## CONTENTS

1 Introduction 1

2 General information 3
   2.1 Feature overview 3
   2.2 Setup 4
   2.3 Using ball arithmetic 6
   2.4 Technical conventions and potential issues 13
   2.5 Example programs 17

3 Floating-point numbers 29
   3.1 mag.h – fixed-precision unsigned floating-point numbers for bounds 29
   3.2 arf.h – arbitrary-precision floating-point numbers 35

4 Real and complex numbers 45
   4.1 arb.h – real numbers 45
   4.2 acb.h – complex numbers 65

5 Polynomials and power series 79
   5.1 arb_poly.h – polynomials over the real numbers 79
   5.2 acb_poly.h – polynomials over the complex numbers 94
   5.3 arb_fmpz_poly.h – extra methods for integer polynomials 110

6 Transforms 113
   6.1 acb_dft.h – Discrete Fourier transform 113

7 Matrices 117
   7.1 arb_mat.h – matrices over the real numbers 117
   7.2 acb_mat.h – matrices over the complex numbers 123

8 Higher mathematical functions 129
   8.1 acb_hypgeom.h – hypergeometric functions of complex variables 129
   8.2 arb_hypgeom.h – hypergeometric functions of real variables 142
   8.3 acb_elliptic.h – elliptic integrals and functions of complex variables 148
   8.4 acb_modular.h – modular forms of complex variables 152
   8.5 dirichlet.h – Dirichlet characters 158
   8.6 acb_dirichlet.h – Dirichlet L-functions, zeta functions, and related functions 162
   8.7 bernoulli.h – support for Bernoulli numbers 170
   8.8 hypgeom.h – support for hypergeometric series 171
   8.9 partitions.h – computation of the partition function 173

9 Calculus 175
   9.1 arb_calc.h – calculus with real-valued functions 175
   9.2 acb_calc.h – calculus with complex-valued functions 178

10 Extra utility modules 183
   10.1 fmpz_extras.h – extra methods for FLINT integers 183
10.2  bool_mat.h – matrices over booleans ........................................ 184
10.3  dlog.h – discrete logarithms mod ulong primes .......................... 187
10.4  fmpr.h – Arb 1.x floating-point numbers (deprecated) .................... 191

11 Supplementary algorithm notes ................................................... 199
   11.1  General formulas and bounds ............................................. 199
   11.2  Algorithms for mathematical constants ................................. 201
   11.3  Algorithms for the gamma function ..................................... 203
   11.4  Algorithms for the Hurwitz zeta function ............................... 204
   11.5  Algorithms for polylogarithms ............................................ 204
   11.6  Algorithms for hypergeometric functions ............................... 205
   11.7  Algorithms for the arithmetic-geometric mean ....................... 209

12 History, credits and references ................................................. 211
   12.1  Credits and references ................................................... 211
   12.2  History and changes ...................................................... 213

Bibliography .................................................................................. 237

Index ............................................................................................ 241
Welcome to Arb’s documentation! Arb is a C library for arbitrary-precision interval arithmetic, using a midpoint-radius representation (“ball arithmetic”). It supports real and complex numbers, polynomials, power series, matrices, and evaluation of many transcendental functions. All operations are done with automatic, rigorous error bounds. The code is thread-safe, portable, and extensively tested.

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

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

Arb is developed by Fredrik Johansson (fredrik.johansson@gmail.com), with help from many contributors (see Credits and references). Questions and discussion about Arb are welcome on the flint-devel mailing list. There is also an issue tracker for bug reports and feature requests. Development progress is sometimes covered on Fredrik’s blog.

This documentation is available in HTML format at http://arblib.org and in PDF format at http://arblib.org/arb.pdf. The version of the documentation you are currently reading was updated February 23, 2018 and describes Arb 2.13.0. Documentation for specific release versions is also available in PDF format.
Chapter Two

General Information

Feature overview

Arb builds upon FLINT, which deals with efficient computation over exact domains such as the rational numbers and finite fields. Arb extends FLINT to cover computations with real and complex numbers. The problem when computing with real and complex numbers is that approximations (typically floating-point numbers) must be used, potentially leading to incorrect results.

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 is designed for computer algebra and computational number theory, but may be useful in any area demanding reliable or precise numerical computing. Arb scales seamlessly from tens of digits up to billions of digits. Efficiency is achieved by low level optimizations and use of asymptotically fast algorithms.

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.
Arb Documentation, Release 2.13.0

- 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.

Setup

Package managers

The easiest way to install Arb including all dependencies is via ready-made packages available for various distributions. Note that some packages may not be up to date.

- Debian / Ubuntu / Linux Mint
  - https://packages.debian.org/source/sid/flint-arb
- Fedora
  - https://admin.fedoraproject.org/pkgdb/package/rpms/arb/
- Arch Linux
  - https://www.archlinux.org/packages/community/x86_64/arb/
  - https://www.archlinux.org/packages/community/i686/arb/
- Guix
  - https://www.gnu.org/software/guix/packages/

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 improvements and bug fixes.

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 2.5 or later. You may also use a git checkout of https://github.com/fredrik-johansson/flint2
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).

After the installation, you may have to run `ldconfig` to make sure that the system’s dynamic linker finds the library.

On a multicore system, `make` can be run with the `-j` flag to build in parallel. For example, use `make -j4` on a quad-core machine.

Running tests

After running `make`, it is recommended to also run `make check` to verify that all unit tests pass.

By default, the unit tests run a large number of iterations to improve the chances of detecting subtle problems. The whole test suite might take around 20 minutes on a single core (`make -jN check` if you have more cores to spare). If you are in a hurry, you can adjust the number of test iterations via the `ARB_TEST_MULTIPLIER` environment variable. For example, the following will only run 10% of the default iterations:

```
export ARB_TEST_MULTIPLIER=0.1
make check
```

It is also possible to run the unit tests for a single module, for instance:

```
make check MOD=arb_poly
```

Installation as part of FLINT (deprecated)

WARNING: this feature is being deprecated. Please install Arb as a separate library, as detailed above.

With some versions 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.

Building with MSVC

To compile arb with MSVC, compile MPIR, MPFR, pthreads-win32 and FLINT using MSVC. Install CMake >=2.8.12 and make sure it is in the path. Then go to the Arb source directory and run:
To build a Debug build, create a new build directory and pass `-DCMAKE_BUILD_TYPE=Debug` to `cmake`. To create a dll library, pass `-DBUILD_SHARED_LIBS=yes` to `cmake`. Note that creating a dll library requires CMake >= 3.5.0

If the dependencies are not found, pass `-DCMAKE_PREFIX_PATH=/path/to/deps` to `cmake` to find the dependencies.

To build tests add, pass `-DBUILD_TESTS=yes` to `cmake` and run `ctest` to run the tests.

Running code

Here is an example program to get started using Arb:

```c
#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 test.c -larb
```

Depending on the environment, you may also have to pass the flags `-lflint`, `-lmpfr`, `-lgmp` to the compiler.

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

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

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:

```
[3.1415926535897932384626433832795028841971693993751 +/- 6.28e-50]
Computed with arb-2.4.0
```

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.
Ball semantics

Let \( f : A \rightarrow B \) 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 \(\text{NaN}\)).

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 \( \text{prec} \) bits in the component \( u \), and add an upper bound for the rounding error to \( r \). In Arb, ball functions that take a \( \text{prec} \) argument as input (e.g. \( \text{arb_add()} \)) always round their output to \( \text{prec} \) bits. Some functions are always exact (e.g. \( \text{arb_neg()} \)), and thus do not take a \( \text{prec} \) argument.

The programming interface resembles that of GMP. Each \( \text{arb_t} \) variable must be initialized with \( \text{arb_init()} \) before use (this also sets its value to zero), and deallocated with \( \text{arb_clear()} \) 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:

```c
#include "arb.h"

int main()
{
    arb_t x, y;
    arb_init(x); arb_init(y);
    arb_set_ui(x, 3); /* x = 3 */
    arb_set_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);
}
```

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:

- \( \text{arb_print()} \) shows the exact internal representation of a ball, with binary exponents.
- \( \text{arb_printd()} \) shows an inexact view of the internal representation, approximated by decimal floating-point numbers.
- \( \text{arb_printn()} \) 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 \( \pi \) and prints it in three ways:

```c
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:
\[(884279719003555 \times 2^{-48}) +/- (536870913 \times 2^{-80})
3.141592653589793116 +/- 4.4409e-16
[3.141592653589793 +/- 5.61e-16]\]

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:

```c
arb_set_str(x, "2.3", prec)
arb_set_ui(x, 23);
arb_div_ui(x, x, 10, prec)
```

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

### 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 \(\pi\)), 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.

### Predicates

A ball implementation of a predicate \(f : \mathbb{R} \rightarrow \{\text{True, False}\}\) 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:

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

Instead, the following can be used:

```c
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 $x \leq y$ in mathematical notation with the meaning that $x \leq y$ 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:

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

### 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 $|t| \leq 2^{-\text{prec}} < 1$. 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 $\sin(x)$.
```c
#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 \(x^k/k!\), 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.

```c
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:

<table>
<thead>
<tr>
<th>Precision</th>
<th>sin(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bits</td>
<td>+/- 2.67e+859</td>
</tr>
<tr>
<td>128 bits</td>
<td>+/- 1.30e+840</td>
</tr>
<tr>
<td>256 bits</td>
<td>+/- 3.60e+801</td>
</tr>
<tr>
<td>512 bits</td>
<td>+/- 3.01e+724</td>
</tr>
<tr>
<td>1024 bits</td>
<td>+/- 2.18e+570</td>
</tr>
<tr>
<td>2048 bits</td>
<td>+/- 1.22e+262</td>
</tr>
<tr>
<td>4096 bits</td>
<td>+/- 0.7190842207 +/- 1.20e-11</td>
</tr>
</tbody>
</table>

As an exercise, the reader may improve the naive algorithm by making it subtract a well-chosen multiple of $2\pi$ 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 $\sin(\pi)$:

<table>
<thead>
<tr>
<th>Precision</th>
<th>sin(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bits</td>
<td>+/- 3.96e-18</td>
</tr>
<tr>
<td>128 bits</td>
<td>+/- 2.17e-37</td>
</tr>
<tr>
<td>256 bits</td>
<td>+/- 6.10e-76</td>
</tr>
<tr>
<td>512 bits</td>
<td>+/- 5.13e-153</td>
</tr>
<tr>
<td>1024 bits</td>
<td>+/- 4.01e-307</td>
</tr>
<tr>
<td>2048 bits</td>
<td>+/- 2.13e-615</td>
</tr>
<tr>
<td>4096 bits</td>
<td>+/- 6.85e-1232</td>
</tr>
<tr>
<td>8192 bits</td>
<td>+/- 6.46e-2465</td>
</tr>
<tr>
<td>16384 bits</td>
<td>+/- 5.09e-4931</td>
</tr>
<tr>
<td>32768 bits</td>
<td>+/- 5.41e-9863</td>
</tr>
</tbody>
</table>

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 $|\sin(x)| < 10^{-1000}$ 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 $\sin(\pi) = 0$.

**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** \(|f(m) - f(m + \varepsilon)|\), \(|\varepsilon| \leq r\). 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 formuals 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.

**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 \( x = 2^2 \) as input. The time and space required to compute an accurate floating-point approximation of \( \sin(x) \) or \( \exp(x) \) increases as \( 2^n \). The first case is because \( \sin(x) \) oscillates between \( -1 \) and \( 1 \) and the size of the output exponent is \( n \) 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 \( \text{prec} \). When the limit is exceeded, the output is set to a crude bound. For example, if \( x \) is too large, \( \text{arb\_sin}() \) will simply return \([\pm 1]\), and \( \text{arb\_exp}() \) 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 \([\pm 1] : [\pm 2^{-m}]\) 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 \( \alpha \) 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.

## Technical conventions and potential issues

### Integer types

Arb generally uses the \textit{int} type for boolean values and status flags.

The \textit{char}, \textit{short} and \textit{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 \textit{long} and \textit{unsigned long} do not have a standardized size in practice, FLINT defines \textit{slong} and \textit{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 \textit{mp\_limb\_t}. GMP builds with a different \textit{limb size} configuration are not supported at all. For convenience, the macro \textit{FLINT\_BITS} specifies the word length (32 or 64) of the system.

The \textit{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 \textit{slong} parameter is denoted by \textit{si}, for example \textit{arb\_add\_si()}.

The constants \textit{WORD\_MIN} and \textit{WORD\_MAX} give the range of this type. This type can be printed with \texttt{flint\_printf} using the format string \texttt{%wd}.
ulong

The \textit{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 \textit{ulong} parameter is denoted by \textit{ui}, for example \texttt{arb_add_ui()}.

The constant \texttt{UWORD\_MAX} gives the range of this type. This type can be printed with \texttt{flint\_printf} using the format string \texttt{%wu}.

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

\texttt{mp\_limb\_t}

A single limb.

\texttt{mp\_ptr}

Pointer to a writable array of limbs.

\texttt{mp\_srcptr}

Pointer to a read-only array of limbs.

\texttt{mp\_size\_t}

A limb count (always nonnegative).

\texttt{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.

\texttt{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 \texttt{fmpz\_t} type is functionally identical to the GMP \texttt{mpz\_t} type, but faster for small values.

\texttt{fmpq\_t}

FLINT multi-precision rational number.

\texttt{fmpz\_poly\_t}

\texttt{fmpq\_poly\_t}

\texttt{fmpz\_mat\_t}

\texttt{fmpq\_mat\_t}

FLINT polynomials and matrices with integer and rational coefficients.

**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 \texttt{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 \texttt{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 $2^{64}$ 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 $2^{100}$ 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.

**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.

**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 $\pi$ 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()`.

**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.

**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.

**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.

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.

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 \( \pi \) 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{...999995 digits...}42209010610577945815 +/- 3e-1000000]
```

The program prints an interval guaranteed to contain \( \pi \), 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 \( m = 20 \)).

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 \( 1/4^n \)) 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 +/- 6.3868e-1924
prec=320: 0 +/- 6.3868e-1924
prec=640: 0 +/- 6.3868e-1924
prec=1280: 0 +/- 6.3868e-1924
prec=2560: 2.955454297e-23924 +/- 6.4586e-24044
success!
```

2.5. Example programs 17
**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 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
time/peak/res/peak(MB): 167.98 294.69 5.09 7.43
log: cpu/wall(s): 0.02 0.016
gamma: cpu/wall(s): 0.02 0.016
binomial transform: cpu/wall(s): 0.01 0.016
0: -0.69314718055994530941723212145817656807550013436026 +/- 6.5389e-347
1: 0.023095708966121033184310247906495291621932127152051 +/- 2.0924e-345
2: 0.0461728676140233519286424309633943387066108314123 +/- 1.674e-344
3: 0.0692129735181082679304973488726101689942120263932 +/- 5.0219e-344
4: 0.0921961987306040964762782409439018065641673490213 +/- 2.0089e-343
5: 0.1151085429223549048622128109857276671349132303596 +/- 1.0044e-342
6: 0.13792766871372988290416713700341666356138966078654 +/- 6.0264e-342
7: 0.16063715965294229940402827538566629228442046163 +/- 2.1092e-341
8: 0.183219459643383257908193931777472185984999098273432 +/- 8.4368e-341
9: 0.2056573837091704617028938742134304741236553410044 +/- 7.5931e-340
10: 0.227933936193157746390340573684453380748385942738 +/- 7.5931e-339
991: 2.319661796161336792823738996599468256210143081341 +/- 2.4616e-11
992: 2.3203766239254884035349896518332550233162909717288 +/- 9.5363e-11
993: 2.321092061239733282116591163326282023437592414 +/- 1.8495e-10
994: 2.3218073540188462110258826121503879112747188888893 +/- 3.5907e-10
995: 2.322517392821587269287290521253140273358152533 +/- 6.978e-10
996: 2.32323448581462387333322360941370391235823071281 +/- 1.3574e-09
997: 2.3239471148860146822889542667580382034526509232475 +/- 2.6433e-09
998: 2.32465175910237000003944414232403526505148856688922209 +/- 5.1524e-09
999: 2.325354827586138211981257605206526988544993162101 +/- 1.0053e-08
1000: 2.326053161666664574065046940832238158044982041872 +/- 3.927e-08
virt/peak/res/peak(MB): 170.18 294.69 7.51 7.51
```

**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 0.001
```

Chapter 2. General information
> 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]

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 \texttt{arb\_calc} module. The program takes the following arguments:

\begin{verbatim}
real_roots function a b [-refine d] [-verbose] [-maxdepth n] [-maxeval n] [-maxfound n] [-prec n]
\end{verbatim}

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 - \(\text{Ai}(x)\) (Airy function)
- 5 - \(\text{Ai}'(x)\) (Airy function)
- 6 - \(\text{Bi}(x)\) (Airy function)
- 7 - \(\text{Bi}'(x)\) (Airy function)

The following options are available:

- \texttt{-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):

```plaintext
> 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
```

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

```plaintext
> 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 (0/6):
[14.13472514173469379045725198356247027078425711569924317568567460149963429809 +/- 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:

```plaintext
> 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):
[-9.022650853340980380158190839980089256524677753156083 +/- 4.85e-52]
refined root (1/6):
[-7.944133587120853123138280555798268532140674396972215 +/- 1.92e-52]
refined root (2/6):
[-6.786708000717589987802463844961769660538824737393494 +/- 3.84e-52]
refined root (3/6):
[-5.5205598280955105912985512931293573797214280617525 +/- 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

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

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.

**poly_roots.c**

This program finds the complex roots of an integer polynomial by calling `arb_fmpz_poly_complex_roots()`, which in turn calls `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.

If -refine d is passed, the roots are refined to a relative 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> Mignotte-like polynomial x^n + (100x+1)^m, n > m
- coeffs <c0 c1 ... cn> c0 + c1 x + ... + cn x^n

Concatenate to multiply polynomials, e.g.: p 5 t 6 coeffs 1 2 3
for P_5(x)*T_6(x)*(1+2x+3x^2)

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
computing squarefree factorization...
roots with multiplicity 1
searching for 100 roots, 100 deflated
prec=32: 0 isolated roots | cpu/wall(s): 0.098 0.098
prec=64: 0 isolated roots | cpu/wall(s): 0.247 0.247
prec=128: 0 isolated roots | cpu/wall(s): 0.498 0.497
prec=256: 0 isolated roots | cpu/wall(s): 0.713 0.713
prec=512: 100 isolated roots | cpu/wall(s): 0.104 0.105
done!
[1.00000000000000 +/- 3e-20]
[2.00000000000000 +/- 3e-19]
[3.00000000000000 +/- 1e-19]
[4.00000000000000 +/- 1e-19]
[5.00000000000000 +/- 1e-19]
...
[96.00000000000000 +/- 1e-17]
[97.00000000000000 +/- 1e-17]
[98.00000000000000 +/- 3e-17]
[99.00000000000000 +/- 3e-17]
[100.00000000000000 +/- 3e-17]
cpu/wall(s): 1.664 1.664
```

This finds the roots of a Bernoulli polynomial which has both real and complex roots:

```
> build/examples/poly_roots -refine 100 -print 20 b 16
computing squarefree factorization...
roots with multiplicity 1
searching for 16 roots, 16 deflated
prec=32: 16 isolated roots | cpu/wall(s): 0.006 0.006
prec=64: 16 isolated roots | cpu/wall(s): 0.001 0.001
```
Roots are automatically separated by multiplicity by performing an initial squarefree factorization:

```plaintext
> build/examples/poly_roots -print 5 p 5 p 5 t 7 coeffs 1 5 10 10 5 1
calculating squarefree factorization...
roots with multiplicity 1
searching for 6 roots, 3 deflated
prec=32: 3 isolated roots | cpu/wall(s): 0 0.001
done!
[-0.97493 +/- 2.10e-6]
[-0.78183 +/- 1.49e-6]
[0.43388 +/- 3.75e-6]
[0.43388 +/- 3.75e-6]
[0.78183 +/- 1.49e-6]
[0.97493 +/- 2.10e-6]
roots with multiplicity 2
searching for 4 roots, 2 deflated
prec=32: 2 isolated roots | cpu/wall(s): 0 0
 done!
[-0.90618 +/- 1.56e-7]
[-0.53847 +/- 6.91e-7]
[0.53847 +/- 6.91e-7]
[0.90618 +/- 1.56e-7]
roots with multiplicity 3
searching for 1 roots, 0 deflated
prec=32: 0 isolated roots | cpu/wall(s): 0 0
 done!
0
roots with multiplicity 5
searching for 1 roots, 1 deflated
prec=32: 1 isolated roots | cpu/wall(s): 0 0
 done!
-1.0000
cpu/wall(s): 0 0.001
```

**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.

---

2.5. Example programs 23
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 $xn = yn = 512$.

A color function can be selected with `-color`. Valid options are 0 (phase=hue, magnitude=brightness) and 1 (phase only, white-gold-black-blue-white counterclockwise).

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 $\text{Ai}$
- `bi` - Airy function $\text{Bi}$
- `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
```

This program evaluates Dirichlet $L$-functions. It takes the following input:

```
> build/examples/lvalue
```

Print value of Dirichlet $L$-function at $s = a + bi$.
Default $a = 0.5$, $b = 0$, $p = 53$, $(q, n) = (1, 0)$ (Riemann zeta)

- `[-z]` - compute $Z(s)$ instead of $L(s)$
- `[-deflate]` - remove singular term at $s = 1$
- `[-len l]` - compute $l$ terms in Taylor series at $s$

Evaluating the Riemann zeta function and the Dirichlet beta function at $s = 2$:
Evaluating the L-function for character number 101 modulo 1009 at $s = 1/2$ and $s = 1$:

> build/examples/lvalue -character 1009 101
L(s) = [-0.459256562383872 +/- 5.24e-16] + [1.346937111206009 +/- 3.03e-16]*I
cpu/wall(s): 0.012 0.012
virt/peak/res/peak(MB): 26.86 26.88 2.30 2.30

Computing the first few coefficients in the Laurent series of the Riemann zeta function at $s = 1$:

> build/examples/lvalue -re 1 -deflate -len 8
L(s) = [0.577215664901532861 +/- 7.00e-19]
L(s+x) = [-0.000342305736717224 +/- 4.20e-19]
[x^2] L(s+x) = [0.9680419394471e-5 +/- 3.87e-19]
[x^3] L(s+x) = [-6.1103181084226e-6 +/- 4.51e-20]
[x^4] L(s+x) = [-3.316240908753e-7 +/- 3.85e-20]
[x^5] L(s+x) = [1.046209484479e-7 +/- 7.78e-21]
cpu/wall(s): 0.003 0.004
virt/peak/res/peak(MB): 26.86 26.88 2.30 2.30

Evaluating the Riemann zeta function near the first nontrivial root:

> build/examples/lvalue -re 0.5 -im 14.134725
L(s) = [1.76743e-8 +/- 1.93e-14] + [-1.110203e-7 +/- 2.84e-14]*I
cpu/wall(s): 0.001 0.001
virt/peak/res/peak(MB): 26.86 26.88 2.31 2.31

This program computes the central value $L(1/2)$ for each Dirichlet L-function character modulo $q$ for each $q$ in the range $q_{\text{min}}$ to $q_{\text{max}}$. Usage:
> build/examples/lcentral
Computes central values (s = 0.5) of Dirichlet L-functions.


The first few values:

```
> build/examples/lcentral 1 8
3,2: [0.48086755769662862816122006324 +/- 7.35e-30]
4,3: [0.6676914571896091766586992930 +/- 1.62e-30]
5,2: [0.7637478801172868722451215264 +/- 2.32e-30] + [0.2166964765188606936385659310 +/- 3.06e-30]*I
5,4: [0.231750947505040157558338366176 +/- 2.21e-30]
5,3: [0.7637478801172868722451215264 +/- 2.32e-30] + [-0.2166964765188606936385659310 +/- 3.06e-30]*I
7,2: [0.31008936259836766059195052534 +/- 5.29e-30] + [-0.07264193137017790524562171245 +/- 5.48e-30]*I
7,6: [1.14658866690370833667712697646 +/- 1.95e-30]
7,4: [0.31008936259836766059195052534 +/- 5.29e-30] + [0.07264193137017790524562171245 +/- 5.48e-30]*I
7,5: [0.7637478801172868722451215264 +/- 2.32e-30] + [-0.2166964765188606936385659310 +/- 3.06e-30]*I
8,5: [0.373691712912547347038158695002 +/- 4.01e-30]
8,3: [1.10042140952554837756713576997 +/- 3.37e-30]
```

cpu/wall(s): 0.002 0.003
virt/peak/res/peak(MB): 26.32 26.34 2.35 2.35

Testing a large q:

```
> build/examples/lcentral --quiet --check --prec 256 100000 100000
```

cpu/wall(s): 1.668 1.667

```
virt/peak/res/peak(MB): 35.67 46.66 11.67 22.61
```

It is conjectured that the central value never vanishes. Running with --check verifies that the interval certainly is nonzero. This can fail with insufficient precision:

```
> build/examples/lcentral --check --prec 15 100000 100000
100000,71877: [0.1 +/- 0.0772] + [+/- 0.136]*I
100000,90629: [2e+0 +/- 0.106] + [+/- 0.920]*I
100000,28133: [+/+ 0.811] + [-2e+0 +/- 0.501]*I
100000,3141: [0.8 +/- 0.0407] + [-0.1 +/- 0.0243]*I
100000,53189: [4.0 +/- 0.0826] + [+/- 0.107]*I
100000,53253: [1.9 +/- 0.0855] + [-3.9 +/- 0.0681]*I
Value could be zero!
100000,53381: [+/+ 0.0329] + [+/- 0.0413]*I
```

Aborted

integrals.c

This program computes integrals using `acb_calc_integrate()`. Invoking the program without parameters shows usage:

```
> build/examples/integrals
Compute integrals using acb_calc_integrate.
Usage: integrals -i n [-prec p] [-tol eps] [-twice] [...]

-i n       - compute integral n (0 <= n <= 23), or "-i all"
-prec p     - precision in bits (default p = 64)
-goal p     - approximate relative accuracy goal (default p)
-tol eps    - approximate absolute error goal (default 2^-p)
-twice      - run twice (to see overhead of computing nodes)
-heap       - use heap for subinterval queue
-verbose    - show information
-verbose2   - show more information
-deg n      - use quadrature degree up to n
-eval n     - limit number of function evaluations to n
```

Chapter 2. General information
-depth n - limit subinterval queue size to n

Implemented integrals:
I0 = int_0^{100} \sin(x) dx
I1 = 4 \int_0^1 1/(1+x^2) dx
I2 = 2 \int_0^{\text{inf}} 1/(1+x^2) dx (using domain truncation)
I3 = 4 \int_0^1 \sqrt{1-x^2} dx
I4 = \int_0^8 \sin(x+\exp(x)) dx
I5 = \int_1^{101} \text{floor}(x) dx
I6 = \int_0^1 |x^4+10x^3+19x^2-6x-6| \exp(x) dx
I7 = 1/(2 pi i) \int \zeta(s) ds \text{ (closed path around s = 1)}
I8 = \int_0^1 \sin(1/x) dx \text{ (slow convergence, use -heap and/or -tol)}
I9 = \int_0^1 x \sin(1/x) dx \text{ (slow convergence, use -heap and/or -tol)}
I10 = \int_0^{10000} x^{1000} \exp(-x) dx
I11 = \int_1^{1+1000i} \gamma(x) dx
I12 = \int_{(-10)^{-10}} \sin(x) + \exp(-200-x^2) dx
I13 = \int_{(-10)0}\{1010\} \exp(x) dx \text{ (use -tol 0 for relative error)}
I14 = \int_0^{\text{inf}} \exp(-x^2) dx (using domain truncation)
I15 = \int_0^1 \text{sech}(10(x-0.2))^{-2} + \text{sech}(100(x-0.4))^{-4} + \text{sech}(1000(x-0.6))^{-6} dx
I16 = \int_0^8 (\exp(x)-\text{floor}(\exp(x))) \sin(x+\exp(x)) dx (use higher -eval)
I17 = \int_0^1 \text{sech}(x) dx (using domain truncation)
I18 = \int_0^1 \text{sech}(3x) dx (using domain truncation)
I19 = \int_0^1 -\log(x)/(1+x) dx (using domain truncation)
I20 = \int_0^1 \text{exp}(-x)/(1+\exp(-x)) dx (using domain truncation)
I21 = \int_0^{\text{inf}} \wp(x)/x^{11} dx (contour for 10th Laurent coefficient of Weierstrass p-function)
I22 = N(1000) = count zeros with 0 < t <= 1000 of \zeta(s) using argument principle
I23 = \int_0^{1000} \text{W}_0(x) dx
I24 = \int_0^1 \text{max}(|\sin(x)|, |\cos(x)|) dx
I25 = \int_{(-1)-1} \text{erf}(x/\sqrt{0.0002})*0.5+1.5)*\exp(-x) dx
I26 = \int_{(-1)-10} \text{Ai}(x) dx
I27 = \int_0^10 (x-\text{floor}(x)-1/2) \max(|\sin(x)|,|\cos(x)|) dx
I28 = \int_{(-1)-1} \text{sqrt}(x) dx
I29 = \int_0^1 \text{exp}(-x^2+ix) dx (using domain truncation)
I30 = \int_0^3 \exp(-x) \text{Ai}(-x) dx (using domain truncation)
I31 = \int_0^1 x \sin(x) / (1 + \cos(x)^2) dx

A few examples:

```
build/examples/integrals -i 4
I4 = \int_0^8 \sin(x+\exp(x)) dx ...
cpu/wall(s): 0.02 0.02
I4 = [0.34740017265725 +/- 3.95e-15]

> build/examples/integrals -i 3 -prec 333 -tol 1e-80
I3 = 4 \int_0^1 \sqrt{1-x^2} dx ...
cpu/wall(s): 0.024 0.024
I3 = [3.141592653589793238462643383279502884197169399375105820974944592307816406286209 +/- 4.24e-79]

> build/examples/integrals -i 9 -heap
I9 = \int_0^1 x \sin(1/x) dx (slow convergence, use -heap and/or -tol) ...
cpu/wall(s): 0.019 0.018
I9 = [0.3785300 +/- 3.17e-8]
```
FLOATING-POINT NUMBERS

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

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

The \texttt{mag} type is an unsigned floating-point type with a fixed-precision mantissa (30 bits) and an arbitrary-precision exponent (represented as an \texttt{fmpz} type), 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 \texttt{arf} type instead.

Types, macros and constants

\texttt{mag_struct}

A \texttt{mag_struct} holds a mantissa and an exponent. Special values are encoded by the mantissa being set to zero.

\texttt{mag}

A \texttt{mag} is defined as an array of length one of type \texttt{mag_struct}, permitting a \texttt{mag} to be passed by reference.

Memory management

void \texttt{mag_init}(\texttt{mag} \texttt{x})

Initializes the variable \texttt{x} for use. Its value is set to zero.

void \texttt{mag_clear}(\texttt{mag} \texttt{x})

Cleans the variable \texttt{x}, freeing or recycling its allocated memory.

void \texttt{mag_init_set}(\texttt{mag} \texttt{x}, const \texttt{mag} \texttt{y})

Initializes \texttt{x} and sets it to the value of \texttt{y}.

void \texttt{mag_swap}(\texttt{mag} \texttt{x}, \texttt{mag} \texttt{y})

Swaps \texttt{x} and \texttt{y} efficiently.

void \texttt{mag_set}(\texttt{mag} \texttt{x}, const \texttt{mag} \texttt{y})

Sets \texttt{x} to the value of \texttt{y}.

\texttt{mag_ptr \_mag_vec_init}(\texttt{long} \texttt{n})

Allocates a vector of length \texttt{n}. All entries are set to zero.

void \_\texttt{mag_vec_clear}(\texttt{mag_ptr} \texttt{v}, \texttt{long} \texttt{n})

Clears a vector of length \texttt{n}.
slong **mag_allocated_bytes**(const **mag_t** *x)
  Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(mag_struct)` to get the size of the object as a whole.

**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**()).

**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)
  Sets *z* respectively to the smaller or the larger of *x* and *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*.

**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.

**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.
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)
   Sets z to an upper bound for |x| \times 2^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_set_d_lower(mag_t y, double x)
   Sets y to a lower bound for |x|.
void mag_set_d_2exp_fmpz_lower(mag_t z, double x, const fmpz_t y)
void mag_set_ui_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.
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. This user must check in advance that x is of reasonable magnitude before calling this function. If x is infinite or has a bignum exponent, an abort will be raised. If the exponent otherwise is too large or too small, the available memory could be exhausted resulting in undefined behavior.
void mag_get_fmpz(fmpz_t y, const mag_t x)
   Sets y to the ceiling of x. This user must check in advance that x is of reasonable magnitude before calling this function. If x is infinite or has a bignum exponent, an abort will be raised. If the exponent otherwise is too large or too small, the available memory could be exhausted resulting in undefined behavior.
void mag_get_fmpz_lower(fmpz_t y, const mag_t x)
   Sets y to the floor of x. This user must check in advance that x is of reasonable magnitude before calling this function. If x is infinite or has a bignum exponent, an abort will be raised. If the exponent otherwise is too large or too small, the available memory could be exhausted resulting in undefined behavior.

double mag_get_d(const mag_t x)
   Returns a double giving an upper bound for x.
double mag_get_d_log2_approx(const mag_t x)
   Returns a double approximating \log_2(x), suitable for estimating magnitudes (not for rigorous bounds). The value is clamped between COEFF_MIN and COEFF_MAX.
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.

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_div_lower(mag_t z, const mag_t x, const mag_t y)
    Sets z to a lower bound for \(x/y\).
void mag_inv(mag_t z, const mag_t x)
    Sets z to an upper bound for \(1/x\).
void mag_inv_lower(mag_t z, const mag_t x)
    Sets z to a lower bound for \(1/x\).
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 \(\max(x - y, 0)\).

**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 `x 2^e`.

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 >= 0`.

void `mag_pow_ui_lower`(`mag_t z`, `const mag_t x`, `ulong e`)  
void `mag_pow_fmpz_lower`(`mag_t z`, `const mag_t x`, `const fmpz_t e`)  
Sets `z` to a lower bound for `x^e`. Requires `e >= 0`.

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

void `mag_sqrt_lower`(`mag_t z`, `const mag_t x`)  
Sets `z` to a lower 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_rsqrt_lower`(`mag_t z`, `const mag_t x`)  
Sets `z` to a lower 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 \(\exp(\log(1 + 2^{kn})/n)2^{-k}\), where \(k\) is chosen so that \(2^{kn} \approx 2^{30}\).

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_log`(`mag_t z`, `const mag_t x`)  
Sets `z` to an upper bound for `\log(\max(1,x))`.

void `mag_log_lower`(`mag_t z`, `const mag_t x`)  
Sets `z` to a lower bound for `\log(\max(1,x))`.

void `mag_neg_log`(`mag_t z`, `const mag_t x`)  
Sets `z` to an upper bound for `\log(\min(1,x))`, i.e. an upper bound for `|\log(x)|` for `x <= 1`.

void `mag_neg_log_lower`(`mag_t z`, `const mag_t x`)  
Sets `z` to a lower bound for `\log(\min(1,x))`, i.e. a lower bound for `|\log(x)|` for `x <= 1`.

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

void `mag_exp_lower`(`mag_t z`, `const mag_t x`)  
Sets `z` to a lower bound for `\exp(x)`.
void **mag_expinv** (*mag_t* *z*, *const* *mag_t* *x*)

Sets *z* to an upper bound for \(\exp(-x)\).

void **mag_expinv_lower** (*mag_t* *z*, *const* *mag_t* *x*)

Sets *z* to a lower bound for \(\exp(-x)\).

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_exptail** (*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 \(\sum_{k=N}^{\infty} x^k\).

**Special functions**

void **mag_const_pi** (*mag_t* *res*)

void **mag_const_pi_lower** (*mag_t* *res*)

Sets *res* to an upper (respectively lower) bound for \(\pi\).

void **mag_atan** (*mag_t* *res*, *const* *mag_t* *x*)

void **mag_atan_lower** (*mag_t* *res*, *const* *mag_t* *x*)

Sets *res* to an upper (respectively lower) bound for \(\atan(x)\).

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_bin_uiui** (*mag_t* *res*, *ulong* *n*, *ulong* *k*)

Sets *res* to an upper bound for the binomial coefficient \(\binom{n}{k}\).

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*, *ulong* *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 \(s' \leq \text{Re}(s)\).

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

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

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

\[
\sum_{k=N}^{\infty} T(k) \leq T(N) \sum_{k=0}^{\infty} U(N)^k = \frac{T(N)}{1 - U(N)}.
\]

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

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

and

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

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

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

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

**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 $1/2 \leq |m| < 1$.

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.

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

\texttt{arf_rnd_t}
\begin{itemize}
  \item Specifies the rounding mode for the result of an approximate operation.
\end{itemize}

\texttt{ARF_RND_DOWN}
\begin{itemize}
  \item Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards zero.
\end{itemize}

\texttt{ARF_RND_UP}
\begin{itemize}
  \item Specifies that the result of an operation should be rounded to the nearest representable number in the direction away from zero.
\end{itemize}

\texttt{ARF_RND_FLOOR}
\begin{itemize}
  \item Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards minus infinity.
\end{itemize}

\texttt{ARF_RND_CEIL}
\begin{itemize}
  \item Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards plus infinity.
\end{itemize}

\texttt{ARF_RND_NEAR}
\begin{itemize}
  \item 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.
\end{itemize}

\texttt{ARF_PREC_EXACT}
\begin{itemize}
  \item If passed as the precision parameter to a function, indicates that no rounding is to be performed.
\end{itemize}

\textbf{Warning}: use of this value is unsafe in general. It must only be passed as input under the following two conditions:

\begin{itemize}
  \item The operation in question can inherently be viewed as an exact operation in $\mathbb{Z}[\frac{1}{4}]$ 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.
  \item 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.
\end{itemize}

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.

\section*{Memory management}

\begin{description}
  \item [\texttt{arf_init}]\texttt{(arf_t x)} \begin{itemize} \item Initializes the variable \texttt{x} for use. Its value is set to zero. \end{itemize} \\
  \item [\texttt{arf_clear}]\texttt{(arf_t x)} \begin{itemize} \item Clears the variable \texttt{x}, freeing or recycling its allocated memory. \end{itemize} \\
  \item [\texttt{slong}]\texttt{arf_allocated_bytes(const arf_t x)} \begin{itemize} \item Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add \texttt{sizeof(arf_struct)} to get the size of the object as a whole. \end{itemize}
\end{description}

\section*{Special values}

\begin{description}
  \item [\texttt{arf_zero}]\texttt{(arf_t x)} \\
  \item [\texttt{arf_one}]\texttt{(arf_t x)} \\
  \item [\texttt{arf_pos_inf}]\texttt{(arf_t x)}
\end{description}
void arf_neg_inf(arf_t x)
void arf_nan(arf_t x)
    Sets x respectively to 0, 1, +∞, −∞, 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, +∞, −∞, NaN.

int arf_is_inf(const arf_t x)
    Returns nonzero iff x equals either +∞ or −∞.

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, +∞, −∞, NaN.

int arf_is_special(const arf_t x)
    Returns nonzero iff x is one of the special values 0, +∞, −∞, 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 +∞, −∞, NaN. (Note that this is not equivalent to the negation of arf_is_inf().)

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)
int arf_set_round_fmpz_2exp(arf_t y, const fmpz_t x, const fmpz_t e, slong prec, arf_rnd_t rnd)
void arf_set_fmpz_2exp(arf_t y, const fmpz_t m, const fmpz_t e)
void arf_set_fmpz_2exp_fmpz(arf_t y, const fmpz_t m, const fmpz_t e, const fmpz_t x, const fmpz_t e, slong prec, arf_rnd_t rnd)

Sets $y$ to $m \times 2^e$. 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_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_fmpfr(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.

int arf_get_si(slong x, 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(const arf_t x, arf_rnd_t rnd)

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.

Chapter 3. Floating-point numbers
Comparisons and bounds

```c
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)
int arf_cmp_si(const arf_t x, slong y)
int arf_cmp_ui(const arf_t x, ulong y)
int arf_cmp_d(const arf_t x, double 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_d(const arf_t x, double 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 $|x| \leq 2^b$. 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.
```

Magnitude functions

```c
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\((mag\_t z, const \ arf\_t y, \ \ \ \ \ \ \ \ \ \ \ arf\_t y, slong \ prec)\)
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\((mag\_t z, const \ arf\_t x, const \ arf\_t y, \ \ \ \ \ \ \ \ \ \ \ slong \ prec)\)
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.

### 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 $x$.

### 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.

### 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.
Addition and multiplication

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

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

int arf_neg_round(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_mul2exp_si(const arf_t x, slong e)
void arf_mul2exp_fmpz(const arf_t x, fmpz_t e)

int arf_mul(const arf_t x, const arf_t y, const arf_t z)
Sets $y = x \cdot z$, rounded to $prec$ bits in the direction specified by $rnd$, returning nonzero iff the operation is inexact.

int arf_mul2exp(const arf_t x, const arf_t y, slong e)
Sets $y = x \cdot 2^e$ exactly.

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

int arf_mul2exp_mpz(const arf_t x, const mpz_t y, slong prec, arf_rnd_t rnd)
Sets $z = x \cdot 2^e$, rounded to $prec$ bits in the direction specified by $rnd$, returning nonzero iff the operation is inexact.

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

int arf_add_mpz(const arf_t x, const mpz_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_sub(const arf_t x, const arf_t y, const arf_t z)
Sets $z = x - y$, rounded to $prec$ bits in the direction specified by $rnd$, returning nonzero iff the operation is inexact.

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

void arf_addmul(const arf_t x, const arf_t y, const arf_t z)
Returns $z = xy$, rounding to $prec$ bits in the direction specified by $rnd$.

void arf_addmul_mpz(const arf_t x, const mpz_t y, slong prec, arf_rnd_t rnd)

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

int arf_sosq(arf_t z, const arf_t x, const arf_t y, slong prec, arf_rnd_t rnd)
    Sets $z = x^2 + y^2$, rounded to prec bits in the direction specified by rnd, returning nonzero iff the operation is inexact.

### 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.

### 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.

### 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 \( \text{prec} \) bits in the direction specified by \( \text{rnd} \), 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 \( k \).

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_mulFallback(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 \( \text{prec} \) bits in the direction specified by \( \text{rnd} \). 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 \( \text{arf_complex_mul()} \) (with \( c = a, b = d \)), but is faster.

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 \( \text{exp} \), 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 \( \text{negative} \) is set, rounding \( z \) to \( \text{prec} \) bits in the direction \( \text{rnd} \) 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 \( \text{slong} \), 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 \( \text{sgnbit} \) is 1, rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \). There are no assumptions on \( x \).

int _arf_set_round_uiu(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 \( \text{sgnbit} \) is 1, rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \). Requires that not both \( hi \) and \( lo \) are zero. Writes the exponent shift to \( \text{fix} \) 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 \( \text{sgnbit} \) is 1, rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \). 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 \( \text{exp_shift} \). This method does not write the exponent of \( z \) directly. Requires that \( z \) does not point to the limbs of \( z \).
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.

**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 \([\text{NaN} \pm c]\), 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 \([\text{NaN} \pm \infty]\) while preserving mathematical correctness (this is currently not done automatically by the library).

## Types, macros and constants

### \texttt{arb\_struct}

#### \texttt{arb\_t}

An \texttt{arb\_struct} consists of an \texttt{arf\_struct} (the midpoint) and a \texttt{mag\_struct} (the radius). An \texttt{arb\_t} is defined as an array of length one of type \texttt{arb\_struct}, permitting an \texttt{arb\_t} to be passed by reference.

### \texttt{arb\_ptr}

Alias for \texttt{arb\_struct *}, used for vectors of numbers.

### \texttt{arb\_srcptr}

Alias for \texttt{const arb\_struct *}, used for vectors of numbers when passed as constant input to functions.

#### \texttt{arb\_midref(x)}

Macro returning a pointer to the midpoint of \(x\) as an \texttt{arf\_t}.

#### \texttt{arb\_radref(x)}

Macro returning a pointer to the radius of \(x\) as a \texttt{mag\_t}.

## Memory management

### \texttt{void arb\_init(arb\_t x)}

Initializes the variable \(x\) for use. Its midpoint and radius are both set to zero.

### \texttt{void arb\_clear(arb\_t x)}

Clears the variable \(x\), freeing or recycling its allocated memory.

### \texttt{arb\_ptr \_arb\_vec\_init(slong n)}

Returns a pointer to an array of \(n\) initialized \texttt{arb\_struct} entries.

### \texttt{void \_arb\_vec\_clear(arb\_ptr v, slong n)}

Clears an array of \(n\) initialized \texttt{arb\_struct} entries.

### \texttt{void arb\_swap(arb\_t x, arb\_t y)}

Swaps \(x\) and \(y\) efficiently.

### \texttt{slong arb\_allocated\_bytes(const arb\_t x)}

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add \texttt{sizeof(arb\_struct)} to get the size of the object as a whole.

### \texttt{slong \_arb\_vec\_allocated\_bytes(arb\_srcptr vec, slong len)}

Returns the total number of bytes allocated for this vector, i.e. the space taken up by the vector itself plus the sum of the internal heap allocation sizes for all its member elements.

### \texttt{double \_arb\_vec\_estimate\_allocated\_bytes(slong len, slong prec)}

Estimates the number of bytes that need to be allocated for a vector of \(len\) elements with \(prec\) bits of precision, including the space for internal limb data. This function returns a \texttt{double} to avoid overflow issues when both \(len\) and \(prec\) are large.

This is only an approximation of the physical memory that will be used by an actual vector. In practice, the space varies with the content of the numbers; for example, zeros and small integers require no internal heap allocation even if the precision is huge. The estimate assumes that exponents will not be bignums. The actual amount may also be higher or lower due to overhead in the memory allocator or overcommitment by the operating system.
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)
void arb_set_fmpz_2exp(arb_t y, const fmpz_t x, const fmpz_t 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)
void arb_set_round_fmpz_2exp(arb_t y, const fmpz_t x, const fmpz_t e, slong prec)
void arb_set_fmpq(arb_t y, const fmpq_t x, slong prec)
int arb_set_str(arb_t res, const char * inp, slong prec)

Sets y to the value of x without rounding.
Sets y to x \cdot 2^e.
Sets y to the value of x, rounded to prec bits.
Sets y to x \cdot 2^e, rounded to prec bits.
Sets y to the rational number x, rounded to prec bits.
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 “-3.14159e-1”, and may also consist of two such literals separated by the symbol “+/-” and optionally enclosed in brackets, e.g. “[3.25 +/-.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.

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).

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 \([\text{NaN} \pm \infty]\), representing an indeterminate result.

Input and output

The arb_print... functions print to standard output, while arb_fprint... functions print to the stream file.

void arb_print(const arb_t x)

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

void arb_printd(const arb_t x, slong digits)

void arb_fprintd(FILE * file, 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)

void arb_fprintn(FILE * file, const arb_t x, slong digits, ulong flags)
    Prints a nice decimal representation of x. By default, the output shows the midpoint with a
    guaranteed error of at most one unit in the last decimal place. In addition, an explicit error bound
    is printed so that the displayed decimal interval is guaranteed to enclose x. See arb_get_str() for
details.

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.

## 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 \( r \) 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 \( \text{err} \) 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 \( \text{arb_overlaps()} \), 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_nonnegative_part** (arb_t res, const arb_t x)

Sets \( \text{res} \) to the intersection of \( x \) with \([0, \infty]\). If \( x \) is nonnegative, an exact copy is made. If \( x \) is finite and contains negative numbers, an interval of the form \([r/2 \pm r/2]\) is produced, which certainly contains no negative points. In the special case when \( x \) is strictly negative, \( \text{res} \) is set to zero.

**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 \( \text{prec} \) 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 \( \text{prec} \) 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 \( \text{prec} \) 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 \( \text{prec} \) 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_mag(arb_t x, const mag_t a, const mag_t b, slong prec)
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 $[a, b]$.

void arb_set_interval_neg_pos_mag(arb_t x, const mag_t a, const mag_t b, slong prec)
Sets x to a ball containing the interval $[-a, b]$.

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, constarb_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)

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.

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_is_int_2exp_si(const arb_t x, slong e)
Returns nonzero iff x exactly equals $n2^e$ for some integer n.

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 \( t \text{(op)} u \) 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.

Arithmetic

void `arb_neg(const arb_t x)`
void `arb_neg_round(const arb_t x, slong prec)`
Sets \( y \) to the negation of \( x \).
void \texttt{arb\_abs}(arb\_t y, const arb\_t x)
\hspace*{1em}Sets \(y\) to the absolute value of \(x\). No attempt is made to improve the interval represented by \(x\) if it contains zero.

void \texttt{arb\_sgn}(arb\_t y, const arb\_t x)
\hspace*{1em}Sets \(y\) to the sign function of \(x\). The result is \([0 \pm 1]\) if \(x\) contains both zero and nonzero numbers.

void \texttt{arb\_min}(arb\_t z, const arb\_t x, const arb\_t y, slong prec)

void \texttt{arb\_max}(arb\_t z, const arb\_t x, const arb\_t y, slong prec)
\hspace*{1em}Sets \(z\) respectively to the minimum and the maximum of \(x\) and \(y\).

void \texttt{arb\_add}(arb\_t z, const arb\_t x, const arb\_t y, slong prec)

void \texttt{arb\_add\_arf}(arb\_t z, const arb\_t x, const arf\_t y, slong prec)

void \texttt{arb\_add\_si}(arb\_t z, const arb\_t x, slong y, slong prec)

void \texttt{arb\_add\_fmpz}(arb\_t z, const arb\_t x, const fmpz\_t y, slong prec)
\hspace*{1em}Sets \(z = x + y\), rounded to \(prec\) bits. The precision can be \texttt{ARF\_PREC\_EXACT} provided that the result fits in memory.

void \texttt{arb\_add\_fmpz\_2exp}(arb\_t z, const arb\_t x, const fmpz\_t m, const fmpz\_t e, slong prec)
\hspace*{1em}Sets \(z = x + m \cdot 2^e\), rounded to \(prec\) bits. The precision can be \texttt{ARF\_PREC\_EXACT} provided that the result fits in memory.

void \texttt{arb\_sub}(arb\_t z, const arb\_t x, const arb\_t y, slong prec)

void \texttt{arb\_sub\_arf}(arb\_t z, const arb\_t x, const arf\_t y, slong prec)

void \texttt{arb\_sub\_ui}(arb\_t z, const arb\_t x, ulong y, slong prec)

void \texttt{arb\_sub\_si}(arb\_t z, const arb\_t x, slong y, slong prec)

void \texttt{arb\_sub\_fmpz}(arb\_t z, const arb\_t x, const fmpz\_t y, slong prec)
\hspace*{1em}Sets \(z = x - y\), rounded to \(prec\) bits. The precision can be \texttt{ARF\_PREC\_EXACT} provided that the result fits in memory.

void \texttt{arb\_mul}(arb\_t z, const arb\_t x, const arb\_t y, slong prec)

void \texttt{arb\_mul\_arf}(arb\_t z, const arb\_t x, const arf\_t y, slong prec)

void \texttt{arb\_mul\_si}(arb\_t z, const arb\_t x, slong y, slong prec)

void \texttt{arb\_mul\_ui}(arb\_t z, const arb\_t x, ulong y, slong prec)

void \texttt{arb\_mul\_fmpz}(arb\_t z, const arb\_t x, const fmpz\_t y, slong prec)
\hspace*{1em}Sets \(z = x \cdot y\), rounded to \(prec\) bits. The precision can be \texttt{ARF\_PREC\_EXACT} provided that the result fits in memory.

void \texttt{arb\_mul\_2exp\_si}(arb\_t y, const arb\_t x, slong e)

void \texttt{arb\_mul\_2exp\_fmpz}(arb\_t y, const arb\_t x, const fmpz\_t e)
\hspace*{1em}Sets \(y\) to \(x\) multiplied by \(2^e\).

void \texttt{arb\_addmul}(arb\_t z, const arb\_t x, const arb\_t y, slong prec)

void \texttt{arb\_addmul\_arf}(arb\_t z, const arb\_t x, const arf\_t y, slong prec)

void \texttt{arb\_addmul\_si}(arb\_t z, const arb\_t x, slong y, slong prec)

void \texttt{arb\_addmul\_ui}(arb\_t z, const arb\_t x, ulong y, slong prec)

void \texttt{arb\_addmul\_fmpz}(arb\_t z, const arb\_t x, const fmpz\_t y, slong prec)
\hspace*{1em}Sets \(z = z + x \cdot y\), rounded to \(prec\) bits. The precision can be \texttt{ARF\_PREC\_EXACT} provided that the result fits in memory.

void \texttt{arb\_submul}(arb\_t z, const arb\_t x, const arb\_t y, slong prec)

void \texttt{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, 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

\[
\frac{x}{y} = \frac{x + \xi_1 a}{y + \xi_2 b} \leq \frac{|x b| + |y c|}{|y|(|y| - b)}
\]

where \(-1 \leq \xi_1, \xi_2 \leq 1\), 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.

**Powers and roots**

void arb_sqr(arb_t z, const arb_t x, slong prec)
void arb_sqr_arf(arb_t z, const arf_t x, slong prec)
void arb_sqr_fmpz(arb_t z, const fmpz_t x, slong prec)
void arb_sqr_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_sqrtppos(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_sqrtlpm1(arb_t z, const arb_t x, slong prec)

Sets \( z = \sqrt{1+x} - 1 \), computed accurately when \( x \approx 0 \).
void \texttt{arb_root\_ui} (\texttt{arb\_t} z, \texttt{const arb\_t} x, \texttt{ulong} k, \texttt{slong} prec)

Sets \( z \) to the \( k \)-th root of \( x \), rounded to \( \text{prec} \) bits. This function selects between different algorithms. For large \( k \), it evaluates \( \exp(\log(x)/k) \). For small \( k \), it uses \texttt{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

\[
m^{1/k} - (m-r)^{1/k} = m^{1/k}[1 - (1 - r/m)^{1/k}]
\]

\[
= m^{1/k}[1 - \exp(\log(1 - r/m)/k)]
\]

\[
\leq m^{1/k}\min(1, -\log(1 - r/m)/k)
\]

\[
= m^{1/k}\min(1, \log(1 + r/(m - r))/k).
\]

This is evaluated using \texttt{mag\_log1p()}.

void \texttt{arb\_log\_ui} (\texttt{arb\_t} y, \texttt{const arb\_t} x, \texttt{ulong} k, \texttt{slong} prec)

Alias for \texttt{arb\_root\_ui()}, provided for backwards compatibility.

void \texttt{arb\_sqr} (\texttt{arb\_t} y, \texttt{const arb\_t} x, \texttt{slong} prec)

Sets \( y \) to be the square of \( x \).

void \texttt{arb\_pow\_fmpz\_binexp} (\texttt{arb\_t} y, \texttt{const arb\_t} b, \texttt{const fmpz\_t} e, \texttt{slong} prec)

void \texttt{arb\_pow\_fmpz} (\texttt{arb\_t} y, \texttt{const arb\_t} b, \texttt{const fmpz\_t} e, \texttt{slong} prec)

void \texttt{arb\_pow\_ui} (\texttt{arb\_t} y, \texttt{const arb\_t} b, \texttt{ulong} e, \texttt{slong} prec)

void \texttt{arb\_ui\_pow\_ui} (\texttt{arb\_t} y, \texttt{ulong} b, \texttt{ulong} e, \texttt{slong} prec)

void \texttt{arb\_si\_pow\_ui} (\texttt{arb\_t} y, \texttt{slong} b, \texttt{ulong} e, \texttt{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 \texttt{ARF\_PREC\_EXACT}.

Note that these functions can get slow if the exponent is extremely large (in such cases \texttt{arb\_pow()} may be superior).

void \texttt{arb\_pow\_fmpq} (\texttt{arb\_t} y, \texttt{const arb\_t} x, \texttt{const fmpq\_t} a, \texttt{slong} prec)

Sets \( y = b^a \), 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 \texttt{arb\_pow()} may be superior).

void \texttt{arb\_pow} (\texttt{arb\_t} z, \texttt{const arb\_t} x, \texttt{const arb\_t} y, \texttt{slong} prec)

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

Exponentials and logarithms

void \texttt{arb\_log\_ui} (\texttt{arb\_t} z, \texttt{ulong} x, \texttt{slong} prec)

void \texttt{arb\_log\_fmpz} (\texttt{arb\_t} z, \texttt{const fmpz\_t} x, \texttt{slong} prec)

void \texttt{arb\_log\_arf} (\texttt{arb\_t} z, \texttt{const arf\_t} x, \texttt{slong} prec)

void \texttt{arb\_log} (\texttt{arb\_t} z, \texttt{const arb\_t} x, \texttt{slong} prec)

Sets \( z = \log(x) \).

At low to medium precision (up to about 4096 bits), \texttt{arb\_log\_arf()} uses table-based argument reduction and fast Taylor series evaluation via \texttt{arb\_atan\_taylor\_rs()}. At high precision, it falls back to MPFR. The function \texttt{arb\_log()} simply calls \texttt{arb\_log\_arf()} with the midpoint as input, and separately adds the propagated error.

void \texttt{arb\_log\_ui\_from\_prev} (\texttt{arb\_t} log\_k1, \texttt{ulong} k1, \texttt{arb\_t} log\_k0, \texttt{ulong} k0, \texttt{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\text{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.

