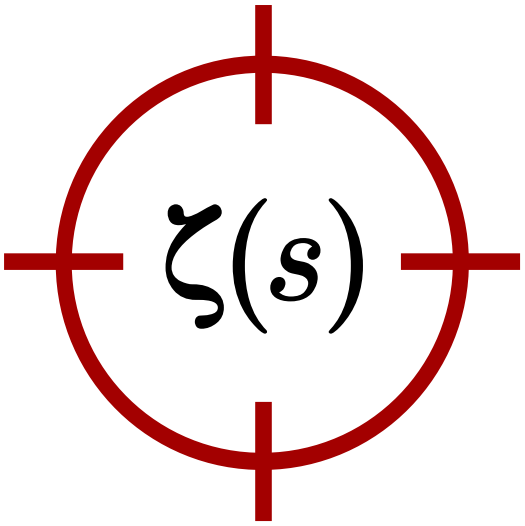

Arb

Arb Documentation

Release 2.18.1

Fredrik Johansson

Jun 25, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

Welcome to Arb's documentation! Arb is a C library for rigorous real and complex arithmetic with arbitrary precision. Arb tracks numerical errors automatically using *ball arithmetic*, a form of interval arithmetic based on a midpoint-radius representation. On top of this, Arb provides a wide range of mathematical functionality, including polynomials, power series, matrices, integration, root-finding, and transcendental functions. Arb is designed with efficiency as a primary goal, and is usually competitive with or faster than other arbitrary-precision packages. 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`. This edition of the documentation was updated Jun 25, 2020 and describes Arb 2.18.1. Documentation for *specific release versions* is also available in PDF format.

CHAPTER

TWO

GENERAL INFORMATION

2.1 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.

- Modules (*arb_poly*, *acb_poly*) for polynomials or power series over the real and complex numbers, implemented using balls as coefficients, with asymptotically fast polynomial multiplication and many other operations.
- Modules (*arb_mat*, *acb_mat*) for matrices over the real and complex numbers, implemented using balls as coefficients. At the moment, only rudimentary linear algebra operations are provided.
- Functions for high-precision evaluation of various mathematical constants and special functions, implemented using ball arithmetic with rigorous error bounds.

Arb 1.x used a different set of numerical base types (*fmpr*, *fmprb* and *fmpcb*). These types had a slightly simpler internal representation, but generally had worse performance. All methods for the Arb 1.x types have now been ported to faster equivalents for the Arb 2.x types. The last version to include both the Arb 1.x and Arb 2.x types and methods was Arb 2.2. As of Arb 2.9, only a small set of *fmpr* methods are left for fallback and testing purposes.

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

2.2 Setup

2.2.1 Package managers

The easiest way to install Arb including all dependencies is via ready-made packages available for various distributions. Note that some package managers may not have the latest version of Arb.

- 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/
- Guix
  - https://www.gnu.org/software/guix/packages/A/
- Anaconda
  - https://anaconda.org/conda-forge/arb

Installing SageMath or Nemo (see below) will also create an installation of Arb local to those systems. It is possible to link user code to that installation by setting the proper paths.

2.2.2 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 recommended for keeping up with the latest improvements and bug fixes and should be safe to use at all times (only stable code that passes the test suite gets merged into the git master).

2.2.3 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

2.2.4 Standalone installation

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

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

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

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.

2.2.5 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
```

2.2.6 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:

```
mkdir build
cd build
cmake ..                                          # configure
cmake --build . --config Release                  # build
cmake --build . --config Release --target install # install
```

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_TESTING=yes` to `cmake` and run `ctest` to run the tests.

2.2.7 Running code

Here is an example program to get started using Arb:

```
#include "arb.h"

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

Compile it with:

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

Depending on the environment, you may also have to pass the flags `-lflint`, `-lmpfr`, `-lgmp` to the compiler. On some Debian based systems, `-larb` needs to be replaced with `-lflint-arb`.

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:

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

2.2.8 Computer algebra systems and wrappers

- Python-FLINT (https://github.com/fredrik-johansson/python-flint) is a convenient Python interface to both FLINT and Arb.
- SageMath (http://sagemath.org/) includes Arb as a standard package and contains a high-level Python interface. Refer to the SageMath documentation:
  - RealBallField: http://doc.sagemath.org/html/en/reference/rings_numerical/sage/rings/real_arb.html
  - ComplexBallField: http://doc.sagemath.org/html/en/reference/rings_numerical/sage/rings/complex_arb.html

- Nemo (http://nemocas.org/) is a computer algebra package for the Julia programming language which includes a high-level Julia interface to Arb. The Nemo installation script will create a local installation of Arb along with other dependencies.
  - Real balls: http://nemocas.github.io/Nemo.jl/latest/arb.html
  - Complex balls: http://nemocas.github.io/Nemo.jl/latest/acb.html
- Arblib.jl (https://github.com/kalmarek/Arblib.jl) is a thin, efficient Julia wrapper around Arb.
- Other wrappers include:
  - ArbNumerics (Julia): https://github.com/JeffreySarnoff/ArbNumerics.jl
  - ArbFloats (Julia): https://github.com/JuliaArbTypes/ArbFloats.jl
  - A Java wrapper using JNA: https://github.com/crowlogic/arb/

2.3 Using ball arithmetic

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

2.3.1 Ball semantics

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

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

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

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

General subsets of $\mathbb{R}$ are not possible to represent on a computer. Instead, we work with subsets of the form $[m \pm r] = [m - r, m + r]$ where the midpoint m and radius r are binary floating-point numbers, i.e. numbers of the form $u2^v$ with $u, v \in \mathbb{Z}$ (to make this scheme complete, we also need to adjoin the special floating-point values $-\infty$, $+\infty$ and NaN).

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

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

```
#include "arb.h"

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

(continues on next page)

(continued from previous page)

```
    arb_const_pi(y, 128);   /* y = pi, to 128 bits */
    arb_sub(y, y, x, 53);   /* y = y - x, to 53 bits */
    arb_clear(x); arb_clear(y);
}
```

2.3.2 Binary and decimal

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

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

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

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

The output is:

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

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

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

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

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

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

2.3.3 Quality of enclosures

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

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

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

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

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

2.3.4 Predicates

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

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

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

Instead, the following can be used:

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

(continues on next page)

(continued from previous page)

```
{
    ... /* do things assuming that the sign of x is unknown */
}
```

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

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

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

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

2.3.5 A worked example: the sine function

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

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

```
#include "arb.h"

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

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

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

(continues on next page)

(continued from previous page)

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

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

    }

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

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

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

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

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

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

The program produces the following output:

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

As an exercise, the reader may improve the naive algorithm by making it subtract a well-chosen multiple of 2π from x before invoking the Taylor series (hint: use *arb_const_pi()*, *arb_div()* and *arf_get_fmpz()*). If done correctly, 64 bits of precision should be more than enough to compute sin(2016.1), and with minor adjustments to the code, the user should be able to compute sin(exp(2016.1)) quite easily as well.

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

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

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

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

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

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

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

2.3.6 More on precision and accuracy

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

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

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

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

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

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

the absolute or relative error will be bounded by

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

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

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

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

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

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

2.3.7 Polynomial time guarantee

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

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

As motivation, consider evaluating $\sin(x)$ or $\exp(x)$ with the exact floating-point number $x = 2^{2^n}$ as input. The time and space required to compute an accurate floating-point approximation of $\sin(x)$ or $\exp(x)$ increases as 2^n, in the first case because because of the need to subtract an accurate multiple of 2π and in the second case due to the size of the output exponent and the internal subtraction of an accurate multiple of $\log(2)$. This is despite the fact that the size of x as an object in memory only increases linearly with n. Already $n = 33$ would require at least 1 GB of memory, and $n = 100$ would be physically impossible to process. For functions that are computed by direct use of power series

expansions, e.g. $f(x) = \sum_{k=0}^{\infty} c_k x^k$, without having fast argument-reduction techniques like those for elementary functions, the time would be exponential in n already when $x = 2^n$.

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

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

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

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

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

2.4 Technical conventions and potential issues

2.4.1 Integer types

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

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

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

type slong

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

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

type ulong

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

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

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

type mp_limb_t

A single limb.

type mp_ptr
Pointer to a writable array of limbs.

type mp_srcptr
Pointer to a read-only array of limbs.

type mp_size_t
A limb count (always nonnegative).

type flint_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.

type fmpz

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

An *fmpz_t* is defined as an array of length one of type *fmpz* (which is just an alias for *slong*), permitting an *fmpz_t* to be passed by reference.

type fmpq_t
FLINT multi-precision rational number.

type fmpz_poly_t

type fmpq_poly_t

type fmpz_mat_t

type fmpq_mat_t
FLINT polynomials and matrices with integer and rational coefficients.

2.4.2 Integer overflow

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

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

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

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

2.4.3 Aliasing

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

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

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

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

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

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

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

and not the (generally larger) set

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

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

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

2.4.4 Thread safety and caches

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

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

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

2.4.5 Use of hardware floating-point arithmetic

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

First, `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.

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

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

2.4.6 Interface changes

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

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

2.4.7 General note on correctness

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

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

All bug reports are highly appreciated.

2.5 Contributing to Arb

The Arb project welcomes new feature contributions in addition to patches for bugs and performance problems.

What are appropriate new features? Most of the numerical functionality that can be found in a general-purpose computer algebra system is certainly within scope (of course, the main restriction is that the algorithm must have a proof of correctness). However, if the functionality is easily accomplished by combining existing functions in Arb, consider whether it is worth the increase in code size, maintenance effort and test time. Prospective contributors are recommended to discuss their ideas on the mailing list or the issue tracker.

The process for actually submitting code is simple: anyone can submit a pull request on GitHub. If the patch looks good to the maintainer and the test code passes, the patch will get merged into the git master.

2.5.1 Code conventions

Four steps are needed to add a new function:

- Add the function `module_foo()` in a new file `module/foo.c`.
- Add a corresponding test program in a new file `module/test/t-foo.c`.
- Add the function prototype to `module.h`.
- Document the function in `doc/source/module.rst`.

The build system takes care of everything else automatically.

Test code (see below) can be omitted if `module_foo()` is a trivial helper function, but it should at least be tested indirectly via another function in that case. Auxiliary functions needed to implement `module_foo()` but which have no use elsewhere should be declared as `static` in `module/foo.c`. If `module_foo()` is very short, it can be declared inline directly in `module.h` with the `MODULE_INLINE` macro.

Use the following checklist regarding code style:

- Try to keep names and function arguments consistent with existing code.
- Follow the conventions regarding types, aliasing rules, etc. described in *Technical conventions and potential issues* and in `code_conventions.txt` in FLINT (https://github.com/wbhart/flint2/blob/trunk/code_conventions.txt).
- Use basic FLINT constants, types and functions: `FLINT_BITS`, `flint_malloc`/`flint_free`, `flint_abort`, `flint_printf`, etc.
- Complex macros should be avoided.
- Indentation is four spaces.
- Curly braces normally go on a new line.
- Binary operators are surrounded by spaces (but parentheses and brackets are not).
- Logically distinct chunks of code (variable declarations, initialization, precomputations, the main loop, cleanup, etc.) should be separated by a single blank line.
- Lines are up to 79 characters long, but this rule can be broken if it helps readability.
- Add correct copyright notices at the top of each file.

2.5.2 Test code

See *Setup* for instructions on running test code.

The easiest way to write a test program for a new function is to adapt the test code for an existing, similar function.

Most of the test code in Arb uses the strategy of computing the same mathematical quantity in two or more different ways (for example, using functional equations, interchanging the order of parameter, or varying the precision and other algorithm parameters) and verifying that the results are consistent. It is also a good idea to test that aliasing works. Input data is usually generated randomly, but in some cases including precomputed reference values also makes sense.

Faster test code is better. A single test program should not take more than 10 seconds to run, and preferably no more than 1 second. Most functions can be tested effectively in less than 0.1 seconds. Think of what the corner cases are and try to generate random input biased toward such cases. The `randtest()` functions attempt to generate corner cases automatically, but some thought may be needed to use them optimally. Try to ensure that the test code fails if you deliberately break the tested function in any way. It is also a good idea to run the test code once with `ARB_TEST_MULTIPLIER=10.0` or higher. If a function's input space is too large to probe effectively for corner cases with random input, that can be a hint that the function should be split into smaller logical parts that can be tested separately.

The test code must complete without errors when run with `valgrind`. The most common mistake leading to memory corruption or memory leaks is to miss or duplicate an `init()` or `clear()` call. Check that the `init()` and `clear()` calls exactly match the variable declarations in each code block, including the test code itself.

Profiling code is not needed in most cases, but it is often a good idea to run some benchmarks at least during the initial development of a new feature. The `TIMEIT_START`/`TIMEIT_STOP` and `SHOW_MEMORY_USAGE` macros in FLINT are useful for quick measurements.

2.6 Credits and references

2.6.1 License

2.6.2 Authors

Fredrik Johansson is the main author. The project was started in 2012 as a numerical extension of FLINT, and the initial design was heavily based on FLINT 2.0 (with particular credit to Bill Hart and Sebastian Pancratz).

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, bug reports, Debian testing
- 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
- arbguest - preconditioned linear algebra algorithms
- Ralf Stephan - return exact real parts in acos and acosh
- Vincent Delecroix - various feedback and patches, work on Sage interface
- D.H.J Polymath - Riemann xi function
- Joel Dahne - feedback and improvements for Legendre functions
- Gianfranco Costamagna - bug reports, Debian testing
- Julian Rüth - serialization support

2.6.3 Funding

From 2012 to July 2014, Fredrik's work on Arb was supported by Austrian Science Fund FWF Grant Y464-N18 (Fast Computer Algebra for Special Functions). During that period, he was a PhD student (and briefly a postdoc) at RISC, Johannes Kepler University, Linz, supervised by Manuel Kauers.

From September 2014 to October 2015, Fredrik was a postdoc at INRIA Bordeaux and Institut de Mathématiques de Bordeaux, in the LFANT project-team headed by Andreas Enge. During that period, Fredrik's work on Arb was supported by ERC Starting Grant ANTICS 278537 (Algorithmic Number Theory in Computer Science) http://cordis.europa.eu/project/rcn/101288_en.html Since October 2015, Fredrik is a CR2 researcher in the LFANT team, funded by INRIA.

2.6.4 Software

The following software has been helpful in the development of Arb.

- GMP (Torbjörn Granlund and others), http://gmplib.org
- MPIR (Brian Gladman, Jason Moxham, William Hart and others), http://mpir.org
- MPFR (Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Philippe Théveny, Paul Zimmermann and others), http://mpfr.org
- FLINT (William Hart, Sebastian Pancratz, Andy Novocin, Fredrik Johansson, David Harvey and others), http://flintlib.org
- Sage (William Stein and others), http://sagemath.org
- Pari/GP (The Pari group), http://pari.math.u-bordeaux.fr/
- SymPy (Ondřej Čertík, Aaron Meurer and others), http://sympy.org
- mpmath (Fredrik Johansson and others), http://mpmath.org
- Mathematica (Wolfram Research), http://www.wolfram.com/mathematica
- HolonomicFunctions (Christoph Koutschan), http://www.risc.jku.at/research/combinat/software/HolonomicFunctions/
- Sphinx (George Brandl and others), http://sphinx.pocoo.org
- CM (Andreas Enge), http://www.multiprecision.org/index.php?prog=cm
- ore_algebra (Manuel Kauers, Maximilian Jaroschek, Fredrik Johansson), http://www.risc.jku.at/research/combinat/software/ore_algebra/

2.6.5 Citing Arb

To cite Arb in a scientific paper, the following reference can be used:

F. Johansson. "Arb: efficient arbitrary-precision midpoint-radius interval arithmetic", *IEEE Transactions on Computers*, 66(8):1281-1292, 2017. DOI: 10.1109/TC.2017.2690633.

In BibTeX format:

```
@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:

F. Johansson. "Arb: a C library for ball arithmetic", *ACM Communications in Computer Algebra*, 47(4):166-169, 2013.

2.6.6 Bibliography

(In the PDF edition, this section is empty. See the bibliography listing at the end of the document.)

CHAPTER

THREE

EXAMPLE PROGRAMS

3.1 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`.

3.1.1 pi.c

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

Sample output, computing π to one million digits:

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

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

3.1.2 hilbert_matrix.c

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

Sample output:

```
$ build/examples/hilbert_matrix 200
prec=20: [+/- 1.32e-335]
prec=40: [+/- 1.63e-545]
prec=80: [+/- 1.30e-933]
prec=160: [+/- 3.62e-1926]
prec=320: [+/- 1.81e-4129]
prec=640: [+/- 3.84e-8838]
prec=1280: [2.955454297e-23924 +/- 8.29e-23935]
```

(continues on next page)

(continued from previous page)

```
success!
cpu/wall(s): 8.494 8.513
virt/peak/res/peak(MB): 134.98 134.98 111.57 111.57
```

Called with `-eig n`, instead of computing the determinant, the program computes the smallest eigenvalue of the Hilbert matrix (in fact, it isolates all eigenvalues and prints the smallest eigenvalue):

```
$ build/examples/hilbert_matrix -eig 50
prec=20: nan
prec=40: nan
prec=80: nan
prec=160: nan
prec=320: nan
prec=640: [1.459157797e-74 +/- 2.49e-84]
success!
cpu/wall(s): 1.84 1.841
virt/peak/res/peak(MB): 33.97 33.97 10.51 10.51
```

3.1.3 keiper_li.c

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

The program takes the following parameters:

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

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

Sample output:

```
> build/examples/keiper_li 1000 -threads 2
zeta: cpu/wall(s): 0.4 0.244
virt/peak/res/peak(MB): 167.98 294.69 5.09 7.43
log: cpu/wall(s): 0.03 0.038
gamma: cpu/wall(s): 0.02 0.016
binomial transform: cpu/wall(s): 0.01 0.018
0: -0.693147180559945309417232121458176568075500134360 26 +/- 6.5389e-347
1: 0.023095708966121033814310247906495291621932127152051 +/- 2.0924e-345
2: 0.046172867614023335192864243096033943387066108314123 +/- 1.674e-344
3: 0.069212973518108267930497348872601068994212026393 2 +/- 5.0219e-344
4: 0.092197619873060409647627872409439018065541673490213 +/- 2.0089e-343
5: 0.115108542892235490486221281098572766713491323035 96 +/- 1.0044e-342
6: 0.137927668713729882904167137003416663561389660786 54 +/- 6.0264e-342
7: 0.160637159652994212940402872573853662922824420461 63 +/- 2.1092e-341
8: 0.183219459643382579081939317747218598489980982734 32 +/- 8.4368e-341
9: 0.205657338709170461702893874213433047412365534100 44 +/- 7.5931e-340
10: 0.227933936319315774369303405736844533807483859427 38 +/- 7.5931e-339
991: 2.319661796161336792837389965699468256210143081334 1 +/- 2.461e-11
992: 2.320376623925488403534989651833255023316290971728 8 +/- 9.5363e-11
993: 2.321092061239733282811659116333262802034375592414 +/- 1.8495e-10
994: 2.321807354018846211025882612150387011274718888889 3 +/- 3.5907e-10
995: 2.322521739281518572692870295122531402377335815253 3 +/- 6.978e-10
996: 2.323234448581462387333322360941370391235828307128 1 +/- 1.3574e-09
```

(continues on next page)

(continued from previous page)

```
997: 2.32394471148860145228895426675803820345265092324 75 +/- 2.6433e-09
998: 2.32465175910327008083441432403526051488568693222 09 +/- 5.1524e-09
999: 2.32535482758613821198125760520605269885449931621 01 +/- 1.0053e-08
1000: 2.32605316168646645740650469408322381580449820418 72 +/- 3.927e-08
virt/peak/res/peak(MB): 170.18 294.69 7.51 7.51
```

3.1.4 logistic.c

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

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

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

Sample output:

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

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

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

> build/examples/logistic 1234 0.1 3.99 30
Trying prec=64 bits...ran out of accuracy at step 0
Trying prec=128 bits...ran out of accuracy at step 10
Trying prec=256 bits...ran out of accuracy at step 76
Trying prec=512 bits...ran out of accuracy at step 205
Trying prec=1024 bits...ran out of accuracy at step 461
Trying prec=2048 bits...ran out of accuracy at step 974
Trying prec=4096 bits...success!
cpu/wall(s): 0.009 0.009
x_1234 = [0.25644539195865141057967794 5635 +/- 3.92e-31]
```

3.1.5 real_roots.c

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

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

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

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

The following options are available:

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

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

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

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

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

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

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

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

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

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

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

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

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

---------------------------------------------------------------
Found roots: 6
Subintervals possibly containing undetected roots: 0
Function evaluations: 200
cpu/wall(s): 0.003 0.003
virt/peak/res/peak(MB): 26.12 26.14 2.24 2.24
```

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

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

This does not miss any roots:

```
> build/examples/real_roots 2 1 100
interval: [1, 100]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
---------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
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.

3.1.6 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> <m>      The 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...
cpu/wall(s): 0.001 0.001
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]
...
[96.0000000000000 +/- 1e-17]
[97.0000000000000 +/- 1e-17]
[98.0000000000000 +/- 3e-17]
[99.0000000000000 +/- 3e-17]
[100.000000000000 +/- 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...
cpu/wall(s): 0.001 0
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
prec=128: 16 isolated roots | cpu/wall(s): 0.001 0.001
prec=256: 16 isolated roots | cpu/wall(s): 0.001 0.002
prec=512: 16 isolated roots | cpu/wall(s): 0.002 0.001
done!
[-0.94308706466055783383 +/- 2.02e-21]
[-0.75534059252067985752 +/- 2.70e-21]
[-0.24999757119077421009 +/- 4.27e-21]
[0.24999757152512726002 +/- 4.43e-21]
[0.75000242847487273998 +/- 4.43e-21]
[1.2499975711907742101 +/- 1.43e-20]
[1.7553405925206798575 +/- 1.74e-20]
[1.9430870646605578338 +/- 3.21e-20]
[-0.99509334829256233279 +/- 9.42e-22] + [0.44547958157103608805 +/- 3.59e-21]*I
[-0.99509334829256233279 +/- 9.42e-22] + [-0.44547958157103608805 +/- 3.59e-21]*I
[1.9950933482925623328 +/- 1.10e-20] + [0.44547958157103608805 +/- 3.59e-21]*I
[1.9950933482925623328 +/- 1.10e-20] + [-0.44547958157103608805 +/- 3.59e-21]*I
[-0.92177327714429290564 +/- 4.68e-21] + [-1.0954360955079385542 +/- 1.71e-21]*I
[-0.92177327714429290564 +/- 4.68e-21] + [1.0954360955079385542 +/- 1.71e-21]*I
[1.9217732771442929056 +/- 3.54e-20] + [1.0954360955079385542 +/- 1.71e-21]*I
[1.9217732771442929056 +/- 3.54e-20] + [-1.0954360955079385542 +/- 1.71e-21]*I
cpu/wall(s): 0.011 0.012
```

Roots are automatically separated by multiplicity by performing an initial squarefree factorization:

```
> build/examples/poly_roots -print 5 p 5 p 5 t 7 coeffs 1 5 10 10 5 1
computing squarefree factorization...
cpu/wall(s): 0 0
roots with multiplicity 1
searching for 6 roots, 3 deflated
```

(continues on next page)

(continued from previous page)

```
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
```

3.1.7 complex_plot.c

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

The program takes the parameters:

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

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

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 Ai
- `bi` - Airy function Bi
- `besselj` - Bessel function J_0
- `bessely` - Bessel function Y_0

- `besseli` - Bessel function I_0
- `besselk` - Bessel function K_0
- `modj` - Modular j-function
- `modeta` - Dedekind eta function
- `barnesg` - Barnes G-function
- `agm` - Arithmetic geometric mean

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

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

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

3.1.8 lvalue.c

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

```
> build/examples/lvalue
lvalue [-character q n] [-re a] [-im b] [-prec p] [-z] [-deflate] [-len l]

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 $s = 2$:

```
> build/examples/lvalue -re 2 -prec 128
L(s) = [1.64493406684822643647241516664602518922 +/- 4.37e-39]
cpu/wall(s): 0.001 0.001
virt/peak/res/peak(MB): 26.86 26.88 2.05 2.05

> build/examples/lvalue -character 4 3 -re 2 -prec 128
L(s) = [0.91596559417721901505460351493238411077 +/- 7.86e-39]
cpu/wall(s): 0.002 0.003
virt/peak/res/peak(MB): 26.86 26.88 2.31 2.31
```

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

> build/examples/lvalue -character 1009 101 -re 1
L(s) = [0.657952586112728 +/- 6.02e-16] + [1.004145273214022 +/- 3.10e-16]*I
cpu/wall(s): 0.017 0.018
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 +/- 5.29e-19]
L'(s) = [0.072815845483676725 +/- 2.68e-19]
[x^2] L(s+x) = [-0.004845181596436159 +/- 3.87e-19]
```

(continues on next page)

(continued from previous page)

```
[x^3] L(s+x) = [-0.000342305736717224 +/- 4.20e-19]
[x^4] L(s+x) = [9.6890419394471e-5 +/- 2.40e-19]
[x^5] L(s+x) = [-6.6110318108422e-6 +/- 4.51e-20]
[x^6] L(s+x) = [-3.316240908753e-7 +/- 3.85e-20]
[x^7] L(s+x) = [1.0462094584479e-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

> build/examples/lvalue -z -re 14.134725 -prec 200
Z(s) = [-1.124183498394175330011149435812825749786292793565 8e-7 +/- 4.62e-58]
cpu/wall(s): 0.001 0.001
virt/peak/res/peak(MB): 26.86 26.88 2.57 2.57

> build/examples/lvalue -z -re 14.134725 -len 4
Z(s) = [-1.124184e-7 +/- 7.00e-14]
Z'(s) = [0.793160414884 +/- 4.09e-13]
[x^2] Z(s+x) = [0.065164586492 +/- 5.39e-13]
[x^3] Z(s+x) = [-0.020707762705 +/- 5.37e-13]
cpu/wall(s): 0.002 0.003
virt/peak/res/peak(MB): 26.86 26.88 2.57 2.57
```

3.1.9 lcentral.c

This program computes the central value $L(1/2)$ for each Dirichlet L-function character modulo q for each q in the range *qmin* to *qmax*. Usage:

```
> build/examples/lcentral
Computes central values (s = 0.5) of Dirichlet L-functions.

usage: build/examples/lcentral [--quiet] [--check] [--prec <bits>] qmin qmax
```

The first few values:

```
> build/examples/lcentral 1 8
3,2: [0.48086755769682862618122006324 +/- 7.35e-30]
4,3: [0.66769145718960917665869092930 +/- 1.62e-30]
5,2: [0.76374788011728687822451215264 +/- 2.32e-30] + [0.21696476751886069363858659310 +/- 3.
↪06e-30]*I
5,4: [0.23175094750401575588338366176 +/- 2.21e-30]
5,3: [0.76374788011728687822451215264 +/- 2.32e-30] + [-0.21696476751886069363858659310 +/- 3.
↪06e-30]*I
7,3: [0.71394334376831949285993820742 +/- 1.21e-30] + [0.47490218277139938263745243935 +/- 4.
↪52e-30]*I
7,2: [0.31008936259836766059195052534 +/- 5.29e-30] + [-0.07264193137017790524562171245 +/- 5.
↪48e-30]*I
7,6: [1.14658566690370833367712697646 +/- 1.95e-30]
7,4: [0.31008936259836766059195052534 +/- 5.29e-30] + [0.07264193137017790524562171245 +/- 5.
↪48e-30]*I
7,5: [0.71394334376831949285993820742 +/- 1.21e-30] + [-0.47490218277139938263745243935 +/- 4.
↪52e-30]*I
8,5: [0.37369171291254730738158695002 +/- 4.01e-30]
8,3: [1.10042140952554837756713576997 +/- 3.37e-30]
```

(continues on next page)

(continued from previous page)

```
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
```

3.1.10 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
-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^{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 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  (closed path around s = 1)
I8 = int_0^1 sin(1/x) dx  (slow convergence, use -heap and/or -tol)
I9 = int_0^1 x sin(1/x) dx  (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_{-1020}^{-1010} exp(x) dx  (use -tol 0 for relative error)
```

(continues on next page)

(continued from previous page)

```
I14 = int_0^{inf} exp(-x^2) dx   (using domain truncation)
I15 = int_0^1 sech(10(x-0.2))^2 + sech(100(x-0.4))^4 + sech(1000(x-0.6))^6 dx
I16 = int_0^8 (exp(x)-floor(exp(x))) sin(x+exp(x)) dx  (use higher -eval)
I17 = int_0^{inf} sech(x) dx   (using domain truncation)
I18 = int_0^{inf} sech^3(x) dx   (using domain truncation)
I19 = int_0^1 -log(x)/(1+x) dx   (using domain truncation)
I20 = int_0^{inf} x exp(-x)/(1+exp(-x)) dx   (using domain truncation)
I21 = int_C 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} W_0(x) dx
I24 = int_0^pi max(sin(x), cos(x)) dx
I25 = int_{-1}^1 erf(x/sqrt(0.0002)*0.5+1.5)*exp(-x) dx
I26 = int_{-10}^10 Ai(x) dx
I27 = int_0^10 (x-floor(x)-1/2) max(sin(x),cos(x)) dx
I28 = int_{-1-i}^{-1+i} sqrt(x) dx
I29 = int_0^{inf} exp(-x^2+ix) dx   (using domain truncation)
I30 = int_0^{inf} exp(-x) Ai(-x) dx   (using domain truncation)
I31 = int_0^pi 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 = [3.14159265358979323846264338327950288419716939937510582097494459230781640628620
↪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]
```

CHAPTER

FOUR

FLOATING-POINT NUMBERS

Arb uses two custom floating-point types in its implementation of ball arithmetic. The radius of a ball is represented using the type *mag_t* which is unsigned and has a fixed precision. The midpoint is represented using the type *arf_t* which has arbitrary precision.

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

The *mag_t* type holds an unsigned floating-point number with a fixed-precision mantissa (30 bits) and an arbitrary-precision exponent (represented as an *fmpz_t*), suited for representing magnitude bounds. The special values zero and positive infinity are supported, but not NaN.

Operations that involve rounding will always produce a valid upper bound, or a lower bound if the function name has the suffix *lower*. For performance reasons, no attempt is made to compute the best possible bounds: in general, a bound may be several ulps larger/smaller than the optimal bound. Some functions such as *mag_set()* and *mag_mul_2exp_si()* are always exact and therefore do not require separate *lower* versions.

A common mistake is to forget computing a lower bound for the argument of a decreasing function that is meant to be bounded from above, or vice versa. For example, to compute an upper bound for $(x+1)/(y+1)$, the parameter x should initially be an upper bound while y should be a lower bound, and one should do:

```
mag_add_ui(tmp1, x, 1);
mag_add_ui_lower(tmp2, y, 1);
mag_div(res, tmp1, tmp2);
```

For a lower bound of the same expression, x should be a lower bound while y should be an upper bound, and one should do:

```
mag_add_ui_lower(tmp1, x, 1);
mag_add_ui(tmp2, y, 1);
mag_div_lower(res, tmp1, tmp2);
```

Applications requiring floating-point arithmetic with more flexibility (such as correct rounding, or higher precision) should use the *arf_t* type instead. For calculations where a complex alternation between upper and lower bounds is necessary, it may be cleaner to use *arb_t* arithmetic and convert to a *mag_t* bound only in the end.

4.1.1 Types, macros and constants

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

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

4.1.2 Memory management

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

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

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

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

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

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.

4.1.3 Special values

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

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

void **mag_inf**(*mag_t* *res*)
Sets *res* 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 logical negation of *mag_is_inf()*).

4.1.4 Assignment and conversions

void **mag_init_set**(*mag_t* *res*, **const** *mag_t* *x*)
Initializes *res* and sets it to the value of *x*. This operation is always exact.

void **mag_set**(*mag_t* *res*, **const** *mag_t* *x*)
Sets *res* to the value of *x*. This operation is always exact.

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

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

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

void **mag_set_fmpz**(*mag_t* *res*, **const** *fmpz_t* *x*)
Sets *res* to an upper bound for $|x|$. The operation may be inexact even if x is exactly representable.

void **mag_set_d_lower**(*mag_t* *res*, double *x*)

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

void **mag_set_fmpz_lower**(*mag_t* *res*, **const** *fmpz_t* *x*)
Sets *res* to a lower bound for $|x|$. The operation may be inexact even if x is exactly representable.

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

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

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

void **mag_set_d_2exp_fmpz_lower**(*mag_t* *res*, double *x*, **const** *fmpz_t* *y*)

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

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 (warning: not a rigorous bound). The value is clamped between *COEFF_MIN* and *COEFF_MAX*.

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

void **mag_get_fmpq**(*fmpq_t* *res*, **const** *mag_t* *x*)

void **mag_get_fmpz**(*fmpz_t* *res*, **const** *mag_t* *x*)

void **mag_get_fmpz_lower**(*fmpz_t* *res*, **const** *mag_t* *x*)
Sets *res*, respectively, to the exact rational number represented by *x*, the integer exactly representing the ceiling function of *x*, or the integer exactly representing the floor function of *x*.

These functions are unsafe: the user must check in advance that *x* is of reasonable magnitude. 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.

4.1.5 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 res*, **const** *mag_t x*, **const** *mag_t y*)

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

4.1.6 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*.

char ***mag_dump_str**(**const** *mag_t x*)
Allocates a string and writes a binary representation of x to it that can be read by *mag_load_str()*. The returned string needs to be deallocated with *flint_free*.

int **mag_load_str**(*mag_t x*, **const** char **str*)
Parses *str* into x. Returns a nonzero value if *str* is not formatted correctly.

int **mag_dump_file**(FILE **stream*, **const** *mag_t x*)
Writes a binary representation of x to *stream* that can be read by *mag_load_file()*. Returns a nonzero value if the data could not be written.

int **mag_load_file**(*mag_t x*, FILE **stream*)
Reads x from *stream*. Returns a nonzero value if the data is not formatted correctly or the read failed. Note that the data is assumed to be delimited by a whitespace or end-of-file, i.e., when writing multiple values with *mag_dump_file()* make sure to insert a whitespace to separate consecutive values.

4.1.7 Random generation

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

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

4.1.8 Arithmetic

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

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

void **mag_add_lower**(*mag_t res*, **const** *mag_t x*, **const** *mag_t y*)

void **mag_add_ui_lower**(*mag_t res*, **const** *mag_t x*, *ulong y*)
Sets *res* to a lower bound for $x + y$.

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

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

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

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

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

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

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

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

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

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

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

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

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_div**(*mag_t res*, **const** *mag_t x*, **const** *mag_t y*)

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

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

void **mag_div_lower**(*mag_t res*, **const** *mag_t x*, **const** *mag_t y*)
Sets *res* to a lower bound for x/y.

void **mag_inv**(*mag_t res*, **const** *mag_t x*)
Sets *res* to an upper bound for $1/x$.

void **mag_inv_lower**(*mag_t res*, **const** *mag_t x*)
Sets *res* to a lower bound for $1/x$.

4.1.9 Fast, unsafe arithmetic

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

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

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

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

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

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

4.1.10 Powers and logarithms

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

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

void **mag_pow_ui_lower**(*mag_t res*, **const** *mag_t x*, *ulong e*)

void **mag_pow_fmpz_lower**(*mag_t res*, **const** *mag_t x*, **const** *fmpz_t e*)
Sets *res* to a lower bound for x^e. Requires $e \geq 0$.

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

void **mag_sqrt_lower**(*mag_t res*, **const** *mag_t x*)
Sets *res* to a lower bound for $\sqrt{x}$.

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

void **mag_rsqrt_lower**(*mag_t res*, **const** *mag_t x*)
Sets *res* to an lower bound for $1/\sqrt{x}$.

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

void **mag_root**(*mag_t res*, **const** *mag_t x*, *ulong n*)
Sets *res* to an upper bound for $x^{1/n}$.

void **mag_log**(*mag_t res*, **const** *mag_t x*)
Sets *res* to an upper bound for $\log(\max(1, x))$.

void **mag_log_lower**(*mag_t res*, **const** *mag_t x*)
Sets *res* to a lower bound for $\log(\max(1, x))$.

void **mag_neg_log**(*mag_t res*, **const** *mag_t x*)
Sets *res* to an upper bound for $-\log(\min(1, x))$, i.e. an upper bound for $|\log(x)|$ for $x \leq 1$.

void **mag_neg_log_lower**(*mag_t res*, **const** *mag_t x*)
Sets *res* to a lower bound for $-\log(\min(1, x))$, i.e. a lower bound for $|\log(x)|$ for $x \leq 1$.

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

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

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

void **mag_exp_lower**(*mag_t res*, **const** *mag_t x*)
Sets *res* to a lower bound for $\exp(x)$.

void **mag_expinv**(*mag_t res*, **const** *mag_t x*)
Sets *res* to an upper bound for $\exp(-x)$.

void **mag_expinv_lower**(*mag_t res*, **const** *mag_t x*)
Sets *res* to a lower bound for $\exp(-x)$.

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

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

void **mag_binpow_uiui**(*mag_t res*, *ulong m*, *ulong n*)
Sets *res* 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 $\sum_{k=N}^{\infty} x^k$.

4.1.11 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 π.

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 $\operatorname{atan}(x)$.

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

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

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

void **mag_sinh_lower**(*mag_t res*, **const** *mag_t x*)
Sets *res* to an upper or lower bound for $\cosh(x)$ or $\sinh(x)$.

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

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

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

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

The bounding strategy is described in *Algorithms for polylogarithms*. Note: in applications where s in this formula may be real or complex, the user can simply substitute any convenient integer s' such that $s' \le \operatorname{Re}(s)$.

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

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

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

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

A variable of type *arf_t* holds an arbitrary-precision binary floating-point number: that is, a rational number of the form $x \cdot 2^y$ 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). There is currently no support for negative zero, unsigned infinity, or a NaN with a payload.

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 $x \cdot 2^y = m \cdot 2^e$ where $0.5 \le |m| < 1$. The internal representation of an *arf_t* stores the exponent in the latter format.

Except where otherwise noted, functions have the following semantics:

- Functions taking *prec* and *rnd* parameters at the end of the argument list and returning an `int` flag round the result in the output variable to *prec* bits in the direction specified by *rnd*. The return flag is 0 if the result is exact (not rounded) and 1 if the result is inexact (rounded). Correct rounding is guaranteed: the result is the floating-point number obtained by viewing the inputs as exact numbers, in principle carrying out the mathematical 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. In particular, the error is at most 1 ulp with directed rounding modes and 0.5 ulp when rounding to nearest.
- Other functions perform the operation exactly.

Since exponents are bignums, overflow or underflow cannot occur.

4.2.1 Types, macros and constants

type arf_struct

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

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

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

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

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

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

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

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

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

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

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

4.2.2 Memory management

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

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

slong **arf_allocated_bytes**(**const** *arf_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(arf_struct)` to get the size of the object as a whole.

