).

[GS2003] X. Gourdon and P. Sebah, "Numerical evaluation of the Riemann Zeta-function" (2003), http://numbers.computation.free.fr/Constants/Miscellaneous/zetaevaluations.pdf

[HZ2004] G. Hanrot and P. Zimmermann, "Newton Iteration Revisited" (2004), http://www.loria.fr/~zimmerma/papers/fastnewton.ps.gz

[Hoe2009] J. van der Hoeven, "Ball arithmetic", Technical Report, HAL 00432152 (2009), http://www.texmacs.org/joris/ball/ball-abs.html

[Hoe2001] J. van der Hoeven. "Fast evaluation of holonomic functions near and in regular singularities", Journal of Symbolic Computation, 31(6):717-743 (2001).

[Joh2012] F. Johansson, "Efficient implementation of the Hardy-Ramanujan-Rademacher formula", LMS Journal of Computation and Mathematics, Volume 15 (2012), 341-359, http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=8710297

[Joh2013] F. Johansson, "Rigorous high-precision computation of the Hurwitz zeta function and its derivatives", Numerical Algorithms, http://arxiv.org/abs/1309.2877 http://dx.doi.org/10.1007/s11075-014-9893-1

[Joh2014a] F. Johansson, *Fast and rigorous computation of special functions to high precision*, PhD thesis, RISC, Johannes Kepler University, Linz, 2014. http://fredrikj.net/thesis/

[Joh2014b] F. Johansson, "Evaluating parametric holonomic sequences using rectangular splitting", ISSAC 2014, 256-263. http://dx.doi.org/10.1145/2608628.2608629

[Joh2014c] F. Johansson, "Efficient implementation of elementary functions in the medium-precision range", http://arxiv.org/abs/1410.7176

[Joh2015] F. Johansson, "Computing Bell numbers", http://fredrikj.net/blog/2015/08/computing-bell-numbers/

[Kar1998] E. A. Karatsuba, "Fast evaluation of the Hurwitz zeta function and Dirichlet L-series", Problems of Information Transmission 34:4 (1998), 342-353, http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=ppi&paperid=425&option_lang=eng

[Kob2010] A. Kobel, "Certified Complex Numerical Root Finding", Seminar on Computational Geometry and Geometric Computing (2010), http://www.mpi-inf.mpg.de/departments/d1/teaching/ss10/Seminar_CGGC/Slides/02_Kobel_NRS.pdf

[Kri2013] A. Krishnamoorthy and D. Menon, "Matrix Inversion Using Cholesky Decomposition" Proc. of the International Conference on Signal Processing Algorithms, Architectures, Arrangements, and Applications (SPA-2013), pp. 70-72, 2013.

[MPFR2012] The MPFR team, "MPFR Algorithms" (2012), http://www.mpfr.org/algo.html

[NIST2012] National Institute of Standards and Technology, *Digital Library of Mathematical Functions* (2012), http://dlmf.nist.gov/

[Olv1997] F. Olver, *Asymptotics and special functions*, AKP Classics, AK Peters Ltd., Wellesley, MA, 1997. Reprint of the 1974 original.

[Rad1973] H. Rademacher, *Topics in analytic number theory*, Springer, 1973.

[PS1973] M. S. Paterson and L. J. Stockmeyer, "On the number of nonscalar multiplications necessary to evaluate polynomials", SIAM J. Comput (1973)

[Smi2001] D. M. Smith, "Algorithm: Fortran 90 Software for Floating-Point Multiple Precision Arithmetic, Gamma and Related Functions", Transactions on Mathematical Software 27 (2001) 377-387, http://myweb.lmu.edu/dmsmith/toms2001.pdf

[Tak2000] D. Takahashi, "A fast algorithm for computing large Fibonacci numbers", Information Processing Letters 75 (2000) 243-246, http://www.ii.uni.wroc.pl/~lorys/IPL/article75-6-1.pdf

Symbols

A

B

F

H

M

P

S

U