```c
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_log_hypot(arb_t res, const arb_t x, const arb_t y, slong prec)
  Sets res to $\log(\sqrt{x^2 + y^2})$.

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 $x = m + r$, the error is largest at $m + r$, and we have $\exp(m + r) - \exp(m) = \exp(m)(\exp(r) - 1) \leq r\exp(m + r)$.

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, const 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.

### Trigonometric functions

```c
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 $|\sin(m \pm r) - \sin(m)| \leq \min(r, 2)$.

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 $\pi$ 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_sec(arb_t res, const arb_t x, slong prec)
  Computes $\sec(x) = 1/\cos(x)$.
void \texttt{arb\_csc}(\texttt{arb\_t res}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} Computes $\csc(x) = 1/\sin(x)$.

void \texttt{arb\_sinc}(\texttt{arb\_t z}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} Sets $z = \sin(x) = \sin(x)/x$.

void \texttt{arb\_sinc\_pi}(\texttt{arb\_t z}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} Sets $z = \sin(\pi x) = \sin(\pi x)/(\pi x)$.

### Inverse trigonometric functions

void \texttt{arb\_atan\_arf}(\texttt{arb\_t z}, \texttt{const arf\_t x}, \texttt{slong prec})
\hspace{1em} Computes $\arctan(x)$.

void \texttt{arb\_atan}(\texttt{arb\_t z}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} Sets $z = \arctan(x)$.

At low to medium precision (up to about 4096 bits), \texttt{arb\_atan\_arf()} uses table-based argument reduction and fast Taylor series evaluation via \_\texttt{arb\_atan\_taylor\_rs}(). At high precision, it falls back to MPFR. The function \texttt{arb\_atan()} simply calls \texttt{arb\_atan\_arf()} with the midpoint as input, and separately adds the propagated error.

The function \texttt{arb\_atan\_arf()} uses lookup tables if possible, and otherwise falls back to \texttt{arb\_atan\_arf\_bb()}.

void \texttt{arb\_atan2}(\texttt{arb\_t z}, \texttt{const arb\_t x}, \texttt{const arb\_t y}, \texttt{slong prec})
\hspace{1em} Sets $z$ to an the argument (phase) of the complex number $a + bi$, with the branch cut discontinuity on $(-\infty, 0]$. We define \texttt{atan2}(0,0) = 0, and for $a < 0$, \texttt{atan2}(0,a) = \pi$.

void \texttt{arb\_asin}(\texttt{arb\_t z}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} Sets $z = \arcsin(x) = \arctan(x/\sqrt{1-x^2})$. If $x$ is not contained in the domain $[-1,1]$, the result is an indeterminate interval.

void \texttt{arb\_acos}(\texttt{arb\_t z}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} Sets $z = \arccos(x) = \pi/2 - \arcsin(x)$. If $x$ is not contained in the domain $[-1,1]$, the result is an indeterminate interval.

### Hyperbolic functions

void \texttt{arb\_sinh}(\texttt{arb\_t s}, \texttt{const arb\_t x}, \texttt{slong prec})

void \texttt{arb\_cosh}(\texttt{arb\_t c}, \texttt{const arb\_t x}, \texttt{slong prec})

void \texttt{arb\_sinh\_cosh}(\texttt{arb\_t s}, \texttt{arb\_t c}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} 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 \texttt{arb\_expm1()} to avoid loss of accuracy. Otherwise evaluates $(e^x \pm e^{-x})/2$.

void \texttt{arb\_tanh}(\texttt{arb\_t y}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} Sets $y = \tanh(x) = \sinh(x)/\cosh(x)$, evaluated via \texttt{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 \texttt{arb\_coth}(\texttt{arb\_t y}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} Sets $y = \coth(x) = \cosh(x)/\sinh(x)$, evaluated using the same strategy as \texttt{arb\_tanh}().

void \texttt{arb\_sech}(\texttt{arb\_t res}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} Computes $\sech(x) = 1/\cosh(x)$.

void \texttt{arb\_csch}(\texttt{arb\_t res}, \texttt{const arb\_t x}, \texttt{slong prec})
\hspace{1em} Computes $\csch(x) = 1/\sinh(x)$.
Inverse hyperbolic functions

```c
void arb_atanh(arb_t z, const arb_t x, slong prec)
Sets z = atanh(x).

void arb_asinh(arb_t z, const arb_t x, slong prec)
Sets z = asinh(x).

void arb_acosh(arb_t z, const arb_t x, slong prec)
Sets z = acosh(x). If x < 1, the result is an indeterminate interval.
```

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.

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

void arb_const_sqrt_pi(arb_t z, slong prec)
Computes √𝜋.

void arb_const_log_sqrt2pi(arb_t z, slong prec)
Computes log √2𝜋.

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 𝛾 = limk→∞(Hk − log k) where Hk = 1 + 1/2 + ... + 1/k.

void arb_const_catalan(arb_t z, slong prec)
Computes Catalan’s constant 𝐶 = ∑n=0∞(−1)n/(2n + 1)2.

void arb_const_e(arb_t z, long prec)
Computes e = exp(1).

void arb_const_khinchin(arb_t z, slong prec)
Computes Khinchin’s constant K0.

void arb_const_glaisher(arb_t z, slong prec)
Computes the Glaisher-Kinkelin constant A = exp(1/12 − 𝜁′(−1)).

void arb_const_apery(arb_t z, slong prec)
Computes Apery’s constant 𝜁(3).
```

Lambert W function

```c
void arb_lambertw(arb_t res, const arb_t x, int flags, slong prec)
Computes the Lambert W function, which solves the equation we^w = x.

The Lambert W function has infinitely many complex branches W_k(x), two of which are real on a part of the real line. The principal branch W_0(x) is selected by setting flags to 0, and the W_-1 branch is selected by setting flags to 1. The principal branch is real-valued for x ≥ −1/e (taking values in [−1, +∞)) and the W_-1 branch is real-valued for −1/e ≤ x < 0 and takes values in (−∞, −1). Elsewhere, the Lambert W function is complex and acb_lambertw() should be used.

The implementation first computes a floating-point approximation heuristically and then computes a rigorously certified enclosure around this approximation. Some asymptotic cases are handled specially. The algorithm used to compute the Lambert W function is described in [Joh2017b], which follows the main ideas in [CGHJK1996].
```
Gamma function and factorials

```c
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_fmpz_ui(arb_t z, const fmpz_t x, ulong n, slong prec)
void arb_rising2_ui_bs(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_fmpq_ui(arb_t z, const fmpq_t x, ulong n, slong prec)
void arb_rising2_ui(arb_t z, const arb_t x, const arb_t n, slong prec)
void arb_rising2_ui(arb_t z, const arb_t x, const arb_t n, slong prec)
void arb_rising2_ui(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 \( \text{step} \) parameter for tuning purposes (to use the default step length, pass zero).

```c
void arb_rising2_ui_vi(arb_t z, const arb_t x, ulong n, slong prec)
void arb_rising2_ui_vi(arb_t z, const arb_t x, ulong n, slong prec)
void arb_rising2_ui_vi(arb_t z, const arb_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 \( \text{prec} \), it is more efficient to convert \( x \) to an approximation and use \( \text{arb_rising}() \).

```c
void arb_fac_ui(arb_t z, ulong n, slong prec)
void arb_doublefac_ui(arb_t z, const arb_t x, ulong n, slong prec)
void arb_bin_ui(arb_t z, const arb_t x, ulong n, slong prec)
void arb_bin_uuiui(arb_t z, ulong n, ulong k, slong prec)
void arb_bin_uiuii(arb_t z, const arb_t x, ulong n, ulong k, slong prec)
void arb_gamma_ui(arb_t z, const arb_t x, slong prec)
void arb_gamma_fmpz_ui(arb_t z, const fmpz_t x, ulong n, slong prec)
void arb_gamma_fmpq_ui(arb_t z, const fmpq_t x, ulong n, slong prec)
void arb_gamma_fmpz_ui(arb_t z, const fmpq_t x, ulong n, slong prec)
```

- Computes the factorial \( z = n! \) via the gamma function.
- Computes the factorial \( z = n!! \) via the gamma function.
- Computes the binomial coefficient \( z = \binom{n}{k} \), via the rising factorial as \( \binom{n}{k} = (n-k+1)k! \).
- Computes the gamma function \( z = \Gamma(x) \).
- Computes the logarithmic gamma function \( z = \log \Gamma(x) \). The complex branch structure is assumed, so if \( x \leq 0 \), the result is an indeterminate interval.
- Computes the reciprocal gamma function \( z = 1/\Gamma(x) \), avoiding division by zero at the poles of the gamma function.
- Computes the digamma function \( z = \psi(x) = (\log \Gamma(x))' = \Gamma'(x)/\Gamma(x) \).

Zeta function

```c
void arb_zeta_ui_vec_borwein(arb_ptr z, ulong start, slong num, ulong step, slong prec)
```

- Evaluates \( \zeta(s) \) at \( n \) consecutive integers \( s \) beginning with \( \text{start} \) 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 \( \geq 2 \). 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 \( 2^{\text{prec}} \). 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 \( k^\text{step} \). 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 \textbf{arb\_zeta\_ui\_asymp}(arb\_t x, along s, slong prec)

void \textbf{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 \textbf{arb\_zeta\_ui\_bernoulli}(arb\_t x, ulong s, slong prec)

Computes \( \zeta(s) \) for even s via the corresponding Bernoulli number.

void \textbf{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 \textbf{arb\_zeta\_ui\_vec}(arb\_ptr x, along start, slong num, slong prec)

void \textbf{arb\_zeta\_ui\_vec\_even}(arb\_ptr x, along start, slong num, slong prec)

void \textbf{arb\_zeta\_ui\_vec\_odd}(arb\_ptr x, along 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 \textbf{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 \textbf{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 \textbf{arb\_poly\_zeta\_series}().

void \textbf{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 \textbf{arb\_poly\_zeta\_series}().

\section*{Bernoulli numbers and polynomials}

void \textbf{arb\_bernoulli\_ui}(arb\_t b, ulong n, slong prec)

void \textbf{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 \textbf{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.

```c
void arb_bernoulli_ui_zeta(arb_t b, ulong n, slong prec)
```

Sets \( b \) to the numerical value of \( B_n \) accurate to \( \text{prec} \) 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.

```c
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.

```c
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 \( \text{len} - 1 \), 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.

**Polylogarithms**

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

```c
void arb_polylog_si(arb_t w, slong s, const arb_t z, slong prec)
```

Sets \( w \) to the polylogarithm \( \text{Li}_s(z) \).

**Other special functions**

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

```c
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 \( \text{ARF_PREC_EXACT} \).

```c
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 \( y \).

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

```c
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 \( a = U_n(x) \).

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

```c
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.

```c
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)
```

4.1. arb.h – real numbers 61
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 $\sum_{k=0}^{b-1} k^n/k!$. If mmag is non-NULL, it may be used to indicate that the target error tolerance should be $2^{\text{mmag} - \text{prec}}$.

void arb_bell_fmpz(arb_t res, const fmpz_t n)

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)

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.

### 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 $N \leq 255$, $0 \leq x \leq 1/16$, 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 $N \leq 287$, $0 \leq x \leq 1/16$, 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 yc, 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 $y_c = \sum_{k=0}^{N-1} (-1)^k x^{2k}/(2k)!$. 

---

62 Chapter 4. Real and complex numbers
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 \( N \leq 143, 0 \leq x \leq 1/16 \), and \( x^n \) positive. The input \( x \) and outputs \( y \sin, y \cos \) are fixed-point numbers with \( x^n \) fractional limbs. A bound for the ulp error is written to \( \text{error} \).

If \( \text{sinonly} \) is 1, only the sine is computed; if \( \text{sinonly} \) 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 \( \text{sinonly} \)).

```c
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 \( 0 \leq w < \log(2) \), where \( w \) is a fixed-point number with \( w \) limbs and ulp error \( \text{error} \). 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 \( 0 \leq w < \pi/4 \), where \( w \) is a fixed-point number with \( w \) limbs and ulp error \( \text{error} \). Returns success.

The value of \( q \mod 8 \) is written to \( \text{octant} \). The output variable \( q \) can be NULL, in which case the full value of \( q \) is not stored.
```

```c
long _arb_taylor_bound(slong mag, long prec)
Returns \( n \) such that \( \sum_{k=n}^{\infty} x^k/k! \leq 2^{-\text{mag}} \), assuming \( |x| \leq 2^{\text{mag}} \leq 1/4 \).

void _arb_exp_arf_bb(arb_t z, const arf_t x, long 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.
```

```c
void _arb_exp_sum_bs_simple(fmpz_t T, fmpz_t Q, mp_bitcnt_t * Qexp, const fmpz_t x, mp_bitcnt_t r, long 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, long N)
Computes \( T, Q \), and \( Q\exp \) such that \( T/(Q^Q\exp) = \sum_{k=1}^{N} x/(2^k) \) 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.
```

```c
void _arb_exp_arf_rs_generic(arb_t res, const arf_t x, long prec, int minus_one)
Computes the exponential function using a generic version of the rectangular splitting strategy, intended for intermediate precision.
```

```c
void _arb_atan_sum_bs_simple(fmpz_t T, fmpz_t Q, mp_bitcnt_t * Qexp, const fmpz_t x, mp_bitcnt_t r, long 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, long N)
Computes \( T, Q \), and \( Q\exp \) such that \( T/(Q^Q\exp) = \sum_{k=1}^{N} (-1)^k (x/2^k) \) using binary splitting. Note that the sum is taken to \( N \) inclusive, omits the linear term, and requires a final multiplication by \( x/2^k \) 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 \texttt{arb\_atan\_arf\_bb}(\texttt{arb\_t} z, \texttt{const\ arf\_t} x, \texttt{slong} prec)

Computes the arctangent of \(x\). Initially, the argument-halving formula
\[
\text{atan}(x) = 2 \text{atan}\left(\frac{x}{1 + \sqrt{1 + x^2}}\right)
\]
is applied up to 8 times to get a small argument. Then a version of the bit-burst algorithm is used. The functional equation
\[
\text{atan}(x) = \text{atan}(p/q) + \text{atan}(w), \quad w = \frac{qx - p}{px + q}, \quad p = \lfloor qx \rfloor
\]
is applied repeatedly instead of integrating a differential equation for the arctangent, as this appears to be more efficient.

void \texttt{arb\_sin\_cos\_arf\_generic}(\texttt{arb\_t} s, \texttt{arb\_t} c, \texttt{const\ arf\_t} x, \texttt{slong} prec)

Computes the sine and cosine of \(x\) using a generic strategy. This function gets called internally by the main sin and cos functions when the precision for argument reduction or series evaluation based on lookup tables is exhausted.

This function first performs a cheap test to see if \(|x| < \pi/2 - \varepsilon\). If the test fails, it uses \(\pi\) to reduce the argument to the first octant, and then evaluates the sin and cos functions recursively (this call cannot result in infinite recursion).

If no argument reduction is needed, this function uses a generic version of the rectangular splitting algorithm if the precision is not too high, and otherwise invokes the asymptotically fast bit-burst algorithm.

void \texttt{arb\_sin\_cos\_arf\_bb}(\texttt{arb\_t} s, \texttt{arb\_t} c, \texttt{const\ arf\_t} x, \texttt{slong} prec)

Computes the sine and cosine of \(x\) using the bit-burst algorithm. It is required that \(|x| < \pi/2\) (this is not checked).

\section*{Vector functions}

void \texttt{_arb\_vec\_zero}(\texttt{arb\_ptr} vec, \texttt{slong} n)

Sets all entries in \(vec\) to zero.

int \texttt{_arb\_vec\_is\_zero}(\texttt{arb\_srsrptr} vec, \texttt{slong} len)

Returns nonzero iff all entries in \(x\) are zero.

int \texttt{_arb\_vec\_is\_finite}(\texttt{arb\_srsrptr} x, \texttt{slong} len)

Returns nonzero iff all entries in \(x\) certainly are finite.

void \texttt{_arb\_vec\_set}(\texttt{arb\_ptr} res, \texttt{arb\_srsrptr} vec, \texttt{slong} len)

Sets \(res\) to a copy of \(vec\).

void \texttt{_arb\_vec\_set\_round}(\texttt{arb\_ptr} res, \texttt{arb\_srsrptr} vec, \texttt{slong} len, \texttt{slong} prec)

Sets \(res\) to a copy of \(vec\), rounding each entry to \(prec\) bits.

void \texttt{_arb\_vec\_swap}(\texttt{arb\_ptr} vec1, \texttt{arb\_ptr} vec2, \texttt{slong} len)

Swaps the entries of \(vec1\) and \(vec2\).

void \texttt{_arb\_vec\_neg}(\texttt{arb\_ptr} B, \texttt{arb\_srsrptr} A, \texttt{slong} n)

void \texttt{_arb\_vec\_sub}(\texttt{arb\_ptr} C, \texttt{arb\_srsrptr} A, \texttt{arb\_srsrptr} B, \texttt{slong} n, \texttt{slong} prec)

void \texttt{_arb\_vec\_add}(\texttt{arb\_ptr} C, \texttt{arb\_srsrptr} A, \texttt{arb\_srsrptr} B, \texttt{slong} n, \texttt{slong} prec)

void \texttt{_arb\_vec\_scalar\_mul}(\texttt{arb\_ptr} res, \texttt{arb\_srsrptr} vec, \texttt{slong} len, \texttt{const\ arb\_t} c, \texttt{slong} prec)

void \texttt{_arb\_vec\_scalar\_div}(\texttt{arb\_ptr} res, \texttt{arb\_srsrptr} vec, \texttt{slong} len, \texttt{const\ arb\_t} c, \texttt{slong} prec)

void \texttt{_arb\_vec\_scalar\_mul\_fmpz}(\texttt{arb\_ptr} res, \texttt{arb\_srsrptr} vec, \texttt{slong} len, \texttt{const\ fmpz\_t} c, \texttt{slong} prec)

void \texttt{_arb\_vec\_scalar\_mul\_2exp\_si}(\texttt{arb\_ptr} res, \texttt{arb\_srsrptr} src, \texttt{slong} len, \texttt{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 len1, 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, x², ..., 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.

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.

Types, macros and constants

acb_struct

acb_t
    An acb_struct consists of a pair of arb_structs. 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_realfref(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.
Memory management

void \texttt{acb\_init(acb\_t x)}
\begin{itemize}
\item Initializes the variable \texttt{x} for use, and sets its value to zero.
\end{itemize}

void \texttt{acb\_clear(acb\_t x)}
\begin{itemize}
\item Clears the variable \texttt{x}, freeing or recycling its allocated memory.
\end{itemize}

\texttt{acb\_ptr \_acb\_vec\_init(slong n)}
\begin{itemize}
\item Returns a pointer to an array of \texttt{n} initialized \texttt{acb\_structs}.
\end{itemize}

void \texttt{\_acb\_vec\_clear(acb\_ptr v, slong n)}
\begin{itemize}
\item Clears an array of \texttt{n} initialized \texttt{acb\_structs}.
\end{itemize}

\texttt{slong \acb\_allocated\_bytes(const acb\_t x)}
\begin{itemize}
\item Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add \texttt{sizeof(acb\_struct)} to get the size of the object as a whole.
\end{itemize}

\texttt{slong \_acb\_vec\_allocated\_bytes(acb\_srcptr vec, slong len)}
\begin{itemize}
\item Returns the total number of bytes allocated for this vector, i.e. the space taken up by the vector itself plus the sum of the internal heap allocation sizes for all its member elements.
\end{itemize}

\texttt{double \_acb\_vec\_estimate\_allocated\_bytes(slong len, slong prec)}
\begin{itemize}
\item Estimates the number of bytes that need to be allocated for a vector of \texttt{len} elements with \texttt{prec} bits of precision, including the space for internal limb data. See comments for \texttt{\_arb\_vec\_estimate\_allocated\_bytes()}.\end{itemize}

Basic manipulation

void \texttt{acb\_zero(acb\_t z)}
\begin{itemize}
\item Sets \texttt{z} to 0.
\end{itemize}

void \texttt{acb\_one(acb\_t z)}
\begin{itemize}
\item Sets \texttt{z} to 1.
\end{itemize}

void \texttt{acb\_onei(acb\_t z)}
\begin{itemize}
\item Sets \texttt{z} to \sqrt{-1}.
\end{itemize}

void \texttt{acb\_set(acb\_t z, const acb\_t x)}
\begin{itemize}
\item Sets \texttt{z} to the value of \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_ui(acb\_t z, slong x)}
\begin{itemize}
\item Sets \texttt{z} to \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_si(acb\_t z, slong x)}
\begin{itemize}
\item Sets \texttt{z} to \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_d(acb\_t z, double x)}
\begin{itemize}
\item Sets \texttt{z} to \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_fmpz(acb\_t z, const fmpz\_t x)}
\begin{itemize}
\item Sets \texttt{z} to \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_arb(acb\_t z, const arb\_t c)}
\begin{itemize}
\item Sets \texttt{z} to \texttt{c}.
\end{itemize}

void \texttt{acb\_set\_si\_si(acb\_t z, slong x, slong y)}
\begin{itemize}
\item Sets \texttt{z} to the value of \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_d\_d(acb\_t z, double x, double y)}
\begin{itemize}
\item Sets \texttt{z} to the value of \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_fmpz\_fmpz(acb\_t z, const fmpz\_t x, const fmpz\_t y)}
\begin{itemize}
\item Sets \texttt{z} to the value of \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_arb\_arb(acb\_t z, const arb\_t x, const arb\_t y)}
\begin{itemize}
\item Sets \texttt{z} to the value of \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_fmpq(acb\_t z, const fmpq\_t x, slong prec)}
\begin{itemize}
\item Sets \texttt{z} to the value of \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_round(acb\_t z, const acb\_t x, slong prec)}
\begin{itemize}
\item Sets \texttt{z} to \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_round\_fmpz(acb\_t z, const fmpz\_t x, slong prec)}
\begin{itemize}
\item Sets \texttt{z} to \texttt{x}.
\end{itemize}

void \texttt{acb\_set\_round\_arb(acb\_t z, const arb\_t x, slong prec)}
\begin{itemize}
\item Sets \texttt{z} to \texttt{x}.
\end{itemize}

void \texttt{acb\_swap(acb\_t z, acb\_t x)}
\begin{itemize}
\item Swaps \texttt{z} and \texttt{x} efficiently.
\end{itemize}
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.

**Input and output**

The acb_print... functions print to standard output, while acb_fprint... functions print to the stream file.

void acb_print(const acb_t x)
void acb_fprint(FILE * file, const acb_t x)

Prints the internal representation of x.

void acb_printd(const acb_t x, slong digits)
void acb_fprintd(FILE * file, const acb_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 acb_printn(const acb_t x, slong digits, ulong flags)
void acb_fprintn(FILE * file, const acb_t x, slong digits, ulong flags)

Prints a nice decimal representation of x, using the format of arb_get_str() (or the corresponding arb_printn()) for the real and imaginary parts.

By default, the output shows the midpoint of both the real and imaginary parts with a guaranteed error of at most one unit in the last decimal place. In addition, explicit error bounds are printed so that the displayed decimal interval is guaranteed to enclose x.

Any flags understood by arb_get_str() can be passed via flags to control the format of the real and imaginary parts.

**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.

**Precision and comparisons**

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.

int acb_is_int_2exp_si(const acb_t x, slong e)
    Returns nonzero iff \( z \) exactly equals \( n2^e \) for some integer \( n \).

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 \( \text{acb_overlaps()} \) or \( \text{acb_contains()} \), 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 \( \text{arb_eq()} \) 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 \( \text{arb_ne()} \) 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_union(acb_t z, const acb_t x, const acb_t y, slong prec)
    Sets \( z \) to a complex interval containing both \( x \) and \( y \).

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 \( \text{prec} \) 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 \( \text{prec} \) 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_fmpz(const acb_t x, const fmpz_t y)
int acb_contains_fmpq(const acb_t x, const fmpq_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 \( \text{arb_rel_error_bits()} \) 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 radiiuses 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 [NaN ±∞] + [NaN ±∞]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.

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 (−∞, 0], with the special value arg(0) = 0, and arg(a + 0𝑖) = π for a < 0. Equivalently, if z = a + bi, the argument is given by 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(arb_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, csgn(z) = z/√z̅² except that the value is 0 when z is exactly zero.

Arithmetic

void acb_neg(acb_t z, const acb_t x)

void acb_neg_round(acb_t z, const acb_t x, slong prec)
    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 \texttt{acb\_add}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ acb\_t \ y, \text{slong} \ \text{prec})
Sets \(z\) to the sum of \(x\) and \(y\).

void \texttt{acb\_sub\_ui}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{ulong} \ y, \text{slong} \ \text{prec})

void \texttt{acb\_sub\_si}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{slong} \ y, \text{slong} \ \text{prec})

void \texttt{acb\_sub\_fmpz}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ fmzp\_t \ y, \text{slong} \ \text{prec})

void \texttt{acb\_sub\_arb}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ arb\_t \ y, \text{slong} \ \text{prec})

void \texttt{acb\_sub}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ acb\_t \ y, \text{slong} \ \text{prec})
Sets \(z\) to the difference of \(x\) and \(y\).

void \texttt{acb\_mul\_onei}(acb\_t \ z, \text{const} \ acb\_t \ x)
Sets \(z\) to \(x\) multiplied by the imaginary unit.

void \texttt{acb\_div\_onei}(acb\_t \ z, \text{const} \ acb\_t \ x)
Sets \(z\) to \(x\) divided by the imaginary unit.

void \texttt{acb\_mul\_ui}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{ulong} \ y, \text{slong} \ \text{prec})

void \texttt{acb\_mul\_si}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{slong} \ y, \text{slong} \ \text{prec})

void \texttt{acb\_mul\_fmpz}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ fmzp\_t \ y, \text{slong} \ \text{prec})

void \texttt{acb\_mul\_arb}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ arb\_t \ y, \text{slong} \ \text{prec})
Sets \(z\) to the product of \(x\) and \(y\).

void \texttt{acb\_mul}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ acb\_t \ y, \text{slong} \ \text{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 \texttt{acb\_mul\_2exp\_si}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{slong} \ \text{e})

void \texttt{acb\_mul\_2exp\_fmpz}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ fmzp\_t \ \text{e})
Sets \(z\) to \(x\) multiplied by \(2^e\), without rounding.

void \texttt{acb\_sqr}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{slong} \ \text{prec})
Sets \(z\) to \(x\) squared.

void \texttt{acb\_cube}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{slong} \ \text{prec})
Sets \(z\) to \(x\) cubed, computed efficiently using two real squarings, two real multiplications, and scalar operations.

void \texttt{acb\_addmul}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ acb\_t \ y, \text{slong} \ \text{prec})

void \texttt{acb\_addmul\_ui}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{ulong} \ y, \text{slong} \ \text{prec})

void \texttt{acb\_addmul\_si}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{slong} \ y, \text{slong} \ \text{prec})

void \texttt{acb\_addmul\_fmpz}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ fmzp\_t \ y, \text{slong} \ \text{prec})

void \texttt{acb\_addmul\_arb}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \arb\_t \ y, \text{slong} \ \text{prec})
Sets \(z\) to \(z\) plus the product of \(x\) and \(y\).

void \texttt{acb\_submul}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ acb\_t \ y, \text{slong} \ \text{prec})

void \texttt{acb\_submul\_ui}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{ulong} \ y, \text{slong} \ \text{prec})

void \texttt{acb\_submul\_si}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{slong} \ y, \text{slong} \ \text{prec})

void \texttt{acb\_submul\_fmpz}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \ fmzp\_t \ y, \text{slong} \ \text{prec})

void \texttt{acb\_submul\_arb}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{const} \arb\_t \ y, \text{slong} \ \text{prec})
Sets \(z\) to \(z\) minus the product of \(x\) and \(y\).

void \texttt{acb\_inv}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{slong} \ \text{prec})
Sets \(z\) to the multiplicative inverse of \(x\).

void \texttt{acb\_div\_ui}(acb\_t \ z, \text{const} \ acb\_t \ x, \text{ulong} \ y, \text{slong} \ \text{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.

Mathematical constants

void acb_const_pi(acb_t y, slong prec)
Sets y to the constant π.

Powers and roots

void acb_pow_ui(acb_t z, const acb_t x, slong prec)
void acb_pow_fmpz(acb_t z, const acb_t x, const fmpz_t y, slong prec)
void acb_pow_analytic(acb_t z, const acb_t x, const fmpz_t y, slong prec)

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)

Computes the power \( x^y \) to the quotient of \( x \) and \( y \).

void acb_rsqrt(acb_t z, const acb_t r, int analytic, slong prec)

void acb_rsqrt_analytic(acb_t z, const acb_t r, int analytic, slong prec)

Sets \( z \) to \( r^{1/2} \) 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 \texttt{acb_pow()} may be superior).

void acb_sqrt(acb_t z, const acb_t r, int analytic, slong prec)

void acb_sqrt_analytic(acb_t z, const acb_t r, int analytic, slong prec)

Sets \( z \) to \( r^{1/2} \) 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 \( z = \exp(y \log x) \).

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)

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 \( y \).

sets z to the quotient of x and y.
Exponentials and logarithms

void \texttt{acb\_exp}(\texttt{acb\_t y}, \texttt{const acb\_t z}, \texttt{slong prec})
Sets \( y \) to the exponential function of \( z \), computed as \( \exp(a + bi) = \exp(a)(\cos(b) + \sin(b)i) \).

void \texttt{acb\_exp\_pi}(\texttt{acb\_t y}, \texttt{const acb\_t z}, \texttt{slong prec})
Sets \( y \) to \( \exp(\pi i z) \).

void \texttt{acb\_exp\_invexp}(\texttt{acb\_t s}, \texttt{acb\_t t}, \texttt{const acb\_t z}, \texttt{slong prec})
Sets \( v = \exp(z) \) and \( w = \exp(-z) \).

void \texttt{acb\_expml}(\texttt{acb\_t res}, \texttt{const acb\_t z}, \texttt{slong prec})
Computes \( \exp(z) - 1 \), using an accurate method when \( z \approx 0 \).

void \texttt{acb\_log}(\texttt{acb\_t y}, \texttt{const acb\_t z}, \texttt{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 \texttt{acb\_log\_analytic}(\texttt{acb\_t r}, \texttt{const acb\_t z}, \texttt{int analytic}, \texttt{slong prec})
Computes the natural logarithm. If \texttt{analytic} is set, gives a NaN-containing result if \( z \) touches the branch cut.

void \texttt{acb\_loglp}(\texttt{acb\_t z}, \texttt{const acb\_t x}, \texttt{slong prec})
Sets \( z = \log(1 + x) \), computed accurately when \( x \approx 0 \).

Trigonometric functions

void \texttt{acb\_sin}(\texttt{acb\_t s}, \texttt{const acb\_t z}, \texttt{slong prec})
void \texttt{acb\_cos}(\texttt{acb\_t c}, \texttt{const acb\_t z}, \texttt{slong prec})

void \texttt{acb\_sin\_cos}(\texttt{acb\_t s}, \texttt{acb\_t c}, \texttt{const acb\_t z}, \texttt{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 \texttt{acb\_tan}(\texttt{acb\_t s}, \texttt{const acb\_t z}, \texttt{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 \texttt{acb\_cot}(\texttt{acb\_t s}, \texttt{const acb\_t z}, \texttt{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 \texttt{acb\_sin\_pi}(\texttt{acb\_t s}, \texttt{const acb\_t z}, \texttt{slong prec})
void \texttt{acb\_cos\_pi}(\texttt{acb\_t s}, \texttt{const acb\_t z}, \texttt{slong prec})

void \texttt{acb\_sin\_cos\_pi}(\texttt{acb\_t s}, \texttt{acb\_t c}, \texttt{const acb\_t z}, \texttt{slong prec})
Sets \( s = \sin(\pi z) \), \( c = \cos(\pi z) \), evaluating the trigonometric factors of the real and imaginary part accurately via \texttt{arb\_sin\_cos\_pi()}.

void \texttt{acb\_tan\_pi}(\texttt{acb\_t s}, \texttt{const acb\_t z}, \texttt{slong prec})
Sets \( s = \tan(\pi z) \). Uses the same algorithm as \texttt{acb\_tan()} , but evaluates the sine and cosine accurately via \texttt{arb\_sin\_cos\_pi()}.

void \texttt{acb\_cot\_pi}(\texttt{acb\_t s}, \texttt{const acb\_t z}, \texttt{slong prec})
Sets \( s = \cot(\pi z) \). Uses the same algorithm as \texttt{acb\_cot()} , but evaluates the sine and cosine accurately via \texttt{arb\_sin\_cos\_pi()}.

void \texttt{acb\_sec}(\texttt{acb\_t res}, \texttt{const acb\_t z}, \texttt{slong prec})
Computes \( \sec(z) = 1/\cos(z) \).

void \texttt{acb\_csc}(\texttt{acb\_t res}, \texttt{const acb\_t z}, \texttt{slong prec})
Computes \( \csc(x) = 1/\sin(z) \).
void \texttt{acb\_sinc}(acb\_t s, const acb\_t z, slong prec)
Sets $s = \sin(x) = \sin(z)/z$.

void \texttt{acb\_sinc\_pi}(acb\_t s, const acb\_t z, slong prec)
Sets $s = \sin(\pi x) = \sin(\pi z)/\pi z$.

**Inverse trigonometric functions**

void \texttt{acb\_asin}(acb\_t res, const acb\_t z, slong prec)
Sets \texttt{res} to \texttt{asin}(z) = -i \log(i z + \sqrt{1 - z^2})

void \texttt{acb\_acos}(acb\_t res, const acb\_t z, slong prec)
Sets \texttt{res} to \texttt{acos}(z) = \frac{1}{2} \pi - \texttt{asin}(z).

void \texttt{acb\_atan}(acb\_t res, const acb\_t z, slong prec)
Sets \texttt{res} to \texttt{atan}(z) = \frac{1}{2} i (\log(1 - iz) - \log(1 + iz)).

**Hyperbolic functions**

void \texttt{acb\_sinh}(acb\_t s, const acb\_t z, slong prec)
void \texttt{acb\_cosh}(acb\_t c, const acb\_t z, slong prec)
void \texttt{acb\_sinh\_cosh}(acb\_t s, acb\_t c, const acb\_t z, slong prec)
void \texttt{acb\_tanh}(acb\_t s, const acb\_t z, slong prec)
void \texttt{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).

void \texttt{acb\_sech}(acb\_t res, const acb\_t z, slong prec)
Computes sech(z) = 1/cosh(z).

void \texttt{acb\_csch}(acb\_t res, const arb\_t z, slong prec)
Computes csch(z) = 1/sinh(z).

**Inverse hyperbolic functions**

void \texttt{acb\_asinh}(acb\_t res, const acb\_t z, slong prec)
Sets \texttt{res} to \texttt{asinh}(z) = -i \sinh(iz).

void \texttt{acb\_acosh}(acb\_t res, const acb\_t z, slong prec)
Sets \texttt{res} to \texttt{acosh}(z) = \log(z + \sqrt{z^2 + 1}).

void \texttt{acb\_atanh}(acb\_t res, const acb\_t z, slong prec)
Sets \texttt{res} to \texttt{atanh}(z) = -i \arctan(iz).

**Lambert W function**

void \texttt{acb\_lambertw\_asymp}(acb\_t res, const acb\_t z, const fmpz\_t k, slong L, slong M, slong prec)
Sets \texttt{res} to the Lambert W function $W_k(z)$ computed using \texttt{L} and \texttt{M} terms in the bivariate series giving the asymptotic expansion at zero or infinity. This algorithm is valid everywhere, but the error bound is only finite when $|\log(z)|$ is sufficiently large.

int \texttt{acb\_lambertw\_check\_branch}(const acb\_t w, const fmpz\_t k, slong prec)
Tests if \texttt{w} definitely lies in the image of the branch $W_k(z)$. This function is used internally to verify that a computed approximation of the Lambert W function lies on the intended branch. Note that this will necessarily evaluate to false for points exactly on (or overlapping) the branch cuts, where a different algorithm has to be used.

4.2. acb.h – complex numbers 73
void \texttt{acb\_lambertw\_bound\_deriv}(\texttt{mag\_t res}, \texttt{acb\_t z}, \texttt{acb\_t ez1}, \texttt{fmpz\_t k})

Sets \texttt{res} to an upper bound for \(|W_k'(z)|\). The input \texttt{ez1} should contain the precomputed value of \(ez + 1\).

Along the real line, the directional derivative of \(W_k(z)\) is understood to be taken. As a result, the user must handle the branch cut discontinuity separately when using this function to bound perturbations in the value of \(W_k(z)\).

void \texttt{acb\_lambertw}(\texttt{acb\_t res}, \texttt{acb\_t z}, \texttt{fmpz\_t k}, \texttt{int flags}, \texttt{slong prec})

Sets \texttt{res} to the Lambert W function \(W_k(z)\) where the index \(k\) selects the branch (with \(k = 0\) giving the principal branch). The placement of branch cuts follows [CGHJK1996].

If \texttt{flags} is nonzero, nonstandard branch cuts are used.

If \texttt{flags} is set to \texttt{ACB\_LAMBERTW\_LEFT}, computes \(W_{k\leftarrow k}(z)\) which corresponds to \(W_k(z)\) in the upper half plane and \(W_{k+1}(z)\) in the lower half plane, connected continuously to the left of the branch points. In other words, the branch cut on \((-\infty, 0)\) is rotated counterclockwise to \((0, +\infty)\). (For \(k = -1\) and \(k = 0\), there is also a branch cut on \((-1/e, 0)\), continuous from below instead of from above to maintain counterclockwise continuity.)

If \texttt{flags} is set to \texttt{ACB\_LAMBERTW\_MIDDLE}, computes \(W_{\text{middle}}(z)\) which corresponds to \(W_{-1}(z)\) in the upper half plane and \(W_1(z)\) in the lower half plane, connected continuously through \((-1/e, 0)\) with branch cuts on \((-\infty, -1/e)\) and \((0, +\infty)\). \(W_{\text{middle}}(z)\) extends the real analytic function \(W_{-1}(x)\) defined on \((-1/e, 0)\) to a complex analytic function, whereas the standard branch \(W_{-1}(z)\) has a branch cut along the real segment.

The algorithm used to compute the Lambert W function is described in [Joh2017b].

**Rising factorials**

void \texttt{acb\_rising\_ui\_bs}(\texttt{acb\_t z}, \texttt{acb\_t x}, \texttt{ulong n}, \texttt{slong prec})

void \texttt{acb\_rising\_ui\_rs}(\texttt{acb\_t z}, \texttt{acb\_t x}, \texttt{ulong n}, \texttt{ulong step}, \texttt{slong prec})

void \texttt{acb\_rising\_ui\_rec}(\texttt{acb\_t z}, \texttt{acb\_t x}, \texttt{ulong n}, \texttt{slong prec})

void \texttt{acb\_rising\_ui}(\texttt{acb\_t z}, \texttt{acb\_t x}, \texttt{ulong n}, \texttt{slong prec})

void \texttt{acb\_rising}(\texttt{acb\_t z}, \texttt{acb\_t x}, \texttt{acb\_t n}, \texttt{slong prec})

Computes the rising factorial \(z = x(x+1)(x+2)\cdots(x+n-1)\).

The \texttt{bs} version uses binary splitting. The \texttt{rs} version uses rectangular splitting. The \texttt{rec} version uses either \texttt{bs} or \texttt{rs} depending on the input. The default version uses the gamma function unless \(n\) is a small integer.

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

void \texttt{acb\_rising\_2\_ui\_bs}(\texttt{acb\_t u}, \texttt{acb\_t v}, \texttt{acb\_t x}, \texttt{ulong n}, \texttt{slong prec})

void \texttt{acb\_rising\_2\_ui\_rs}(\texttt{acb\_t u}, \texttt{acb\_t v}, \texttt{acb\_t x}, \texttt{ulong n}, \texttt{ulong step}, \texttt{slong prec})

void \texttt{acb\_rising\_2\_ui}(\texttt{acb\_t u}, \texttt{acb\_t v}, \texttt{acb\_t x}, \texttt{ulong n}, \texttt{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 \texttt{step} to override the default choice), and an automatic algorithm choice.

void \texttt{acb\_rising\_ui\_get\_mag}(\texttt{mag\_t bound}, \texttt{acb\_t x}, \texttt{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 \(n\).
### 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 \( \text{acb_log_sin_pi()} \).

- 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_polygamma**(*acb_t y, const acb_t x, const acb_t s, slong prec*)
  Computes the polygamma function \( y = \psi^{(s)}(x) \).

  Sets \( \text{res} \) 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 \( \text{acb_poly_digamma_series()} \) 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 y, 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

$$\log G(z) = (z-1) \log \Gamma(z) - \zeta'(1) + \zeta'(-1).$$

**Zeta function**

```c
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.

This is a wrapper of `acb_dirichlet_zeta()`.

```c
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.

This is a wrapper of `acb_dirichlet_hurwitz()`.

```c
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.

**Polylogarithms**

```c
void acb_polylog(acb_t w, const acb_t s, const acb_t z, slong prec)
```
```c
void acb_polylog_si(acb_t w, slong s, const acb_t z, slong prec)
```
Sets $w$ to the polylogarithm $\mathrm{Li}_s(z)$.

**Arithmetic-geometric mean**

See *Algorithms for the arithmetic-geometric mean* for implementation details.

```c
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.

```c
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. $M(z + x) \in \mathbb{C}[[x]]$.

**Other special functions**

```c
void acb_chebyshev_t_ui(acb_t a, ulong n, const acb_t x, slong prec)
```
```c
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 $a = U_n(x)$.

```c
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.

Piecewise real functions

The following methods extend common piecewise real functions to piecewise complex analytic functions, useful together with the `acb_calc.h` module. If `analytic` is set, evaluation on a discontinuity or non-analytic point gives a NaN result.

void acb_real_abs(acb_t res, const acb_t z, int analytic, slong prec)
   The absolute value is extended to $+z$ in the right half plane and $-z$ in the left half plane, with a discontinuity on the vertical line $\Re(z) = 0$.

void acb_real_sgn(acb_t res, const acb_t z, int analytic, slong prec)
   The sign function is extended to $+1$ in the right half plane and $-1$ in the left half plane, with a discontinuity on the vertical line $\Re(z) = 0$. If `analytic` is not set, this is effectively the same function as `acb_csgn()`.

void acb_real_heaviside(acb_t res, const acb_t z, int analytic, slong prec)
   The Heaviside step function (or unit step function) is extended to $+1$ in the right half plane and $0$ in the left half plane, with a discontinuity on the vertical line $\Re(z) = 0$.

void acb_real_floor(acb_t res, const acb_t z, int analytic, slong prec)
   The floor function is extended to a piecewise constant function equal to $n$ in the strips with real part $(n, n+1)$, with discontinuities on the vertical lines $\Re(z) = n$.

void acb_real ceil(acb_t res, const acb_t z, int analytic, slong prec)
   The ceiling function is extended to a piecewise constant function equal to $n + 1$ in the strips with real part $(n, n+1)$, with discontinuities on the vertical lines $\Re(z) = n$.

void acb_real_max(acb_t res, const acb_t x, const acb_t y, int analytic, slong prec)
   The real function $\max(x, y)$ is extended to a piecewise analytic function of two variables by returning $x$ when $\Re(x) \geq \Re(y)$ and returning $y$ when $\Re(x) < \Re(y)$, with discontinuities where $\Re(x) = \Re(y)$.

void acb_real_min(acb_t res, const acb_t x, const acb_t y, int analytic, slong prec)
   The real function $\min(x, y)$ is extended to a piecewise analytic function of two variables by returning $x$ when $\Re(x) \leq \Re(y)$ and returning $y$ when $\Re(x) > \Re(y)$, with discontinuities where $\Re(x) = \Re(y)$.

void acb_real_sqrt_pos(acb_t res, const acb_t z, int analytic, slong prec)
   Extends the real square root function on $[0, +\infty)$ to the usual complex square root on the cut plane. Like `arb_sqrt_pos()`, only the nonnegative part of $z$ is considered if $z$ is purely real and `analytic` is not set. This is useful for integrating $\sqrt{f(x)}$ where it is known that $f(x) \geq 0$: unlike `acb_sqrt_analytic()`, no spurious imaginary terms $[\pm \varepsilon]i$ are created when the balls computed for $f(x)$ straddle zero.

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 vec, 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`. 
Arb Documentation, Release 2.13.0

Chapter 4. Real and complex numbers

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, acb_t c, slong prec)

void _acb_vec_scalar_addmul(acb_ptr res, acb_srcptr vec, acb_t c, slong prec)

void _acb_vec_mul(acb_ptr res, acb_srcptr vec1, acb_srcptr vec2, slong len, slong prec)

void _acb_vec_scalar_mul(acb_ptr res, acb_srcptr vec, acb_t c, slong prec)

void _acb_vec_scalar_mul_onei(acb_ptr res, acb_srcptr vec, slong len, slong c)

void _acb_vec_scalar_mul_fmpz(acb_ptr res, acb_srcptr vec, fmpz * c, slong prec)

void _acb_vec_scalar_mul_arb(acb_ptr res, acb_srcptr vec, arb_t c, slong prec)

void _acb_vec_scalar_div(acb_ptr res, acb_srcptr vec, arb_t c, slong prec)

void _acb_vec_scalar_div_ui(acb_ptr res, acb_srcptr vec, ulong c, slong prec)

void _acb_vec_scalar_div_fmpz(acb_ptr res, acb_srcptr vec, fmpz_t c, slong prec)

void _acb_vec_scalar_div_2exp_si(acb_ptr res, acb_srcptr vec, slong c)

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.

void _acb_vec_get_unique_fmpz_vec(fmpz_t * 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_set_powers(acb_ptr res, acb_srcptr vec, slong len, const arb_t c, slong order)
    Sets res to a copy of vec, each entry raised to the powers of c, where c = exp(2πi order/len).

void _acb_vec_unit_roots(acb_ptr res, slong order, slong len, slong prec)
    Sets res to the powers 1, z, z^2, ..., z^{len-1} where z = exp(2πi order/len) to precision prec. order can be taken negative.