4.2.3 Special values

void **arf_zero**(*arf_t res*)

void **arf_one**(*arf_t res*)

void **arf_pos_inf**(*arf_t res*)

void **arf_neg_inf**(*arf_t res*)

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

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

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

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

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

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

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

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

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

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

4.2.4 Assignment, rounding and conversions

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

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

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

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

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

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

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

void **arf_set_d**(*arf_t res*, double *x*)
Sets *res* to the exact value of x.

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

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

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

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

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

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

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

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

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

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

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

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

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

void **arf_frexp**(*arf_t m*, *fmpz_t e*, **const** *arf_t x*)
Writes x as $m \cdot 2^e$, where $0.5 \le |m| < 1$ if x is a normal value. If x is a special value, copies this to m and sets e to zero. Note: for the inverse operation (*ldexp*), use *arf_mul_2exp_fmpz()*.

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

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

int **arf_get_mpfr**(mpfr_t *res*, **const** *arf_t x*, mpfr_rnd_t *rnd*)
Sets the MPFR variable *res* to the value of x. If the precision of x is too small to allow *res* 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.

If x has an exponent too large or small to fit in the MPFR type, the result overflows to an infinity or underflows to a (signed) zero, and the corresponding MPFR exception flags are set.

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

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

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

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

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

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

The warnings for *arf_get_fmpz()* apply.

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

void **arf_ceil**(*arf_t res*, **const** *arf_t x*)
Sets *res* 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.

4.2.5 Comparisons and bounds

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

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

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

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*, *ulong y*)

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

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

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

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

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

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

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

4.2.6 Magnitude functions

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

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

void **arf_set_mag**(*arf_t* *res*, const *mag_t* *x*)
Sets *res* to x. This operation is exact.

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

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

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

void **arf_mag_add_ulp**(*mag_t* *res*, const *mag_t* *x*, const *arf_t* *y*, *slong* *prec*)
Sets *res* 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* *res*, const *mag_t* *x*, const *arf_t* *y*, *slong* *prec*)
Sets *res* 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.

4.2.7 Shallow assignment

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

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

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

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

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

4.2.8 Random number generation

void **arf_randtest**(*arf_t* *res*, 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* *res*, 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* *res*, 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.

4.2.9 Input and output

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

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

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

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

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

char ***arf_dump_str**(**const** *arf_t* *x*)
Allocates a string and writes a binary representation of x to it that can be read by *arf_load_str()*. The returned string needs to be deallocated with *flint_free*.

int **arf_load_str**(*arf_t* *x*, **const** char **str*)
Parses *str* into x. Returns a nonzero value if *str* is not formatted correctly.

int **arf_dump_file**(FILE **stream*, **const** *arf_t* *x*)
Writes a binary representation of x to *stream* that can be read by *arf_load_file()*. Returns a nonzero value if the data could not be written.

int **arf_load_file**(*arf_t* *x*, FILE **stream*)
Reads x from *stream*. Returns a nonzero value if the data is not formatted correctly or the read failed. Note that the data is assumed to be delimited by a whitespace or end-of-file, i.e.,

when writing multiple values with *arf_dump_file()* make sure to insert a whitespace to separate consecutive values.

4.2.10 Addition and multiplication

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

void **arf_neg**(*arf_t res*, **const** *arf_t x*)
Sets *res* to $-x$ exactly.

int **arf_neg_round**(*arf_t res*, **const** *arf_t x*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* to $-x$.

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

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

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

int **arf_add_fmpz**(*arf_t res*, **const** *arf_t x*, **const** *fmpz_t y*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* to $x + y$.

int **arf_add_fmpz_2exp**(*arf_t res*, **const** *arf_t x*, **const** *fmpz_t y*, **const** *fmpz_t e*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* to $x + y2^e$.

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

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

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

int **arf_sub_fmpz**(*arf_t res*, **const** *arf_t x*, **const** *fmpz_t y*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* to $x - y$.

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

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

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

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

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

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

int **arf_mul_fmpz**(*arf_t res*, **const** *arf_t x*, **const** *fmpz_t y*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* to $x \cdot y$.

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

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

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

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

int **arf_addmul_fmpz**(*arf_t z*, **const** *arf_t x*, **const** *fmpz_t y*, *slong prec*, *arf_rnd_t rnd*)
Performs a fused multiply-add $z = z + x \cdot y$, updating z in-place.

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*)
Performs a fused multiply-subtract $z = z - x \cdot y$, updating z in-place.

int **arf_sosq**(*arf_t res*, **const** *arf_t x*, **const** *arf_t y*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* to $x^2 + y^2$, rounded to *prec* bits in the direction specified by *rnd*.

4.2.11 Summation

int **arf_sum**(*arf_t res*, arf_srcptr *terms*, *slong len*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* 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.

4.2.12 Division

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

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

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

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

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

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

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

int **arf_fmpz_div_fmpz**(*arf_t res*, **const** *fmpz_t x*, **const** *fmpz_t y*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* to 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.

4.2.13 Square roots

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

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

int **arf_sqrt_fmpz**(*arf_t res*, **const** *fmpz_t x*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* to $\sqrt{x}$. The result is NaN if x is negative.

int **arf_rsqrt**(*arf_t res*, **const** *arf_t x*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* to $1/\sqrt{x}$. The result is NaN if x is negative, and $+\infty$ if x is zero.

int **arf_root**(*arf_t res*, **const** *arf_t x*, *ulong k*, *slong prec*, *arf_rnd_t rnd*)
Sets *res* to $x^{1/k}$. 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. Consider working with *arb_root_ui()* for large k instead of using this function directly.

4.2.14 Complex arithmetic

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

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

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

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

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

int **arf_complex_sqr**(*arf_t e*, *arf_t f*, **const** *arf_t a*, **const** *arf_t b*, *slong prec*, *arf_rnd_t rnd*)

Computes the complex square $e + fi = (a + bi)^2$. This function has identical semantics to *arf_complex_mul()* (with $c = a, b = d$), but is faster.

4.2.15 Low-level methods

int **_arf_get_integer_mpn**(*mp_ptr y*, *mp_srcptr xp*, *mp_size_t xn*, *slong exp*)

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

int **_arf_set_mpn_fixed**(*arf_t z*, *mp_srcptr xp*, *mp_size_t xn*, *mp_size_t fixn*, int *negative*, *slong prec*, *arf_rnd_t rnd*)

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

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

Sets z to the integer x, negated if *sgnbit* is 1, rounded to *prec* bits in the direction specified by *rnd*. There are no assumptions on x.

int **_arf_set_round_uiui**(*arf_t z*, *slong *fix*, *mp_limb_t hi*, *mp_limb_t lo*, int *sgnbit*, *slong prec*, *arf_rnd_t rnd*)

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

int **_arf_set_round_mpn**(*arf_t z*, *slong *exp_shift*, *mp_srcptr x*, *mp_size_t xn*, int *sgnbit*, *slong prec*, *arf_rnd_t rnd*)

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

CHAPTER

FIVE

REAL AND COMPLEX NUMBERS

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

5.1 arb.h – real numbers

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

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

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

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

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

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

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

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

5.1.1 Types, macros and constants

type arb_struct

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

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

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

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

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

5.1.2 Memory management

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

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

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

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

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

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 `sizeof(arb_struct)` to get the size of the object as a whole.

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.

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 *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.

5.1.3 Assignment and rounding

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1.4 Assignment of special values

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

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

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

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

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

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

void **arb_zero_pm_one**(*arb_t x*)
Sets x to the interval $[0 \pm 1]$.

void **arb_unit_interval**(*arb_t x*)
Sets x to the interval $[0, 1]$.

5.1.5 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.

char ***arb_dump_str**(**const** *arb_t x*)
Returns a serialized representation of x as a null-terminated ASCII string that can be read by *arb_load_str()*. The format consists of four hexadecimal integers representing the midpoint mantissa, midpoint exponent, radius mantissa and radius exponent (with special values to indicate zero, infinity and NaN values), separated by single spaces. The returned string needs to be deallocated with *flint_free*.

int **arb_load_str**(*arb_t x*, **const** char **str*)
Sets x to the serialized representation given in *str*. Returns a nonzero value if *str* is not formatted correctly (see *arb_dump_str()*).

int **arb_dump_file**(FILE **stream*, **const** *arb_t x*)
Writes a serialized ASCII representation of x to *stream* in a form that can be read by *arb_load_file()*. Returns a nonzero value if the data could not be written.

int **arb_load_file**(*arb_t* *x*, FILE **stream*)
Reads *x* from a serialized ASCII representation in *stream*. Returns a nonzero value if the data is not formatted correctly or the read failed. Note that the data is assumed to be delimited by a whitespace or end-of-file, i.e., when writing multiple values with *arb_dump_file()* make sure to insert a whitespace to separate consecutive values.

It is possible to serialize and deserialize a vector as follows (warning: without error handling):

```
fp = fopen("data.txt", "w");
for (i = 0; i < n; i++)
{
    arb_dump_file(fp, vec + i);
    fprintf(fp, "\n");    // or any whitespace character
}
fclose(fp);

fp = fopen("data.txt", "r");
for (i = 0; i < n; i++)
{
    arb_load_file(vec + i, fp);
}
fclose(fp);
```

5.1.6 Random number generation

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

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

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

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

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

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

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

5.1.7 Radius and interval operations

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

void **arb_get_rad_arb**(*arb_t* *r*, **const** *arb_t* *x*)
Sets *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 *err* to the radius of x (the operation is done in-place).

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

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

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

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

void **arb_nonnegative_part**(*arb_t* *res*, **const** *arb_t* *x*)
Sets *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, *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 *prec* bits. If x contains NaN, the result is NaN.

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

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

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

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

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

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

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

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

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

void **arb_set_interval_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 $[a, b]$. We require that $a \le b$.

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_rel_one_accuracy_bits**(**const** *arb_t* *x*)
Given a ball with midpoint m and radius r, returns an approximation of the relative accuracy of $[\max(1, |m|) \pm r]$ measured in bits.

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

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

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

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

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

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

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

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

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

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

In other words, if this function returns nonzero, applying *arf_set_round()*, or *arf_get_mpfr()*, or *arf_get_d()* to the midpoint of x is guaranteed to return a correctly rounded *arf_t*, *mpfr_t*

(provided that *prec* is the precision of the output variable), or *double* (provided that *prec* is 53). Moreover, *arf_get_mpfr()* is guaranteed to return the correct ternary value according to MPFR semantics.

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

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

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

5.1.8 Comparisons

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

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

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

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

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

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

int **arb_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 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_contains_interior(const** *arb_t x*, **const** *arb_t y*)
Tests if y is contained in the interior of x; that is, contained in x and not touching either endpoint.

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

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

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

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

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

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

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

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

5.1.9 Arithmetic

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

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

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

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

int **arb_sgn_nonzero(const** *arb_t x*)
Returns 1 if x is strictly positive, -1 if x is strictly negative, and 0 if x is zero or a ball containing zero so that its sign is not determined.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1.10 Dot product

void **arb_dot_precise**(*arb_t* *res*, **const** *arb_t* *s*, int *subtract*, *arb_srcptr* *x*, *slong* *xstep*, *arb_srcptr* *y*, *slong* *ystep*, *slong* *len*, *slong* *prec*)

void **arb_dot_simple**(*arb_t* *res*, **const** *arb_t* *s*, int *subtract*, *arb_srcptr* *x*, *slong* *xstep*, *arb_srcptr* *y*, *slong* *ystep*, *slong* *len*, *slong* *prec*)

void **arb_dot**(*arb_t* *res*, **const** *arb_t* *s*, int *subtract*, *arb_srcptr* *x*, *slong* *xstep*, *arb_srcptr* *y*, *slong* *ystep*, *slong* *len*, *slong* *prec*)
Computes the dot product of the vectors x and y, setting *res* to $s + (-1)^{subtract} \sum_{i=0}^{len-1} x_i y_i$.

The initial term s is optional and can be omitted by passing *NULL* (equivalently, $s = 0$). The parameter *subtract* must be 0 or 1. The length *len* is allowed to be negative, which is equivalent to a length of zero. The parameters *xstep* or *ystep* specify a step length for traversing subsequences of the vectors x and y; either can be negative to step in the reverse direction starting from the initial pointer. Aliasing is allowed between *res* and s but not between *res* and the entries of x and y.

The default version determines the optimal precision for each term and performs all internal calculations using mpn arithmetic with minimal overhead. This is the preferred way to compute a dot product; it is generally much faster and more precise than a simple loop.

The *simple* version performs fused multiply-add operations in a simple loop. This can be used for testing purposes and is also used as a fallback by the default version when the exponents are out of range for the optimized code.

The *precise* version computes the dot product exactly up to the final rounding. This can be extremely slow and is only intended for testing.

void **arb_approx_dot**(*arb_t* *res*, **const** *arb_t* *s*, int *subtract*, *arb_srcptr* *x*, *slong* *xstep*, *arb_srcptr* *y*, *slong* *ystep*, *slong* *len*, *slong* *prec*)
Computes an approximate dot product *without error bounds*. The radii of the inputs are ignored (only the midpoints are read) and only the midpoint of the output is written.

5.1.11 Powers and roots

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

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

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

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

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

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

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

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

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

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

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

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

This is evaluated using *mag_log1p()*.

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

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

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

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

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

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

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

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

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

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

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

5.1.12 Exponentials and logarithms

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

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

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

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

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

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

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

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

void **arb_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 $x = m \pm r$, the error is largest at $m + r$, and we have $\exp(m + r) - \exp(m) = \exp(m)(\exp(r) - 1) \le r \exp(m + r)$.

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

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

5.1.13 Trigonometric functions

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

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

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

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

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

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

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

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

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

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

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

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

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

void **arb_sec**(*arb_t* *res*, **const** *arb_t* *x*, *slong* *prec*)
Computes $\sec(x) = 1/\cos(x)$.

void **arb_csc**(*arb_t* *res*, **const** *arb_t* *x*, *slong* *prec*)
Computes $\csc(x) = 1/\sin(x)$.

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

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

5.1.14 Inverse trigonometric functions

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

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

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

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

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

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

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

5.1.15 Hyperbolic functions

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

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

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

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

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

void **arb_sech**(*arb_t res*, **const** *arb_t x*, *slong prec*)
Computes $\operatorname{sech}(x) = 1/\cosh(x)$.

void **arb_csch**(*arb_t res*, **const** *arb_t x*, *slong prec*)
Computes $\operatorname{csch}(x) = 1/\sinh(x)$.

5.1.16 Inverse hyperbolic functions

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

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

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

5.1.17 Constants

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

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

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

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

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

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

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

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

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

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

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

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

5.1.18 Lambert W function

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 $x \geq -1/e$ (taking values in $[-1, +\infty)$) and the W_{-1} branch is real-valued for $-1/e \leq x < 0$ and takes values in $(-\infty, -1]$. 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].

5.1.19 Gamma function and factorials

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1.20 Zeta function

void **arb_zeta_ui_vec_borwein**(*arb_ptr* *z*, *ulong* *start*, *slong* *num*, *ulong* *step*, *slong* *prec*)
Evaluates $\zeta(s)$ at num consecutive integers s beginning with *start* and proceeding in increments of *step*. Uses Borwein's formula ([Bor2000], [GS2003]), implemented to support fast multi-evaluation (but also works well for a single s).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1.21 Bernoulli numbers and polynomials

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The computation uses the generating function for Bernoulli polynomials.

5.1.22 Polylogarithms

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

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

5.1.23 Other special functions

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

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

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

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

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

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

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

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

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

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

void **arb_bell_ui**(*arb_t* *res*, *ulong* *n*, *slong* *prec*)
Sets *res* to the Bell number B_n. If the number is too large to fit exactly in *prec* bits, a numerical approximation is computed efficiently.

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

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

void **arb_euler_number_ui**(*arb_t* *res*, *ulong* *n*, *slong* *prec*)
Sets *res* to the Euler number E_n, which is defined by having the exponential generating function $1/\cosh(x)$.

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

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

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

5.1.24 Internals for computing elementary functions

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

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

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

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

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

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

Computes an approximation of $y = \sum_{k=0}^{N-1} x^k/k!$. Used internally for computing exponentials. The *naive* version uses the forward recurrence, and the *rs* version uses a division-avoiding rectangular splitting scheme.

Requires $N \le 287$, $0 \le x \le 1/16$, and *xn* positive. The input x is a fixed-point number with *xn* fractional limbs, and the output y is a fixed-point number with *xn* fractional limbs plus one extra limb for the integer part of the result.

A bound for the ulp error is written to *error*.

void **_arb_sin_cos_taylor_naive**(*mp_ptr ysin*, *mp_ptr ycos*, *mp_limb_t* *error, *mp_srcptr x*, *mp_size_t xn*, *ulong N*)

void **_arb_sin_cos_taylor_rs**(*mp_ptr ysin*, *mp_ptr ycos*, *mp_limb_t* *error, *mp_srcptr x*, *mp_size_t xn*, *ulong N*, int *sinonly*, int *alternating*)

Computes approximations of $y_s = \sum_{k=0}^{N-1} (-1)^k x^{2k+1}/(2k+1)!$ and $y_c = \sum_{k=0}^{N-1} (-1)^k x^{2k}/(2k)!$. Used internally for computing sines and cosines. The *naive* version uses the forward recurrence, and the *rs* version uses a division-avoiding rectangular splitting scheme.

Requires $N \le 143$, $0 \le x \le 1/16$, and *xn* positive. The input x and outputs *ysin*, *ycos* are fixed-point numbers with *xn* fractional limbs. A bound for the ulp error is written to *error*.

If *sinonly* is 1, only the sine is computed; if *sinonly* is 0 both the sine and cosine are computed. To compute sin and cos, *alternating* should be 1. If *alternating* is 0, the hyperbolic sine is computed (this is currently only intended to be used together with *sinonly*).

int **_arb_get_mpn_fixed_mod_log2**(*mp_ptr w*, *fmpz_t q*, *mp_limb_t* *error, **const** *arf_t x*, *mp_size_t wn*)

Attempts to write $w = x - q\log(2)$ with $0 \le w < \log(2)$, where w is a fixed-point number with *wn* limbs and ulp error *error*. Returns success.

int **_arb_get_mpn_fixed_mod_pi4**(*mp_ptr w*, *fmpz_t q*, int *octant, *mp_limb_t* *error, **const** *arf_t x*, *mp_size_t wn*)

Attempts to write $w = |x| - q\pi/4$ with $0 \le w < \pi/4$, where w is a fixed-point number with *wn* limbs and ulp error *error*. Returns success.

The value of q mod 8 is written to *octant*. The output variable q can be NULL, in which case the full value of q is not stored.

slong **_arb_exp_taylor_bound**(*slong mag*, *slong prec*)

Returns n such that $\left|\sum_{k=n}^{\infty} x^k/k!\right| \le 2^{-\text{prec}}$, assuming $|x| \le 2^{\text{mag}} \le 1/4$.

void **arb_exp_arf_bb**(*arb_t z*, **const** *arf_t x*, *slong prec*, int *m1*)

Computes the exponential function using the bit-burst algorithm. If *m1* is nonzero, the exponential function minus one is computed accurately.

Aborts if x is extremely small or large (where another algorithm should be used).

For large x, repeated halving is used. In fact, we always do argument reduction until $|x|$ is smaller than about 2^{-d} where $d \approx 16$ to speed up convergence. If $|x| \approx 2^m$, we thus need about $m + d$ squarings.

Computing log(2) costs roughly 100-200 multiplications, so is not usually worth the effort at very high precision. However, this function could be improved by using log(2) based reduction at precision low enough that the value can be assumed to be cached.

void **_arb_exp_sum_bs_simple**(*fmpz_t T*, *fmpz_t Q*, *flint_bitcnt_t *Qexp*, **const** *fmpz_t x*, *flint_bitcnt_t r*, *slong N*)

void **_arb_exp_sum_bs_powtab**(*fmpz_t T*, *fmpz_t Q*, *flint_bitcnt_t *Qexp*, **const** *fmpz_t x*, *flint_bitcnt_t r*, *slong N*)

Computes T, Q and $Qexp$ such that $T/(Q2^{\mathrm{Qexp}}) = \sum_{k=1}^{N} (x/2^r)^k/k!$ using binary splitting. Note that the sum is taken to N inclusive and omits the constant term.

The *powtab* version precomputes a table of powers of x, resulting in slightly higher memory usage but better speed. For best efficiency, N should have many trailing zero bits.

void **arb_exp_arf_rs_generic**(*arb_t res*, **const** *arf_t x*, *slong prec*, int *minus_one*)

Computes the exponential function using a generic version of the rectangular splitting strategy, intended for intermediate precision.

void **_arb_atan_sum_bs_simple**(*fmpz_t T*, *fmpz_t Q*, *flint_bitcnt_t *Qexp*, **const** *fmpz_t x*, *flint_bitcnt_t r*, *slong N*)

void **_arb_atan_sum_bs_powtab**(*fmpz_t T*, *fmpz_t Q*, *flint_bitcnt_t *Qexp*, **const** *fmpz_t x*, *flint_bitcnt_t r*, *slong N*)

Computes T, Q and $Qexp$ such that $T/(Q2^{\mathrm{Qexp}}) = \sum_{k=1}^{N} (-1)^k (x/2^r)^{2k}/(2k+1)$ using binary splitting. Note that the sum is taken to N inclusive, omits the linear term, and requires a final multiplication by $(x/2^r)$ to give the true series for atan.

The *powtab* version precomputes a table of powers of x, resulting in slightly higher memory usage but better speed. For best efficiency, N should have many trailing zero bits.

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

Computes the arctangent of x. Initially, the argument-halving formula

$$\operatorname{atan}(x) = 2 \operatorname{atan}\left(\frac{x}{1 + \sqrt{1 + x^2}}\right)$$

is applied up to 8 times to get a small argument. Then a version of the bit-burst algorithm is used. The functional equation

$$\operatorname{atan}(x) = \operatorname{atan}(p/q) + \operatorname{atan}(w), \quad w = \frac{qx - p}{px + q}, \quad p = \lfloor qx \rfloor$$

is applied repeatedly instead of integrating a differential equation for the arctangent, as this appears to be more efficient.

void **arb_sin_cos_arf_generic**(*arb_t s*, *arb_t c*, **const** *arf_t x*, *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 π 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 **arb_sin_cos_arf_bb**(*arb_t s*, *arb_t c*, **const** *arf_t x*, *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).

void **arb_sin_cos_wide**(*arb_t s*, *arb_t c*, **const** *arb_t x*, *slong prec*)
Computes an accurate enclosure (with both endpoints optimal to within about 2^{-30} as afforded by the radius format) of the range of sine and cosine on a given wide interval. The computation is done by evaluating the sine and cosine at the interval endpoints and determining whether peaks of -1 or 1 occur between the endpoints. The interval is then converted back to a ball.

The internal computations are done with doubles, using a simple floating-point algorithm to approximate the sine and cosine. It is easy to see that the cumulative errors in this algorithm add up to less than 2^{-30}, with the dominant source of error being a single approximate reduction by $\pi/2$. This reduction is done safely using doubles up to a magnitude of about 2^{20}. For larger arguments, a slower reduction using *arb_t* arithmetic is done as a preprocessing step.

void **arb_sin_cos_generic**(*arb_t s*, *arb_t c*, **const** *arb_t x*, *slong prec*)
Computes the sine and cosine of x by taking care of various special cases and computing the propagated error before calling *arb_sin_cos_arf_generic()*. This is used as a fallback inside *arb_sin_cos()* to take care of all cases without a fast path in that function.

5.1.25 Vector functions

void **_arb_vec_zero**(*arb_ptr vec*, *slong n*)
Sets all entries in *vec* to zero.

int **_arb_vec_is_zero**(*arb_srcptr vec*, *slong len*)
Returns nonzero iff all entries in x are zero.

int **_arb_vec_is_finite**(*arb_srcptr x*, *slong len*)
Returns nonzero iff all entries in x certainly are finite.

void **_arb_vec_set**(*arb_ptr res*, *arb_srcptr vec*, *slong len*)
Sets *res* to a copy of *vec*.

void **_arb_vec_set_round**(*arb_ptr res*, *arb_srcptr vec*, *slong len*, *slong prec*)
Sets *res* to a copy of *vec*, rounding each entry to *prec* bits.

void **_arb_vec_swap**(*arb_ptr vec1*, *arb_ptr vec2*, *slong len*)
Swaps the entries of *vec1* and *vec2*.

void **_arb_vec_neg**(*arb_ptr B*, *arb_srcptr A*, *slong n*)

void **_arb_vec_sub**(*arb_ptr C*, *arb_srcptr A*, *arb_srcptr B*, *slong n*, *slong prec*)

void **_arb_vec_add**(*arb_ptr C*, *arb_srcptr A*, *arb_srcptr B*, *slong n*, *slong prec*)

void **_arb_vec_scalar_mul**(*arb_ptr res*, *arb_srcptr vec*, *slong len*, **const** *arb_t c*, *slong prec*)

void **_arb_vec_scalar_div**(*arb_ptr res*, *arb_srcptr vec*, *slong len*, **const** *arb_t c*, *slong prec*)

void **_arb_vec_scalar_mul_fmpz**(*arb_ptr res*, *arb_srcptr vec*, *slong len*, **const** *fmpz_t c*, *slong prec*)

void **_arb_vec_scalar_mul_2exp_si**(*arb_ptr res*, *arb_srcptr src*, *slong len*, *slong c*)

void **_arb_vec_scalar_addmul**(*arb_ptr res*, *arb_srcptr vec*, *slong len*, **const** *arb_t c*, *slong prec*)
Performs the respective scalar operation elementwise.

void **_arb_vec_get_mag**(*mag_t bound*, *arb_srcptr vec*, *slong len*, *slong prec*)
Sets *bound* to an upper bound for the entries in *vec*.

slong **_arb_vec_bits**(*arb_srcptr x*, *slong len*)
Returns the maximum of *arb_bits()* for all entries in *vec*.

void **_arb_vec_set_powers**(*arb_ptr xs*, **const** *arb_t x*, *slong len*, *slong prec*)
Sets *xs* to the powers $1, x, x^2, \ldots, x^{len-1}$.

void **_arb_vec_add_error_arf_vec**(*arb_ptr res*, arf_srcptr *err*, *slong len*)

void **_arb_vec_add_error_mag_vec**(*arb_ptr res*, mag_srcptr *err*, *slong len*)
Adds the magnitude of each entry in *err* to the radius of the corresponding entry in *res*.

void **_arb_vec_indeterminate**(*arb_ptr vec*, *slong len*)
Applies *arb_indeterminate()* elementwise.

void **_arb_vec_trim**(*arb_ptr res*, *arb_srcptr vec*, *slong len*)
Applies *arb_trim()* elementwise.

int **_arb_vec_get_unique_fmpz_vec**(*fmpz* *res, *arb_srcptr vec*, *slong len*)
Calls *arb_get_unique_fmpz()* elementwise and returns nonzero if all entries can be rounded uniquely to integers. If any entry in *vec* cannot be rounded uniquely to an integer, returns zero.

5.2 acb.h – complex numbers

An *acb_t* represents a complex number with error bounds. An *acb_t* consists of a pair of real number balls of type *arb_struct*, representing the real and imaginary part with separate error bounds.

An *acb_t* thus represents a rectangle $[m_1 - r_1, m_1 + r_1] + [m_2 - r_2, m_2 + r_2]i$ in the complex plane. This is used instead of a disk or square representation (consisting of a complex floating-point midpoint with a single radius), since it allows implementing many operations more conveniently by splitting into ball operations on the real and imaginary parts. It also allows tracking when complex numbers have an exact (for example exactly zero) real part and an inexact imaginary part, or vice versa.

The interface for the *acb_t* type is slightly less developed than that for the *arb_t* type. In many cases, the user can easily perform missing operations by directly manipulating the real and imaginary parts.

5.2.1 Types, macros and constants

type acb_struct

type acb_t
An *acb_struct* consists of a pair of *arb_struct*:s. An *acb_t* is defined as an array of length one of type *acb_struct*, permitting an *acb_t* to be passed by reference.

type acb_ptr
Alias for `acb_struct *`, used for vectors of numbers.

type acb_srcptr
Alias for `const acb_struct *`, used for vectors of numbers when passed as constant input to functions.

acb_realref(*x*)
Macro returning a pointer to the real part of *x* as an *arb_t*.

acb_imagref(*x*)
Macro returning a pointer to the imaginary part of *x* as an *arb_t*.

5.2.2 Memory management

void **acb_init**(*acb_t x*)
Initializes the variable *x* for use, and sets its value to zero.

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

acb_ptr **_acb_vec_init**(*slong n*)
Returns a pointer to an array of *n* initialized *acb_struct*:s.

void **_acb_vec_clear**(*acb_ptr v*, *slong n*)
Clears an array of *n* initialized *acb_struct*:s.

slong **acb_allocated_bytes**(**const** *acb_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_struct)` to get the size of the object as a whole.

slong **_acb_vec_allocated_bytes**(*acb_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.

double **_acb_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. See comments for *_arb_vec_estimate_allocated_bytes()*.

5.2.3 Basic manipulation

void **acb_zero**(*acb_t z*)

void **acb_one**(*acb_t z*)

void **acb_onei**(*acb_t z*)
Sets z respectively to 0, 1, $i = \sqrt{-1}$.

void **acb_set**(*acb_t z*, **const** *acb_t x*)

void **acb_set_ui**(*acb_t z*, *slong x*)

void **acb_set_si**(*acb_t z*, *slong x*)

void **acb_set_d**(*acb_t z*, double *x*)

void **acb_set_fmpz**(*acb_t z*, **const** *fmpz_t x*)

void **acb_set_arb**(*acb_t z*, **const** *arb_t c*)
Sets z to the value of x.

void **acb_set_si_si**(*acb_t z*, *slong x*, *slong y*)

void **acb_set_d_d**(*acb_t z*, double *x*, double *y*)

void **acb_set_fmpz_fmpz**(*acb_t z*, **const** *fmpz_t x*, **const** *fmpz_t y*)

void **acb_set_arb_arb**(*acb_t z*, **const** *arb_t x*, **const** *arb_t y*)
Sets the real and imaginary part of z to the values x and y respectively

void **acb_set_fmpq**(*acb_t z*, **const** *fmpq_t x*, *slong prec*)

void **acb_set_round**(*acb_t z*, **const** *acb_t x*, *slong prec*)

void **acb_set_round_fmpz**(*acb_t z*, **const** *fmpz_t x*, *slong prec*)

void **acb_set_round_arb**(*acb_t z*, **const** *arb_t x*, *slong prec*)
Sets z to x, rounded to *prec* bits.

void **acb_swap**(*acb_t z*, *acb_t x*)
Swaps z and x efficiently.

void **acb_add_error_mag**(*acb_t x*, **const** *mag_t err*)
Adds *err* to the error bounds of both the real and imaginary parts of x, modifying x in-place.

void **acb_get_mid**(*acb_t m*, **const** *acb_t x*)
Sets m to the midpoint of x.

5.2.4 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.

5.2.5 Random number generation

void **acb_randtest**(*acb_t z*, flint_rand_t *state*, *slong prec*, *slong mag_bits*)
Generates a random complex number by generating separate random real and imaginary parts.

void **acb_randtest_special**(*acb_t z*, flint_rand_t *state*, *slong prec*, *slong mag_bits*)
Generates a random complex number by generating separate random real and imaginary parts. Also generates NaNs and infinities.

void **acb_randtest_precise**(*acb_t z*, flint_rand_t *state*, *slong prec*, *slong mag_bits*)
Generates a random complex number with precise real and imaginary parts.

void **acb_randtest_param**(*acb_t z*, flint_rand_t *state*, *slong prec*, *slong mag_bits*)
Generates a random complex number, with very high probability of generating integers and half-integers.

5.2.6 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 *acb_overlaps()* or *acb_contains()*, depending on the circumstance.

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

int **acb_eq**(**const** *acb_t* *x*, **const** *acb_t* *y*)
Returns nonzero iff x and y are certainly equal, as determined by testing that *arb_eq()* holds for both the real and imaginary parts.

int **acb_ne**(**const** *acb_t* *x*, **const** *acb_t* *y*)
Returns nonzero iff x and y are certainly not equal, as determined by testing that *arb_ne()* holds for either the real or imaginary parts.

int **acb_overlaps**(**const** *acb_t* *x*, **const** *acb_t* *y*)
Returns nonzero iff x and y have some point in common.

void **acb_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 *prec* bits.

void **acb_get_abs_lbound_arf**(*arf_t* *u*, **const** *acb_t* *z*, *slong* *prec*)
Sets u to a lower bound for the absolute value of z, computed using a working precision of *prec* bits.

void **acb_get_rad_ubound_arf**(*arf_t* *u*, **const** *acb_t* *z*, *slong* *prec*)
Sets u to an upper bound for the error radius of z (the value is currently not computed tightly).

void **acb_get_mag**(*mag_t* *u*, **const** *acb_t* *x*)
Sets u to an upper bound for the absolute value of x.

void **acb_get_mag_lower**(*mag_t* *u*, **const** *acb_t* *x*)
Sets u to a lower bound for the absolute value of x.

int **acb_contains_fmpq**(**const** *acb_t* *x*, **const** *fmpq_t* *y*)

int **acb_contains_fmpz**(**const** *acb_t* *x*, **const** *fmpz_t* *y*)

int **acb_contains**(**const** *acb_t* *x*, **const** *acb_t* *y*)
Returns nonzero iff y is contained in x.

int **acb_contains_zero**(**const** *acb_t* *x*)
Returns nonzero iff zero is contained in x.

int **acb_contains_int**(**const** *acb_t* *x*)
Returns nonzero iff the complex interval represented by x contains an integer.

int **acb_contains_interior**(**const** *acb_t* *x*, **const** *acb_t* *y*)
Tests if y is contained in the interior of x. This predicate always evaluates to false if x and y are both real-valued, since an imaginary part of 0 is not considered contained in the interior of the point interval 0. More generally, the same problem occurs for intervals with an exact real or imaginary part. Such intervals must be handled specially by the user where a different interpretation is intended.

slong **acb_rel_error_bits**(**const** *acb_t* *x*)
Returns the effective relative error of x measured in bits. This is computed as if calling *arb_rel_error_bits()* on the real ball whose midpoint is the larger out of the real and imaginary midpoints of x, and whose radius is the larger out of the real and imaginary radiuses of x.

slong **acb_rel_accuracy_bits(const** *acb_t x***)**
Returns the effective relative accuracy of x measured in bits, equal to the negative of the return value from *acb_rel_error_bits()*.

slong **acb_rel_one_accuracy_bits(const** *acb_t x***)**
Given a ball with midpoint m and radius r, returns an approximation of the relative accuracy of $[\max(1, |m|) \pm r]$ measured in bits.

slong **acb_bits(const** *acb_t x***)**
Returns the maximum of *arb_bits* applied to the real and imaginary parts of x, i.e. the minimum precision sufficient to represent x exactly.

void **acb_indeterminate(***acb_t x***)**
Sets x to $[\text{NaN} \pm\infty] + [\text{NaN} \pm\infty]i$, representing an indeterminate result.

void **acb_trim(***acb_t y*, **const** *acb_t x***)**
Sets y to a a copy of x with both the real and imaginary parts trimmed (see *arb_trim()*).

int **acb_is_real(const** *acb_t x***)**
Returns nonzero iff the imaginary part of x is zero. It does not test whether the real part of x also is finite.

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

5.2.7 Complex parts

void **acb_get_real(***arb_t re*, **const** *acb_t z***)**
Sets *re* to the real part of z.

void **acb_get_imag(***arb_t im*, **const** *acb_t z***)**
Sets *im* to the imaginary part of z.

void **acb_arg(***arb_t r*, **const** *acb_t z*, *slong prec***)**
Sets r to a real interval containing the complex argument (phase) of z. We define the complex argument have a discontinuity on $(-\infty, 0]$, with the special value $\arg(0) = 0$, and $\arg(a + 0i) = \pi$ for $a < 0$. Equivalently, if $z = a + bi$, the argument is given by $\operatorname{atan2}(b, a)$ (see *arb_atan2()*).

void **acb_abs(***arb_t r*, **const** *acb_t z*, *slong prec***)**
Sets r to the absolute value of z.

void **acb_sgn(***acb_t r*, **const** *acb_t z*, *slong prec***)**
Sets r to the complex sign of z, defined as 0 if z is exactly zero and the projection onto the unit circle $z/|z| = \exp(i \arg(z))$ otherwise.

void **acb_csgn(***arb_t r*, **const** *acb_t z***)**
Sets r to the extension of the real sign function taking the value 1 for z strictly in the right half plane, -1 for z strictly in the left half plane, and the sign of the imaginary part when z is on the imaginary axis. Equivalently, $\operatorname{csgn}(z) = z/\sqrt{z^2}$ except that the value is 0 when z is exactly zero.

5.2.8 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 **acb_add**(*acb_t z*, **const** *acb_t x*, **const** *acb_t y*, *slong prec*)
Sets z to the sum of x and y.

void **acb_sub_ui**(*acb_t z*, **const** *acb_t x*, *ulong y*, *slong prec*)

void **acb_sub_si**(*acb_t z*, **const** *acb_t x*, *slong y*, *slong prec*)

void **acb_sub_fmpz**(*acb_t z*, **const** *acb_t x*, **const** *fmpz_t y*, *slong prec*)

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

void **acb_sub**(*acb_t z*, **const** *acb_t x*, **const** *acb_t y*, *slong prec*)
Sets z to the difference of x and y.

void **acb_mul_onei**(*acb_t z*, **const** *acb_t x*)
Sets z to x multiplied by the imaginary unit.

void **acb_div_onei**(*acb_t z*, **const** *acb_t x*)
Sets z to x divided by the imaginary unit.

void **acb_mul_ui**(*acb_t z*, **const** *acb_t x*, *ulong y*, *slong prec*)

void **acb_mul_si**(*acb_t z*, **const** *acb_t x*, *slong y*, *slong prec*)

void **acb_mul_fmpz**(*acb_t z*, **const** *acb_t x*, **const** *fmpz_t y*, *slong prec*)

void **acb_mul_arb**(*acb_t z*, **const** *acb_t x*, **const** *arb_t y*, *slong prec*)
Sets z to the product of x and y.

void **acb_mul**(*acb_t z*, **const** *acb_t x*, **const** *acb_t y*, *slong prec*)
Sets z to the product of x and y. If at least one part of x or y is zero, the operations is reduced to two real multiplications. If x and y are the same pointers, they are assumed to represent the same mathematical quantity and the squaring formula is used.

void **acb_mul_2exp_si**(*acb_t z*, **const** *acb_t x*, *slong e*)

void **acb_mul_2exp_fmpz**(*acb_t z*, **const** *acb_t x*, **const** *fmpz_t e*)
Sets z to x multiplied by 2^e, without rounding.

void **acb_sqr**(*acb_t z*, **const** *acb_t x*, *slong prec*)
Sets z to x squared.

void **acb_cube**(*acb_t z*, **const** *acb_t x*, *slong prec*)
Sets z to x cubed, computed efficiently using two real squarings, two real multiplications, and scalar operations.

void **acb_addmul**(*acb_t z*, **const** *acb_t x*, **const** *acb_t y*, *slong prec*)

void **acb_addmul_ui**(*acb_t z*, **const** *acb_t x*, *ulong y*, *slong prec*)

void **acb_addmul_si**(*acb_t z*, **const** *acb_t x*, *slong y*, *slong prec*)

void **acb_addmul_fmpz**(*acb_t z*, **const** *acb_t x*, **const** *fmpz_t y*, *slong prec*)

void **acb_addmul_arb**(*acb_t z*, **const** *acb_t x*, **const** *arb_t y*, *slong prec*)
Sets z to z plus the product of x and y.

void **acb_submul**(*acb_t z*, **const** *acb_t x*, **const** *acb_t y*, *slong prec*)

void **acb_submul_ui**(*acb_t z*, **const** *acb_t x*, *ulong y*, *slong prec*)

void **acb_submul_si**(*acb_t z*, **const** *acb_t x*, *slong y*, *slong prec*)

void **acb_submul_fmpz**(*acb_t z*, **const** *acb_t x*, **const** *fmpz_t y*, *slong prec*)

void **acb_submul_arb**(*acb_t z*, **const** *acb_t x*, **const** *arb_t y*, *slong prec*)
Sets z to z minus the product of x and y.

void **acb_inv**(*acb_t z*, **const** *acb_t x*, *slong prec*)
Sets z to the multiplicative inverse of x.

void **acb_div_ui**(*acb_t z*, **const** *acb_t x*, *ulong y*, *slong prec*)

void **acb_div_si**(*acb_t z*, **const** *acb_t x*, *slong y*, *slong prec*)

void **acb_div_fmpz**(*acb_t z*, **const** *acb_t x*, **const** *fmpz_t y*, *slong prec*)

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

void **acb_div**(*acb_t z*, **const** *acb_t x*, **const** *acb_t y*, *slong prec*)
Sets z to the quotient of x and y.