In order to avoid precision loss, this function does not simply compute powers of a primitive root.

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)

void _acb_vec_round(acb_ptr res, acb_srcptr vec, slong len)
    Applies _acb_set_round() elementwise.

void _acb_vec_trunc(acb_ptr res, acb_srcptr vec, slong len)
    Applies _acb_set_round() elementwise.

int _acb_vec_get_unique_fmpz(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_trim(acb_ptr res, acb_srcptr vec, slong len)
    Applies _acb_trim() elementwise.

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)

void _acb_vec_round(acb_ptr res, acb_srcptr vec, slong len)
    Applies _acb_set_round() elementwise.

void _acb_vec_trunc(acb_ptr res, acb_srcptr vec, slong len)
    Applies _acb_set_round() elementwise.

int _acb_vec_get_unique_fmpz(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.
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.

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.

Types, macros and constants

```c
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.
```

Memory management

```c
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.

slong arb_poly_allocated_bytes(const arb_poly_t x)
Returns the total number of bytes heap-allocated internally by this object. The count excludes
```
the size of the structure itself. Add `sizeof(arb_poly_struct)` to get the size of the object as a whole.

### Basic manipulation

```c
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.

```c
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.

```c
int arb_poly_is_zero(const arb_poly_t poly)
```
```c
int arb_poly_is_one(const arb_poly_t poly)
```
```c
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.

```c
void arb_poly_zero(arb_poly_t poly)
```
```c
void arb_poly_one(arb_poly_t poly)
```
Sets `poly` to the constant 0 respectively 1.

```c
void arb_poly_set(arb_poly_t dest, const arb_poly_t src)
```
Sets `dest` to a copy of `src`.

```c
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.

```c
void arb_poly_set_trunc(arb_poly_t dest, const arb_poly_t src, slong n)
```
```c
void arb_poly_set_trunc_round(arb_poly_t dest, const arb_poly_t src, slong n, slong prec)
```
Sets `dest` to a copy of `src`, truncated to length `n` and rounded to `prec` bits.

```c
void arb_poly_set_coeff_si(arb_poly_t poly, slong n, slong c)
```
```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.

```c
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.

```c
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`.

```c
void _arb_poly_shift_right(arb_ptr res, arb_srcptr poly, slong len, slong n)
```
```c
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.

```c
void _arb_poly_shift_left(arb_ptr res, arb_srcptr poly, slong len, slong n)
```
```c
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.

```c
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`. We require that `n` is nonnegative.

```c
slong arb_poly_valuation(const arb_poly_t poly)
```
Returns the degree of the lowest term that is not exactly zero in `poly`. Returns -1 if `poly` is the zero polynomial.
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.

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.

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.

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 is 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.

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.
Arithmetic

```c
void _arb_poly_add污泥 C污泥 srcptr A污泥 long lenA污泥 srcptr B污泥 long lenB污泥 long prec)
Sets \( \{ C, \max(\text{lenA}, \text{lenB}) \} \) to the sum of \( \{ A, \text{lenA} \} \) and \( \{ B, \text{lenB} \} \). Allows aliasing of the input and output operands.
```
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.

```c
void _arb_poly_mullow_classical(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 \( n \).

```c
void _arb_poly_mul(arb_ptr C, const arb_srcptr A, slong lenA, const arb_srcptr B, slong lenB, slong prec)
```

Sets \( \{ C, \text{lenA} + \text{lenB} - 1 \} \) to the product of \( \{ A, \text{lenA} \} \) and \( \{ B, \text{lenB} \} \). 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 \_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.

```c
void _arb_poly_inv_series(arb_ptr Q, const arb_srcptr A, slong Alen, slong len, slong prec)
```

Sets \( \{ Q, \text{len} \} \) to the power series inverse of \( \{ A, \text{Alen} \} \). Uses Newton iteration.

```c
void _arb_poly_div_series(arb_ptr Q, const arb_srcptr A, slong Alen, const arb_srcptr B, slong Blen, slong n, slong prec)
```

Sets \( \{ Q, \text{n} \} \) to the power series quotient of \( \{ A, \text{Alen} \} \) by \( \{ B, \text{Blen} \} \). Uses Newton iteration followed by multiplication.

```c
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 \( n \).

```c
int _arb_poly_divrem(arb_ptr Q, const arb_srcptr A, slong lenA, const arb_srcptr B, slong lenB, 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.

```c
void _arb_poly_div_root(arb_ptr Q, const arb_t R, const 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 \( R = f(c) \).
Composition

```c
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)
void _arb_poly_compose_series_horner(arb_ptr res, arb_srcptr poly1, slong len1, arb_srcptr poly2, slong len2, slong prec)
void arb_poly_compose_series_horner(arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, 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)
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)
```

The underscore methods act in-place on $g = f(x + c)$, 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 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.

Sets $g$ to the Taylor shift $f(x + c)$, computed respectively using Horner’s rule, divide-and-conquer, a single convolution, and an automatic choice between the three algorithms.

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.

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.

```c
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_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.

### Evaluation

```c
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.

```c
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.

```c
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_rectangular(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)

Product trees

void _arb_poly_product_roots(arb_ptr poly, arb_srcptr xs, slong n, slong prec)

Generates the polynomial \((x - x_0)(x - x_1)...(x - x_{n-1})\).

void _arb_poly_product_roots_complex(arb_ptr poly, arb_srcptr r, slong rn, acb_srcptr c, slong cn, slong prec)

Generates the polynomial

\[
\left( \prod_{i=0}^{rn-1} (x - r_i) \right) \left( \prod_{i=0}^{cn-1} (x - c_i) (x - \bar{c}_i) \right)
\]

having \( rn \) real roots given by the array \( r \) and having \( 2cn \) complex roots in conjugate pairs given by the length-\( cn \) array \( c \). Either \( rn \) or \( cn \) or both may be zero.

Note that only one representative from each complex conjugate pair is supplied (unless a pair is supposed to be repeated with higher multiplicity). To construct a polynomial from complex roots where the conjugate pairs have not been distinguished, use \( \text{acb_poly_product_roots()} \) instead.

void _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().

**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.

**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)

Recovery 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)

Recovery 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)

Recovery 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.

**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 \( \{\text{res, len}\} \) to the integral of \( \{\text{poly, len - 1}\} \). Allows aliasing of the input and output.

void arb_poly_integral(arb_poly_t res, const arb_poly_t poly, slong prec)
Sets \( \text{res} \) to the integral of \( \text{poly} \).

Transforms

void _arb_poly_borel_transform(arb_ptr res, arb_srcptr poly, slong len, 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_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)
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_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)
Computes the binomial transform of the input polynomial, truncating the output to length \( \text{len} \). The binomial transform maps the coefficients \( a_k \) in the input polynomial to the coefficients \( b_k \) in the output polynomial via \( b_k = \sum_{k=0}^{\text{alen}} (-1)^k \binom{k}{n} a_k \). 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 underscore version automatically chooses an algorithm.

The underscore methods do not support aliasing, and assume that the lengths are nonzero.

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 \( \{\text{res, len}\} \) to \( \{f, \text{flen}\} \) raised to the power \( \text{exp} \), truncated to length \( \text{len} \). Requires that \( \text{len} \) 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 \( \text{flen} \) and \( \text{len} \) are positive. Uses binary exponentiation.

void arb_poly_pow_ui_trunc_binexp(arb_poly_t res, const arb_poly_t poly, ulong exp, slong len, slong prec)
Sets \( \text{res} \) to \( \text{poly} \) raised to the power \( \text{exp} \), truncated to length \( \text{len} \). Uses binary exponentiation.
void _arb_poly_pow_ui(arb_ptr res, arb_srcptr f, slong flen, ulong exp, slong prec)

Sets res to \( f \) raised to the power \( \exp \). Does not support aliasing of the input and output, and requires that \( \text{flen} \) 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 \( \{h, \text{len}\} \) to the power series \( f(x)^g(x) = \exp(g(x) \log f(x)) \) truncated to length \( \text{len} \). 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 \( \text{flen} \) and \( \text{glen} \) do not exceed \( \text{len} \).

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 \( \text{len} \). 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_series(arb_poly_t h, arb_srcptr f, slong flen, const arb_poly_t g, slong len, slong prec)

Sets \( \{h, \text{len}\} \) to the power series \( f(x)^g = \exp(g \log f(x)) \) truncated to length \( \text{len} \). 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 \( \text{flen} \) does not exceed \( \text{len} \).

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 \( f(x)^g = \exp(g \log f(x)) \) truncated to length \( \text{len} \).

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 \( \text{hlen} \) 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 \( \text{hlen} \) 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 \( \text{flen} \) and \( n \) are greater than zero.

void _arb_poly_log1p_series(arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec)

Computes the power series \( \log(1 + f) \), with better accuracy when the constant term of \( f \) is small.

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)
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_basecase(arb_poly_t h, slong n, slong prec)
void _arb_poly_sin_cos_series(arb_poly_t h, 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
\[
\tan^{-1}(f(x)) = \int \frac{f'(x)}{(1 + f(x)^2)} dx,
\]
\[
\sin^{-1}(f(x)) = \int \frac{f'(x)}{(1 - f(x)^2)^{1/2}} dx,
\]
\[
\cos^{-1}(f(x)) = -\int \frac{f'(x)}{(1 - f(x)^2)^{1/2}} dx,
\]
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 \(f\) 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)

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 \(O(nmn)\) 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 underscore methods support aliasing and require the lengths to be nonzero.

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 $\pi$.

The underscore methods support aliasing and require the lengths to be nonzero.

```c
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_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)
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 c, arb_srcptr h, slong hlen, slong n, slong prec)
void arb_poly_sinh_series(arb_poly_t c, const arb_poly_t h, slong n, slong prec)
void _arb_poly_cosh_series(arb_ptr s, arb_srcptr h, slong hlen, slong n, slong prec)
void arb_poly_cosh_series(arb_poly_t s, const arb_poly_t h, 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.

Lambert W function

void _arb_poly_lambertw_series(arb_ptr res, arb_srcptr z, slong zlen, int flags, slong len, slong prec)
void arb_poly_lambertw_series(arb_poly_t res, const arb_poly_t z, int flags, slong len, slong prec)

Sets res to the Lambert W function of the power series z. If flags is 0, the principal branch is computed; if flags is 1, the second real branch $W_{-1}(z)$ is computed. The underscore method allows aliasing, but assumes that the lengths are nonzero.

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)
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.

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 \( \text{deflate} \) 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.

### Root-finding

void \_arb_poly_root_bound_fujiwara(mag_t bound, arb_srcptr poly, slong len)

Sets \( \text{bound} \) to an upper bound for the magnitude of all the complex roots of \( \text{poly} \). Uses Fujiwara’s bound

\[
2 \max \left\{ \frac{a_{n-1}}{a_n}, \left| \frac{a_{n-2}}{a_n} \right|^{1/2}, \ldots, \left| \frac{a_1}{a_n} \right|^{1/(n-1)}, \left| \frac{a_0}{2a_n} \right|^{1/n} \right\}
\]

where \( a_0, \ldots, a_n \) are the coefficients of \( \text{poly} \).

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 \( \text{convergence_interval} \), 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 \( \{\text{poly}, \text{len}\} \). 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 \(\{\text{poly, len}\}\), an interval $x$ = \([m - r, m + r]\)$ known to contain a single root of $f$, an interval $I$ (\textit{convergence_interval}) containing $x$ with an associated bound (\textit{convergence_factor}) for $C = \sup_{t, u \in I} \frac{1}{2} |f''(t)|/|f'(u)|$, and a working precision \textit{prec}.

The Newton update consists of setting 

\[
x' = [m' - r', m' + r'] \quad \text{where} \quad m' = m - f(m)/f'(m) \quad \text{and} \quad r' = C r^2.
\]

The expression $m - f(m)/f'(m)$ is evaluated using ball arithmetic at a working precision of \textit{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: \textit{prec} is the target accuracy and \textit{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).

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 \textit{prec} is set to zero, a precision is chosen automatically such that 

\texttt{arb_poly_get_unique_fmpz_poly()} should be successful. Otherwise a working precision of \textit{prec} bits is used.

The underscore version accepts an additional \textit{trunc} parameter. Even when computing a truncated polynomial, the array \textit{poly} must have room for $2^n + 1$ coefficients, used as temporary space.

\texttt{acb\_poly\_h \textit{-- polynomials over the complex numbers}}

An \texttt{acb\_poly\_t} represents a polynomial over the complex numbers, implemented as an array of coefficients of type \texttt{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.

\textbf{Types, macros and constants}

\texttt{acb\_poly\_struct}

\texttt{acb\_poly\_t}

Contains a pointer to an array of coefficients (\texttt{coeffs}), the used length (\texttt{length}), and the allocated size of the array (\texttt{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.

**Memory management**

```c
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.

slong acb_poly_allocated_bytes(const acb_poly_t x)
    Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(acb_poly_struct)` to get the size of the object as a whole.
```

**Basic properties and manipulation**

```c
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)
    Returns 1 if `poly` is exactly the polynomial 0, 1 or `x` respectively. Returns 0 otherwise.

int acb_poly_is_one(const acb_poly_t poly)

int acb_poly_is_x(const acb_poly_t poly)

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_trunc(acb_poly_t dest, const acb_poly_t src, slong n)
    Sets `dest` to a copy of `src`, truncated to length `n` and rounded to `prec` bits.

void acb_poly_set_coeff_si(acb_poly_t poly, slong n, slong c)
```

5.2. `acb_poly.h` — polynomials over the complex numbers
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 ≥ 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. We require that n is nonnegative.

slong acb_poly_valuation(const acb_poly_t poly)
    Returns the degree of the lowest term that is not exactly zero in poly. Returns -1 if poly is the zero polynomial.

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.

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.

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 is 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.

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 \( src \).

Bounds

void _acb_poly_majorant(arb_ptr res, acb_srcptr poly, slong len, slong prec)
void acb_poly_majorant(acb_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.

Arithmetic

void _acb_poly_add(acb_ptr C, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong prec)
Sets \( \{ C, \max\{\text{len}A, \text{len}B\} \} \) to the sum of \( \{ A, \text{len}A \} \) and \( \{ B, \text{len}B \} \). 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 \( \{ C, \max\{\text{len}A, \text{len}B\} \} \) to the difference of \( \{ A, \text{len}A \} \) and \( \{ B, \text{len}B \} \). 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_add_series(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong len, slong prec)
Sets \( C \) to the sum of \( A \) and \( B \), truncated to length \( \text{len} \).

void acb_poly_sub_series(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong len, slong prec)
Sets \( C \) to the difference of \( A \) and \( B \), truncated to length \( \text{len} \).
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 \(\{A, \text{lenA}\}\) and \(\{B, \text{lenB}\}\), 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 between 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 \(\{C, n\}\) to the product of \(\{A, \text{lenA}\}\) and \(\{B, \text{lenB}\}\). 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 \(\{Q\inv, \text{len}\}\) to the power series inverse of \(\{Q, \text{Qlen}\}\). Uses Newton iteration.

void acb_poly_inv_series(acb_poly_t Qinv, const acb_poly_t Q, slong n, slong prec)
  Sets \(Q\inv\) 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)

int 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 \(R = f(c)\).

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)
Evaluation

The underscore methods do not support aliasing of the output with either input polynomial.

Sets res to the composition \( h(x) = f(g(x)) \) where \( f \) is given by \( \text{poly1} \) and \( g \) is given by \( \text{poly2} \), 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.

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.

Sets res to the power series composition \( h(x) = f(g(x)) \) truncated to order \( O(x^n) \) where \( f \) is given by \( \text{poly1} \) and \( g \) is given by \( \text{poly2} \), respectively using Horner’s rule, the Brent-Kung baby step-giant step algorithm, and an automatic choice between the two algorithms.

We require that the constant term in \( f(x) \) is exactly zero and that the linear term is nonzero. The underscore methods assume that \( \text{flen} \) is at least 2, and do not support aliasing.

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 \( \text{flen} \) is at least 2, and do not support aliasing.
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 si-
      multaneously 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.

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 - x0)(x - x1)⋯(x - xn-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().

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() re-
      peatedly.
void _acb_poly_evaluate_vec_fast_precmp(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 \texttt{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.

**Interpolation**

void \texttt{\_acb\_poly\_interpolate\_newton}(acb\_ptr poly, acb\_srcptr xs, acb\_srcptr ys, slong n, slong prec)

void \texttt{acb\_poly\_interpolate\_newton}(acb\_poly\_t poly, acb\_srcptr xs, acb\_srcptr ys, slong n, slong prec)

Recover 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 \texttt{\_acb\_poly\_interpolate\_barycentric}(acb\_ptr poly, acb\_srcptr xs, acb\_srcptr ys, slong n, slong prec)

void \texttt{acb\_poly\_interpolate\_barycentric}(acb\_poly\_t poly, acb\_srcptr xs, acb\_srcptr ys, slong n, slong prec)

Recover 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 \texttt{\_acb\_poly\_interpolation\_weights}(acb\_ptr w, acb\_ptr \* tree, slong len, slong prec)

void \texttt{\_acb\_poly\_interpolate\_fast\_precomp}(acb\_ptr poly, acb\_srcptr ys, acb\_ptr \* tree, acb\_srcptr weights, slong len, slong prec)

void \texttt{acb\_poly\_interpolate\_fast}(acb\_ptr poly, acb\_srcptr xs, acb\_srcptr ys, slong len, slong prec)

void \texttt{acb\_poly\_interpolate\_fast}(acb\_poly\_t poly, acb\_srcptr xs, acb\_srcptr ys, slong n, slong prec)

Recover 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.

**Differentiation**

void \texttt{\_acb\_poly\_derivative}(acb\_ptr res, acb\_srcptr poly, slong len, slong prec)

Sets \( \{\text{res}, \text{len} - 1\} \) to the derivative of \( \{\text{poly}, \text{len}\} \). Allows aliasing of the input and output.

void \texttt{acb\_poly\_derivative}(acb\_poly\_t res, const acb\_poly\_t poly, slong prec)

Sets \( \text{res} \) to the derivative of \( \text{poly} \).

void \texttt{acb\_poly\_integral}(acb\_ptr res, acb\_srcptr poly, slong len, slong prec)

Sets \( \{\text{res, len}\} \) to the integral of \( \{\text{poly, len}-1\} \). Allows aliasing of the input and output.

void \texttt{acb\_poly\_integral}(acb\_poly\_t res, const acb\_poly\_t poly, slong prec)

Sets \( \text{res} \) to the integral of \( \text{poly} \).

**Transforms**

void \texttt{\_acb\_poly\_borel\_transform}(acb\_ptr res, acb\_srcptr poly, slong len, slong prec)

void \texttt{acb\_poly\_borel\_transform}(acb\_poly\_t res, const acb\_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 \texttt{\_acb\_poly\_inv\_borel\_transform}(acb\_ptr res, acb\_srcptr poly, slong len, slong prec)

void \texttt{acb\_poly\_inv\_borel\_transform}(acb\_poly\_t res, const acb\_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.
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 \(\text{len}\). Requires that \(\text{len}\) 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 \(\text{flen}\) and \(\text{len}\) are positive. Uses binary exponentiation.

void acb_poly_pow_ui_trunc_binexp(acb_poly_t res, const acb_poly_t poly, ulong exp, slong len, slong prec)

Sets \(\text{res}\) to \(\text{poly}\) raised to the power \(\exp\), truncated to length \(\text{len}\). Uses binary exponentiation.

void _acb_poly_pow_ui(acb_ptr res, acb_srcptr f, slong flen, ulong exp, slong prec)

Sets \(\text{res}\) to \(\{f, \text{flen}\}\) raised to the power \(\exp\). Does not support aliasing of the input and output, and requires that \(\text{flen}\) is positive.

void acb_poly_pow_ui(acb_poly_t res, const acb_poly_t poly, ulong exp, slong prec)

Sets \(\text{res}\) to \(\text{poly}\) 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 \(\{h, \text{len}\}\) to the power series \(f(x)^g(x) = \exp(g(x) \log f(x))\) truncated to length \(\text{len}\). 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 \(\text{flen}\) and \(\text{glen}\) do not exceed \(\text{len}\).

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 \(\text{len}\). 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 \(\{h, \text{len}\}\) to the power series \(f(x)^g = \exp(g \log f(x))\) truncated to length \(\text{len}\). 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 \(\text{flen}\) does not exceed \(\text{len}\).

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 \(\text{len}\).

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 f, acb_srcptr h, slong hlen, slong n, slong prec)
void acb_poly_rsqrt_series(acb_poly_t f, 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, acb_poly_t f, slong n, slong prec)
Sets $res$ to the power series logarithm of $f$, truncated to length $n$. Uses the formula $\log(f(x)) = \int \frac{f'(x)}{f(x)} dx$, 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_log1p_series(acb_ptr res, acb_srcptr f, slong flen, slong n, slong prec)
void acb_poly_log1p_series(acb_poly_t res, acb_poly_t f, slong n, slong prec)
Computes the power series $\log(1 + f(x))$ for $f$ truncated to length $n$. 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, acb_poly_t f, slong n, slong prec)
Sets $res$ to the power series inverse tangent of $f$, truncated to length $n$.

Uses the formula
$$\tan^{-1}(f(x)) = \int \frac{f'(x)}{1 + f(x)^2} dx,$$
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_exp_pi_i_series(acb_ptr f, acb_srcptr h, slong hlen, slong n, slong prec)
void acb_poly_exp_pi_i_series(acb_poly_t f, const acb_poly_t h, slong n, slong prec)
Sets $f$ to the power series $\exp(\pi i h)$ truncated to length $n$. The underscore method supports aliasing and allows the input to be shorter than the output, but requires 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)
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
\[
\sin(h_0 + h_1) = \sin(h_0) \cos(h_1) + \cos(h_0) \sin(h_1),
\]
\[
\cos(h_0 + h_1) = \cos(h_0) \cos(h_1) - \sin(h_0) \sin(h_1),
\]
respectively. For small \( h_0 \), uses Newton iteration to invert the inverse tangent series. The complexity is \( O(M(n)) \).

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.

Respectively evaluates the power series tangent of \( h \).

Respectively evaluates the power series tangent of \( h \).

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 \( O(M(n)) \).

Respectively evaluates the power series tangent of \( h \).

Respectively evaluates the power series tangent of \( h \).

Respectively evaluates the power series tangent of \( h \).

Respectively evaluates the power series tangent of \( h \).

Respectively evaluates the power series tangent of \( h \).
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 \( \pi \).

void _acb_poly_sinh_cosh_series_basecase(acb_ptr s, acb_ptr c, acb_srchpr 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_srchpr 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_srchpr 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_srchpr 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_srchpr 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)
void _acb_poly_sinc_series(acb_ptr s, acb_srchpr 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)

Lambert W function

void _acb_poly_lambertw_series(acb_ptr res, acb_srchpr z, slong zlen, const fmpz_t k, int flags, slong len, slong prec)
void acb_poly_lambertw_series(acb_poly_t res, const acb_poly_t z, const fmpz_t k, int flags, slong len, slong prec)
Sets res to branch \( k \) of the Lambert W function of the power series \( z \). The argument flags is reserved for future use. The underscore method allows aliasing, but assumes that the lengths are nonzero.

Gamma function

void _acb_poly_gamma_series(acb_ptr res, acb_srchpr 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_srchpr 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_srchpr 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_srchpr 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 $\log \Gamma(h(x))$, 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.

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.

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].

5.2. acb_poly.h – polynomials over the complex numbers 107
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, slong len, 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 $\text{prec}$ bits. With $a = 1$, this gives the usual Riemann zeta function.

If $\text{deflate}$ 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 $\text{deflate}$ 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 $\text{res}$ 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 $\text{deflate}$ 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.

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 $Li_{s+z}(z)$, where $s$ and $z$ are given complex constants. The output is computed to length $\text{len}$ 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 len, 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 $Li_s(z)$ where $s$ is a given power series, truncating the output to length
void _acb_poly_agm1_series(acb_ptr res, acb_srcptr z, slong len, slong prec)

Sets \( \text{res} \) to the arithmetic-geometric mean of \( 1 \) and the power series \( z \), truncated to length \( n \). These methods are provided for backwards compatibility. See acb_hypgeom_agm1_series(), acb_hypgeom_elliptic_p_series() for backwards compatibility.

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.

Given any complex number \( z \) that is guaranteed to contain at least one root of \( f \), \( |f(z)| < \epsilon \) for some \( \epsilon \), and a nonconstant polynomial \( f \) and its derivative \( f' \), sets \( r \) to the arithmetic-geometric mean of 1 and the power series \( f'/f \) of degree \( n \). Then

\[
\left| \frac{f'(m)}{f(m)} \right| \leq \sum_i \frac{1}{m - \zeta_i} \leq \sum_i \frac{1}{|m|} < \frac{n}{r} = \left| \frac{f'(m)}{f(m)} \right|
\]

which is a contradiction (see [Kob2010]).

void _acb_poly_agm1_series(acb_ptr res, acb_srcptr z, slong len, slong prec)

Sets \( \text{res} \) to the arithmetic-geometric mean of 1 and the power series \( z \), truncated to length \( n \).

See the acb_elliptic.h module for power series of elliptic functions. The following wrappers are available for backwards compatibility.

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 \( \text{bound} \) to an upper bound for the magnitude of all the complex roots of \( \text{poly} \). Uses Fujiwara’s bound

\[
2 \max \left\{ \frac{a_{n-1}}{a_n}, \frac{a_{n-2}}{a_n}, \ldots, \frac{a_1}{a_n}, \frac{a_0}{2a_n} \right\}^{1/n}
\]

where \( a_0, \ldots, a_n \) are the coefficients of \( \text{poly} \).

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

\[
\left| \frac{f'(m)}{f(m)} \right| = \sum_i \frac{1}{m - \zeta_i} \leq \sum_i \frac{1}{|m|} \leq \frac{n}{r} = \left| \frac{f'(m)}{f(m)} \right|
\]

which is a contradiction (see [Kob2010]).
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).

Attempts to compute all the roots of the given nonzero polynomial \( \text{poly} \) using a working precision of \( \text{prec} \) bits. If \( n \) denotes the degree of \( \text{poly} \), the function writes \( n \) approximate roots with rigorous error bounds to the preallocated array \( \text{roots} \), and returns the number of roots that are isolated.

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 \( \text{maxiter} \), 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 \( \text{initial} \). If \( \text{initial} \) is set to \( \text{NULL} \), default values \( (0.4 + 0.9i) \) 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.

Given a strictly real polynomial \( \text{poly} \) (of length \( \text{len} \)) 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 \( \text{arb_contains_zero()} \) 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 \( \text{prec} \).

This module provides methods for FLINT polynomials with integer and rational coefficients \( (\text{fmpz\_poly\_t}) \) and \( (\text{fmpq\_poly\_t}) \) requiring use of Arb real or complex numbers.

Some methods output real or complex numbers while others use real and complex numbers internally to produce an exact result. This module also contains some useful helper functions not specifically related to real and complex numbers.

Note that methods that combine Arb polynomials and FLINT polynomials are found in the respective Arb polynomial modules, such as \( \text{arb\_poly\_set\_fmpz\_poly()} \) and \( \text{arb\_poly\_get\_unique\_fmpz\_poly()} \).

**Evaluation**

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).

Attempts to compute all the roots of the given nonzero polynomial \( \text{poly} \) using a working precision of \( \text{prec} \) bits. If \( n \) denotes the degree of \( \text{poly} \), the function writes \( n \) approximate roots with rigorous error bounds to the preallocated array \( \text{roots} \), and returns the number of roots that are isolated.

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 \( \text{maxiter} \), 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 \( \text{initial} \). If \( \text{initial} \) is set to \( \text{NULL} \), default values \( (0.4 + 0.9i) \) 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.

Given a strictly real polynomial \( \text{poly} \) (of length \( \text{len} \)) 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 \( \text{arb\_contains\_zero()} \) 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 \( \text{prec} \).
void _arb_fmpz_poly_evaluate_arb_rectangular(arb_t res, const fmpz * poly, slong len, const arb_t x, slong prec)
void arf_fmpz_poly_evaluate_arb_rectangular(arb_t res, const fmpz_poly_t poly, const arb_t x, slong prec)
void _arb_fmpz_poly_evaluate_arb(arb_t res, const fmpz * poly, slong len, const arb_t x, slong prec)
void arf_fmpz_poly_evaluate_arb(arb_t res, const fmpz_poly_t poly, const arb_t x, slong prec)
void _arb_fmpz_poly_evaluate_acb_horner(acb_t res, const fmpz * poly, slong len, const acb_t x, slong prec)
void arf_fmpz_poly_evaluate_acb_horner(acb_t res, const fmpz_poly_t poly, const acb_t x, slong prec)
void _arb_fmpz_poly_evaluate_acb_rectangular(acb_t res, const fmpz * poly, slong len, const acb_t x, slong prec)
void arf_fmpz_poly_evaluate_acb_rectangular(acb_t res, const fmpz_poly_t poly, const acb_t x, slong prec)
void _arb_fmpz_poly_evaluate_acb(acb_t res, const fmpz * poly, slong len, const acb_t x, slong prec)
void arf_fmpz_poly_evaluate_acb(acb_t res, const fmpz_poly_t poly, const acb_t x, slong prec)

Evaluates poly (given by a polynomial object or an array with len coefficients) at the given real or complex number, respectively using Horner’s rule, rectangular splitting, or a default algorithm choice.

Utility methods

ulong arf_fmpz_poly_deflation(const fmpz_poly_t poly)
Finds the maximal exponent by which poly can be deflated.

void arf_fmpz_poly_deflate(fmpz_poly_t res, const fmpz_poly_t poly, ulong deflation)
Sets res to a copy of poly deflated by the exponent deflation.

Polynomial roots

void arf_fmpz_poly_complex_roots(acb_ptr roots, const fmpz_poly_t poly, int flags, slong prec)
Writes to roots all the real and complex roots of the polynomial poly, computed to prec accurate bits. The real roots are written first in ascending order (with the imaginary parts set exactly to zero). The following nonreal roots are written in arbitrary order, but with conjugate pairs grouped together (the root in the upper plane leading the root in the lower plane).

The input polynomial must be squarefree. For a general polynomial, compute the squarefree part \( f/gcd(f,f') \) or do a full squarefree factorization to obtain the multiplicities of the roots:

```c
fmpz_poly_factor_t fac;
fmpz_poly_factor_init(fac);
fmpz_poly_factor_squarefree(fac, poly);

for (i = 0; i < fac->num; i++)
{
    deg = fmpz_poly_degree(fac->p + i);
    flint_printf("%wd roots of multiplicity %wd\n", deg, fac->exp[i]);
    roots = _acb_vec_init(deg);
    arb_fmpz_poly_complex_roots(roots, fac->p + i, 0, prec);
    _acb_vec_clear(roots, deg);
}

fmpz_poly_factor_clear(fac);
```
All roots are refined to a relative accuracy of at least \( \text{prec} \) bits. The output values will generally have higher actual precision, depending on the precision used internally by the algorithm.

This implementation should be adequate for general use, but it is not currently competitive with state-of-the-art isolation methods for finding real roots alone.

The following flags are supported:

- \texttt{ARB_FMPZ\_POLY\_ROOTS\_VERBOS}\texttt{E}

\section*{Special polynomials}

Note: see also the methods available in FLINT (e.g. for cyclotomic polynomials).

\begin{verbatim}
void arb_fmpz_poly_cos_minpoly(fmpz_poly_t res, ulong n)
    Sets \( \text{res} \) to the monic minimal polynomial of \( 2 \cos(2\pi/n) \). This is a wrapper of FLINT’s \texttt{fmpz\_poly\_cos\_minpoly}, provided here for backward compatibility.

void arb_fmpz_poly_gauss_period_minpoly(fmpz_poly_t res, ulong q, ulong n)
    Sets \( \text{res} \) to the minimal polynomial of the Gaussian periods \( \sum_{a \in H} \zeta^a \) where \( \zeta = \exp(2\pi i/q) \) and \( H \) are the cosets of the subgroups of order \( d = (q - 1)/n \) of \( (\mathbb{Z}/q\mathbb{Z})^\times \). The resulting polynomial has degree \( n \). When \( d = 1 \), the result is the cyclotomic polynomial \( \Phi_q \).

The implementation assumes that \( q \) is prime, and that \( n \) is a divisor of \( q - 1 \) such that \( n \) is coprime with \( d \). If any condition is not met, \( \text{res} \) is set to the zero polynomial.

This method provides a fast (in practice) way to construct finite field extensions of prescribed degree. If \( q \) satisfies the conditions stated above and \( (q - 1)/f \) additionally is coprime with \( n \), where \( f \) is the multiplicative order of \( p \) mod \( q \), then the Gaussian period minimal polynomial is irreducible over \( \text{GF}(p) \) [CP2005].
\end{verbatim}
acb_dft.h – Discrete Fourier transform

Warning: the interfaces in this module are experimental and may change without notice.

All functions support aliasing.

Let $G$ be a finite abelian group, and $\chi$ a character of $G$. For any map $f : G \to \mathbb{C}$, the discrete fourier transform $\hat{f} : G \to \mathbb{C}$ is defined by

$$\hat{f}(\chi) = \sum_{x \in G} \chi(x) f(x)$$

Note that by the inversion formula

$$\hat{\hat{f}}(\chi) = \#G \times f(\chi^{-1})$$

it is straightforward to recover $f$ from its DFT $\hat{f}$.

**Main DFT functions**

If $G = \mathbb{Z}/n\mathbb{Z}$, we compute the DFT according to the usual convention

$$w_x = \sum_{y \mod n} v_y e^{-\frac{2\pi i xy}{n}}$$

void **acb_dft**(acb_ptr w, acb_srcptr v, slong n, slong prec)
\hspace{1cm} Set $w$ to the DFT of $v$ of length $len$, using an automatic choice of algorithm.

void **acb_dft_inverse**(acb_ptr w, acb_srcptr v, slong n, slong prec)
\hspace{1cm} Compute the inverse DFT of $v$ into $w$.

If several computations are to be done on the same group, the FFT scheme should be reused.

**acb_dft_pre_struct**

**acb_dft_pre_t**
\hspace{1cm} Stores a fast DFT scheme on $\mathbb{Z}/n\mathbb{Z}$ as a recursive decomposition into simpler DFT with some tables of roots of unity.

An **acb_dft_pre_t** is defined as an array of **acb_dft_pre_struct** of length 1, permitting it to be passed by reference.

void **acb_dft_precomp_init**(acb_dft_pre_t pre, slong len, slong prec)
\hspace{1cm} Initializes the fast DFT scheme of length $len$, using an automatic choice of algorithms depending on the factorization of $len$.

The length $len$ is stored as $pre->n$. 
void `acb_dft_precomp_clear(acb_dft_pre_t pre)`
Clears pre.

void `acb_dft_precomp(acb_ptr w, acb_srcptr v, const acb_dft_pre_t pre, slong prec)`
Computes the DFT of the sequence v into w by applying the precomputed scheme pre. Both v and w must have length pre->n.

void `acb_dft_inverse_precomp(acb_ptr w, acb_srcptr v, const acb_dft_pre_t pre, slong prec)`
Computes the inverse DFT of v into w.

**DFT on products**

A finite abelian group is isomorphic to a product of cyclic components

\[ G = \bigoplus_{i=1}^{r} \mathbb{Z}/n_i \mathbb{Z} \]

Characters are product of component characters and the DFT reads

\[ \hat{f}(x_1,\ldots,x_r) = \sum_{y_1\ldots y_r} f(y_1,\ldots,y_r) e^{-2\pi i \sum \frac{x_i y_i}{n_i}} \]

We assume that f is given by a vector of length \( \prod n_i \) corresponding to a lexicographic ordering of the values \( y_1,\ldots,y_r \), and the computation returns the same indexing for values of \( \hat{f} \).

void `acb_dirichlet_dft_prod(acb_ptr w, acb_srcptr v, slong * cyc, slong num, slong prec)`
Computes the DFT on the group product of num cyclic components of sizes cyc. Assume the entries of v are indexed according to lexicographic ordering of the cyclic components.

`acb_dft_prod_struct`

`acb_dft_prod_t`
Stores a fast DFT scheme on a product of cyclic groups.

An `acb_dft_prod_t` is defined as an array of `acb_dft_prod_struct` of length 1, permitting it to be passed by reference.

void `acb_dft_prod_init(acb_dft_prod_t t, slong * cyc, slong num, slong prec)`
Stores in t a DFT scheme for the product of num cyclic components whose sizes are given in the array cyc.

void `acb_dft_prod_clear(acb_dft_prod_t t)`
Clears t.

void `acb_dirichlet_dft_prod_precomp(acb_ptr w, acb_srcptr v, const acb_dft_prod_t prod, slong prec)`
Sets w to the DFT of v. Assume the entries are lexicographically ordered according to the product of cyclic groups initialized in t.

**Convolution**

For functions f and g on G we consider the convolution

\[ (f \ast g)(x) = \sum_{y \in G} f(x - y)g(y) \]

void `acb_dft_convol_naive(acb_ptr w, acb_srcptr f, acb_srcptr g, slong len, slong prec)`

void `acb_dft_convol_rad2(acb_ptr w, acb_srcptr f, acb_srcptr g, slong len, slong prec)`

114 Chapter 6. Transforms
void acb_dft_convol(acb_ptr w, acb_srcptr f, acb_srcptr g, slong len, slong prec)
Sets \( w \) to the convolution of \( f \) and \( g \) of length \( len \).

The \textit{naive} version simply uses the definition.

The \textit{rad2} version embeds the sequence into a power of 2 length and uses the formula

\[
\hat{f} \ast \hat{g}(\chi) = \hat{f}(\chi) \hat{g}(\chi)
\]

to compute it using three radix 2 FFT.

The default version uses radix 2 FFT unless \( len \) is a product of small primes where a non padded FFT is faster.

\textbf{FFT algorithms}

Fast Fourier transform techniques allow to compute efficiently all values \( \hat{f}(\chi) \) by reusing common computations.

Specifically, if \( H < G \) is a subgroup of size \( M \) and index \([G : H] = m\), then writing \( f_x(h) = f(xh) \) the translate of \( f \) by representatives \( x \) of \( G/H \), one has a decomposition

\[
\hat{f}(\chi) = \sum_{x \in G/H} \overline{\chi(x)} \hat{f}_x(\chi_H)
\]

so that the DFT on \( G \) can be computed using \( m \) DFT on \( H \) (of appropriate translates of \( f \)), then \( M \) DFT on \( G/H \), one for each restriction \( \chi_H \).

This decomposition can be done recursively.

\textbf{Naive algorithm}

void acb_dft_naive(acb_ptr w, acb_srcptr v, slong n, slong prec)
Computes the DFT of \( v \) into \( w \), where \( v \) and \( w \) have size \( n \), using the naive \( O(n^2) \) algorithm.

acb_dft_naive_struct

acb_dft_naive_t

void acb_dft_naive_init(acb_dft_naive_t t, slong len, slong prec)

void acb_dft_naive_clear(acb_dft_naive_t t)
Stores a table of roots of unity in \( t \). The length \( len \) is stored as \( t->n \).

void acb_dft_naive_precomp(acb_ptr w, acb_srcptr v, const acb_dft_naive_t t, slong prec)
Sets \( w \) to the DFT of \( v \) of size \( t->n \), using the naive algorithm data \( t \).

\textbf{CRT decomposition}

void acb_dft_crt(acb_ptr w, acb_srcptr v, slong n, slong prec)
Computes the DFT of \( v \) into \( w \), where \( v \) and \( w \) have size \( len \), using CRT to express \( \mathbb{Z}/n\mathbb{Z} \) as a product of cyclic groups.

acb_dft_crt_struct

acb_dft_crt_t

void acb_dft_crt_init(acb_dft_crt_t t, slong len, slong prec)

void acb_dft_crt_clear(acb_dft_crt_t t)
Initialize a CRT decomposition of \( \mathbb{Z}/n\mathbb{Z} \) as a direct product of cyclic groups. The length \( len \) is stored as \( t->n \).

void acb_dft_crt_precomp(acb_ptr w, acb_srcptr v, const acb_dft_crt_t t, slong prec)
Sets \( w \) to the DFT of \( v \) of size \( t->n \), using the CRT decomposition scheme \( t \).
Cooley-Tukey decomposition

```c
void acb_dft_cyc(acb_ptr w, acb_srcptr v, slong n, slong prec)
    Computes the DFT of v into w, where v and w have size n, using each prime factor of m of n to
decompose with the subgroup H = m\mathbb{Z}/n\mathbb{Z}.

acb_dft_cyc_struct
acb_dft_cyc_t

void acb_dft_cyc_init(acb_dft_cyc_t t, slong len, slong prec)
    Initialize a decomposition of \mathbb{Z}/n\mathbb{Z} into cyclic subgroups. The length len is stored as t->n.

void acb_dft_cyc_clear(acb_dft_cyc_t t)

void acb_dft_cyc_precomp(acb_ptr w, acb_srcptr v, const acb_dft_cyc_t t, slong prec)
    Sets w to the DFT of v of size t->n, using the cyclic decomposition scheme t.
```

Radix 2 decomposition

```c
void acb_dft_rad2(acb_ptr w, acb_srcptr v, int e, slong prec)
    Computes the DFT of v into w, where v and w have size 2^e, using a radix 2 FFT.

void acb_dft_inverse_rad2(acb_ptr w, acb_srcptr v, int e, slong prec)
    Computes the inverse DFT of v into w, where v and w have size 2^e, using a radix 2 FFT.

acb_dft_rad2_struct
acb_dft_rad2_t

void acb_dft_rad2_init(acb_dft_rad2_t t, int e, slong prec)

void acb_dft_rad2_clear(acb_dft_rad2_t t)
    Initialize and clear a radix 2 FFT of size 2^e, stored as t->n.

void acb_dft_rad2_precomp(acb_ptr w, acb_srcptr v, const acb_dft_rad2_t t, slong prec)
    Sets w to the DFT of v of size t->n, using the precomputed radix 2 scheme t.
```

Bluestein transform

```c
void acb_dft_bluestein(acb_ptr w, acb_srcptr v, slong n, slong prec)
    Computes the DFT of v into w, where v and w have size n, by conversion to a radix 2 one using
Bluestein’s convolution trick.

acb_dft_bluestein_struct
acb_dft_bluestein_t

Stores a Bluestein scheme for some length n : that is a acb_dft_rad2_t of size 2^e \geq 2n - 1 and a
size n array of convolution factors.

void acb_dft_bluestein_init(acb_dft_bluestein_t t, slong len, slong prec)

void acb_dft_bluestein_clear(acb_dft_bluestein_t t)
    Initialize and clear a Bluestein scheme to compute DFT of size len.

void acb_dft_bluestein_precomp(acb_ptr w, acb_srcptr v, const acb_dft_bluestein_t t, slong prec)
    Sets w to the DFT of v of size t->n, using the precomputed Bluestein scheme t.
```
These modules implement dense matrices with real and complex coefficients. Rudimentary linear algebra is supported.

**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.

**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.

**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.

`slong arb_mat_allocated_bytes(const arb_mat_t x)`  
Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(arb_mat_struct)` to get the size of the object as a whole.
Conversions

```c
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.

Random generation

```c
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`.

Input and output

```c
void arb_mat_printd (const arb_mat_t mat, slong digits)
void arb_mat_fprintd (FILE *, const arb_mat_t mat, slong digits)
```

Prints each entry in the matrix with the specified number of decimal digits.

Comparisons

```c
int arb_mat_equal  (const arb_mat_t mat1, const arb_mat_t mat2)
int arb_mat_overlaps(const arb_mat_t mat1, const arb_mat_t 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)
int arb_mat_eq (const arb_mat_t mat1, const arb_mat_t mat2)
int arb_mat_ne (const arb_mat_t mat1, const arb_mat_t mat2)
int arb_mat_is_empty(const arb_mat_t mat)
int arb_mat_is_square(const arb_mat_t mat)
```

Returns nonzero iff the matrices have the same dimensions and identical entries.

Returns nonzero iff the matrices have the same dimensions and each entry in `mat1` overlaps with the corresponding entry in `mat2`.

Returns nonzero iff the matrices have the same dimensions and each entry in `mat2` is contained in the corresponding entry in `mat1`.

Returns nonzero iff `mat1` and `mat2` certainly represent the same matrix.

Returns nonzero iff `mat1` and `mat2` certainly do not represent the same matrix.

Returns nonzero iff the number of rows or the number of columns in `mat` is zero.

Returns nonzero iff the number of rows is equal to the number of columns in `mat`.

Special matrices