5.2.9 Dot product

void **acb_dot_precise**(*acb_t res*, **const** *acb_t s*, int *subtract*, *acb_srcptr x*, *slong xstep*, *acb_srcptr y*, *slong ystep*, *slong len*, *slong prec*)

void **acb_dot_simple**(*acb_t res*, **const** *acb_t s*, int *subtract*, *acb_srcptr x*, *slong xstep*, *acb_srcptr y*, *slong ystep*, *slong len*, *slong prec*)

void **acb_dot**(*acb_t res*, **const** *acb_t s*, int *subtract*, *acb_srcptr x*, *slong xstep*, *acb_srcptr y*, *slong ystep*, *slong len*, *slong prec*)
Computes the dot product of the vectors x and y, setting *res* to $s + (-1)^{subtract} \sum_{i=0}^{len-1} x_i y_i$.

The initial term s is optional and can be omitted by passing *NULL* (equivalently, $s = 0$). The parameter *subtract* must be 0 or 1. The length *len* is allowed to be negative, which is equivalent to a length of zero. The parameters *xstep* or *ystep* specify a step length for traversing subsequences of the vectors x and y; either can be negative to step in the reverse direction starting from the initial pointer. Aliasing is allowed between *res* and s but not between *res* and the entries of x and y.

The default version determines the optimal precision for each term and performs all internal calculations using mpn arithmetic with minimal overhead. This is the preferred way to compute a dot product; it is generally much faster and more precise than a simple loop.

The *simple* version performs fused multiply-add operations in a simple loop. This can be used for testing purposes and is also used as a fallback by the default version when the exponents are out of range for the optimized code.

The *precise* version computes the dot product exactly up to the final rounding. This can be extremely slow and is only intended for testing.

void **acb_approx_dot**(*acb_t res*, **const** *acb_t s*, int *subtract*, *acb_srcptr x*, *slong xstep*, *acb_srcptr y*, *slong ystep*, *slong len*, *slong prec*)
Computes an approximate dot product *without error bounds*. The radii of the inputs are ignored (only the midpoints are read) and only the midpoint of the output is written.

5.2.10 Mathematical constants

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

5.2.11 Powers and roots

void **acb_sqrt**(*acb_t r*, **const** *acb_t z*, *slong prec*)
Sets r to the square root of z. If either the real or imaginary part is exactly zero, only a single real square root is needed. Generally, we use the formula $\sqrt{a+bi} = u/2 + ib/u, u = \sqrt{2(|a+bi|+a)}$, requiring two real square root extractions.

void **acb_sqrt_analytic**(*acb_t r*, **const** *acb_t z*, int *analytic*, *slong prec*)
Computes the square root. If *analytic* is set, gives a NaN-containing result if z touches the branch cut.

void **acb_rsqrt**(*acb_t r*, **const** *acb_t z*, *slong prec*)
Sets r to the reciprocal square root of z. If either the real or imaginary part is exactly zero, only a single real reciprocal square root is needed. Generally, we use the formula $1/\sqrt{a+bi} = ((a+r)-bi)/v, r = |a+bi|, v = \sqrt{r|a+bi+r|^2}$, requiring one real square root and one real reciprocal square root.

void **acb_rsqrt_analytic**(*acb_t r*, **const** *acb_t z*, int *analytic*, *slong prec*)
Computes the reciprocal square root. If *analytic* is set, gives a NaN-containing result if z touches the branch cut.

void **acb_quadratic_roots_fmpz**(*acb_t r1*, *acb_t r2*, **const** *fmpz_t a*, **const** *fmpz_t b*, **const** *fmpz_t c*, *slong prec*)
Sets *r1* and *r2* to the roots of the quadratic polynomial $ax^2 + bx + c$. Requires that a is nonzero. This function is implemented so that both roots are computed accurately even when direct use of the quadratic formula would lose accuracy.

void **acb_root_ui**(*acb_t r*, **const** *acb_t z*, *ulong k*, *slong prec*)
Sets r to the principal k-th root of z.

void **acb_pow_fmpz**(*acb_t y*, **const** *acb_t b*, **const** *fmpz_t e*, *slong prec*)

void **acb_pow_ui**(*acb_t y*, **const** *acb_t b*, *ulong e*, *slong prec*)

void **acb_pow_si**(*acb_t y*, **const** *acb_t b*, *slong e*, *slong prec*)
Sets $y = b^e$ using binary exponentiation (with an initial division if $e < 0$). Note that these functions can get slow if the exponent is extremely large (in such cases *acb_pow()* may be superior).

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

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

void **acb_pow_analytic**(*acb_t r*, **const** *acb_t x*, **const** *acb_t y*, int *analytic*, *slong prec*)
Computes the power x^y. If *analytic* is set, gives a NaN-containing result if x touches the branch cut (unless y is an integer).

void **acb_unit_root**(*acb_t res*, *ulong order*, *slong prec*)
Sets *res* to $\exp(\frac{2i\pi}{\text{order}})$ to precision *prec*.

5.2.12 Exponentials and logarithms

void **acb_exp**(*acb_t y*, **const** *acb_t z*, *slong prec*)
Sets y to the exponential function of z, computed as $\exp(a+bi) = \exp(a)\,(\cos(b) + \sin(b)i)$.

void **acb_exp_pi_i**(*acb_t y*, **const** *acb_t z*, *slong prec*)
Sets y to $\exp(\pi i z)$.

void **acb_exp_invexp**(*acb_t s*, *acb_t t*, **const** *acb_t z*, *slong prec*)
Sets $v = \exp(z)$ and $w = \exp(-z)$.

void **acb_expm1**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes $\exp(z) - 1$, using an accurate method when $z \approx 0$.

void **acb_log**(*acb_t* *y*, **const** *acb_t* *z*, *slong* *prec*)
Sets y to the principal branch of the natural logarithm of z, computed as $\log(a+bi) = \frac{1}{2}\log(a^2+b^2) + i\arg(a+bi)$.

void **acb_log_analytic**(*acb_t* *r*, **const** *acb_t* *z*, int *analytic*, *slong* *prec*)
Computes the natural logarithm. If *analytic* is set, gives a NaN-containing result if z touches the branch cut.

void **acb_log1p**(*acb_t* *z*, **const** *acb_t* *x*, *slong* *prec*)
Sets $z = \log(1+x)$, computed accurately when $x \approx 0$.

5.2.13 Trigonometric functions

void **acb_sin**(*acb_t* *s*, **const** *acb_t* *z*, *slong* *prec*)

void **acb_cos**(*acb_t* *c*, **const** *acb_t* *z*, *slong* *prec*)

void **acb_sin_cos**(*acb_t* *s*, *acb_t* *c*, **const** *acb_t* *z*, *slong* *prec*)
Sets $s = \sin(z)$, $c = \cos(z)$, evaluated as $\sin(a+bi) = \sin(a)\cosh(b) + i\cos(a)\sinh(b)$, $\cos(a+bi) = \cos(a)\cosh(b) - i\sin(a)\sinh(b)$.

void **acb_tan**(*acb_t* *s*, **const** *acb_t* *z*, *slong* *prec*)
Sets $s = \tan(z) = \sin(z)/\cos(z)$. For large imaginary parts, the function is evaluated in a numerically stable way as $\pm i$ plus a decreasing exponential factor.

void **acb_cot**(*acb_t* *s*, **const** *acb_t* *z*, *slong* *prec*)
Sets $s = \cot(z) = \cos(z)/\sin(z)$. For large imaginary parts, the function is evaluated in a numerically stable way as $\pm i$ plus a decreasing exponential factor.

void **acb_sin_pi**(*acb_t* *s*, **const** *acb_t* *z*, *slong* *prec*)

void **acb_cos_pi**(*acb_t* *s*, **const** *acb_t* *z*, *slong* *prec*)

void **acb_sin_cos_pi**(*acb_t* *s*, *acb_t* *c*, **const** *acb_t* *z*, *slong* *prec*)
Sets $s = \sin(\pi z)$, $c = \cos(\pi z)$, evaluating the trigonometric factors of the real and imaginary part accurately via *arb_sin_cos_pi()*.

void **acb_tan_pi**(*acb_t* *s*, **const** *acb_t* *z*, *slong* *prec*)
Sets $s = \tan(\pi z)$. Uses the same algorithm as *acb_tan()*, but evaluates the sine and cosine accurately via *arb_sin_cos_pi()*.

void **acb_cot_pi**(*acb_t* *s*, **const** *acb_t* *z*, *slong* *prec*)
Sets $s = \cot(\pi z)$. Uses the same algorithm as *acb_cot()*, but evaluates the sine and cosine accurately via *arb_sin_cos_pi()*.

void **acb_sec**(*acb_t* *res*, **const** *acb_t* *z*, *slong* *prec*)
Computes $\sec(z) = 1/\cos(z)$.

void **acb_csc**(*acb_t* *res*, **const** *acb_t* *z*, *slong* *prec*)
Computes $\csc(x) = 1/\sin(z)$.

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

void **acb_sinc_pi**(*acb_t* *s*, **const** *acb_t* *z*, *slong* *prec*)
Sets $s = \operatorname{sinc}(\pi x) = \sin(\pi z)/(\pi z)$.

5.2.14 Inverse trigonometric functions

void **acb_asin**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{asin}(z) = -i \log(iz + \sqrt{1 - z^2})$.

void **acb_acos**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{acos}(z) = \frac{1}{2}\pi - \operatorname{asin}(z)$.

void **acb_atan**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{atan}(z) = \frac{1}{2}i(\log(1 - iz) - \log(1 + iz))$.

5.2.15 Hyperbolic functions

void **acb_sinh**(*acb_t s*, **const** *acb_t z*, *slong prec*)

void **acb_cosh**(*acb_t c*, **const** *acb_t z*, *slong prec*)

void **acb_sinh_cosh**(*acb_t s*, *acb_t c*, **const** *acb_t z*, *slong prec*)

void **acb_tanh**(*acb_t s*, **const** *acb_t z*, *slong prec*)

void **acb_coth**(*acb_t s*, **const** *acb_t z*, *slong prec*)
Respectively computes $\sinh(z) = -i \sin(iz)$, $\cosh(z) = \cos(iz)$, $\tanh(z) = -i \tan(iz)$, $\coth(z) = i \cot(iz)$.

void **acb_sech**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes $\operatorname{sech}(z) = 1/\cosh(z)$.

void **acb_csch**(*acb_t res*, **const** *arb_t z*, *slong prec*)
Computes $\operatorname{csch}(z) = 1/\sinh(z)$.

5.2.16 Inverse hyperbolic functions

void **acb_asinh**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{asinh}(z) = -i \operatorname{asin}(iz)$.

void **acb_acosh**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{acosh}(z) = \log(z + \sqrt{z+1}\sqrt{z-1})$.

void **acb_atanh**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Sets *res* to $\operatorname{atanh}(z) = -i \operatorname{atan}(iz)$.

5.2.17 Lambert W function

void **acb_lambertw_asymp**(*acb_t res*, **const** *acb_t z*, **const** *fmpz_t k*, *slong L*, *slong M*, *slong prec*)
Sets *res* to the Lambert W function $W_k(z)$ computed using L and 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 **acb_lambertw_check_branch**(**const** *acb_t w*, **const** *fmpz_t k*, *slong prec*)
Tests if 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.

void **acb_lambertw_bound_deriv**(*mag_t res*, **const** *acb_t z*, **const** *acb_t ez1*, **const** *fmpz_t k*)
Sets *res* to an upper bound for $|W_k'(z)|$. The input *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 **acb_lambertw**(*acb_t res*, **const** *acb_t z*, **const** *fmpz_t k*, int *flags*, *slong prec*)
Sets *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 *flags* is nonzero, nonstandard branch cuts are used.

If *flags* is set to *ACB_LAMBERTW_LEFT*, computes $W_{\text{left}|k}(z)$ 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 *flags* is set to *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].

5.2.18 Rising factorials

void **acb_rising_ui_bs**(*acb_t z*, **const** *acb_t x*, *ulong n*, *slong prec*)

void **acb_rising_ui_rs**(*acb_t z*, **const** *acb_t x*, *ulong n*, *ulong step*, *slong prec*)

void **acb_rising_ui_rec**(*acb_t z*, **const** *acb_t x*, *ulong n*, *slong prec*)

void **acb_rising_ui**(*acb_t z*, **const** *acb_t x*, *ulong n*, *slong prec*)

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

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

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

void **acb_rising2_ui_bs**(*acb_t u*, *acb_t v*, **const** *acb_t x*, *ulong n*, *slong prec*)

void **acb_rising2_ui_rs**(*acb_t u*, *acb_t v*, **const** *acb_t x*, *ulong n*, *ulong step*, *slong prec*)

void **acb_rising2_ui**(*acb_t u*, *acb_t v*, **const** *acb_t x*, *ulong n*, *slong prec*)
Letting $u(x) = x(x+1)(x+2)\cdots(x+n-1)$, simultaneously compute $u(x)$ and $v(x) = u'(x)$, respectively using binary splitting, rectangular splitting (with optional nonzero step length *step* to override the default choice), and an automatic algorithm choice.

void **acb_rising_ui_get_mag**(*mag_t bound*, **const** *acb_t x*, *ulong n*)
Computes an upper bound for the absolute value of the rising factorial $z = x(x+1)(x+2)\cdots(x+n-1)$. Not currently optimized for large n.

5.2.19 Gamma function

void **acb_gamma**(*acb_t* *y*, **const** *acb_t* *x*, *slong* *prec*)
Computes the gamma function $y = \Gamma(x)$.

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

void **acb_lgamma**(*acb_t* *y*, **const** *acb_t* *x*, *slong* *prec*)
Computes the logarithmic gamma function $y = \log \Gamma(x)$.

The branch cut of the logarithmic gamma function is placed on the negative half-axis, which means that $\log \Gamma(z)+\log z = \log \Gamma(z+1)$ holds for all z, whereas $\log \Gamma(z) \neq \log(\Gamma(z))$ in general. In the left half plane, the reflection formula with correct branch structure is evaluated via *acb_log_sin_pi()*.

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

void **acb_log_sin_pi**(*acb_t* *res*, **const** *acb_t* *z*, *slong* *prec*)
Computes the logarithmic sine function defined by

$$S(z) = \log(\pi) - \log \Gamma(z) + \log \Gamma(1 - z)$$

which is equal to

$$S(z) = \int_{1/2}^{z} \pi \cot(\pi t) dt$$

where the path of integration goes through the upper half plane if $0 < \arg(z) \leq \pi$ and through the lower half plane if $-\pi < \arg(z) \leq 0$. Equivalently,

$$S(z) = \log(\sin(\pi(z - n))) \mp n\pi i, \quad n = \lfloor \operatorname{re}(z) \rfloor$$

where the negative sign is taken if $0 < \arg(z) \leq \pi$ and the positive sign is taken otherwise (if the interval $\arg(z)$ does not certainly satisfy either condition, the union of both cases is computed). After subtracting n, we have $0 \leq \operatorname{re}(z) < 1$. In this strip, we use use $S(z) = \log(\sin(\pi(z)))$ if the imaginary part of z is small. Otherwise, we use $S(z) = i\pi(z - 1/2) + \log((1 + e^{-2i\pi z})/2)$ in the lower half-plane and the conjugated expression in the upper half-plane to avoid exponent overflow.

The function is evaluated at the midpoint and the propagated error is computed from $S'(z)$ to get a continuous change when z is non-real and n spans more than one possible integer value.

void **acb_polygamma**(*acb_t* *res*, **const** *acb_t* *s*, **const** *acb_t* *z*, *slong* *prec*)
Sets *res* to the value of the generalized polygamma function $\psi(s, z)$.

If s is a nonnegative order, this is simply the s-order derivative of the digamma function. If $s = 0$, this function simply calls the digamma function internally. For integers $s \geq 1$, it calls the Hurwitz zeta function. Note that for small integers $s \geq 1$, it can be faster to use *acb_poly_digamma_series()* and read off the coefficients.

The generalization to other values of s is due to Espinosa and Moll [EM2004]:

$$\psi(s, z) = \frac{\zeta'(s + 1, z) + (\gamma + \psi(-s))\zeta(s + 1, z)}{\Gamma(-s)}$$

void **acb_barnes_g**(*acb_t* *res*, **const** *acb_t* *z*, *slong* *prec*)

void **acb_log_barnes_g**(*acb_t* *res*, **const** *acb_t* *z*, *slong* *prec*)
Computes Barnes G-function or the logarithmic Barnes G-function, respectively. The logarithmic version has branch cuts on the negative real axis and is continuous elsewhere in the complex plane, in analogy with the logarithmic gamma function. The functional equation

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

holds for all z.

For small integers, we directly use the recurrence relation $G(z+1) = \Gamma(z)G(z)$ together with the initial value $G(1) = 1$. For general z, we use the formula

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

5.2.20 Zeta function

void **acb_zeta**(*acb_t z*, **const** *acb_t s*, *slong prec*)
Sets z to the value of the Riemann zeta function $\zeta(s)$. Note: for computing derivatives with respect to s, use *acb_poly_zeta_series()* or related methods.

This is a wrapper of *acb_dirichlet_zeta()*.

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()*.

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.

5.2.21 Polylogarithms

void **acb_polylog**(*acb_t w*, **const** *acb_t s*, **const** *acb_t z*, *slong prec*)

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

5.2.22 Arithmetic-geometric mean

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

void **acb_agm1**(*acb_t m*, **const** *acb_t z*, *slong prec*)
Sets m to the arithmetic-geometric mean $M(z) = \mathrm{agm}(1,z)$, defined such that the function is continuous in the complex plane except for a branch cut along the negative half axis (where it is continuous from above). This corresponds to always choosing an "optimal" branch for the square root in the arithmetic-geometric mean iteration.

void **acb_agm1_cpx**(*acb_ptr m*, **const** *acb_t z*, *slong len*, *slong prec*)
Sets the coefficients in the array m to the power series expansion of the arithmetic-geometric mean at the point z truncated to length *len*, i.e. $M(z+x) \in \mathbb{C}[[x]]$.

void **acb_agm**(*acb_t m*, **const** *acb_t x*, **const** *acb_t y*, *slong prec*)
Sets m to the arithmetic-geometric mean of x and y. The square roots in the AGM iteration are chosen so as to form the "optimal" AGM sequence. This gives a well-defined function of x and y except when x/y is a negative real number, in which case there are two optimal AGM sequences. In that case, an arbitrary but consistent choice is made (if a decision cannot be made due to inexact arithmetic, the union of both choices is returned).

5.2.23 Other special functions

void **acb_chebyshev_t_ui**(*acb_t* *a*, *ulong* *n*, **const** *acb_t* *x*, *slong* *prec*)

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

void **acb_chebyshev_t2_ui**(*acb_t* *a*, *acb_t* *b*, *ulong* *n*, **const** *acb_t* *x*, *slong* *prec*)

void **acb_chebyshev_u2_ui**(*acb_t* *a*, *acb_t* *b*, *ulong* *n*, **const** *acb_t* *x*, *slong* *prec*)
Simultaneously evaluates $a = T_n(x), b = T_{n-1}(x)$ or $a = U_n(x), b = U_{n-1}(x)$. Aliasing between a, b and x is not permitted.

5.2.24 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 $\operatorname{Re}(z) = 0$.

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 $\operatorname{Re}(z) = 0$. 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 $\operatorname{Re}(z) = 0$.

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 $\operatorname{Re}(z) = n$.

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 $\operatorname{Re}(z) = n$.

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 $\operatorname{Re}(x) \geq \operatorname{Re}(y)$ and returning y when $\operatorname{Re}(x) < \operatorname{Re}(y)$, with discontinuities where $\operatorname{Re}(x) = \operatorname{Re}(y)$.

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 $\operatorname{Re}(x) \leq \operatorname{Re}(y)$ and returning y when $\operatorname{Re}(x) > \operatorname{Re}(y)$, with discontinuities where $\operatorname{Re}(x) = \operatorname{Re}(y)$.

void **acb_real_sqrtpos**(*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_sqrtpos()*, 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.

5.2.25 Vector functions

void **_acb_vec_zero**(*acb_ptr A*, *slong n*)
Sets all entries in *vec* to zero.

int **_acb_vec_is_zero**(*acb_srcptr vec*, *slong len*)
Returns nonzero iff all entries in x are zero.

int **_acb_vec_is_real**(*acb_srcptr v*, *slong len*)
Returns nonzero iff all entries in x have zero imaginary part.

void **_acb_vec_set**(*acb_ptr res*, *acb_srcptr vec*, *slong len*)
Sets *res* to a copy of *vec*.

void **_acb_vec_set_round**(*acb_ptr res*, *acb_srcptr vec*, *slong len*, *slong prec*)
Sets *res* to a copy of *vec*, rounding each entry to *prec* bits.

void **_acb_vec_neg**(*acb_ptr res*, *acb_srcptr vec*, *slong len*)

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

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

void **_acb_vec_scalar_submul**(*acb_ptr res*, *acb_srcptr vec*, *slong len*, **const** *acb_t c*, *slong prec*)

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

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

void **_acb_vec_scalar_mul_ui**(*acb_ptr res*, *acb_srcptr vec*, *slong len*, *ulong c*, *slong prec*)

void **_acb_vec_scalar_mul_2exp_si**(*acb_ptr res*, *acb_srcptr vec*, *slong len*, *slong c*)

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

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

void **_acb_vec_scalar_div**(*acb_ptr res*, *acb_srcptr vec*, *slong len*, **const** *acb_t c*, *slong prec*)

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

void **_acb_vec_scalar_div_arb**(*acb_ptr res*, *acb_srcptr vec*, *slong len*, **const** *arb_t c*, *slong prec*)

void **_acb_vec_scalar_mul_fmpz**(*acb_ptr res*, *acb_srcptr vec*, *slong len*, **const** *fmpz_t c*, *slong prec*)

void **_acb_vec_scalar_div_fmpz**(*acb_ptr res*, *acb_srcptr vec*, *slong len*, **const** *fmpz_t c*, *slong prec*)
Performs the respective scalar operation elementwise.

slong **_acb_vec_bits**(*acb_srcptr vec*, *slong len*)
Returns the maximum of *arb_bits()* for all entries in *vec*.

void **_acb_vec_set_powers**(*acb_ptr xs*, **const** *acb_t x*, *slong len*, *slong prec*)
Sets *xs* to the powers $1, x, x^2, \ldots, x^{len-1}$.

void **_acb_vec_unit_roots**(*acb_ptr z*, *slong order*, *slong len*, *slong prec*)
Sets z to the powers $1, z, z^2, \ldots z^{\text{len}-1}$ where $z = \exp(\frac{2i\pi}{\text{order}})$ 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*)
Adds the magnitude of each entry in *err* to the radius of the corresponding entry in *res*.

void **_acb_vec_indeterminate**(*acb_ptr vec*, *slong len*)
Applies *acb_indeterminate()* elementwise.

void **_acb_vec_trim**(*acb_ptr res*, *acb_srcptr vec*, *slong len*)
Applies *acb_trim()* elementwise.

int **_acb_vec_get_unique_fmpz_vec**(*fmpz* **res*, *acb_srcptr vec*, *slong len*)
Calls *acb_get_unique_fmpz()* elementwise and returns nonzero if all entries can be rounded uniquely to integers. If any entry in *vec* cannot be rounded uniquely to an integer, returns zero.

void **_acb_vec_sort_pretty**(*acb_ptr vec*, *slong len*)
Sorts the vector of complex numbers based on the real and imaginary parts. This is intended to reveal structure when printing a set of complex numbers, not to apply an order relation in a rigorous way.

CHAPTER

SIX

POLYNOMIALS AND POWER SERIES

These modules implement dense univariate polynomials with real and complex coefficients. Truncated power series are supported via methods acting on polynomials, without introducing a separate power series type.

6.1 arb_poly.h – polynomials over the real numbers

An *arb_poly_t* represents a polynomial over the real numbers, implemented as an array of coefficients of type *arb_struct*.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

6.1.1 Types, macros and constants

type arb_poly_struct

type 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.

6.1.2 Memory management

void **arb_poly_init**(*arb_poly_t poly*)
Initializes the polynomial for use, setting it to the zero polynomial.

void **arb_poly_clear**(*arb_poly_t poly*)
Clears the polynomial, deallocating all coefficients and the coefficient array.

void **arb_poly_fit_length**(*arb_poly_t poly*, *slong len*)
Makes sure that the coefficient array of the polynomial contains at least *len* initialized coefficients.

void **_arb_poly_set_length**(*arb_poly_t poly*, *slong len*)
Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void **_arb_poly_normalise**(*arb_poly_t poly*)
Strips any trailing coefficients which are identical to zero.

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.

6.1.3 Basic manipulation

slong **arb_poly_length**(**const** *arb_poly_t poly*)
Returns the length of *poly*, i.e. zero if *poly* is identically zero, and otherwise one more than the index of the highest term that is not identically zero.

slong **arb_poly_degree**(**const** *arb_poly_t poly*)
Returns the degree of *poly*, defined as one less than its length. Note that if one or several leading coefficients are balls containing zero, this value can be larger than the true degree of the exact polynomial represented by *poly*, so the return value of this function is effectively an upper bound.

int **arb_poly_is_zero**(**const** *arb_poly_t poly*)

int **arb_poly_is_one**(**const** *arb_poly_t poly*)

int **arb_poly_is_x**(**const** *arb_poly_t poly*)
Returns 1 if *poly* is exactly the polynomial 0, 1 or x respectively. Returns 0 otherwise.

void **arb_poly_zero**(*arb_poly_t poly*)

void **arb_poly_one**(*arb_poly_t poly*)
Sets *poly* to the constant 0 respectively 1.

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

void **arb_poly_set_round**(*arb_poly_t dest*, **const** *arb_poly_t src*, *slong prec*)
Sets *dest* to a copy of *src*, rounded to *prec* bits.

void **arb_poly_set_trunc**(*arb_poly_t dest*, **const** *arb_poly_t src*, *slong n*)

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.

void **arb_poly_set_coeff_si**(*arb_poly_t poly*, *slong n*, *slong c*)

void **arb_poly_set_coeff_arb**(*arb_poly_t poly*, *slong n*, **const** *arb_t c*)
Sets the coefficient with index n in *poly* to the value c. We require that n is nonnegative.

void **arb_poly_get_coeff_arb**(*arb_t v*, **const** *arb_poly_t poly*, *slong n*)
Sets v to the value of the coefficient with index n in *poly*. We require that n is nonnegative.

arb_poly_get_coeff_ptr(*poly*, *n*)
Given $n \geq 0$, returns a pointer to coefficient n of *poly*, or *NULL* if n exceeds the length of *poly*.

void **_arb_poly_shift_right**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong n*)

void **arb_poly_shift_right**(*arb_poly_t res*, **const** *arb_poly_t poly*, *slong n*)
Sets *res* to *poly* divided by x^n, throwing away the lower coefficients. We require that n is nonnegative.

void **_arb_poly_shift_left**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong n*)

void **arb_poly_shift_left**(*arb_poly_t res*, **const** *arb_poly_t poly*, *slong n*)
Sets *res* to *poly* multiplied by x^n. We require that n is nonnegative.

void **arb_poly_truncate**(*arb_poly_t poly*, *slong n*)
Truncates *poly* to have length at most n, i.e. degree strictly smaller than n. We require that n is nonnegative.

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.

6.1.4 Conversions

void **arb_poly_set_fmpz_poly**(*arb_poly_t poly*, **const** *fmpz_poly_t src*, *slong prec*)

void **arb_poly_set_fmpq_poly**(*arb_poly_t poly*, **const** *fmpq_poly_t src*, *slong prec*)

void **arb_poly_set_si**(*arb_poly_t poly*, *slong src*)
Sets *poly* to *src*, rounding the coefficients to *prec* bits.

6.1.5 Input and output

void **arb_poly_printd**(**const** *arb_poly_t poly*, *slong digits*)
Prints the polynomial as an array of coefficients, printing each coefficient using *arb_printd*.

void **arb_poly_fprintd**(FILE ***file*, **const** *arb_poly_t poly*, *slong digits*)
Prints the polynomial as an array of coefficients to the stream *file*, printing each coefficient using *arb_fprintd*.

6.1.6 Random generation

void **arb_poly_randtest**(*arb_poly_t poly*, flint_rand_t *state*, *slong len*, *slong prec*, *slong mag_bits*)
Creates a random polynomial with length at most *len*.

6.1.7 Comparisons

int **arb_poly_contains**(**const** *arb_poly_t poly1*, **const** *arb_poly_t poly2*)

int **arb_poly_contains_fmpz_poly**(**const** *arb_poly_t poly1*, **const** *fmpz_poly_t poly2*)

int **arb_poly_contains_fmpq_poly**(**const** *arb_poly_t poly1*, **const** *fmpq_poly_t poly2*)
Returns nonzero iff *poly1* contains *poly2*.

int **arb_poly_equal**(**const** *arb_poly_t A*, **const** *arb_poly_t B*)
Returns nonzero iff A and B are equal as polynomial balls, i.e. all coefficients have equal midpoint and radius.

int **_arb_poly_overlaps**(*arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*)

int **arb_poly_overlaps**(**const** *arb_poly_t poly1*, **const** *arb_poly_t poly2*)
Returns nonzero iff *poly1* overlaps with *poly2*. The underscore function requires that *len1* ist at least as large as *len2*.

int **arb_poly_get_unique_fmpz_poly**(*fmpz_poly_t z*, **const** *arb_poly_t x*)
If x contains a unique integer polynomial, sets z to that value and returns nonzero. Otherwise (if x represents no integers or more than one integer), returns zero, possibly partially modifying z.

6.1.8 Bounds

void **_arb_poly_majorant**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong prec*)

void **arb_poly_majorant**(*arb_poly_t res*, **const** *arb_poly_t poly*, *slong prec*)
Sets *res* to an exact real polynomial whose coefficients are upper bounds for the absolute values of the coefficients in *poly*, rounded to *prec* bits.

6.1.9 Arithmetic

void **_arb_poly_add**(*arb_ptr C*, *arb_srcptr A*, *slong lenA*, *arb_srcptr B*, *slong lenB*, *slong prec*)
Sets *{C, max(lenA, lenB)}* to the sum of *{A, lenA}* and *{B, lenB}*. Allows aliasing of the input and output operands.

void **arb_poly_add**(*arb_poly_t C*, **const** *arb_poly_t A*, **const** *arb_poly_t B*, *slong prec*)

void **arb_poly_add_si**(*arb_poly_t C*, **const** *arb_poly_t A*, *slong B*, *slong prec*)
Sets C to the sum of A and B.

void **_arb_poly_sub**(*arb_ptr C*, *arb_srcptr A*, *slong lenA*, *arb_srcptr B*, *slong lenB*, *slong prec*)
Sets *{C, max(lenA, lenB)}* to the difference of *{A, lenA}* and *{B, lenB}*. Allows aliasing of the input and output operands.

void **arb_poly_sub**(*arb_poly_t C*, **const** *arb_poly_t A*, **const** *arb_poly_t B*, *slong prec*)
Sets C to the difference of A and B.

void **arb_poly_add_series**(*arb_poly_t C*, **const** *arb_poly_t A*, **const** *arb_poly_t B*, *slong len*, *slong prec*)
Sets C to the sum of A and B, truncated to length *len*.

void **arb_poly_sub_series**(*arb_poly_t C*, **const** *arb_poly_t A*, **const** *arb_poly_t B*, *slong len*, *slong prec*)
Sets C to the difference of A and B, truncated to length *len*.

void **arb_poly_neg**(*arb_poly_t C*, **const** *arb_poly_t A*)
Sets C to the negation of A.

void **arb_poly_scalar_mul_2exp_si**(*arb_poly_t C*, **const** *arb_poly_t A*, *slong c*)
Sets C to A multiplied by 2^c.

void **arb_poly_scalar_mul**(*arb_poly_t C*, **const** *arb_poly_t A*, **const** *arb_t c*, *slong prec*)
Sets C to A multiplied by c.

void **arb_poly_scalar_div**(*arb_poly_t C*, **const** *arb_poly_t A*, **const** *arb_t c*, *slong prec*)
Sets C to A divided by c.

void **_arb_poly_mullow_classical**(*arb_ptr C*, *arb_srcptr A*, *slong lenA*, *arb_srcptr B*, *slong lenB*, *slong n*, *slong prec*)

void **_arb_poly_mullow_block**(*arb_ptr C*, *arb_srcptr A*, *slong lenA*, *arb_srcptr B*, *slong lenB*, *slong n*, *slong prec*)

void **_arb_poly_mullow**(*arb_ptr C*, *arb_srcptr A*, *slong lenA*, *arb_srcptr B*, *slong lenB*, *slong n*, *slong prec*)
Sets *{C, n}* to the product of *{A, lenA}* and *{B, lenB}*, truncated to length n. The output is not allowed to be aliased with either of the inputs. We require $\text{lenA} \geq \text{lenB} > 0$, $n > 0$, $\text{lenA} + \text{lenB} - 1 \geq n$.

The *classical* version uses a plain loop. This has good numerical stability but gets slow for large n.

The *block* version decomposes the product into several subproducts which are computed exactly over the integers.

It first attempts to find an integer c such that $A(2^c x)$ and $B(2^c x)$ have slowly varying coefficients, to reduce the number of blocks.

The scaling factor c is chosen in a quick, heuristic way by picking the first and last nonzero terms in each polynomial. If the indices in A are a_2, a_1 and the log-2 magnitudes are e_2, e_1, and the indices in B are b_2, b_1 with corresponding magnitudes f_2, f_1, then we compute c as the weighted arithmetic mean of the slopes, rounded to the nearest integer:

$$c = \left\lfloor \frac{(e_2 - e_1) + (f_2 + f_1)}{(a_2 - a_1) + (b_2 - b_1)} + \frac{1}{2} \right\rfloor.$$

This strategy is used because it is simple. It is not optimal in all cases, but will typically give good performance when multiplying two power series with a similar decay rate.

The default algorithm chooses the *classical* algorithm for short polynomials and the *block* algorithm for long polynomials.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void **arb_poly_mullow_classical**(*arb_poly_t* *C*, **const** *arb_poly_t* *A*, **const** *arb_poly_t* *B*, *slong* *n*, *slong* *prec*)

void **arb_poly_mullow_ztrunc**(*arb_poly_t* *C*, **const** *arb_poly_t* *A*, **const** *arb_poly_t* *B*, *slong* *n*, *slong* *prec*)

void **arb_poly_mullow_block**(*arb_poly_t* *C*, **const** *arb_poly_t* *A*, **const** *arb_poly_t* *B*, *slong* *n*, *slong* *prec*)

void **arb_poly_mullow**(*arb_poly_t* *C*, **const** *arb_poly_t* *A*, **const** *arb_poly_t* *B*, *slong* *n*, *slong* *prec*)

Sets C to the product of A and B, truncated to length n. If the same variable is passed for A and B, sets C to the square of A truncated to length n.

void **_arb_poly_mul**(*arb_ptr* *C*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *prec*)

Sets *{C, lenA + lenB - 1}* to the product of *{A, lenA}* and *{B, lenB}*. The output is not allowed to be aliased with either of the inputs. We require lenA $\geq$ lenB > 0. This function is implemented as a simple wrapper for *_arb_poly_mullow()*.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void **arb_poly_mul**(*arb_poly_t* *C*, **const** *arb_poly_t* *A*, **const** *arb_poly_t* *B*, *slong* *prec*)

Sets C to the product of A and B. If the same variable is passed for A and B, sets C to the square of A.

void **_arb_poly_inv_series**(*arb_ptr* *Q*, *arb_srcptr* *A*, *slong* *Alen*, *slong* *len*, *slong* *prec*)

Sets *{Q, len}* to the power series inverse of *{A, Alen}*. Uses Newton iteration.

void **arb_poly_inv_series**(*arb_poly_t* *Q*, **const** *arb_poly_t* *A*, *slong* *n*, *slong* *prec*)

Sets Q to the power series inverse of A, truncated to length n.

void **_arb_poly_div_series**(*arb_ptr* *Q*, *arb_srcptr* *A*, *slong* *Alen*, *arb_srcptr* *B*, *slong* *Blen*, *slong* *n*, *slong* *prec*)

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

void **arb_poly_div_series**(*arb_poly_t* *Q*, **const** *arb_poly_t* *A*, **const** *arb_poly_t* *B*, *slong* *n*, *slong* *prec*)

Sets Q to the power series quotient A divided by B, truncated to length n.

void **_arb_poly_div**(*arb_ptr* *Q*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *prec*)

void **_arb_poly_rem**(*arb_ptr* *R*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *prec*)

void **_arb_poly_divrem**(*arb_ptr* *Q*, *arb_ptr* *R*, *arb_srcptr* *A*, *slong* *lenA*, *arb_srcptr* *B*, *slong* *lenB*, *slong* *prec*)

int **arb_poly_divrem**(*arb_poly_t* *Q*, *arb_poly_t* *R*, **const** *arb_poly_t* *A*, **const** *arb_poly_t* *B*, *slong* *prec*)

Performs polynomial division with remainder, computing a quotient Q and a remainder R such that $A = BQ + R$. The implementation reverses the inputs and performs power series division.

If the leading coefficient of B contains zero (or if B is identically zero), returns 0 indicating failure without modifying the outputs. Otherwise returns nonzero.

void **_arb_poly_div_root**(*arb_ptr* *Q*, *arb_t* *R*, *arb_srcptr* *A*, *slong* *len*, **const** *arb_t* *c*, *slong* *prec*)

Divides A by the polynomial $x - c$, computing the quotient Q as well as the remainder $R = f(c)$.

6.1.10 Composition

void **_arb_poly_taylor_shift_horner**(*arb_ptr g*, **const** *arb_t c*, *slong n*, *slong prec*)

void **arb_poly_taylor_shift_horner**(*arb_poly_t g*, **const** *arb_poly_t f*, **const** *arb_t c*, *slong prec*)

void **_arb_poly_taylor_shift_divconquer**(*arb_ptr g*, **const** *arb_t c*, *slong n*, *slong prec*)

void **arb_poly_taylor_shift_divconquer**(*arb_poly_t g*, **const** *arb_poly_t f*, **const** *arb_t c*, *slong prec*)

void **_arb_poly_taylor_shift_convolution**(*arb_ptr g*, **const** *arb_t c*, *slong n*, *slong prec*)

void **arb_poly_taylor_shift_convolution**(*arb_poly_t g*, **const** *arb_poly_t f*, **const** *arb_t c*, *slong prec*)

void **_arb_poly_taylor_shift**(*arb_ptr g*, **const** *arb_t c*, *slong n*, *slong prec*)

void **arb_poly_taylor_shift**(*arb_poly_t g*, **const** *arb_poly_t f*, **const** *arb_t c*, *slong prec*)
Sets g to the Taylor shift $f(x+c)$, computed respectively using an optimized form of Horner's rule, divide-and-conquer, a single convolution, and an automatic choice between the three algorithms.

The underscore methods act in-place on $g = f$ which has length n.

void **_arb_poly_compose_horner**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong prec*)

void **arb_poly_compose_horner**(*arb_poly_t res*, **const** *arb_poly_t poly1*, **const** *arb_poly_t poly2*, *slong prec*)

void **_arb_poly_compose_divconquer**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong prec*)

void **arb_poly_compose_divconquer**(*arb_poly_t res*, **const** *arb_poly_t poly1*, **const** *arb_poly_t poly2*, *slong prec*)

void **_arb_poly_compose**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong prec*)

void **arb_poly_compose**(*arb_poly_t res*, **const** *arb_poly_t poly1*, **const** *arb_poly_t poly2*, *slong prec*)
Sets *res* to the composition $h(x) = f(g(x))$ where f is given by *poly1* and g is given by *poly2*, respectively using Horner's rule, divide-and-conquer, and an automatic choice between the two algorithms.

The default algorithm also handles special-form input $g = ax^n + c$ efficiently by performing a Taylor shift followed by a rescaling.

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

void **_arb_poly_compose_series_horner**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong n*, *slong prec*)

void **arb_poly_compose_series_horner**(*arb_poly_t res*, **const** *arb_poly_t poly1*, **const** *arb_poly_t poly2*, *slong n*, *slong prec*)

void **_arb_poly_compose_series_brent_kung**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong n*, *slong prec*)

void **arb_poly_compose_series_brent_kung**(*arb_poly_t res*, **const** *arb_poly_t poly1*, **const** *arb_poly_t poly2*, *slong n*, *slong prec*)

void **_arb_poly_compose_series**(*arb_ptr res*, *arb_srcptr poly1*, *slong len1*, *arb_srcptr poly2*, *slong len2*, *slong n*, *slong prec*)

void **arb_poly_compose_series**(*arb_poly_t res*, **const** *arb_poly_t poly1*, **const** *arb_poly_t poly2*, *slong n*, *slong prec*)
Sets *res* to the power series composition $h(x) = f(g(x))$ truncated to order $O(x^n)$ where f is given by *poly1* and g is given by *poly2*, respectively using Horner's rule, the Brent-Kung baby step-giant step algorithm, and an automatic choice between the two algorithms.

The default algorithm also handles special-form input $g = ax^n$ efficiently.

We require that the constant term in $g(x)$ is exactly zero. The underscore methods do not support aliasing of the output with either input polynomial.

void **_arb_poly_revert_series_lagrange**(*arb_ptr* *h*, *arb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **arb_poly_revert_series_lagrange**(*arb_poly_t* *h*, **const** *arb_poly_t* *f*, *slong* *n*, *slong* *prec*)

void **_arb_poly_revert_series_newton**(*arb_ptr* *h*, *arb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **arb_poly_revert_series_newton**(*arb_poly_t* *h*, **const** *arb_poly_t* *f*, *slong* *n*, *slong* *prec*)

void **_arb_poly_revert_series_lagrange_fast**(*arb_ptr* *h*, *arb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **arb_poly_revert_series_lagrange_fast**(*arb_poly_t* *h*, **const** *arb_poly_t* *f*, *slong* *n*, *slong* *prec*)

void **_arb_poly_revert_series**(*arb_ptr* *h*, *arb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **arb_poly_revert_series**(*arb_poly_t* *h*, **const** *arb_poly_t* *f*, *slong* *n*, *slong* *prec*)
Sets h to the power series reversion of f, i.e. the expansion of the compositional inverse function $f^{-1}(x)$, truncated to order $O(x^n)$, using respectively Lagrange inversion, Newton iteration, fast Lagrange inversion, and a default algorithm choice.

We require that the constant term in f is exactly zero and that the linear term is nonzero. The underscore methods assume that *flen* is at least 2, and do not support aliasing.

6.1.11 Evaluation

void **_arb_poly_evaluate_horner**(*arb_t* *y*, *arb_srcptr* *f*, *slong* *len*, **const** *arb_t* *x*, *slong* *prec*)

void **arb_poly_evaluate_horner**(*arb_t* *y*, **const** *arb_poly_t* *f*, **const** *arb_t* *x*, *slong* *prec*)

void **_arb_poly_evaluate_rectangular**(*arb_t* *y*, *arb_srcptr* *f*, *slong* *len*, **const** *arb_t* *x*, *slong* *prec*)

void **arb_poly_evaluate_rectangular**(*arb_t* *y*, **const** *arb_poly_t* *f*, **const** *arb_t* *x*, *slong* *prec*)

void **_arb_poly_evaluate**(*arb_t* *y*, *arb_srcptr* *f*, *slong* *len*, **const** *arb_t* *x*, *slong* *prec*)

void **arb_poly_evaluate**(*arb_t* *y*, **const** *arb_poly_t* *f*, **const** *arb_t* *x*, *slong* *prec*)
Sets $y = f(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

void **_arb_poly_evaluate_acb_horner**(*acb_t* *y*, *arb_srcptr* *f*, *slong* *len*, **const** *acb_t* *x*, *slong* *prec*)

void **arb_poly_evaluate_acb_horner**(*acb_t* *y*, **const** *arb_poly_t* *f*, **const** *acb_t* *x*, *slong* *prec*)

void **_arb_poly_evaluate_acb_rectangular**(*acb_t* *y*, *arb_srcptr* *f*, *slong* *len*, **const** *acb_t* *x*, *slong* *prec*)

void **arb_poly_evaluate_acb_rectangular**(*acb_t* *y*, **const** *arb_poly_t* *f*, **const** *acb_t* *x*, *slong* *prec*)

void **_arb_poly_evaluate_acb**(*acb_t* *y*, *arb_srcptr* *f*, *slong* *len*, **const** *acb_t* *x*, *slong* *prec*)

void **arb_poly_evaluate_acb**(*acb_t* *y*, **const** *arb_poly_t* *f*, **const** *acb_t* *x*, *slong* *prec*)
Sets $y = f(x)$ where x is a complex number, evaluating the polynomial respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

void **_arb_poly_evaluate2_horner**(*arb_t* *y*, *arb_t* *z*, *arb_srcptr* *f*, *slong* *len*, **const** *arb_t* *x*, *slong* *prec*)

void **arb_poly_evaluate2_horner**(*arb_t* *y*, *arb_t* *z*, **const** *arb_poly_t* *f*, **const** *arb_t* *x*, *slong* *prec*)

void **_arb_poly_evaluate2_rectangular**(*arb_t y*, *arb_t z*, *arb_srcptr f*, *slong len*, **const** *arb_t x*, *slong prec*)

void **arb_poly_evaluate2_rectangular**(*arb_t y*, *arb_t z*, **const** *arb_poly_t f*, **const** *arb_t x*, *slong prec*)

void **_arb_poly_evaluate2**(*arb_t y*, *arb_t z*, *arb_srcptr f*, *slong len*, **const** *arb_t x*, *slong prec*)

void **arb_poly_evaluate2**(*arb_t y*, *arb_t z*, **const** *arb_poly_t f*, **const** *arb_t x*, *slong prec*)
Sets $y = f(x), z = f'(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

When Horner's rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.

void **_arb_poly_evaluate2_acb_horner**(*acb_t y*, *acb_t z*, *arb_srcptr f*, *slong len*, **const** *acb_t x*, *slong prec*)

void **arb_poly_evaluate2_acb_horner**(*acb_t y*, *acb_t z*, **const** *arb_poly_t f*, **const** *acb_t x*, *slong prec*)

void **_arb_poly_evaluate2_acb_rectangular**(*acb_t y*, *acb_t z*, *arb_srcptr f*, *slong len*, **const** *acb_t x*, *slong prec*)

void **arb_poly_evaluate2_acb_rectangular**(*acb_t y*, *acb_t z*, **const** *arb_poly_t f*, **const** *acb_t x*, *slong prec*)

void **_arb_poly_evaluate2_acb**(*acb_t y*, *acb_t z*, *arb_srcptr f*, *slong len*, **const** *acb_t x*, *slong prec*)

void **arb_poly_evaluate2_acb**(*acb_t y*, *acb_t z*, **const** *arb_poly_t f*, **const** *acb_t x*, *slong prec*)
Sets $y = f(x), z = f'(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

6.1.12 Product trees

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

void **arb_poly_product_roots**(*arb_poly_t poly*, *arb_srcptr xs*, *slong n*, *slong prec*)
Generates the polynomial $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$.

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

void **arb_poly_product_roots_complex**(*arb_poly_t 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 *acb_poly_product_roots()* instead.

arb_ptr ***_arb_poly_tree_alloc**(*slong len*)
Returns an initialized data structured capable of representing a remainder tree (product tree) of *len* roots.

void **_arb_poly_tree_free**(*arb_ptr* **tree*, *slong len*)
Deallocates a tree structure as allocated using *_arb_poly_tree_alloc*.

void **_arb_poly_tree_build**(*arb_ptr* **tree*, *arb_srcptr roots*, *slong len*, *slong prec*)
Constructs a product tree from a given array of *len* roots. The tree structure must be pre-allocated to the specified length using *_arb_poly_tree_alloc()*.

6.1.13 Multipoint evaluation

void **_arb_poly_evaluate_vec_iter**(*arb_ptr ys*, *arb_srcptr poly*, *slong plen*, *arb_srcptr xs*, *slong n*, *slong prec*)

void **arb_poly_evaluate_vec_iter**(*arb_ptr ys*, **const** *arb_poly_t poly*, *arb_srcptr xs*, *slong n*, *slong prec*)
Evaluates the polynomial simultaneously at n given points, calling *_arb_poly_evaluate()* repeatedly.

void **_arb_poly_evaluate_vec_fast_precomp**(*arb_ptr vs*, *arb_srcptr poly*, *slong plen*, *arb_ptr* **tree*, *slong len*, *slong prec*)

void **_arb_poly_evaluate_vec_fast**(*arb_ptr ys*, *arb_srcptr poly*, *slong plen*, *arb_srcptr xs*, *slong n*, *slong prec*)

void **arb_poly_evaluate_vec_fast**(*arb_ptr ys*, **const** *arb_poly_t poly*, *arb_srcptr xs*, *slong n*, *slong prec*)
Evaluates the polynomial simultaneously at n given points, using fast multipoint evaluation.

6.1.14 Interpolation

void **_arb_poly_interpolate_newton**(*arb_ptr poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong n*, *slong prec*)

void **arb_poly_interpolate_newton**(*arb_poly_t poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong n*, *slong prec*)
Recovers the unique polynomial of length at most n that interpolates the given x and y values. This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

void **_arb_poly_interpolate_barycentric**(*arb_ptr poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong n*, *slong prec*)

void **arb_poly_interpolate_barycentric**(*arb_poly_t poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong n*, *slong prec*)
Recovers the unique polynomial of length at most n that interpolates the given x and y values. This implementation uses the barycentric form of Lagrange interpolation.

void **_arb_poly_interpolation_weights**(*arb_ptr w*, *arb_ptr* **tree*, *slong len*, *slong prec*)

void **_arb_poly_interpolate_fast_precomp**(*arb_ptr poly*, *arb_srcptr ys*, *arb_ptr* **tree*, *arb_srcptr weights*, *slong len*, *slong prec*)

void **_arb_poly_interpolate_fast**(*arb_ptr poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong len*, *slong prec*)

void **arb_poly_interpolate_fast**(*arb_poly_t poly*, *arb_srcptr xs*, *arb_srcptr ys*, *slong n*, *slong prec*)
Recovers the unique polynomial of length at most n that interpolates the given x and y values, using fast Lagrange interpolation. The precomp function takes a precomputed product tree over the x values and a vector of interpolation weights as additional inputs.

6.1.15 Differentiation

void **_arb_poly_derivative**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong prec*)
Sets *{res, len - 1}* to the derivative of *{poly, len}*. Allows aliasing of the input and output.

void **arb_poly_derivative**(*arb_poly_t res*, **const** *arb_poly_t poly*, *slong prec*)
Sets *res* to the derivative of *poly*.

void **_arb_poly_integral**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong prec*)
Sets *{res, len}* to the integral of *{poly, len - 1}*. Allows aliasing of the input and output.

void **arb_poly_integral**(*arb_poly_t res*, **const** *arb_poly_t poly*, *slong prec*)
Sets *res* to the integral of *poly*.

6.1.16 Transforms

void **_arb_poly_borel_transform**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong prec*)

void **arb_poly_borel_transform**(*arb_poly_t res*, **const** *arb_poly_t poly*, *slong prec*)
Computes the Borel transform of the input polynomial, mapping $\sum_k a_k x^k$ to $\sum_k (a_k/k!)x^k$. The underscore method allows aliasing.

void **_arb_poly_inv_borel_transform**(*arb_ptr res*, *arb_srcptr poly*, *slong len*, *slong prec*)

void **arb_poly_inv_borel_transform**(*arb_poly_t res*, **const** *arb_poly_t poly*, *slong prec*)
Computes the inverse Borel transform of the input polynomial, mapping $\sum_k a_k x^k$ to $\sum_k a_k k! x^k$. The underscore method allows aliasing.

void **_arb_poly_binomial_transform_basecase**(*arb_ptr b*, *arb_srcptr a*, *slong alen*, *slong len*, *slong prec*)

void **arb_poly_binomial_transform_basecase**(*arb_poly_t b*, **const** *arb_poly_t a*, *slong len*, *slong prec*)

void **_arb_poly_binomial_transform_convolution**(*arb_ptr b*, *arb_srcptr a*, *slong alen*, *slong len*, *slong prec*)

void **arb_poly_binomial_transform_convolution**(*arb_poly_t b*, **const** *arb_poly_t a*, *slong len*, *slong prec*)

void **_arb_poly_binomial_transform**(*arb_ptr b*, *arb_srcptr a*, *slong alen*, *slong len*, *slong prec*)

void **arb_poly_binomial_transform**(*arb_poly_t b*, **const** *arb_poly_t a*, *slong len*, *slong prec*)
Computes the binomial transform of the input polynomial, truncating the output to length *len*. The binomial transform maps the coefficients a_k in the input polynomial to the coefficients b_k in the output polynomial via $b_n = \sum_{k=0}^{n} (-1)^k \binom{n}{k} a_k$. The binomial transform is equivalent to the power series composition $f(x) \to (1-x)^{-1} f(x/(x-1))$, and is its own inverse.

The *basecase* version evaluates coefficients one by one from the definition, generating the binomial coefficients by a recurrence relation.

The *convolution* version uses the identity $T(f(x)) = B^{-1}(e^x B(f(-x)))$ where T denotes the binomial transform operator and B denotes the Borel transform operator. This only costs a single polynomial multiplication, plus some scalar operations.

The default version automatically chooses an algorithm.

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

6.1.17 Powers and elementary functions

void **_arb_poly_pow_ui_trunc_binexp**(*arb_ptr res*, *arb_srcptr f*, *slong flen*, *ulong exp*, *slong len*, *slong prec*)

Sets *{res, len}* to *{f, flen}* raised to the power *exp*, truncated to length *len*. Requires that *len* is no longer than the length of the power as computed without truncation (i.e. no zero-padding is performed). Does not support aliasing of the input and output, and requires that *flen* and *len* are positive. Uses binary expontiation.

void **arb_poly_pow_ui_trunc_binexp**(*arb_poly_t res*, **const** *arb_poly_t poly*, *ulong exp*, *slong len*, *slong prec*)

Sets *res* to *poly* raised to the power *exp*, truncated to length *len*. Uses binary exponentiation.

void **_arb_poly_pow_ui**(*arb_ptr res*, *arb_srcptr f*, *slong flen*, *ulong exp*, *slong prec*)

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

void **arb_poly_pow_ui**(*arb_poly_t res*, **const** *arb_poly_t poly*, *ulong exp*, *slong prec*)

Sets *res* to *poly* raised to the power *exp*.

void **_arb_poly_pow_series**(*arb_ptr h*, *arb_srcptr f*, *slong flen*, *arb_srcptr g*, *slong glen*, *slong len*, *slong prec*)

Sets *{h, len}* to the power series $f(x)^{g(x)} = \exp(g(x) \log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that *flen* and *glen* do not exceed *len*.

void **arb_poly_pow_series**(*arb_poly_t h*, **const** *arb_poly_t f*, **const** *arb_poly_t g*, *slong len*, *slong prec*)

Sets h to the power series $f(x)^{g(x)} = \exp(g(x) \log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently.

void **_arb_poly_pow_arb_series**(*arb_ptr h*, *arb_srcptr f*, *slong flen*, **const** *arb_t g*, *slong len*, *slong prec*)

Sets *{h, len}* to the power series $f(x)^g = \exp(g \log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that *flen* does not exceed *len*.

void **arb_poly_pow_arb_series**(*arb_poly_t h*, **const** *arb_poly_t f*, **const** *arb_t g*, *slong len*, *slong prec*)

Sets h to the power series $f(x)^g = \exp(g \log f(x))$ truncated to length *len*.

void **_arb_poly_sqrt_series**(*arb_ptr g*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_sqrt_series**(*arb_poly_t g*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

Sets g to the power series square root of h, truncated to length n. Uses division-free Newton iteration for the reciprocal square root, followed by a multiplication.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and n are greater than zero.

void **_arb_poly_rsqrt_series**(*arb_ptr g*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_rsqrt_series**(*arb_poly_t g*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

Sets g to the reciprocal power series square root of h, truncated to length n. Uses division-free Newton iteration.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and n are greater than zero.

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

void **arb_poly_log_series**(*arb_poly_t res*, **const** *arb_poly_t f*, *slong n*, *slong prec*)
Sets *res* to the power series logarithm of *f*, truncated to length *n*. Uses the formula $\log(f(x)) = \int f'(x)/f(x)dx$, adding the logarithm of the constant term in f as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that *flen* and n are greater than zero.

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

void **arb_poly_log1p_series**(*arb_poly_t res*, **const** *arb_poly_t f*, *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*)
Sets *res* respectively to the power series inverse tangent, inverse sine and inverse cosine of *f*, truncated to length n.

Uses the formulas

$$\tan^{-1}(f(x)) = \int f'(x)/(1 + f(x)^2)dx,$$
$$\sin^{-1}(f(x)) = \int f'(x)/(1 - f(x)^2)^{1/2}dx,$$
$$\cos^{-1}(f(x)) = -\int f'(x)/(1 - f(x)^2)^{1/2}dx,$$

adding the inverse function of the constant term in f as the constant of integration.

The underscore methods supports aliasing of the input and output arrays. They require that *flen* and n are greater than zero.

void **_arb_poly_exp_series_basecase**(*arb_ptr f*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_exp_series_basecase**(*arb_poly_t f*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_exp_series**(*arb_ptr f*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_exp_series**(*arb_poly_t f*, **const** *arb_poly_t h*, *slong n*, *slong prec*)
Sets f to the power series exponential of h, truncated to length n.

The basecase version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where m is the length of h.

The main implementation uses Newton iteration, starting from a small number of terms given by the basecase algorithm. The complexity is $O(M(n))$. Redundant operations in the Newton iteration are avoided by using the scheme described in [HZ2004].

The underscore methods support aliasing and allow the input to be shorter than the output, but require the lengths to be nonzero.

void **_arb_poly_sin_cos_series_basecase**(*arb_ptr s*, *arb_ptr c*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*, int *times_pi*)

void **arb_poly_sin_cos_series_basecase**(*arb_poly_t s*, *arb_poly_t c*, **const** *arb_poly_t h*, *slong n*, *slong prec*, int *times_pi*)

void **_arb_poly_sin_cos_series_tangent**(*arb_ptr s*, *arb_ptr c*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*, int *times_pi*)

void **arb_poly_sin_cos_series_tangent**(*arb_poly_t s*, *arb_poly_t c*, **const** *arb_poly_t h*, *slong n*, *slong prec*, int *times_pi*)

void **_arb_poly_sin_cos_series**(*arb_ptr* *s*, *arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sin_cos_series**(*arb_poly_t* *s*, *arb_poly_t* *c*, **const** *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Sets s and c to the power series sine and cosine of h, computed simultaneously.

The *basecase* version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where m is the length of h.

The *tangent* version uses the tangent half-angle formulas to compute the sine and cosine via *_arb_poly_tan_series()*. This requires $O(M(n))$ operations. When $h = h_0 + h_1$ where the constant term h_0 is nonzero, the evaluation is done as $\sin(h_0 + h_1) = \cos(h_0)\sin(h_1) + \sin(h_0)\cos(h_1)$, $\cos(h_0 + h_1) = \cos(h_0)\cos(h_1) - \sin(h_0)\sin(h_1)$, to improve accuracy and avoid dividing by zero at the poles of the tangent function.

The default version automatically selects between the *basecase* and *tangent* algorithms depending on the input.

The *basecase* and *tangent* versions take a flag *times_pi* specifying that the input is to be multiplied by π.

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

void **_arb_poly_sin_series**(*arb_ptr* *s*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sin_series**(*arb_poly_t* *s*, **const** *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_arb_poly_cos_series**(*arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_cos_series**(*arb_poly_t* *c*, **const** *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Respectively evaluates the power series sine or cosine. These functions simply wrap *_arb_poly_sin_cos_series()*. The underscore methods support aliasing and require the lengths to be nonzero.

void **_arb_poly_tan_series**(*arb_ptr* *g*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *len*, *slong* *prec*)

void **arb_poly_tan_series**(*arb_poly_t* *g*, **const** *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Sets g to the power series tangent of h.

For small n takes the quotient of the sine and cosine as computed using the basecase algorithm. For large n, uses Newton iteration to invert the inverse tangent series. The complexity is $O(M(n))$.

The underscore version does not support aliasing, and requires the lengths to be nonzero.

void **_arb_poly_sin_cos_pi_series**(*arb_ptr* *s*, *arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sin_cos_pi_series**(*arb_poly_t* *s*, *arb_poly_t* *c*, **const** *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_arb_poly_sin_pi_series**(*arb_ptr* *s*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sin_pi_series**(*arb_poly_t* *s*, **const** *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_arb_poly_cos_pi_series**(*arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_cos_pi_series**(*arb_poly_t* *c*, **const** *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_arb_poly_cot_pi_series**(*arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_cot_pi_series**(*arb_poly_t* *c*, **const** *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Compute the respective trigonometric functions of the input multiplied by π.

void **_arb_poly_sinh_cosh_series_basecase**(*arb_ptr* *s*, *arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sinh_cosh_series_basecase**(*arb_poly_t* *s*, *arb_poly_t* *c*, **const** *arb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_arb_poly_sinh_cosh_series_exponential**(*arb_ptr* *s*, *arb_ptr* *c*, *arb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **arb_poly_sinh_cosh_series_exponential**(*arb_poly_t s*, *arb_poly_t c*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_sinh_cosh_series**(*arb_ptr s*, *arb_ptr c*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_sinh_cosh_series**(*arb_poly_t s*, *arb_poly_t c*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_sinh_series**(*arb_ptr s*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_sinh_series**(*arb_poly_t s*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_cosh_series**(*arb_ptr c*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_cosh_series**(*arb_poly_t c*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

Sets s and c respectively to the hyperbolic sine and cosine of the power series h, truncated to length n.

The implementations mirror those for sine and cosine, except that the *exponential* version computes both functions using the exponential function instead of the hyperbolic tangent.

void **_arb_poly_sinc_series**(*arb_ptr s*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_sinc_series**(*arb_poly_t s*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

Sets c to the sinc function of the power series h, truncated to length n.

void **_arb_poly_sinc_pi_series**(*arb_ptr s*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_sinc_pi_series**(*arb_poly_t s*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

Compute the sinc function of the input multiplied by π.

6.1.18 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.

6.1.19 Gamma function and factorials

void **_arb_poly_gamma_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_gamma_series**(*arb_poly_t res*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_rgamma_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_rgamma_series**(*arb_poly_t res*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_lgamma_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_lgamma_series**(*arb_poly_t res*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

void **_arb_poly_digamma_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_digamma_series**(*arb_poly_t res*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

Sets *res* to the series expansion of $\Gamma(h(x))$, $1/\Gamma(h(x))$, or $\log\Gamma(h(x))$, $\psi(h(x))$, truncated to length n.

These functions first generate the Taylor series at the constant term of h, and then call *_arb_poly_compose_series()*. The Taylor coefficients are generated using the Riemann zeta function if the constant term of h is a small integer, and with Stirling's series otherwise.

The underscore methods support aliasing of the input and output arrays, and require that *hlen* and n are greater than zero.

void **_arb_poly_rising_ui_series**(*arb_ptr res*, *arb_srcptr f*, *slong flen*, *ulong r*, *slong trunc*, *slong prec*)

void **arb_poly_rising_ui_series**(*arb_poly_t res*, **const** *arb_poly_t f*, *ulong r*, *slong trunc*, *slong prec*)

Sets *res* to the rising factorial $(f)(f+1)(f+2)\cdots(f+r-1)$, truncated to length *trunc*. The underscore method assumes that *flen*, r and *trunc* are at least 1, and does not support aliasing. Uses binary splitting.

6.1.20 Zeta function

void **arb_poly_zeta_series**(*arb_poly_t res*, **const** *arb_poly_t s*, **const** *arb_t a*, int *deflate*, *slong n*, *slong prec*)

Sets *res* to the Hurwitz zeta function $\zeta(s,a)$ where s a power series and a is a constant, truncated to length n. To evaluate the usual Riemann zeta function, set $a = 1$.

If *deflate* is nonzero, evaluates $\zeta(s,a)+1/(1-s)$, which is well-defined as a limit when the constant term of s is 1. In particular, expanding $\zeta(s,a)+1/(1-s)$ with $s = 1+x$ gives the Stieltjes constants

$$\sum_{k=0}^{n-1} \frac{(-1)^k}{k!} \gamma_k(a) x^k.$$

If $a = 1$, this implementation uses the reflection formula if the midpoint of the constant term of s is negative.

void **_arb_poly_riemann_siegel_theta_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_riemann_siegel_theta_series**(*arb_poly_t res*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

Sets *res* to the series expansion of the Riemann-Siegel theta function

$$\theta(h) = \arg\left(\Gamma\left(\frac{2ih+1}{4}\right)\right) - \frac{\log \pi}{2} h$$

where the argument of the gamma function is chosen continuously as the imaginary part of the log gamma function.

The underscore method does not support aliasing of the input and output arrays, and requires that the lengths are greater than zero.

void **_arb_poly_riemann_siegel_z_series**(*arb_ptr res*, *arb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **arb_poly_riemann_siegel_z_series**(*arb_poly_t res*, **const** *arb_poly_t h*, *slong n*, *slong prec*)

Sets *res* to the series expansion of the Riemann-Siegel Z-function

$$Z(h) = e^{i\theta(h)}\zeta(1/2+ih).$$

The zeros of the Z-function on the real line precisely correspond to the imaginary parts of the zeros of the Riemann zeta function on the critical line.

The underscore method supports aliasing of the input and output arrays, and requires that the lengths are greater than zero.

6.1.21 Root-finding

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

void **arb_poly_root_bound_fujiwara**(*mag_t bound*, *arb_poly_t poly*)
Sets *bound* to an upper bound for the magnitude of all the complex roots of *poly*. Uses Fujiwara's bound

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

where $a_0, \ldots, a_n$ are the coefficients of *poly*.

void **_arb_poly_newton_convergence_factor**(*arf_t convergence_factor*, *arb_srcptr poly*, *slong len*, **const** *arb_t convergence_interval*, *slong prec*)
Given an interval I specified by *convergence_interval*, evaluates a bound for $C = \sup_{t,u \in I} \frac{1}{2}|f''(t)|/|f'(u)|$, where f is the polynomial defined by the coefficients *{poly, len}*. The bound is obtained by evaluating $f'(I)$ and $f''(I)$ directly. If f has large coefficients, I must be extremely precise in order to get a finite factor.

int **_arb_poly_newton_step**(*arb_t xnew*, *arb_srcptr poly*, *slong len*, **const** *arb_t x*, **const** *arb_t convergence_interval*, **const** *arf_t convergence_factor*, *slong prec*)
Performs a single step with Newton's method.

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

The Newton update consists of setting $x' = [m' - r', m' + r']$ where $m' = m - f(m)/f'(m)$ and $r' = Cr^2$. The expression $m - f(m)/f'(m)$ is evaluated using ball arithmetic at a working precision of *prec* bits, and the rounding error during this evaluation is accounted for in the output. We now check that $x' \in I$ and $m' < m$. If both conditions are satisfied, we set *xnew* to x' and return nonzero. If either condition fails, we set *xnew* to x and return zero, indicating that no progress was made.

void **_arb_poly_newton_refine_root**(*arb_t r*, *arb_srcptr poly*, *slong len*, **const** *arb_t start*, **const** *arb_t convergence_interval*, **const** *arf_t convergence_factor*, *slong eval_extra_prec*, *slong prec*)
Refines a precise estimate of a polynomial root to high precision by performing several Newton steps, using nearly optimally chosen doubling precision steps.

The inputs are defined as for *_arb_poly_newton_step*, except for the precision parameters: *prec* is the target accuracy and *eval_extra_prec* is the estimated number of guard bits that need to be added to evaluate the polynomial accurately close to the root (typically, if the polynomial has large coefficients of alternating signs, this needs to be approximately the bit size of the coefficients).

6.1.22 Other special polynomials

void **_arb_poly_swinnerton_dyer_ui**(*arb_ptr poly*, *ulong n*, *slong trunc*, *slong prec*)

void **arb_poly_swinnerton_dyer_ui**(*arb_poly_t poly*, *ulong n*, *slong prec*)
Computes the Swinnerton-Dyer polynomial S_n, which has degree 2^n and is the rational minimal polynomial of the sum of the square roots of the first n prime numbers.

If *prec* is set to zero, a precision is chosen automatically such that *arb_poly_get_unique_fmpz_poly()* should be successful. Otherwise a working precision of *prec* bits is used.

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

6.2 acb_poly.h – polynomials over the complex numbers

An *acb_poly_t* represents a polynomial over the complex numbers, implemented as an array of coefficients of type *acb_struct*.

Most functions are provided in two versions: an underscore method which operates directly on preallocated 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.

6.2.1 Types, macros and constants

type acb_poly_struct

type acb_poly_t
Contains a pointer to an array of coefficients (coeffs), the used length (length), and the allocated size of the array (alloc).

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

6.2.2 Memory management

void **acb_poly_init**(*acb_poly_t poly*)
Initializes the polynomial for use, setting it to the zero polynomial.

void **acb_poly_clear**(*acb_poly_t poly*)
Clears the polynomial, deallocating all coefficients and the coefficient array.

void **acb_poly_fit_length**(*acb_poly_t poly*, *slong len*)
Makes sure that the coefficient array of the polynomial contains at least *len* initialized coefficients.

void **_acb_poly_set_length**(*acb_poly_t poly*, *slong len*)
Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void **_acb_poly_normalise**(*acb_poly_t poly*)
Strips any trailing coefficients which are identical to zero.

void **acb_poly_swap**(*acb_poly_t poly1*, *acb_poly_t poly2*)
Swaps *poly1* and *poly2* efficiently.

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.

6.2.3 Basic properties and manipulation

slong **acb_poly_length**(**const** *acb_poly_t poly*)
Returns the length of *poly*, i.e. zero if *poly* is identically zero, and otherwise one more than the index of the highest term that is not identically zero.

slong **acb_poly_degree**(**const** *acb_poly_t poly*)
Returns the degree of *poly*, defined as one less than its length. Note that if one or several leading coefficients are balls containing zero, this value can be larger than the true degree of the exact polynomial represented by *poly*, so the return value of this function is effectively an upper bound.

int **acb_poly_is_zero**(**const** *acb_poly_t poly*)

int **acb_poly_is_one**(**const** *acb_poly_t poly*)

int **acb_poly_is_x**(**const** *acb_poly_t poly*)
Returns 1 if *poly* is exactly the polynomial 0, 1 or x respectively. Returns 0 otherwise.

void **acb_poly_zero**(*acb_poly_t poly*)
Sets *poly* to the zero polynomial.

void **acb_poly_one**(*acb_poly_t poly*)
Sets *poly* to the constant polynomial 1.

void **acb_poly_set**(*acb_poly_t dest*, **const** *acb_poly_t src*)
Sets *dest* to a copy of *src*.

void **acb_poly_set_round**(*acb_poly_t dest*, **const** *acb_poly_t src*, *slong prec*)
Sets *dest* to a copy of *src*, rounded to *prec* bits.

void **acb_poly_set_trunc**(*acb_poly_t dest*, **const** *acb_poly_t src*, *slong n*)

void **acb_poly_set_trunc_round**(*acb_poly_t dest*, **const** *acb_poly_t src*, *slong n*, *slong prec*)
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*)

void **acb_poly_set_coeff_acb**(*acb_poly_t poly*, *slong n*, **const** *acb_t c*)
Sets the coefficient with index n in *poly* to the value c. We require that n is nonnegative.

void **acb_poly_get_coeff_acb**(*acb_t v*, **const** *acb_poly_t poly*, *slong n*)
Sets v to the value of the coefficient with index n in *poly*. We require that n is nonnegative.

acb_poly_get_coeff_ptr(*poly*, *n*)
Given $n \geq 0$, returns a pointer to coefficient n of *poly*, or *NULL* if n exceeds the length of *poly*.

void **_acb_poly_shift_right**(*acb_ptr res*, *acb_srcptr poly*, *slong len*, *slong n*)

void **acb_poly_shift_right**(*acb_poly_t res*, **const** *acb_poly_t poly*, *slong n*)
Sets *res* to *poly* divided by x^n, throwing away the lower coefficients. We require that n is nonnegative.

void **_acb_poly_shift_left**(*acb_ptr res*, *acb_srcptr poly*, *slong len*, *slong n*)

void **acb_poly_shift_left**(*acb_poly_t res*, **const** *acb_poly_t poly*, *slong n*)
Sets *res* to *poly* multiplied by x^n. We require that n is nonnegative.

void **acb_poly_truncate**(*acb_poly_t poly*, *slong n*)
Truncates *poly* to have length at most n, i.e. degree strictly smaller than n. 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.

6.2.4 Input and output

void **acb_poly_printd**(**const** *acb_poly_t poly*, *slong digits*)
Prints the polynomial as an array of coefficients, printing each coefficient using *acb_printd*.

void **acb_poly_fprintd**(FILE **file*, **const** *acb_poly_t poly*, *slong digits*)
Prints the polynomial as an array of coefficients to the stream *file*, printing each coefficient using *acb_fprintd*.

6.2.5 Random generation

void **acb_poly_randtest**(*acb_poly_t poly*, flint_rand_t *state*, *slong len*, *slong prec*, *slong mag_bits*)
Creates a random polynomial with length at most *len*.

6.2.6 Comparisons

int **acb_poly_equal**(**const** *acb_poly_t A*, **const** *acb_poly_t B*)
Returns nonzero iff A and B are identical as interval polynomials.

int **acb_poly_contains**(**const** *acb_poly_t poly1*, **const** *acb_poly_t poly2*)

int **acb_poly_contains_fmpz_poly**(**const** *acb_poly_t poly1*, **const** *fmpz_poly_t poly2*)

int **acb_poly_contains_fmpq_poly**(**const** *acb_poly_t poly1*, **const** *fmpq_poly_t poly2*)
Returns nonzero iff *poly2* is contained in *poly1*.

int **_acb_poly_overlaps**(*acb_srcptr poly1*, *slong len1*, *acb_srcptr poly2*, *slong len2*)

int **acb_poly_overlaps**(**const** *acb_poly_t poly1*, **const** *acb_poly_t poly2*)
Returns nonzero iff *poly1* overlaps with *poly2*. The underscore function requires that *len1* ist at least as large as *len2*.

int **acb_poly_get_unique_fmpz_poly**(*fmpz_poly_t z*, **const** *acb_poly_t x*)
If x contains a unique integer polynomial, sets z to that value and returns nonzero. Otherwise (if x represents no integers or more than one integer), returns zero, possibly partially modifying z.

int **acb_poly_is_real**(**const** *acb_poly_t poly*)
Returns nonzero iff all coefficients in *poly* have zero imaginary part.

6.2.7 Conversions

void **acb_poly_set_fmpz_poly**(*acb_poly_t poly*, **const** *fmpz_poly_t re*, *slong prec*)

void **acb_poly_set2_fmpz_poly**(*acb_poly_t poly*, **const** *fmpz_poly_t re*, **const** *fmpz_poly_t im*, *slong prec*)

void **acb_poly_set_arb_poly**(*acb_poly_t poly*, **const** *arb_poly_t re*)

void **acb_poly_set2_arb_poly**(*acb_poly_t poly*, **const** *arb_poly_t re*, **const** *arb_poly_t im*)

void **acb_poly_set_fmpq_poly**(*acb_poly_t poly*, **const** *fmpq_poly_t re*, *slong prec*)

void **acb_poly_set2_fmpq_poly**(*acb_poly_t poly*, **const** *fmpq_poly_t re*, **const** *fmpq_poly_t im*, *slong prec*)
Sets *poly* to the given real part *re* plus the imaginary part *im*, both rounded to *prec* bits.

void **acb_poly_set_acb**(*acb_poly_t poly*, **const** *acb_t src*)

void **acb_poly_set_si**(*acb_poly_t poly*, *slong src*)
Sets *poly* to *src*.