```c
void arb_mat_zero (arb_mat_t mat)
void arb_mat_one (arb_mat_t mat)
```

Sets all entries in `mat` to zero.

Sets the entries on the main diagonal to ones, and all other entries to zero.
Transpose

void \texttt{arb\_mat\_transpose}(\texttt{arb\_mat\_t} \texttt{dest}, \texttt{const arb\_mat\_t} \texttt{src})

Sets \texttt{dest} to the exact transpose \texttt{src}. The operands must have compatible dimensions. Aliasing is allowed.

Norms

void \texttt{arb\_mat\_bound\_inf\_norm}(\texttt{mag\_t} \texttt{b}, \texttt{const arb\_mat\_t} \texttt{A})

Sets \texttt{b} to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of \texttt{A}.

void \texttt{arb\_mat\_frobenius\_norm}(\texttt{arb\_t} \texttt{res}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to the Frobenius norm (i.e. the square root of the sum of squares of entries) of \texttt{A}.

void \texttt{arb\_mat\_bound\_frobenius\_norm}(\texttt{mag\_t} \texttt{res}, \texttt{const arb\_mat\_t} \texttt{A})

Sets \texttt{res} to an upper bound for the Frobenius norm of \texttt{A}.

Arithmetic

void \texttt{arb\_mat\_neg}(\texttt{arb\_mat\_t} \texttt{dest}, \texttt{const arb\_mat\_t} \texttt{src})

Sets \texttt{dest} to the exact negation of \texttt{src}. The operands must have the same dimensions.

void \texttt{arb\_mat\_add}(\texttt{arb\_mat\_t} \texttt{res}, \texttt{const arb\_mat\_t} \texttt{mat1}, \texttt{const arb\_mat\_t} \texttt{mat2}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to the sum of \texttt{mat1} and \texttt{mat2}. The operands must have the same dimensions.

void \texttt{arb\_mat\_sub}(\texttt{arb\_mat\_t} \texttt{res}, \texttt{const arb\_mat\_t} \texttt{mat1}, \texttt{const arb\_mat\_t} \texttt{mat2}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to the difference of \texttt{mat1} and \texttt{mat2}. The operands must have the same dimensions.

void \texttt{arb\_mat\_mul\_classical}(\texttt{arb\_mat\_t} \texttt{C}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{const arb\_mat\_t} \texttt{B}, \texttt{slong} \texttt{prec})

void \texttt{arb\_mat\_mul\_threaded}(\texttt{arb\_mat\_t} \texttt{C}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{const arb\_mat\_t} \texttt{B}, \texttt{slong} \texttt{prec})

void \texttt{arb\_mat\_mul}(\texttt{arb\_mat\_t} \texttt{res}, \texttt{const arb\_mat\_t} \texttt{mat1}, \texttt{const arb\_mat\_t} \texttt{mat2}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to the matrix product of \texttt{mat1} and \texttt{mat2}. The operands must have compatible dimensions for matrix multiplication.

The \texttt{threaded} version splits the computation over the number of threads returned by \texttt{flint\_get\_num\_threads()}. The default version automatically calls the \texttt{threaded} version if the matrices are sufficiently large and more than one thread can be used.

void \texttt{arb\_mat\_mul\_entrywise}(\texttt{arb\_mat\_t} \texttt{C}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{const arb\_mat\_t} \texttt{B}, \texttt{slong} \texttt{prec})

Sets \texttt{C} to the entrywise product of \texttt{A} and \texttt{B}. The operands must have the same dimensions.

void \texttt{arb\_mat\_sqr\_classical}(\texttt{arb\_mat\_t} \texttt{B}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{slong} \texttt{prec})

void \texttt{arb\_mat\_sqr}(\texttt{arb\_mat\_t} \texttt{res}, \texttt{const arb\_mat\_t} \texttt{mat}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to the matrix square of \texttt{mat}. The operands must both be square with the same dimensions.

void \texttt{arb\_mat\_pow\_ui}(\texttt{arb\_mat\_t} \texttt{res}, \texttt{const arb\_mat\_t} \texttt{mat}, \texttt{ulong} \texttt{exp}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to \texttt{mat} raised to the power \texttt{exp}. Requires that \texttt{mat} is a square matrix.

Scalar arithmetic

void \texttt{arb\_mat\_scalar\_mul\_2exp\_si}(\texttt{arb\_mat\_t} \texttt{B}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{slong} \texttt{c})

Sets \texttt{B} to \texttt{A} multiplied by $2^c$.

void \texttt{arb\_mat\_scalar\_addmul\_si}(\texttt{arb\_mat\_t} \texttt{B}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{slong} \texttt{c}, \texttt{slong} \texttt{prec})

void \texttt{arb\_mat\_scalar\_addmul\_fmpz}(\texttt{arb\_mat\_t} \texttt{B}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{const fmpz\_t} \texttt{c}, \texttt{slong} \texttt{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 $A/c$.

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.

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 \texttt{arb\_mat\_inv\_ldl\_precomp}(\texttt{arb\_mat\_t} \ X, \texttt{const arb\_mat\_t} \ L, \texttt{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 \texttt{arb\_mat\_ldl()}. The inverse is calculated using the method of [Kri2013] which is more efficient than solving \( AX = I \) with \texttt{arb\_mat\_solve\_ldl\_precomp()}. 

Characteristic polynomial

void \_\texttt{arb\_mat\_charpoly}(\texttt{arb\_ptr} \ cp, \texttt{const arb\_mat\_t} \ mat, \texttt{slong} \ prec) \\
void \texttt{arb\_mat\_charpoly}(\texttt{arb\_poly\_t} \ cp, \texttt{const arb\_mat\_t} \ mat, \texttt{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.

Special functions

void \texttt{arb\_mat\_exp\_taylor\_sum}(\texttt{arb\_mat\_t} \ S, \texttt{const arb\_mat\_t} \ A, \texttt{slong} \ N, \texttt{slong} \ prec) \\
Sets \( S \) to the truncated exponential Taylor series \( S = \sum_{k=0}^{N-1} \frac{A^k}{k!} \). 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 \texttt{arb\_mat\_exp}(\texttt{arb\_mat\_t} \ B, \texttt{const arb\_mat\_t} \ A, \texttt{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

\[
\left\| \exp(2^{-r}A) - \sum_{k=0}^{N-1} \frac{(2^{-r}A)^k}{k!} \right\|_\infty \leq \sum_{k=N}^{\infty} \frac{(2^{-r}\|A\|_\infty)^k}{k!}.
\]

We bound the sum on the right using \texttt{mag\_exp\_tail()}. Truncation error is not added to entries whose values are determined by the sparsity structure of \( A \).

void \texttt{arb\_mat\_trace}(\texttt{arb\_t} \ trace, \texttt{const arb\_mat\_t} \ mat, \texttt{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.

Sparsity structure

void \texttt{arb\_mat\_entrywise\_is\_zero}(\texttt{fmpz\_mat\_t} \ dest, \texttt{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 \texttt{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 \texttt{arb\_mat\_entrywise\_not\_is\_zero}(\texttt{fmpz\_mat\_t} \ dest, \texttt{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 \texttt{arb\_mat\_entrywise\_is\_zero()}. 

\texttt{slong} \ \texttt{arb\_mat\_count\_is\_zero}(\texttt{const arb\_mat\_t} \ mat) \\
Returns the number of entries of \( mat \) that are certainly zero according to \texttt{arb\_is\_zero()}. 

\texttt{slong} \ \texttt{arb\_mat\_count\_not\_is\_zero}(\texttt{const arb\_mat\_t} \ mat) \\
Returns the number of entries of \( mat \) that are not certainly zero.
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.

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.

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.

`slong acb_mat_allocated_bytes(const acb_mat_t x)`

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(acb_mat_struct)` to get the size of the object as a whole.

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.

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`. 
Input and output

void \texttt{acb\_mat\_printd}(\texttt{const acb\_mat\_t mat, slong digits})
\hspace{1em}Prints each entry in the matrix with the specified number of decimal digits.

void \texttt{acb\_mat\_fprintd}(\texttt{FILE * file, const acb\_mat\_t mat, slong digits})
\hspace{1em}Prints each entry in the matrix with the specified number of decimal digits to the stream file.

Comparisons

\begin{itemize}
\item \texttt{int acb\_mat\_equal}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})
\hspace{1em}Returns nonzero iff the matrices have the same dimensions and identical entries.
\item \texttt{int acb\_mat\_overlaps}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})
\hspace{1em}Returns nonzero iff the matrices have the same dimensions and each entry in \textit{mat1} overlaps with the corresponding entry in \textit{mat2}.
\item \texttt{int acb\_mat\_contains}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})
\item \texttt{int acb\_mat\_contains\_fmpz\_mat}(\texttt{const acb\_mat\_t mat1, const fmpz\_mat\_t mat2})
\item \texttt{int acb\_mat\_contains\_fmpq\_mat}(\texttt{const acb\_mat\_t mat1, const fmpq\_mat\_t mat2})
\hspace{1em}Returns nonzero iff the matrices have the same dimensions and each entry in \textit{mat2} is contained in the corresponding entry in \textit{mat1}.
\item \texttt{int acb\_mat\_eq}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})
\hspace{1em}Returns nonzero iff \textit{mat1} and \textit{mat2} certainly represent the same matrix.
\item \texttt{int acb\_mat\_ne}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})
\hspace{1em}Returns nonzero iff \textit{mat1} and \textit{mat2} certainly do not represent the same matrix.
\item \texttt{int acb\_mat\_is\_real}(\texttt{const acb\_mat\_t mat})
\hspace{1em}Returns nonzero iff all entries in \textit{mat} have zero imaginary part.
\item \texttt{int acb\_mat\_is\_empty}(\texttt{const acb\_mat\_t mat})
\hspace{1em}Returns nonzero iff the number of rows or the number of columns in \textit{mat} is zero.
\item \texttt{int acb\_mat\_is\_square}(\texttt{const acb\_mat\_t mat})
\hspace{1em}Returns nonzero iff the number of rows is equal to the number of columns in \textit{mat}.
\end{itemize}

Special matrices

\begin{itemize}
\item \texttt{void acb\_mat\_zero}(\texttt{acb\_mat\_t mat})
\hspace{1em}Sets all entries in \textit{mat} to zero.
\item \texttt{void acb\_mat\_one}(\texttt{acb\_mat\_t mat})
\hspace{1em}Sets the entries on the main diagonal to ones, and all other entries to zero.
\end{itemize}

Transpose

\begin{itemize}
\item \texttt{void acb\_mat\_transpose}(\texttt{acb\_mat\_t dest, const acb\_mat\_t src})
\hspace{1em}Sets \textit{dest} to the exact transpose \textit{src}. The operands must have compatible dimensions. Aliasing is allowed.
\end{itemize}

Norms

\begin{itemize}
\item \texttt{void acb\_mat\_bound\_inf\_norm}(\texttt{mag\_t b, const acb\_mat\_t A})
\hspace{1em}Sets \textit{b} to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of \textit{A}.
\item \texttt{void acb\_mat\_frobenius\_norm}(\texttt{acb\_t res, const acb\_mat\_t A, slong prec})
\hspace{1em}Sets \textit{res} to the Frobenius norm (i.e. the square root of the sum of squares of entries) of \textit{A}.
\end{itemize}
void \texttt{acb\_mat\_bound\_frobenius\_norm}(\texttt{mag\_t res}, \texttt{const acb\_mat\_t A})

Sets \texttt{res} to an upper bound for the Froebius norm of \texttt{A}.

**Arithmetic**

void \texttt{acb\_mat\_neg}(\texttt{acb\_mat\_t dest}, \texttt{const acb\_mat\_t src})

Sets \texttt{dest} to the exact negation of \texttt{src}. The operands must have the same dimensions.

void \texttt{acb\_mat\_add}(\texttt{acb\_mat\_t res}, \texttt{const acb\_mat\_t mat1}, \texttt{const acb\_mat\_t mat2}, \texttt{slong prec})

Sets \texttt{res} to the sum of \texttt{mat1} and \texttt{mat2}. The operands must have the same dimensions.

void \texttt{acb\_mat\_sub}(\texttt{acb\_mat\_t res}, \texttt{const acb\_mat\_t mat1}, \texttt{const acb\_mat\_t mat2}, \texttt{slong prec})

Sets \texttt{res} to the difference of \texttt{mat1} and \texttt{mat2}. The operands must have the same dimensions.

void \texttt{acb\_mat\_mul}(\texttt{acb\_mat\_t res}, \texttt{const acb\_mat\_t mat1}, \texttt{const acb\_mat\_t mat2}, \texttt{slong prec})

Sets \texttt{res} to the matrix product of \texttt{mat1} and \texttt{mat2}. The operands must have compatible dimensions for matrix multiplication.

The \texttt{threaded} version splits the computation over the number of threads returned by \texttt{flint\_get\_num\_threads()}. The default version automatically calls the \texttt{threaded} version if the matrices are sufficiently large and more than one thread can be used.

void \texttt{acb\_mat\_mul\_entrywise}(\texttt{acb\_mat\_t res}, \texttt{const acb\_mat\_t mat1}, \texttt{const acb\_mat\_t mat2}, \texttt{slong prec})

Sets \texttt{res} to the entrywise product of \texttt{mat1} and \texttt{mat2}. The operands must have the same dimensions.

void \texttt{acb\_mat\_sqr}(\texttt{acb\_mat\_t res}, \texttt{const acb\_mat\_t mat}, \texttt{slong prec})

Sets \texttt{res} to the matrix square of \texttt{mat}. The operands must both be square with the same dimensions.

void \texttt{acb\_mat\_pow\_ui}(\texttt{acb\_mat\_t res}, \texttt{const acb\_mat\_t mat}, \texttt{ulong exp}, \texttt{slong prec})

Sets \texttt{res} to \texttt{mat} raised to the power \texttt{exp}. Requires that \texttt{mat} is a square matrix.

**Scalar arithmetic**

void \texttt{acb\_mat\_scalar\_mul\_2exp\_si}(\texttt{acb\_mat\_t B}, \texttt{const acb\_mat\_t A}, \texttt{slong c})

Sets \texttt{B} to \texttt{A} multiplied by \texttt{2^c}.

void \texttt{acb\_mat\_scalar\_addmul\_si}(\texttt{acb\_mat\_t B}, \texttt{const acb\_mat\_t A}, \texttt{slong c}, \texttt{slong prec})

void \texttt{acb\_mat\_scalar\_addmul\_fmpz}(\texttt{acb\_mat\_t B}, \texttt{const acb\_mat\_t A}, \texttt{const fmpz\_t c}, \texttt{slong prec})

void \texttt{acb\_mat\_scalar\_addmul\_arb}(\texttt{acb\_mat\_t B}, \texttt{const acb\_mat\_t A}, \texttt{const arb\_t c}, \texttt{slong prec})

Sets \texttt{B} to \texttt{B + A \times c}.

void \texttt{acb\_mat\_scalar\_mul\_si}(\texttt{acb\_mat\_t B}, \texttt{const acb\_mat\_t A}, \texttt{slong c}, \texttt{slong prec})

void \texttt{acb\_mat\_scalar\_mul\_fmpz}(\texttt{acb\_mat\_t B}, \texttt{const acb\_mat\_t A}, \texttt{const fmpz\_t c}, \texttt{slong prec})

void \texttt{acb\_mat\_scalar\_mul\_arb}(\texttt{acb\_mat\_t B}, \texttt{const acb\_mat\_t A}, \texttt{const arb\_t c}, \texttt{slong prec})

Sets \texttt{B} to \texttt{A \times c}.

void \texttt{acb\_mat\_scalar\_div\_si}(\texttt{acb\_mat\_t B}, \texttt{const acb\_mat\_t A}, \texttt{slong c}, \texttt{slong prec})

void \texttt{acb\_mat\_scalar\_div\_fmpz}(\texttt{acb\_mat\_t B}, \texttt{const acb\_mat\_t A}, \texttt{const fmpz\_t c}, \texttt{slong prec})

void \texttt{acb\_mat\_scalar\_div\_arb}(\texttt{acb\_mat\_t B}, \texttt{const acb\_mat\_t A}, \texttt{const arb\_t c}, \texttt{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 $A/c$.

**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.

**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.

**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.
These modules implement mathematical functions with complexity that goes beyond the basics covered directly in the \textit{arb} and \textit{acb} modules.

\textbf{acb_hypgeom.h – hypergeometric functions of complex variables}

The generalized hypergeometric function is formally defined by

\[
pFq(a_1, \ldots, a_p; b_1, \ldots, b_q; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \ldots (a_p)_k}{(b_1)_k \ldots (b_q)_k} \frac{z^k}{k!}.
\]

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.

\textbf{Convergent series}

In this section, we define

\[
T(k) = \prod_{i=0}^{p-1} (a_i)_k \frac{z^k}{(b_i)_k}
\]

and

\[
pFq(a_0, \ldots, a_{p-1}; b_1 \ldots b_{q-1}; z) = p+1Fq(a_0, \ldots, a_{p-1}, 1; b_0 \ldots b_{q-1}; z) = \sum_{k=0}^{\infty} T(k)
\]

For the conventional generalized hypergeometric function \(pFq\), compute \(pFq+1\) with the explicit parameter \(b_q = 1\), or remove a 1 from the \(a_i\) parameters if there is one.

\begin{verbatim}
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 \(|\sum_{k=n}^{\infty} T(k)| \leq C |T(n)|\). 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.
}
\end{verbatim}
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 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_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.

```c
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 $p f_q(z)$ 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.

**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*.

```c
void acb_hypgeom_u_asympt(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 ≥ 0`.

```c
int acb_hypgeom_u_use_asympt(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).

**Generalized hypergeometric function**

```c
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 $p F_q(z)$, 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.

**Confluent hypergeometric functions**

```c
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.

```c
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 \texttt{acb\_hypgeom\_u}(\texttt{acb\_t res}, \texttt{const acb\_t a}, \texttt{const acb\_t b}, \texttt{const acb\_t z}, \texttt{slong prec})
Computes $U(a,b,z)$ using an automatic algorithm choice. The function \texttt{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 \texttt{acb\_hypgeom\_u\_1f1()} is used.

void \texttt{acb\_hypgeom\_m\_asymp}(\texttt{acb\_t res}, \texttt{const acb\_t a}, \texttt{const acb\_t b}, \texttt{const acb\_t z}, \texttt{int regularized}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_1f1}(\texttt{acb\_t res}, \texttt{const acb\_t a}, \texttt{const acb\_t b}, \texttt{const acb\_t z}, \texttt{int regularized}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_m}(\texttt{acb\_t res}, \texttt{const acb\_t a}, \texttt{const acb\_t b}, \texttt{const acb\_t z}, \texttt{int regularized}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_m\_1f1}(\texttt{acb\_t res}, \texttt{const acb\_t a}, \texttt{const acb\_t b}, \texttt{const acb\_t z}, \texttt{int regularized}, \texttt{slong prec})

Computes the confluent hypergeometric function $M(a,b,z) = {}_1F_1(a,b,z)$, or $M(a,b,z) = \Gamma(b) {}_1F_1(a,b,z)$ if regularized is set.

void \texttt{acb\_hypgeom\_1f1\_asymp}(\texttt{acb\_t res}, \texttt{const acb\_t a}, \texttt{const acb\_t b}, \texttt{const acb\_t z}, \texttt{int regularized}, \texttt{slong prec})

Alias for \texttt{acb\_hypgeom\_m\_1f1()}.

void \texttt{acb\_hypgeom\_0f1\_asymp}(\texttt{acb\_t res}, \texttt{const acb\_t a}, \texttt{const acb\_t z}, \texttt{int regularized}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_0f1\_direct}(\texttt{acb\_t res}, \texttt{const acb\_t a}, \texttt{const acb\_t z}, \texttt{int regularized}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_0f1}(\texttt{acb\_t res}, \texttt{const acb\_t a}, \texttt{const acb\_t z}, \texttt{int regularized}, \texttt{slong prec})

Computes the confluent hypergeometric function $a {}_1F_1(a,z)$, or $\frac{1}{\Gamma(a)} {}_1F_1(a,z)$ if regularized is set, using asymptotic expansions, direct summation, or an automatic algorithm choice. The \texttt{asymp} version uses the asymptotic expansions of Bessel functions, together with the connection formulas

\[
\frac{a {}_1F_1(a,z)}{\Gamma(a)} = (-z)^{(1-a)/2} J_{a-1}(2 \sqrt{-z}) = z^{(1-a)/2} I_{a-1}(2 \sqrt{z}).
\]

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.

**Error functions and Fresnel integrals**

void \texttt{acb\_hypgeom\_erf\_propagated\_error}(\texttt{mag\_t re}, \texttt{mag\_t im}, \texttt{const acb\_t z})
Sets $\texttt{re}$ and $\texttt{im}$ 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 \texttt{acb\_hypgeom\_erf\_1f1\_a}(\texttt{acb\_t res}, \texttt{const acb\_t z}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_erf\_1f1\_b}(\texttt{acb\_t res}, \texttt{const acb\_t z}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_erf\_asymp}(\texttt{acb\_t res}, \texttt{const acb\_t z}, \texttt{int complementary}, \texttt{slong prec}, \texttt{slong prec}2)

Computes the error function respectively using

\[
erf(z) = \frac{2z}{\sqrt{\pi}} {}_1F_1\left(\frac{1}{2}, \frac{3}{2}, -z^2\right)
erf(z) = \frac{2ze^{-z^2}}{\sqrt{\pi}} {}_1F_1\left(1, \frac{3}{2}, z^2\right)
erf(z) = \frac{z}{\sqrt{2\pi}} \left(1 - \frac{e^{-z^2}}{\sqrt{\pi}} {}_2F_1\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, z^2\right)\right) = \frac{z}{\sqrt{2\pi}} \left(1 - \frac{e^{-z^2}}{\sqrt{\pi}} {}_2F_1\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, z^2\right)\right).
\]

The \texttt{asymp} version takes a second precision to use for the $U$ term. It also takes an extra flag \texttt{complementary}, computing the complementary error function if set.

void \texttt{acb\_hypgeom\_erf}(\texttt{acb\_t res}, \texttt{const acb\_t z}, \texttt{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()`.

```c
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`.

```c
void acb_hypgeom_erfc(acb_t res, const acb_t z, slong prec)
```

Computes the complementary error function `erfc(z) = 1 - erf(z)`. This function avoids catastrophic cancellation for large positive `z`.

```c
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`.

```c
void acb_hypgeom_erfi(acb_t res, const acb_t z, slong prec)
```

Computes the imaginary error function `erfi(z) = -i erf(iz)`. This is a trivial wrapper of `acb_hypgeom_erf()`.

```c
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`.

```c
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.

```c
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.

### Bessel functions

```c
void acb_hypgeom_bessel_j_asympt(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_asympt()`. For all complex `\nu`, `z`, we have

\[ J_\nu(z) = \frac{z^n}{2^{\nu+1}} \Gamma(n+1) F_1(n+\frac{1}{2},2n+1,2iz) = A_+ B_+ + A_- B_- \]

where

\[ A_\pm = z^{-\nu}(z^2 - 1)^{-\nu/2}(\pm iz)^{\nu} (2\pi)^{-1/2} = (\pm iz)^{-1/2-\nu} z^{\nu} (2\pi)^{-1/2} \]

\[ B_\pm = e^{\pm iz} U^\nu(n+\frac{1}{2},2n+1,\mp 2iz). \]

Nicer representations of the factors `A_\pm` can be given depending conditionally on the parameters. If \( n + \frac{1}{2} = \pi \in \mathbb{Z} \), we have `A_\pm = (\pm i)^n (2\pi z)^{-1/2}`. And if `Re(z) > 0`, we have `A_\pm = \exp(\pm i(2n + 1)/4\pi)(2\pi z)^{-1/2}`.

```c
void acb_hypgeom_bessel_j_0fi(acb_t res, const acb_t nu, const acb_t z, slong prec)
```

Computes the Bessel function of the first kind from

\[ J_\nu(z) = \frac{1}{\Gamma(n+1)} \left( \frac{z}{2} \right)^\nu n F_1 \left( \nu + 1, -\frac{z^2}{4} \right). \]
void \texttt{acb_hypgeom_bessel\_j}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{slong prec})

Computes the Bessel function of the first kind $J_\nu(z)$ using an automatic algorithm choice.

void \texttt{acb_hypgeom_bessel\_y}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{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

$$Y_n(z) = -\frac{2}{\pi} (\nu^n K_n(iz) + [\log(iz) - \log(z)] J_n(z))$$

is computed. As currently implemented, the output is indeterminate if $\nu$ is nonexact and contains an integer.

void \texttt{acb_hypgeom_bessel\_jy}(\texttt{acb\_t res1}, \texttt{acb\_t res2}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{slong prec})

Sets \texttt{res1} to $J_\nu(z)$ and \texttt{res2} to $Y_\nu(z)$, computed simultaneously. From these values, the user can easily construct the Bessel functions of the third kind (Hankel functions) $H^{(1)}_\nu(z), H^{(2)}_\nu(z) = J_\nu(z) \pm iY_\nu(z)$.

void \texttt{acb_hypgeom_bessel\_i\_asymp}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{slong prec})

void \texttt{acb_hypgeom_bessel\_i\_0f1}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{slong prec})

void \texttt{acb_hypgeom_bessel\_i}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{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 \texttt{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 \texttt{acb_hypgeom_bessel\_k\_asymp}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{slong prec})

Computes the modified Bessel function of the second kind via \texttt{acb_hypgeom\_u\_asymp()}.

For all $\nu$ and all $z \neq 0$, we have

$$K_\nu(z) = \left( \frac{2z}{\pi} \right)^{-1/2} e^{-z} U^* (\nu + \frac{1}{2}, 2\nu + 1, 2z).$$

void \texttt{acb_hypgeom_bessel\_k\_0f1\_series}(\texttt{acb\_poly\_t res}, \texttt{const acb\_poly\_t nu}, \texttt{const acb\_poly\_t z}, \texttt{const slong len}, \texttt{slong prec})

Computes the modified Bessel function of the second kind $K_\nu(z)$ as a power series truncated to length $\texttt{len}$, given $\nu, z \in \mathbb{C}[\llbracket x \rrbracket]$. Uses the formula

$$K_\nu(z) = \frac{1}{2} \frac{\pi}{\sin(\pi\nu)} \left[ \left( \frac{z}{2} \right)^{-\nu} {}_0F_1 \left( 1 - \nu, \frac{z^2}{4} \right) - \left( \frac{z}{2} \right)^{\nu} {}_0F_1 \left( 1 + \nu, \frac{z^2}{4} \right) \right].$$

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 \texttt{acb_hypgeom_bessel\_k\_0f1}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{slong prec})

Computes the modified Bessel function of the second kind $K_\nu(z)$

$$K_\nu(z) = \frac{1}{2} \left[ \left( \frac{z}{2} \right)^{-\nu} \Gamma(\nu) {}_0F_1 \left( 1 - \nu, \frac{z^2}{4} \right) - \left( \frac{z}{2} \right)^{\nu} \frac{\pi}{\nu \sin(\pi\nu) \Gamma(\nu)} {}_0F_1 \left( \nu + 1, \frac{z^2}{4} \right) \right],$$

if $\nu \notin \mathbb{Z}$. If $\nu \in \mathbb{Z}$, it computes the limit value via \texttt{acb_hypgeom\_bessel\_k\_0f1\_series()}. As currently implemented, the output is indeterminate if $\nu$ is nonexact and contains an integer.

void \texttt{acb_hypgeom_bessel\_k}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{slong prec})

Computes the modified Bessel function of the second kind $K_\nu(z)$ using an automatic algorithm choice.
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 \( Ai(z) \), \( Bi(z) \). 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 \( (Ai(z), Ai'(z), Bi(z), Bi'(z)) \) simultaneously. Any of the four function values can be omitted by passing \( NULL \) for the unwanted output variables, speeding up the evaluation.

```c
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.

```c
void acb_hypgeom_airy_asym(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*.

```c
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_asym()` with \( n = 1 \).

```c
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_asym()` 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()`.

```c
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()`.

```c
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)
```
```c
void acb_hypgeom_airy_series(acb_poly_t ai, acb_poly_t ai_prime, acb_poly_t bi, acb_poly_t bi_prime, 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 \( NULL \).
Incomplete gamma and beta functions

```c
void acb_hypgeom_gamma_upper_asympt(acb_t res, const acb_t s, const acb_t z, int regularized, slong prec)
void acb_hypgeom_gamma_upper_if1a(acb_t res, const acb_t s, const acb_t z, int regularized, slong prec)
void acb_hypgeom_gamma_upper_if1b(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 $Q(s,z) = \frac{\Gamma(s,z)}{\Gamma(s)}$.

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

\[
\Gamma(s,z) = e^{-z}U(1-s,1-s, z)
\]

\[
\Gamma(s,z) = \Gamma(s) - \frac{z^s}{s} F_1(s, s + 1, -z)
\]

\[
\Gamma(s,z) = \frac{z^s e^{-z}}{s} F_1(1, s + 1, z)
\]

\[
\Gamma(s,z) = \frac{(-1)^n}{n!} \left(\psi(n + 1) - \log(z)\right) + \frac{(-1)^n}{(n + 1)!} z^2 F_2(1, 1, 2, 2 + n, -z) - z^{-n} \sum_{k=0}^{n-1} \frac{(-z)^k}{(k-n)k!}
\]

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.

```c
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 $\gamma(s,z) = \frac{z^s}{s} F_1(s, s + 1, -z)$.

If `regularized` is 1, computes the regularized lower incomplete gamma function $P(s,z) = \frac{\gamma(s,z)}{\Gamma(s)}$.

If `regularized` is 2, computes a further regularized lower incomplete gamma function $\gamma^*(s,z) = z^{-s}P(s,z)$.

```c
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 \( B(a, b; z) = \int_0^z t^{a-1}(1-t)^{b-1} \, dt \), 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

\[
B(a, b; z) = \frac{z^a}{\Gamma(b)} \, _2F_1(1-b, a+1, z)
\]

\[
I(a, b; z) = \frac{\Gamma(a+b)}{\Gamma(b)} \, z^a \, _2F_1(1-b, a+1, z).
\]

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_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.

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_asym(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 \( Ei(z) \), respectively using

\[
Ei(z) = -e^z U(1, 1, -z) - \log(-z) + \frac{1}{2} \left( \log(z) - \log \left( \frac{1}{z} \right) \right)
\]

\[
Ei(z) = z_2 F_2(1, 1; 2, 2; z) + \gamma + \frac{1}{2} \left( \log(z) - \log \left( \frac{1}{z} \right) \right)
\]

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_asym(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 $\Si(z)$, respectively using
\[
\begin{align*}
\Si(z) &= \frac{i}{2} \left[ e^{iz} U(1, 1, 1 - iz) - e^{-iz} U(1, 1, iz) + \log(-iz) - \log(iz) \right] \\
\Si(z) &= z_1 F_2 (\frac{1}{2}, \frac{3}{2}; -\frac{z^2}{4})
\end{align*}
\]
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_asym(p(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 $\Ci(z)$, respectively using
\[
\begin{align*}
\Ci(z) &= \log(z) - \frac{1}{2} \left[ e^{iz} U(1, 1, 1 - iz) + e^{-iz} U(1, 1, iz) + \log(-iz) + \log(iz) \right] \\
\Ci(z) &= -\frac{z^2}{4} F_2 (1, 1; 2, 2; -\frac{z^2}{4}) + \log(z) + \gamma
\end{align*}
\]
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)
void acb_hypgeom_shi_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)
Computes the hyperbolic sine integral $\Shi(z) = -i \Si(iz)$. 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_asym(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 $\Chii(z)$, respectively using
\[
\begin{align*}
\Chii(z) &= -\frac{1}{2} \left[ e^{iz} U(1, 1, -z) + e^{-iz} U(1, 1, z) + \log(-z) - \log(z) \right] \\
\Chii(z) &= \frac{z^2}{4} F_2 (1, 1; 2, 2; \frac{z^2}{4}) + \log(z) + \gamma
\end{align*}
\]
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 $\Li(z) = \Ei(\log(z))$.
If offset is nonzero, computes the offset logarithmic integral $\Li(z) = \Li(z) - \Li(2)$. 

138 Chapter 8. Higher mathematical functions
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 \( F(z) = F(a, b, c, z) = 2F_1(a, b, c, z) / \Gamma(c) \) if the flag `regularized` is set.

**Gauss hypergeometric function**

The following bit fields can be set in

```
acb_t flags;
```

- `flags & 1` for \( F(z) = F(a, b, c, z) \) using direct summation of the hypergeometric series.
- `flags & 2` for \( F(z) = F(a, b, c, z) \) using direct summation of the hypergeometric series.
- `flags & 4` for \( F(z) = F(a, b, c, z) \) using an argument transformation determined by the flag `which`.
- `flags & 8` for \( F(z) = F(a, b, c, z) \) 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`.

**Examples**

- `acb_hypgeom_2f1_series_direct(a, b, c, z, len, regularized, flags)`
- `acb_hypgeom_2f1_continuation(a, b, c, z)`
- `acb_hypgeom_2f1_corner(a, b, c, z)`
- `acb_hypgeom_2f1_transform(a, b, c, z, flags)`
- `acb_hypgeom_2f1_choose(a, b, c)`
- `acb_hypgeom_2f1(a, b, c, z, flags)`
• **ACB_HYPGEOM_2F1_REGULARIZED** - computes the regularized hypergeometric function $F(z)$. 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 $\frac{1}{2}F_1(\sqrt{2}, 1/2, \sqrt{2} + 3/2, 9/10)$, 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.

### Orthogonal polynomials and functions

**void acb_hypgeom_chebyshev_t(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)

$$C_n^m(z) = \frac{(2m)_n}{\Gamma(n + 1)}_2F_1 \left(-n, 2m + n, m + 1, \frac{1 - z}{2} \right).$$

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 \texttt{acb_hypgeom_laguerre\_l}(\texttt{acb\_t res}, \texttt{const acb\_t n}, \texttt{const acb\_t m}, \texttt{const acb\_t z}, \texttt{slong prec})

Computes the Laguerre polynomial (or Laguerre function)
\[
L_n^m(z) = \frac{(m + 1)_n}{\Gamma(n + 1)} F_1 \left( -n, m + 1, z \right).
\]

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 \(L_n^0(z) = e^z L_n^0(-z)\). Another possibility is to cover this case with the recurrence relation \(L_n^m(z) + L_{n-1}^m(z) = L_n^m(z)\). Currently, we leave this case undefined (returning indeterminate).

void \texttt{acb_hypgeom_hermite\_h}(\texttt{acb\_t res}, \texttt{const acb\_t n}, \texttt{const acb\_t z}, \texttt{slong prec})

Computes the Hermite polynomial (or Hermite function)
\[
H_n(z) = 2^n \sqrt{\pi} \left( \frac{1}{(1 - n/2)!} F_1 \left( -\frac{n}{2}, 1, z \right) - \frac{2z}{\Gamma(-n/2)} F_1 \left( \frac{1 - n}{2}, \frac{3}{2}, z^2 \right) \right).
\]

void \texttt{acb_hypgeom_legendre\_p}(\texttt{acb\_t res}, \texttt{const acb\_t n}, \texttt{const acb\_t m}, \texttt{const acb\_t z}, \texttt{int type}, \texttt{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
\[
P_n^m(z) = \frac{(1 + z)^{m/2}}{(1 - z)^{m/2}} F \left( -n, n + 1, 1 - m, \frac{1 - z}{2} \right)
\]
is computed, and if \(type\) is 1, the alternative version
\[
P_n^m(z) = \frac{(z + 1)^{m/2}}{(z - 1)^{m/2}} F \left( -n, n + 1, 1 - m, \frac{1 - z}{2} \right).
\]
is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in \textit{Mathematica} and \textit{mpmath}.

void \texttt{acb_hypgeom_legendre\_q}(\texttt{acb\_t res}, \texttt{const acb\_t n}, \texttt{const acb\_t m}, \texttt{const acb\_t z}, \texttt{int type}, \texttt{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
\[
Q_n^m(z) = \frac{\pi}{2 \sin(\pi m)} e^{\pi im} \left( P_n^m(z) - \frac{\Gamma(1 + m + n)}{\Gamma(1 - m + n)} P_n^{-m}(z) \right)
\]
is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in \textit{Mathematica} and \textit{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.
(\textit{The formula} \([WQ3d]\) \textit{would be useful, but is incorrect in the lower half plane}.)

void \texttt{acb_hypgeom_legendre\_p\_uiui\_rec}(\texttt{acb\_t res}, \texttt{ulong n}, \texttt{ulong m}, \texttt{const acb\_t z}, \texttt{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 \(z\).
The dilogarithm function is given by

\[ \text{Dilogarithm} \]

\[ \text{void acb_hypgeom_dilog}(\text{acb}_t \text{ res}, \text{const acb}_t \text{ theta}, \text{const acb}_t \text{ phi}, \text{slong prec}) \]

Computes the spherical harmonic of degree \( n \), order \( m \), latitude angle \( \theta \), and longitude angle \( \phi \), normalized such that

\[ Y_n^m(\theta, \phi) = \frac{2n + 1}{4\pi} \frac{(n - m)!}{(n + m)!} \sum \text{imag} P_n^m(\cos(\theta)). \]

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 \text{acb_hypgeom_legendre_p_uiui_rec()}. 

### Dilogarithm

The dilogarithm function is given by \( \text{Li}_2(z) = - \int_0^z \frac{\log(1-t)}{t} dt \). 

\[ \text{void acb_hypgeom_dilog_bernoulli}(\text{acb}_t \text{ res}, \text{const acb}_t \text{ z}, \text{slong prec}) \]

Computes the dilogarithm using a series expansion in \( w = \log(z) \), with rate of convergence \(|w/(2n)|^n\). This provides good convergence near \( z = e^{i\pi/3} \), where hypergeometric series expansions fail. Since the coefficients involve Bernoulli numbers, this method should only be used at moderate precision.

\[ \text{void acb_hypgeom_dilog_zero_taylor}(\text{acb}_t \text{ res}, \text{const acb}_t \text{ z}, \text{slong prec}) \]

Computes the dilogarithm for \( z \) close to 0 using the hypergeometric series (effective only when \(|z| \ll 1\)).

\[ \text{void acb_hypgeom_dilog_zero}(\text{acb}_t \text{ res}, \text{const acb}_t \text{ z}, \text{slong prec}) \]

Computes the dilogarithm for \( z \) close to 0, using the bit-burst algorithm instead of the hypergeometric series directly at very high precision.

\[ \text{void acb_hypgeom_dilog_transform}(\text{acb}_t \text{ res}, \text{const acb}_t \text{ z}, \text{int algorithm}, \text{slong prec}) \]

Computes the dilogarithm by applying one of the transformations \( 1/z, 1-z, z/(z-1), 1/(1-z) \), indexed by \text{algorithm} from 1 to 4, and calling \text{acb_hypgeom_dilog_zero()} with the reduced variable. Alternatively, for \text{algorithm} between 5 and 7, starts from the respective point \pm i, (1 \pm i)/2, (1 \pm i)/2 (with the sign chosen according to the midpoint of \( z \)) and computes the dilogarithm by the bit-burst method.

\[ \text{void acb_hypgeom_dilog_continuation}(\text{acb}_t \text{ res}, \text{const acb}_t \text{ z}, \text{const acb}_t \text{ a}, \text{const acb}_t \text{ z}, \text{slong prec}) \]

Computes \( \text{Li}_2(z) - \text{Li}_2(a) \) using Taylor expansion at \( a \). Binary splitting is used. Both \( a \) and \( z \) should be well isolated from the points 0 and 1, except that \( a \) may be exactly 0. If the straight line path from \( a \) to \( b \) crosses the branch cut, this method provides continuous analytic continuation instead of computing the principal branch.

\[ \text{void acb_hypgeom_dilog_bitburst}(\text{acb}_t \text{ res}, \text{acb}_t \text{ z0}, \text{const acb}_t \text{ z}, \text{slong prec}) \]

Sets \( z0 \) to a point with short bit expansion close to \( z \) and sets \text{res} to \( \text{Li}_2(z) - \text{Li}_2(z0) \), computed using the bit-burst algorithm.

\[ \text{void acb_hypgeom_dilog}(\text{acb}_t \text{ res}, \text{const acb}_t \text{ z}, \text{slong prec}) \]

Computes the dilogarithm using a default algorithm choice.

### arb_hypgeom.h – hypergeometric functions of real variables

See \text{arb_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 \text{arb}_t and \text{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.
**Generalized hypergeometric function**

```c
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.

**Confluent hypergeometric functions**

```c
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.

```c
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 \( M(a, b, z) = \frac{1}{\Gamma(b)} _1F_1(a, b, z) \) if `regularized` is set.

```c
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()`.

```c
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 \( U(a, b, z) \).

**Gauss hypergeometric function**

```c
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 \( F(a, b, c, z) = \frac{1}{\Gamma(c)} _2F_1(a, b, c, z) \) if `regularized` is set.

Additional evaluation flags can be passed via the `regularized` argument; see `acb_hypgeom_2f1()` for documentation.

**Error functions and Fresnel integrals**

```c
void arb_hypgeom_erf(arb_t res, const arb_t z, slong prec)
```

Computes the error function \( \text{erf}(z) \).

```c
void _arb_hypgeom_erf_series(arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

Computes the error function of the power series \( z \), truncated to length \( len \).

```c
void arb_hypgeom_erfc(arb_t res, const arb_t z, slong prec)
```

Computes the complementary error function \( \text{erfc}(z) = 1 - \text{erf}(z) \). This function avoids catastrophic cancellation for large positive \( z \).

```c
void _arb_hypgeom_erfc_series(arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

Computes the complementary error function of the power series \( z \), truncated to length \( len \).

```c
void arb_hypgeom_erfi(arb_t res, const arb_t z, slong prec)
```

Computes the imaginary error function \( \text{erfi}(z) \).

```c
void _arb_hypgeom_erfi_series(arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

Computes the imaginary error function of the power series \( z \), truncated to length \( len \).
Incomplete gamma and beta functions

void _arb_hypgeom_fresnel_series(arb_ptr res1, arb_ptr res2, arb_srcptr z, slong zlen,
    int normalized, slong len, 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_beta_lower_series(arb_t res1, arb_t res, const arb_t a, const arb_t z, int normalized, slong prec)

Sets res1 to 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 $I(a, b; z) = B(a, b; z)/B(a, b; 1)$. The definition is 0, computes the upper incomplete gamma function $\Gamma(s)$. If normalized is 1, computes the regularized upper incomplete gamma function $Q(s, z) = \Gamma(s, z)/\Gamma(s)$. If normalized 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; _arb_hypgeom_expint() is the intended interface for computing the exponential integral).

void arb_hypgeom_gamma_upper_series(arb_poly_t res1, arb_poly_t res2, const arb_poly_t z,
    int normalized, slong len, slong prec)

Sets res1 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 arb_hypgeom_beta_lower().
void arb_hypgeom_beta_lower_series(const arb_poly_t res, const arb_t a, const arb_t b, const arb_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.

**Exponential and trigonometric integrals**

void arb_hypgeom_expint(const arb_t res, const arb_t s, const arb_t z, slong prec)
Computes the generalized exponential integral \( E_s(z) \).

void arb_hypgeom_ei(const arb_t res, const arb_t z, slong prec)
Computes the exponential integral \( \text{Ei}(z) \).

void _arb_hypgeom_ei_series(const arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
Computes the exponential integral of the power series \( z \), truncated to length \( len \).

void arb_hypgeom_si(const arb_t res, const arb_t z, slong prec)
Computes the sine integral \( \text{Si}(z) \).

void _arb_hypgeom_si_series(const arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
Computes the sine integral of the power series \( z \), truncated to length \( len \).

void arb_hypgeom_ci(const arb_t res, const arb_t z, slong prec)
Computes the cosine integral \( \text{Ci}(z) \). The result is indeterminate if \( z < 0 \) since the value of the function would be complex.

void _arb_hypgeom_ci_series(const arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
Computes the cosine integral of the power series \( z \), truncated to length \( len \).

void arb_hypgeom_shi(const arb_t res, const arb_t z, slong prec)
Computes the hyperbolic sine integral \( \text{Shi}(z) = -i \text{Si}(iz) \).

void _arb_hypgeom_shi_series(const arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
Computes the hyperbolic sine integral of the power series \( z \), truncated to length \( len \).

void arb_hypgeom_chi(const arb_t res, const arb_t z, slong prec)
Computes the hyperbolic cosine integral \( \text{Chi}(z) \). The result is indeterminate if \( z < 0 \) since the value of the function would be complex.

void _arb_hypgeom_chi_series(const arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
Computes the hyperbolic cosine integral of the power series \( z \), truncated to length \( len \).

void arb_hypgeom_li(const arb_t res, const arb_t z, int offset, slong prec)
If \( offset \) is zero, computes the logarithmic integral \( \text{li}(z) = \text{Ei}(\log(z)) \).
If \( offset \) is nonzero, computes the offset logarithmic integral \( \text{Li}(z) = \text{li}(z) - \text{li}(2) \).
The result is indeterminate if \( z < 0 \) since the value of the function would be complex.

void _arb_hypgeom_li_series(const arb_ptr res, arb_srcptr z, slong zlen, int offset, slong len, slong prec)
Computes the logarithmic integral (optionally the offset version) of the power series \( z \), truncated to length \( len \).
Bessel functions

void arb_hypgeom_bessel_j (arb_t res, const arb_t nu, const arb_t z, slong prec)
Computes the Bessel function of the first kind $J_{\nu}(z)$.

void arb_hypgeom_bessel_y (arb_t res, const arb_t nu, const arb_t z, slong prec)
Computes the Bessel function of the second kind $Y_{\nu}(z)$.

void arb_hypgeom_bessel_jy (arb_t res1, arb_t res2, const arb_t nu, const arb_t z, slong prec)
Sets res1 to $J_{\nu}(z)$ and res2 to $Y_{\nu}(z)$, computed simultaneously.

void arb_hypgeom_bessel_i (arb_t res, const arb_t nu, const arb_t z, slong prec)
Computes the modified Bessel function of the first kind $I_{\nu}(z)$.

void arb_hypgeom_bessel_k (arb_t res, const arb_t nu, const arb_t z, slong prec)
Computes the modified Bessel function of the second kind $K_{\nu}(z)$.

Airy functions

void arb_hypgeom_airy (arb_t ai, arb_t ai_prime, arb_t bi, arb_t bi_prime, const arb_t z, slong prec)
Computes the Airy functions $(Ai(z), Ai'(z), Bi(z), Bi'(z))$ simultaneously. Any of the four function values can be omitted by passing NULL for the unwanted output variables, speeding up the evaluation.

void arb_hypgeom_airy_jet (arb_ptr ai, arb_ptr bi, const arb_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 _arb_poly_derivative().

void _arb_hypgeom_airy_series (arb_ptr ai, arb_ptr ai_prime, arb_ptr bi, arb_ptr bi_prime, arb_poly_t z, slong zlen, 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.

Orthogonal polynomials and functions

void arb_hypgeom_chebyshev_t (arb_t res, const arb_t nu, const arb_t z, slong prec)
void arb_hypgeom_chebyshev_u (arb_t res, const arb_t nu, const arb_t z, slong prec)
void arb_hypgeom_jacobi_p (arb_t res, const arb_t n, const arb_t a, const arb_t b, const arb_t z, slong prec)
void arb_hypgeom_gegenbauer_c (arb_t res, const arb_t n, const arb_t m, const arb_t z, slong prec)
void arb_hypgeom_laguerre_l (arb_t res, const arb_t n, const arb_t m, const arb_t z, slong prec)
void arb_hypgeom_hermite_h (arb_t res, const arb_t nu, const arb_t z, slong prec)
Computes Chebyshev, Jacobi, Gegenbauer, Laguerre or Hermite polynomials, or their extensions to non-integer orders.

void arb_hypgeom_legendre_p (arb_t res, const arb_t n, const arb_t m, const arb_t z, int type, slong prec)
void \texttt{arb\_hypgeom\_legendre\_p\_ui\_deriv\_bound}(\texttt{mag\_t dp, mag\_t dp2, ulong n, const arb\_t x, const arb\_t x2sub1})

Sets \(dp\) to an upper bound for \(P'_n(x)\) and \(dp2\) to an upper bound for \(P''_n(x)\) given \(x\) assumed to represent a real number with \(|x| \leq 1\). The variable \(x2sub1\) must contain the precomputed value \(1 - x^2\) (or \(x^2 - 1\)). This method is used internally to bound the propagated error for Legendre polynomials.

void \texttt{arb\_hypgeom\_legendre\_p\_ui\_zero}(\texttt{arb\_t res, arb\_t res\_prime, ulong n, const arb\_t x, slong K, slong prec})

void \texttt{arb\_hypgeom\_legendre\_p\_ui\_one}(\texttt{arb\_t res, arb\_t res\_prime, ulong n, const arb\_t x, slong K, slong prec})

void \texttt{arb\_hypgeom\_legendre\_p\_ui\_asymp}(\texttt{arb\_t res, arb\_t res\_prime, ulong n, const arb\_t x, slong K, slong prec})

void \texttt{arb\_hypgeom\_legendre\_p\_rec}(\texttt{arb\_t res, arb\_t res\_prime, ulong n, const arb\_t x, slong prec})

void \texttt{arb\_hypgeom\_legendre\_p\_ui}(\texttt{arb\_t res, arb\_t res\_prime, ulong n, const arb\_t x, slong prec})

Evaluates the ordinary Legendre polynomial \(P_n(x)\). If \(res\_prime\) is non-NULL, simultaneously evaluates the derivative \(P'_n(x)\).

The overall algorithm is described in [JM2018].

The versions \texttt{zero}, \texttt{one} respectively use the hypergeometric series expansions at \(x = 0\) and \(x = 1\) while the \texttt{asymp} version uses an asymptotic series on \((-1, 1)\) intended for large \(n\). The parameter \(K\) specifies the exact number of expansion terms to use (if the series expansion truncated at this point does not give the exact polynomial, an error bound is computed automatically). The asymptotic expansion with error bounds is given in [Bog2012]. The \texttt{rec} version uses the forward recurrence implemented using fixed-point arithmetic; it is only intended for the interval \((-1, 1)\), moderate \(n\) and modest precision.

The default version attempts to choose the best algorithm automatically. It also estimates the amount of cancellation in the hypergeometric series and increases the working precision to compensate, bounding the propagated error using derivative bounds.

void \texttt{arb\_hypgeom\_legendre\_p\_ui\_root}(\texttt{arb\_t res, arb\_t weight, ulong n, ulong k, slong prec})

Sets \(res\) to the \(k\)-th root of the Legendre polynomial \(P_n(x)\). We index the roots in decreasing order

\[
1 > x_0 > x_1 > \ldots > x_{n-1} > -1
\]

(which corresponds to ordering the roots of \(P_n(\cos(\theta))\) in order of increasing \(\theta\)). If \texttt{weight} is non-NULL, it is set to the weight corresponding to the node \(x_k\) for Gaussian quadrature on \([-1, 1]\). Note that only \(\lfloor n/2 \rfloor\) roots need to be computed, since the remaining roots are given by \(x_k = -x_{n-1-k}\).

We compute an enclosing interval using an asymptotic approximation followed by some number of Newton iterations, using the error bounds given in [Pet1999]. If very high precision is requested, the root is subsequently refined using interval Newton steps with doubling working precision.

**Hypergeometric sequences**

void \texttt{arb\_hypgeom\_central\_bin\_ui}(\texttt{arb\_t res, ulong n, slong prec})

Computes the central binomial coefficient \(\binom{2n}{n}\).
acb_elliptic.h – elliptic integrals and functions of complex variables

This module supports computation of elliptic (doubly periodic) functions, and their inverses, elliptic integrals. See `acb_modular.h` for the closely related modular forms and Jacobi theta functions.

Warning: incomplete elliptic integrals have very complicated branch structure when extended to complex variables. For some functions in this module, branch cuts may be artifacts of the evaluation algorithm rather than having a natural mathematical justification. The user should, accordingly, watch out for edge cases where the functions implemented here may differ from other systems or literature. There may also exist points where a function should be well-defined but the implemented algorithm fails to produce a finite result due to artificial internal singularities.

Complete elliptic integrals

```c
void acb_elliptic_k(acb_t res, const acb_t m, slong prec)
    Computes the complete elliptic integral of the first kind
    \[ K(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \int_0^1 \frac{dt}{\sqrt{1 - t^2} \sqrt{1 - mt^2}} \]
    using the arithmetic-geometric mean: \[ K(m) = \pi/(2M(\sqrt{1-m})) \].
```

```c
void acb_elliptic_k_jet(acb_ptr res, const acb_t m, slong len, slong prec)
    Sets the coefficients in the array res to the power series expansion of the complete elliptic integral of the first kind at the point m truncated to length len, i.e. \( K(m + x) \in \mathbb{C}[x] \).
```

```c
void _acb_elliptic_k_series(acb_ptr res, acb_srcptr m, slong mlen, slong len, slong prec)
void acb_elliptic_k_series(acb_poly_t res, const acb_poly_t m, slong len, slong prec)
    Sets res to the complete elliptic integral of the first kind of the power series m, truncated to length len.
```

```c
void acb_elliptic_e(acb_t res, const acb_t m, slong prec)
    Computes the complete elliptic integral of the second kind
    \[ E(m) = \int_0^{\pi/2} \sqrt{1 - m \sin^2 t} \, dt = \int_0^1 \frac{\sqrt{1 - mt^2}}{\sqrt{1 - t^2}} \, dt \]
    using \( E(m) = (1 - m)(2mK'(m) + K(m)) \) (where the prime denotes a derivative, not a complementary integral).
```

```c
void acb_elliptic_pi(acb_t res, const acb_t n, const acb_t m, slong prec)
    Evaluates the complete elliptic integral of the third kind
    \[ \Pi(n, m) = \int_0^{\pi/2} \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}} = \int_0^1 \frac{dt}{(1 - nt^2)\sqrt{1 - t^2} \sqrt{1 - mt^2}} \]
    This implementation currently uses the same algorithm as the corresponding incomplete integral. It is therefore less efficient than the implementations of the first two complete elliptic integrals which use the AGM.
```

Legendre incomplete elliptic integrals

```c
void acb_elliptic_f(acb_t res, const acb_t phi, const acb_t m, int pi, slong prec)
    Evaluates the Legendre incomplete elliptic integral of the first kind, given by
```
\[ F(\phi, m) = \int_0^\phi \frac{dt}{\sqrt{1 - m \sin^2 t}} = \int_0^{\sin \phi} \frac{dt}{\sqrt{(1 - t^2)(1 - mt^2)}} \]

on the standard strip \(-\pi/2 \leq \text{Re}(\phi) \leq \pi/2\). Outside this strip, the function extends quasiperiodically as

\[ F(\phi + n\pi, m) = 2nK(m) + F(\phi, m), n \in \mathbb{Z}. \]

Inside the standard strip, the function is computed via the symmetric integral \(R_F\).

If the flag \(pi\) is set to 1, the variable \(\phi\) is replaced by \(\pi \phi\), changing the quasiperiod to 1.

The function reduces to a complete elliptic integral of the first kind when \(\phi = \frac{\pi}{2}\); that is, \(F\left(\frac{\pi}{2}, m\right) = K(m)\).

void \textbf{acb_elliptic_e_inc}(acb_t res, const acb_t phi, const acb_t m, int pi, slong prec)

Evaluates the Legendre incomplete elliptic integral of the second kind, given by

\[ E(\phi, m) = \int_0^\phi \sqrt{1 - m \sin^2 t} \, dt = \int_0^{\sin \phi} \frac{dt}{\sqrt{1 - t^2} \sqrt{1 - mt^2}} \]

on the standard strip \(-\pi/2 \leq \text{Re}(\phi) \leq \pi/2\). Outside this strip, the function extends quasiperiodically as

\[ E(\phi + n\pi, m) = 2nE(m) + E(\phi, m), n \in \mathbb{Z}. \]

Inside the standard strip, the function is computed via the symmetric integrals \(R_F\) and \(R_D\).

If the flag \(pi\) is set to 1, the variable \(\phi\) is replaced by \(\pi \phi\), changing the quasiperiod to 1.

The function reduces to a complete elliptic integral of the second kind when \(\phi = \frac{\pi}{2}\); that is, \(E\left(\frac{\pi}{2}, m\right) = E(m)\).

void \textbf{acb_elliptic_pi_inc}(acb_t res, const acb_t n, const acb_t phi, const acb_t m, int pi, slong prec)

Evaluates the Legendre incomplete elliptic integral of the third kind, given by

\[ \Pi(n, \phi, m) = \int_0^\phi \frac{dt}{(1 - n \sin^2 t)\sqrt{1 - m \sin^2 t}} = \int_0^{\sin \phi} \frac{dt}{(1 - nt^2)\sqrt{1 - t^2} \sqrt{1 - mt^2}} \]

on the standard strip \(-\pi/2 \leq \text{Re}(\phi) \leq \pi/2\). Outside this strip, the function extends quasiperiodically as

\[ \Pi(n, \phi + k\pi, m) = 2k\Pi(n, m) + \Pi(n, \phi, m), k \in \mathbb{Z}. \]

Inside the standard strip, the function is computed via the symmetric integrals \(R_F\) and \(R_J\).

If the flag \(pi\) is set to 1, the variable \(\phi\) is replaced by \(\pi \phi\), changing the quasiperiod to 1.

The function reduces to a complete elliptic integral of the third kind when \(\phi = \frac{\pi}{2}\); that is, \(\Pi\left(n, \frac{\pi}{2}, m\right) = \Pi(n, m)\).

**Carlson symmetric elliptic integrals**

Carlson symmetric forms are the preferred form of incomplete elliptic integrals, due to their neat properties and relatively simple computation based on duplication theorems. There are five named functions: \(R_F, R_G, R_J, R_C, R_D\) which are special cases of \(R_F\) and \(R_J\) respectively. We largely follow the definitions and algorithms in [Car1995] and chapter 19 in [NIST2012].
void acb_elliptic_rf(acb_t res, const acb_t x, const acb_t y, const acb_t z, int flags, slong prec)

Evaluates the Carlson symmetric elliptic integral of the first kind

\[ R_F(x, y, z) = \frac{1}{2} \int_0^\infty \frac{dt}{\sqrt{(t + x)(t + y)(t + z)}} \]

where the square root extends continuously from positive infinity. The integral is well-defined for \( x, y, z \not\in (-\infty, 0) \), and with at most one of \( x, y, z \) being zero. When some parameters are negative real numbers, the function is still defined by analytic continuation.

In general, one or more duplication steps are applied until \( x, y, z \) are close enough to use a multivariate Taylor series.

The special case \( R_C(x, y) = R_F(x, y, y) = \frac{1}{2} \int_0^\infty (t + x)^{-1/2}(t + y)^{-1}dt \) may be computed by setting \( y \) and \( z \) to the same variable. (This case is not yet handled specially, but might be optimized in the future.)

The flags parameter is reserved for future use and currently does nothing. Passing 0 results in default behavior.

void acb_elliptic_rg(acb_t res, const acb_t x, const acb_t y, const acb_t z, int flags, slong prec)

Evaluates the Carlson symmetric elliptic integral of the second kind

\[ R_G(x, y, z) = \frac{1}{4} \int_0^\infty \frac{t}{\sqrt{(t + x)(t + y)(t + z)}} \left( \frac{x}{t + x} + \frac{y}{t + y} + \frac{z}{t + z} \right) dt \]

where the square root is taken continuously as in \( R_F \). The evaluation is done by expressing \( R_G \) in terms of \( R_F \) and \( R_D \). There are no restrictions on the variables.

void acb_elliptic_rj(acb_t res, const acb_t x, const acb_t y, const acb_t z, const acb_t p, int flags, slong prec)

Evaluates the Carlson symmetric elliptic integral of the third kind

\[ R_J(x, y, z, p) = \frac{3}{2} \int_0^\infty \frac{dt}{(t + p)\sqrt{(t + x)(t + y)(t + z)}} \]

where the square root is taken continuously as in \( R_F \).

In general, one or more duplication steps are applied until \( x, y, z, p \) are close enough to use a multivariate Taylor series.

The duplication algorithm is not correct for all possible combinations of complex variables, since the square roots taken during the computation can introduce spurious branch cuts. According to [Car1995], a sufficient (but not necessary) condition for correctness is that \( x, y, z \) have nonnegative real part and that \( p \) has positive real part.

In other cases, the algorithm might still be correct, but no attempt is made to check this; it is up to the user to verify that the duplication algorithm is appropriate for the given parameters before calling this function.

The special case \( R_D(x, y, z) = R_J(x, y, z, z) \) may be computed by setting \( z \) and \( p \) to the same variable. This case is handled specially to avoid redundant arithmetic operations. In this case, the algorithm is correct for all \( x, y \) and \( z \).

The flags parameter is reserved for future use and currently does nothing. Passing 0 results in default behavior.

void acb_elliptic_rc1(acb_t res, const acb_t x, slong prec)

This helper function computes the special case \( R_C(1, 1+x) = \frac{\arctan(\sqrt{x})}{\sqrt{x}} = _2F_1(1,1/2,3/2,-x) \), which is needed in the evaluation of \( R_J \).
Weierstrass elliptic functions

Elliptic functions may be defined on a general lattice \( \Lambda = \{ m2\omega_1 + n2\omega_2 : m, n \in \mathbb{Z} \} \) with half-periods \( \omega_1, \omega_2 \). We simplify by setting \( 2\omega_1 = 1, 2\omega_2 = \tau \) with \( \mathrm{im}(\tau) > 0 \). To evaluate the functions on a general lattice, it is enough to make a linear change of variables. The main reference is chapter 23 in [NIST2012].