6.2.8 Bounds

void **_acb_poly_majorant**(*arb_ptr res*, *acb_srcptr poly*, *slong len*, *slong prec*)

void **acb_poly_majorant**(*arb_poly_t res*, **const** *acb_poly_t poly*, *slong prec*)
Sets *res* to an exact real polynomial whose coefficients are upper bounds for the absolute values of the coefficients in *poly*, rounded to *prec* bits.

6.2.9 Arithmetic

void **_acb_poly_add**(*acb_ptr C*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)
Sets *{C, max(lenA, lenB)}* to the sum of *{A, lenA}* and *{B, lenB}*. Allows aliasing of the input and output operands.

void **acb_poly_add**(*acb_poly_t C*, **const** *acb_poly_t A*, **const** *acb_poly_t B*, *slong prec*)

void **acb_poly_add_si**(*acb_poly_t C*, **const** *acb_poly_t A*, *slong B*, *slong prec*)
Sets C to the sum of A and B.

void **_acb_poly_sub**(*acb_ptr C*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)
Sets *{C, max(lenA, lenB)}* to the difference of *{A, lenA}* and *{B, lenB}*. Allows aliasing of the input and output operands.

void **acb_poly_sub**(*acb_poly_t C*, **const** *acb_poly_t A*, **const** *acb_poly_t B*, *slong prec*)
Sets C to the difference of A and B.

void **acb_poly_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 *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 *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, lenA}* and *{B, lenB}*, truncated to length n. The output is not allowed to be aliased with either of the inputs. We require $\text{lenA} \geq \text{lenB} > 0$, $n > 0$, $\text{lenA} + \text{lenB} - 1 \geq n$.

The *classical* version uses a plain loop.

The *transpose* version evaluates the product using four real polynomial multiplications (via *_arb_poly_mullow()*).

The *transpose_gauss* version evaluates the product using three real polynomial multiplications. This is almost always faster than *transpose*, but has worse numerical stability when the coefficients vary in magnitude.

The default function *_acb_poly_mullow()* automatically switches been *classical* and *transpose* multiplication.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void **acb_poly_mullow_classical**(*acb_poly_t C*, **const** *acb_poly_t A*, **const** *acb_poly_t B*, *slong n*, *slong prec*)

void **acb_poly_mullow_transpose**(*acb_poly_t C*, **const** *acb_poly_t A*, **const** *acb_poly_t B*, *slong n*, *slong prec*)

void **acb_poly_mullow_transpose_gauss**(*acb_poly_t C*, **const** *acb_poly_t A*, **const** *acb_poly_t B*, *slong n*, *slong prec*)

void **acb_poly_mullow**(*acb_poly_t C*, **const** *acb_poly_t A*, **const** *acb_poly_t B*, *slong n*, *slong prec*)

Sets C to the product of A and B, truncated to length n. If the same variable is passed for A and B, sets C to the square of A truncated to length n.

void **_acb_poly_mul**(*acb_ptr C*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)

Sets *{C, lenA + lenB - 1}* to the product of *{A, lenA}* and *{B, lenB}*. The output is not allowed to be aliased with either of the inputs. We require $\text{lenA} \geq \text{lenB} > 0$. This function is implemented as a simple wrapper for *_acb_poly_mullow()*.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void **acb_poly_mul**(*acb_poly_t C*, **const** *acb_poly_t A1*, **const** *acb_poly_t B2*, *slong prec*)

Sets C to the product of A and B. If the same variable is passed for A and B, sets C to the square of A.

void **_acb_poly_inv_series**(*acb_ptr Qinv*, *acb_srcptr Q*, *slong Qlen*, *slong len*, *slong prec*)

Sets *{Qinv, len}* to the power series inverse of *{Q, Qlen}*. Uses Newton iteration.

void **acb_poly_inv_series**(*acb_poly_t Qinv*, **const** *acb_poly_t Q*, *slong n*, *slong prec*)

Sets *Qinv* to the power series inverse of *Q*.

void **_acb_poly_div_series**(*acb_ptr Q*, *acb_srcptr A*, *slong Alen*, *acb_srcptr B*, *slong Blen*, *slong n*, *slong prec*)

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

void **acb_poly_div_series**(*acb_poly_t Q*, **const** *acb_poly_t A*, **const** *acb_poly_t B*, *slong n*, *slong prec*)

Sets Q to the power series quotient A divided by B, truncated to length n.

void **_acb_poly_div**(*acb_ptr Q*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)

void **_acb_poly_rem**(*acb_ptr R*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)

void **_acb_poly_divrem**(*acb_ptr Q*, *acb_ptr R*, *acb_srcptr A*, *slong lenA*, *acb_srcptr B*, *slong lenB*, *slong prec*)

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 $R = f(c)$.

6.2.10 Composition

void **_acb_poly_taylor_shift_horner**(*acb_ptr* *g*, **const** *acb_t* *c*, *slong* *n*, *slong* *prec*)

void **acb_poly_taylor_shift_horner**(*acb_poly_t* *g*, **const** *acb_poly_t* *f*, **const** *acb_t* *c*, *slong* *prec*)

void **_acb_poly_taylor_shift_divconquer**(*acb_ptr* *g*, **const** *acb_t* *c*, *slong* *n*, *slong* *prec*)

void **acb_poly_taylor_shift_divconquer**(*acb_poly_t* *g*, **const** *acb_poly_t* *f*, **const** *acb_t* *c*, *slong* *prec*)

void **_acb_poly_taylor_shift_convolution**(*acb_ptr* *g*, **const** *acb_t* *c*, *slong* *n*, *slong* *prec*)

void **acb_poly_taylor_shift_convolution**(*acb_poly_t* *g*, **const** *acb_poly_t* *f*, **const** *acb_t* *c*, *slong* *prec*)

void **_acb_poly_taylor_shift**(*acb_ptr* *g*, **const** *acb_t* *c*, *slong* *n*, *slong* *prec*)

void **acb_poly_taylor_shift**(*acb_poly_t* *g*, **const** *acb_poly_t* *f*, **const** *acb_t* *c*, *slong* *prec*)

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

The underscore methods act in-place on $g = f$ which has length n.

void **_acb_poly_compose_horner**(*acb_ptr* *res*, *acb_srcptr* *poly1*, *slong* *len1*, *acb_srcptr* *poly2*, *slong* *len2*, *slong* *prec*)

void **acb_poly_compose_horner**(*acb_poly_t* *res*, **const** *acb_poly_t* *poly1*, **const** *acb_poly_t* *poly2*, *slong* *prec*)

void **_acb_poly_compose_divconquer**(*acb_ptr* *res*, *acb_srcptr* *poly1*, *slong* *len1*, *acb_srcptr* *poly2*, *slong* *len2*, *slong* *prec*)

void **acb_poly_compose_divconquer**(*acb_poly_t* *res*, **const** *acb_poly_t* *poly1*, **const** *acb_poly_t* *poly2*, *slong* *prec*)

void **_acb_poly_compose**(*acb_ptr* *res*, *acb_srcptr* *poly1*, *slong* *len1*, *acb_srcptr* *poly2*, *slong* *len2*, *slong* *prec*)

void **acb_poly_compose**(*acb_poly_t* *res*, **const** *acb_poly_t* *poly1*, **const** *acb_poly_t* *poly2*, *slong* *prec*)

Sets *res* to the composition $h(x) = f(g(x))$ where f is given by *poly1* and g is given by *poly2*, respectively using Horner's rule, divide-and-conquer, and an automatic choice between the two algorithms.

The default algorithm also handles special-form input $g = ax^n + c$ efficiently by performing a Taylor shift followed by a rescaling.

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

void **_acb_poly_compose_series_horner**(*acb_ptr* *res*, *acb_srcptr* *poly1*, *slong* *len1*, *acb_srcptr* *poly2*, *slong* *len2*, *slong* *n*, *slong* *prec*)

void **acb_poly_compose_series_horner**(*acb_poly_t* *res*, **const** *acb_poly_t* *poly1*, **const** *acb_poly_t* *poly2*, *slong* *n*, *slong* *prec*)

void **_acb_poly_compose_series_brent_kung**(*acb_ptr* *res*, *acb_srcptr* *poly1*, *slong* *len1*, *acb_srcptr* *poly2*, *slong* *len2*, *slong* *n*, *slong* *prec*)

void **acb_poly_compose_series_brent_kung**(*acb_poly_t* *res*, **const** *acb_poly_t* *poly1*, **const** *acb_poly_t* *poly2*, *slong* *n*, *slong* *prec*)

void **_acb_poly_compose_series**(*acb_ptr* *res*, *acb_srcptr* *poly1*, *slong* *len1*, *acb_srcptr* *poly2*, *slong* *len2*, *slong* *n*, *slong* *prec*)

void **acb_poly_compose_series**(*acb_poly_t* *res*, **const** *acb_poly_t* *poly1*, **const** *acb_poly_t* *poly2*, *slong* *n*, *slong* *prec*)

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

The default algorithm also handles special-form input $g = ax^n$ efficiently.

We require that the constant term in $g(x)$ is exactly zero. The underscore methods do not support aliasing of the output with either input polynomial.

void **_acb_poly_revert_series_lagrange**(*acb_ptr* *h*, *acb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **acb_poly_revert_series_lagrange**(*acb_poly_t* *h*, **const** *acb_poly_t* *f*, *slong* *n*, *slong* *prec*)

void **_acb_poly_revert_series_newton**(*acb_ptr* *h*, *acb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **acb_poly_revert_series_newton**(*acb_poly_t* *h*, **const** *acb_poly_t* *f*, *slong* *n*, *slong* *prec*)

void **_acb_poly_revert_series_lagrange_fast**(*acb_ptr* *h*, *acb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **acb_poly_revert_series_lagrange_fast**(*acb_poly_t* *h*, **const** *acb_poly_t* *f*, *slong* *n*, *slong* *prec*)

void **_acb_poly_revert_series**(*acb_ptr* *h*, *acb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void **acb_poly_revert_series**(*acb_poly_t* *h*, **const** *acb_poly_t* *f*, *slong* *n*, *slong* *prec*)
Sets h to the power series reversion of f, i.e. the expansion of the compositional inverse function $f^{-1}(x)$, truncated to order $O(x^n)$, using respectively Lagrange inversion, Newton iteration, fast Lagrange inversion, and a default algorithm choice.

We require that the constant term in f is exactly zero and that the linear term is nonzero. The underscore methods assume that *flen* is at least 2, and do not support aliasing.

6.2.11 Evaluation

void **_acb_poly_evaluate_horner**(*acb_t* *y*, *acb_srcptr* *f*, *slong* *len*, **const** *acb_t* *x*, *slong* *prec*)

void **acb_poly_evaluate_horner**(*acb_t* *y*, **const** *acb_poly_t* *f*, **const** *acb_t* *x*, *slong* *prec*)

void **_acb_poly_evaluate_rectangular**(*acb_t* *y*, *acb_srcptr* *f*, *slong* *len*, **const** *acb_t* *x*, *slong* *prec*)

void **acb_poly_evaluate_rectangular**(*acb_t* *y*, **const** *acb_poly_t* *f*, **const** *acb_t* *x*, *slong* *prec*)

void **_acb_poly_evaluate**(*acb_t* *y*, *acb_srcptr* *f*, *slong* *len*, **const** *acb_t* *x*, *slong* *prec*)

void **acb_poly_evaluate**(*acb_t* *y*, **const** *acb_poly_t* *f*, **const** *acb_t* *x*, *slong* *prec*)
Sets $y = f(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

void **_acb_poly_evaluate2_horner**(*acb_t* *y*, *acb_t* *z*, *acb_srcptr* *f*, *slong* *len*, **const** *acb_t* *x*, *slong* *prec*)

void **acb_poly_evaluate2_horner**(*acb_t* *y*, *acb_t* *z*, **const** *acb_poly_t* *f*, **const** *acb_t* *x*, *slong* *prec*)

void **_acb_poly_evaluate2_rectangular**(*acb_t* *y*, *acb_t* *z*, *acb_srcptr* *f*, *slong* *len*, **const** *acb_t* *x*, *slong* *prec*)

void **acb_poly_evaluate2_rectangular**(*acb_t* *y*, *acb_t* *z*, **const** *acb_poly_t* *f*, **const** *acb_t* *x*, *slong* *prec*)

void **_acb_poly_evaluate2**(*acb_t* *y*, *acb_t* *z*, *acb_srcptr* *f*, *slong* *len*, **const** *acb_t* *x*, *slong* *prec*)

void **acb_poly_evaluate2**(*acb_t* *y*, *acb_t* *z*, **const** *acb_poly_t* *f*, **const** *acb_t* *x*, *slong* *prec*)
Sets $y = f(x), z = f'(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

When Horner's rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.

6.2.12 Product trees

void **_acb_poly_product_roots**(*acb_ptr poly*, *acb_srcptr xs*, *slong n*, *slong prec*)

void **acb_poly_product_roots**(*acb_poly_t poly*, *acb_srcptr xs*, *slong n*, *slong prec*)
Generates the polynomial $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$.

acb_ptr ***_acb_poly_tree_alloc**(*slong len*)
Returns an initialized data structured capable of representing a remainder tree (product tree) of *len* roots.

void **_acb_poly_tree_free**(*acb_ptr *tree*, *slong len*)
Deallocates a tree structure as allocated using *_acb_poly_tree_alloc*.

void **_acb_poly_tree_build**(*acb_ptr *tree*, *acb_srcptr roots*, *slong len*, *slong prec*)
Constructs a product tree from a given array of *len* roots. The tree structure must be pre-allocated to the specified length using *_acb_poly_tree_alloc()*.

6.2.13 Multipoint evaluation

void **_acb_poly_evaluate_vec_iter**(*acb_ptr ys*, *acb_srcptr poly*, *slong plen*, *acb_srcptr xs*, *slong n*, *slong prec*)

void **acb_poly_evaluate_vec_iter**(*acb_ptr ys*, **const** *acb_poly_t poly*, *acb_srcptr xs*, *slong n*, *slong prec*)
Evaluates the polynomial simultaneously at n given points, calling *_acb_poly_evaluate()* repeatedly.

void **_acb_poly_evaluate_vec_fast_precomp**(*acb_ptr vs*, *acb_srcptr poly*, *slong plen*, *acb_ptr *tree*, *slong len*, *slong prec*)

void **_acb_poly_evaluate_vec_fast**(*acb_ptr ys*, *acb_srcptr poly*, *slong plen*, *acb_srcptr xs*, *slong n*, *slong prec*)

void **acb_poly_evaluate_vec_fast**(*acb_ptr ys*, **const** *acb_poly_t poly*, *acb_srcptr xs*, *slong n*, *slong prec*)
Evaluates the polynomial simultaneously at n given points, using fast multipoint evaluation.

6.2.14 Interpolation

void **_acb_poly_interpolate_newton**(*acb_ptr poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong n*, *slong prec*)

void **acb_poly_interpolate_newton**(*acb_poly_t poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong n*, *slong prec*)
Recovers the unique polynomial of length at most n that interpolates the given x and y values. This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

void **_acb_poly_interpolate_barycentric**(*acb_ptr poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong n*, *slong prec*)

void **acb_poly_interpolate_barycentric**(*acb_poly_t poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong n*, *slong prec*)
Recovers the unique polynomial of length at most n that interpolates the given x and y values. This implementation uses the barycentric form of Lagrange interpolation.

void **_acb_poly_interpolation_weights**(*acb_ptr w*, *acb_ptr *tree*, *slong len*, *slong prec*)

void **_acb_poly_interpolate_fast_precomp**(*acb_ptr poly*, *acb_srcptr ys*, *acb_ptr *tree*, *acb_srcptr weights*, *slong len*, *slong prec*)

void **_acb_poly_interpolate_fast**(*acb_ptr poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong len*, *slong prec*)

void **acb_poly_interpolate_fast**(*acb_poly_t poly*, *acb_srcptr xs*, *acb_srcptr ys*, *slong n*, *slong prec*)
Recovers the unique polynomial of length at most n that interpolates the given x and y values, using fast Lagrange interpolation. The precomp function takes a precomputed product tree over the x values and a vector of interpolation weights as additional inputs.

6.2.15 Differentiation

void **_acb_poly_derivative**(*acb_ptr res*, *acb_srcptr poly*, *slong len*, *slong prec*)
Sets *{res, len - 1}* to the derivative of *{poly, len}*. Allows aliasing of the input and output.

void **acb_poly_derivative**(*acb_poly_t res*, **const** *acb_poly_t poly*, *slong prec*)
Sets *res* to the derivative of *poly*.

void **_acb_poly_integral**(*acb_ptr res*, *acb_srcptr poly*, *slong len*, *slong prec*)
Sets *{res, len}* to the integral of *{poly, len - 1}*. Allows aliasing of the input and output.

void **acb_poly_integral**(*acb_poly_t res*, **const** *acb_poly_t poly*, *slong prec*)
Sets *res* to the integral of *poly*.

6.2.16 Transforms

void **_acb_poly_borel_transform**(*acb_ptr res*, *acb_srcptr poly*, *slong len*, *slong prec*)

void **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 **_acb_poly_inv_borel_transform**(*acb_ptr res*, *acb_srcptr poly*, *slong len*, *slong prec*)

void **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.

void **_acb_poly_binomial_transform_basecase**(*acb_ptr b*, *acb_srcptr a*, *slong alen*, *slong len*, *slong prec*)

void **acb_poly_binomial_transform_basecase**(*acb_poly_t b*, **const** *acb_poly_t a*, *slong len*, *slong prec*)

void **_acb_poly_binomial_transform_convolution**(*acb_ptr b*, *acb_srcptr a*, *slong alen*, *slong len*, *slong prec*)

void **acb_poly_binomial_transform_convolution**(*acb_poly_t b*, **const** *acb_poly_t a*, *slong len*, *slong prec*)

void **_acb_poly_binomial_transform**(*acb_ptr b*, *acb_srcptr a*, *slong alen*, *slong len*, *slong prec*)

void **acb_poly_binomial_transform**(*acb_poly_t b*, **const** *acb_poly_t a*, *slong len*, *slong prec*)
Computes the binomial transform of the input polynomial, truncating the output to length *len*. See *arb_poly_binomial_transform()* for details.

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

6.2.17 Elementary functions

void **_acb_poly_pow_ui_trunc_binexp**(*acb_ptr res*, *acb_srcptr f*, *slong flen*, *ulong exp*, *slong len*, *slong prec*)

Sets *{res, len}* to *{f, flen}* raised to the power *exp*, truncated to length *len*. Requires that *len* is no longer than the length of the power as computed without truncation (i.e. no zero-padding is performed). Does not support aliasing of the input and output, and requires that *flen* and *len* are positive. Uses binary expontiation.

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

Sets *res* to *poly* raised to the power *exp*, truncated to length *len*. Uses binary exponentiation.

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

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

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

Sets *res* to *poly* raised to the power *exp*.

void **_acb_poly_pow_series**(*acb_ptr h*, *acb_srcptr f*, *slong flen*, *acb_srcptr g*, *slong glen*, *slong len*, *slong prec*)

Sets *{h, len}* to the power series $f(x)^{g(x)} = \exp(g(x) \log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that *flen* and *glen* do not exceed *len*.

void **acb_poly_pow_series**(*acb_poly_t h*, **const** *acb_poly_t f*, **const** *acb_poly_t g*, *slong len*, *slong prec*)

Sets h to the power series $f(x)^{g(x)} = \exp(g(x) \log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently.

void **_acb_poly_pow_acb_series**(*acb_ptr h*, *acb_srcptr f*, *slong flen*, **const** *acb_t g*, *slong len*, *slong prec*)

Sets *{h, len}* to the power series $f(x)^g = \exp(g \log f(x))$ truncated to length *len*. This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that *flen* does not exceed *len*.

void **acb_poly_pow_acb_series**(*acb_poly_t h*, **const** *acb_poly_t f*, **const** *acb_t g*, *slong len*, *slong prec*)

Sets h to the power series $f(x)^g = \exp(g \log f(x))$ truncated to length *len*.

void **_acb_poly_sqrt_series**(*acb_ptr g*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_sqrt_series**(*acb_poly_t g*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

Sets g to the power series square root of h, truncated to length n. Uses division-free Newton iteration for the reciprocal square root, followed by a multiplication.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and n are greater than zero.

void **_acb_poly_rsqrt_series**(*acb_ptr g*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_rsqrt_series**(*acb_poly_t g*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

Sets g to the reciprocal power series square root of h, truncated to length n. Uses division-free Newton iteration.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and n are greater than zero.

void **_acb_poly_log_series**(*acb_ptr res*, *acb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **acb_poly_log_series**(*acb_poly_t res*, **const** *acb_poly_t f*, *slong n*, *slong prec*)
Sets *res* to the power series logarithm of *f*, truncated to length *n*. Uses the formula $\log(f(x)) = \int f'(x)/f(x)dx$, adding the logarithm of the constant term in *f* as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that *flen* and *n* are greater than zero.

void **_acb_poly_log1p_series**(*acb_ptr res*, *acb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **acb_poly_log1p_series**(*acb_poly_t res*, **const** *acb_poly_t f*, *slong n*, *slong prec*)
Computes the power series $\log(1 + f)$, with better accuracy when the constant term of *f* is small.

void **_acb_poly_atan_series**(*acb_ptr res*, *acb_srcptr f*, *slong flen*, *slong n*, *slong prec*)

void **acb_poly_atan_series**(*acb_poly_t res*, **const** *acb_poly_t f*, *slong n*, *slong prec*)
Sets *res* the power series inverse tangent of *f*, truncated to length *n*.

Uses the formula

$$\tan^{-1}(f(x)) = \int f'(x)/(1 + f(x)^2)dx,$$

adding the function of the constant term in *f* as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that *flen* and *n* are greater than zero.

void **_acb_poly_exp_series_basecase**(*acb_ptr f*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_exp_series_basecase**(*acb_poly_t f*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_exp_series**(*acb_ptr f*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_exp_series**(*acb_poly_t f*, **const** *acb_poly_t h*, *slong n*, *slong prec*)
Sets *f* to the power series exponential of *h*, truncated to length *n*.

The basecase version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where *m* is the length of *h*.

The main implementation uses Newton iteration, starting from a small number of terms given by the basecase algorithm. The complexity is $O(M(n))$. Redundant operations in the Newton iteration are avoided by using the scheme described in [HZ2004].

The underscore methods support aliasing and allow the input to be shorter than the output, but require the lengths to be nonzero.

void **_acb_poly_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*)
Sets *s* and *c* to the power series sine and cosine of *h*, computed simultaneously.

The *basecase* version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where m is the length of h.

The *tangent* version uses the tangent half-angle formulas to compute the sine and cosine via `_acb_poly_tan_series()`. This requires $O(M(n))$ operations. When $h = h_0 + h_1$ where the constant term h_0 is nonzero, the evaluation is done as $\sin(h_0 + h_1) = \cos(h_0)\sin(h_1) + \sin(h_0)\cos(h_1)$, $\cos(h_0 + h_1) = \cos(h_0)\cos(h_1) - \sin(h_0)\sin(h_1)$, to improve accuracy and avoid dividing by zero at the poles of the tangent function.

The default version automatically selects between the *basecase* and *tangent* algorithms depending on the input.

The *basecase* and *tangent* versions take a flag *times_pi* specifying that the input is to be multiplied by π.

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

void **_acb_poly_sin_series**(*acb_ptr* *s*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sin_series**(*acb_poly_t* *s*, **const** *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_acb_poly_cos_series**(*acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_cos_series**(*acb_poly_t* *c*, **const** *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Respectively evaluates the power series sine or cosine. These functions simply wrap `_acb_poly_sin_cos_series()`. The underscore methods support aliasing and require the lengths to be nonzero.

void **_acb_poly_tan_series**(*acb_ptr* *g*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *len*, *slong* *prec*)

void **acb_poly_tan_series**(*acb_poly_t* *g*, **const** *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Sets g to the power series tangent of h.

For small n takes the quotient of the sine and cosine as computed using the basecase algorithm. For large n, uses Newton iteration to invert the inverse tangent series. The complexity is $O(M(n))$.

The underscore version does not support aliasing, and requires the lengths to be nonzero.

void **_acb_poly_sin_cos_pi_series**(*acb_ptr* *s*, *acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sin_cos_pi_series**(*acb_poly_t* *s*, *acb_poly_t* *c*, **const** *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_acb_poly_sin_pi_series**(*acb_ptr* *s*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sin_pi_series**(*acb_poly_t* *s*, **const** *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_acb_poly_cos_pi_series**(*acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_cos_pi_series**(*acb_poly_t* *c*, **const** *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_acb_poly_cot_pi_series**(*acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_cot_pi_series**(*acb_poly_t* *c*, **const** *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Compute the respective trigonometric functions of the input multiplied by π.

void **_acb_poly_sinh_cosh_series_basecase**(*acb_ptr* *s*, *acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sinh_cosh_series_basecase**(*acb_poly_t* *s*, *acb_poly_t* *c*, **const** *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_acb_poly_sinh_cosh_series_exponential**(*acb_ptr* *s*, *acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sinh_cosh_series_exponential**(*acb_poly_t* *s*, *acb_poly_t* *c*, **const** *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void **_acb_poly_sinh_cosh_series**(*acb_ptr* *s*, *acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sinh_cosh_series**(*acb_poly_t s*, *acb_poly_t c*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_sinh_series**(*acb_ptr s*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_sinh_series**(*acb_poly_t s*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_cosh_series**(*acb_ptr c*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_cosh_series**(*acb_poly_t c*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

Sets s and c respectively to the hyperbolic sine and cosine of the power series h, truncated to length n.

The implementations mirror those for sine and cosine, except that the *exponential* version computes both functions using the exponential function instead of the hyperbolic tangent.

void **_acb_poly_sinc_series**(*acb_ptr s*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_sinc_series**(*acb_poly_t s*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

Sets s to the sinc function of the power series h, truncated to length n.

6.2.18 Lambert W function

void **_acb_poly_lambertw_series**(*acb_ptr res*, *acb_srcptr 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.

6.2.19 Gamma function

void **_acb_poly_gamma_series**(*acb_ptr res*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_gamma_series**(*acb_poly_t res*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_rgamma_series**(*acb_ptr res*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_rgamma_series**(*acb_poly_t res*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_lgamma_series**(*acb_ptr res*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_lgamma_series**(*acb_poly_t res*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

void **_acb_poly_digamma_series**(*acb_ptr res*, *acb_srcptr h*, *slong hlen*, *slong n*, *slong prec*)

void **acb_poly_digamma_series**(*acb_poly_t res*, **const** *acb_poly_t h*, *slong n*, *slong prec*)

Sets *res* to the series expansion of $\Gamma(h(x))$, $1/\Gamma(h(x))$, or $\log\Gamma(h(x))$, $\psi(h(x))$, truncated to length n.

These functions first generate the Taylor series at the constant term of h, and then call *_acb_poly_compose_series()*. The Taylor coefficients are generated using Stirling's series.

The underscore methods support aliasing of the input and output arrays, and require that *hlen* and n are greater than zero.

void **_acb_poly_rising_ui_series**(*acb_ptr res*, *acb_srcptr f*, *slong flen*, *ulong r*, *slong trunc*, *slong prec*)

void **acb_poly_rising_ui_series**(*acb_poly_t res*, **const** *acb_poly_t f*, *ulong r*, *slong trunc*, *slong prec*)

Sets *res* to the rising factorial $(f)(f+1)(f+2)\cdots(f+r-1)$, truncated to length *trunc*. The underscore method assumes that *flen*, r and *trunc* are at least 1, and does not support aliasing. Uses binary splitting.

6.2.20 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.

6.2.21 Zeta function

void **_acb_poly_zeta_em_choose_param**(*mag_t bound*, *ulong* *N, *ulong* *M, **const** *acb_t s*, **const** *acb_t a*, *slong d*, *slong target*, *slong prec*)

Chooses N and M for Euler-Maclaurin summation of the Hurwitz zeta function, using a default algorithm.

void **_acb_poly_zeta_em_bound1**(*mag_t bound*, **const** *acb_t s*, **const** *acb_t a*, *slong N*, *slong M*, *slong d*, *slong wp*)

void **_acb_poly_zeta_em_bound**(*arb_ptr vec*, **const** *acb_t s*, **const** *acb_t a*, *ulong N*, *ulong M*, *slong d*, *slong wp*)

Compute bounds for Euler-Maclaurin evaluation of the Hurwitz zeta function or its power series, using the formulas in [Joh2013].

void **_acb_poly_zeta_em_tail_naive**(*acb_ptr z*, **const** *acb_t s*, **const** *acb_t Na*, *acb_srcptr Nasx*, *slong M*, *slong len*, *slong prec*)

void **_acb_poly_zeta_em_tail_bsplit**(*acb_ptr z*, **const** *acb_t s*, **const** *acb_t Na*, *acb_srcptr Nasx*, *slong M*, *slong len*, *slong prec*)

Evaluates the tail in the Euler-Maclaurin sum for the Hurwitz zeta function, respectively using the naive recurrence and binary splitting.

void **_acb_poly_zeta_em_sum**(*acb_ptr z*, **const** *acb_t s*, **const** *acb_t a*, int *deflate*, *ulong N*, *ulong M*, *slong d*, *slong prec*)

Evaluates the truncated Euler-Maclaurin sum of order N, M for the length-d truncated Taylor series of the Hurwitz zeta function $\zeta(s, a)$ at s, using a working precision of *prec* bits. With $a = 1$, this gives the usual Riemann zeta function.

If *deflate* is nonzero, $\zeta(s, a) - 1/(s-1)$ is evaluated (which permits series expansion at $s = 1$).

void **_acb_poly_zeta_cpx_series**(*acb_ptr z*, **const** *acb_t s*, **const** *acb_t a*, int *deflate*, *slong d*, *slong prec*)

Computes the series expansion of $\zeta(s+x,a)$ (or $\zeta(s+x,a)-1/(s+x-1)$ if *deflate* is nonzero) to order d.

This function wraps *_acb_poly_zeta_em_sum()*, automatically choosing default values for N, M using *_acb_poly_zeta_em_choose_param()* to target an absolute truncation error of $2^{-\text{prec}}$.

void **_acb_poly_zeta_series**(*acb_ptr res*, *acb_srcptr h*, *slong hlen*, **const** *acb_t a*, int *deflate*, *slong len*, *slong prec*)

void **acb_poly_zeta_series**(*acb_poly_t res*, **const** *acb_poly_t f*, **const** *acb_t a*, int *deflate*, *slong n*, *slong prec*)

Sets *res* to the Hurwitz zeta function $\zeta(s,a)$ where s a power series and a is a constant, truncated to length n. To evaluate the usual Riemann zeta function, set $a = 1$.

If *deflate* is nonzero, evaluates $\zeta(s,a)+1/(1-s)$, which is well-defined as a limit when the constant term of s is 1. In particular, expanding $\zeta(s,a)+1/(1-s)$ with $s = 1+x$ gives the Stieltjes constants

$$\sum_{k=0}^{n-1} \frac{(-1)^k}{k!} \gamma_k(a) x^k.$$

If $a = 1$, this implementation uses the reflection formula if the midpoint of the constant term of s is negative.

6.2.22 Other special functions

void **_acb_poly_polylog_cpx_small**(*acb_ptr w*, **const** *acb_t s*, **const** *acb_t z*, *slong len*, *slong prec*)

void **_acb_poly_polylog_cpx_zeta**(*acb_ptr w*, **const** *acb_t s*, **const** *acb_t z*, *slong len*, *slong prec*)

void **_acb_poly_polylog_cpx**(*acb_ptr w*, **const** *acb_t s*, **const** *acb_t z*, *slong len*, *slong prec*)

Sets w to the Taylor series with respect to x of the polylogarithm $\mathrm{Li}_{s+x}(z)$, where s and z are given complex constants. The output is computed to length *len* which must be positive. Aliasing between w and s or z is not permitted.

The *small* version uses the standard power series expansion with respect to z, convergent when $|z| < 1$. The *zeta* version evaluates the polylogarithm as a sum of two Hurwitz zeta functions. The default version automatically delegates to the *small* version when z is close to zero, and the *zeta* version otherwise. For further details, see *Algorithms for polylogarithms*.

void **_acb_poly_polylog_series**(*acb_ptr w*, *acb_srcptr s*, *slong slen*, **const** *acb_t z*, *slong len*, *slong prec*)

void **acb_poly_polylog_series**(*acb_poly_t w*, **const** *acb_poly_t s*, **const** *acb_t z*, *slong len*, *slong prec*)

Sets w to the polylogarithm $\mathrm{Li}_s(z)$ where s is a given power series, truncating the output to length *len*. The underscore method requires all lengths to be positive and supports aliasing between all inputs and outputs.

void **_acb_poly_erf_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong n*, *slong prec*)

void **acb_poly_erf_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong n*, *slong prec*)

Sets *res* to the error function of the power series z, truncated to length n. These methods are provided for backwards compatibility. See *acb_hypgeom_erf_series()*, *acb_hypgeom_erfc_series()*, *acb_hypgeom_erfi_series()*.

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

void **acb_poly_agm1_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong n*, *slong prec*)

Sets *res* to the arithmetic-geometric mean of 1 and the power series z, truncated to length n.

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

void **_acb_poly_elliptic_k_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_poly_elliptic_k_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong n*, *slong prec*)

void **_acb_poly_elliptic_p_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, **const** *acb_t tau*, *slong len*, *slong prec*)

void **acb_poly_elliptic_p_series**(*acb_poly_t res*, **const** *acb_poly_t z*, **const** *acb_t tau*, *slong n*, *slong prec*)

6.2.23 Root-finding

void **_acb_poly_root_bound_fujiwara**(*mag_t bound*, *acb_srcptr poly*, *slong len*)

void **acb_poly_root_bound_fujiwara**(*mag_t bound*, *acb_poly_t poly*)

Sets *bound* to an upper bound for the magnitude of all the complex roots of *poly*. Uses Fujiwara's bound

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

where $a_0, \ldots, a_n$ are the coefficients of *poly*.

void **_acb_poly_root_inclusion**(*acb_t r*, **const** *acb_t m*, *acb_srcptr poly*, *acb_srcptr polyder*, *slong len*, *slong prec*)

Given any complex number m, and a nonconstant polynomial f and its derivative f', sets r to a complex interval centered on m that is guaranteed to contain at least one root of f. Such an interval is obtained by taking a ball of radius $|f(m)/f'(m)|n$ where n is the degree of f. Proof: assume that the distance to the nearest root exceeds $r = |f(m)/f'(m)|n$. Then

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

which is a contradiction (see [Kob2010]).

slong **_acb_poly_validate_roots**(*acb_ptr roots*, *acb_srcptr poly*, *slong len*, *slong prec*)

Given a list of approximate roots of the input polynomial, this function sets a rigorous bounding interval for each root, and determines which roots are isolated from all the other roots. It then rearranges the list of roots so that the isolated roots are at the front of the list, and returns the count of isolated roots.

If the return value equals the degree of the polynomial, then all roots have been found. If the return value is smaller, all the remaining output intervals are guaranteed to contain roots, but it is possible that not all of the polynomial's roots are contained among them.

void **_acb_poly_refine_roots_durand_kerner**(*acb_ptr roots*, *acb_srcptr poly*, *slong len*, *slong prec*)

Refines the given roots simultaneously using a single iteration of the Durand-Kerner method. The radius of each root is set to an approximation of the correction, giving a rough estimate of its error (not a rigorous bound).

slong **_acb_poly_find_roots**(*acb_ptr roots*, *acb_srcptr poly*, *acb_srcptr initial*, *slong len*, *slong maxiter*, *slong prec*)

slong **acb_poly_find_roots**(*acb_ptr roots*, **const** *acb_poly_t poly*, *acb_srcptr initial*, *slong maxiter*, *slong prec*)

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

If the return value equals the degree of the polynomial, then all roots have been found. If the return value is smaller, all the output intervals are guaranteed to contain roots, but it is possible that not all of the polynomial's roots are contained among them.

The roots are computed numerically by performing several steps with the Durand-Kerner method and terminating if the estimated accuracy of the roots approaches the working precision or if the number of steps exceeds *maxiter*, which can be set to zero in order to use a default value. Finally, the approximate roots are validated rigorously.

Initial values for the iteration can be provided as the array *initial*. If *initial* is set to *NULL*, default values $(0.4 + 0.9i)^k$ are used.

The polynomial is assumed to be squarefree. If there are repeated roots, the iteration is likely to find them (with low numerical accuracy), but the error bounds will not converge as the precision increases.

int **_acb_poly_validate_real_roots**(*acb_srcptr roots*, *acb_srcptr poly*, *slong len*, *slong prec*)

int **acb_poly_validate_real_roots**(*acb_srcptr roots*, **const** *acb_poly_t poly*, *slong prec*)

Given a strictly real polynomial *poly* (of length *len*) and isolating intervals for all its complex roots, determines if all the real roots are separated from the non-real roots. If this function returns nonzero, every root enclosure that touches the real axis (as tested by applying *arb_contains_zero()* to the imaginary part) corresponds to a real root (its imaginary part can be set to zero), and every other root enclosure corresponds to a non-real root (with known sign for the imaginary part).

If this function returns zero, then the signs of the imaginary parts are not known for certain, based on the accuracy of the inputs and the working precision *prec*.

6.3 arb_fmpz_poly.h – extra methods for integer polynomials

This module provides methods for FLINT polynomials with integer and rational coefficients (*fmpz_poly_t*) and (*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 *arb_poly_set_fmpz_poly()* and *arb_poly_get_unique_fmpz_poly()*.

6.3.1 Evaluation

void **_arb_fmpz_poly_evaluate_arb_horner**(*arb_t res*, **const** *fmpz* *poly, *slong len*, **const** *arb_t x*, *slong prec*)

void **arb_fmpz_poly_evaluate_arb_horner**(*arb_t res*, **const** *fmpz_poly_t poly*, **const** *arb_t x*, *slong prec*)

void **_arb_fmpz_poly_evaluate_arb_rectangular**(*arb_t res*, **const** *fmpz* *poly, *slong len*, **const** *arb_t x*, *slong prec*)

void **arb_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 **arb_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 **arb_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 **arb_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 **arb_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.

6.3.2 Utility methods

ulong **arb_fmpz_poly_deflation**(**const** *fmpz_poly_t poly*)

Finds the maximal exponent by which *poly* can be deflated.

void **arb_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*.

6.3.3 Polynomial roots

void **arb_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:

```
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 *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:

- *ARB_FMPZ_POLY_ROOTS_VERBOSE*

6.3.4 Special polynomials

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

void **arb_fmpz_poly_cos_minpoly**(*fmpz_poly_t res*, *ulong n*)
Sets *res* to the monic minimal polynomial of $2\cos(2\pi/n)$. This is a wrapper of FLINT's *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 *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 Φ_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, *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 $\mathrm{GF}(p)$ [CP2005].

CHAPTER

SEVEN

TRANSFORMS

7.1 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 χ a character of G. For any map $f : G \to \mathbb{C}$, the discrete fourier transform $\hat{f} : \hat{G} \to \mathbb{C}$ is defined by

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

Note that by the inversion formula

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

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

7.1.1 Main DFT functions

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

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

void **acb_dft**(*acb_ptr* *w*, *acb_srcptr* *v*, *slong* *n*, *slong* *prec*)
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*)
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.

type acb_dft_pre_struct

type acb_dft_pre_t
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*)
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*)
Compute the inverse DFT of v into w.

7.1.2 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

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

We assume that f is given by a vector of length $\prod n_i$ corresponding to a lexicographic ordering of the values $y_1, \dots y_r$, 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.

type acb_dft_prod_struct

type 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.

7.1.3 Convolution

For functions f and g on G we consider the convolution

$$(f \star g)(x) = \sum_{y \in G} f(x - y) g(y)$$

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*)

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 *naive* version simply uses the definition.

The *rad2* version embeds the sequence into a power of 2 length and uses the formula

$$\widehat{f \star g}(\chi) = \hat{f}(\chi)\hat{g}(\chi)$$

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.

7.1.4 FFT algorithms

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

Specifically, if $H \triangleleft G$ 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 χ_H.

This decomposition can be done recursively.

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.

type acb_dft_naive_struct

type 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*.

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.

type acb_dft_crt_struct

type 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

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 $H = m\mathbb{Z}/n\mathbb{Z}$.

type acb_dft_cyc_struct

type acb_dft_cyc_t

void **acb_dft_cyc_init**(*acb_dft_cyc_t* *t*, *slong* *len*, *slong* *prec*)

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

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

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.

type acb_dft_rad2_struct

type 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

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.

type acb_dft_bluestein_struct

type 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*.

CHAPTER

EIGHT

MATRICES

These modules implement dense matrices with real and complex coefficients. Rudimentary linear algebra is supported.

8.1 arb_mat.h – matrices over the real numbers

An *arb_mat_t* represents a dense matrix over the real numbers, implemented as an array of entries of type *arb_struct*. The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

Note: Methods prefixed with *arb_mat_approx* treat all input entries as floating-point numbers (ignoring the radii of the balls) and compute floating-point output (balls with zero radius) representing approximate solutions *without error bounds*. All other methods compute rigorous error bounds. The *approx* methods are typically useful for computing initial values or preconditioners for rigorous solvers. Some users may also find *approx* methods useful for doing ordinary numerical linear algebra in applications where error bounds are not needed.

8.1.1 Types, macros and constants

type arb_mat_struct

type 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.

8.1.2 Memory management

void **arb_mat_init**(*arb_mat_t mat*, *slong r*, *slong c*)
Initializes the matrix, setting it to the zero matrix with r rows and c columns.

void **arb_mat_clear**(*arb_mat_t mat*)
Clears the matrix, deallocating all entries.

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.

void **arb_mat_window_init**(*arb_mat_t window*, **const** *arb_mat_t mat*, *slong r1*, *slong c1*, *slong r2*, *slong c2*)
Initializes *window* to a window matrix into the submatrix of *mat* starting at the corner at row *r1* and column *c1* (inclusive) and ending at row *r2* and column *c2* (exclusive).

void **arb_mat_window_clear**(*arb_mat_t window*)
Frees the window matrix.

8.1.3 Conversions

void **arb_mat_set**(*arb_mat_t dest*, **const** *arb_mat_t src*)

void **arb_mat_set_fmpz_mat**(*arb_mat_t dest*, **const** *fmpz_mat_t src*)

void **arb_mat_set_round_fmpz_mat**(*arb_mat_t dest*, **const** *fmpz_mat_t src*, *slong prec*)

void **arb_mat_set_fmpq_mat**(*arb_mat_t dest*, **const** *fmpq_mat_t src*, *slong prec*)
Sets *dest* to *src*. The operands must have identical dimensions.

8.1.4 Random generation

void **arb_mat_randtest**(*arb_mat_t mat*, flint_rand_t *state*, *slong prec*, *slong mag_bits*)
Sets *mat* to a random matrix with up to *prec* bits of precision and with exponents of width up to *mag_bits*.

8.1.5 Input and output

void **arb_mat_printd**(**const** *arb_mat_t mat*, *slong digits*)
Prints each entry in the matrix with the specified number of decimal digits.

void **arb_mat_fprintd**(FILE ***file*, **const** *arb_mat_t mat*, *slong digits*)
Prints each entry in the matrix with the specified number of decimal digits to the stream *file*.

8.1.6 Comparisons

Predicate methods return 1 if the property certainly holds and 0 otherwise.

int **arb_mat_equal**(**const** *arb_mat_t mat1*, **const** *arb_mat_t mat2*)
Returns whether the matrices have the same dimensions and identical intervals as entries.

int **arb_mat_overlaps**(**const** *arb_mat_t mat1*, **const** *arb_mat_t mat2*)
Returns whether the matrices have the same dimensions and each entry in *mat1* overlaps with the corresponding entry in *mat2*.

int **arb_mat_contains**(**const** *arb_mat_t mat1*, **const** *arb_mat_t mat2*)

int **arb_mat_contains_fmpz_mat**(**const** *arb_mat_t mat1*, **const** *fmpz_mat_t mat2*)

int **arb_mat_contains_fmpq_mat**(**const** *arb_mat_t mat1*, **const** *fmpq_mat_t mat2*)
Returns whether the matrices have the same dimensions and each entry in *mat2* is contained in the corresponding entry in *mat1*.

int **arb_mat_eq**(**const** *arb_mat_t mat1*, **const** *arb_mat_t mat2*)
Returns whether *mat1* and *mat2* certainly represent the same matrix.

int **arb_mat_ne**(**const** *arb_mat_t mat1*, **const** *arb_mat_t mat2*)
Returns whether *mat1* and *mat2* certainly do not represent the same matrix.

int **arb_mat_is_empty**(**const** *arb_mat_t mat*)
Returns whether the number of rows or the number of columns in *mat* is zero.

int **arb_mat_is_square**(**const** *arb_mat_t mat*)
Returns whether the number of rows is equal to the number of columns in *mat*.

int **arb_mat_is_exact**(**const** *arb_mat_t mat*)
Returns whether all entries in *mat* have zero radius.

int **arb_mat_is_zero**(**const** *arb_mat_t mat*)
Returns whether all entries in *mat* are exactly zero.

int **arb_mat_is_finite**(**const** *arb_mat_t mat*)
Returns whether all entries in *mat* are finite.

int **arb_mat_is_triu**(**const** *arb_mat_t mat*)
Returns whether *mat* is upper triangular; that is, all entries below the main diagonal are exactly zero.

int **arb_mat_is_tril**(**const** *arb_mat_t mat*)
Returns whether *mat* is lower triangular; that is, all entries above the main diagonal are exactly zero.

int **arb_mat_is_diag**(**const** *arb_mat_t mat*)
Returns whether *mat* is a diagonal matrix; that is, all entries off the main diagonal are exactly zero.

8.1.7 Special matrices

void **arb_mat_zero**(*arb_mat_t mat*)
Sets all entries in mat to zero.

void **arb_mat_one**(*arb_mat_t mat*)
Sets the entries on the main diagonal to ones, and all other entries to zero.

void **arb_mat_ones**(*arb_mat_t mat*)
Sets all entries in the matrix to ones.

void **arb_mat_indeterminate**(*arb_mat_t mat*)
Sets all entries in the matrix to indeterminate (NaN).

void **arb_mat_hilbert**(*arb_mat_t mat*)
Sets *mat* to the Hilbert matrix, which has entries $A_{j,k} = 1/(j + k + 1)$.

void **arb_mat_pascal**(*arb_mat_t mat*, int *triangular*, *slong prec*)
Sets *mat* to a Pascal matrix, whose entries are binomial coefficients. If *triangular* is 0, constructs a full symmetric matrix with the rows of Pascal's triangle as successive antidiagonals. If *triangular* is 1, constructs the upper triangular matrix with the rows of Pascal's triangle as columns, and if *triangular* is -1, constructs the lower triangular matrix with the rows of Pascal's triangle as rows.

The entries are computed using recurrence relations. When the dimensions get large, some precision loss is possible; in that case, the user may wish to create the matrix at slightly higher precision and then round it to the final precision.

void **arb_mat_stirling**(*arb_mat_t mat*, int *kind*, *slong prec*)
Sets *mat* to a Stirling matrix, whose entries are Stirling numbers. If *kind* is 0, the entries are set to the unsigned Stirling numbers of the first kind. If *kind* is 1, the entries are set to the signed Stirling numbers of the first kind. If *kind* is 2, the entries are set to the Stirling numbers of the second kind.

The entries are computed using recurrence relations. When the dimensions get large, some precision loss is possible; in that case, the user may wish to create the matrix at slightly higher precision and then round it to the final precision.

void **arb_mat_dct**(*arb_mat_t mat*, int *type*, *slong prec*)
Sets *mat* to the DCT (discrete cosine transform) matrix of order n where n is the smallest dimension of *mat* (if *mat* is not square, the matrix is extended periodically along the larger dimension). There are many different conventions for defining DCT matrices; here, we use the normalized "DCT-II" transform matrix

$$A_{j,k} = \sqrt{\frac{2}{n}} \cos\left(\frac{\pi j}{n}\left(k + \frac{1}{2}\right)\right)$$

which satisfies $A^{-1} = A^T$. The *type* parameter is currently ignored and should be set to 0. In the future, it might be used to select a different convention.

8.1.8 Transpose

void **arb_mat_transpose**(*arb_mat_t dest*, **const** *arb_mat_t src*)
Sets *dest* to the exact transpose *src*. The operands must have compatible dimensions. Aliasing is allowed.

8.1.9 Norms

void **arb_mat_bound_inf_norm**(*mag_t b*, **const** *arb_mat_t A*)
Sets b to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of A.

void **arb_mat_frobenius_norm**(*arb_t res*, **const** *arb_mat_t A*, *slong prec*)
Sets *res* to the Frobenius norm (i.e. the square root of the sum of squares of entries) of A.

void **arb_mat_bound_frobenius_norm**(*mag_t res*, **const** *arb_mat_t A*)
Sets *res* to an upper bound for the Frobenius norm of A.

8.1.10 Arithmetic

void **arb_mat_neg**(*arb_mat_t dest*, **const** *arb_mat_t src*)
Sets *dest* to the exact negation of *src*. The operands must have the same dimensions.

void **arb_mat_add**(*arb_mat_t res*, **const** *arb_mat_t mat1*, **const** *arb_mat_t mat2*, *slong prec*)
Sets res to the sum of *mat1* and *mat2*. The operands must have the same dimensions.

void **arb_mat_sub**(*arb_mat_t res*, **const** *arb_mat_t mat1*, **const** *arb_mat_t mat2*, *slong prec*)
Sets *res* to the difference of *mat1* and *mat2*. The operands must have the same dimensions.

void **arb_mat_mul_classical**(*arb_mat_t C*, **const** *arb_mat_t A*, **const** *arb_mat_t B*, *slong prec*)

void **arb_mat_mul_threaded**(*arb_mat_t C*, **const** *arb_mat_t A*, **const** *arb_mat_t B*, *slong prec*)

void **arb_mat_mul_block**(*arb_mat_t C*, **const** *arb_mat_t A*, **const** *arb_mat_t B*, *slong prec*)

void **arb_mat_mul**(*arb_mat_t res*, **const** *arb_mat_t mat1*, **const** *arb_mat_t mat2*, *slong prec*)
Sets *res* to the matrix product of *mat1* and *mat2*. The operands must have compatible dimensions for matrix multiplication.

The *classical* version performs matrix multiplication in the trivial way.

The *block* version decomposes the input matrices into one or several blocks of uniformly scaled matrices and multiplies large blocks via *fmpz_mat_mul*. It also invokes *_arb_mat_addmul_rad_mag_fast()* for the radius matrix multiplications.

The *threaded* version performs classical multiplication but splits the computation over the number of threads returned by *flint_get_num_threads()*.

The default version chooses an algorithm automatically.

void **arb_mat_mul_entrywise**(*arb_mat_t C*, **const** *arb_mat_t A*, **const** *arb_mat_t B*, *slong prec*)
Sets C to the entrywise product of A and B. The operands must have the same dimensions.

void **arb_mat_sqr_classical**(*arb_mat_t B*, **const** *arb_mat_t A*, *slong prec*)

void **arb_mat_sqr**(*arb_mat_t res*, **const** *arb_mat_t mat*, *slong prec*)
Sets *res* to the matrix square of *mat*. The operands must both be square with the same dimensions.

void **arb_mat_pow_ui**(*arb_mat_t res*, **const** *arb_mat_t mat*, *ulong exp*, *slong prec*)
Sets *res* to *mat* raised to the power *exp*. Requires that *mat* is a square matrix.

void **_arb_mat_addmul_rad_mag_fast**(*arb_mat_t C*, mag_srcptr *A*, mag_srcptr *B*, *slong ar*, *slong ac*, *slong bc*)
Helper function for matrix multiplication. Adds to the radii of C the matrix product of the matrices represented by A and B, where A is a linear array of coefficients in row-major order and B is a linear array of coefficients in column-major order. This function assumes that all exponents are small and is unsafe for general use.

void **arb_mat_approx_mul**(*arb_mat_t res*, **const** *arb_mat_t mat1*, **const** *arb_mat_t mat2*, *slong prec*)
Approximate matrix multiplication. The input radii are ignored and the output matrix is set to an approximate floating-point result. The radii in the output matrix will *not* necessarily be zeroed.

8.1.11 Scalar arithmetic

void **arb_mat_scalar_mul_2exp_si**(*arb_mat_t B*, **const** *arb_mat_t A*, *slong c*)
Sets B to A multiplied by 2^c.

void **arb_mat_scalar_addmul_si**(*arb_mat_t B*, **const** *arb_mat_t A*, *slong c*, *slong prec*)

void **arb_mat_scalar_addmul_fmpz**(*arb_mat_t B*, **const** *arb_mat_t A*, **const** *fmpz_t c*, *slong prec*)

void **arb_mat_scalar_addmul_arb**(*arb_mat_t B*, **const** *arb_mat_t A*, **const** *arb_t c*, *slong prec*)
Sets B to $B + A \times c$.

void **arb_mat_scalar_mul_si**(*arb_mat_t B*, **const** *arb_mat_t A*, *slong c*, *slong prec*)

void **arb_mat_scalar_mul_fmpz**(*arb_mat_t B*, **const** *arb_mat_t A*, **const** *fmpz_t c*, *slong prec*)

void **arb_mat_scalar_mul_arb**(*arb_mat_t B*, **const** *arb_mat_t A*, **const** *arb_t c*, *slong prec*)
Sets B to $A \times c$.

void **arb_mat_scalar_div_si**(*arb_mat_t B*, **const** *arb_mat_t A*, *slong c*, *slong prec*)

void **arb_mat_scalar_div_fmpz**(*arb_mat_t B*, **const** *arb_mat_t A*, **const** *fmpz_t c*, *slong prec*)

void **arb_mat_scalar_div_arb**(*arb_mat_t B*, **const** *arb_mat_t A*, **const** *arb_t c*, *slong prec*)
Sets B to A/c.

8.1.12 Gaussian elimination and solving

int **arb_mat_lu_classical**(*slong* *perm, *arb_mat_t* LU, **const** *arb_mat_t* A, *slong* prec)

int **arb_mat_lu_recursive**(*slong* *perm, *arb_mat_t* LU, **const** *arb_mat_t* A, *slong* prec)

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.

The *classical* version uses Gaussian elimination directly while the *recursive* version performs the computation in a block recursive way to benefit from fast matrix multiplication. The default version chooses an algorithm automatically.

void **arb_mat_solve_tril_classical**(*arb_mat_t* X, **const** *arb_mat_t* L, **const** *arb_mat_t* B, int *unit*, *slong* prec)

void **arb_mat_solve_tril_recursive**(*arb_mat_t* X, **const** *arb_mat_t* L, **const** *arb_mat_t* B, int *unit*, *slong* prec)

void **arb_mat_solve_tril**(*arb_mat_t* X, **const** *arb_mat_t* L, **const** *arb_mat_t* B, int *unit*, *slong* prec)

void **arb_mat_solve_triu_classical**(*arb_mat_t* X, **const** *arb_mat_t* U, **const** *arb_mat_t* B, int *unit*, *slong* prec)

void **arb_mat_solve_triu_recursive**(*arb_mat_t* X, **const** *arb_mat_t* U, **const** *arb_mat_t* B, int *unit*, *slong* prec)

void **arb_mat_solve_triu**(*arb_mat_t* X, **const** *arb_mat_t* U, **const** *arb_mat_t* B, int *unit*, *slong* prec)

Solves the lower triangular system $LX = B$ or the upper triangular system $UX = B$, respectively. If *unit* is set, the main diagonal of L or U is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The *classical* versions perform the computations iteratively while the *recursive* versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

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)

int **arb_mat_solve_lu**(*arb_mat_t* X, **const** *arb_mat_t* A, **const** *arb_mat_t* B, *slong* prec)

int **arb_mat_solve_precond**(*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.

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.

Three algorithms are provided:

- The *lu* version performs LU decomposition directly in ball arithmetic. This is fast, but the bounds typically blow up exponentially with n, even if the system is well-conditioned. This algorithm is usually the best choice at very high precision.
- The *precond* version computes an approximate inverse to precondition the system [HS1967]. This is usually several times slower than direct LU decomposition, but the bounds do not blow up with n if the system is well-conditioned. This algorithm is usually the best choice for large systems at low to moderate precision.
- The default version selects between *lu* and *precomp* automatically.

The automatic choice should be reasonable most of the time, but users may benefit from trying either *lu* or *precond* in specific applications. For example, the *lu* solver often performs better for ill-conditioned systems where use of very high precision is unavoidable.

int **arb_mat_solve_preapprox**(*arb_mat_t X*, **const** *arb_mat_t A*, **const** *arb_mat_t B*, **const** *arb_mat_t R*, **const** *arb_mat_t T*, *slong prec*)

Solves $AX = B$ where A is a nonsingular $n \times n$ matrix and X and B are $n \times m$ matrices, given an approximation R of the matrix inverse of A, and given the approximation T of the solution X.

If $m > 0$ and A cannot be inverted numerically (indicating either that A is singular or that the precision is insufficient, or that R is not a close enough approximation of the inverse of A), 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_lu**(*arb_t det*, **const** *arb_mat_t A*, *slong prec*)

void **arb_mat_det_precond**(*arb_t det*, **const** *arb_mat_t A*, *slong prec*)

void **arb_mat_det**(*arb_t det*, **const** *arb_mat_t A*, *slong prec*)

Sets *det* to the determinant of the matrix A.

The *lu* version uses 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.

The *precond* version computes an approximate LU factorization of A and multiplies by the inverse L and U martices as preconditioners to obtain a matrix close to the identity matrix [Rum2010]. An enclosure for this determinant is computed using Gershgorin circles. This is about four times slower than direct Gaussian elimination, but much more numerically stable.

The default version automatically selects between the *lu* and *precond* versions and additionally handles small or triangular matrices by direct formulas.

void **arb_mat_approx_solve_triu**(*arb_mat_t X*, **const** *arb_mat_t U*, **const** *arb_mat_t B*, int *unit*, *slong prec*)

void **arb_mat_approx_solve_tril**(*arb_mat_t X*, **const** *arb_mat_t L*, **const** *arb_mat_t B*, int *unit*, *slong prec*)

int **arb_mat_approx_lu**(*slong *P*, *arb_mat_t LU*, **const** *arb_mat_t A*, *slong prec*)

void **arb_mat_approx_solve_lu_precomp**(*arb_mat_t X*, **const** *slong *perm*, **const** *arb_mat_t A*, **const** *arb_mat_t B*, *slong prec*)

int **arb_mat_approx_solve**(*arb_mat_t X*, **const** *arb_mat_t A*, **const** *arb_mat_t B*, *slong prec*)

int **arb_mat_approx_inv**(*arb_mat_t X*, **const** *arb_mat_t A*, *slong prec*)

These methods perform approximate solving *without any error control*. The radii in the input matrices are ignored, the computations are done numerically with floating-point arithmetic (using

ordinary Gaussian elimination and triangular solving, accelerated through the use of block recursive strategies for large matrices), and the output matrices are set to the approximate floating-point results with zeroed error bounds.

Approximate solutions are useful for computing preconditioning matrices for certified solutions. Some users may also find these methods useful for doing ordinary numerical linear algebra in applications where error bounds are not needed.

8.1.13 Cholesky decomposition and solving

int **_arb_mat_cholesky_banachiewicz**(*arb_mat_t* *A*, *slong* *prec*)

int **arb_mat_cho**(*arb_mat_t* *L*, **const** *arb_mat_t* *A*, *slong* *prec*)
Computes the Cholesky decomposition of A, returning nonzero iff the symmetric matrix defined by the lower triangular part of A is certainly positive definite.

If a nonzero value is returned, then L is set to the lower triangular matrix such that $A = L * L^T$.

If zero is returned, then either the matrix is not symmetric positive definite, the input matrix was computed to insufficient precision, or the decomposition was attempted at insufficient precision.

The underscore method computes L from A in-place, leaving the strict upper triangular region undefined.

void **arb_mat_solve_cho_precomp**(*arb_mat_t* *X*, **const** *arb_mat_t* *L*, **const** *arb_mat_t* *B*, *slong* *prec*)
Solves $AX = B$ given the precomputed Cholesky decomposition $A = LL^T$. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with L.

int **arb_mat_spd_solve**(*arb_mat_t* *X*, **const** *arb_mat_t* *A*, **const** *arb_mat_t* *B*, *slong* *prec*)
Solves $AX = B$ where A is a symmetric positive definite matrix and X and B are $n \times m$ matrices, using Cholesky decomposition.

If $m > 0$ and A cannot be factored using Cholesky decomposition (indicating either that A is not symmetric positive definite or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the symmetric matrix defined through the lower triangular part of A is invertible and that the exact solution matrix is contained in the output.

void **arb_mat_inv_cho_precomp**(*arb_mat_t* *X*, **const** *arb_mat_t* *L*, *slong* *prec*)
Sets $X = A^{-1}$ where A is a symmetric positive definite matrix whose Cholesky decomposition L has been computed with *arb_mat_cho()*. The inverse is calculated using the method of [Kri2013] which is more efficient than solving $AX = I$ with *arb_mat_solve_cho_precomp()*.

int **arb_mat_spd_inv**(*arb_mat_t* *X*, **const** *arb_mat_t* *A*, *slong* *prec*)
Sets $X = A^{-1}$ where A is a symmetric positive definite matrix. It is calculated using the method of [Kri2013] which computes fewer intermediate results than solving $AX = I$ with *arb_mat_spd_solve()*.

If A cannot be factored using Cholesky decomposition (indicating either that A is not symmetric positive definite or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the symmetric matrix defined through the lower triangular part of A is invertible and that the exact inverse is contained in the output.

int **_arb_mat_ldl_inplace**(*arb_mat_t* *A*, *slong* *prec*)

int **_arb_mat_ldl_golub_and_van_loan**(*arb_mat_t* *A*, *slong* *prec*)

int **arb_mat_ldl**(*arb_mat_t* *res*, **const** *arb_mat_t* *A*, *slong* *prec*)
Computes the LDL^T decomposition of A, returning nonzero iff the symmetric matrix defined by the lower triangular part of A is certainly positive definite.

If a nonzero value is returned, then *res* is set to a lower triangular matrix that encodes the $L*D*L^T$ decomposition of A. In particular, L is a lower triangular matrix with ones on its diagonal and

whose strictly lower triangular region is the same as that of *res*. D is a diagonal matrix with the same diagonal as that of *res*.

If zero is returned, then either the matrix is not symmetric positive definite, the input matrix was computed to insufficient precision, or the decomposition was attempted at insufficient precision.

The underscore methods compute *res* from A in-place, leaving the strict upper triangular region undefined. The default method uses algorithm 4.1.2 from [GVL1996].

void **arb_mat_solve_ldl_precomp**(*arb_mat_t X*, **const** *arb_mat_t L*, **const** *arb_mat_t B*, *slong prec*)
Solves $AX = B$ given the precomputed $A = LDL^T$ decomposition encoded by L. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with L.

void **arb_mat_inv_ldl_precomp**(*arb_mat_t X*, **const** *arb_mat_t L*, *slong prec*)
Sets $X = A^{-1}$ where A is a symmetric positive definite matrix whose LDL^T decomposition encoded by L has been computed with *arb_mat_ldl()*. The inverse is calculated using the method of [Kri2013] which is more efficient than solving $AX = I$ with *arb_mat_solve_ldl_precomp()*.

8.1.14 Characteristic polynomial and companion matrix

void **_arb_mat_charpoly**(*arb_ptr poly*, **const** *arb_mat_t mat*, *slong prec*)

void **arb_mat_charpoly**(*arb_poly_t poly*, **const** *arb_mat_t mat*, *slong prec*)
Sets *poly* 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.

void **_arb_mat_companion**(*arb_mat_t mat*, *arb_srcptr poly*, *slong prec*)

void **arb_mat_companion**(*arb_mat_t mat*, **const** *arb_poly_t poly*, *slong prec*)
Sets the n by n matrix *mat* to the companion matrix of the polynomial *poly* which must have degree n. The underscore method reads $n + 1$ input coefficients.

8.1.15 Special functions

void **arb_mat_exp_taylor_sum**(*arb_mat_t S*, **const** *arb_mat_t A*, *slong N*, *slong prec*)
Sets S to the truncated exponential Taylor series $S = \sum_{k=0}^{N-1} A^k/k!$. Uses rectangular splitting to compute the sum using $O(\sqrt{N})$ matrix multiplications. The recurrence relation for factorials is used to get scalars that are small integers instead of full factorials. As in [Joh2014b], all divisions are postponed to the end by computing partial factorials of length $O(\sqrt{N})$. The scalars could be reduced by doing more divisions, but this appears to be slower in most cases.

void **arb_mat_exp**(*arb_mat_t B*, **const** *arb_mat_t A*, *slong prec*)
Sets B to the exponential of the matrix A, defined by the Taylor series

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

The function is evaluated as $\exp(A/2^r)^{2^r}$, where r is chosen to give rapid convergence.

The elementwise error when truncating the Taylor series after N terms is bounded by the error in the infinity norm, for which we have

$$\left\| \exp(2^{-r}A) - \sum_{k=0}^{N-1} \frac{(2^{-r}A)^k}{k!} \right\|_\infty = \left\| \sum_{k=N}^{\infty} \frac{(2^{-r}A)^k}{k!} \right\|_\infty \le \sum_{k=N}^{\infty} \frac{(2^{-r}\|A\|_\infty)^k}{k!}.$$

We bound the sum on the right using *mag_exp_tail()*. Truncation error is not added to entries whose values are determined by the sparsity structure of A.

void **arb_mat_trace**(*arb_t trace*, **const** *arb_mat_t mat*, *slong prec*)
Sets *trace* to the trace of the matrix, i.e. the sum of entries on the main diagonal of *mat*. The matrix is required to be square.

void **_arb_mat_diag_prod**(*arb_t res*, **const** *arb_mat_t mat*, *slong a*, *slong b*, *slong prec*)

void **arb_mat_diag_prod**(*arb_t res*, **const** *arb_mat_t mat*, *slong prec*)
Sets *res* to the product of the entries on the main diagonal of *mat*. The underscore method computes the product of the entries between index *a* inclusive and *b* exclusive (the indices must be in range).

8.1.16 Sparsity structure

void **arb_mat_entrywise_is_zero**(*fmpz_mat_t dest*, **const** *arb_mat_t src*)
Sets each entry of *dest* to indicate whether the corresponding entry of *src* is certainly zero. If the entry of *src* at row *i* and column *j* is zero according to *arb_is_zero()* then the entry of *dest* at that row and column is set to one, otherwise that entry of *dest* is set to zero.

void **arb_mat_entrywise_not_is_zero**(*fmpz_mat_t dest*, **const** *arb_mat_t src*)
Sets each entry of *dest* to indicate whether the corresponding entry of *src* is not certainly zero. This the complement of *arb_mat_entrywise_is_zero()*.

slong **arb_mat_count_is_zero**(**const** *arb_mat_t mat*)
Returns the number of entries of *mat* that are certainly zero according to *arb_is_zero()*.

slong **arb_mat_count_not_is_zero**(**const** *arb_mat_t mat*)
Returns the number of entries of *mat* that are not certainly zero.

8.1.17 Component and error operations

void **arb_mat_get_mid**(*arb_mat_t B*, **const** *arb_mat_t A*)
Sets the entries of *B* to the exact midpoints of the entries of *A*.

void **arb_mat_add_error_mag**(*arb_mat_t mat*, **const** *mag_t err*)
Adds *err* in-place to the radii of the entries of *mat*.

8.1.18 Eigenvalues and eigenvectors

To compute eigenvalues and eigenvectors, one can convert to an *acb_mat_t* and use the functions in *acb_mat.h: Eigenvalues and eigenvectors*. In the future dedicated methods for real matrices will be added here.

8.2 acb_mat.h – matrices over the complex numbers

An *acb_mat_t* represents a dense matrix over the complex numbers, implemented as an array of entries of type *acb_struct*. The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

Note: Methods prefixed with *acb_mat_approx* treat all input entries as floating-point numbers (ignoring the radii of the balls) and compute floating-point output (balls with zero radius) representing approximate solutions *without error bounds*. All other methods compute rigorous error bounds. The *approx* methods are typically useful for computing initial values or preconditioners for rigorous solvers. Some users may also find *approx* methods useful for doing ordinary numerical linear algebra in applications where error bounds are not needed.

8.2.1 Types, macros and constants

type acb_mat_struct

type 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.

8.2.2 Memory management

void **acb_mat_init**(*acb_mat_t mat*, *slong r*, *slong c*)
Initializes the matrix, setting it to the zero matrix with *r* rows and *c* columns.

void **acb_mat_clear**(*acb_mat_t mat*)
Clears the matrix, deallocating all entries.

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.

void **acb_mat_window_init**(*acb_mat_t window*, **const** *acb_mat_t mat*, *slong r1*, *slong c1*, *slong r2*, *slong c2*)
Initializes *window* to a window matrix into the submatrix of *mat* starting at the corner at row *r1* and column *c1* (inclusive) and ending at row *r2* and column *c2* (exclusive).

void **acb_mat_window_clear**(*acb_mat_t window*)
Frees the window matrix.

8.2.3 Conversions

void **acb_mat_set**(*acb_mat_t dest*, **const** *acb_mat_t src*)

void **acb_mat_set_fmpz_mat**(*acb_mat_t dest*, **const** *fmpz_mat_t src*)

void **acb_mat_set_round_fmpz_mat**(*acb_mat_t dest*, **const** *fmpz_mat_t src*, *slong prec*)

void **acb_mat_set_fmpq_mat**(*acb_mat_t dest*, **const** *fmpq_mat_t src*, *slong prec*)

void **acb_mat_set_arb_mat**(*acb_mat_t dest*, **const** *arb_mat_t src*)

void **acb_mat_set_round_arb_mat**(*acb_mat_t dest*, **const** *arb_mat_t src*, *slong prec*)
Sets *dest* to *src*. The operands must have identical dimensions.

8.2.4 Random generation

void **acb_mat_randtest**(*acb_mat_t* *mat*, flint_rand_t *state*, *slong* *prec*, *slong* *mag_bits*)
Sets *mat* to a random matrix with up to *prec* bits of precision and with exponents of width up to *mag_bits*.

void **acb_mat_randtest_eig**(*acb_mat_t* *mat*, flint_rand_t *state*, *acb_srcptr* *E*, *slong* *prec*)
Sets *mat* to a random matrix with the prescribed eigenvalues supplied as the vector *E*. The output matrix is required to be square. We generate a random unitary matrix via a matrix exponential, and then evaluate an inverse Schur decomposition.

8.2.5 Input and output

void **acb_mat_printd**(**const** *acb_mat_t* *mat*, *slong* *digits*)
Prints each entry in the matrix with the specified number of decimal digits.

void **acb_mat_fprintd**(FILE ***file*, **const** *acb_mat_t* *mat*, *slong* *digits*)
Prints each entry in the matrix with the specified number of decimal digits to the stream *file*.

8.2.6 Comparisons

Predicate methods return 1 if the property certainly holds and 0 otherwise.

int **acb_mat_equal**(**const** *acb_mat_t* *mat1*, **const** *acb_mat_t* *mat2*)
Returns whether the matrices have the same dimensions and identical intervals as entries.

int **acb_mat_overlaps**(**const** *acb_mat_t* *mat1*, **const** *acb_mat_t* *mat2*)
Returns whether the matrices have the same dimensions and each entry in *mat1* overlaps with the corresponding entry in *mat2*.

int **acb_mat_contains**(**const** *acb_mat_t* *mat1*, **const** *acb_mat_t* *mat2*)

int **acb_mat_contains_fmpz_mat**(**const** *acb_mat_t* *mat1*, **const** *fmpz_mat_t* *mat2*)

int **acb_mat_contains_fmpq_mat**(**const** *acb_mat_t* *mat1*, **const** *fmpq_mat_t* *mat2*)
Returns whether the matrices have the same dimensions and each entry in *mat2* is contained in the corresponding entry in *mat1*.

int **acb_mat_eq**(**const** *acb_mat_t* *mat1*, **const** *acb_mat_t* *mat2*)
Returns whether *mat1* and *mat2* certainly represent the same matrix.

int **acb_mat_ne**(**const** *acb_mat_t* *mat1*, **const** *acb_mat_t* *mat2*)
Returns whether *mat1* and *mat2* certainly do not represent the same matrix.

int **acb_mat_is_real**(**const** *acb_mat_t* *mat*)
Returns whether all entries in *mat* have zero imaginary part.

int **acb_mat_is_empty**(**const** *acb_mat_t* *mat*)
Returns whether the number of rows or the number of columns in *mat* is zero.

int **acb_mat_is_square**(**const** *acb_mat_t* *mat*)
Returns whether the number of rows is equal to the number of columns in *mat*.

int **acb_mat_is_exact**(**const** *acb_mat_t* *mat*)
Returns whether all entries in *mat* have zero radius.

int **acb_mat_is_zero**(**const** *acb_mat_t* *mat*)
Returns whether all entries in *mat* are exactly zero.

int **acb_mat_is_finite**(**const** *acb_mat_t* *mat*)
Returns whether all entries in *mat* are finite.

int **acb_mat_is_triu**(**const** *acb_mat_t mat*)
Returns whether *mat* is upper triangular; that is, all entries below the main diagonal are exactly zero.

int **acb_mat_is_tril**(**const** *acb_mat_t mat*)
Returns whether *mat* is lower triangular; that is, all entries above the main diagonal are exactly zero.

int **acb_mat_is_diag**(**const** *acb_mat_t mat*)
Returns whether *mat* is a diagonal matrix; that is, all entries off the main diagonal are exactly zero.

8.2.7 Special matrices

void **acb_mat_zero**(*acb_mat_t mat*)
Sets all entries in mat to zero.

void **acb_mat_one**(*acb_mat_t mat*)
Sets the entries on the main diagonal to ones, and all other entries to zero.

void **acb_mat_ones**(*acb_mat_t mat*)
Sets all entries in the matrix to ones.

void **acb_mat_indeterminate**(*acb_mat_t mat*)
Sets all entries in the matrix to indeterminate (NaN).

void **acb_mat_dft**(*acb_mat_t mat*, int *type*, *slong prec*)
Sets *mat* to the DFT (discrete Fourier transform) matrix of order n where n is the smallest dimension of *mat* (if *mat* is not square, the matrix is extended periodically along the larger dimension). Here, we use the normalized DFT matrix

$$A_{j,k} = \frac{\omega^{jk}}{\sqrt{n}}, \quad \omega = e^{-2\pi i/n}.$$

The *type* parameter is currently ignored and should be set to 0. In the future, it might be used to select a different convention.

8.2.8 Transpose

void **acb_mat_transpose**(*acb_mat_t dest*, **const** *acb_mat_t src*)
Sets *dest* to the exact transpose *src*. The operands must have compatible dimensions. Aliasing is allowed.

void **acb_mat_conjugate_transpose**(*acb_mat_t dest*, **const** *acb_mat_t src*)
Sets *dest* to the conjugate transpose of *src*. The operands must have compatible dimensions. Aliasing is allowed.

void **acb_mat_conjugate**(*acb_mat_t dest*, **const** *acb_mat_t src*)
Sets *dest* to the elementwise complex conjugate of *src*.

8.2.9 Norms

void **acb_mat_bound_inf_norm**(*mag_t b*, **const** *acb_mat_t A*)
Sets b to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of A.

void **acb_mat_frobenius_norm**(*acb_t res*, **const** *acb_mat_t A*, *slong prec*)
Sets *res* to the Frobenius norm (i.e. the square root of the sum of squares of entries) of A.

void **acb_mat_bound_frobenius_norm**(*mag_t res*, **const** *acb_mat_t A*)
Sets *res* to an upper bound for the Frobenius norm of A.

8.2.10 Arithmetic

void **acb_mat_neg**(*acb_mat_t dest*, **const** *acb_mat_t src*)
Sets *dest* to the exact negation of *src*. The operands must have the same dimensions.

void **acb_mat_add**(*acb_mat_t res*, **const** *acb_mat_t mat1*, **const** *acb_mat_t mat2*, *slong prec*)