```c
void acb_elliptic_p(acb_t res, 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

\[
\wp(z, \tau) = \pi^2 \theta_2^4(0, \tau)\theta_3^4(0, \tau)\theta_4^2(z, \tau) - \frac{\pi^2}{3} \left[ \theta_2^4(0, \tau) + \theta_4^4(0, \tau) \right].
\]

void acb_elliptic_p_jet(acb_ptr res, 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 \( \text{len} \). In particular, with \( \text{len} = 2 \), simultaneously computes \( \wp(z, \tau), \wp'(z, \tau) \) which together generate the field of elliptic functions with periods 1 and \( \tau \).

void _acb_elliptic_p_series(acb_ptr res, acb_srcptr z, slong zlen, const acb_t tau, slong len, slong prec)

Sets \( \text{res} \) to the Weierstrass elliptic function of the power series \( z \), with periods 1 and \( \tau \), truncated to length \( \text{len} \).

void acb_elliptic_invariants(acb_t g2, acb_t g3, const acb_t tau, slong prec)

Computes the lattice invariants \( g_2, g_3 \). The Weierstrass elliptic function satisfies the differential equation \( [\wp'(z, \tau)]^2 = 4[\wp(z, \tau)]^3 - g_2\wp(z, \tau) - g_3 \). Up to constant factors, the lattice invariants are the first two Eisenstein series (see \texttt{acb_modular_eisenstein()}).

void acb_elliptic_roots(acb_t e1, acb_t e2, acb_t e3, const acb_t tau, slong prec)

Computes the lattice roots \( e_1, e_2, e_3 \), which are the roots of the polynomial \( 4z^3 - g_2z - g_3 \).

void acb_elliptic_inv_p(acb_t res, const acb_t z, const acb_t tau, slong prec)

Computes the inverse of the Weierstrass elliptic function, which satisfies \( \wp(\wp^{-1}(z, \tau), \tau) = z \). This function is given by the elliptic integral

\[
\wp^{-1}(z, \tau) = \frac{1}{2} \int_z^\infty \frac{dt}{\sqrt{(t - e_1)(t - e_2)(t - e_3)}} = R_F(z - e_1, z - e_2, z - e_3).
\]

void acb_elliptic_zeta(acb_t res, const acb_t z, const acb_t tau, slong prec)

Computes the Weierstrass zeta function

\[
\zeta(z, \tau) = \frac{1}{z} + \sum_{n^2 + m^2 \neq 0} \left[ \frac{1}{z - m - n\tau} + \frac{1}{m + n\tau} + \frac{z}{(m + n\tau)^2} \right]
\]

which is quasiperiodic with \( \zeta(z + 1, \tau) = \zeta(z, \tau) + \zeta(1/2, \tau) \) and \( \zeta(z + \tau, \tau) = \zeta(z, \tau) + \zeta(\tau/2, \tau) \).

void acb_elliptic_sigma(acb_t res, const acb_t z, const acb_t tau, slong prec)

Computes the Weierstrass sigma function

\[
\sigma(z, \tau) = z \prod_{n^2 + m^2 \neq 0} \left( 1 - \frac{z}{m + n\tau} \right) \exp \left( \frac{z}{m + n\tau} + \frac{z^2}{2(m + n\tau)^2} \right)
\]

which is quasiperiodic with \( \sigma(z + 1, \tau) = -e^{2\pi i/3}(z + 1/2)\sigma(z, \tau) \) and \( \sigma(z + \tau, \tau) = -e^{2\pi i/3}(z/2 + \tau/2)\sigma(z, \tau) \).
acb_modular.h – modular forms of complex variables

This module provides methods for numerical evaluation of modular forms and Jacobi theta functions. See \texttt{acb_elliptic.h} for the closely related elliptic functions and integrals.

In the context of this module, \( \tau \) or \( \tau \) always denotes an element of the complex upper half-plane \( \mathbb{H} = \{ z \in \mathbb{C} : \text{Im}(z) > 0 \} \). We also often use the variable \( q \), variously defined as \( q = e^{\pi i \tau} \) (usually in relation to modular forms) or \( q = e^{2 \pi i \tau} \) (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 \( \tau \).

As usual, the numerical functions in this module compute strict error bounds: if \( \tau \) is represented by an \texttt{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 \(|q| < 1\).

The modular group

\texttt{psl2z_struct}

\texttt{psl2z_t}
代表一个模群 PSL\((2,\mathbb{Z})\) 的元素，即一个整数矩阵

\[
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix}
\]

与 \( ad - bc = 1 \)，且带有规范化的符号，其中 \( c \geq 0 \)，且 \( d > 0 \) 当 \( c = 0 \)。结构成员 \( a, b, c, d \) 是 \texttt{fmpz} 类型。

void \texttt{psl2z_init}(\texttt{psl2z_t g})
初始化 \( g \) 并将其设置为单位元。

void \texttt{psl2z_clear}(\texttt{psl2z_t g})
清除 \( g \)。

void \texttt{psl2z_swap}(\texttt{psl2z_t f, psl2z_t g})
交换 \( f \) 和 \( g \)。

void \texttt{psl2z_set}(\texttt{psl2z_t f, const psl2z_t g})
将 \( f \) 设置为 \( g \) 的副本。

void \texttt{psl2z_one}(\texttt{psl2z_t g})
将 \( g \) 设置为单位元。

int \texttt{psl2z_is_one}(\texttt{const psl2z_t g})
返回非零值，当且仅当 \( g \) 是单位元。

void \texttt{psl2z_print}(\texttt{const psl2z_t g})
打印 \( g \) 到标准输出。

void \texttt{psl2z_fprint}(\texttt{FILE * file, const psl2z_t g})
打印 \( g \) 到流 \( file \)。

int \texttt{psl2z_equal}(\texttt{const psl2z_t f, const psl2z_t g})
返回非零值，如果 \( f \) 和 \( g \) 相等。

void \texttt{psl2z_mul}(\texttt{psl2z_t h, const psl2z_t f, const psl2z_t g})
将 \( h \) 设置为 \( f \) 和 \( g \) 的矩阵乘积，带有符号规范化的。

void \texttt{psl2z_inv}(\texttt{psl2z_t h, const psl2z_t g})
将 \( h \) 设置为 \( g \) 的逆。

int \texttt{psl2z_is_correct}(\texttt{const psl2z_t g})
返回非零值，当且仅当 \( g \) 包含正确的数据，即满足 \( ad - bc = 1 \)，\( c \geq 0 \)，且 \( d > 0 \) 当 \( c = 0 \)。
void psl2z_randtest(psl2z_t g, flint_rand_t state, slong bits)
Sets $g$ to a random element of $\text{PSL}(2,\mathbb{Z})$ with entries of bit length at most $\text{bits}$ (or 1, if $\text{bits}$ is not positive). We first generate $a$ and $d$, compute their Bezout coefficients, divide by the GCD, and then correct the signs.

**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)
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 $\text{one\_minus\_eps}$. The inputs are assumed to be finite numbers, with $y$ positive.

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 $\text{one\_minus\_eps}$. 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 $\text{one\_minus\_eps}$. It also computes the transformed value $w = gz$. This function first tries to use $\text{acb\_modular\_fundamental\_domain\_approx\_d()}$ and checks if the result is acceptable. If this fails, it calls $\text{acb\_modular\_fundamental\_domain\_approx\_arf()}$ 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 $|\text{Re}(z)| \leq 1/2 + \varepsilon$ where $\varepsilon$ is specified by $\text{tol}$. Returns zero if this is false or cannot be determined.

**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 $\text{tab}$ to $-1$ if that index is to be part of the addition sequence, and to $0$ otherwise. On output, entry $i$ in $\text{tab}$ 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 $\text{tab}$ are ignored (the number 1 is always assumed to be part of the sequence).
Jacobi theta functions

Unfortunately, there are many inconsistent notational variations for Jacobi theta functions in the literature. Unless otherwise noted, we use the functions

\[
\theta_1(z, \tau) = -i \sum_{n=-\infty}^{\infty} (-1)^n \exp(i[(n + 1/2)\tau + (2n + 1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} (-1)^n q^{n(n+1)} \sin((2n + 1)\pi z)
\]

\[
\theta_2(z, \tau) = \sum_{n=-\infty}^{\infty} \exp(i[(n + 1/2)\tau + (2n + 1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos((2n + 1)\pi z)
\]

\[
\theta_3(z, \tau) = \sum_{n=-\infty}^{\infty} \exp(i[n^2\tau + 2nz]) = 1 + 2 \sum_{n=1}^{\infty} q^n \cos(2n\pi z)
\]

\[
\theta_4(z, \tau) = \sum_{n=-\infty}^{\infty} (-1)^n \exp(i[n^2\tau + 2nz]) = 1 + 2 \sum_{n=1}^{\infty} (-1)^n q^n \cos(2n\pi z)
\]

where \( q = \exp(i\pi \tau) \) and \( q_{1/4} = \exp(i\pi \tau / 4) \). Note that many authors write \( q_{1/4} \) as \( q^{1/4} \), but the principal fourth root \( (q^{1/4}) = \exp(i\log q) \) differs from \( q_{1/4} \) in general and some formulas are only correct if one reads \( "q^{1/4} = \exp(i\pi \tau / 4)" \). To avoid confusion, we only write \( q^k \) when \( k \) is an integer.

void \texttt{acb_modular_theta_transform}(int * \( R \), int * \( S \), int * \( C \), const \texttt{psl2z_t} \( g \))

We wish to write a theta function with quasiperiod \( \tau' = g\tau \), given some \( g = (a, b; c, d) \in \text{PSL}(2, \mathbb{Z}) \). 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

\[
\theta_{1+i}(z, \tau) = \exp(iR_i/4) \cdot A \cdot B \cdot \theta_{1+i}(z', \tau')
\]

where \( z' = z, A = B = 1 \) if \( C = 0 \), and

\[
z' = \frac{-z}{cr + d}, \quad A = \sqrt{\frac{1}{cr + d}}, \quad B = \exp\left(-\pi ic\frac{z^2}{cr + d}\right)
\]

if \( C = 1 \). Note that \( A \) is well-defined with the principal branch of the square root since \( A^2 = i/(cr + d) \) lies in the right half-plane.

Firstly, if \( c = 0 \), we have \( \theta_i(z, \tau) = \exp(-\pi ib/4) \theta_i(z, \tau + b) \) for \( i = 1, 2 \), whereas \( \theta_3 \) and \( \theta_4 \) 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

\[
\theta_1(z, \tau) = \varepsilon_1 AB \theta_1(z', \tau')
\]

\[
\theta_2(z, \tau) = \varepsilon_2 AB \theta_1-c_{1-a}(z', \tau')
\]

\[
\theta_3(z, \tau) = \varepsilon_3 AB \theta_{1+a-b}(z', \tau')
\]

\[
\theta_4(z, \tau) = \varepsilon_4 AB \theta_{1+a-b+1}(z', \tau')
\]
where $\varepsilon_i$ 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 \texttt{acb_modular\_epson\_arg()}), then:

$$
\varepsilon_1(a, b, c, d) = \exp(\pi i R(-d, b, c, -a) + 1)/4)
$$

$$
\varepsilon_2(a, b, c, d) = \exp(\pi i [-R(a, b, c, d) + (5 + (2 - c)a)]/4)
$$

$$
\varepsilon_3(a, b, c, d) = \exp(\pi i [-R(a, b, c, d) + (4 + (c - d - 2)(b - a))] / 4)
$$

$$
\varepsilon_4(a, b, c, d) = \exp(\pi i [-R(a, b, c, d) + (3 - (2 + d)b)] / 4)
$$

These formulas are easily derived from the formulas in [Rad1973] (Rademacher has the transformed/untransformed variables exchanged, and his “$\varepsilon$” differs from ours by a constant offset in the phase).

\begin{verbatim}
void acb_modular_addseq_theta(slong * exponents, slong * aindex, slong * bindex, slong num)
...
\end{verbatim}

Constructs an addition sequence for the first $\text{num}$ squares and triangular numbers interleaved (excluding zero), i.e. 1, 2, 4, 6, 9, 12, 16, 20, 25, 30 etc.

\begin{verbatim}
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)
...
\end{verbatim}

Simultaneously computes the first $\text{len}$ 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 $\text{len}$ equal to 1, computes the respective value of the theta function at the point $z$. We require $\text{len}$ 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 $\theta_1$ and $\theta_2$. 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 $(\theta_1, \theta_2)$ or $(\theta_3, \theta_4)$. For $k = 0, 1, 2, \ldots$, the powers of $q$ are $[k + 2]^2 / 4] = 1, 2, 4, 6, 9$ etc. and the powers of $w$ are $\pm(k + 2) = \pm2, \pm3, \pm4, \ldots$ etc. The scheme is illustrated by the following table:

$$
\begin{array}{c|c|c}
k & \theta_1, \theta_2 & q^k & (w^1 \pm w^{-1}) \\
0 & \theta_3, \theta_4 & q^1 & (w^2 \pm w^{-2}) \\
1 & \theta_1, \theta_2 & q^2 & (w^3 \pm w^{-3}) \\
2 & \theta_3, \theta_4 & q^4 & (w^4 \pm w^{-4}) \\
3 & \theta_1, \theta_2 & q^6 & (w^5 \pm w^{-5}) \\
4 & \theta_3, \theta_4 & q^9 & (w^6 \pm w^{-6}) \\
5 & \theta_1, \theta_2 & q^{12} & (w^7 \pm w^{-7}) \\
\end{array}
$$

For some integer $N \geq 1$, the summation is stopped just before term $k = N$. Let $Q = |q|$, $W = \max(|w|, |w^{-1}|)$, $E = [(N + 2)^2 / 4] + 1$. The error of the zeroth derivative can be bounded as

$$
2Q^E W^{N+2} \left[ 1 + Q^E W + Q^{2E} W^2 + \ldots \right] = \frac{2Q^E W^{N+2}}{1 - Q^E W}
$$

provided that the denominator is positive (otherwise we set the error bound to infinity). When $\text{len}$ 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 $\pi^r$, but we omit this until we rescale the coefficients at the end of the computation). Thus we have the error bound

$$2Q_F W^{N+2}(N+2)^r \left[ 1 + Q_F W^{(N+3)^r} (N+2)^r + Q_F^2 W^2 (N+4)^r (N+2)^r + \ldots \right]$$

which by the inequality $(1 + m/(N + 2))^r \leq \exp(mr/(N + 2))$ can be bounded as

$$\frac{2Q_F W^{N+2}(N+2)^r}{1 - Q_F W \exp(r/(N + 2))},$$

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 $w^{2n} - w^{-2n-2}$, and the sine terms as $w^{2n} - w^{-2n-2}$. 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.

```c
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 short addition sequence. The rs version uses rectangular splitting. The algorithms are described in [EHJ2016].

```c
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.

```c
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 $\tau$ to the fundamental domain. This is generally worse than `acb_modular_theta()`, but can be slightly better for moderate input.

```c
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 $\tau$ to the fundamental domain and then also reduces $z$ modulo $\tau$ before calling `acb_modular_theta_sum()`.

```c
void acb_modular_theta_jet_notransform(acb_ptr theta1, acb_ptr theta2, acb_ptr theta3, acb_ptr theta4, const acb_t z, const acb_t tau, slong len, slong prec)
void acb_modular_theta_jet(acb_ptr theta1, acb_ptr theta2, acb_ptr theta3, acb_ptr theta4, const acb_t z, const acb_t tau, slong len, slong prec)
```

Evaluates the Jacobi theta functions along with their derivatives with respect to $z$, writing the first $len$ coefficients in the power series $\theta_i(z + x, \tau) \in \mathbb{C}[z]$ to each respective output variable. The notransform version does not move $\tau$ to the fundamental domain or reduce $z$ during the computation.

### The Dedekind eta function

```c
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.
Given \( g \) on the number of terms. The algorithms are described in [EHJ2016].

The series is evaluated using either a short addition sequence or rectangular splitting, depending on the number of terms. The algorithms are described in [EHJ2016].

The series is evaluated using either a short addition sequence or rectangular splitting, depending on the number of terms. The algorithms are described in [EHJ2016].

void \texttt{acb_modular_eta_sum()}\( \) Computes the Dedekind eta function \( \eta(\tau) \) given \( \tau \) in the upper half-plane. This function applies the functional equation to move \( \tau \) to the fundamental domain before calling \texttt{acb_modular_eta_sum()}.

**Modular forms**

void \texttt{acb_modular_j()}\( \) Computes Klein’s \( j \)-invariant \( j(\tau) \) given \( \tau \) in the upper half-plane. The function is normalized so that \( j(i) = 1728 \). We first move \( \tau \) to the fundamental domain, which does not change the value of the function. Then we use the formula \( j(\tau) = 32(\theta_3^4 + \theta_4^4 + \theta_5^4)/(\theta_3 \theta_4 \theta_5)^4 \) where \( \theta_i = \theta_i(0, \tau) \).

void \texttt{acb_modular_lambda()}\( \) 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 \texttt{acb_modular_delta()}\( \) 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 \texttt{acb_modular_eisenstein()}\( \) Computes simultaneously the first \( \text{len} \) 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.
Elliptic integrals and functions

See the \texttt{acb\_elliptic.h} module for elliptic integrals and functions. The following wrappers are available for backwards compatibility.

```c
void acb_modular_elliptic_k(acb_t w, const acb_t m, slong prec)
void acb_modular_elliptic_k_cpx(acb_ptr w, const acb_t m, slong len, slong prec)
void acb_modular_elliptic_e(acb_t w, const acb_t m, slong prec)
void acb_modular_elliptic_p(acb_t wp, const acb_t z, const acb_t tau, slong prec)
void acb_modular_elliptic_p_zpx(acb_ptr wp, const acb_t z, const acb_t tau, slong len, slong prec)
```

Class polynomials

```c
void acb_modular_hilbert_class_poly(fmpz_poly_t res, slong D)
```

Sets \( \text{res} \) to the Hilbert class polynomial of discriminant \( D \), defined as

\[
H_D(x) = \prod_{(a,b,c)} \left( x - j \left( -b + \sqrt{D} \right) \right)
\]

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, \( \text{res} \) is set to the zero polynomial.

dirichlet.h – Dirichlet characters

Warning: the interfaces in this module are experimental and may change without notice.

This module allows working with Dirichlet characters algebraically. For evaluations of characters as complex numbers, see \texttt{acb\_dirichlet.h} – Dirichlet L-functions, zeta functions, and related functions.

Dirichlet characters

Working with Dirichlet characters mod \( q \) consists mainly in going from residue classes mod \( q \) to exponents on a set of generators of the group.

This implementation relies on the Conrey numbering scheme introduced in the L-functions and Modular Forms DataBase, which is an explicit choice of generators allowing to represent Dirichlet characters via the pairing

\[
\left( \mathbb{Z}/q\mathbb{Z} \right)^\times \times \left( \mathbb{Z}/q\mathbb{Z} \right)^\times \rightarrow \bigoplus_i \mathbb{Z}/\phi(q) \times \mathbb{Z}/\phi(q) \rightarrow \mathbb{C} \rightarrow \chi_q(m,n) = \exp(2\pi \sum \frac{a_i b_i}{\phi_i})
\]

We call \textit{number} a residue class \( m \) modulo \( q \), and \textit{log} the corresponding vector \((a_i)\) of exponents of Conrey generators.

Going from a \textit{log} to the corresponding \textit{number} is a cheap operation we call exponential, while the converse requires computing discrete logarithms.
Multiplicative group modulo \( q \)

**dirichlet_group_struct**

**dirichlet_group_t**

Represents the group of Dirichlet characters mod \( q \).

An `dirichlet_group_t` is defined as an array of `dirichlet_group_struct` of length 1, permitting it to be passed by reference.

**void dirichlet_group_init(dirichlet_group_t G, ulong q)**

Initializes \( G \) to the group of Dirichlet characters mod \( q \).

This method computes a canonical decomposition of \( G \) in terms of cyclic groups, which are the mod \( p^e \) subgroups for \( p^e \| q \), plus the specific generator described by Conrey for each subgroup.

In particular \( G \) contains:

- the number `num` of components
- the generators
- the exponent `expo` of the group

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 dirichlet_subgroup_init(dirichlet_group_t H, const dirichlet_group_t G, ulong h)**

Given an already computed group \( G \) mod \( q \), initialize its subgroup \( H \) defined mod \( h \| q \). Precomputed discrete log tables are inherited.

**void dirichlet_group_clear(dirichlet_group_t G)**

Clears \( G \). Remark this function does *not* clear the discrete logarithm tables stored in \( G \) (which may be shared with another group).

**ulong dirichlet_group_size(const dirichlet_group_t G)**

Returns the number of elements in \( G \), i.e. \( \varphi(q) \).

**ulong dirichlet_group_num_primitive(const dirichlet_group_t G)**

Returns the number of primitive elements in \( G \).

**void dirichlet_group_dlog_precompute(dirichlet_group_t G, ulong num)**

Precompute decomposition and tables for discrete log computations in \( G \), so as to minimize the complexity of \( num \) calls to discrete logarithms.

If \( num \) gets very large, the entire group may be indexed.

**void dirichlet_group_dlog_clear(dirichlet_group_t G, ulong num)**

Clear discrete logarithm tables in \( G \). When discrete logarithm tables are shared with subgroups, those subgroups must be cleared before clearing the tables.

**Character type**

**dirichlet_char_struct**

**dirichlet_char_t**

Represents a Dirichlet character. This structure contains both a number (residue class) and the corresponding log (exponents on the group generators).

An `dirichlet_char_t` is defined as an array of `dirichlet_char_struct` of length 1, permitting it to be passed by reference.

**void dirichlet_char_init(dirichlet_char_t chi, const dirichlet_group_t G)**

Initializes \( chi \) to an element of the group \( G \) and sets its value to the principal character.
void dirichlet_char_clear(dirichlet_char_t chi)
    Clears chi.

void dirichlet_char_print(const dirichlet_group_t G, const dirichlet_char_t chi)
    Prints the array of exponents representing this character.

void dirichlet_char_log(dirichlet_char_t x, const dirichlet_group_t G, ulong m)
    Sets x to the character of number m, computing its log using discrete logarithm in G.

ulong dirichlet_char_exp(const dirichlet_group_t G, const dirichlet_char_t x)
    Returns the number m corresponding to exponents in x.

ulong _dirichlet_char_exp(dirichlet_char_t x, const dirichlet_group_t G)
    Computes and returns the number m corresponding to exponents in x. This function is for internal use.

void dirichlet_char_one(dirichlet_char_t x, const dirichlet_group_t G)
    Sets x to the principal character in G, having log[0,...,0].

void dirichlet_char_first_primitive(dirichlet_char_t x, const dirichlet_group_t G)
    Sets x to the first primitive character of G, having log[1,...,1], or [0,1,...,1] if 8 | q.

void dirichlet_char_set(dirichlet_char_t x, const dirichlet_group_t G, const dirichlet_char_t y)
    Sets x to the element y.

int dirichlet_char_next(dirichlet_char_t x, const dirichlet_group_t G)
    Sets x to the next character in G according to lexicographic ordering of log.

    The return value is the index of the last updated exponent of x, or -1 if the last element has been reached.

    This function allows to iterate on all elements of G looping on their log. Note that it produces elements in seemingly random number order.

    The following template can be used for such a loop:

    dirichlet_char_one(chi, G);
    do {
        /* use character chi */
    } while (dirichlet_char_next(chi, G) >= 0);

int dirichlet_char_next_primitive(dirichlet_char_t x, const dirichlet_group_t G)
    Same as dirichlet_char_next(), but jumps to the next primitive character of G.

ulong dirichlet_index_char(const dirichlet_group_t G, const dirichlet_char_t x)
    Returns the lexicographic index of the log of x as an integer in 0...\phi(q).

void dirichlet_char_index(dirichlet_char_t x, const dirichlet_group_t G, ulong j)
    Sets x to the character whose log has lexicographic index j.

int dirichlet_char_eq(const dirichlet_char_t x, const dirichlet_char_t y)
int dirichlet_char_eq_deep(const dirichlet_group_t G, const dirichlet_char_t x, const dirichlet_char_t y)
    Return 1 if x equals y.

    The second version checks every byte of the representation and is intended for testing only.

Character properties

As a consequence of the Conrey numbering, all these numbers are available at the level of number and char object. Both case require no discrete log computation.

int dirichlet_char_is_principal(const dirichlet_group_t G, const dirichlet_char_t chi)
    Returns 1 if chi is the principal character mod q.
ulong dirichlet_conductor_ui(const dirichlet_group_t G, ulong a)
ulong dirichlet_conductor_char(const dirichlet_group_t G, const dirichlet_char_t x)

Returns the conductor of \( \chi_q(a, \cdot) \), that is the smallest \( r \) dividing \( q \) such that \( \chi_q(a, -1) = (-1)^r \).

int dirichlet_parity_ui(const dirichlet_group_t G, ulong a)
int dirichlet_parity_char(const dirichlet_group_t G, const dirichlet_char_t x)

Returns the parity \( \lambda \) in \{0, 1\} of \( \chi_q(a, \cdot) \), such that \( \chi_q(a, -1) = (-1)^\lambda \).

ulong dirichlet_order_ui(const dirichlet_group_t G, ulong a)
ulong dirichlet_order_char(const dirichlet_group_t G, const dirichlet_char_t x)

Returns the order of \( \chi_q(a, \cdot) \) which is the order of \( a \) mod \( q \).

Character evaluation

Dirichlet characters take value in a finite cyclic group of roots of unity plus zero.

Evaluation functions return a ulong, this number corresponds to the power of a primitive root of unity, the special value DIRICHLET_CHI_NULL encoding the zero value.

ulong dirichlet_pairing(const dirichlet_group_t G, ulong m, ulong n)
ulong dirichlet_pairing_char(const dirichlet_group_t G, const dirichlet_char_t chi, const dirichlet_char_t psi)

Compute the value of the Dirichlet pairing on numbers \( m \) and \( n \), as exponent modulo \( G->expo \).

The char variant takes as input two characters, so that no discrete logarithm is computed.

The returned value is the numerator of the actual value exponent mod the group exponent \( G->expo \).

ulong dirichlet_chi(const dirichlet_group_t G, const dirichlet_char_t chi, ulong n)

Compute the value \( \chi(n) \) as the exponent modulo \( G->expo \).

void dirichlet_chi_vec(ulong * v, const dirichlet_group_t G, const dirichlet_char_t chi, slong nv)
void dirichlet_chi_vec_order(ulong * v, const dirichlet_group_t G, const dirichlet_char_t chi, ulong order, slong nv)

Compute the list of exponent values \( v[k] \) for \( 0 \leq k < nv \), as exponents modulo \( G->expo \) or order, which is assumed to be a multiple of the order of \( \chi \).

Character operations

void dirichlet_char_mul(dirichlet_char_t chi12, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2)
Multiply two characters of the same group \( G \).

void dirichlet_char_pow(dirichlet_char_t c, const dirichlet_group_t G, const dirichlet_char_t a, ulong n)
Take the power of a character.

void dirichlet_char_lift(dirichlet_char_t chi_G, const dirichlet_group_t G, const dirichlet_char_t chi_H, const dirichlet_group_t H)

If \( H \) is a subgroup of \( G \), computes the character in \( G \) corresponding to \( \chi_H \) in \( H \).
void dirichlet_char_lower(dirichlet_char_t chi_H, const dirichlet_group_t H, const dirichlet_char_t chi_G, const dirichlet_group_t G).

If \( \chi_G \) is a character of \( G \) which factors through \( H \), sets \( \chi_H \) to the corresponding restriction in \( H \).

This requires \( c(\chi_G) \mid q_H \mid q_G \), where \( c(\chi_G) \) is the conductor of \( \chi_G \) and \( q_G, q_H \) are the moduli of \( G \) and \( H \).

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 allows working with values of Dirichlet characters, Dirichlet L-functions, and related functions. Working with Dirichlet characters is documented in dirichlet.h – Dirichlet characters.

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.

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.

Roots of unity

acb_dirichlet_roots_struct

acb_dirichlet_roots_t

void acb_dirichlet_roots_init(acb_dirichlet_roots_t roots, ulong n, slong num, slong prec)

Initializes \( \text{roots} \) with precomputed data for fast evaluation of roots of unity \( e^{2\pi i k/n} \) of a fixed order \( n \). The precomputation is optimized for \( \text{num} \) evaluations.

For very small \( \text{num} \), only the single root \( e^{2\pi i/n} \) will be precomputed, which can then be raised to a power. For small \( \text{prec} \) and large \( n \), this method might even skip precomputing this single root if it estimates that evaluating roots of unity from scratch will be faster than powering.

If \( \text{num} \) is large enough, the whole set of roots in the first quadrant will be precomputed at once. However, this is automatically avoided for large \( n \) if too much memory would be used. For intermediate \( \text{num} \), baby-step giant-step tables are computed.

void acb_dirichlet_roots_clear(acb_dirichlet_roots_t roots)

Clears the structure.

void acb_dirichlet_root(acb_t res, const acb_dirichlet_roots_t roots, ulong k, slong prec)

Computes \( e^{2\pi i k/n} \).

Truncated L-series and power sums

void acb_dirichlet_powsum_term(acb_ptr res, arb_t log_prev, ulong * prev, const acb_t s, ulong k, int integer, int critical_line, slong len, slong prec)

Sets \( \text{res} \) to \( k^{-(s+\nu)} \) as a power series in \( x \) truncated to length \( \text{len} \). The flags \( \text{integer} \) and \( \text{critical\_line} \) respectively specify optimizing for \( s \) being an integer or having real part 1/2.

On input \( \text{log\_prev} \) should contain the natural logarithm of the integer at \( \text{prev} \). If \( \text{prev} \) is close to \( k \), this can be used to speed up computations. If \( \log(k) \) is computed internally by this function, then \( \text{log\_prev} \) is overwritten by this value, and the integer at \( \text{prev} \) is overwritten by \( k \), allowing \( \text{log\_prev} \) to be recycled for the next term when evaluating a power sum.
void acb_dirichlet_powsum_sieved(acb_ptr res, const acb_t s, ulong n, slong len, slong prec)
Sets res to $\sum_{k=1}^{n} k^{-(s+x)}$ as a power series in $x$ truncated to length len. This function stores a table of powers that have already been calculated, computing $(ij)^x$ as $i^x j^x$ whenever $k = ij$ is composite. As a further optimization, it groups all even $k$ and evaluates the sum as a polynomial in $2^{-(s+x)}$. 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.

void acb_dirichlet_powsum_smooth(acb_ptr res, const acb_t s, ulong n, slong len, slong prec)
Sets res to $\sum_{k=1}^{n} k^{-(s+x)}$ as a power series in $x$ truncated to length len. This function performs partial sieving by adding multiples of 5-smooth $k$ into separate buckets. Asymptotically, this requires computing $4/15$ of the powers, which is slower than sieved, but only requires logarithmic extra space. It is also faster for large $len$, since most power series multiplications are traded for additions. A slightly bigger gain for larger $n$ could be achieved by using more small prime factors, at the expense of space.

Riemann zeta function

void acb_dirichlet_zeta(acb_t res, const acb_t s, slong prec)
Computes $\zeta(s)$ using an automatic choice of algorithm.

void acb_dirichlet_zeta_sieve(acb_t res, const acb_t s, int deflate, slong len, slong prec)
Computes the first $len$ terms of the Taylor series of the Riemann zeta function at $s$. If $deflate$ is nonzero, computes the deflated function $\zeta(s) - 1/(s-1)$ instead.

void acb_dirichlet_zeta_bound(mag_t res, const acb_t s)
Computes an upper bound for $|\zeta(s)|$ quickly. On the critical strip (and slightly outside of it), formula (433) in [Rad1979] is used. To the right, evaluating at the real part of $s$ gives a trivial bound. To the left, the functional equation is used.

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.

Riemann-Siegel formula

The Riemann-Siegel (RS) formula is implemented closely following J. Arias de Reyna [Ari2011]. For $s = \sigma + it$ with $t > 0$, the expansion takes the form

$$\zeta(s) = \mathcal{R}(s) + X(s)\mathcal{R}(1-s), \quad X(s) = \pi^{s-1/2} \frac{\Gamma((1-s)/2)}{\Gamma(s/2)}$$

where

$$\mathcal{R}(s) = \sum_{k=1}^{N} \frac{1}{k^s} + (-1)^{N-1} U a^{-\sigma} \left[ \sum_{k=0}^{K} \frac{C_k(p)}{a^k} + RS_K \right]$$

$$U = \exp \left( -i \left[ \frac{t}{2} \log \left( \frac{t}{2\pi} \right) - \frac{t}{2} - \frac{\pi}{8} \right] \right), \quad a = \sqrt{\frac{t}{2\pi}}, \quad N = \lfloor a \rfloor, \quad p = 1 - 2(a - N).$$

The coefficients $C_k(p)$ in the asymptotic part of the expansion are expressed in terms of certain auxiliary coefficients $d_j^{(k)}$ and $F^{(j)}(p)$. Because of artificial discontinuities, $s$ should be exact inside the evaluation.

void acb_dirichlet_zeta_rs_f_coeffs(acb_ptr f, const arb_t p, slong n, slong prec)
Computes the coefficients $F^{(j)}(p)$ for $0 \leq j < n$. Uses power series division. This method breaks down when $p = 1/2$ (which is not problem if $s$ is an exact floating-point number).
void \texttt{acb\_dirichlet\_zeta\_rs\_d\_coeffs} (\texttt{arb\_ptr d, const arb\_t sigma, slong k, slong prec})

Computes the coefficients $d_j^{(k)}$ for $0 \leq j \leq [3k/2] + 1$. On input, the array $d$ must contain the coefficients for $d_j^{(k-1)}$ unless $k = 0$, and these coefficients will be updated in-place.

void \texttt{acb\_dirichlet\_zeta\_rs\_bound} (\texttt{mag\_t err, const acb\_t s, slong K})

Bounds the error term $R_S$ following Theorem 4.2 in Arias de Reyna.

void \texttt{acb\_dirichlet\_zeta\_rs\_r} (\texttt{acb\_t res, const acb\_t s, slong K, slong prec})

Computes $R(s)$ in the upper half plane. Uses precisely $K$ asymptotic terms in the RS formula if this input parameter is positive; otherwise chooses the number of terms automatically based on $s$ and the precision.

void \texttt{acb\_dirichlet\_zeta\_rs} (\texttt{acb\_t res, const acb\_t s, slong K, slong prec})

Computes $\zeta(s)$ using the Riemann-Siegel formula. Uses precisely $K$ asymptotic terms in the RS formula if this input parameter is positive; otherwise chooses the number of terms automatically based on $s$ and the precision.

void \texttt{acb\_dirichlet\_zeta\_jet\_rs} (\texttt{acb\_t res, const acb\_t s, slong len, slong prec})

Computes the first $\texttt{len}$ terms of the Taylor series of the Riemann zeta function at $s$ using the Riemann siegel formula. This function currently only supports $\texttt{len} = 1$ or $\texttt{len} = 2$. A finite difference is used to compute the first derivative.

### Hurwitz zeta function

void \texttt{acb\_dirichlet\_hurwitz} (\texttt{acb\_t res, const acb\_t s, const acb\_t a, slong prec})

Computes the Hurwitz zeta function $\zeta(s,a)$. This function automatically delegates to the code for the Riemann zeta function when $a = 1$. Some other special cases may also be handled by direct formulas. In general, Euler-Maclaurin summation is used.

### Hurwitz zeta function precomputation

\texttt{acb\_dirichlet\_hurwitz\_precomp\_struct}

\texttt{acb\_dirichlet\_hurwitz\_precomp\_t}

void \texttt{acb\_dirichlet\_hurwitz\_precomp\_init} (\texttt{acb\_dirichlet\_hurwitz\_precomp\_t pre, const acb\_t s, int deflate, ulong A, ulong K, ulong N, slong prec})

Precomputes a grid of Taylor polynomials for fast evaluation of $\zeta(s,a)$ on $a \in (0,1]$ with fixed $s$. $A$ is the initial shift to apply to $s$. $K$ is the number of Taylor terms, $N$ is the number of grid points.

The precomputation requires $NK$ evaluations of the Hurwitz zeta function, and each subsequent evaluation requires $2K$ simple arithmetic operations (polynomial evaluation) plus $A$ powers. As $K$ grows, the error is at most $O(1/(2AN)^K)$.

This function can be called with $A$ set to zero, in which case no Taylor series precomputation is performed. This means that evaluation will be identical to calling \texttt{acb\_dirichlet\_hurwitz()} directly.

Otherwise, we require that $A$, $K$ and $N$ are all positive. For a finite error bound, we require $K + \Re(s) > 1$. To avoid an initial “bump” that steals precision and slows convergence, $AN$ should be at least roughly as large as $|s|$, e.g. it is a good idea to have at least $AN > 0.5|s|$.

If \texttt{deflate} is set, the deflated Hurwitz zeta function is used, removing the pole at $s = 1$.

void \texttt{acb\_dirichlet\_hurwitz\_precomp\_init\_num} (\texttt{acb\_dirichlet\_hurwitz\_precomp\_t pre, const acb\_t s, int deflate, double num\_eval, slong prec})

Initializes \texttt{pre}, choosing the parameters $A$, $K$, and $N$ automatically to minimize the cost of \texttt{num\_eval} evaluations of the Hurwitz zeta function at argument $s$ to precision \texttt{prec}.

void \texttt{acb\_dirichlet\_hurwitz\_precomp\_clear} (\texttt{acb\_dirichlet\_hurwitz\_precomp\_t pre})

Clears the precomputed data.
void acb_dirichler_hurwitz_precomp_choose_param(ulong * A, ulong * K, ulong * N, const acb_t s, double num_eval, slong prec)

Chooses precomputation parameters A, K and N to minimize the cost of num_eval evaluations of the Hurwitz zeta function at argument s to precision prec. If it is estimated that evaluating each Hurwitz zeta function from scratch would be better than performing a precomputation, A, K and N are all set to 0.

void acb_dirichlet_hurwitz_precomp_bound(mag_t res, const acb_t s, ulong A, ulong K, ulong N)

Computes an upper bound for the truncation error (not accounting for roundoff error) when evaluating \( \zeta(s, a) \) with precomputation parameters A, K, N, assuming that 0 < a \leq 1. For details, see Algorithms for the Hurwitz zeta function.

void acb_dirichlet_hurwitz_precomp_eval(acb_t res, const acb_dirichlet_hurwitz_precomp_t pre, ulong p, ulong q, slong prec)

Evaluates \( \zeta(s, p/q) \) using precomputed data, assuming that 0 < p/q \leq 1.

**Dirichlet character evaluation**

void acb_dirichlet_chi(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, ulong n, slong prec)

Sets res to \( \chi(n) \), the value of the Dirichlet character chi at the integer n.

void acb_dirichlet_chi_vec(acb_ptr v, const dirichlet_group_t G, const dirichlet_char_t chi, slong nv, slong prec)

Compute the nv first Dirichlet values.

void acb_dirichlet_pairing(acb_t res, const dirichlet_group_t G, ulong m, ulong n, slong prec)

void acb_dirichlet_pairing_char(acb_t res, const dirichlet_group_t G, const dirichlet_char_t a, const dirichlet_char_t b, slong prec)

Sets res to the value of the Dirichlet pairing \( \chi(m, n) \) at numbers m and n. The second form takes two characters as input.

**Dirichlet character Gauss, Jacobi and theta sums**

void acb_dirichlet_gauss_sum_naive(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)

void acb_dirichlet_gauss_sum_factor(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)

void acb_dirichlet_gauss_sum_order2(acb_t res, const dirichlet_char_t chi, slong prec)

void acb_dirichlet_gauss_sum_theta(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)

void acb_dirichlet_gauss_sum(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)

void acb_dirichlet_gauss_sum_ui(acb_t res, const dirichlet_group_t G, ulong a, slong prec)

Sets res to the Gauss sum

\[
G_q(a) = \sum_{x \mod q} \chi_q(a, x)e^{2\pi i x}/q
\]

- the naive version computes the sum as defined.
- the factor version writes it as a product of local Gauss sums by chinese remainder theorem.
- the order2 version assumes \( \chi \) is real and primitive and returns \( ip\sqrt{q} \) where \( p \) is the parity of \( \chi \).
• the theta version assumes that chi is primitive to obtain the Gauss sum by functional equation of the theta series at $t = 1$. An abort will be raised if the theta series vanishes at $t = 1$. Only 4 exceptional characters of conductor 300 and 600 are known to have this particularity, and none with primepower modulus.

• the default version automatically combines the above methods.

• the ui version only takes the Conrey number $a$ as parameter.

```c
void acb_dirichlet_jacobi_sum_naive(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2, slong prec)
void acb_dirichlet_jacobi_sum_factor(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2, slong prec)
void acb_dirichlet_jacobi_sum_gauss(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2, slong prec)
void acb_dirichlet_jacobi_sum(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2, slong prec)
void acb_dirichlet_jacobi_sum_ui(acb_t res, const dirichlet_group_t G, ulong a, ulong b, slong prec)
```

Computes the Jacobi sum

$$J_q(a, b) = \sum_{x \mod q} \chi_q(a, x)\chi_q(b, 1 - x)$$

• the naive version computes the sum as defined.

• the factor version writes it as a product of local Jacobi sums

• the gauss version assumes $ab$ is primitive and uses the formula $J_q(a, b)G_q(ab) = G_q(a)G_q(b)$

• the default version automatically combines the above methods.

• the ui version only takes the Conrey numbers $a$ and $b$ as parameters.

```c
void acb_dirichlet_chi_theta_arb(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, const arb_t t, slong prec)
void acb_dirichlet_ui_theta_arb(acb_t res, const dirichlet_group_t G, ulong a, const arb_t t, slong prec)
```

Compute the theta series $\Theta_q(a, t)$ for real argument $t > 0$. Beware that if $t < 1$ the functional equation

$$t \theta(a, t) = \epsilon(\chi)\theta\left(\frac{1}{a}, \frac{1}{t}\right)$$

should be used, which is not done automatically (to avoid recomputing the Gauss sum).

We call theta series of a Dirichlet character the quadratic series

$$\Theta_q(a) = \sum_{n \geq 0} \chi_q(a, n)n^p x^{n^2}$$

where $p$ is the parity of the character $\chi_q(a, \cdot)$.

For $\Re(t) > 0$ we write $x(t) = \exp(-\frac{x}{N}t^2)$ and define

$$\Theta_q(a, t) = \sum_{n \geq 0} \chi_q(a, n)x(t)^n.$$
void \texttt{acb\_dirichlet\_qseries\_powers\_naive}(\texttt{acb\_t res}, \texttt{const arb\_t x}, \texttt{int p}, \texttt{const ulong } *a, \texttt{const acb\_dirichlet\_powers\_t z}, \texttt{slong len}, \texttt{slong prec})

void \texttt{acb\_dirichlet\_qseries\_powers\_smallorder}(\texttt{acb\_t res}, \texttt{const arb\_t x}, \texttt{int p}, \texttt{const ulong } *a, \texttt{const acb\_dirichlet\_powers\_t z}, \texttt{slong len}, \texttt{slong prec})

Compute the series $\sum \mathit{n}^p z^a x^n$ for exponent list $a$, precomputed powers $z$ and parity $p$ (being 0 or 1).

The \textit{naive} version sums the series as defined, while the \textit{smallorder} variant evaluates the series on the quotient ring by a cyclotomic polynomial before evaluating at the root of unity, ignoring its argument $z$.

**Discrete Fourier transforms (DFT)**

If $f$ is a function $\mathbb{Z}/q\mathbb{Z} \to \mathbb{C}$, its discrete Fourier transform is the function defined on Dirichlet characters mod $q$ by

$$\hat{f}(\chi) = \sum_{x \mod q} \chi(x) f(x)$$

See the \texttt{acb\_dft.h} – Discrete Fourier transform module.

Here we take advantage of the Conrey isomorphism $G \to \hat{G}$ to consider the Fourier transform on Conrey labels as

$$g(a) = \sum_{b \mod q} \chi_q(a, b) f(b)$$

void \texttt{acb\_dirichlet\_dft\_conrey}(\texttt{acb\_ptr w}, \texttt{acb\_srcptr v}, \texttt{const dirichlet\_group\_t G}, \texttt{slong prec})

Compute the DFT of $v$ using Conrey indices. This function assumes $v$ and $w$ are vectors of size $G->\phi_q$, whose values correspond to a lexicographic ordering of Conrey logs (as obtained using \texttt{dirichlet\_char\_next()} or by \texttt{dirichlet\_char\_index()}).

For example, if $q = 15$, the Conrey elements are stored in following order

<table>
<thead>
<tr>
<th>index</th>
<th>log = [e,f]</th>
<th>number = $7^e 11^f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0, 0]</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[0, 1]</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>[0, 2]</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>[0, 3]</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>[0, 4]</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>[1, 0]</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>[1, 1]</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>[1, 2]</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>[1, 3]</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>[1, 4]</td>
<td>11</td>
</tr>
</tbody>
</table>

void \texttt{acb\_dirichlet\_dft}(\texttt{acb\_ptr w}, \texttt{acb\_srcptr v}, \texttt{const dirichlet\_group\_t G}, \texttt{slong prec})

Compute the DFT of $v$ using Conrey numbers. This function assumes $v$ and $w$ are vectors of size $G->q$. All values at index not coprime to $G->q$ are ignored.

**Dirichlet L-functions**

void \texttt{acb\_dirichlet\_root\_number\_theta}(\texttt{acb\_t res}, \texttt{const dirichlet\_group\_t G}, \texttt{const dirichlet\_char\_t chi}, \texttt{slong prec})
void \texttt{acb_dirichlet_root_number}(acb\_t res, const \texttt{dirichlet\_group\_t} G, const \texttt{dirichlet\_char\_t} chi, slong prec)

Sets \( \text{res} \) to the root number \( \epsilon(\chi) \) for a primitive character \( \chi \), which appears in the functional equation (where \( p \) is the parity of \( \chi \)):

\[
\left( \frac{q}{\pi} \right)^{\frac{s}{2}} \Gamma \left( \frac{s + p}{2} \right) L(s, \chi) = \epsilon(\chi) \left( \frac{q}{\pi} \right)^{\frac{1-s-p}{2}} \Gamma \left( \frac{1-s+p}{2} \right) L(1-s, \overline{\chi})
\]

- The \texttt{theta} variant uses the evaluation at \( t = 1 \) of the Theta series.
- The default version computes it via the Gauss sum.

void \texttt{acb_dirichlet\_l\_hurwitz}(acb\_t res, \texttt{acb\_t} s, \texttt{const \_acb\_dirichlet\_euler\_product\_real\_ui} precomp, \texttt{const \_acb\_dirichlet\_group\_t} G, const \texttt{dirichlet\_char\_t} chi, \texttt{slong prec})

Computes \( L(s, \chi) \) using decomposition in terms of the Hurwitz zeta function

\[
L(s, \chi) = q^{-s} \sum_{k=1}^{q} \chi(k) \zeta \left( s, \frac{k}{q} \right).
\]

If \( s = 1 \) and \( \chi \) is non-principal, the deflated Hurwitz zeta function is used to avoid poles.

If \texttt{precomp} is \texttt{NULL}, each Hurwitz zeta function value is computed directly. If a pre-initialized \texttt{precomp} object is provided, this will be used instead to evaluate the Hurwitz zeta function.

void \texttt{acb_dirichlet\_l\_euler\_product}(acb\_t res, \texttt{acb\_t} s, \texttt{const \_acb\_dirichlet\_hurwitz\_precomp\_t} precomp, \texttt{const \_acb\_dirichlet\_group\_t} G, \texttt{const \_dirichlet\_char\_t} chi, \texttt{slong prec})

Computes \( L(s, \chi) \) directly using the Euler product. This is efficient if \( s \) has large positive real part. As implemented, this function only gives a finite result if \( \text{re}(s) \geq 2 \).

An error bound is computed via \texttt{mag\_hurwitz\_zeta\_uiui()} . If \( s \) is complex, replace it with its real part. 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

\[
\left| \prod_{p < N} \left( 1 - \frac{\chi(p)}{p^s} \right) - \frac{1}{L(s, \chi)} \right| \leq \sum_{k=N}^{\infty} \frac{1}{k^s} = \zeta(s, N).
\]

The underscore version specialized for integer \( s \) assumes that \( \chi \) is a real Dirichlet character given by the explicit list \texttt{chi} of character values at 0, 1, ..., \texttt{mod} - 1. If \texttt{reciprocal} is set, it computes 1/\( L(s, \chi) \) (this is faster if the reciprocal can be used directly).

void \texttt{acb_dirichlet\_l}(acb\_t res, \texttt{acb\_t} s, \texttt{const \_dirichlet\_group\_t} G, \texttt{const \_dirichlet\_char\_t} chi, \texttt{slong prec})

Computes \( L(s, \chi) \) using a default choice of algorithm.

void \texttt{acb_dirichlet\_l\_vec\_hurwitz}(acb\_ptr res, \texttt{acb\_t} s, \texttt{const \_acb\_dirichlet\_hurwitz\_precomp\_t} precomp, \texttt{const \_dirichlet\_group\_t} G, \texttt{slong prec})

Compute all values \( L(s, \chi) \) for \( \chi \mod q \), using the Hurwitz zeta function and a discrete Fourier transform. The output \texttt{res} is assumed to have length \( G->\phi_iq \) and values are stored by lexicographically ordered Conrey logs. See \texttt{acb\_dirichlet\_dft\_conrey()} .

If \texttt{precomp} is \texttt{NULL}, each Hurwitz zeta function value is computed directly. If a pre-initialized \texttt{precomp} object is provided, this will be used instead to evaluate the Hurwitz zeta function.
void `acb_dirichlet_l_jet(acb_ptr res, const acb_t s, const dirichlet_group_t G, const dirichlet_char_t chi, int deflate, slong len, slong prec)`

Computes the Taylor expansion of $L(s, \chi)$ to length len, i.e. $L(s), L'(s), \ldots, L^{(len-1)}(s)/(len - 1)!$. If `deflate` is set, computes the expansion of

$$L(s, \chi) = \sum_{k=1}^{q} \chi(k)/(s-1)^q$$

instead. If `chi` is a principal character, then this has the effect of subtracting the pole with residue $\sum_{k=1}^{q} \chi(k) = \phi(q)/q$ that is located at $s = 1$. In particular, when evaluated at $s = 1$, this gives the regular part of the Laurent expansion. When `chi` is non-principal, `deflate` has no effect.

void `_acb_dirichlet_l_series(acb_ptr res, acb_srcptr s, slong slen, const dirichlet_group_t G, const dirichlet_char_t chi, int deflate, slong len, slong prec)`

void `acb_dirichlet_l_series(acb_poly_t res, const acb_poly_t s, const dirichlet_group_t G, const dirichlet_char_t chi, int deflate, slong len, slong prec)`

Sets `res` to the power series $\sum_{k=0}^{q} \chi(k)/(s-1)^k$, which is real-valued for real $s$. The first $\text{len}$ terms in the Taylor expansion are written to the output.

void `acb_dirichlet_hardy_theta(acb_ptr res, const acb_t t, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)`

Computes the phase function used to construct the Z-function. We have

$$\theta(t) = -\frac{t}{2} \log(\pi/\chi) - i \frac{\log(\epsilon)}{2} + \frac{\log \Gamma((s + \delta)/2) - \log \Gamma((1-s + \delta)/2)}{2i}$$

where $s = 1/2 + it$, $\delta$ is the parity of $\chi$, and $\epsilon$ is the root number as computed by `acb_dirichlet_root_number()`. The first `len` terms in the Taylor expansion are written to the output.

void `acb_dirichlet_hardy_z(acb_t res, const acb_t t, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)`

Computes the Hardy Z-function, also known as the Riemann-Siegel Z-function $Z(t) = e^{\theta(t)} L(1/2 + it)$, which is real-valued for real $t$. The first `len` terms in the Taylor expansion are written to the output.

void `_acb_dirichlet_hardy_theta_series(acb_ptr res, acb_srcptr t, slong tlen, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)`

void `acb_dirichlet_hardy_theta_series(acb_poly_t res, const acb_poly_t t, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)`

Sets `res` to the power series $\theta(t)$ where $t$ is a given power series, truncating the result to length `len`.

void `_acb_dirichlet_hardy_z_series(acb_ptr res, acb_srcptr t, slong tlen, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)`

void `acb_dirichlet_hardy_z_series(acb_poly_t res, const acb_poly_t t, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)`

Sets `res` to the power series $Z(t)$ where $t$ is a given power series, truncating the result to length `len`.

8.6. `acb_dirichlet.h` – Dirichlet L-functions, zeta functions, and related functions
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

\[
x \frac{e^x - 1}{e^x - 1} = \sum_{n=0}^{\infty} B_n \frac{x^n}{n!}.
\]

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).

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.

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.

Bounding

slong bernoulli_bound_2exp_si(ulong n)
Returns an integer \(b\) such that \(|B_n| \leq 2^b\). 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.