Sets res to the sum of *mat1* and *mat2*. The operands must have the same dimensions.

void **acb_mat_sub**(*acb_mat_t res*, **const** *acb_mat_t mat1*, **const** *acb_mat_t mat2*, *slong prec*)
Sets *res* to the difference of *mat1* and *mat2*. The operands must have the same dimensions.

void **acb_mat_mul_classical**(*acb_mat_t res*, **const** *acb_mat_t mat1*, **const** *acb_mat_t mat2*, *slong prec*)

void **acb_mat_mul_threaded**(*acb_mat_t res*, **const** *acb_mat_t mat1*, **const** *acb_mat_t mat2*, *slong prec*)

void **acb_mat_mul_reorder**(*acb_mat_t res*, **const** *acb_mat_t mat1*, **const** *acb_mat_t mat2*, *slong prec*)

void **acb_mat_mul**(*acb_mat_t res*, **const** *acb_mat_t mat1*, **const** *acb_mat_t mat2*, *slong prec*)
Sets *res* to the matrix product of *mat1* and *mat2*. The operands must have compatible dimensions for matrix multiplication.

The *classical* version performs matrix multiplication in the trivial way.

The *threaded* version performs classical multiplication but splits the computation over the number of threads returned by *flint_get_num_threads()*.

The *reorder* version reorders the data and performs one to four real matrix multiplications via *arb_mat_mul()*.

The default version chooses an algorithm automatically.

void **acb_mat_mul_entrywise**(*acb_mat_t res*, **const** *acb_mat_t mat1*, **const** *acb_mat_t mat2*, *slong prec*)
Sets *res* to the entrywise product of *mat1* and *mat2*. The operands must have the same dimensions.

void **acb_mat_sqr_classical**(*acb_mat_t res*, **const** *acb_mat_t mat*, *slong prec*)

void **acb_mat_sqr**(*acb_mat_t res*, **const** *acb_mat_t mat*, *slong prec*)
Sets *res* to the matrix square of *mat*. The operands must both be square with the same dimensions.

void **acb_mat_pow_ui**(*acb_mat_t res*, **const** *acb_mat_t mat*, *ulong exp*, *slong prec*)
Sets *res* to *mat* raised to the power *exp*. Requires that *mat* is a square matrix.

void **acb_mat_approx_mul**(*acb_mat_t res*, **const** *acb_mat_t mat1*, **const** *acb_mat_t mat2*, *slong prec*)
Approximate matrix multiplication. The input radii are ignored and the output matrix is set to an approximate floating-point result. For performance reasons, the radii in the output matrix will *not* necessarily be written (zeroed), but will remain zero if they are already zeroed in *res* before calling this function.

8.2.11 Scalar arithmetic

void **acb_mat_scalar_mul_2exp_si**(*acb_mat_t B*, **const** *acb_mat_t A*, *slong c*)
Sets B to A multiplied by 2^c.

void **acb_mat_scalar_addmul_si**(*acb_mat_t B*, **const** *acb_mat_t A*, *slong c*, *slong prec*)

void **acb_mat_scalar_addmul_fmpz**(*acb_mat_t B*, **const** *acb_mat_t A*, **const** *fmpz_t c*, *slong prec*)

void **acb_mat_scalar_addmul_arb**(*acb_mat_t B*, **const** *acb_mat_t A*, **const** *arb_t c*, *slong prec*)

void **acb_mat_scalar_addmul_acb**(*acb_mat_t* *B*, **const** *acb_mat_t* *A*, **const** *acb_t* *c*, *slong* *prec*)
Sets B to $B + A \times c$.

void **acb_mat_scalar_mul_si**(*acb_mat_t* *B*, **const** *acb_mat_t* *A*, *slong* *c*, *slong* *prec*)

void **acb_mat_scalar_mul_fmpz**(*acb_mat_t* *B*, **const** *acb_mat_t* *A*, **const** *fmpz_t* *c*, *slong* *prec*)

void **acb_mat_scalar_mul_arb**(*acb_mat_t* *B*, **const** *acb_mat_t* *A*, **const** *arb_t* *c*, *slong* *prec*)

void **acb_mat_scalar_mul_acb**(*acb_mat_t* *B*, **const** *acb_mat_t* *A*, **const** *acb_t* *c*, *slong* *prec*)
Sets B to $A \times c$.

void **acb_mat_scalar_div_si**(*acb_mat_t* *B*, **const** *acb_mat_t* *A*, *slong* *c*, *slong* *prec*)

void **acb_mat_scalar_div_fmpz**(*acb_mat_t* *B*, **const** *acb_mat_t* *A*, **const** *fmpz_t* *c*, *slong* *prec*)

void **acb_mat_scalar_div_arb**(*acb_mat_t* *B*, **const** *acb_mat_t* *A*, **const** *arb_t* *c*, *slong* *prec*)

void **acb_mat_scalar_div_acb**(*acb_mat_t* *B*, **const** *acb_mat_t* *A*, **const** *acb_t* *c*, *slong* *prec*)
Sets B to A/c.

8.2.12 Gaussian elimination and solving

int **acb_mat_lu_classical**(*slong* **perm*, *acb_mat_t* *LU*, **const** *acb_mat_t* *A*, *slong* *prec*)

int **acb_mat_lu_recursive**(*slong* **perm*, *acb_mat_t* *LU*, **const** *acb_mat_t* *A*, *slong* *prec*)

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.

The *classical* version uses Gaussian elimination directly while the *recursive* version performs the computation in a block recursive way to benefit from fast matrix multiplication. The default version chooses an algorithm automatically.

void **acb_mat_solve_tril_classical**(*acb_mat_t* *X*, **const** *acb_mat_t* *L*, **const** *acb_mat_t* *B*, int *unit*, *slong* *prec*)

void **acb_mat_solve_tril_recursive**(*acb_mat_t* *X*, **const** *acb_mat_t* *L*, **const** *acb_mat_t* *B*, int *unit*, *slong* *prec*)

void **acb_mat_solve_tril**(*acb_mat_t* *X*, **const** *acb_mat_t* *L*, **const** *acb_mat_t* *B*, int *unit*, *slong* *prec*)

void **acb_mat_solve_triu_classical**(*acb_mat_t* *X*, **const** *acb_mat_t* *U*, **const** *acb_mat_t* *B*, int *unit*, *slong* *prec*)

void **acb_mat_solve_triu_recursive**(*acb_mat_t* *X*, **const** *acb_mat_t* *U*, **const** *acb_mat_t* *B*, int *unit*, *slong* *prec*)

void **acb_mat_solve_triu**(*acb_mat_t* *X*, **const** *acb_mat_t* *U*, **const** *acb_mat_t* *B*, int *unit*, *slong* *prec*)
Solves the lower triangular system $LX = B$ or the upper triangular system $UX = B$, respectively. If *unit* is set, the main diagonal of L or U is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The *classical* versions perform the computations iteratively while the *recursive* versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

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*)

int **acb_mat_solve_lu**(*acb_mat_t X*, **const** *acb_mat_t A*, **const** *acb_mat_t B*, *slong prec*)

int **acb_mat_solve_precond**(*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.

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.

Three algorithms are provided:

- The *lu* version performs LU decomposition directly in ball arithmetic. This is fast, but the bounds typically blow up exponentially with n, even if the system is well-conditioned. This algorithm is usually the best choice at very high precision.
- The *precond* version computes an approximate inverse to precondition the system. This is usually several times slower than direct LU decomposition, but the bounds do not blow up with n if the system is well-conditioned. This algorithm is usually the best choice for large systems at low to moderate precision.
- The default version selects between *lu* and *precomp* automatically.

The automatic choice should be reasonable most of the time, but users may benefit from trying either *lu* or *precond* in specific applications. For example, the *lu* solver often performs better for ill-conditioned systems where use of very high precision is unavoidable.

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_lu**(*acb_t det*, **const** *acb_mat_t A*, *slong prec*)

void **acb_mat_det_precond**(*acb_t det*, **const** *acb_mat_t A*, *slong prec*)

void **acb_mat_det**(*acb_t det*, **const** *acb_mat_t A*, *slong prec*)

Sets *det* to the determinant of the matrix A.

The *lu* version uses 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.

The *precond* version computes an approximate LU factorization of A and multiplies by the inverse L and U martices as preconditioners to obtain a matrix close to the identity matrix [Rum2010]. An enclosure for this determinant is computed using Gershgorin circles. This is about four times slower than direct Gaussian elimination, but much more numerically stable.

The default version automatically selects between the *lu* and *precond* versions and additionally handles small or triangular matrices by direct formulas.

void **acb_mat_approx_solve_triu**(*acb_mat_t X*, **const** *acb_mat_t U*, **const** *acb_mat_t B*, int *unit*, *slong prec*)

void **acb_mat_approx_solve_tril**(*acb_mat_t X*, **const** *acb_mat_t L*, **const** *acb_mat_t B*, int *unit*, *slong prec*)

int **acb_mat_approx_lu**(*slong *P*, *acb_mat_t LU*, **const** *acb_mat_t A*, *slong prec*)

void **acb_mat_approx_solve_lu_precomp**(*acb_mat_t X*, **const** *slong *perm*, **const** *acb_mat_t A*, **const** *acb_mat_t B*, *slong prec*)

int **acb_mat_approx_solve**(*acb_mat_t X*, **const** *acb_mat_t A*, **const** *acb_mat_t B*, *slong prec*)

int **acb_mat_approx_inv**(*acb_mat_t X*, **const** *acb_mat_t A*, *slong prec*)

These methods perform approximate solving *without any error control*. The radii in the input matrices are ignored, the computations are done numerically with floating-point arithmetic (using ordinary Gaussian elimination and triangular solving, accelerated through the use of block recursive strategies for large matrices), and the output matrices are set to the approximate floating-point results with zeroed error bounds.

8.2.13 Characteristic polynomial and companion matrix

void **_acb_mat_charpoly**(*acb_ptr poly*, **const** *acb_mat_t mat*, *slong prec*)

void **acb_mat_charpoly**(*acb_poly_t poly*, **const** *acb_mat_t mat*, *slong prec*)

Sets *poly* 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.

void **_acb_mat_companion**(*acb_mat_t mat*, *acb_srcptr poly*, *slong prec*)

void **acb_mat_companion**(*acb_mat_t mat*, **const** *acb_poly_t poly*, *slong prec*)

Sets the n by n matrix *mat* to the companion matrix of the polynomial *poly* which must have degree n. The underscore method reads $n+1$ input coefficients.

8.2.14 Special functions

void **acb_mat_exp_taylor_sum**(*acb_mat_t S*, **const** *acb_mat_t A*, *slong N*, *slong prec*)

Sets S to the truncated exponential Taylor series $S = \sum_{k=0}^{N-1} A^k/k!$. See *arb_mat_exp_taylor_sum()* for implementation notes.

void **acb_mat_exp**(*acb_mat_t B*, **const** *acb_mat_t A*, *slong prec*)

Sets B to the exponential of the matrix A, defined by the Taylor series

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

The function is evaluated as $\exp(A/2^r)^{2^r}$, where r is chosen to give rapid convergence of the Taylor series. Error bounds are computed as for *arb_mat_exp()*.

void **acb_mat_trace**(*acb_t trace*, **const** *acb_mat_t mat*, *slong prec*)

Sets *trace* to the trace of the matrix, i.e. the sum of entries on the main diagonal of *mat*. The matrix is required to be square.

void **_acb_mat_diag_prod**(*acb_t res*, **const** *acb_mat_t mat*, *slong a*, *slong b*, *slong prec*)

void **acb_mat_diag_prod**(*acb_t res*, **const** *acb_mat_t mat*, *slong prec*)

Sets *res* to the product of the entries on the main diagonal of *mat*. The underscore method computes the product of the entries between index a inclusive and b exclusive (the indices must be in range).

8.2.15 Component and error operations

void **acb_mat_get_mid**(*acb_mat_t B*, **const** *acb_mat_t A*)
Sets the entries of B to the exact midpoints of the entries of A.

void **acb_mat_add_error_mag**(*acb_mat_t mat*, **const** *mag_t err*)
Adds *err* in-place to the radii of the entries of *mat*.

8.2.16 Eigenvalues and eigenvectors

The functions in this section are experimental. There are classes of matrices where the algorithms fail to converge even as *prec* is increased, or for which the error bounds are much worse than necessary. In some cases, it can help to manually precondition the matrix A by applying a similarity transformation $T^{-1}AT$.

- If A is badly scaled, take T to be a matrix such that the entries of $T^{-1}AT$ are more uniform (this is known as balancing).
- Simply taking T to be a random invertible matrix can help if an algorithm fails to converge despite A being well-scaled. (This can be the case when dealing with multiple eigenvalues.)

int **acb_mat_approx_eig_qr**(*acb_ptr E*, *acb_mat_t L*, *acb_mat_t R*, **const** *acb_mat_t A*, **const** *mag_t tol*, *slong maxiter*, *slong prec*)
Computes floating-point approximations of all the n eigenvalues (and optionally eigenvectors) of the given n by n matrix A. The approximations of the eigenvalues are written to the vector E, in no particular order. If L is not *NULL*, approximations of the corresponding left eigenvectors are written to the rows of L. If R is not *NULL*, approximations of the corresponding right eigenvectors are written to the columns of R.

The parameters *tol* and *maxiter* can be used to control the target numerical error and the maximum number of iterations allowed before giving up. Passing *NULL* and 0 respectively results in default values being used.

Uses the implicitly shifted QR algorithm with reduction to Hessenberg form. No guarantees are made about the accuracy of the output. A nonzero return value indicates that the QR iteration converged numerically, but this is only a heuristic termination test and does not imply any statement whatsoever about error bounds. The output may also be accurate even if this function returns zero.

void **acb_mat_eig_global_enclosure**(*mag_t eps*, **const** *acb_mat_t A*, *acb_srcptr E*, **const** *acb_mat_t R*, *slong prec*)
Given an n by n matrix A, a length-n vector E containing approximations of the eigenvalues of A, and an n by n matrix R containing approximations of the corresponding right eigenvectors, computes a rigorous bound ε such that every eigenvalue λ of A satisfies $|\lambda - \hat{\lambda}_k| \le \varepsilon$ for some $\hat{\lambda}_k$ in E. In other words, the union of the balls $B_k = \{z : |z - \hat{\lambda}_k| \le \varepsilon\}$ is guaranteed to be an enclosure of all eigenvalues of A.

Note that there is no guarantee that each ball B_k can be identified with a single eigenvalue: it is possible that some balls contain several eigenvalues while other balls contain no eigenvalues. In other words, this method is not powerful enough to compute isolating balls for the individual eigenvalues (or even for clusters of eigenvalues other than the whole spectrum). Nevertheless, in practice the balls B_k will represent eigenvalues one-to-one with high probability if the given approximations are good.

The output can be used to certify that all eigenvalues of A lie in some region of the complex plane (such as a specific half-plane, strip, disk, or annulus) without the need to certify the individual eigenvalues. The output is easily converted into lower or upper bounds for the absolute values or real or imaginary parts of the spectrum, and with high probability these bounds will be tight. Using *acb_add_error_mag()* and *acb_union()*, the output can also be converted to a single *acb_t* enclosing the whole spectrum of A in a rectangle, but note that to test whether a condition holds for all eigenvalues of A, it is typically better to iterate over the individual balls B_k.

This function implements the fast algorithm in Theorem 1 in [Miy2010] which extends the Bauer-Fike theorem. Approximations E and R can, for instance, be computed using *acb_mat_approx_eig_qr()*. No assumptions are made about the structure of A or the quality of the given approximations.

void **acb_mat_eig_enclosure_rump**(*acb_t lambda*, *acb_mat_t J*, *acb_mat_t R*, **const** *acb_mat_t A*, **const** *acb_t lambda_approx*, **const** *acb_mat_t R_approx*, *slong prec*)

Given an n by n matrix A and an approximate eigenvalue-eigenvector pair *lambda_approx* and *R_approx* (where *R_approx* is an n by 1 matrix), computes an enclosure *lambda* guaranteed to contain at least one of the eigenvalues of A, along with an enclosure R for a corresponding right eigenvector.

More generally, this function can handle clustered (or repeated) eigenvalues. If *R_approx* is an n by k matrix containing approximate eigenvectors for a presumed cluster of k eigenvalues near *lambda_approx*, this function computes an enclosure *lambda* guaranteed to contain at least k eigenvalues of A along with a matrix R guaranteed to contain a basis for the k-dimensional invariant subspace associated with these eigenvalues. Note that for multiple eigenvalues, determining the individual eigenvectors is an ill-posed problem; describing an enclosure of the invariant subspace is the best we can hope for.

For $k = 1$, it is guaranteed that $AR - R\lambda$ contains the zero matrix. For $k > 2$, this cannot generally be guaranteed (in particular, A might not diagonalizable). In this case, we can still compute an approximately diagonal k by k interval matrix $J \approx \lambda I$ such that $AR - RJ$ is guaranteed to contain the zero matrix. This matrix has the property that the Jordan canonical form of (any exact matrix contained in) A has a k by k submatrix equal to the Jordan canonical form of (some exact matrix contained in) J. The output J is optional (the user can pass *NULL* to omit it).

The algorithm follows section 13.4 in [Rum2010], corresponding to the `verifyeig()` routine in INTLAB. The initial approximations can, for instance, be computed using *acb_mat_approx_eig_qr()*. No assumptions are made about the structure of A or the quality of the given approximations.

int **acb_mat_eig_simple_rump**(*acb_ptr E*, *acb_mat_t L*, *acb_mat_t R*, **const** *acb_mat_t A*, *acb_srcptr E_approx*, **const** *acb_mat_t R_approx*, *slong prec*)

int **acb_mat_eig_simple_vdhoeven_mourrain**(*acb_ptr E*, *acb_mat_t L*, *acb_mat_t R*, **const** *acb_mat_t A*, *acb_srcptr E_approx*, **const** *acb_mat_t R_approx*, *slong prec*)

int **acb_mat_eig_simple**(*acb_ptr E*, *acb_mat_t L*, *acb_mat_t R*, **const** *acb_mat_t A*, *acb_srcptr E_approx*, **const** *acb_mat_t R_approx*, *slong prec*)

Computes all the eigenvalues (and optionally corresponding eigenvectors) of the given n by n matrix A.

Attempts to prove that A has n simple (isolated) eigenvalues, returning 1 if successful and 0 otherwise. On success, isolating complex intervals for the eigenvalues are written to the vector E, in no particular order. If L is not *NULL*, enclosures of the corresponding left eigenvectors are written to the rows of L. If R is not *NULL*, enclosures of the corresponding right eigenvectors are written to the columns of R.

The left eigenvectors are normalized so that $L = R^{-1}$. This produces a diagonalization $LAR = D$ where D is the diagonal matrix with the entries in E on the diagonal.

The user supplies approximations *E_approx* and *R_approx* of the eigenvalues and the right eigenvectors. The initial approximations can, for instance, be computed using *acb_mat_approx_eig_qr()*. No assumptions are made about the structure of A or the quality of the given approximations.

Two algorithms are implemented:

- The *rump* version calls *acb_mat_eig_enclosure_rump()* repeatedly to certify eigenvalue-eigenvector pairs one by one. The iteration is stopped to return non-success if a new eigenvalue overlaps with previously computed one. Finally, L is computed by a matrix inversion. This has complexity $O(n^4)$.

- The *vdhoeven_mourrain* version uses the algorithm in [HM2017] to certify all eigenvalues and eigenvectors in one step. This has complexity $O(n^3)$.

The default version currently uses *vdhoeven_mourrain*.

By design, these functions terminate instead of attempting to compute eigenvalue clusters if some eigenvalues cannot be isolated. To compute all eigenvalues of a matrix allowing for overlap, `acb_mat_eig_multiple_rump()` may be used as a fallback, or `acb_mat_eig_multiple()` may be used in the first place.

int **acb_mat_eig_multiple_rump**(*acb_ptr* *E*, **const** *acb_mat_t* *A*, *acb_srcptr* *E_approx*, **const** *acb_mat_t* *R_approx*, *slong* *prec*)

int **acb_mat_eig_multiple**(*acb_ptr* *E*, **const** *acb_mat_t* *A*, *acb_srcptr* *E_approx*, **const** *acb_mat_t* *R_approx*, *slong* *prec*)

Computes all the eigenvalues of the given n by n matrix A. On success, the output vector E contains n complex intervals, each representing one eigenvalue of A with the correct multiplicities in case of overlap. The output intervals are either disjoint or identical, and identical intervals are guaranteed to be grouped consecutively. Each complete run of k identical intervals thus represents a cluster of exactly k eigenvalues which could not be separated from each other at the current precision, but which could be isolated from the other $n - k$ eigenvalues of the matrix.

The user supplies approximations *E_approx* and *R_approx* of the eigenvalues and the right eigenvectors. The initial approximations can, for instance, be computed using `acb_mat_approx_eig_qr()`. No assumptions are made about the structure of A or the quality of the given approximations.

The *rump* algorithm groups approximate eigenvalues that are close and calls `acb_mat_eig_enclosure_rump()` repeatedly to validate each cluster. The complexity is $O(mn^3)$ for m clusters.

The default version, as currently implemented, first attempts to call `acb_mat_eig_simple_vdhoeven_mourrain()` hoping that the eigenvalues are actually simple. It then uses the *rump* algorithm as a fallback.

CHAPTER

NINE

SPECIAL FUNCTIONS

These modules implement mathematical functions with complexity that goes beyond the basics covered directly in the *arb* and *acb* modules.

9.1 acb_hypgeom.h – hypergeometric functions of complex variables

The generalized hypergeometric function is formally defined by

$$ {}_pF_q(a_1,\ldots,a_p;b_1,\ldots,b_q;z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \ldots (a_p)_k}{(b_1)_k \ldots (b_q)_k} \frac{z^k}{k!}. $$

It can be interpreted using analytic continuation or regularization when the sum does not converge. In a looser sense, we understand "hypergeometric functions" to be linear combinations of generalized hypergeometric functions with prefactors that are products of exponentials, powers, and gamma functions.

9.1.1 Convergent series

In this section, we define

$$ T(k) = \frac{\prod_{i=0}^{p-1} (a_i)_k}{\prod_{i=0}^{q-1} (b_i)_k} z^k $$

and

$$ {}_pf_q(a_0,\ldots,a_{p-1};b_0\ldots b_{q-1};z) = {}_{p+1}F_q(a_0,\ldots,a_{p-1},1;b_0\ldots b_{q-1};z) = \sum_{k=0}^{\infty} T(k) $$

For the conventional generalized hypergeometric function ${}_pF_q$, compute ${}_pf_{q+1}$ with the explicit parameter $b_q = 1$, or remove a 1 from the a_i parameters if there is one.

void **acb_hypgeom_pfq_bound_factor**(*mag_t* *C*, *acb_srcptr* *a*, *slong* *p*, *acb_srcptr* *b*, *slong* *q*, **const** *acb_t* *z*, *ulong* *n*)

Computes a factor C such that $|\sum_{k=n}^{\infty} T(k)| \le C|T(n)|$. See *Convergent series*. As currently implemented, the bound becomes infinite when n is too small, even if the series converges.

slong **acb_hypgeom_pfq_choose_n**(*acb_srcptr* *a*, *slong* *p*, *acb_srcptr* *b*, *slong* *q*, **const** *acb_t* *z*, *slong* *prec*)

Heuristically attempts to choose a number of terms n to sum of a hypergeometric series at a working precision of *prec* bits.

Uses double precision arithmetic internally. As currently implemented, it can fail to produce a good result if the parameters are extremely large or extremely close to nonpositive integers.

Numerical cancellation is assumed to be significant, so truncation is done when the current term is *prec* bits smaller than the largest encountered term.

This function will also attempt to pick a reasonable truncation point for divergent series.

void **acb_hypgeom_pfq_sum_forward**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, **const** *acb_t z*, *slong n*, *slong prec*)

void **acb_hypgeom_pfq_sum_rs**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, **const** *acb_t z*, *slong n*, *slong prec*)

void **acb_hypgeom_pfq_sum_bs**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, **const** *acb_t z*, *slong n*, *slong prec*)

void **acb_hypgeom_pfq_sum_fme**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, **const** *acb_t z*, *slong n*, *slong prec*)

void **acb_hypgeom_pfq_sum**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, **const** *acb_t z*, *slong n*, *slong prec*)

Computes $s = \sum_{k=0}^{n-1} T(k)$ and $t = T(n)$. Does not allow aliasing between input and output variables. We require $n \geq 0$.

The *forward* version computes the sum using forward recurrence.

The *bs* version computes the sum using binary splitting.

The *rs* version computes the sum in reverse order using rectangular splitting. It only computes a magnitude bound for the value of t.

The *fme* version uses fast multipoint evaluation.

The default version automatically chooses an algorithm depending on the inputs.

void **acb_hypgeom_pfq_sum_bs_invz**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, **const** *acb_t w*, *slong n*, *slong prec*)

void **acb_hypgeom_pfq_sum_invz**(*acb_t s*, *acb_t t*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, **const** *acb_t z*, **const** *acb_t w*, *slong n*, *slong prec*)

Like *acb_hypgeom_pfq_sum()*, but taking advantage of $w = 1/z$ possibly having few bits.

void **acb_hypgeom_pfq_direct**(*acb_t res*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, **const** *acb_t z*, *slong n*, *slong prec*)

Computes

$$ {}_pf_q(z) = \sum_{k=0}^{\infty} T(k) = \sum_{k=0}^{n-1} T(k) + \varepsilon $$

directly from the defining series, including a rigorous bound for the truncation error ε in the output.

If $n < 0$, this function chooses a number of terms automatically using *acb_hypgeom_pfq_choose_n()*.

void **acb_hypgeom_pfq_series_sum_forward**(*acb_poly_t s*, *acb_poly_t t*, **const** *acb_poly_struct* **a*, *slong p*, **const** *acb_poly_struct* **b*, *slong q*, **const** *acb_poly_t z*, int *regularized*, *slong n*, *slong len*, *slong prec*)

void **acb_hypgeom_pfq_series_sum_bs**(*acb_poly_t s*, *acb_poly_t t*, **const** *acb_poly_struct* **a*, *slong p*, **const** *acb_poly_struct* **b*, *slong q*, **const** *acb_poly_t z*, int *regularized*, *slong n*, *slong len*, *slong prec*)

void **acb_hypgeom_pfq_series_sum_rs**(*acb_poly_t s*, *acb_poly_t t*, **const** *acb_poly_struct* **a*, *slong p*, **const** *acb_poly_struct* **b*, *slong q*, **const** *acb_poly_t z*, int *regularized*, *slong n*, *slong len*, *slong prec*)

void **acb_hypgeom_pfq_series_sum**(*acb_poly_t s*, *acb_poly_t t*, **const** *acb_poly_struct* **a*, *slong p*, **const** *acb_poly_struct* **b*, *slong q*, **const** *acb_poly_t z*, int *regularized*, *slong n*, *slong len*, *slong prec*)

Computes $s = \sum_{k=0}^{n-1} T(k)$ and $t = T(n)$ given parameters and argument that are power series. Does not allow aliasing between input and output variables. We require $n \geq 0$ and that *len* is positive.

If *regularized* is set, the regularized sum is computed, avoiding division by zero at the poles of the gamma function.

The *forward*, *bs*, *rs* and default versions use forward recurrence, binary splitting, rectangular splitting, and an automatic algorithm choice.

void **acb_hypgeom_pfq_series_direct**(*acb_poly_t res*, **const** *acb_poly_struct* *a, *slong p*, **const** *acb_poly_struct* *b, *slong q*, **const** *acb_poly_t z*, int *regularized*, *slong n*, *slong len*, *slong prec*)

Computes ${}_pf_q(z)$ directly using the defining series, given parameters and argument that are power series. The result is a power series of length *len*. We require that *len* is positive.

An error bound is computed automatically as a function of the number of terms n. If $n < 0$, the number of terms is chosen automatically.

If *regularized* is set, the regularized hypergeometric function is computed instead.

9.1.2 Asymptotic series

$U(a, b, z)$ is the confluent hypergeometric function of the second kind with the principal branch cut, and $U^* = z^a U(a, b, z)$. For details about how error bounds are computed, see *Asymptotic series for the confluent hypergeometric function*.

void **acb_hypgeom_u_asymp**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t z*, *slong n*, *slong prec*)

Sets *res* to $U^*(a, b, z)$ computed using n terms of the asymptotic series, with a rigorous bound for the error included in the output. We require $n \geq 0$.

int **acb_hypgeom_u_use_asymp**(**const** *acb_t z*, *slong prec*)

Heuristically determines whether the asymptotic series can be used to evaluate $U(a, b, z)$ to *prec* accurate bits (assuming that a and b are small).

9.1.3 Generalized hypergeometric function

void **acb_hypgeom_pfq**(*acb_poly_t res*, *acb_srcptr a*, *slong p*, *acb_srcptr b*, *slong q*, **const** *acb_t z*, int *regularized*, *slong prec*)

Computes the generalized hypergeometric function ${}_pF_q(z)$, or the regularized version if *regularized* is set.

This function automatically delegates to a specialized implementation when the order (p, q) is one of (0,0), (1,0), (0,1), (1,1), (2,1). Otherwise, it falls back to direct summation.

While this is a top-level function meant to take care of special cases automatically, it does not generally perform the optimization of deleting parameters that appear in both a and b. This can be done ahead of time by the user in applications where duplicate parameters are likely to occur.

9.1.4 Confluent hypergeometric functions

void **acb_hypgeom_u_1f1_series**(*acb_poly_t res*, **const** *acb_poly_t a*, **const** *acb_poly_t b*, **const** *acb_poly_t z*, *slong len*, *slong prec*)

Computes $U(a, b, z)$ as a power series truncated to length *len*, given $a, b, z \in \mathbb{C}[[x]]$. If $b[0] \in \mathbb{Z}$, it computes one extra derivative and removes the singularity (it is then assumed that $b[1] \neq 0$). As currently implemented, the output is indeterminate if b is nonexact and contains an integer.

void **acb_hypgeom_u_1f1**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t z*, *slong prec*)

Computes $U(a, b, z)$ as a sum of two convergent hypergeometric series. If $b \in \mathbb{Z}$, it computes the limit value via *acb_hypgeom_u_1f1_series()*. As currently implemented, the output is indeterminate if b is nonexact and contains an integer.

void **acb_hypgeom_u**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t z*, *slong prec*)
Computes $U(a,b,z)$ using an automatic algorithm choice. The function *acb_hypgeom_u_asymp()* is used if a or $a-b+1$ is a nonpositive integer (in which case the asymptotic series terminates), or if z is sufficiently large. Otherwise *acb_hypgeom_u_1f1()* is used.

void **acb_hypgeom_m_asymp**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_m_1f1**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_m**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t z*, int *regularized*, *slong prec*)
Computes the confluent hypergeometric function $M(a,b,z) = {}_1F_1(a,b,z)$, or $\mathbf{M}(a,b,z) = \frac{1}{\Gamma(b)}{}_1F_1(a,b,z)$ if *regularized* is set.

void **acb_hypgeom_1f1**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t z*, int *regularized*, *slong prec*)
Alias for *acb_hypgeom_m()*.

void **acb_hypgeom_0f1_asymp**(*acb_t res*, **const** *acb_t a*, **const** *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_0f1_direct**(*acb_t res*, **const** *acb_t a*, **const** *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_0f1**(*acb_t res*, **const** *acb_t a*, **const** *acb_t z*, int *regularized*, *slong prec*)
Computes the confluent hypergeometric function ${}_0F_1(a,z)$, or $\frac{1}{\Gamma(a)}{}_0F_1(a,z)$ if *regularized* is set, using asymptotic expansions, direct summation, or an automatic algorithm choice. The *asymp* version uses the asymptotic expansions of Bessel functions, together with the connection formulas

$$\frac{{}_0F_1(a,z)}{\Gamma(a)} = (-z)^{(1-a)/2} J_{a-1}(2\sqrt{-z}) = z^{(1-a)/2} I_{a-1}(2\sqrt{z}).$$

The Bessel-J function is used in the left half-plane and the Bessel-I function is used in the right half-plane, to avoid loss of accuracy due to evaluating the square root on the branch cut.

9.1.5 Error functions and Fresnel integrals

void **acb_hypgeom_erf_propagated_error**(*mag_t re*, *mag_t im*, **const** *acb_t z*)
Sets *re* and *im* to upper bounds for the error in the real and imaginary part resulting from approximating the error function of z by the error function evaluated at the midpoint of z. Uses the first derivative.

void **acb_hypgeom_erf_1f1a**(*acb_t res*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_erf_1f1b**(*acb_t res*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_erf_asymp**(*acb_t res*, **const** *acb_t z*, int *complementary*, *slong prec*, *slong prec2*)
Computes the error function respectively using

$$\operatorname{erf}(z) = \frac{2z}{\sqrt{\pi}} {}_1F_1(\tfrac{1}{2}, \tfrac{3}{2}, -z^2)$$

$$\operatorname{erf}(z) = \frac{2ze^{-z^2}}{\sqrt{\pi}} {}_1F_1(1, \tfrac{3}{2}, z^2)$$

$$\operatorname{erf}(z) = \frac{z}{\sqrt{z^2}} \left(1 - \frac{e^{-z^2}}{\sqrt{\pi}} U(\tfrac{1}{2}, \tfrac{1}{2}, z^2)\right) = \frac{z}{\sqrt{z^2}} - \frac{e^{-z^2}}{z\sqrt{\pi}} U^*(\tfrac{1}{2}, \tfrac{1}{2}, z^2).$$

The *asymp* version takes a second precision to use for the U term. It also takes an extra flag *complementary*, computing the complementary error function if set.

void **acb_hypgeom_erf**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes the error function using an automatic algorithm choice. If z is too small to use the asymptotic expansion, a working precision sufficient to circumvent cancellation in the hypergeometric series is determined automatically, and a bound for the propagated error is computed with *acb_hypgeom_erf_propagated_error()*.

void **_acb_hypgeom_erf_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_erf_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong len*, *slong prec*)
Computes the error function of the power series z, truncated to length *len*.

void **acb_hypgeom_erfc**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes the complementary error function $\operatorname{erfc}(z) = 1 - \operatorname{erf}(z)$. This function avoids catastrophic cancellation for large positive z.

void **_acb_hypgeom_erfc_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_erfc_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong len*, *slong prec*)
Computes the complementary error function of the power series z, truncated to length *len*.

void **acb_hypgeom_erfi**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes the imaginary error function $\operatorname{erfi}(z) = -i \operatorname{erf}(iz)$. This is a trivial wrapper of *acb_hypgeom_erf()*.

void **_acb_hypgeom_erfi_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_erfi_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong len*, *slong prec*)
Computes the imaginary error function of the power series z, truncated to length *len*.

void **acb_hypgeom_fresnel**(*acb_t res1*, *acb_t res2*, **const** *acb_t z*, int *normalized*, *slong prec*)
Sets *res1* to the Fresnel sine integral $S(z)$ and *res2* to the Fresnel cosine integral $C(z)$. Optionally, just a single function can be computed by passing *NULL* as the other output variable. The definition $S(z) = \int_0^z \sin(t^2) dt$ is used if *normalized* is 0, and $S(z) = \int_0^z \sin(\frac{1}{2}\pi t^2) dt$ is used if *normalized* is 1 (the latter is the Abramowitz & Stegun convention). $C(z)$ is defined analogously.

void **_acb_hypgeom_fresnel_series**(*acb_ptr res1*, *acb_ptr res2*, *acb_srcptr z*, *slong zlen*, int *normalized*, *slong len*, *slong prec*)

void **acb_hypgeom_fresnel_series**(*acb_poly_t res1*, *acb_poly_t res2*, **const** *acb_poly_t z*, int *normalized*, *slong len*, *slong prec*)
Sets *res1* to the Fresnel sine integral and *res2* to the Fresnel cosine integral of the power series z, truncated to length *len*. Optionally, just a single function can be computed by passing *NULL* as the other output variable.

9.1.6 Bessel functions

void **acb_hypgeom_bessel_j_asymp**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, *slong prec*)
Computes the Bessel function of the first kind via *acb_hypgeom_u_asymp()*. For all complex ν, z, we have

$$J_\nu(z) = \frac{z^\nu}{2^\nu e^{iz} \Gamma(\nu+1)} {}_1F_1(\nu + \tfrac{1}{2}, 2\nu + 1, 2iz) = A_+ B_+ + A_- B_-$$

where

$$A_\pm = z^\nu (z^2)^{-\frac{1}{2}-\nu} (\mp iz)^{\frac{1}{2}+\nu} (2\pi)^{-1/2} = (\pm iz)^{-1/2-\nu} z^\nu (2\pi)^{-1/2}$$

$$B_\pm = e^{\pm iz} U^*(\nu + \tfrac{1}{2}, 2\nu + 1, \mp 2iz).$$

Nicer representations of the factors $A_\pm$ can be given depending conditionally on the parameters. If $\nu + \frac{1}{2} = n \in \mathbb{Z}$, we have $A_\pm = (\pm i)^n (2\pi z)^{-1/2}$. And if $\operatorname{Re}(z) > 0$, we have $A_\pm = \exp(\mp i[(2\nu + 1)/4]\pi)(2\pi z)^{-1/2}$.

void **acb_hypgeom_bessel_j_0f1**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, *slong prec*)
Computes the Bessel function of the first kind from

$$J_\nu(z) = \frac{1}{\Gamma(\nu+1)} \left(\frac{z}{2}\right)^\nu {}_0F_1\left(\nu+1, -\frac{z^2}{4}\right).$$

void **acb_hypgeom_bessel_j**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, *slong prec*)
Computes the Bessel function of the first kind $J_\nu(z)$ using an automatic algorithm choice.

void **acb_hypgeom_bessel_y**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, *slong prec*)
Computes the Bessel function of the second kind $Y_\nu(z)$ from the formula

$$Y_\nu(z) = \frac{\cos(\nu\pi)J_\nu(z) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

unless $\nu = n$ is an integer in which case the limit value

$$Y_n(z) = -\frac{2}{\pi}\left(i^n K_n(iz) + [\log(iz) - \log(z)]\, J_n(z)\right)$$

is computed. As currently implemented, the output is indeterminate if ν is nonexact and contains an integer.

void **acb_hypgeom_bessel_jy**(*acb_t res1*, *acb_t res2*, **const** *acb_t nu*, **const** *acb_t z*, *slong prec*)
Sets *res1* to $J_\nu(z)$ and *res2* to $Y_\nu(z)$, computed simultaneously. From these values, the user can easily construct the Bessel functions of the third kind (Hankel functions) $H_\nu^{(1)}(z), H_\nu^{(2)}(z) = J_\nu(z) \pm iY_\nu(z)$.

9.1.7 Modified Bessel functions