```c
void _bernoulli_fmpz_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 \( \text{arbbernoulliui}() \), 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).

```c
void _bernoulli_fmpz_ui(fmpz_t num, fmpz_t den, 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.

### hypgeom.h – support for hypergeometric series

This module provides functions for high-precision evaluation of series of the form

\[
\sum_{k=0}^{n-1} \frac{A(k)}{B(k)} \prod_{j=1}^{k} P(j) \frac{z^k}{Q(k)}
\]

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 \( \bar{P}(k) = P(k)A(k)B(k-1) \), \( \bar{Q}(k) = Q(k)A(k-1)B(k) \). However, separating out \( A \) and \( B \) is convenient and improves efficiency during evaluation.

### 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

\[
F(k) = \frac{1 + \tilde{a}_2/k + \tilde{a}_2/k^2 + \ldots + \tilde{a}_p/k^p}{1 + b_1/k + b_2/k^2 + \ldots + b_q/k^q} = 1 + O(1/k)
\]

8.8. hypgeom.h – support for hypergeometric series 171
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).

**Types, macros and constants**

hypgeom_struct

hypgeom_t

Stores polynomials \( A, B, P, Q \) and precomputed bounds, representing a fixed hypergeometric series.

**Memory management**

void hypgeom_init(hypgeom_t hyp)

void hypgeom_clear(hypgeom_t hyp)

**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 \( z = z(a_p/b_q) \), such that for \( k > K \), the hypergeometric term ratio is bounded by
\[
\frac{k(k-D)}{k^r (k-C)(k-2D)}.
\]

Given this information, we compute a \( \varepsilon \) 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 \( \varepsilon \), and \( n \) is returned.

void hypgeom_precompute(hypgeom_t hyp)
Precomputes the bounds data \( C, D, K \) and an upper bound for \( T(K) \).

### Summation

void arb_hypgeom_sum(arb_t P, arb_t Q, const hypgeom_t hyp, const slong n, slong prec)
Computes \( P/Q = \sum_{k=0}^{n-1} T(k) \) 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 \( P/Q = \sum_{k=0}^{\infty} T(k) \) 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 \( P \).

### 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 \( \sum_{k=N0}^{N} t(n, k) \) 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 \texttt{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 \texttt{flint\_set\_num\_threads()}, the computation time will be reduced by using two threads.

See \texttt{partitions\_hrr\_sum\_arb()} for an explanation of the \texttt{use\_doubles} option.

void \texttt{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 \texttt{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 \texttt{partitions\_hrr\_sum\_arb()}).

void \texttt{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
NINE

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.

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.

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.
Debugging

int arb_calc_verbose
   If set, enables printing information about the calculation to standard output.

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_srpctr
   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 \texttt{maxeval}, the algorithm is terminated and any untested subintervals are added to the output. The total number of calls to \texttt{func} is thereby restricted to a small multiple of \texttt{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 \texttt{maxfound} roots have been isolated. In particular, setting \texttt{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, \texttt{LONG_MAX} may be passed.

The argument \texttt{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 \texttt{maxeval} to exceed \texttt{prec}.

Warning: it is assumed that subdivision points of \texttt{interval} can be represented exactly as floating-point numbers in memory. Do not pass $1 \pm 2^{-10^{100}}$ as input.

\begin{verbatim}
int \texttt{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)
\end{verbatim}

Given an interval \texttt{start} known to contain a single root of \texttt{func}, refines it using \texttt{iter} bisection steps. The algorithm can return a failure code if the sign of the function at an evaluation point is ambiguous. The output \texttt{r} is set to a valid isolating interval (possibly just \texttt{start}) even if the algorithm fails.

\section*{Newton-based root finding}

\begin{verbatim}
void \texttt{arb\_calc\_newton\_conv\_factor}(arf\_t conv\_factor, arb\_calc\_func\_t func, void \* param, const arb\_t conv\_region, slong prec)
\end{verbatim}

Given an interval \texttt{I} specified by \texttt{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 \texttt{func} and \texttt{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 $C$.

\begin{verbatim}
int \texttt{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)
\end{verbatim}

Performs a single step with a single version of Newton’s method. The input consists of the function $f$ specified by \texttt{func} and \texttt{param}, a ball $x = [m - r, m + r]$ known to contain a single root of $f$, a ball $I$ (\texttt{conv\_region}) containing $x$ with an associated bound (\texttt{conv\_factor}) for $C = \sup_{t,u \in I} \frac{1}{2} |f''(t)|/|f'(u)|$, and a working precision \texttt{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 \texttt{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 \texttt{xnew} to $x'$ and return \texttt{ARB\_CALC\_SUCCESS}. If either condition fails, we set \texttt{xnew} to $x$ and return \texttt{ARB\_CALC\_NO\_CONVERGENCE}, indicating that no progress is made.

\begin{verbatim}
int \texttt{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)
\end{verbatim}

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 \texttt{arb\_calc\_newton\_step}, except for the precision parameters: \texttt{prec} is the target accuracy and \texttt{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 \texttt{ARB\_CALC\_SUCCESS} if all attempted Newton steps are successful (note that this does not guarantee that the computed root is accurate to \texttt{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).

### 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). The numerical integration code is described in [Joh2018a].

#### Types, macros and constants

<table>
<thead>
<tr>
<th>acb_calc_func_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typedef for a pointer to a function with signature:</td>
</tr>
<tr>
<td><code>int func(acb_ptr out, const acb_t inp, void * param, slong order, slong prec)</code></td>
</tr>
</tbody>
</table>

implementing a univariate complex function $f(z)$. 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.

When called with `order = 0`, `func` should write to `out` the value of $f(z)$ at the point `inp`, evaluated at a precision of `prec` bits. In this case, $f$ can be an arbitrary complex function, which may have branch cuts or even be non-holomorphic.

When called with `order = n` for $n \geq 1$, `func` should write to `out` the first $n$ coefficients in the Taylor series expansion of $f(z)$ at the point `inp`, evaluated at a precision of `prec` bits. In this case, the implementation of `func` must verify that $f$ is holomorphic on the complex interval defined by $z$, and set the coefficients in `out` to non-finite values otherwise.

For algorithms that do not rely on derivatives, `func` will always get called with `order = 0` or `order = 1`, in which case the user only needs to implement evaluation of the direct function value $f(z)$ (without derivatives). With `order = 1`, `func` must verify holomorphicity (unlike the `order = 0` case).

If $f$ is built from field operations and meromorphic functions, then no special action is necessary when `order` is positive since division by zero or evaluation of built-in functions at poles automatically produces infinite enclosures. However, manual action is needed for bounded functions with branch cuts. For example, when evaluating $\sqrt{z}$, the output must be set to a non-finite value if $z$ overlaps with the branch cut $[-\infty, 0]$. The easiest way to accomplish this is to use versions of basic functions (sqrt, log, pow, etc.) that test holomorphicity of their arguments individually.

Some functions with branch cut detection are available as builtins: see `acb_sqrt_analytic()`, `acb_rsqrt_analytic()`, `acb_log_analytic()`, `acb_pow_analytic()`. It is not difficult to write custom functions of this type, using the following pattern:

```c
/* Square root function on C with detection of the branch cut. */
void sqrt_analytic(acb_t res, const acb_t z, int analytic, slong prec)
{
    if (analytic &&
        arb_contains_zero(acb_imagref(z)) &&
        arb_contains_nonpositive(acb_realref(z)))
    {
        acb_indeterminate(res);
    }
    else
    {
        acb_sqrt(res, z, prec);
    }
}
```
The built-in methods \texttt{acb_real_abs()}, \texttt{acb_real_sgn()}, \texttt{acb_real_heaviside()}, \texttt{acb_real_floor()}, \texttt{acb_real_ceil()}, \texttt{acb_real_max()}, \texttt{acb_real_min()} provide piecewise holomorphic functions that are useful for integrating piecewise-defined real functions.

For example, here we define a piecewise holomorphic extension of the function \( f(z) = \sqrt{\lfloor z \rfloor} \) (for simplicity, without implementing derivatives):

```c
int func(acb_ptr out, const acb_t inp, void * param, slong order, slong prec)
{
    if (order > 1) flint_abort(); /* derivatives not implemented */
    acb_real_floor(out, inp, order != 0, prec);
    acb_sqrt_analytic(out, out, order != 0, prec);
    return 0;
}
```

(Here, \texttt{acb_real_sqrtpos()} may be slightly better if it is known that \( z \) will be nonnegative on the path.)

See the demo program \texttt{examples/integrals.c} for more examples.

\section*{Integration}

\textbf{int \texttt{acb_calc_integrate}(acb_t res, acb_calc_func_t func, void * param, const acb_t a, const acb_t b, slong rel_goal, const mag_t abs_tol, const acb_calc_integrate_opt_t options, slong prec)}

Computes a rigorous enclosure of the integral

\[ I = \int_a^b f(t)dt \]

where \( f \) is specified by \( (\text{func}, \text{param}) \), following a straight-line path between the complex numbers \( a \) and \( b \). For finite results, \( a, b \) must be finite and \( f \) must be bounded on the path of integration. To compute improper integrals, the user should therefore truncate the path of integration manually (or make a regularizing change of variables, if possible). Returns ARB\_CALC\_SUCCESS if the integration converged to the target accuracy on all subintervals, and returns ARB\_CALC\_NO\_CONVERGENCE otherwise.

By default, the integrand \texttt{func} will only be called with \texttt{order} = 0 or \texttt{order} = 1; that is, derivatives are not required.

- The integrand will be called with \texttt{order} = 0 to evaluate \( f \) normally on the integration path (either at a single point or on a subinterval). In this case, \( f \) is treated as a pointwise defined function and can have arbitrary discontinuities.
- The integrand will be called with \texttt{order} = 1 to evaluate \( f \) on a domain surrounding a segment of the integration path for the purpose of bounding the error of a quadrature formula. In this case, \texttt{func} must verify that \( f \) is holomorphic on this domain (and output a non-finite value if it is not).

The integration algorithm combines direct interval enclosures, Gauss-Legendre quadrature where \( f \) is holomorphic, and adaptive subdivision. This strategy supports integrands with discontinuities while providing exponential convergence for typical piecewise holomorphic integrands.

The following parameters control accuracy:

- \texttt{rel_goal} - relative accuracy goal as a number of bits, i.e. target a relative error less than \( \varepsilon_{rel} = 2^{-r} \) where \( r = \text{rel}\_goal \) (note the sign: \texttt{rel}\_goal should be nonnegative).
- \texttt{abs_tol} - absolute accuracy goal as a \texttt{mag_t} describing the error tolerance, i.e. target an absolute error less than \( \varepsilon_{abs} = \text{abs}\_tol \).
- \texttt{prec} - working precision. This is the working precision used to evaluate the integrand and manipulate interval endpoints. As currently implemented, the algorithm does not attempt to
adjust the working precision by itself, and adaptive control of the working precision must be handled by the user.

For typical usage, set \( \text{rel\_goal} = \text{prec} \) and \( \text{abs\_tol} = 2^{-\text{prec}} \). It usually only makes sense to have \( \text{rel\_goal} \) between 0 and \( \text{prec} \).

The algorithm attempts to achieve an error of \( \max(\varepsilon_{\text{abs}}, M \varepsilon_{\text{rel}}) \) on each subinterval, where \( M \) is the magnitude of the integral. These parameters are only guidelines; the cumulative error may be larger than both the prescribed absolute and relative error goals, depending on the number of subdivisions, cancellation between segments of the integral, and numerical errors in the evaluation of the integrand.

To compute tiny integrals with high relative accuracy, one should set \( \varepsilon_{\text{abs}} \approx M \varepsilon_{\text{rel}} \) where \( M \) is a known estimate of the magnitude. Setting \( \varepsilon_{\text{abs}} \) to 0 is also allowed, forcing use of a relative instead of an absolute tolerance goal. This can be handy for exponentially small or large functions of unknown magnitude. It is recommended to avoid setting \( \varepsilon_{\text{abs}} \) very small if possible since the algorithm might need many extra subdivisions to estimate \( M \) automatically; if the approximate magnitude can be estimated by some external means (for example if a midpoint-width or endpoint-width estimate is known to be accurate), providing an appropriate \( \varepsilon_{\text{abs}} \approx M \varepsilon_{\text{rel}} \) will be more efficient.

If the integral has very large magnitude, setting the absolute tolerance to a corresponding large value is recommended for best performance, but it is not necessary for convergence since the absolute tolerance is increased automatically during the execution of the algorithm if the partial integrals are found to have larger error.

Additional options for the integration can be provided via the \textit{options} parameter (documented below). To use all defaults, \texttt{NULL} can be passed for \textit{options}.

**Options for integration**

\texttt{acb\_calc\_integrate\_opt\_struct}

\texttt{acb\_calc\_integrate\_opt\_t}

This structure contains several fields, explained below. An \texttt{acb\_calc\_integrate\_opt\_t} is defined as an array of \texttt{acb\_calc\_integrate\_opt\_struct} of length 1, permitting it to be passed by reference. An \texttt{acb\_calc\_integrate\_opt\_t} must be initialized before use, which sets all fields to 0 or \texttt{NULL}.

For fields that have not been set to other values, the integration algorithm will choose defaults automatically (based on the precision and accuracy goals). This structure will most likely be extended in the future to accommodate more options.

\texttt{slong deg\_limit}

Maximum quadrature degree for each subinterval. If a zero or negative value is provided, the limit is set to a default value which currently equals \( 0.5 \cdot \min(\text{prec}, \text{rel\_goal}) + 60 \) for Gauss-Legendre quadrature. A higher quadrature degree can be beneficial for functions that are holomorphic on a large domain around the integration path and yet behave irregularly, such as oscillatory entire functions. The drawback of increasing the degree is that the precomputation time for quadrature nodes increases.

\texttt{slong eval\_limit}

Maximum number of function evaluations. If a zero or negative value is provided, the limit is set to a default value which currently equals \( 1000 \cdot \text{prec} + \text{prec}^2 \). This is the main parameter used to limit the amount of work before aborting due to possible slow convergence or non-convergence. A lower limit allows aborting faster. A higher limit may be needed for integrands with many discontinuities or many singularities close to the integration path. This limit is only taken as a rough guideline, and the actual number of function evaluations may be slightly higher depending on the actual subdivisions.

\texttt{slong depth\_limit}

Maximum search depth for adaptive subdivision. Technically, this is not the limit on the local bisection depth but the limit on the number of simultaneously queued subintervals. If a zero or negative value is provided, the limit is set to the default value \( 2 \cdot \text{prec} \). Warning: memory usage may increase in proportion to this limit.
int use_heap
By default (if set to 0), new subintervals generated by adaptive bisection will be appended to the top of a stack. If set to 1, a binary heap will be used to maintain a priority queue where the subintervals with larger error have higher priority. This sometimes gives better results in case of convergence failure, but can lead to a much larger array of subintervals (requiring a higher depth_limit) when many global bisections are needed.

int verbose
If set to 1, some information about the overall integration process is printed to standard output. If set to 2, information about each subinterval is printed.

void acb_calc_integrate_opt_init(acb_calc_integrate_opt_t options)
Initializes options for use, setting all fields to 0 indicating default values.

Local integration algorithms

int acb_calc_integrate_gl_auto_deg(acb_t res, slong * num_eval, acb_calc_func_t func, void * param, const acb_t a, const acb_t b, const mag_t tol, slong deg_limit, int flags, slong prec)
Attempts to compute \( I = \int_{a}^{b} f(t)dt \) using a single application of Gauss-Legendre quadrature with automatic determination of the quadrature degree so that the error is smaller than tol. Returns ARB_CALC_SUCCESS if the integral has been evaluated successfully or ARB_CALC_NO_CONVERGENCE if the tolerance could not be met. The total number of function evaluations is written to num_eval.

For the interval \([-1, 1]\), the error of the \(n\)-point Gauss-Legendre rule is bounded by
\[
| I - \sum_{k=0}^{n-1} w_k f(x_k) | \leq \frac{64M}{15(p-1)p^{2n-1}}
\]
if \(f\) is holomorphic with \(|f(z)| \leq M\) inside the ellipse with foci \(\pm 1\) and semiaxes \(X\) and \(Y = \sqrt{X^2 - 1}\) such that \(\rho = X + Y\) with \(\rho > 1\) [Tre2008].

For an arbitrary interval, we use \( \int_{a}^{b} f(t)dt = \int_{-1}^{1} g(t)dt \) where \(g(t) = \Delta f(\Delta t + m), \Delta = \frac{1}{2}(b - a), m = \frac{1}{2}(a + b)\). With \(I = [\pm X] + [\pm Y]i\), this means that we evaluate \(\Delta f(\Delta t + m)\) to get the bound \(M\). (An improvement would be to reduce the wrapping effect of rotating the ellipse when the path is not rectilinear).

We search for an \(X\) that makes the error small by trying steps \(2^n\). Larger \(X\) will give smaller \(1/\rho^{2n-1}\) but larger \(M\). If we try successive larger values of \(k\), we can abort when \(M = \infty\) since this either means that we have hit a singularity or a branch cut or that overestimation in the evaluation of \(f\) is becoming too severe.

Integration (old)

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
\[
C(x, r) = \frac{1}{2\pi r} \int_{|z-x|=r} |f(z)|dz = \int_{0}^{1} |f(x + re^{2\pi it})|dt
\]
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.
Computes the integral
\[ I = \int_{a}^{b} f(t) dt \]
where \( f \) is specified by \((\text{func}, \text{param})\), 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
\[
\int f(m + x) = \left( \sum_{n=0}^{N-1} \frac{a_n x^{n+1}}{n+1} \right) + \left( \sum_{n=N}^{\infty} \frac{a_n x^{n+1}}{n+1} \right).
\]
For sufficiently small \( x \), the second series converges and its absolute value is bounded by
\[
\sum_{n=N}^{\infty} \frac{C(m, R) |x|^{n+1}}{R^n (N + 1)} = \frac{C(m, R) R x}{(R - x)(N + 1)} \left( \frac{x}{R} \right)^N.
\]
It is required that any singularities of \( f \) are isolated from the path of integration by a distance strictly greater than the positive value \( \text{outer}_\text{radius} \) (which is the integration radius used for the Cauchy bound). Taylor series step lengths are chosen so as not to exceed \( \text{inner}_\text{radius} \), which must be strictly smaller than \( \text{outer}_\text{radius} \) for convergence. A smaller \( \text{inner}_\text{radius} \) 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 \( \text{inner}_\text{radius} \) to half the value of \( \text{outer}_\text{radius} \), 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 \( \text{accuracy}_\text{goal} \) (in the future, this might change to a relative accuracy). Arithmetic operations and function evaluations are performed at a precision of \( \text{prec} \) 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.
Mainly for internal use.

**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.

**Memory-related methods**

```c
slong fmpz_allocated_bytes(const fmpz_t x)
```

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(fmpz)` to get the size of the object as a whole.

**Convenience methods**

```c
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`.

```c
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.

```c
void fmpz_ui_mul_ui(fmpz_t x, ulong a, ulong b)
```

Sets `x` to `a` times `b`.

```c
void fmpz_ui_pow_ui(fmpz_t x, ulong b, ulong e)
```

Sets `x` to `b` raised to the power `e`.

```c
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`.

**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 \texttt{fmpz_add_inline} (\texttt{fmpz} z, const \texttt{fmpz} x, const \texttt{fmpz} y)

void \texttt{fmpz_add_si_inline} (\texttt{fmpz} z, const \texttt{fmpz} x, \texttt{slong} y)

void \texttt{fmpz_add_ui_inline} (\texttt{fmpz} z, const \texttt{fmpz} x, \texttt{ulong} y)

Sets \texttt{z} to the sum of \texttt{x} and \texttt{y}.

void \texttt{fmpz_sub_si_inline} (\texttt{fmpz} z, const \texttt{fmpz} x, \texttt{slong} y)

Sets \texttt{z} to the difference of \texttt{x} and \texttt{y}.

\texttt{mp_size_t} \texttt{_fmpz_size} (const \texttt{fmpz} x)

Returns the number of limbs required to represent \texttt{x}.

\texttt{slong} \texttt{_fmpz_sub_small} (const \texttt{fmpz} x, const \texttt{fmpz} y)

Computes the difference of \texttt{x} and \texttt{y} and returns the result as an \texttt{slong}. The result is clamped between \(-\text{WORD\_MAX}\) and \text{WORD\_MAX}, i.e. between \(\pm (2^{63} - 1)\) inclusive on a 64-bit machine.

\textbf{Low-level conversions}

void \texttt{fmpz_set_mpn_large} (\texttt{fmpz} z, \texttt{mp} \texttt{srcptr} src, \texttt{mp} \texttt{size_t} n, \texttt{int} negative)

Sets \texttt{z} to the integer represented by the \texttt{n} limbs in the array \texttt{src}, or minus this value if \texttt{negative} is \textsl{1}. Requires \texttt{n} \(\geq 2\) and that the top limb of \texttt{src} is nonzero. Note that \texttt{fmpz_set_ui}, \texttt{fmpz_neg_ui} can be used for single-limb integers.

void \texttt{FMPZ\_GET\_MPN\_READONLY} (\texttt{sSIGN}, \texttt{zn}, \texttt{zptr}, \texttt{ztmp}, \texttt{zv})

Given an \texttt{fmpz} \texttt{zv}, this macro sets \texttt{zptr} to a pointer to the limbs of \texttt{zv}, \texttt{zn} to the number of limbs, and \texttt{zsign} to a sign bit (\textsl{0} if nonnegative, \textsl{1} if negative). The variable \texttt{ztmp} must be a single \texttt{mp\_limb\_t}, which is used as a buffer. If \texttt{zv} is a small value, \texttt{zv} itself contains no limb array that \texttt{zptr} could point to, so the single limb is copied to \texttt{ztmp} and \texttt{zptr} is set to point to \texttt{ztmp}. The case where \texttt{zv} is zero is not handled specially, and \texttt{zn} is set to \textsl{1}.

void \texttt{fmpz\_lshift\_mpn} (\texttt{fmpz} z, \texttt{mp} \texttt{srcptr} src, \texttt{mp} \texttt{size_t} n, \texttt{int} negative, \texttt{mp} \texttt{bitcnt_t} shift)

Sets \texttt{z} to the integer represented by the \texttt{n} limbs in the array \texttt{src}, or minus this value if \texttt{negative} is \textsl{1}, shifted left by \texttt{shift} bits. Requires \texttt{n} \(\geq 1\) and that the top limb of \texttt{src} is nonzero.

\texttt{bool\_mat.h} – matrices over booleans

A \textit{bool\_mat\_t} represents a dense matrix over the boolean semiring \(\langle \{0,1\}, \lor, \land \rangle\), implemented as an array of entries of type \texttt{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.

\textbf{Types, macros and constants}

\texttt{bool\_mat\_struct}

\texttt{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 \texttt{bool\_mat\_t} is defined as an array of length one of type \texttt{bool\_mat\_struct}, permitting an \texttt{bool\_mat\_t} to be passed by reference.

\texttt{int} \texttt{bool\_mat\_get\_entry} (const \texttt{bool\_mat\_t} mat, \texttt{slong} i, \texttt{slong} j)

Returns the entry of matrix \texttt{mat} at row \texttt{i} and column \texttt{j}. 

184 Chapter 10. Extra utility modules
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.

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`.

Conversions

void `bool_mat_set`(bool_mat_t dest, const bool_mat_t src)  
Sets `dest` to `src`. The operands must have identical dimensions.

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`.

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 ≠ j ⇒ A_{ij}`.

int `bool_mat_is_lower_triangular`(const bool_mat_t A)  
Returns nonzero iff `i < j ⇒ A_{ij}`.

int `bool_mat_is_transitive`(const bool_mat_t mat)  
Returns nonzero iff `A_{ij} ∧ A_{jk} ⇒ A_{ik}`.

int `bool_mat_is_nilpotent`(const bool_mat_t A)  
Returns nonzero iff some positive matrix power of `A` is zero.
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.

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.

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.

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 \( \overline{A}_{ij} \). 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.

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 indicator 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 $N > B_{ij}$.

dlog.h – discrete logarithms mod ulong primes

This module implements discrete logarithms, with the application to Dirichlet characters in mind.

In particular, this module defines a dlog_precomp_t structure permitting to describe a discrete log problem in some subgroup of $(\mathbb{Z}/p^e\mathbb{Z})^\times$ for primepower moduli $p^e$, and store precomputed data for faster computation of several such discrete logarithms.

When initializing this data, the user provides both a group description and the expected number of subsequent discrete logarithms calls. The choice of algorithm and the amount of stored data depend both on the structure of the group and this number.

No particular effort has been made towards single discrete logarithm computation. Currently only machine size primepower moduli are supported.

Types, macros and constants

DLOG_NONE

Return value when the discrete logarithm does not exist

dlog_precomp_struct
dlog_precomp_t

Structure for discrete logarithm precomputed data.

A dlog_precomp_t is defined as an array of length one of type dlog_precomp_struct, permitting a dlog_precomp_t to be passed by reference.
Single evaluation

ulong dlog_once(ulong b, ulong a, const nmod_t mod, ulong n)
Return $x$ such that $b = a^x \pmod{\mod^n}$, where $a$ is known to have order $n$.

Precomputations

void dlog_precomp_n_init(dlog_precomp_t pre, ulong a, ulong mod, ulong n, ulong num)
Precompute data for $\num$ discrete logarithms evaluations in the subgroup generated by $a$ modulo $mod$, where $a$ is known to have order $n$.

ulong dlog_precomp(const dlog_precomp_t pre, ulong b)
Return $\log(b)$ for the group described in $pre$.

void dlog_precomp_clear(dlog_precomp_t pre)
Clears $t$.

Specialized versions of $dlog_precomp_n_init()$ are available when specific information is known about the group:

void dlog_precomp_modpe_init(dlog_precomp_t pre, ulong a, ulong p, ulong e, ulong pe, ulong num)
Assume that $a$ generates the group of residues modulo $pe$ equal $p^e$ for prime $p$.

void dlog_precomp_p_init(dlog_precomp_t pre, ulong a, ulong mod, ulong p, ulong num)
Assume that $a$ has prime order $p$.

void dlog_precomp_pe_init(dlog_precomp_t pre, ulong a, ulong mod, ulong p, ulong e, ulong pe, ulong num)
Assume that $a$ has primepower order $p^e$.

void dlog_precomp_small_init(dlog_precomp_t pre, ulong a, ulong mod, ulong n, ulong num)
Make a complete lookup table of size $n$. If $mod$ is small, this is done using an element-indexed array (see $dlog_table_t$), otherwise with a sorted array allowing binary search.

Vector evaluations

These functions compute all logarithms of successive integers $1 \ldots n$.

void dlog_vec_fill(ulong * v, ulong nv, ulong x)
Sets values $v[k]$ to $x$ for all $k$ less than $nv$.

void dlog_vec_set_not_found(ulong * v, ulong nv, nmod_t mod)
Sets values $v[k]$ to $DLOG_NONE$ for all $k$ not coprime to $mod$.

void dlog_vec(ulong * v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)
Sets $v[k]$ to $\log(k, a) \times va$ for $0 \leq k < nv$, where $a$ has order $na$. $va$ should be $1$ for usual log computation.

void dlog_vec_add(ulong * v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)
Same parameters as before, but adds $\log(k, a) \times va$ to $v[k]$ and reduce modulo $order$ instead of replacing the value. Indices $k$ such that $v[k]$ equals $DLOG_NONE$ are ignored.

Depending on the relative size of $nv$ and $na$, these two $dlog_vec$ functions call one of the following functions.

void dlog_vec_loop(ulong * v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)
void dlog_vec_loop_add(ulong * v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)
Perform a complete loop of size $na$ on powers of $a$ to fill the logarithm values, discarding powers outside the bounds of $v$. This requires no discrete logarithm computation.
void dlog_vec_eratos(ulong * v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)
Compute discrete logarithms of prime numbers less than \( nv \) and propagate to composite numbers.

void dlog_vec_eratos_add(ulong * v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)

void dlog_vec_sieve_add(ulong * v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)

void dlog_vec_sieve(ulong * v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)
Compute the discrete logarithms of the first few prime numbers, then use them as a factor base to obtain the logarithms of larger primes by sieving techniques.

In the present implementation, the full index-calculus method is not implemented.

**Internal discrete logarithm strategies**

Several discrete logarithms strategies are implemented:

- Complete lookup table for small groups.
- Baby-step giant-step table.

combined with mathematical reductions:

- Pohlig-Hellman decomposition (Chinese remainder decomposition on the order of the group and base \( p \) decomposition for primepower order).
- \( p \)-adic log for primepower modulus \( p^e \).

The `dlog_precomp` structure makes recursive use of the following method-specific structures.

**Complete table**

dlog_table
dlog_table_t

Structure for complete lookup table.

ulong dlog_table_init(dlog_table_t t, ulong a, ulong mod)
Initialize a table of powers of \( a \) modulo \( mod \), storing all elements in an array of size \( mod \).

void dlog_table_clear(dlog_table_t t)
Clears \( t \).

ulong dlog_table(dlog_table_t t, ulong b)
Return \( \log(b, a) \) using the precomputed data \( t \).

**Baby-step giant-step table**

dlog_bsgs_struct
dlog_bsgs_t

Structure for Baby-Step Giant-Step decomposition.

ulong dlog_bsgs_init(dlog_bsgs_t t, ulong a, ulong mod, ulong n, ulong m)
Initialize \( t \) and store the first \( m \) powers of \( a \) in a sorted array. The return value is a rough measure of the cost of each logarithm using this table. The user should take \( m \approx \sqrt{k n} \) to compute \( k \) logarithms in a group of size \( n \).

void dlog_bsgs_clear(dlog_bsgs_t t)
Clears \( t \).
ulong dlog_bsgs(dlog_bsgs_t t, ulong b)
    Return log(b, a) using the precomputed data t.

Prime-power modulus decomposition

dlog_modpe_struct

dlog_modpe_t
    Structure for discrete logarithm modulo primepower $p^e$.
    A dlog_modpe_t is defined as an array of length one of type dlog_modpe_struct, permitting a
dlog_modpe_t to be passed by reference.

ulong dlog_modpe_init(dlog_modpe_t t, ulong a, ulong p, ulong e, ulong pe, ulong num)
void dlog_modpe_clear(dlog_modpe_t t)
    Clears t.

ulong dlog_modpe(dlog_modpe_t t, ulong b)
    Return log(b, a) using the precomputed data t.

CRT decomposition

dlog_crt_struct

dlog_crt_t
    Structure for discrete logarithm for groups of composite order. A dlog_crt_t is defined as an
array of length one of type dlog_crt_struct, permitting a dlog_crt_t to be passed by reference.

ulong dlog_crt_init(dlog_crt_t t, ulong a, ulong mod, ulong n, ulong num)
    Precompute data for num evaluations of discrete logarithms in base $a$ modulo mod, where $a$ has
composite order $n$, using chinese remainder decomposition.

void dlog_crt_clear(dlog_crt_t t)
    Clears t.

ulong dlog_crt(dlog_crt_t t, ulong b)
    Return log(b, a) using the precomputed data t.

padic decomposition

dlog_power_struct

dlog_power_t
    Structure for discrete logarithm for groups of primepower order. A dlog_power_t is defined as an
array of length one of type dlog_power_struct, permitting a dlog_power_t to be passed by reference.

ulong dlog_power_init(dlog_power_t t, ulong a, ulong mod, ulong p, ulong e, ulong num)
    Precompute data for num evaluations of discrete logarithms in base $a$ modulo mod, where $a$ has
prime power order $pe$ equals $p^e$, using decomposition in base $p$.

void dlog_power_clear(dlog_power_t t)
    Clears t.

ulong dlog_power(dlog_power_t t, ulong b)
    Return log(b, a) using the precomputed data t.
Pollard rho method

dlog_rho_struct

dlog_rho_t
  Structure for discrete logarithm using Pollard rho. A dlog_rho_t is defined as an array of length
  one of type dlog_rho_struct, permitting a dlog_rho_t to be passed by reference.

ulong dlog_rho_init(dlog_rho_t t, ulong a, ulong mod, ulong n, ulong num)
  Initialize random walks for evaluations of discrete logarithms in base \(a\) modulo \(mod\), where \(a\) has
  order \(n\).

void dlog_rho_clear(dlog_rho_t t)
  Clears \(t\).

ulong dlog_rho(dlog_rho_t t, ulong b)
  Return \(\log(b, a)\) by the rho method in the group described by \(t\).

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.

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.

**Memory management**

```c
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.
```

**Special values**

```c
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 \( \text{fmpr_is_\text{inf}}() \).)

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)
    Sets \( y \) to a copy of \( x \) rounded in the direction specified by \( \text{rnd} \) to the number of bits specified by \( \text{prec} \).

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 \( \text{negative} \) is nonzero), sets \( \text{man} \) to the corresponding floating-point mantissa rounded to \( \text{prec} \) bits in direction \( \text{rnd} \), sets \( \text{shift} \) 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 \( \text{exp} \), negating the value if \( \text{negative} \) is nonzero, and rounding the result to \( \text{prec} \) bits in direction \( \text{rnd} \).

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 \( \text{exp} \), negating the value if \( \text{negative} \) is nonzero, and rounding the result to \( \text{prec} \) bits in direction \( \text{rnd} \).

void fmpr_set_error_result(fmpr_t err, const fmpr_t result, slong rret)
    Given the return value \( \text{rret} \) and output variable \( \text{result} \) from a function performing a rounding (e.g. \( \text{fmpr_set_round} \) or \( \text{fmpr_add} \)), sets \( \text{err} \) 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 \( \text{fmpr_set_error_result} \), but adds \( \text{err_in} \) 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 \( \text{fmpr_set_error_result}() \) gives exactly 0 or 1 ulp as expected.
int \texttt{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 \texttt{fmpr\_set\_mpfr(fmpr\_t x, const mpfr\_t y)}

Sets \(x\) to the exact value of the MPFR variable \(y\).

double \texttt{fmpr\_get\_d(const fmpr\_t x, mpfr\_rnd\_t rnd)}

Returns \(x\) rounded to a \textit{double} in the direction specified by \textit{rnd}.

void \texttt{fmpr\_set\_d(fmpr\_t x, double v)}

Sets \(x\) the the exact value of the argument \(v\) of type \textit{double}.

void \texttt{fmpr\_set\_ui(fmpr\_t x, ulong c)}

Sets \(x\) exactly to the integer \(c\).

void \texttt{fmpr\_set\_si(fmpr\_t x, slong c)}

Sets \(x\) exactly to the integer \(c\).

void \texttt{fmpr\_set\_fmpz(fmpr\_t x, const fmpz\_t c)}

Sets \(x\) exactly to the integer \(c\).

void \texttt{fmpr\_set\_fmpz_2exp(fmpr\_t x, const fmpz\_t man, const fmpz\_t exp)}

Sets \(x\) to \(\text{man} \times 2^\text{exp}\).

void \texttt{fmpr\_set\_si_2exp\_si(fmpr\_t x, slong man, slong exp)}

Sets \(x\) to \(\text{man} \times 2^\text{exp}\).

void \texttt{fmpr\_set\_ui_2exp\_si(fmpr\_t x, ulong man, slong exp)}

Sets \(x\) to \(\text{man} \times 2^\text{exp}\).

void \texttt{fmpr\_get\_fmpz\_2exp(fmpz\_t man, fmpz\_t exp), fmpr\_rnd\_t rnd)}

Sets \(x\) to \(\text{man} \times 2^\text{exp}\), rounded according to \textit{prec} and \textit{rnd}.

double \texttt{fmpr\_get\_d(const fmpr\_t x, mpfr\_ rnd\_t rnd)}

Returns \(x\) rounded to a \textit{double} in the direction specified by \textit{rnd}.

void \texttt{fmpr\_set\_d(fmpr\_t x, double v)}

Sets \(x\) the the exact value of the argument \(v\) of type \textit{double}.

void \texttt{fmpr\_set\_ui(fmpr\_t x, ulong c)}

Sets \(x\) the the exact value of the argument \(v\) of type \textit{double}.

void \texttt{fmpr\_set\_fmpz_2exp(fmpr\_t x, const fmpz\_t man, const fmpz\_t exp)}

Sets \(x\) to \(\text{man} \times 2^\text{exp}\).

void \texttt{fmpr\_set\_si_2exp\_si(fmpr\_t x, slong man, slong exp)}

Sets \(x\) to \(\text{man} \times 2^\text{exp}\).

void \texttt{fmpr\_set\_ui_2exp\_si(fmpr\_t x, ulong man, slong exp)}

Sets \(x\) to \(\text{man} \times 2^\text{exp}\).

\begin{verbatim}
void \texttt{fmpr\_get\_fmpz\_fixed\_fmpz(fmpz\_t y, const fmpr\_t x, const fmpz\_t 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.

int \texttt{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.
\end{verbatim}
int fmp_r_cmp(const fmp_r_t x, const fmp_r_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 fmp_r_cmpabs(const fmp_r_t x, const fmp_r_t y)
int fmp_r_cmpabs_ui(const fmp_r_t x, ulong y)
    Compares the absolute values of x and y.

int fmp_r_cmp_2exp_si(const fmp_r_t x, slong e)
int fmp_r_cmpabs_2exp_si(const fmp_r_t x, slong e)
    Compares x (respectively its absolute value) with $2^e$.

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

void fmp_r_min(fmp_r_t z, const fmp_r_t a, const fmp_r_t b)
void fmp_r_max(fmp_r_t z, const fmp_r_t a, const fmp_r_t b)
    Sets z respectively to the minimum and the maximum of a and b.

slong fmp_r_bits(const fmp_r_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 fmp_r_is_int(const fmp_r_t x)
    Returns nonzero iff x is integer-valued.

int fmp_r_is_int_2exp_si(const fmp_r_t x, slong e)
    Returns nonzero iff x equals $n2^e$ for some integer $n$.

void fmp_r_abs_bound_le_2exp_fmpz(fmpz_t b, const fmp_r_t x)
    Sets b to the smallest integer such that $|x| \leq 2^b$. If x is zero, infinity or NaN, the result is undefined.

void fmp_r_abs_bound_lt_2exp_fmpz(fmpz_t b, const fmp_r_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 fmp_r_abs_bound_lt_2exp_si(const fmp_r_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.

Random number generation

void fmp_r_randtest(fmp_r_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 fmp_r_randtest_not_zero(fmp_r_t x, flint_rand_t state, slong bits, slong mag_bits)
    Identical to fmp_r_randtest, except that zero is never produced as an output.

void fmp_r_randtest_special(fmp_r_t x, flint_rand_t state, slong bits, slong mag_bits)
    Identical to fmp_r_randtest, except that the output occasionally is set to an infinity or NaN.

Input and output

void fmp_r_print(const fmp_r_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.

Arithmetic

void fmpr_neg(const fmpr_t y, const fmpr_t x)
    Sets y to the negation of x.

slong fmpr_neg_round(const 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(const fmpr_t y, const fmpr_t x)
    Sets y to the absolute value of x.

slong fmpr_add(const fmpr_t z, const fmpr_t x, const fmpr_t y, slong prec, fmpr_rnd_t rnd)

slong fmpr_add_ui(const fmpr_t z, const fmpr_t x, ulong y, slong prec, fmpr_rnd_t rnd)

slong fmpr_add_si(const fmpr_t z, const fmpr_t x, slong y, slong prec, fmpr_rnd_t rnd)

slong fmpr_add_fmpz(const 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(const 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(const fmpr_t z, const fmpr_t x, const fmpr_t y, slong prec, fmpr_rnd_t rnd)

slong fmpr_sub_ui(const fmpr_t z, const fmpr_t x, ulong y, slong prec, fmpr_rnd_t rnd)

slong fmpr_sub_si(const fmpr_t z, const fmpr_t x, slong y, slong prec, fmpr_rnd_t rnd)

slong fmpr_sub_fmpz(const 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(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(const fmpr_t z, const fmpr_t x, const fmpr_t y, slong prec, fmpr_rnd_t rnd)

slong fmpr_mul_ui(const fmpr_t z, const fmpr_t x, ulong y, slong prec, fmpr_rnd_t rnd)

slong fmpr_mul_si(const fmpr_t z, const fmpr_t x, slong y, slong prec, fmpr_rnd_t rnd)

slong fmpr_mul_fmpz(const 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 multiplication, provided that the result fits in memory.

void fmpr_mul_2exp_si(const fmpr_t y, const fmpr_t x, slong e)
    Sets y to x multiplied by 2^e without rounding.

slong fmpr_div(const fmpr_t z, const fmpr_t x, const fmpr_t y, slong prec, fmpr_rnd_t rnd)

slong fmpr_div_ui(const fmpr_t z, const fmpr_t x, ulong y, slong prec, fmpr_rnd_t rnd)

slong fmpr_ui_div(const fmpr_t z, const fmpz_t y, slong prec, fmpr_rnd_t rnd)

slong fmpr_div_si(const fmpr_t z, const fmpr_t x, slong y, slong prec, fmpr_rnd_t rnd)
```c
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, 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*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, 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*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 y, const fmpr_t x, ulong prec, fmpr_rnd_t rnd)

slong fmpr_sqrt_fmpz(fmpr_t y, 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.

**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 \texttt{fmpr\_log1p}(\texttt{fmpr\_t} \ y, \:\texttt{const \ fmpr\_t} \ x, \:\texttt{slong \ prec, \ fmpr\_rnd\_t \ rnd})

Sets \(y\) to \(\log(1 + x)\), rounded according to \(\text{prec}\) and \(\text{rnd}\). 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 \texttt{fmpr\_exp}(\texttt{fmpr\_t} \ y, \:\texttt{const \ fmpr\_t} \ x, \:\texttt{slong \ prec, \ fmpr\_rnd\_t \ rnd})

Sets \(y\) to \(\exp(x)\), rounded according to \(\text{prec}\) and \(\text{rnd}\). This function is currently implemented using MPFR and does not support large exponents.

slong \texttt{fmpr\_expm1}(\texttt{fmpr\_t} \ y, \:\texttt{const \ fmpr\_t} \ x, \:\texttt{slong \ prec, \ fmpr\_rnd\_t \ rnd})

Sets \(y\) to \(\exp(x) - 1\), rounded according to \(\text{prec}\) and \(\text{rnd}\). This function computes an accurate value when \(x\) is small. This function is currently implemented using MPFR and does not support large exponents.
Here, we give extra proofs, error bounds, and formulas that would be too lengthy to reproduce in the documentation for each module.

General formulas and bounds

This section collects some results from real and complex analysis that are useful when deriving error bounds. Beware of typos.

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 \( |a| \leq r \). 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):**

\[
|f(x + a) - f(x)| \leq \min(2C_0, C_1|a|)
\]

where

\[
C_0 = \sup_{|t| \leq |a|} |f(x + t)|, \quad C_1 = \sup_{|t| \leq |a|} |f'(x + t)|.
\]

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 \( a, b \in \mathbb{C} \),

\[
|(x + a)(y + b) - xy| \leq |xb| + |ya| + |ab|.
\]

**Theorem (quotient):** For \( x, y \in \mathbb{C} \) and \( a, b \in \mathbb{C} \) with \( |b| < |y| \),

\[
\left| \frac{x}{y} - \frac{x + a}{y + b} \right| \leq \frac{|xb| + |ya|}{|y|(|y| - |b|)}.
\]

**Theorem (square root):** For \( x, a \in \mathbb{R} \) with \( 0 \leq |a| \leq x \),

\[
\sqrt{x + a} - \sqrt{x} \leq \sqrt{x} \left( 1 - \sqrt{1 - \frac{|a|}{x}} \right) \leq \frac{\sqrt{x}}{2} \left( \frac{|a|}{x} + \frac{|a|^2}{x^2} \right)
\]

where the first inequality is an equality if \( a \leq 0 \). (When \( x = a = 0 \), the limiting value is 0.)

**Theorem (reciprocal square root):** For \( x, a \in \mathbb{R} \) with \( 0 \leq |a| < x \),

\[
\left| \frac{1}{\sqrt{x + a}} - \frac{1}{\sqrt{x}} \right| \leq \frac{|a|}{2(x - |a|)^{3/2}}.
\]
Theorem (k-th root): For $k > 1$ and $x, a \in \mathbb{R}$ with $0 \leq |a| \leq x$,

$$\left|(x + a)^{1/k} - x^{1/k}\right| \leq x^{1/k} \min\left(1, \frac{1}{k} \log\left(1 + \frac{|a|}{x - |a|}\right)\right).$$

Proof: The error is largest when $a = -r$ is negative, and

$$x^{1/k} - (x - r)^{1/k} = x^{1/k}[1 - \exp((1 - r/x)/k)] \leq x^{1/k} \min(1, -\log(1 - r/x)/k) = x^{1/k} \min(1, \log(1 + r/(x - r))/k).$$

Theorem (sine, cosine): For $x, a \in \mathbb{R}$,

$$|\sin(x + a) - \sin(x)| \leq \min(2, |a|).$$

Theorem (logarithm): For $x, a \in \mathbb{R}$ with $0 \leq |a| < x$,

$$|\log(x + a) - \log(x)| \leq \log\left(1 + \frac{|a|}{x - |a|}\right),$$

with equality if $a \leq 0$.

Theorem (exponential): For $x, a \in \mathbb{R}$,

$$|e^{x + a} - e^x| = e^x (e^a - 1) \leq e^x (e^{|a|} - 1),$$

with equality if $a \geq 0$.

Theorem (inverse tangent): For $x, a \in \mathbb{R}$,

$$|\arctan(x + a) - \arctan(x)| \leq \min(\pi, C_1 |a|).$$

where

$$C_1 = \sup_{|t| \leq |a|} \frac{1}{1 + (x + t)^2}.$$ 

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

$$|\arctan(x + a) - \arctan(x)| = \arctan\left(\frac{|a|}{1 + x(x + a)}\right).$$

In the last formula, a case distinction has to be made depending on the signs of $x$ and $a$.

Sums and series

Theorem (geometric bound): If $|c_k| \leq C$ and $|z| \leq D < 1$, then

$$\left|\sum_{k=N}^{\infty} c_k z^k\right| \leq \frac{CD^N}{1 - D}.$$

Theorem (integral bound): If $f(x)$ is nonnegative and monotone decreasing, then

$$\int_N^\infty f(x) \leq \sum_{k=N}^{\infty} f(k) \leq f(N) + \int_N^\infty f(x)dx.$$

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 $D = \{z : |z| \leq R\}$ 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):

$$|c_k| \leq \frac{C}{R^k}, \quad C = \max_{|z|=R} |f(z)|.$$

Corollary (Taylor series tail): If $0 \leq r < R$ and $|z| \leq r$, then

$$\left| \sum_{k=N}^{\infty} c_k z^k \right| \leq \frac{CD^N}{1 - D}, \quad D = \left| \frac{r}{R} \right|.$$

**Euler-Maclaurin formula**

**Theorem (Euler-Maclaurin):** If $f(t)$ is $2M$-times differentiable, then

$$\sum_{k=L}^{U} f(k) = S + I + T + R$$

$$S = \sum_{k=L}^{N-1} f(k), \quad I = \int_{N}^{U} f(t)dt,$$

$$T = \frac{1}{2} (f(N) + f(U)) + \sum_{k=1}^{M} \frac{B_{2k}}{(2k)!} \left( f^{(2k-1)}(U) - f^{(2k-1)}(N) \right),$$

$$R = -\int_{N}^{U} \frac{B_{2M}(t - \lfloor t \rfloor)}{(2M)!} f^{(2M)}(t)dt.$$

**Lemma (Bernoulli polynomials):** $|B_n(t - \lfloor t \rfloor)| \leq 4n!/(2\pi)^n$.

**Theorem (remainder bound):**

$$|R| \leq \frac{4}{(2\pi)^{2M}} \int_{N}^{U} \left| f^{(2M)}(t) \right| dt.$$

**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

$$|c_k| \leq \frac{4}{(2\pi)^{2M}} \int_{N}^{U} |a_k^{(2M)}(t)| dt.$$

**Algorithms for mathematical constants**

Most mathematical constants are evaluated using the generic hypergeometric summation code.

**Pi**

$\pi$ is computed using the Chudnovsky series

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k(6k)!((13591409 + 545140134k))}{(3k)!k!^3640320^{3k+3/2}},$$

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.
Logarithms of integers

We use the formulas

\[
\log(2) = \frac{3}{4} \sum_{k=0}^{\infty} \frac{(-1)^k (k!)^2}{2^k (2k+1)!}
\]

\[
\log(10) = 46 \arctanh(\frac{1}{31}) + 34 \arctanh(\frac{1}{49}) + 20 \arctanh(\frac{1}{161})
\]

Euler’s constant

Euler’s constant \( \gamma \) 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

\[
S_0(x) = \sum_{k=0}^{\infty} \frac{H_k}{(k!)^2} \left( \frac{x}{2} \right)^{2k}, \quad I_0(x) = \sum_{k=0}^{\infty} \frac{1}{(k!)^2} \left( \frac{x}{2} \right)^{2k}
\]

\[
2xI_0(x)K_0(x) \sim \sum_{k=0}^{\infty} \frac{[(2k)!]^3}{(k!)^4 2^{2k} x^{2k}}.
\]

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 \( 24e^{-8n} \).

Catalan’s constant

Catalan’s constant is computed using the hypergeometric series

\[
C = \sum_{k=0}^{\infty} \frac{(-1)^k 4^{k+1} (40k^2 + 56k + 19)}{(k+1)^3 (2k+1) [(4k+4)!]^2}
\]

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 \( M \geq p/(2 \log_2 N) \).
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 $s = 2$.

Apery’s constant

Apery’s constant $\zeta(3)$ is computed using the hypergeometric series

$$
\zeta(3) = \frac{1}{64} \sum_{k=0}^{\infty} (-1)^k (205k^2 + 250k + 77) \frac{(k!)^{10}}{[(2k + 1)!]^5}.
$$

Algorithms for the gamma function

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 ([Ole1997] pp. 293-295) the remainder term is exactly

$$
R_n(z) = \int_0^\infty \frac{B_{2n} - \tilde{B}_{2n}(x)}{2n(x^2 + z^2)^{2n}} dx.
$$

To evaluate the gamma function of a power series argument, we substitute $z \rightarrow z + t \in \mathbb{C}[t]$.

Using the bound for $|x + z|$ given by [Ole1997] 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

$$
|t^k R_n(z + t)| \leq 2|B_{2n}| \frac{\Gamma(2n + k - 1)}{\Gamma(k + 1)\Gamma(2n + 1)} |z| \left( \frac{b}{|z|} \right)^{2n+k}
$$

where $b = 1/\cos(\text{arg}(z)/2)$. 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 $\Gamma(z) = \Gamma(z + r)/(z + 1)(z + 2)\cdots(z + r - 1)$.

To obtain a remainder smaller than $2^{-p}$, we must choose an $r$ such that, in the real case, $z + r > \beta p$, where $\beta \approx \log(2)/(2\pi) \approx 0.11$. In practice, a slightly larger factor $\beta \approx 0.2$ more closely balances $n$ and $r$. A much larger $\beta$ (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.

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:

$$
\Gamma(2/3) = \frac{2\pi}{3^{1/2}\Gamma(1/3)}, \quad \Gamma(3/4) = \frac{2^{1/2}\pi}{\Gamma(1/4)} ,
$$

$$
\Gamma(1/6) = \frac{(\pi/3)^2}{\Gamma(1/3)^2}, \quad \Gamma(5/6) = \frac{2\pi(\pi/3)^{1/2}}{\Gamma(1/3)^2}.
$$

We compute $\Gamma(1/3)$ and $\Gamma(1/4)$ rapidly to high precision using

$$
\Gamma(1/3) = \left( \frac{12\pi^4}{\sqrt{10}} \sum_{k=0}^{\infty} \frac{(6k)!(-1)^k}{(k!)^3(3k)!160^k} \right)^{1/6}, \quad \Gamma(1/4) = \sqrt{\frac{(2\pi)^{3/2}}{\text{agm}(1, \sqrt{2})}}.
$$

An alternative formula which could be used for $\Gamma(1/3)$ is

$$
\Gamma(1/3) = \frac{2^{4/9}\pi^{2/3}}{3^{1/12} \left( \text{agm} \left( 1, \frac{1}{2}\sqrt{2 + \sqrt{3}} \right) \right)^{1/3}},
$$

but this appears to be slightly slower in practice.

**Algorithms for the Hurwitz zeta function**

**Euler-Maclaurin summation**

The Euler-Maclaurin formula allows evaluating the Hurwitz zeta function and its derivatives for general complex input. The algorithm is described in [Joh2013].

**Parameter Taylor series**

To evaluate $\zeta(s, a)$ for several nearby parameter values, the following Taylor expansion is useful:

$$
\zeta(s, a + x) = \sum_{k=0}^{\infty} (-x)^k \frac{(s)_k}{k!} \zeta(s + k, a)
$$

We assume that $a \geq 1$ is real and that $\sigma = \text{re}(s)$ with $K + \sigma > 1$. The tail is bounded by

$$
\sum_{k=K}^{\infty} |x|^k \frac{|(s)_k|}{k!} \zeta(s + k, a) \leq \sum_{k=K}^{\infty} |x|^k \frac{|(s)_k|}{k!} \left[ \frac{1}{a^{\sigma+k}} + \frac{1}{(\sigma+k-1)a^{\sigma+k-1}} \right].
$$

Denote the term on the right by $T(k)$. Then

$$
\left| \frac{T(k+1)}{T(k)} \right| = \frac{|x|}{a} \frac{(k + \sigma - 1)}{(k + \sigma)} \frac{(k + \sigma + a)}{(k + \sigma + a - 1)} \frac{|k + s|}{(k + 1)} \leq \frac{|x|}{a} \left( 1 + \frac{1}{K + \sigma + a - 1} \right) \left( 1 + \frac{|s - 1|}{K + 1} \right) = C
$$

and if $C < 1$,

$$
\sum_{k=K}^{\infty} T(k) \leq \frac{T(K)}{1 - C}.
$$

**Algorithms for polylogarithms**

The polylogarithm is defined for $s, z \in \mathbb{C}$ with $|z| < 1$ by

$$
\text{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}
$$

and for $|z| \geq 1$ by analytic continuation, except for the singular point $z = 1$. 

204 Chapter 11. Supplementary algorithm notes
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

$$\text{Li}_{s+x}(z) = \sum_{d=0}^{\infty} x^d \frac{(-1)^d}{d!} \sum_{k=1}^{\infty} T(k)$$

where

$$T(k) = \frac{z^k \log^d(k)}{k^s}.$$ 

The remainder term $|\sum_{k=N}^{\infty} T(k)|$ is bounded via $\text{mag\_polylog\_tail()}$.

Expansion for general $z$

For general complex $s, z$, we write the polylogarithm as a sum of two Hurwitz zeta functions

$$\text{Li}_s(z) = \frac{\Gamma(v)}{(2\pi i)^v} \left[ i^v \zeta\left(v, \frac{1}{2} + \frac{\log(-z)}{2\pi i}\right) + i^{-v} \zeta\left(v, \frac{1}{2} - \frac{\log(-z)}{2\pi i}\right) \right]$$

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 $\text{Li}_{s+x}(z)$). 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

$$E_1 \tilde{Z}_1 + E_2 \tilde{Z}_2 = (E_1 + E_2)/x + E_1 \tilde{Z}_1 + E_2 \tilde{Z}_2.$$ 

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.

Algorithms for hypergeometric functions

The algorithms used to compute hypergeometric functions are described in [Joh2016]. Here, we state the most important error bounds.
Convergent series

Let

$$T(k) = \prod_{i=0}^{n-1} (a_i)_k \cdot \prod_{i=0}^{m-1} (b_i)_k \cdot k.$$  

We compute a factor $C$ such that

$$\sum_{k=m}^{\infty} T(k) \leq C|T(n)|.$$  

We check that 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

$$\left| \frac{a + k}{b + k} \right| = 1 + \frac{a - b}{b + k} \leq 1 + \frac{|a - b|}{|b + n|}$$  

and

$$\left| \frac{1}{b + k} \right| \leq \frac{1}{|b + n|}$$

for all $k \geq n$. This gives us a constant $D$ such that $T(k + 1) \leq DT(k)$ for all $k \geq n$. If $D \geq 1$, we set $C$ to infinity. Otherwise, we take $C = \sum_{k=0}^{\infty} D^k = (1 - D)^{-1}$.

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 $\sum_{k=0}^{\infty} T(k) \in \mathbb{C}[[x]]/(x^n)$ given $a_i, b_i, z \in \mathbb{C}[[x]]/(x^n)$. 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 $R(B) = |B[0]| - |B[1:\infty]|$.

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:

$$|A + B| \leq |A| + |B|, \quad |AB| \leq |A||B|, \quad |A/B| \leq |A|/R(B).$$

If $p \leq q$ and Re$(b_i[0] + N) > 0$ for all $b_i$, then we may take

$$D = |z| \prod_{i=1}^{p} \left( 1 + \frac{|a_i - b_i|}{R(b_i + N)} \right) \prod_{i=p+1}^{q} \frac{1}{R(b_i + N)}.$$  

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 $R(B)$, we must then pick a lower bound for $|B[0]|$ and upper bounds for the coefficients of $|B[1:\infty]|$.
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

$$\frac{1F1(a, b, z)}{\Gamma(b)} = \frac{(-z)^{-a}}{\Gamma(b - a)} U^*(a, b, z) + \frac{z^{a-b} e^z}{\Gamma(a)} U^*(b - a, b, -z)$$

which is equivalent to DLMF 13.2.41 (but simpler in form).

We have the asymptotic expansion

$$U^*(a, b, z) \sim 2F0(a, a - b + 1, -1/z)$$

where $2F0(a, b, z)$ 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 $\text{Re}(z) \geq r$, set $R = 1$. Otherwise, if $\text{Im}(z) \geq r$ or $\text{Re}(z) \geq 0 \wedge |z| \geq r$, 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

$$\sigma = |(b - 2a)/z|$$

$$C_n = \begin{cases} 1 & \text{if } R = 1 \\ \chi(n) & \text{if } R = 2 \\ (\chi(n) + \sigma\nu^2 n)\nu^n & \text{if } R = 3 \end{cases}$$

$$\nu = \left(\frac{1}{2} + \frac{1}{2}\sqrt{1 - 4\sigma^2}\right)^{-1/2} \leq 1 + 2\sigma^2$$

$$\chi(n) = \sqrt{\pi} \Gamma\left(\frac{1}{2}n + 1\right)/\Gamma\left(\frac{1}{2}n + \frac{1}{2}\right)$$

$$\sigma' = \begin{cases} \sigma & \text{if } R \neq 3 \\ \nu\sigma & \text{if } R = 3 \end{cases}$$

$$\alpha = (1 - \sigma')^{-1}$$

$$\rho = \frac{1}{2} |2a^2 - 2ab + b| + \sigma'(1 + \frac{1}{4}\sigma')(1 - \sigma')^{-2}$$

11.6. Algorithms for hypergeometric functions 207
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

\[
\text{Ai}(z) = \frac{e^{-\zeta}}{2\sqrt{\pi}z^{1/4}} \left[ S_n(\zeta) + R_n(z) \right], \quad \text{Ai}'(z) = -\frac{z^{1/4}e^{-\zeta}}{2\sqrt{\pi}} \left[ S'_n(\zeta) + R'_n(z) \right]
\]

where

\[
S_n(\zeta) = \sum_{k=0}^{n-1} (-1)^k \frac{u(k)}{\zeta^k}, \quad S'_n(\zeta) = \sum_{k=0}^{n-1} (-1)^k \frac{v(k)}{\zeta^k}
\]

\[
u(k) = \frac{(1/6)_k(5/6)_k}{2^k k!}, \quad v(k) = \frac{6k + 1}{1 - 6k} u(k).
\]

Assuming that \( n \) is positive, the error terms are bounded by

\[
|R_n(z)| \leq C|u(n)||\zeta|^{-n}, \quad |R'_n(z)| \leq C|v(n)||\zeta|^{-n}
\]

where

\[
C = \begin{cases} 
2 \exp(7/(36|\zeta|)) & |\arg(z)| \leq \pi/3 \\
2 \chi(n) \exp(7\pi/(72|\zeta|)) & \pi/3 \leq |\arg(z)| \leq 2\pi/3 \\
4 \chi(n) \exp(7\pi/(36 \text{ re}(\zeta))) \cos(\arg(\zeta)) |\zeta|^{-n} & 2\pi/3 \leq |\arg(z)| < \pi.
\end{cases}
\]

For computing \( \text{Bi} \) when \( z \) is roughly in the positive half-plane, we use the connection formulas

\[
\text{Bi}(z) = -i(2w^{+1} \text{Ai}(zw^{-2}) - \text{Ai}(z)) \\
\text{Bi}(z) = +i(2w^{-1} \text{Ai}(zw^2) - \text{Ai}(z))
\]

where \( w = \exp(\pi i/3) \). Combining roots of unity gives

\[
\text{Bi}(z) = \frac{1}{2\sqrt{\pi}z^{1/4}}[2X + iY]
\]

\[
\text{Bi}(z) = \frac{1}{2\sqrt{\pi}z^{1/4}}[2X - iY]
\]

\[X = \exp(+\zeta)[S_n(-\zeta) + R_n(zw^{+2})], \quad Y = \exp(-\zeta)[S_n(\zeta) + R_n(z)]\]

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 \( \text{Bi} \).

In the negative half-plane, we use the connection formulas

\[
\text{Ai}(z) = e^{+\pi i/3} \text{Ai}(z_1) + e^{-\pi i/3} \text{Ai}(z_2)
\]

\[
\text{Bi}(z) = e^{-\pi i/6} \text{Ai}(z_1) + e^{+\pi i/6} \text{Ai}(z_2)
\]

where \( z_1 = -ze^{+\pi i/3}, z_2 = -ze^{-\pi i/3} \). Provided that \( |\arg(-z)| < 2\pi/3 \), we have \( |\arg(z_1)|, |\arg(z_2)| < \pi \), and thus the asymptotic expansion for \( \text{Ai} \) can be used. As before, we collect roots of unity to obtain

\[
\text{Ai}(z) = A_1[S_n(i\zeta) + R_n(z_1)] + A_2[S_n(-i\zeta) + R_n(z_2)]
\]

\[
\text{Bi}(z) = A_3[S_n(i\zeta) + R_n(z_1)] + A_4[S_n(-i\zeta) + R_n(z_2)]
\]

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.
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

\[
0 \to 0.375 \pm 0.625i \to 0.5 \pm 0.8125i \to z
\]
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

\[
g''(z) = \frac{N + 1}{2} \left( \frac{\nu}{1 - \nu z} \right) g'(z) + \left( \frac{N + 1}{2} \right)^2 \left( \frac{\nu}{1 - \nu z} \right) g(z)
\]

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 \( A, N, \nu \) such that the simple solution \( g(z) = A(1 - \nu z)^{-N} \) of the differential equation provides the bound

\[
|f[k]| \leq g[k] = A \binom{N + k}{k} \nu^k.
\]

Algorithms for the arithmetic-geometric mean

With complex variables, it is convenient to work with the univariate function \( M(z) = \text{agm}(1, z) \). The general case is given by \( \text{agm}(a, b) = aM(1, b/a) \).

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 \( M'(z) = [M(u) - (z - 1)M'(u)]/((1 + z)\sqrt{z})]/2 \).

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 \( \sqrt{ab} \), which is computed as \( \sqrt{ab}, i\sqrt{-ab}, -i\sqrt{-ab}, \sqrt{ab} \) respectively if both \( a \) and \( b \) have positive real part, nonnegative imaginary part, nonpositive imaginary part, or otherwise.

The iteration should be terminated when \( a_n \) and \( b_n \) are close enough. For positive real variables, we can simply take lower and upper bounds to get a correct enclosure at this point. For complex variables, it is shown in [Dup2006], p. 87 that, for \( z \) with nonnegative real part, \( |M(z) - a_n| \leq |a_n - b_n| \), giving a convenient error bound.
Rather than running the AGM iteration until \(a_n\) and \(b_n\) agree to \(p\) bits, it is slightly more efficient to iterate until they agree to about \(p/10\) bits and finish with a series expansion. With \(z = (a - b)/(a + b)\), we have

\[
\text{agm}(a, b) = \frac{(a + b)\pi}{4K(z^2)},
\]
valid at least when \(|z| < 1\) and \(a, b\) have nonnegative real part, and

\[
\frac{\pi}{4K(z^2)} = \frac{1}{2} - \frac{1}{8}z^2 - \frac{5}{128}z^4 - \frac{11}{512}z^6 - \frac{469}{32768}z^8 + \ldots
\]
where the tail is bounded by \(\sum_{k=10}^{\infty}|z|^k/64\).

**First derivative**

Assuming that \(z\) is exact and that \(|\arg(z)| \leq 3\pi/4\), we compute \((M(z), M'(z))\) simultaneously using a finite difference.

The basic inequality we need is \(|M(z)| \leq \max(1, |z|)|z|\), which is an immediate consequence of the AGM iteration.

By Cauchy’s integral formula, \(|M^{(k)}(z)/k!| \leq CD^k\) 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

\[
\begin{align*}
\left| \frac{M(z + h) - M(z)}{h} - M'(z) \right| & \leq \frac{CD^2 h}{1 - Dh} \\
\left| \frac{M(z + h) - M(z - h)}{2h} - M'(z) \right| & \leq \frac{CD^2 h^2}{1 - Dh} \\
\left| \frac{M(z + h) + M(z - h)}{2} - M(z) \right| & \leq \frac{CD^2 h^2}{1 - Dh}
\end{align*}
\]
assuming that \(h\) is chosen so that it satisfies \(hD < 1\).

The forward finite difference would require two function evaluations at doubled precision. We use the central difference as it only requires 1.5 times the precision.

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

\[
\begin{align*}
|M'(z + \varepsilon)| & \leq \frac{\max(1, |z| + |\varepsilon| + r)}{r} \\
|M''(z + \varepsilon)| & \leq \frac{2\max(1, |z| + |\varepsilon| + r)}{r^2}
\end{align*}
\]
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.

**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

\[-2z(z^2 - 1)c_2 = (3z^2 - 1)c_1 + zc_0,\]

\[-(k + 2)(k + 3)z(z^2 - 1)c_{k+3} = (k + 2)^2(3z^2 - 1)c_{k+2} + (3k(k + 3) + 7)zc_{k+1} + (k + 1)^2c_k\]
in general, and

\[-(k + 2)^2c_{k+2} = (3k(k + 3) + 7)c_{k+1} + (k + 1)^2c_k\]
when \(z = 1\).
HISTORY, CREDITS AND REFERENCES

Credits and references

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.

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).

The following authors have developed major new features.

- Pascal Molin - discrete Fourier transform (DFT), Dirichlet characters, Dirichlet L-functions, discrete logarithm computation
- Alex Griffing - sinc function, matrix trace, improved matrix squaring, boolean matrices, improved structured matrix exponentials, Cholesky decomposition, miscellaneous patches
- Marc Mezzarobba - fast evaluation of Legendre polynomials, work on Arb interface in Sage, bug reports, feedback

Several people have contributed patches, bug reports, or substantial feedback. This list (ordered by time of first contribution) 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 - original code for Dirichlet characters, 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
- 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
- Jeroen Demeyer - patch for major bug on PPC64
- Isuru Fernando - continuous integration setup, support for cmake and MSVC builds
- François Bissey - build system patches
- Jean-Pierre Flori - code simplifications for Gauss periods, feedback

**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.

**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
Citing Arb

To cite Arb in a scientific paper, the following reference can be used:


In BibTeX format:

```latex
@article{Johansson2017arb,  
  author = {F. Johansson},  
  title = {Arb: efficient arbitrary-precision midpoint-radius interval arithmetic},  
  journal = {IEEE Transactions on Computers},  
  year = {2017},  
  volume = {66},  
  issue = {8},  
  pages = {1281--1292},  
  doi = {10.1109/TC.2017.2690633},  
}
```

Alternatively, the Arb manual or website can be cited directly.

The *IEEE Transactions on Computers* paper supersedes the following extended abstract, which is now outdated:


Bibliography

(In the PDF edition, this section is empty. See the bibliography listing at the end of the document.)

History and changes

For more details, view the commit log in the git repository [https://github.com/fredrik-johansson/arb](https://github.com/fredrik-johansson/arb)

Old releases of the code can be accessed from [https://github.com/fredrik-johansson/arb/releases](https://github.com/fredrik-johansson/arb/releases)

Old versions of the documentation

- [http://arblib.org/arb-2.11.1.pdf](http://arblib.org/arb-2.11.1.pdf)
- [http://arblib.org/arb-2.11.0.pdf](http://arblib.org/arb-2.11.0.pdf)
- [http://arblib.org/arb-2.10.0.pdf](http://arblib.org/arb-2.10.0.pdf)
- [http://arblib.org/arb-2.9.0.pdf](http://arblib.org/arb-2.9.0.pdf)
- [http://arblib.org/arb-2.8.0.pdf](http://arblib.org/arb-2.8.0.pdf)
- [http://arblib.org/arb-2.7.0.pdf](http://arblib.org/arb-2.7.0.pdf)
- [http://arblib.org/arb-2.6.0.pdf](http://arblib.org/arb-2.6.0.pdf)
2018-02-23 – version 2.13.0

• Major bugs
  - Fixed rounding direction in arb_get_abs_lbound_arf() which in some cases could result in an invalid lower bound being returned, and added forgotten test code for this and related functions (reported by deinst). Although this bug could lead to incorrect results, it probably had limited impact in practice (explaining why it was not caught indirectly by other test code) since a single rounding in the wrong direction in this operation generally will be dwarfed by multiple roundings in the correct direction in surrounding operations.

• Important notes about bounds
  - Many functions have been modified to compute tighter enclosures when the input balls are wide. In most cases the bounds should be improved, but there may be some regressions. Bug reports about any significant regressions are welcome.
  - Division by zero in arb_div() has been changed to return [NaN +/- inf] instead of [+/- inf]. This change might be reverted in the future if it proves to be too inconvenient. In either case, users should only assume that division by zero produces something non-finite, and user code that depends on division by zero to produce [0 +/- inf] should be modified to handle zero-containing denominators as a separate case.

• Improvements to arithmetic and elementary functions
  - Faster implementation of acb_get_mag_lower().
  - Optimized arb_get_mag_lower(), arb_get_mag_lower_nonnegative().
  - Added arb_set_interval_mag() and arb_set_interval_neg_pos_mag() for constructing an arb_t from a pair of mag_t endpoints.
  - Added mag_const_pi_lower(), mag_atan(), mag_atan_lower().
  - Added mag_div_lower(), mag_inv(), mag_inv_lower().
  - Added mag_sqrt_lower() and mag_rsqrt_lower().
  - Added mag_log(), mag_log_lower(), mag_neg_log(), mag_neg_log_lower().
  - Added mag_exp_lower(), mag_expinv_lower() and tweaked mag_exp().
  - Added mag_pow_fmpz_lower(), mag_get_fmpz(), mag_get_fmpz_lower().
  - Improved arb_exp() for wide input.
  - Improved arb_log() for wide input.
  - Improved arb_sqrt() for wide input.
  - Improved arb_rsqrt() for wide input.
  - Improved arb_div() for wide input.
  - Improved arb_atan() for wide input and slightly optimized arb_atan2() for input spanning multiple signs.
  - Improved acb_rsqrt() for wide input and improved stability of this function generally in the left half plane.
  - Added arb_log_hypot() and improved acb_log() for wide input.
  - Slightly optimized trigonometric functions (acb_sin(), acb_sin_pi(), acb_cos(), acb_cos_pi(), acb_sin_cos(), acb_sin_cos_pi()) for pure real or imaginary input.
• Special functions
  – Slightly improved bounds for gamma function (arb_gamma(), acb_gamma(), arb_rgamma(),
    acb_rgamma()) for wide input.
  – Improved bounds for Airy functions for wide input.
  – Simplifications to code for computing Gauss period minimal polynomials (contributed by
    Jean-Pierre Flori).
  – Optimized arb_hypgeom_legendre_p_ui() further by avoiding divisions in the basecase re-
    currence and computing the prefactor more quickly in the asymptotic series (contributed by
    Marc Mezzarobba).
  – Small further optimization of arb_hypgeom_legendre_p_ui_root() (contributed by Marc
    Mezzarobba).
  – Improved derivative bounds for Legendre polynomials (contributed by Marc Mezzarobba).

• Numerical integration
  – Increased default quadrature deg_limit at low precision to improve performance for integration
    of functions without singularities near the path.
  – Added several more integrals to examples/integrals.c
  – Added utility functions acb_real_abs(), acb_real_sgn(), acb_real_heaviside(),
    acb_real_floor(), acb_real_ceil(), acb_real_min(), acb_real_max(), acb_real_sqrtpos(),
    useful for numerical integration.
  – Added utility functions acb_sqrt_analytic(), acb_rsqrt_analytic(), acb_log_analytic(),
    acb_pow_analytic() with branch cut detection, useful for numerical integration.

• Build system and compatibility issues
  – Removed -Wl flag from Makefile.subdirs to fix “-r and -pie may not be used together” compi-
    lation error on some newer Linux distributions (reported by many users).
  – Fixed broken test code for l_vec_hurwitz which resulted in spurious failures on 32-bit systems
    (originally reported by Thierry Monteil on Sage trac).
  – Avoid using deprecated MPFR function mpfr_root() with MPFR versions >= 4.0.0.
  – Remark: the recently released MPFR 4.0.0 has a bug in mpfr_div() leading to test failures
    in Arb (though not affecting correctness of Arb itself). Users should make sure to install the
    patched version MPFR 4.0.1.
  – Added missing C++ include guards in arb_fmpz_poly.h and dlog.h (reported by Marc Mez-
    zarobba).
  – Fixed Travis builds on Mac OS again (contributed by Isuru Fernando).
  – Added missing declaration for arb_bell_ui() (reported by numsys).

2017-11-29 - version 2.12.0

• Numerical integration
  – Added a new function (acb_calc_integrate) for rigorous numerical integration using adaptive
    subdivision and Gauss-Legendre quadrature. This largely obsoletes the old integration code
    using Taylor series.
  – Added new integrals.c example program (old example program moved to integrals_taylor.c).

• Discrete Fourier transforms
  – Added acb_dft module with various FFT algorithm implementations, including top level O(n
    log n) acb_dft and acb_dft_inverse functions (contributed by Pascal Molin).
• Legendre polynomials
  – Added arb_hypgeom_legendre_p_ui for fast and accurate evaluation of Legendre polynomials. This is also used automatically by the Legendre functions, where it is substantially faster and gives better error bounds than the generic algorithm.
  – Added arb_hypgeom_legendre_p_ui_root for fast computation of Legendre polynomial roots and Gauss-Legendre quadrature nodes (used internally by the new integration code).
  – Added arb_hypgeom_central_bin_ui for fast computation of central binomial coefficients (used internally for Legendre polynomials).

• Dirichlet L-functions and zeta functions
  – Fixed a bug in the Riemann zeta function involving a too small error bound in the implementation of the Riemann-Siegel formula for inexact input. This bug could result in a too small enclosure when evaluating the Riemann zeta function at an argument of large imaginary height without also computing derivatives, if the input interval was very wide.
  – Add acb_dirichlet_zeta_jet; also made computation of the first derivative of Riemann zeta function use the Riemann-Siegel formula where appropriate.
  – Added acb_dirichlet_l_vec_hurwitz for fast simultaneous evaluation of Dirichlet L-functions for multiple characters using Hurwitz zeta function and FFT (contributed by Pascal Molin).
  – Simplified interface for using hurwitz_precomp functions.
  – Added lcentral.c example program (contributed by Pascal Molin).
  – Improved error bounds when evaluating Dirichlet L-functions using Euler product.

• Elementary functions
  – Faster custom implementation of sin, cos at 4600 bits and above instead of using MPFR (30-40% asymptotic improvement, up to a factor two speedup).
  – Faster code for exp between 4600 and 19000 bits.
  – Improved error bounds for acb_atan by using derivative.
  – Improved error bounds for arb_sinh_cosh, arb_sinh and arb_cosh when the input has a small midpoint and large radius.
  – Added reciprocal trigonometric and hyperbolic functions (arb_sec, arb_csc, arb_sech, arb_csch, acb_sec, acb_csc, acb_sech, acb_csch).
  – Changed the interface of _acb_vec_unit_roots to take an extra length parameter (compatibility-breaking change).
  – Improved arb_pow and acb_pow with an inexact base and a negative integer or negative half-integer exponent; the inverse is now computed before performing binary exponentiation in this case to avoid spurious blow-up.

• Elliptic functions
  – Improved Jacobi theta functions to reduce the argument modulo the lattice parameter, greatly improving speed and numerical stability for large input.
  – Optimized arb_agm by using a final series expansion and using special code for wide intervals.
  – Optimized acb_agm1 by using a final series expansion and using special code for positive real input.
  – Optimized derivative of AGM for high precision by using a central difference instead of a forward difference.
  – Optimized acb_elliptic_rf and acb_elliptic_rj for high precision by using a variable length series expansion.

• Other
- Fixed incorrect handling of subnormals in arf_set_d.
- Added mag_bin_uiui for bounding binomial coefficients.
- Added mag_set_d_lower, mag_sqrt_lower, mag_set_d_2exp_fmpz_lower.
- Implemented multithreaded complex matrix multiplication.
- Optimized arb_rel_accuracy_bits by adding fast path.
- Fixed a spurious floating-point exception (division by zero) in the t-gauss_period_minpoly test program triggered by new code optimizations in recent versions of GCC that are unsafe together with FLINT inline assembly functions (a workaround was added to the test code, and a proper fix for the assembly code has been added to FLINT).

2017-07-10 - version 2.11.1

- Avoid use of a function that was unavailable in the latest public FLINT release

2017-07-09 - version 2.11.0

- Special functions
  - Added the Lambert W function (arb_lambertw, acb_lambertw, arb_poly_lambertw_series, acb_poly_lambertw_series). All complex branches and evaluation of derivatives are supported.
  - Added the acb_expm1 method, complementing arb_expm1.
  - Added arb_sinc_pi, acb_sinc_pi.
  - Optimized handling of more special cases in the Hurwitz zeta function.

- Polynomials
  - Added the arb_fmpz_poly module to provide Arb methods for FLINT integer polynomials.
  - Added methods for evaluating an fmpz_poly at arb_t and acb_t arguments.
  - Added arb_fmpz_poly_complex_roots for computing the real and complex roots of an integer polynomial, turning the functionality previously available in the poly_roots.c example program into a proper library function.
  - Added a method (arb_fmpz_poly_gauss_period_minpoly) for constructing minimal polynomials of Gaussian periods.
  - Added arb_poly_product_roots_complex for constructing a real polynomial from complex conjugate roots.

- Miscellaneous
  - Fixed test code in the dirichlet module for 32-bit systems (contributed by Pascal Molin).
  - Use flint_abort() instead of abort() (contributed by Tommy Hofmann).
  - Fixed the static library install path (contributed by François Bissey).
  - Made arb_nonnegative_part() a publicly documented method.
  - Arb now requires FLINT version 2.5 or later.

2017-02-27 - version 2.10.0

- General
  - Changed a large number of methods from inline functions to normal functions, substantially reducing the size of the built library.
- Fixed a few minor memory leaks (missing clear() calls).

- **Basic arithmetic**
  - Added `arb_is_int_2exp_si` and `acb_is_int_2exp_si`.
  - Added `arf_sosq` for computing $x^2 + y^2$ of floating-point numbers.
  - Improved error bounds for complex square roots in the left half plane.
  - Improved error bounds for complex reciprocal (`acb_inv`) and division.
  - Added the internal helper `mag_get_d_log2_approx` as a public method.

- **Elliptic functions and integrals**
  - New module `acb_elliptic.h` for elliptic functions and integrals.
  - Added complete elliptic integral of the third kind.
  - Added Legendre incomplete elliptic integrals (first, second, third kinds).
  - Added Carlson symmetric incomplete elliptic integrals (RF, RC, RG, RJ, RD).
  - Added Weierstrass elliptic zeta and sigma functions.
  - Added inverse Weierstrass elliptic p-function.
  - Added utility functions for computing the Weierstrass invariants and lattice roots.
  - Improved computation of derivatives of Jacobi theta functions by using modular transformations, and added a main evaluation function (`acb_modular_theta_jet`).
  - Improved detection of pure real or pure imaginary parts in various cases of evaluating theta and modular functions.

- **Other special functions**
  - New, far more efficient implementation of the dilogarithm function (`acb_polylog` with $s = 2$).
  - Fixed an issue in the Hurwitz zeta function leading to unreasonable slowdown for certain complex input.
  - Added `acb_poly_exp_pi_i_series`.
  - Added `arb_poly_log1p_series`, `acb_poly_log1p_series`.

---

2016-12-02 - version 2.9.0

- **License**
  - Changed license from GPL to LGPL.

- **Build system and compatibility**
  - Fixed FLINT includes to use `flint/foo.h` instead of `foo.h`, simplifying compilation on many systems.
  - Added another alias for the dynamic library to fix `make check` on certain systems (contributed by Andreas Enge).
  - Travis CI support (contributed by Isuru Fernando).
  - Added support for `ARB_TEST_MULTIPLIER` environment variable to control the number of test iterations.
  - Support building with CMake (contributed by Isuru Fernando).
  - Support building with MSVC on Windows (contributed by Isuru Fernando).
  - Fixed unsafe use of `FLINT_ABS` for `slong -> ulong` conversion in `arf.h`, which caused failures on MIPS and ARM systems.
• Basic arithmetic and methods
  – Fixed mag_addmul(x,x,x) with x having a mantissa of all ones. This could produce a non-normalized mag_t value, potentially leading to incorrect results in arb and acb level arithmetic. This bug was caught by new test code, and fortunately would have been hard to trigger accidentally.
  – Added fasth paths for error bound calculations in arb_sqrt and arb_div, speeding up these operations significantly at low precision
  – Added support for round-to-nearest in all arf methods.
  – Added fprintf methods (contributed by Alex Griffing).
  – Added acb_printn and acb_fprintn methods to match arb_printn.
  – Added arb_equal_si and acb_equal_si.
  – Added arb_can_round_mpfr.
  – Added arb_get_ubound_arf, arb_get_lbound_arf (contributed by Tommy Hofmann).
  – Added sign function (arb_sgn).
  – Added complex sign functions (acb_sgn, acb_csgn).
  – Rewrote arb_contains_fmpq to make the test exact.
  – Optimized mag_get_fmpq.
  – Optimized arf_get_fmpz and added more robust test code.
  – Rewrote arb_get_unique_fmpz and arb_get_interval_fmpz_2exp, reducing overhead, making them more robust with huge exponents, and documenting their behavior more carefully.
  – Optimized arb_union.
  – Optimized arf_is_int, arf_is_int_2exp_si and changed these from inline to normal functions.
  – Added mag_const_pi, mag_sub, mag_expinv.
  – Optimized binary-to-decimal conversion for huge exponents by using exponential function instead of binary powering.
  – Added arb_intersection (contributed by Alex Griffing).
  – Added arb_min, arb_max (contributed by Alex Griffing).
  – Fixed a bug in arb_log and in test code on 64-bit Windows due to unsafe use of MPFR which only uses 32-bit exponents on Win64.
  – Improved some test functions to reduce the chance of reporting spurious failures.
  – Added squaring functions (arb_sqr, acb_sqr) (contributed by Ricky Farr).
  – Added arf_frexp.
  – Added arf_cmp_si, arf_cmp_ui, arf_cmp_d.
  – Added methods to count allocated bytes (arb_allocated_bytes, _arb_vec_allocated_bytes, etc.).
  – Added methods to predict memory usage for large vectors (_arb/_acb_vec_estimate_allocated_bytes).
  – Changed clear() methods from inline to normal functions, giving 8% faster compilation and 25% smaller libarb.so.
  – Added acb_unit_root and _acb_vec_unit_roots (contributed by Pascal Molin).

• Polynomials
  – Added sinh and cosh functions of power series (arb/acb_poly_sinh/cosh_series and sinh_cosh_series).
Use basecase series inversion algorithm to improve speed and error bounds in \texttt{arb/acb\_poly\_inv\_series}.

- Added functions for fast polynomial Taylor shift (\texttt{arb\_poly\_taylor\_shift}, \texttt{acb\_poly\_taylor\_shift} and variants).
- Fast handling of special cases in polynomial composition.
- Added \texttt{acb\_poly \_scalar mul} and \texttt{div} convenience methods (contributed by Alex Griffing).
- Added \texttt{set\_trunc}, \texttt{set\_trunc\_round} convenience methods.
- Added \texttt{add\_series}, \texttt{sub\_series} methods for truncating addition.
- Added polynomial \texttt{is\_zero}, \texttt{is\_one}, \texttt{is\_x}, \texttt{valuation} convenience methods.
- Added hack to \texttt{arb\_poly\_mullow} and \texttt{acb\_poly\_mullow} to avoid overhead when doing an in-place multiplication with length at most 2.
- Added binomial and Borel transform methods for \texttt{acb\_poly}.

- Matrices
  - Added Cholesky decomposition plus solving and inverse for positive definite matrices (\texttt{arb\_mat\_cho}, \texttt{arb\_mat\_spd\_solve}, \texttt{arb\_mat\_spd\_inv} and related methods) (contributed by Alex Griffing).
  - Added LDL decomposition and inverse and solving based on LDL decomposition for real matrices (\texttt{arb\_mat\_ldl}, \texttt{arb\_mat\_solve\_ldl\_precomp}, \texttt{arb\_mat\_inv\_ldl\_precomp}) (contributed by Alex Griffing).
  - Improved the entrywise error bounds in matrix exponential computation to preserve sparsity and give exact entries where possible in many cases (contributed by Alex Griffing).
  - Added public functions for computing the truncated matrix exponential Taylor series (\texttt{arb\_mat\_exp\_taylor\_sum}, \texttt{acb\_mat\_exp\_taylor\_sum}).
  - Added functions related to sparsity structure (\texttt{arb\_mat\_entrywise\_is\_zero}, \texttt{arb\_mat\_count\_is\_zero}, etc.) (contributed by Alex Griffing).
  - Entrywise multiplication (\texttt{arb\_mat\_mul\_entrywise}, \texttt{acb\_mat\_mul\_entrywise}) (contributed by Alex Griffing).
  - Added \texttt{is\_empty} and \texttt{is\_square} convenience methods (contributed by Alex Griffing).
  - Added the \texttt{bool\_mat} helper module for matrices over the boolean semiring (contributed by Alex Griffing).
  - Added Frobenius norm computation (contributed by Alex Griffing).

- Miscellaneous special functions
  - Added evaluation of Bernoulli polynomials (\texttt{arb\_beroulli\_poly\_ui}, \texttt{acb\_beroulli\_poly\_ui}).
  - Added convenience function for evaluation of huge Bernoulli numbers (\texttt{arb\_beroulli\_fmpz}).
  - Added Euler numbers (\texttt{arb\_euler\_number\_ui}, \texttt{arb\_euler\_number\_fmpz}).
  - Added fast approximate partition function (\texttt{arb\_partitions\_fmpz/ui}).
  - Optimized partition function for $n < 1000$ by using recurrence for the low 64 bits.
  - Improved the worst-case error bound in \texttt{arb\_atan}.
  - Added \texttt{arb\_log\_base\_ui}.
  - Added complex sinc function (\texttt{acb\_sinc}).
  - Special handling of $z = 1$ when computing polylogarithms.
  - Fixed \texttt{agm(-1,-1)} to output 0 instead of indeterminate.
- Made working precision in arb_gamma and acb_gamma more sensitive to the input accuracy.

**Hypergeometric functions**
- Compute erf and erfc without cancellation problems for large or complex z.
- Avoid re-computing the square root of pi in several places.
- Added generalized hypergeometric function (acb_hypgeom_pfq).
- Implement binary splitting and rectangular splitting for evaluation of hypergeometric series
  with a power series parameter, greatly speeding up Y_n, K_n and other functions at high
  precision, as well as speeding up high-order parameter derivatives.
- Use binary splitting more aggressively in acb_hypgeom_pfq_sum to reduce error bound
  inflation.
- Asymptotic expansions of hypergeometric functions: more accurate parameter selection, and
  better handling of terminating cases.
- Tweaked algorithm selection and working precision in acb_hypgeom_m.
- Avoid dividing by the denominator of the next term in acb_hypgeom_sum, which would lead
  to a division by zero when evaluating hypergeometric polynomials.
- Fixed a bug in hypergeometric series evaluation resulting in near-integers not being skipped
  in some cases, leading to unnecessary loss of precision.
- Added series expansions of Airy functions (acb_hypgeom_airy_series,
  acb_hypgeom_airy_jet).
- Fixed a case where Airy functions accidentally chose the worst algorithm instead of the best
  one.
- Added functions for computing erf, erfc, erfi of power series in the acb_hypgeom module.
- Added series expansion of the logarithmic integral (acb_hypgeom_li_series).
- Added Fresnel integrals (acb_hypgeom_fresnel, acb_hypgeom_fresnel_series).
- Added the lower incomplete gamma function (acb_hypgeom_gamma_lower) (contributed by
  Alex Griffing).
- Added series expansion of the lower incomplete gamma function
  (acb_hypgeom_gamma_lower_series) (contributed by Alex Griffing).
- Added support for computing the regularized incomplete gamma functions.
- Use slightly sharper error bound for analytic continuation of 2F1.
- Added support for computing finite limits of 2F1 with inexact parameters differing by integers.
- Added the incomplete beta function (acb_hypgeom_beta_lower,
  acb_hypgeom_beta_lower_series)
- Improved acb_hypgeom_u to use a division-avoiding algorithm for small polynomial cases.
- Added arb_hypgeom module, wrapping the complex hypergeometric functions for more
  convenient use with the arb_t type.

**Dirichlet L-functions and Riemann zeta function**
- New module dirichlet for working algebraically with Dirichlet groups and characters (con-tributed by Pascal Molin).
- New module acb_dirichlet for numerical evaluation of Dirichlet characters and L-functions
  (contributed by Pascal Molin).
- Efficient representation and manipulation of Dirichlet characters using the Conrey representa-
  tion (contributed by Pascal Molin).
– New module dlog for word-size discrete logarithm evaluation, used to support algorithms on Dirichlet characters (contributed by Pascal Molin).
– Methods for properties, evaluation, iteration, pairing, lift, lowering etc. of Dirichlet characters (contributed by Pascal Molin).
– Added acb_dirichlet_roots methods for fast evaluation of many roots of unity (contributed by Pascal Molin).
– Added acb_dirichlet_hurwitz_precomp methods for fast multi-evaluation of the Hurwitz zeta function for many parameter values.
– Added methods for computing Gauss, Jacobi and theta sums over Dirichlet characters (contributed by Pascal Molin).
– Added methods (acb_dirichlet_l, acb_dirichlet_l_jet, acb_dirichlet_l_series) for evaluation of Dirichlet L-functions and their derivatives.
– Implemented multiple algorithms for evaluation of Dirichlet L-functions depending on the argument (Hurwitz zeta function decomposition, Euler product, functional equation).
– Added methods (acb_dirichlet_hardy_z, acb_dirichlet_hardy_z_series, etc.) for computing the Hardy Z-function corresponding to a Dirichlet L-function.
– Added fast bound for Hurwitz zeta function (mag_hurwitz_zeta_uiui).
– Improved parameter selection in Hurwitz zeta function to target relative instead of absolute error for large positive s.
– Improved parameter selection in Hurwitz zeta function to avoid computing unnecessary Bernoulli numbers for large imaginary s.
– Added Dirichlet eta function (acb_dirichlet_eta).
– Implemented the Riemann-Siegel formula for faster evaluation of the Riemann zeta function at large height.
– Added smooth-index algorithm for the main sum when evaluating the Riemann zeta function, avoiding the high memory usage of the full sieving algorithm when the number of terms gets huge.
– Improved tuning for using the Euler product when computing the Riemann zeta function.

• Example programs
  – Added logistic map example program.
  – Added lvalue example program.
  – Improved poly_roots in several ways: identify roots that are exactly real, automatically perform squarefree factorization, use power hack, and allow specifying a product of polynomials as input on the command line.

• Housekeeping
  – New section in the documentation giving an introduction to ball arithmetic and using the library.
  – Tidied, documented and added test code for the fmpz_extras module.
  – Added proper documentation and test code for many helper methods.
  – Removed the obsolete fmprb module entirely.
  – Documented more algorithms and formulas.
  – Clarified integer overflow issues and use of ARF_PREC_EXACT in the documentation.
  – Added .gitignore file.
  – Miscellaneous improvements to the documentation.
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.

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 acb_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.
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.

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.

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 MPRI.
  • 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_sqrtp1pm1, 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.

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.

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.

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.

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.

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).

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.
• 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 fmpcb_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 fmpcb_const_euler using published error bound.
  – Added fmpc_inv.
  – Added fmpc_trim, fmpcTrim.
  – Added fmpc_rsqrt (complex reciprocal square root).
  – Fixed bug in fmpc_sqrtpos with nonfinite input.
  – Slightly improved fmpc powering code.
  – Added various functions for bounding fmprs by powers of two.
  – Added fmpc_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 fmpc_poly_pow_series, fmpc_poly_pow_ui and related methods.
  – Added fmpc/fmpcb_poly_contains_fmpz_poly.
  – Faster composition by monomials.
  – Implemented Borel transform and binomial transform for real power series.
• Matrices
  – Implemented matrix exponentials.
  – Multithreaded fmpc_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.

### 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 fmprb_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.

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.

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 \texttt{fmprb\_atan2}, also fixing a bug in \texttt{fmpcb\_arg}.

Added \texttt{fmprb\_sin\_pi}, \texttt{cos\_pi}, \texttt{sin\_cos\_pi}, etc.

Added \texttt{fmprb\_sin\_pi\_fmpq} (etc.) using algebraic methods for fast evaluation of roots of unity.

Faster \texttt{fmprb\_poly\_evaluate} and \texttt{evaluate\_fmpcb} using rectangular splitting.

Added \texttt{fmprb\_poly\_evaluate2}, \texttt{evaluate2\_fmpcb} for simultaneously evaluating the derivative.

Added \texttt{fmpr\_root}, \texttt{fmprb\_root} (currently based on MPFR).

Added \texttt{fmpr\_min}, \texttt{fmpr\_max}.

Added \texttt{fmprb\_set\_interval\_fmp}, \texttt{fmprb\_union}.

Added \texttt{fmpr\_bits}, \texttt{fmprb\_bits}, \texttt{fmpcb\_bits} for obtaining the mantissa width.

Added \texttt{fmprb\_hypot}.

Added complex square roots.

Improved \texttt{fmprb\_log} to slightly improve speed, and properly support huge arguments.

Fixed \texttt{exp}, \texttt{cosh}, \texttt{sinh} to work with huge arguments.

Added \texttt{fmprb\_expm1}.

Fixed \texttt{sin}, \texttt{cos}, \texttt{atan} to work with huge arguments.

Improved \texttt{fmprb\_pow} and \texttt{fmpcb\_pow}, including automatic detection of small integer and half-integer exponents.

Added many more elementary functions: \texttt{fmprb\_tan/cot/tanh/coth}, \texttt{fmpcb\_tan/cot}, and pi versions.

Added \texttt{fmprb const\_e}, \texttt{const\_log2}, \texttt{const\_log10}, \texttt{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 \texttt{fmpr} 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 \texttt{fmprb\_agm} (real arithmetic-geometric mean).
  - Added \texttt{fmprb\_gamma\_fmpq}, supporting rapid computation of \( \gamma(\frac{p}{q}) \) 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.

**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.

**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.

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.

2012-09-14 - version 0.1

- 2012-08-05 - Began simplified rewrite.
- 2012-04-05 - Experimental ball and polynomial code (first commit).
[WQ3b] http://functions.wolfram.com/07.12.27.0014.01


[JM2018] F. Johansson and M. Mezzarobba, “Fast and rigorous arbitrary-precision computation of

Dirichlet L-series”, Problems of Information Transmission 34:4 (1998), 342-353,


of the International Conference on Signal Processing Algorithms, Architectures, Arrangements, and
Applications (SPA-2013), pp. 70-72, 2013.


[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,


precision”, Journal of Computational and Applied Mathematics 112 (1999), 253-267

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,


(2008), 67-87, https://doi.org/10.1137/060659831
Symbols

acb_dirichlet_euler_product_real_ui (C function), 168
acb_dirichlet_hardy_theta_series (C function), 169
acb_dirichlet_hardy_z_series (C function), 169
acb_elliptic_k_series (C function), 148
acb_elliptic_p_series (C function), 151
acb_hypgeom_airy_series (C function), 135
acb_hypgeom_beta_lower_series (C function), 137
acb_hypgeom_chi_series (C function), 138
acb_hypgeom_ci_series (C function), 138
acb_hypgeom_erf_series (C function), 133
acb_hypgeom_erfc_series (C function), 133
acb_hypgeom_fresnel_series (C function), 133
acb_hypgeom_gamma_lower_series (C function), 136
acb_hypgeom_gamma_upper_series (C function), 136
acb_hypgeom_li_series (C function), 138
acb_hypgeom_shi_series (C function), 138
acb_mat_charpoly (C function), 126
acb_poly_add (C function), 97
acb_poly_agm1_series (C function), 109
acb_poly_atan_series (C function), 104
acb_poly_binomial_transform (C function), 103
acb_poly_binomial_transform_basecase (C function), 103
acb_poly_binomial_transform_convolution (C function), 103
acb_poly_borel_transform (C function), 102
acb_poly_compose (C function), 100
acb_poly_compose_divconquer (C function), 99
acb_poly_compose_horner (C function), 99
acb_poly_compose_series (C function), 100
acb_poly_compose_series_brent_kung (C function), 100
acb_poly_cos_pi_series (C function), 105
acb_poly_cos_series (C function), 105
acb_poly_cosh_series (C function), 106
acb_poly_cot_pi_series (C function), 105
acb_poly_derivative (C function), 102
acb_poly_digamma_series (C function), 106
acb_poly_div (C function), 99
acb_poly_div_root (C function), 99
acb_poly_div_series (C function), 99
acb_poly_divrem (C function), 99
acb_poly_elliptic_k_series (C function), 109
acb_poly_elliptic_p_series (C function), 109
acb_poly_ erf_series (C function), 109
acb_poly_evaluate (C function), 101
acb_poly_evaluate2 (C function), 101
acb_poly_evaluate2_horner (C function), 101
acb_poly_evaluate2_rectangular (C function), 101
acb_poly_evaluate_horner (C function), 100
acb_poly_evaluate_rectangular (C function), 101
acb_poly_evaluate_vec_fast (C function), 101
acb_poly_evaluate_vec_fast_precomp (C function), 101
acb_poly_find_roots (C function), 110
acb_poly_gamma_series (C function), 106
acb_poly_integral (C function), 102
acb_poly_interpolate_barycentric (C function), 102
acb_poly_interpolate_fast (C function), 102
acb_poly_interpolate_fast_precomp (C function), 102
acb_poly_interpolate_newton (C function), 102
acb_poly_interpolation_weights (C function), 102
acb_poly_inv_borel_transform (C function), 102
acb_poly_interpolate_barycentric (C function), 102
acb_poly_inv_series (C function), 98
acb_poly_lambertw_series (C function), 106
acb_poly_loggamma_series (C function), 106
acb_poly_loglp_series (C function), 104
acb_poly_log_series (C function), 104
acb_poly_majorant (C function), 97
acb_poly_mul (C function), 98
acb_poly_shift_series (C function), 106
acb_poly_smooth (C function), 106
acb_poly_sinc_series (C function), 106
acb_poly_zeta_series (C function), 106
acb_poly_bernoulli_series (C function), 106
_acb_poly_mullow (C function), 98
_acb_poly_mullow_classical (C function), 98
_acb_poly_mullow_transpose (C function), 98
_acb_poly_mullow_transpose_gauss (C function), 98
_acb_poly_normalise (C function), 95
_acb_poly_overlaps (C function), 96
_acb_poly_polylog_cpx (C function), 108
_acb_poly_polylog_cpx_zeta (C function), 108
_acb_poly_polylog_series (C function), 108
_acb_poly_pow_acb_series (C function), 103
_acb_poly_pow_series (C function), 103
_acb_poly_pow_ui (C function), 103
_acb_poly_pow_ui_trunc_binexp (C function), 103
_acb_poly_powsum_one_series_sieved (C function), 107
_acb_poly_powsum_series_naive (C function), 107
_acb_poly_powsum_series_naive_threaded (C function), 107
_acb_poly_product_roots (C function), 101
_acb_poly_reline_roots_durand_kerner (C function), 109
_acb_poly_rem (C function), 99
_acb_poly_revert_series (C function), 100
_acb_poly_revert_series_lagrange (C function), 100
_acb_poly_revert_series_lagrange_fast (C function), 100
_acb_poly_revert_series_newton (C function), 100
_acb_poly_rgamma_series (C function), 106
_acb_poly_rising_ui_series (C function), 107
_acb_poly_root_bound_fujiwara (C function), 109
_acb_poly_root_inclusion (C function), 109
_acb_poly_root_inclusion (C function), 109
_acb_poly_rsqrt_series (C function), 103
_acb_poly_sub (C function), 97
_acb_poly_sub (C function), 97
_acb_poly_sub (C function), 97
_acb_poly_sub (C function), 97
_acb_poly_sub (C function), 97
_acb_poly_sub (C function), 97
_acb_poly_sub (C function), 97
_acb_poly_taylvec_series (C function), 99
_acb_poly_taylor_shift (C function), 99
_acb_poly_taylor_shift_convolution (C function), 99
_acb_poly_taylor_shift_divconquer (C function), 99
_acb_poly_taylor_shift_horner (C function), 99
_acb_poly_tree_alloc (C function), 101
_acb_poly_tree_build (C function), 101
_acb_poly_tree_free (C function), 101
_acb_poly_validate_real_roots (C function), 110
_acb_poly_validate_roots (C function), 109
_acb_poly_zeta_cpx_series (C function), 108
_acb_poly_zeta_em_bound (C function), 107
_acb_poly_zeta_em_bound1 (C function), 107
_acb_poly_zeta_em_choose_param (C function), 107
_acb_poly_zeta_em_tail (C function), 108
_acb_poly_zeta_em_tail_naive (C function), 108
_acb_poly_zeta_series (C function), 108
_acb_vec_add (C function), 78
_acb_vec_add_error_arf_vec (C function), 78
_acb_vec_add_error_mag_vec (C function), 78
_acb_vec_allocated_bytes (C function), 78
_acb_vec_bits (C function), 78
_acb_vec_clear (C function), 78
_acb_vec_estimate_allocated_bytes (C function), 78
_acb_vec_get_unique_fmpz_vec (C function), 78
_acb_vec_indeterminate (C function), 78
_acb_vec_init (C function), 78
_acb_vec_is_real (C function), 77
_acb_vec_is_zero (C function), 77
_acb_vec_neg (C function), 78
_acb_vec_scalar_addmul (C function), 78
_acb_vec_scalar_div (C function), 78
_acb_vec_scalar_div_arb (C function), 78
_acb_vec_scalar_div_fmpz (C function), 78
_acb_vec_scalar_div_ui (C function), 78
_acb_vec_scalar_mul (C function), 78
_acb_vec_scalar_mul_2exp_si (C function), 78
_acb_vec_scalar_mul_arb (C function), 78
_acb_vec_scalar_mul_fmpz (C function), 78
_acb_vec_scalar_mul_onei (C function), 78
_acb_vec_scalar_mul_ui (C function), 78
_acb_vec_scalar_submul (C function), 78
_acb_vec_set (C function), 78
_acb_vec_set_powers (C function), 78
_acb_vec_set_round (C function), 78
_acb_vec_sort_pretty (C function), 78
_acb_vec_sub (C function), 78
_acb_vec_trim (C function), 78
_acb_vec_unit_roots (C function), 78
Index 243
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acb_add_arb</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_add_error_mag</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_add_si</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_add_ui</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_addmul</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_addmul_arb</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_addmul_fmpz</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_addmul_si</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_addmul_ui</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_agm1</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_agm1_cpx</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_allocated_bytes</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_arg</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_asin</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_asinh</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_atan</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_atanh</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_barnes_g</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_bernoulli_poly_ui</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_bits</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cacal_cauxpy_bound</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cacal_func_t</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_cacal_integrate</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cacal_integrate_gl_auto_deg</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cacal_integrate_opt_init</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cacal_integrate_opt_struct</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_cacal_integrate_opt_t</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_cacal_integrate_opt_t.deg_limit</code></td>
<td>C member</td>
</tr>
<tr>
<td><code>acb_cacal_integrate_opt_t.depth_limit</code></td>
<td>C member</td>
</tr>
<tr>
<td><code>acb_cacal_integrate_opt_t.eval_limit</code></td>
<td>C member</td>
</tr>
<tr>
<td><code>acb_cacal_integrate_opt_t.use_heap</code></td>
<td>C member</td>
</tr>
<tr>
<td><code>acb_cacal_integrate_taylor</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_chebyshev_t2_ui</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_chebyshev_t2_ui</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_chebyshev_u2_ui</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_chebyshev_u2_ui</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_clear</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_conj</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_const_pi</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_contains</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_contains_fmpq</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_contains_fmpz</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_contains_int</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_contains_zero</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cos</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cos_pi</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cosh</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cot</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cot_pi</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_coth</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_csc</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_csch</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_csgn</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_cube</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dirichlet_chir</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dirichlet_chi_theta_arb</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dirichlet_chir_vec</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dirichlet_dft</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dirichlet_dft_conrey</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dirichlet_dft_conrey</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dieter_hurwitz_precomp_choose_param</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_blustein</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_bluestein</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_bluestein.clear</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_bluestein_init</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_bluestein_precomp</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_bluestein_struct</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_bluestein_t</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_convol</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_convol_nai</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_convol_rad2</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_crt</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_crt.clear</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_crt_init</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_crt_precomp</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_crt_struct</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_crt_t</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_cyc</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_cyc_clear</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_cyc_init</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_cyc_precomp</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_cyc_struct</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_cyc_t</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_reverse</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_reverse_precomp</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_reverse_rad2</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_naive</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_naive_clear</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_naive_init</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_naive_precomp</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_naive_struct</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_naive_t</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_pre_struct</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_pre_t</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_precomp</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_precomp.clear</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_precomp.init</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_prod</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_prod.clear</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_prod_init</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_prod_struct</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_prod_t</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_rad2</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_rad2.clear</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_rad2_init</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_rad2.precomp</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dft_rad2.struct</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_dft_rad2_t</code></td>
<td>C type</td>
</tr>
<tr>
<td><code>acb_digamma</code></td>
<td>C function</td>
</tr>
<tr>
<td><code>acb_dirichlet hurwitz precomp choose param</code></td>
<td>C function</td>
</tr>
</tbody>
</table>

**Index**: 245
acb_get_real (C function), 69
acb_get_unique_fmpz (C function), 69
acb_hurwitz_zeta (C function), 76
acb_hypgeom_0f1 (C function), 132
acb_hypgeom_0f1_asym (C function), 132
acb_hypgeom_0f1_direct (C function), 132
acb_hypgeom_1f1 (C function), 132
acb_hypgeom_2f1 (C function), 139
acb_hypgeom_2f1_choose (C function), 139
acb_hypgeom_2f1_continuation (C function), 139
acb_hypgeom_2f1_corner (C function), 139
acb_hypgeom_2f1_series_direct (C function), 139
acb_hypgeom_2f1_transform (C function), 139
acb_hypgeom_2f1_transform_limit (C function), 139
acb_hypgeom_airy (C function), 135
acb_hypgeom_airy_asym (C function), 135
acb_hypgeom_airy_bound (C function), 135
acb_hypgeom_airy_direct (C function), 135
acb_hypgeom_airy_jet (C function), 135
acb_hypgeom_airy_series (C function), 135
acb_hypgeom_bessel_i (C function), 134
acb_hypgeom_bessel_i_0f1 (C function), 134
acb_hypgeom_bessel_i_asym (C function), 134
acb_hypgeom_bessel_j (C function), 134
acb_hypgeom_bessel_j_0f1 (C function), 133
acb_hypgeom_bessel_j_asym (C function), 133
acb_hypgeom_bessel_k (C function), 134
acb_hypgeom_bessel_k_0f1 (C function), 134
acb_hypgeom_bessel_k_0f1_series (C function), 134
acb_hypgeom_bessel_k_asym (C function), 134
acb_hypgeom_bessel_y (C function), 134
acb_hypgeom_beta_lower (C function), 137
acb_hypgeom_beta_lower_series (C function), 137
acb_hypgeom_chebyshev_t (C function), 140
acb_hypgeom_chebyshev_u (C function), 140
acb_hypgeom_chi (C function), 138
acb_hypgeom_chi_2f3 (C function), 138
acb_hypgeom_chi_asym (C function), 138
acb_hypgeom_chi_series (C function), 138
acb_hypgeom_ci (C function), 138
acb_hypgeom_ci_2f3 (C function), 138
acb_hypgeom_ci_asym (C function), 138
acb_hypgeom_ci_series (C function), 138
acb_hypgeom_dilog (C function), 142
acb_hypgeom_dilog_bernoulli (C function), 142
acb_hypgeom_dilog_bitburst (C function), 142
acb_hypgeom_dilog_continuation (C function), 142
acb_hypgeom_dilog_transform (C function), 142
acb_hypgeom_dilog_zero (C function), 142
acb_hypgeom_dilog_zero_taylor (C function), 142
acb_hypgeom_ei (C function), 137
acb_hypgeom_ei_2f2 (C function), 137
acb_hypgeom_ei_asym (C function), 137
acb_hypgeom_ei_series (C function), 137
acb_hypgeom_erf (C function), 132
acb_hypgeom_erf_1f1a (C function), 132
acb_hypgeom_erf_1f1b (C function), 132
acb_hypgeom_erf_asym (C function), 132
acb_hypgeom_erf_propagated_error (C function), 132
acb_hypgeom_erf_series (C function), 133
acb_hypgeom_ercf (C function), 133
acb_hypgeom_ercf_series (C function), 133
acb_hypgeom_eri (C function), 133
acb_hypgeom_eri_series (C function), 133
acb_hypgeom_expint (C function), 137
acb_hypgeom_fresnel (C function), 133
acb_hypgeom_fresnel_series (C function), 137
acb_hypgeom_gamma_lower (C function), 136
acb_hypgeom_gamma_lower_series (C function), 136
acb_hypgeom_gamma_upper (C function), 136
acb_hypgeom_gamma_upper_1f1a (C function), 136
acb_hypgeom_gamma_upper_1f1b (C function), 136
acb_hypgeom_gamma_upper_asym (C function), 136
acb_hypgeom_gamma_upper_series (C function), 136
acb_hypgeom_gamma_upper_singular (C function), 136
acb_hypgeom_gegenbauer_c (C function), 140
acb_hypgeom_hermite_h (C function), 141
acb_hypgeom_laguerre_l (C function), 140
acb_hypgeom_legendre_p (C function), 141
acb_hypgeom_legendre_p_uiui_rec (C function), 141
acb_hypgeom_li (C function), 138
acb_hypgeom_legendre_q (C function), 141
acb_hypgeom_l_i (C function), 138
acb_hypgeom_pfq (C function), 131
acb_hypgeom_pfq_bound_factor (C function), 129
acb_hypgeom_pfq_choose_n (C function), 129
acb_hypgeom_pfq_direct (C function), 130
acb_hypgeom_pfq_series_direct (C function), 131
acb_hypgeom_pfq_series_sum (C function), 130
acb_hypgeom_pfq_series_sum_bs (C function), 130
acb_hypgeom_pfq_series_sum_forward (C function), 130
acb_hypgeom_pfq_series_sum_rs (C function), 130
Arb Documentation, Release 2.13.0

acb_modular_elliptic_p_zpx (C function), 158
acb_modular_epsilon_arg (C function), 157
acb_modular_eta (C function), 157
acb_modular_eta_sum (C function), 157
acb_modular_fill_addseq (C function), 153
acb_modular_fundamental_domain_approx (C
function), 153
acb_modular_fundamental_domain_approx_arf
(C function), 153
acb_modular_fundamental_domain_approx_d
(C function), 153
acb_modular_hilbert_class_poly (C function),
158
acb_modular_is_in_fundamental_domain
(C
function), 153
acb_modular_j (C function), 157
acb_modular_lambda (C function), 157
acb_modular_theta (C function), 156
acb_modular_theta_const_sum (C function), 156
acb_modular_theta_const_sum_basecase
(C
function), 156
acb_modular_theta_const_sum_rs (C function),
156
acb_modular_theta_jet (C function), 156
acb_modular_theta_jet_notransform (C function), 156
acb_modular_theta_notransform (C function),
156
acb_modular_theta_sum (C function), 155
acb_modular_theta_transform (C function), 154
acb_modular_transform (C function), 153
acb_mul (C function), 70
acb_mul_2exp_fmpz (C function), 70
acb_mul_2exp_si (C function), 70
acb_mul_arb (C function), 70
acb_mul_fmpz (C function), 70
acb_mul_onei (C function), 70
acb_mul_si (C function), 70
acb_mul_ui (C function), 70
acb_ne (C function), 68
acb_neg (C function), 69
acb_neg_round (C function), 69
acb_one (C function), 66
acb_onei (C function), 66
acb_overlaps (C function), 68
acb_poly_add (C function), 97
acb_poly_add_series (C function), 97
acb_poly_add_si (C function), 97
acb_poly_agm1_series (C function), 109
acb_poly_allocated_bytes (C function), 95
acb_poly_atan_series (C function), 104
acb_poly_binomial_transform (C function), 103
acb_poly_binomial_transform_basecase (C function), 103
acb_poly_binomial_transform_convolution
(C
function), 103
acb_poly_borel_transform (C function), 102
acb_poly_clear (C function), 95

Index

acb_poly_compose (C function), 100
acb_poly_compose_divconquer (C function), 99
acb_poly_compose_horner (C function), 99
acb_poly_compose_series (C function), 100
acb_poly_compose_series_brent_kung (C function), 100
acb_poly_compose_series_horner (C function),
100
acb_poly_contains (C function), 96
acb_poly_contains_fmpq_poly (C function), 96
acb_poly_contains_fmpz_poly (C function), 96
acb_poly_cos_pi_series (C function), 105
acb_poly_cos_series (C function), 105
acb_poly_cosh_series (C function), 106
acb_poly_cot_pi_series (C function), 105
acb_poly_degree (C function), 95
acb_poly_derivative (C function), 102
acb_poly_digamma_series (C function), 106
acb_poly_div_series (C function), 99
acb_poly_divrem (C function), 99
acb_poly_elliptic_k_series (C function), 109
acb_poly_elliptic_p_series (C function), 109
acb_poly_equal (C function), 96
acb_poly_erf_series (C function), 109
acb_poly_evaluate (C function), 101
acb_poly_evaluate2 (C function), 101
acb_poly_evaluate2_horner (C function), 101
acb_poly_evaluate2_rectangular (C function),
101
acb_poly_evaluate_horner (C function), 100
acb_poly_evaluate_rectangular (C function), 101
acb_poly_evaluate_vec_fast (C function), 102
acb_poly_evaluate_vec_iter (C function), 101
acb_poly_exp_pi_i_series (C function), 104
acb_poly_exp_series (C function), 104
acb_poly_exp_series_basecase (C function), 104
acb_poly_find_roots (C function), 110
acb_poly_fit_length (C function), 95
acb_poly_fprintd (C function), 96
acb_poly_gamma_series (C function), 106
acb_poly_get_coeff_acb (C function), 96
acb_poly_get_coeff_ptr (C macro), 96
acb_poly_get_unique_fmpz_poly (C function),
96
acb_poly_init (C function), 95
acb_poly_integral (C function), 102
acb_poly_interpolate_barycentric (C function),
102
acb_poly_interpolate_fast (C function), 102
acb_poly_interpolate_newton (C function), 102
acb_poly_inv_borel_transform (C function), 102
acb_poly_inv_series (C function), 99
acb_poly_is_one (C function), 95
acb_poly_is_real (C function), 97
acb_poly_is_x (C function), 95
acb_poly_is_zero (C function), 95
acb_poly_lambertw_series (C function), 106
acb_poly_length (C function), 95

249


acb_poly_lgamma_series (C function), 106
acb_poly_log1p_series (C function), 104
acb_poly_log_series (C function), 104
acb_poly_majorant (C function), 97
acb_poly_mul (C function), 98
acb_poly_mullow (C function), 98
acb_poly_mullow_classical (C function), 98
acb_poly_mullow_transpose (C function), 98
acb_poly_mullow_transpose_gauss (C function), 98
acb_poly_neg (C function), 97
acb_poly_one (C function), 95
acb_poly_overlaps (C function), 96
acb_poly_polylog_series (C function), 108
acb_poly_pow_acb_series (C function), 103
acb_poly_pow_series (C function), 103
acb_poly_pow_ui (C function), 103
acb_poly_pow_ui_trunc_binexp (C function), 103
acb_poly_printd (C function), 96
acb_poly_product_roots (C function), 101
acb_poly_randtest (C function), 96
acb_poly_revert_series (C function), 100
acb_poly_revert_series_lagrange (C function), 100
acb_poly_revert_series_lagrange_fast (C function), 100
acb_poly_revert_series_newton (C function), 100
acb_poly_rgamma_series (C function), 106
acb_poly_rising_ui_series (C function), 107
acb_poly_root_bound_fujiwara (C function), 109
acb_poly_rsqrt_series (C function), 104
acb_poly_scalar_div (C function), 98
acb_poly_scalar_mul (C function), 98
acb_poly_scalar_mul_2exp_si (C function), 98
acb_poly_set (C function), 95
acb_poly_set2_arb_poly (C function), 97
acb_poly_set2_fmpz_poly (C function), 97
acb_poly_set2_fmpq_poly (C function), 97
acb_poly_set_acb (C function), 97
acb_poly_set_arb_poly (C function), 97
acb_poly_set_coeff_acb (C function), 95
acb_poly_set_coeff_si (C function), 95
acb_poly_set_fmpz_poly (C function), 97
acb_poly_set_fmpq_poly (C function), 97
acb_poly_set_round (C function), 95
acb_poly_set_si (C function), 97
acb_poly_set_trunc (C function), 95
acb_poly_set_trunc_round (C function), 95
acb_poly_shift_left (C function), 96
acb_poly_shift_right (C function), 96
acb_poly_sin_cos_pi_series (C function), 105
acb_poly_sin_cos_series (C function), 105
acb_poly_sin_cos_series_basecase (C function), 105
acb_poly_sin_cos_series_tangent (C function), 105
acb_poly_sin_pi_series (C function), 105
acb_poly_sinc_series (C function), 106
acb_poly_sinh_cosh_series (C function), 106
acb_poly_sinh_cosh_series_basecase (C function), 106
acb_poly_sinh_cosh_series_exponential (C function), 106
acb_poly_sinh_series (C function), 106
acb_poly_sqrt_series (C function), 104
acb_poly_struct (C type), 94
acb_poly_sub (C function), 97
acb_poly_swap (C function), 95
acb_poly_t (C type), 94
acb_poly_taylor_shift (C function), 105
acb_poly_taylor_shift_convolution (C function), 105
acb_poly_taylor_shift_divconquer (C function), 105
acb_poly_taylor_shift_horner (C function), 105
acb_poly_truncate (C function), 96
acb_poly_revert_series_lagrange (C function), 100
acb_poly_validate_real_roots (C function), 110
acb_poly_revert_series_newton (C function), 100
acb_poly_zero (C function), 95
acb_poly_zeta_series (C function), 108
acb_polygamma (C function), 75
acb_pow_analytic (C function), 71
acb_pow_arb (C function), 71
acb_pow_fmpz (C function), 71
acb_pow_si (C function), 71
acb_pow_ui (C function), 71
acb_print (C function), 67
acb_printd (C function), 67
acb_printn (C function), 67
acb_ptr (C type), 65
acb_quadratic_roots_fmpz (C function), 71
acb_randtest (C function), 67
acb_randtest_param (C function), 67
acb_randtest_precise (C function), 67
acb_randtest_special (C function), 67
acb_real_abs (C function), 77
acb_real_cell (C function), 77
acb_real_floor (C function), 77
acb_real_heaviside (C function), 77
acb_real_max (C function), 77
acb_real_min (C function), 77
acb_real_sgn (C function), 77
acb_real_sqrtpos (C function), 77
acb_realf (C macro), 65
acb_red_accuracy_bits (C function), 68
acb_red_error_bits (C function), 68
acb_rgamma (C function), 75
acb_rising (C function), 74
acb_rising2_ui (C function), 74
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>arb_chebyshev_u2_ui (C function)</td>
<td>61</td>
</tr>
<tr>
<td>arb_chebyshev_u_ui (C function)</td>
<td>61</td>
</tr>
<tr>
<td>arb_clear (C function)</td>
<td>46</td>
</tr>
<tr>
<td>arb_const_apery (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_const_catalan (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_const_e (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_const_euler (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_const_glaisher (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_const_khinchin (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_const_log10 (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_const_log2 (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_const_log_sqrt2pi (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_const_pi (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_const_sqrt_pi (C function)</td>
<td>58</td>
</tr>
<tr>
<td>arb_contains (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_contains_arf (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_contains_fmpz (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_contains_int (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_contains_mpfr (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_contains_negative (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_contains_nonnegative (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_contains_nonpositive (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_contains_positive (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_contains_si (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_contains_zero (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_cos (C function)</td>
<td>56</td>
</tr>
<tr>
<td>arb_cos_pi (C function)</td>
<td>56</td>
</tr>
<tr>
<td>arb_cos_pi_fmpq (C function)</td>
<td>56</td>
</tr>
<tr>
<td>arb_cosh (C function)</td>
<td>57</td>
</tr>
<tr>
<td>arb_cot (C function)</td>
<td>56</td>
</tr>
<tr>
<td>arb_cot_pi (C function)</td>
<td>56</td>
</tr>
<tr>
<td>arb_coth (C function)</td>
<td>57</td>
</tr>
<tr>
<td>arb_csc (C function)</td>
<td>56</td>
</tr>
<tr>
<td>arb_csch (C function)</td>
<td>57</td>
</tr>
<tr>
<td>arb_digamma (C function)</td>
<td>59</td>
</tr>
<tr>
<td>arb_div (C function)</td>
<td>54</td>
</tr>
<tr>
<td>arb_div_2exp1m1_ui (C function)</td>
<td>54</td>
</tr>
<tr>
<td>arb_div_arf (C function)</td>
<td>54</td>
</tr>
<tr>
<td>arb_div_mpfr (C function)</td>
<td>54</td>
</tr>
<tr>
<td>arb_div_si (C function)</td>
<td>54</td>
</tr>
<tr>
<td>arb_div_ui (C function)</td>
<td>54</td>
</tr>
<tr>
<td>arb_doublefac_ui (C function)</td>
<td>59</td>
</tr>
<tr>
<td>arb_eq (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_equal (C function)</td>
<td>51</td>
</tr>
<tr>
<td>arb_equal_si (C function)</td>
<td>51</td>
</tr>
<tr>
<td>arb_euler_number_fmpz (C function)</td>
<td>62</td>
</tr>
<tr>
<td>arb_euler_number_ui (C function)</td>
<td>62</td>
</tr>
<tr>
<td>arb_exp (C function)</td>
<td>56</td>
</tr>
<tr>
<td>arb_exp_arf_bb (C function)</td>
<td>63</td>
</tr>
<tr>
<td>arb_exp_arf_rs_generic (C function)</td>
<td>63</td>
</tr>
<tr>
<td>arb_exp_invexp (C function)</td>
<td>56</td>
</tr>
<tr>
<td>arb_expm1 (C function)</td>
<td>56</td>
</tr>
<tr>
<td>arb_fac_ui (C function)</td>
<td>59</td>
</tr>
<tr>
<td>arb_fib_fmpz (C function)</td>
<td>61</td>
</tr>
<tr>
<td>arb_fib_ui (C function)</td>
<td>61</td>
</tr>
<tr>
<td>arb_floor (C function)</td>
<td>50</td>
</tr>
<tr>
<td>arb_fmpz_div_fmpz (C function)</td>
<td>54</td>
</tr>
<tr>
<td>arb_fmpz_poly_complex_roots (C function)</td>
<td>111</td>
</tr>
<tr>
<td>arb_fmpz_poly_cos_minpoly (C function)</td>
<td>112</td>
</tr>
<tr>
<td>arb_fmpz_poly_delate (C function)</td>
<td>111</td>
</tr>
<tr>
<td>arb_fmpz_poly_deflation (C function)</td>
<td>111</td>
</tr>
<tr>
<td>arb_fmpz_poly_evaluate_hcb (C function)</td>
<td>111</td>
</tr>
<tr>
<td>arb_fmpz_poly_evaluate_hcb_horner (C function)</td>
<td>111</td>
</tr>
<tr>
<td>arb_fmpz_poly_evaluate_hcb_rectangular (C function)</td>
<td>111</td>
</tr>
<tr>
<td>arb_fprint (C function)</td>
<td>48</td>
</tr>
<tr>
<td>arb_fprintf (C function)</td>
<td>48</td>
</tr>
<tr>
<td>arb_fprintfn (C function)</td>
<td>48</td>
</tr>
<tr>
<td>arb_gamma (C function)</td>
<td>59</td>
</tr>
<tr>
<td>arb_gamma_fmpq (C function)</td>
<td>59</td>
</tr>
<tr>
<td>arb_gamma_fmpz (C function)</td>
<td>59</td>
</tr>
<tr>
<td>arb_ge (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_get_abs_lbound_arf (C function)</td>
<td>49</td>
</tr>
<tr>
<td>arb_get_abs_ubound_arf (C function)</td>
<td>49</td>
</tr>
<tr>
<td>arb_get_fmpz_mid_rad_10exp (C function)</td>
<td>50</td>
</tr>
<tr>
<td>arb_get_interval_arf (C function)</td>
<td>50</td>
</tr>
<tr>
<td>arb_get_interval_fmpz_2exp (C function)</td>
<td>49</td>
</tr>
<tr>
<td>arb_get_interval_mpfr (C function)</td>
<td>50</td>
</tr>
<tr>
<td>arb_get_lbound_arf (C function)</td>
<td>49</td>
</tr>
<tr>
<td>arb_get_mag (C function)</td>
<td>49</td>
</tr>
<tr>
<td>arb_get_mag_lower (C function)</td>
<td>49</td>
</tr>
<tr>
<td>arb_get_mag_lower_nonnegative (C function)</td>
<td>49</td>
</tr>
<tr>
<td>arb_get_mid_arb (C function)</td>
<td>49</td>
</tr>
<tr>
<td>arb_get_rad_arb (C function)</td>
<td>49</td>
</tr>
<tr>
<td>arb_get_rand_fmpq (C function)</td>
<td>48</td>
</tr>
<tr>
<td>arb_get_str (C function)</td>
<td>47</td>
</tr>
<tr>
<td>arb_get_ubound_arf (C function)</td>
<td>49</td>
</tr>
<tr>
<td>arb_get_unique_fmpz (C function)</td>
<td>50</td>
</tr>
<tr>
<td>arb_gt (C function)</td>
<td>52</td>
</tr>
<tr>
<td>arb_hurwitz_zeta (C function)</td>
<td>60</td>
</tr>
<tr>
<td>arb_hypgeom_0fl (C function)</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_1fl (C function)</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_2fl (C function)</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_airy (C function)</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_airy_jet (C function)</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_airy_series (C function)</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_bessel_i (C function)</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_bessel_j (C function)</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_bessel_iy (C function)</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_bessel_k (C function)</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_bessel_y (C function)</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_beta_lower (C function)</td>
<td>144</td>
</tr>
<tr>
<td>arb_hypgeom_beta_lower_series (C function)</td>
<td>144</td>
</tr>
<tr>
<td>arb_hypgeom_central_bin_ui (C function)</td>
<td>147</td>
</tr>
<tr>
<td>arb_hypgeom_chebyshev_t (C function)</td>
<td>147</td>
</tr>
<tr>
<td>Function Name</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>arb_hypgeom_chebyshev_u</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_chibeta</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_chibeta_series</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_chi</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_chibeta_series</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_erf</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_erf_series</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_erfc</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_erfc_series</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_erfi</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_erfi_series</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_expint</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_fresnel</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_fresnel_series</td>
<td>144</td>
</tr>
<tr>
<td>arb_hypgeom_gamma_lower</td>
<td>144</td>
</tr>
<tr>
<td>arb_hypgeom_gamma_lower_series</td>
<td>144</td>
</tr>
<tr>
<td>arb_hypgeom_gegenbauer_c</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_hermite_h</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_insum</td>
<td>173</td>
</tr>
<tr>
<td>arb_hypgeom_jacobi_p</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_laguerre_e</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_legendre_p</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_legendre_p_rec</td>
<td>147</td>
</tr>
<tr>
<td>arb_hypgeom_legendre_p_ui</td>
<td>147</td>
</tr>
<tr>
<td>arb_hypgeom_legendre_p_ui_asym</td>
<td>147</td>
</tr>
<tr>
<td>arb_hypgeom_legendre_p_ui_deriv_bound</td>
<td>147</td>
</tr>
<tr>
<td>arb_hypgeom_legendre_p_ui_one</td>
<td>147</td>
</tr>
<tr>
<td>arb_hypgeom_legendre_p_ui_root</td>
<td>147</td>
</tr>
<tr>
<td>arb_hypgeom_legendre_p_ui_zero</td>
<td>147</td>
</tr>
<tr>
<td>arb_hypgeom_legendre_q</td>
<td>146</td>
</tr>
<tr>
<td>arb_hypgeom_li</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_li_series</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_m</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_pfq</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypgeom_shi</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_shi_series</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_si</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_shi_series</td>
<td>145</td>
</tr>
<tr>
<td>arb_hypgeom_sum</td>
<td>173</td>
</tr>
<tr>
<td>arb_hypgeom_u</td>
<td>143</td>
</tr>
<tr>
<td>arb_hypot</td>
<td>54</td>
</tr>
<tr>
<td>arb_indeterminate</td>
<td>48</td>
</tr>
<tr>
<td>arb_init</td>
<td>46</td>
</tr>
<tr>
<td>arb_intersection</td>
<td>49</td>
</tr>
<tr>
<td>arb_inv</td>
<td>54</td>
</tr>
<tr>
<td>arb_is_exact</td>
<td>51</td>
</tr>
<tr>
<td>arb_is_finite</td>
<td>51</td>
</tr>
<tr>
<td>arb_is_int</td>
<td>51</td>
</tr>
<tr>
<td>arb_is_int_2exp_si</td>
<td>51</td>
</tr>
<tr>
<td>arb_is_negative</td>
<td>51</td>
</tr>
<tr>
<td>arb_is_nonnegative</td>
<td>51</td>
</tr>
<tr>
<td>arb_is_nonpositive</td>
<td>51</td>
</tr>
<tr>
<td>arb_is_nonzero</td>
<td>51</td>
</tr>
<tr>
<td>arb_is_one</td>
<td>51</td>
</tr>
<tr>
<td>arb_is_positive</td>
<td>51</td>
</tr>
<tr>
<td>arb_is_zero</td>
<td>51</td>
</tr>
<tr>
<td>arb_log</td>
<td>55</td>
</tr>
<tr>
<td>arb_log1p</td>
<td>56</td>
</tr>
<tr>
<td>arb_log_arf</td>
<td>55</td>
</tr>
<tr>
<td>arb_log_base_ui</td>
<td>56</td>
</tr>
<tr>
<td>arb_log_fmpz</td>
<td>55</td>
</tr>
<tr>
<td>arb_log_hypot</td>
<td>55</td>
</tr>
<tr>
<td>arb_log_ui</td>
<td>55</td>
</tr>
<tr>
<td>arb_log_ui_from_prev</td>
<td>55</td>
</tr>
<tr>
<td>arb_lt</td>
<td>52</td>
</tr>
<tr>
<td>arb_mat_add</td>
<td>119</td>
</tr>
<tr>
<td>arb_mat_allocated_bytes</td>
<td>117</td>
</tr>
<tr>
<td>arb_mat_bound_frobenius_norm</td>
<td>119</td>
</tr>
<tr>
<td>arb_mat_bound_inf_norm</td>
<td>119</td>
</tr>
<tr>
<td>arb_mat_charpoly</td>
<td>122</td>
</tr>
<tr>
<td>arb_mat_cho</td>
<td>120</td>
</tr>
<tr>
<td>arb_mat_clear</td>
<td>117</td>
</tr>
<tr>
<td>arb_mat_contains</td>
<td>118</td>
</tr>
<tr>
<td>arb_mat_contains_fmpq_mat</td>
<td>118</td>
</tr>
<tr>
<td>arb_mat_contains_fmpz_mat</td>
<td>118</td>
</tr>
<tr>
<td>arb_mat_count_is_zero</td>
<td>122</td>
</tr>
<tr>
<td>arb_mat_count_not_is_zero</td>
<td>122</td>
</tr>
<tr>
<td>arb_mat_det</td>
<td>120</td>
</tr>
<tr>
<td>arb_mat_entry</td>
<td>117</td>
</tr>
<tr>
<td>arb_mat_entrywise_is_zero</td>
<td>122</td>
</tr>
<tr>
<td>arb_mat_entrywise_not_is_zero</td>
<td>122</td>
</tr>
<tr>
<td>arb_mat_eq</td>
<td>118</td>
</tr>
<tr>
<td>arb_mat_equal</td>
<td>118</td>
</tr>
<tr>
<td>arb_mat_exp</td>
<td>122</td>
</tr>
<tr>
<td>arb_mat_exp_taylor_sum</td>
<td>122</td>
</tr>
<tr>
<td>arb_mat_fprintd</td>
<td>118</td>
</tr>
<tr>
<td>arb_mat_frobenius_norm</td>
<td>119</td>
</tr>
<tr>
<td>arb_mat_init</td>
<td>117</td>
</tr>
<tr>
<td>arb_mat_inv</td>
<td>120</td>
</tr>
<tr>
<td>arb_mat_inv_cho_precomp</td>
<td>121</td>
</tr>
<tr>
<td>arb_mat_inv_ldl_precomp</td>
<td>121</td>
</tr>
<tr>
<td>arb_mat_is_empty</td>
<td>118</td>
</tr>
<tr>
<td>arb_mat_is_square</td>
<td>118</td>
</tr>
<tr>
<td>arb_mat_ldl</td>
<td>118</td>
</tr>
<tr>
<td>arb_mat_lu</td>
<td>120</td>
</tr>
<tr>
<td>arb_mat_mul</td>
<td>119</td>
</tr>
<tr>
<td>arb_mat_mul_classical</td>
<td>119</td>
</tr>
<tr>
<td>arb_mat_mul_entrywise</td>
<td>119</td>
</tr>
<tr>
<td>arb_mat_mul_threaded</td>
<td>119</td>
</tr>
<tr>
<td>arb_mat_ncols</td>
<td>117</td>
</tr>
<tr>
<td>Function Name</td>
<td>Category</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
</tr>
<tr>
<td>arb_poly_integral</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_interpolate_barycentric</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_interpolate_fast</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_interpolate_newton</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_inv_borel_transform</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_inv_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_is_one</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_is_x</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_is_zero</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_lambertw_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_length</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_lgamma_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_log1p_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_log_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_majorant</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_mul</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_mullow</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_mullow_block</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_mullow_classical</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_mullow_ztrunc</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_neg</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_one</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_overlaps</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_pow_arb_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_pow_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_pow_ui</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_pow_ui_trunc_binexp</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_printd</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_product_roots</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_product_roots_complex</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_randtest</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_revert_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_revert_series_lagrange</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_revert_series_lagrange_fast</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_revert_series_newton</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_rgamma_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_rising_ui_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_root_bound_fujiwara</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_rsqrt_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_scalar_div</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_scalar_mul</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_scalar_mul_2exp_si</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_set</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_set_coeff_arb</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_set_coeff_si</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_set_fmpq_poly</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_set_fmpz_poly</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_set_round</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_set_si</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_set_trunc</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_set_trunc_round</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_shift_left</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_shift_right</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sin_cos_pi_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sin_cos_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sin_cos_series_basecase</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sin_cos_series_exponential</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sin_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sinc_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sinh_cosh_series_basecase</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sinh_cosh_series_exponential</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sinh_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sqrt_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_struct</td>
<td>C type</td>
</tr>
<tr>
<td>arb_poly_sub</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_sub_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_swinnerton_dyer_ui</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_t</td>
<td>C type</td>
</tr>
<tr>
<td>arb_poly_taylor_shift</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_taylor_shift_convolucion</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_taylor_shift_divconquer</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_taylor_shift_hornor</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_truncate</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_valuation</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_zero</td>
<td>C function</td>
</tr>
<tr>
<td>arb_poly_zeta_series</td>
<td>C function</td>
</tr>
<tr>
<td>arb_polylog</td>
<td>C function</td>
</tr>
<tr>
<td>arb_polylog_si</td>
<td>C function</td>
</tr>
<tr>
<td>arb_pos_inf</td>
<td>C function</td>
</tr>
<tr>
<td>arb_pow</td>
<td>C function</td>
</tr>
<tr>
<td>arb_pow_fmpq</td>
<td>C function</td>
</tr>
<tr>
<td>arb_pow_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>arb_pow_fmpz_binexp</td>
<td>C function</td>
</tr>
<tr>
<td>arb_pow_ui</td>
<td>C function</td>
</tr>
<tr>
<td>arb_power_sum_vec</td>
<td>C function</td>
</tr>
<tr>
<td>arb_print</td>
<td>C function</td>
</tr>
<tr>
<td>arb_printd</td>
<td>C function</td>
</tr>
<tr>
<td>arb_printn</td>
<td>C function</td>
</tr>
<tr>
<td>arb_ptr</td>
<td>C type</td>
</tr>
<tr>
<td>arb_radref</td>
<td>C macro</td>
</tr>
<tr>
<td>arb_randtest</td>
<td>C function</td>
</tr>
<tr>
<td>arb_randtest_exact</td>
<td>C function</td>
</tr>
<tr>
<td>arb_randtest_precise</td>
<td>C function</td>
</tr>
<tr>
<td>arb_randtest_special</td>
<td>C function</td>
</tr>
<tr>
<td>arb_randtest_wide</td>
<td>C function</td>
</tr>
<tr>
<td>arb_rel_accuracy_bits</td>
<td>C function</td>
</tr>
<tr>
<td>arb_rel_error_bits</td>
<td>C function</td>
</tr>
<tr>
<td>arb_rgamma</td>
<td>C function</td>
</tr>
<tr>
<td>arb_rising</td>
<td>C function</td>
</tr>
<tr>
<td>arb_rising2_ui</td>
<td>C function</td>
</tr>
</tbody>
</table>
arf_div_fmpz (C function), 42
arf_div_si (C function), 42
arf_div_ui (C function), 42
arf_equal (C function), 39
arf_equal_si (C function), 39
arf_floor (C function), 38
arf_fmpz_div (C function), 42
arf_fmpz_div_fmpz (C function), 42
arf_fprint (C function), 40
arf_fprintd (C function), 41
arf_frexp (C function), 40
arf_get_d (C function), 38
arf_get_fmpz (C function), 38
arf_get_fmpz (C function), 38
arf_get_fmpz_2exp (C function), 38
arf_get_fmpz_fixed_fmpz (C function), 38
arf_get_fmpz_fixed_si (C function), 38
arf_get_mag (C function), 39
arf_get_mag_lower (C function), 39
arf_get_mpfr (C function), 38
arf_get_si (C function), 38
arf_init (C function), 36
arf_init_neg_mag_shallow (C function), 40
arf_init_neg_shallow (C function), 40
arf_init_set_mag_shallow (C function), 40
arf_init_set_shallow (C function), 40
arf_init_set_si (C function), 37
arf_init_set_ui (C function), 37
arf_interval_clear (C function), 176
arf_interval_fprintd (C function), 176
arf_interval_get_arb (C function), 176
arf_interval_init (C function), 176
arf_interval_printd (C function), 176
arf_interval_ptr (C type), 176
arf_interval_set (C function), 176
arf_interval_srcptr (C type), 176
arf_interval_struct (C type), 176
arf_interval_swap (C function), 176
arf_interval_t (C type), 176
arf_is_finite (C function), 37
arf_is_inf (C function), 37
arf_is_int (C function), 39
arf_is_int_2exp_si (C function), 39
arf_is_nan (C function), 37
arf_is_neg_inf (C function), 37
arf_is_normal (C function), 37
arf_is_one (C function), 37
arf_is_pos_inf (C function), 37
arf_is_special (C function), 37
arf_is_zero (C function), 37
arf_mag_add_ulp (C function), 40
arf_mag_fast_add_ulp (C function), 40
arf_mag_set_ulp (C function), 40
arf_max (C function), 39
arf_min (C function), 39
arf_mul (C function), 41
arf_mul_2exp_fmpz (C function), 41
arf_mul_2exp_si (C function), 41
arf_mul_fmpz (C function), 41
arf_mul_mpz (C function), 41
arf_mul_si (C function), 41
arf_mul_ui (C function), 41
arf_nan (C function), 37
arf_neg (C function), 41
arf_neg_round (C function), 41
arf_one (C function), 36
arf_pos_inf (C function), 36
arf_PREC_EXACT (C macro), 36
arf_print (C function), 40
arf_printd (C function), 40
arf_randtest (C function), 40
arf_randtest_not_zero (C function), 40
arf_randtest_special (C function), 40
arf_RND_CEIL (C macro), 36
arf_RND_DOWN (C macro), 36
arf_RND_FLOOR (C macro), 36
arf_RND_NEAREST (C macro), 36
arf_rnd_t (C type), 36
arf_root (C function), 42
arf_rsqrt (C function), 42
arf_set (C function), 37
arf_set_d (C function), 37
arf_set_fmpr (C function), 37
arf_set_fmpz (C function), 37
arf_set_round (C function), 37
arf_set_round_fmpz (C function), 37
arf_set_round_fmpz_2exp (C function), 38
arf_set_round_mpz (C function), 37
arf_set_round_si (C function), 37
arf_set_round_ui (C function), 37
arf_sign (C function), 37
arf_sign (C function), 37
arf_si_div (C function), 42
arf_sosq (C function), 42
arf_sqrt (C function), 42
arf_sqrt_fmpz (C function), 42
arf_sqrt_ui (C function), 42
arf_struct (C type), 35
arf_sub (C function), 41
arf_sub_fmpz (C function), 41
arf_sub_si (C function), 41
arf_sub_ui (C function), 41
arf_summul (C function), 42
arf_summul_fmpz (C function), 42
arf_summul_mpz (C function), 42
arf_summul_si (C function), 42
arf_summul_ui (C function), 42

Index 257
Index

D

dirichlet_char_clear (C function), 159
dirichlet_char_eq (C function), 160
dirichlet_char_eq_deep (C function), 160
dirichlet_char_exp (C function), 160
dirichlet_char_first_primitive (C function), 160
dirichlet_char_index (C function), 160
dirichlet_char_init (C function), 159
dirichlet_char_is_primitive (C function), 161
dirichlet_char_is_principal (C function), 160
dirichlet_char_is_real (C function), 161
dirichlet_char_lift (C function), 161
dirichlet_char_log (C function), 160
dirichlet_char_lower (C function), 161
dirichlet_char_mul (C function), 161
dirichlet_char_next (C function), 160
dirichlet_char_next_primitive (C function), 160
dirichlet_char_one (C function), 160
dirichlet_char_pow (C function), 161
dirichlet_char_print (C function), 160
dirichlet_char_set (C function), 160
dirichlet_char_struct (C type), 159
dirichlet_char_t (C type), 159
dirichlet_chi (C function), 161
dirichlet_chi_vec (C function), 161
dirichlet_chi_vec_order (C function), 161
dirichlet_conductor_char (C function), 161
dirichlet_conductor_ui (C function), 160
dirichlet_dirichlet_conductor (C function), 161
dirichlet_group_clear (C function), 159
dirichlet_group_clear (C function), 159
dirichlet_group_num_primitive (C function), 159
dirichlet_group_size (C function), 159
dirichlet_group_struct (C type), 159
dirichlet_group_t (C type), 159

dirichlet_index_char (C function), 160
dirichlet_index_char (C function), 160

dirichlet_order_char (C function), 161
dirichlet_order_char (C function), 161

dirichlet_order_ui (C function), 161

dirichlet_pairing (C function), 161
dirichlet_pairing_char (C function), 161
dirichlet_parity_char (C function), 161
dirichlet_parity_ui (C function), 161

dirichlet_subgroup_init (C function), 159
dlog_bsgs (C function), 189

dlog_bsgs_clear (C function), 189
dlog_bsgs_init (C function), 189
dlog_bsgs_struct (C type), 189
dlog_bsgs_t (C type), 189
dlog_crt (C function), 190
dlog_crt_clear (C function), 190
dlog_crt_init (C function), 190
dlog_crt_struct (C type), 190
dlog_crt_t (C type), 190
dlog_modpe (C function), 190
dlog_modpe_clear (C function), 190
dlog_modpe_init (C function), 190

258 Index
fmpr_print (C function), 195
fmpr_printd (C function), 195
fmpr_randtest (C function), 195
fmpr_randtest_not_zero (C function), 195
fmpr_randtest_special (C function), 195
FMPR_RND_CEIL (C macro), 192
FMPR_RND_DOWN (C macro), 192
FMPR_RND_FLOOR (C macro), 192
FMPR_RND_NEAR (C macro), 192
FMPR_RND_UP (C macro), 192
fmpr_root (C function), 197
fmpr_sqrt (C function), 197
fmpr_sqrt_fmpz (C function), 197
fmpr_set (C function), 193
fmpr_set_d (C function), 194
fmpr_set_error_result (C function), 193
fmpr_set_fmpq (C function), 194
fmpr_set_fmpz (C function), 194
fmpr_set_fmpz_2exp (C function), 194
fmpr_set_mpfr (C function), 194
fmpr_set_round (C function), 193
fmpr_set_round_fmpz (C function), 193
fmpr_set_round_fmpz_2exp (C function), 194
fmpr_set_round_ui_2exp_fmpz (C function), 193
fmpr_set_round_uiui_2exp_fmpz (C function), 193
fmpr_set_si (C function), 194
fmpr_set_si_2exp_si (C function), 194
fmpr_set ui (C function), 194
fmpr_sgn (C function), 195
fmpr_sqrt (C function), 195
fmpr_sqrt ui (C function), 195
fmpr_sqrt_fmpz (C function), 195
fmpr_sqrt ui (C function), 195
fmpr_swap (C function), 193
fmpr_ui_div (C function), 195
fmpr_ui_div (C function), 193
fmpr_t (C function), 191
fmpr_ui_div (C function), 196
fmpr_uulp (C function), 193
fmpr_zero (C function), 192
fmpr_sub2_fmpz si inline (C function), 184
fmpr_sub2 inline (C function), 183
fmpr_sub si (C function), 183
fmpr_sub si inline (C function), 184
fmpr_sub ui inline (C function), 184
fmpr_ulp_q_2exp (C function), 184
FMPZ_GET_MPN_READONLY (C macro), 184
fmpr_ishift_mpn (C function), 184
fmpr_mat t (C function), 14
fmpr_max (C function), 183
fmpr_min (C function), 183
fmpr_poly t (C function), 14
fmpr_set_mpn (C function), 184
fmpr_sub si (C function), 183
fmpr_sub si large (C function), 184
fmpr_t (C function), 14
fmpr ui_mul ui (C function), 183
fmpr ui_pow ui (C function), 183

H
hypgeom_bound (C function), 172
hypgeom_clear (C function), 172
hypgeom_estimate_terms (C function), 172
hypgeom_init (C function), 172
hypgeom_precompute (C function), 173
hypgeom_struct (C type), 172
hypgeom t (C type), 172

M
mag_add (C function), 32
mag_add 2exp fmpz (C function), 32
mag_add lower (C function), 32
mag_add ui (C function), 32
mag_add ui 2exp si (C function), 32
mag_addmul (C function), 32
mag_allocated_bytes (C function), 29
mag_atan (C function), 34
mag atan lower (C function), 34
mag bernoulli div fac ui (C function), 34
mag bin ui (C function), 34
mag binpow uiui (C function), 34
mag clear (C function), 29
mag cmp (C function), 30
mag cmp 2exp si (C function), 30
mag const pi (C function), 34
mag const pi lower (C function), 34
mag div (C function), 32
mag div fmpz (C function), 32
mag div lower (C function), 32
mag div ui (C function), 32
mag equal (C function), 30
mag exp (C function), 33
mag exp lower (C function), 33
mag exp tail (C function), 34
mag expinv (C function), 33
mag expinv lower (C function), 34
mag expmul1 (C function), 34
mag fac ui (C function), 34
mag fast add 2exp si (C function), 33
mag fast addmul (C function), 33
mag fast init set (C function), 32
mag fast init set arf (C function), 40
mag fast is_zero (C function), 32
mag fast mul (C function), 32

260 Index