void **acb_hypgeom_bessel_i_asymp**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, int *scaled*, *slong prec*)

void **acb_hypgeom_bessel_i_0f1**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, int *scaled*, *slong prec*)

void **acb_hypgeom_bessel_i**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_bessel_i_scaled**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, *slong prec*)
Computes the modified Bessel function of the first kind $I_\nu(z) = z^\nu(iz)^{-\nu}J_\nu(iz)$ respectively using asymptotic series (see *acb_hypgeom_bessel_j_asymp()*), the convergent series

$$I_\nu(z) = \frac{1}{\Gamma(\nu+1)} \left(\frac{z}{2}\right)^\nu {}_0F_1\left(\nu+1, \frac{z^2}{4}\right),$$

or an automatic algorithm choice.

The *scaled* version computes the function $e^{-z}I_\nu(z)$. The *asymp* and *0f1* functions implement both variants and allow choosing with a flag.

void **acb_hypgeom_bessel_k_asymp**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, int *scaled*, *slong prec*)
Computes the modified Bessel function of the second kind via via *acb_hypgeom_u_asymp()*. For all ν and all $z \neq 0$, we have

$$K_\nu(z) = \left(\frac{2z}{\pi}\right)^{-1/2} e^{-z} U^*(\nu + \tfrac{1}{2}, 2\nu+1, 2z).$$

If *scaled* is set, computes the function $e^z K_\nu(z)$.

void **acb_hypgeom_bessel_k_0f1_series**(*acb_poly_t res*, **const** *acb_poly_t nu*, **const** *acb_poly_t z*, int *scaled*, *slong len*, *slong prec*)
Computes the modified Bessel function of the second kind $K_\nu(z)$ as a power series truncated to length *len*, given $\nu, z \in \mathbb{C}[[x]]$. Uses the formula

$$K_\nu(z) = \frac{1}{2}\frac{\pi}{\sin(\pi\nu)}\left[\left(\frac{z}{2}\right)^{-\nu} {}_0\widetilde{F}_1\left(1-\nu, \frac{z^2}{4}\right) - \left(\frac{z}{2}\right)^{\nu} {}_0\widetilde{F}_1\left(1+\nu, \frac{z^2}{4}\right)\right].$$

If $\nu[0] \in \mathbb{Z}$, it computes one extra derivative and removes the singularity (it is then assumed that $\nu[1] \neq 0$). As currently implemented, the output is indeterminate if $\nu[0]$ is nonexact and contains an integer.

If *scaled* is set, computes the function $e^z K_\nu(z)$.

void **acb_hypgeom_bessel_k_0f1**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, int *scaled*, *slong prec*)

Computes the modified Bessel function of the second kind from

$$K_\nu(z) = \frac{1}{2}\left[\left(\frac{z}{2}\right)^{-\nu} \Gamma(\nu) {}_0F_1\left(1-\nu, \frac{z^2}{4}\right) - \left(\frac{z}{2}\right)^{\nu} \frac{\pi}{\nu \sin(\pi\nu)\Gamma(\nu)} {}_0F_1\left(\nu+1, \frac{z^2}{4}\right)\right]$$

if $\nu \notin \mathbb{Z}$. If $\nu \in \mathbb{Z}$, it computes the limit value via *acb_hypgeom_bessel_k_0f1_series()*. As currently implemented, the output is indeterminate if ν is nonexact and contains an integer.

If *scaled* is set, computes the function $e^z K_\nu(z)$.

void **acb_hypgeom_bessel_k**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, *slong prec*)

Computes the modified Bessel function of the second kind $K_\nu(z)$ using an automatic algorithm choice.

void **acb_hypgeom_bessel_k_scaled**(*acb_t res*, **const** *acb_t nu*, **const** *acb_t z*, *slong prec*)

Computes the function $e^z K_\nu(z)$.

9.1.8 Airy functions

The Airy functions are linearly independent solutions of the differential equation $y'' - zy = 0$. All solutions are entire functions. The standard solutions are denoted $\mathrm{Ai}(z), \mathrm{Bi}(z)$. For negative z, both functions are oscillatory. For positive z, the first function decreases exponentially while the second increases exponentially.

The Airy functions can be expressed in terms of Bessel functions of fractional order, but this is inconvenient since such formulas only hold piecewise (due to the Stokes phenomenon). Computation of the Airy functions can also be optimized more than Bessel functions in general. We therefore provide a dedicated interface for evaluating Airy functions.

The following methods optionally compute $(\mathrm{Ai}(z), \mathrm{Ai}'(z), \mathrm{Bi}(z), \mathrm{Bi}'(z))$ simultaneously. Any of the four function values can be omitted by passing *NULL* for the unwanted output variables, speeding up the evaluation.

void **acb_hypgeom_airy_direct**(*acb_t ai*, *acb_t ai_prime*, *acb_t bi*, *acb_t bi_prime*, **const** *acb_t z*, *slong n*, *slong prec*)

Computes the Airy functions using direct series expansions truncated at n terms. Error bounds are included in the output.

void **acb_hypgeom_airy_asymp**(*acb_t ai*, *acb_t ai_prime*, *acb_t bi*, *acb_t bi_prime*, **const** *acb_t z*, *slong n*, *slong prec*)

Computes the Airy functions using asymptotic expansions truncated at n terms. Error bounds are included in the output. For details about how the error bounds are computed, see *Asymptotic series for Airy functions*.

void **acb_hypgeom_airy_bound**(*mag_t ai*, *mag_t ai_prime*, *mag_t bi*, *mag_t bi_prime*, **const** *acb_t z*)

Computes bounds for the Airy functions using first-order asymptotic expansions together with error bounds. This function uses some shortcuts to make it slightly faster than calling *acb_hypgeom_airy_asymp()* with $n = 1$.

void **acb_hypgeom_airy**(*acb_t ai*, *acb_t ai_prime*, *acb_t bi*, *acb_t bi_prime*, **const** *acb_t z*, *slong prec*)

Computes Airy functions using an automatic algorithm choice.

We use *acb_hypgeom_airy_asymp()* whenever this gives full accuracy and *acb_hypgeom_airy_direct()* otherwise. In the latter case, we first use hardware double precision arithmetic to determine an accurate estimate of the working precision needed to

compute the Airy functions accurately for given z. This estimate is obtained by comparing the leading-order asymptotic estimate of the Airy functions with the magnitude of the largest term in the power series. The estimate is generic in the sense that it does not take into account vanishing near the roots of the functions. We subsequently evaluate the power series at the midpoint of z and bound the propagated error using derivatives. Derivatives are bounded using *acb_hypgeom_airy_bound()*.

void **acb_hypgeom_airy_jet**(*acb_ptr ai*, *acb_ptr bi*, **const** *acb_t z*, *slong len*, *slong prec*)
Writes to *ai* and *bi* the respective Taylor expansions of the Airy functions at the point z, truncated to length *len*. Either of the outputs can be *NULL* to avoid computing that function. The variable z is not allowed to be aliased with the outputs. To simplify the implementation, this method does not compute the series expansions of the primed versions directly; these are easily obtained by computing one extra coefficient and differentiating the output with *_acb_poly_derivative()*.

void **_acb_hypgeom_airy_series**(*acb_ptr ai*, *acb_ptr ai_prime*, *acb_ptr bi*, *acb_ptr bi_prime*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_airy_series**(*acb_poly_t ai*, *acb_poly_t ai_prime*, *acb_poly_t bi*, *acb_poly_t bi_prime*, **const** *acb_poly_t z*, *slong len*, *slong prec*)
Computes the Airy functions evaluated at the power series z, truncated to length *len*. As with the other Airy methods, any of the outputs can be *NULL*.

9.1.9 Coulomb wave functions

Coulomb wave functions are solutions of the Coulomb wave equation

$$y'' + \left(1 - \frac{2\eta}{z} - \frac{\ell(\ell+1)}{z^2}\right) y = 0$$

which is the radial Schrödinger equation for a charged particle in a Coulomb potential $1/z$, where ℓ is the orbital angular momentum and η is the Sommerfeld parameter. The standard solutions are named $F_\ell(\eta, z)$ (regular at the origin $z = 0$) and $G_\ell(\eta, z)$ (irregular at the origin). The irregular solutions $H_\ell^{\pm}(\eta, z) = G_\ell(\eta, z) \pm iF_\ell(\eta, z)$ are also used.

Coulomb wave functions are special cases of confluent hypergeometric functions. The normalization constants and connection formulas are discussed in [DYF1999], [Gas2018], [Mic2007] and chapter 33 in [NIST2012]. In this implementation, we define the analytic continuations of all the functions so that the branch cut with respect to z is placed on the negative real axis. Precise definitions are given in http://fungrim.org/topic/Coulomb_wave_functions/

The following methods optionally compute $F_\ell(\eta, z), G_\ell(\eta, z), H_\ell^+(\eta, z), H_\ell^-(\eta, z)$ simultaneously. Any of the four function values can be omitted by passing *NULL* for the unwanted output variables. The redundant functions $H^{\pm}$ are provided explicitly since taking the linear combination of F and G suffers from cancellation in parts of the complex plane.

void **acb_hypgeom_coulomb**(*acb_t F*, *acb_t G*, *acb_t Hpos*, *acb_t Hneg*, **const** *acb_t l*, **const** *acb_t eta*, **const** *acb_t z*, *slong prec*)
Writes to *F*, *G*, *Hpos*, *Hneg* the values of the respective Coulomb wave functions. Any of the outputs can be *NULL*.

void **acb_hypgeom_coulomb_jet**(*acb_ptr F*, *acb_ptr G*, *acb_ptr Hpos*, *acb_ptr Hneg*, **const** *acb_t l*, **const** *acb_t eta*, **const** *acb_t z*, *slong len*, *slong prec*)
Writes to *F*, *G*, *Hpos*, *Hneg* the respective Taylor expansions of the Coulomb wave functions at the point z, truncated to length *len*. Any of the outputs can be *NULL*.

void **_acb_hypgeom_coulomb_series**(*acb_ptr F*, *acb_ptr G*, *acb_ptr Hpos*, *acb_ptr Hneg*, **const** *acb_t l*, **const** *acb_t eta*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_coulomb_series**(*acb_poly_t F*, *acb_poly_t G*, *acb_poly_t Hpos*, *acb_poly_t Hneg*, **const** *acb_t l*, **const** *acb_t eta*, **const** *acb_poly_t z*, *slong len*, *slong prec*)
Computes the Coulomb wave functions evaluated at the power series z, truncated to length *len*. Any of the outputs can be *NULL*.

9.1.10 Incomplete gamma and beta functions

void **acb_hypgeom_gamma_upper_asymp**(*acb_t res*, **const** *acb_t s*, **const** *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_gamma_upper_1f1a**(*acb_t res*, **const** *acb_t s*, **const** *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_gamma_upper_1f1b**(*acb_t res*, **const** *acb_t s*, **const** *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_gamma_upper_singular**(*acb_t res*, *slong s*, **const** *acb_t z*, int *regularized*, *slong prec*)

void **acb_hypgeom_gamma_upper**(*acb_t res*, **const** *acb_t s*, **const** *acb_t z*, int *regularized*, *slong prec*)

If *regularized* is 0, computes the upper incomplete gamma function $\Gamma(s,z)$.

If *regularized* is 1, computes the regularized upper incomplete gamma function $Q(s,z) = \Gamma(s,z)/\Gamma(s)$.

If *regularized* is 2, computes the generalized exponential integral $z^{-s}\Gamma(s,z) = E_{1-s}(z)$ instead (this option is mainly intended for internal use; *acb_hypgeom_expint()* is the intended interface for computing the exponential integral).

The different methods respectively implement the formulas

$$\Gamma(s,z) = e^{-z} U(1-s, 1-s, z)$$

$$\Gamma(s,z) = \Gamma(s) - \frac{z^s}{s} {}_1F_1(s, s+1, -z)$$

$$\Gamma(s,z) = \Gamma(s) - \frac{z^s e^{-z}}{s} {}_1F_1(1, s+1, z)$$

$$\Gamma(s,z) = \frac{(-1)^n}{n!}(\psi(n+1) - \log(z)) + \frac{(-1)^n}{(n+1)!} z \, {}_2F_2(1,1,2,2+n,-z) - z^{-n} \sum_{k=0}^{n-1} \frac{(-z)^k}{(k-n)k!}, \quad n = -s \in \mathbb{Z}_{\geq 0}$$

and an automatic algorithm choice. The automatic version also handles other special input such as $z = 0$ and $s = 1, 2, 3$. The *singular* version evaluates the finite sum directly and therefore assumes that s is not too large.

void **_acb_hypgeom_gamma_upper_series**(*acb_ptr res*, **const** *acb_t s*, *acb_srcptr z*, *slong zlen*, int *regularized*, *slong n*, *slong prec*)

void **acb_hypgeom_gamma_upper_series**(*acb_poly_t res*, **const** *acb_t s*, **const** *acb_poly_t z*, int *regularized*, *slong n*, *slong prec*)

Sets *res* to an upper incomplete gamma function where s is a constant and z is a power series, truncated to length n. The *regularized* argument has the same interpretation as in *acb_hypgeom_gamma_upper()*.

void **acb_hypgeom_gamma_lower**(*acb_t res*, **const** *acb_t s*, **const** *acb_t z*, int *regularized*, *slong prec*)

If *regularized* is 0, computes the lower incomplete gamma function $\gamma(s,z) = \frac{z^s}{s} {}_1F_1(s, s+1, -z)$.

If *regularized* is 1, computes the regularized lower incomplete gamma function $P(s,z) = \gamma(s,z)/\Gamma(s)$.

If *regularized* is 2, computes a further regularized lower incomplete gamma function $\gamma^*(s,z) = z^{-s}P(s,z)$.

void **_acb_hypgeom_gamma_lower_series**(*acb_ptr res*, **const** *acb_t s*, *acb_srcptr z*, *slong zlen*, int *regularized*, *slong n*, *slong prec*)

void **acb_hypgeom_gamma_lower_series**(*acb_poly_t res*, **const** *acb_t s*, **const** *acb_poly_t z*, int *regularized*, *slong n*, *slong prec*)

Sets *res* to an lower incomplete gamma function where s is a constant and z is a power series, truncated to length n. The *regularized* argument has the same interpretation as in *acb_hypgeom_gamma_lower()*.

void **acb_hypgeom_beta_lower**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t z*, int *regularized*, *slong prec*)

Computes the (lower) incomplete beta function, defined by $B(a,b;z) = \int_0^z t^{a-1}(1-t)^{b-1}$, optionally the regularized incomplete beta function $I(a,b;z) = B(a,b;z)/B(a,b;1)$.

In general, the integral must be interpreted using analytic continuation. The precise definitions for all parameter values are

$$B(a,b;z) = \frac{z^a}{a} {}_2F_1(a, 1-b, a+1, z)$$

$$I(a,b;z) = \frac{\Gamma(a+b)}{\Gamma(b)} z^a {}_2\widetilde{F}_1(a, 1-b, a+1, z).$$

Note that both functions with this definition are undefined for nonpositive integer a, and I is undefined for nonpositive integer $a+b$.

void **_acb_hypgeom_beta_lower_series**(*acb_ptr res*, **const** *acb_t a*, **const** *acb_t b*, *acb_srcptr z*, *slong zlen*, int *regularized*, *slong n*, *slong prec*)

void **acb_hypgeom_beta_lower_series**(*acb_poly_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_poly_t z*, int *regularized*, *slong n*, *slong prec*)

Sets *res* to the lower incomplete beta function $B(a,b;z)$ (optionally the regularized version $I(a,b;z)$) where a and b are constants and z is a power series, truncating the result to length n. The underscore method requires positive lengths and does not support aliasing.

9.1.11 Exponential and trigonometric integrals

The branch cut conventions of the following functions match Mathematica.

void **acb_hypgeom_expint**(*acb_t res*, **const** *acb_t s*, **const** *acb_t z*, *slong prec*)

Computes the generalized exponential integral $E_s(z)$. This is a trivial wrapper of *acb_hypgeom_gamma_upper()*.

void **acb_hypgeom_ei_asymp**(*acb_t res*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_ei_2f2**(*acb_t res*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_ei**(*acb_t res*, **const** *acb_t z*, *slong prec*)

Computes the exponential integral $\mathrm{Ei}(z)$, respectively using

$$\mathrm{Ei}(z) = -e^z U(1,1,-z) - \log(-z) + \frac{1}{2}\left(\log(z) - \log\left(\frac{1}{z}\right)\right)$$

$$\mathrm{Ei}(z) = z\,{}_2F_2(1,1;2,2;z) + \gamma + \frac{1}{2}\left(\log(z) - \log\left(\frac{1}{z}\right)\right)$$

and an automatic algorithm choice.

void **_acb_hypgeom_ei_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_ei_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong len*, *slong prec*)

Computes the exponential integral of the power series z, truncated to length *len*.

void **acb_hypgeom_si_asymp**(*acb_t res*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_si_1f2**(*acb_t res*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_si**(*acb_t res*, **const** *acb_t z*, *slong prec*)

Computes the sine integral $\mathrm{Si}(z)$, respectively using

$$\mathrm{Si}(z) = \frac{i}{2}\left[e^{iz}U(1,1,-iz) - e^{-iz}U(1,1,iz) + \log(-iz) - \log(iz)\right]$$

$$\mathrm{Si}(z) = z\,{}_1F_2\left(\tfrac{1}{2}; \tfrac{3}{2}, \tfrac{3}{2}; -\tfrac{z^2}{4}\right)$$

and an automatic algorithm choice.

void **_acb_hypgeom_si_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_si_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong len*, *slong prec*)
Computes the sine integral of the power series z, truncated to length *len*.

void **acb_hypgeom_ci_asymp**(*acb_t res*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_ci_2f3**(*acb_t res*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_ci**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes the cosine integral $\operatorname{Ci}(z)$, respectively using

$$\operatorname{Ci}(z) = \log(z) - \frac{1}{2}\left[e^{iz}U(1,1,-iz) + e^{-iz}U(1,1,iz) + \log(-iz) + \log(iz)\right]$$

$$\operatorname{Ci}(z) = -\tfrac{z^2}{4}\,{}_2F_3(1,1;2,2,\tfrac{3}{2};-\tfrac{z^2}{4}) + \log(z) + \gamma$$

and an automatic algorithm choice.

void **_acb_hypgeom_ci_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_ci_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong len*, *slong prec*)
Computes the cosine integral of the power series z, truncated to length *len*.

void **acb_hypgeom_shi**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes the hyperbolic sine integral $\operatorname{Shi}(z) = -i\operatorname{Si}(iz)$. This is a trivial wrapper of *acb_hypgeom_si()*.

void **_acb_hypgeom_shi_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_shi_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong len*, *slong prec*)
Computes the hyperbolic sine integral of the power series z, truncated to length *len*.

void **acb_hypgeom_chi_asymp**(*acb_t res*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_chi_2f3**(*acb_t res*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_chi**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes the hyperbolic cosine integral $\operatorname{Chi}(z)$, respectively using

$$\operatorname{Chi}(z) = -\frac{1}{2}\left[e^{z}U(1,1,-z) + e^{-z}U(1,1,z) + \log(-z) - \log(z)\right]$$

$$\operatorname{Chi}(z) = \tfrac{z^2}{4}\,{}_2F_3(1,1;2,2,\tfrac{3}{2};\tfrac{z^2}{4}) + \log(z) + \gamma$$

and an automatic algorithm choice.

void **_acb_hypgeom_chi_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **acb_hypgeom_chi_series**(*acb_poly_t res*, **const** *acb_poly_t z*, *slong len*, *slong prec*)
Computes the hyperbolic cosine integral of the power series z, truncated to length *len*.

void **acb_hypgeom_li**(*acb_t res*, **const** *acb_t z*, int *offset*, *slong prec*)
If *offset* is zero, computes the logarithmic integral $\operatorname{li}(z) = \operatorname{Ei}(\log(z))$.

If *offset* is nonzero, computes the offset logarithmic integral $\operatorname{Li}(z) = \operatorname{li}(z) - \operatorname{li}(2)$.

void **_acb_hypgeom_li_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, int *offset*, *slong len*, *slong prec*)

void **acb_hypgeom_li_series**(*acb_poly_t res*, **const** *acb_poly_t z*, int *offset*, *slong len*, *slong prec*)
Computes the logarithmic integral (optionally the offset version) of the power series z, truncated to length *len*.

9.1.12 Gauss hypergeometric function

The following methods compute the Gauss hypergeometric function

$$F(z) = {}_2F_1(a,b,c,z) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \frac{z^k}{k!}$$

or the regularized version $\mathbf{F}(z) = \mathbf{F}(a,b,c,z) = {}_2F_1(a,b,c,z)/\Gamma(c)$ if the flag *regularized* is set.

void **acb_hypgeom_2f1_continuation**(*acb_t res0*, *acb_t res1*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t c*, **const** *acb_t z0*, **const** *acb_t z1*, **const** *acb_t f0*, **const** *acb_t f1*, *slong prec*)

Given $F(z_0), F'(z_0)$ in *f0*, *f1*, sets *res0* and *res1* to $F(z_1), F'(z_1)$ by integrating the hypergeometric differential equation along a straight-line path. The evaluation points should be well-isolated from the singular points 0 and 1.

void **acb_hypgeom_2f1_series_direct**(*acb_poly_t res*, **const** *acb_poly_t a*, **const** *acb_poly_t b*, **const** *acb_poly_t c*, **const** *acb_poly_t z*, int *regularized*, *slong len*, *slong prec*)

Computes $F(z)$ of the given power series truncated to length *len*, using direct summation of the hypergeometric series.

void **acb_hypgeom_2f1_direct**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t c*, **const** *acb_t z*, int *regularized*, *slong prec*)

Computes $F(z)$ using direct summation of the hypergeometric series.

void **acb_hypgeom_2f1_transform**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t c*, **const** *acb_t z*, int *flags*, int *which*, *slong prec*)

void **acb_hypgeom_2f1_transform_limit**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t c*, **const** *acb_t z*, int *regularized*, int *which*, *slong prec*)

Computes $F(z)$ using an argument transformation determined by the flag *which*. Legal values are 1 for $z/(z-1)$, 2 for $1/z$, 3 for $1/(1-z)$, 4 for $1-z$, and 5 for $1-1/z$.

The *transform_limit* version assumes that *which* is not 1. If *which* is 2 or 3, it assumes that $b-a$ represents an exact integer. If *which* is 4 or 5, it assumes that $c-a-b$ represents an exact integer. In these cases, it computes the correct limit value.

See *acb_hypgeom_2f1()* for the meaning of *flags*.

void **acb_hypgeom_2f1_corner**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t c*, **const** *acb_t z*, int *regularized*, *slong prec*)

Computes $F(z)$ near the corner cases $\exp(\pm\pi i\sqrt{3})$ by analytic continuation.

int **acb_hypgeom_2f1_choose**(**const** *acb_t z*)

Chooses a method to compute the function based on the location of z in the complex plane. If the return value is 0, direct summation should be used. If the return value is 1 to 5, the transformation with this index in *acb_hypgeom_2f1_transform()* should be used. If the return value is 6, the corner case algorithm should be used.

void **acb_hypgeom_2f1**(*acb_t res*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t c*, **const** *acb_t z*, int *flags*, *slong prec*)

Computes $F(z)$ or $\mathbf{F}(z)$ using an automatic algorithm choice.

The following bit fields can be set in *flags*:

- *ACB_HYPGEOM_2F1_REGULARIZED* - computes the regularized hypergeometric function $\mathbf{F}(z)$. Setting *flags* to 1 is the same as just toggling this option.
- *ACB_HYPGEOM_2F1_AB* - $a-b$ is an integer.
- *ACB_HYPGEOM_2F1_ABC* - $a+b-c$ is an integer.
- *ACB_HYPGEOM_2F1_AC* - $a-c$ is an integer.
- *ACB_HYPGEOM_2F1_BC* - $b-c$ is an integer.

The last four flags can be set to indicate that the respective parameter differences are known to represent exact integers, even if the input intervals are inexact. This allows the correct limits to be evaluated when applying transformation formulas. For example, to evaluate ${}_2F_1(\sqrt{2}, 1/2, \sqrt{2}+3/2, 9/10)$, the *ABC* flag should be set. If not set, the result will be an indeterminate interval due to internally dividing by an interval containing zero. If the parameters are exact floating-point numbers (including exact integers or half-integers), then the limits are computed automatically, and setting these flags is unnecessary.

Currently, only the *AB* and *ABC* flags are used this way; the *AC* and *BC* flags might be used in the future.

9.1.13 Orthogonal polynomials and functions

void **acb_hypgeom_chebyshev_t**(*acb_t res*, **const** *acb_t n*, **const** *acb_t z*, *slong prec*)

void **acb_hypgeom_chebyshev_u**(*acb_t res*, **const** *acb_t n*, **const** *acb_t z*, *slong prec*)

Computes the Chebyshev polynomial (or Chebyshev function) of first or second kind

$$T_n(z) = {}_2F_1\left(-n, n, \frac{1}{2}, \frac{1-z}{2}\right)$$

$$U_n(z) = (n+1){}_2F_1\left(-n, n+2, \frac{3}{2}, \frac{1-z}{2}\right).$$

The hypergeometric series definitions are only used for computation near the point 1. In general, trigonometric representations are used. For word-size integer *n*, *acb_chebyshev_t_ui()* and *acb_chebyshev_u_ui()* are called.

void **acb_hypgeom_jacobi_p**(*acb_t res*, **const** *acb_t n*, **const** *acb_t a*, **const** *acb_t b*, **const** *acb_t z*, *slong prec*)

Computes the Jacobi polynomial (or Jacobi function)

$$P_n^{(a,b)}(z) = \frac{(a+1)_n}{\Gamma(n+1)} {}_2F_1\left(-n, n+a+b+1, a+1, \frac{1-z}{2}\right).$$

For nonnegative integer *n*, this is a polynomial in *a*, *b* and *z*, even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

void **acb_hypgeom_gegenbauer_c**(*acb_t res*, **const** *acb_t n*, **const** *acb_t m*, **const** *acb_t z*, *slong prec*)

Computes the Gegenbauer polynomial (or Gegenbauer function)

$$C_n^m(z) = \frac{(2m)_n}{\Gamma(n+1)} {}_2F_1\left(-n, 2m+n, m+\frac{1}{2}, \frac{1-z}{2}\right).$$

For nonnegative integer *n*, this is a polynomial in *m* and *z*, even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

void **acb_hypgeom_laguerre_l**(*acb_t res*, **const** *acb_t n*, **const** *acb_t m*, **const** *acb_t z*, *slong prec*)

Computes the Laguerre polynomial (or Laguerre function)

$$L_n^m(z) = \frac{(m+1)_n}{\Gamma(n+1)} {}_1F_1\left(-n, m+1, z\right).$$

For nonnegative integer *n*, this is a polynomial in *m* and *z*, even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

There are at least two incompatible ways to define the Laguerre function when *n* is a negative integer. One possibility when $m = 0$ is to define $L_{-n}^0(z) = e^z L_{n-1}^0(-z)$. Another possibility is to cover this case with the recurrence relation $L_{n-1}^m(z) + L_n^{m-1}(z) = L_n^m(z)$. Currently, we leave this case undefined (returning indeterminate).

void **acb_hypgeom_hermite_h**(*acb_t res*, **const** *acb_t n*, **const** *acb_t z*, *slong prec*)
Computes the Hermite polynomial (or Hermite function)

$$H_n(z) = 2^n \sqrt{\pi} \left(\frac{1}{\Gamma((1-n)/2)} {}_1F_1\left(-\frac{n}{2}, \frac{1}{2}, z^2\right) - \frac{2z}{\Gamma(-n/2)} {}_1F_1\left(\frac{1-n}{2}, \frac{3}{2}, z^2\right)\right).$$

void **acb_hypgeom_legendre_p**(*acb_t res*, **const** *acb_t n*, **const** *acb_t m*, **const** *acb_t z*, int *type*, *slong prec*)
Sets *res* to the associated Legendre function of the first kind evaluated for degree n, order m, and argument z. When m is zero, this reduces to the Legendre polynomial $P_n(z)$.

Many different branch cut conventions appear in the literature. If *type* is 0, the version

$$P_n^m(z) = \frac{(1+z)^{m/2}}{(1-z)^{m/2}} \mathbf{F}\left(-n, n+1, 1-m, \frac{1-z}{2}\right)$$

is computed, and if *type* is 1, the alternative version

$$\mathcal{P}_n^m(z) = \frac{(z+1)^{m/2}}{(z-1)^{m/2}} \mathbf{F}\left(-n, n+1, 1-m, \frac{1-z}{2}\right).$$

is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in *Mathematica* and *mpmath*.

void **acb_hypgeom_legendre_q**(*acb_t res*, **const** *acb_t n*, **const** *acb_t m*, **const** *acb_t z*, int *type*, *slong prec*)
Sets *res* to the associated Legendre function of the second kind evaluated for degree n, order m, and argument z. When m is zero, this reduces to the Legendre function $Q_n(z)$.

Many different branch cut conventions appear in the literature. If *type* is 0, the version

$$Q_n^m(z) = \frac{\pi}{2\sin(\pi m)} \left(\cos(\pi m) P_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} P_n^{-m}(z) \right)$$

is computed, and if *type* is 1, the alternative version

$$\mathcal{Q}_n^m(z) = \frac{\pi}{2\sin(\pi m)} e^{\pi i m} \left(\mathcal{P}_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} \mathcal{P}_n^{-m}(z) \right)$$

is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in *Mathematica* and *mpmath*.

When m is an integer, either expression is interpreted as a limit. We make use of the connection formulas [WQ3a], [WQ3b] and [WQ3c] to allow computing the function even in the limiting case. (The formula [WQ3d] would be useful, but is incorrect in the lower half plane.)

void **acb_hypgeom_legendre_p_uiui_rec**(*acb_t res*, *ulong n*, *ulong m*, **const** *acb_t z*, *slong prec*)
For nonnegative integer n and m, uses recurrence relations to evaluate $(1-z^2)^{-m/2} P_n^m(z)$ which is a polynomial in z.

void **acb_hypgeom_spherical_y**(*acb_t res*, *slong n*, *slong m*, **const** *acb_t theta*, **const** *acb_t phi*, *slong prec*)
Computes the spherical harmonic of degree n, order m, latitude angle *theta*, and longitude angle *phi*, normalized such that

$$Y_n^m(\theta, \phi) = \sqrt{\frac{2n+1}{4\pi} \frac{(n-m)!}{(n+m)!}} e^{im\phi} P_n^m(\cos(\theta)).$$

The definition is extended to negative m and n by symmetry. This function is a polynomial in $\cos(\theta)$ and $\sin(\theta)$. We evaluate it using *acb_hypgeom_legendre_p_uiui_rec()*.

9.1.14 Dilogarithm

The dilogarithm function is given by $\mathrm{Li}_2(z) = -\int_0^z \frac{\log(1-t)}{t} dt = z\,{}_3F_2(1,1,1,2,2,z)$.

void **acb_hypgeom_dilog_bernoulli**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes the dilogarithm using a series expansion in $w = \log(z)$, with rate of convergence $|w/(2\pi)|^n$. This provides good convergence near $z = e^{\pm i\pi/3}$, where hypergeometric series expansions fail. Since the coefficients involve Bernoulli numbers, this method should only be used at moderate precision.

void **acb_hypgeom_dilog_zero_taylor**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes the dilogarithm for z close to 0 using the hypergeometric series (effective only when $|z| \ll 1$).

void **acb_hypgeom_dilog_zero**(*acb_t res*, **const** *acb_t z*, *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.

void **acb_hypgeom_dilog_transform**(*acb_t res*, **const** *acb_t z*, int *algorithm*, *slong prec*)
Computes the dilogarithm by applying one of the transformations $1/z$, $1-z$, $z/(z-1)$, $1/(1-z)$, indexed by *algorithm* from 1 to 4, and calling *acb_hypgeom_dilog_zero()* with the reduced variable. Alternatively, for *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.

void **acb_hypgeom_dilog_continuation**(*acb_t res*, **const** *acb_t a*, **const** *acb_t z*, *slong prec*)
Computes $\mathrm{Li}_2(z) - \mathrm{Li}_2(a)$ 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.

void **acb_hypgeom_dilog_bitburst**(*acb_t res*, *acb_t z0*, **const** *acb_t z*, *slong prec*)
Sets *z0* to a point with short bit expansion close to z and sets *res* to $\mathrm{Li}_2(z) - \mathrm{Li}_2(z_0)$, computed using the bit-burst algorithm.

void **acb_hypgeom_dilog**(*acb_t res*, **const** *acb_t z*, *slong prec*)
Computes the dilogarithm using a default algorithm choice.

9.2 arb_hypgeom.h – hypergeometric functions of real variables

See *acb_hypgeom.h – hypergeometric functions of complex variables* for the general implementation of hypergeometric functions.

For convenience, this module provides versions of the same functions for real variables represented using *arb_t* and *arb_poly_t*. Most methods are simple wrappers around the complex versions, but some of the functions in this module have been further optimized specifically for real variables.

This module also provides certain functions exclusive to real variables, such as functions for computing real roots of common special functions.

9.2.1 Generalized hypergeometric function

void **arb_hypgeom_pfq**(*arb_t res*, *arb_srcptr a*, *slong p*, *arb_srcptr b*, *slong q*, **const** *arb_t z*, int *regularized*, *slong prec*)
Computes the generalized hypergeometric function ${}_pF_q(z)$, or the regularized version if *regularized* is set.

9.2.2 Confluent hypergeometric functions

void **arb_hypgeom_0f1**(*arb_t res*, **const** *arb_t a*, **const** *arb_t z*, int *regularized*, *slong prec*)
Computes the confluent hypergeometric limit function ${}_0F_1(a,z)$, or $\frac{1}{\Gamma(a)}{}_0F_1(a,z)$ if *regularized* is set.

void **arb_hypgeom_m**(*arb_t res*, **const** *arb_t a*, **const** *arb_t b*, **const** *arb_t z*, int *regularized*, *slong prec*)
Computes the confluent hypergeometric function $M(a,b,z) = {}_1F_1(a,b,z)$, or $\mathbf{M}(a,b,z) = \frac{1}{\Gamma(b)}{}_1F_1(a,b,z)$ if *regularized* is set.

void **arb_hypgeom_1f1**(*arb_t res*, **const** *arb_t a*, **const** *arb_t b*, **const** *arb_t z*, int *regularized*, *slong prec*)
Alias for *arb_hypgeom_m()*.

void **arb_hypgeom_u**(*arb_t res*, **const** *arb_t a*, **const** *arb_t b*, **const** *arb_t z*, *slong prec*)
Computes the confluent hypergeometric function $U(a,b,z)$.

9.2.3 Gauss hypergeometric function

void **arb_hypgeom_2f1**(*arb_t res*, **const** *arb_t a*, **const** *arb_t b*, **const** *arb_t c*, **const** *arb_t z*, int *regularized*, *slong prec*)
Computes the Gauss hypergeometric function ${}_2F_1(a,b,c,z)$, or $\mathbf{F}(a,b,c,z) = \frac{1}{\Gamma(c)}{}_2F_1(a,b,c,z)$ if *regularized* is set.

Additional evaluation flags can be passed via the *regularized* argument; see *acb_hypgeom_2f1()* for documentation.

9.2.4 Error functions and Fresnel integrals

void **arb_hypgeom_erf**(*arb_t res*, **const** *arb_t z*, *slong prec*)
Computes the error function $\operatorname{erf}(z)$.

void **_arb_hypgeom_erf_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_erf_series**(*arb_poly_t res*, **const** *arb_poly_t z*, *slong len*, *slong prec*)
Computes the error function of the power series z, truncated to length *len*.

void **arb_hypgeom_erfc**(*arb_t res*, **const** *arb_t z*, *slong prec*)
Computes the complementary error function $\operatorname{erfc}(z) = 1 - \operatorname{erf}(z)$. This function avoids catastrophic cancellation for large positive z.

void **_arb_hypgeom_erfc_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_erfc_series**(*arb_poly_t res*, **const** *arb_poly_t z*, *slong len*, *slong prec*)
Computes the complementary error function of the power series z, truncated to length *len*.

void **arb_hypgeom_erfi**(*arb_t res*, **const** *arb_t z*, *slong prec*)
Computes the imaginary error function $\operatorname{erfi}(z) = -i \operatorname{erf}(iz)$.

void **_arb_hypgeom_erfi_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_erfi_series**(*arb_poly_t res*, **const** *arb_poly_t z*, *slong len*, *slong prec*)
Computes the imaginary error function of the power series z, truncated to length *len*.

void **arb_hypgeom_fresnel**(*arb_t res1*, *arb_t res2*, **const** *arb_t z*, int *normalized*, *slong prec*)
Sets *res1* to the Fresnel sine integral $S(z)$ and *res2* to the Fresnel cosine integral $C(z)$. Optionally, just a single function can be computed by passing *NULL* as the other output variable. The definition $S(z) = \int_0^z \sin(t^2)dt$ is used if *normalized* is 0, and $S(z) = \int_0^z \sin(\frac{1}{2}\pi t^2)dt$ is used if *normalized* is 1 (the latter is the Abramowitz & Stegun convention). $C(z)$ is defined analogously.

void **_arb_hypgeom_fresnel_series**(*arb_ptr res1*, *arb_ptr res2*, *arb_srcptr z*, *slong zlen*, int *normalized*, *slong len*, *slong prec*)

void **arb_hypgeom_fresnel_series**(*arb_poly_t res1*, *arb_poly_t res2*, **const** *arb_poly_t z*, int *normalized*, *slong len*, *slong prec*)
Sets *res1* to the Fresnel sine integral and *res2* to the Fresnel cosine integral of the power series *z*, truncated to length *len*. Optionally, just a single function can be computed by passing *NULL* as the other output variable.

9.2.5 Incomplete gamma and beta functions

void **arb_hypgeom_gamma_upper**(*arb_t res*, **const** *arb_t s*, **const** *arb_t z*, int *regularized*, *slong prec*)
If *regularized* is 0, computes the upper incomplete gamma function $\Gamma(s, z)$.

If *regularized* is 1, computes the regularized upper incomplete gamma function $Q(s, z) = \Gamma(s, z)/\Gamma(s)$.

If *regularized* is 2, computes the generalized exponential integral $z^{-s}\Gamma(s, z) = E_{1-s}(z)$ instead (this option is mainly intended for internal use; *arb_hypgeom_expint()* is the intended interface for computing the exponential integral).

void **_arb_hypgeom_gamma_upper_series**(*arb_ptr res*, **const** *arb_t s*, *arb_srcptr z*, *slong zlen*, int *regularized*, *slong n*, *slong prec*)

void **arb_hypgeom_gamma_upper_series**(*arb_poly_t res*, **const** *arb_t s*, **const** *arb_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 *arb_hypgeom_gamma_upper()*.

void **arb_hypgeom_gamma_lower**(*arb_t res*, **const** *arb_t s*, **const** *arb_t z*, int *regularized*, *slong prec*)
If *regularized* is 0, computes the lower incomplete gamma function $\gamma(s, z) = \frac{z^s}{s}{}_1F_1(s, s+1, -z)$.

If *regularized* is 1, computes the regularized lower incomplete gamma function $P(s, z) = \gamma(s, z)/\Gamma(s)$.

If *regularized* is 2, computes a further regularized lower incomplete gamma function $\gamma^*(s, z) = z^{-s}P(s, z)$.

void **_arb_hypgeom_gamma_lower_series**(*arb_ptr res*, **const** *arb_t s*, *arb_srcptr z*, *slong zlen*, int *regularized*, *slong n*, *slong prec*)

void **arb_hypgeom_gamma_lower_series**(*arb_poly_t res*, **const** *arb_t s*, **const** *arb_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 *arb_hypgeom_gamma_lower()*.

void **arb_hypgeom_beta_lower**(*arb_t res*, **const** *arb_t a*, **const** *arb_t b*, **const** *arb_t z*, int *regularized*, *slong prec*)
Computes the (lower) incomplete beta function, defined by $B(a, b; z) = \int_0^z t^{a-1}(1-t)^{b-1}$, optionally the regularized incomplete beta function $I(a, b; z) = B(a, b; z)/B(a, b; 1)$.

void **_arb_hypgeom_beta_lower_series**(*arb_ptr res*, **const** *arb_t a*, **const** *arb_t b*, *arb_srcptr z*, *slong zlen*, int *regularized*, *slong n*, *slong prec*)

void **arb_hypgeom_beta_lower_series**(*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.

9.2.6 Exponential and trigonometric integrals

void **arb_hypgeom_expint**(*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**(*arb_t res*, **const** *arb_t z*, *slong prec*)
Computes the exponential integral $\operatorname{Ei}(z)$.

void **_arb_hypgeom_ei_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_ei_series**(*arb_poly_t res*, **const** *arb_poly_t z*, *slong len*, *slong prec*)
Computes the exponential integral of the power series z, truncated to length *len*.

void **arb_hypgeom_si**(*arb_t res*, **const** *arb_t z*, *slong prec*)
Computes the sine integral $\operatorname{Si}(z)$.

void **_arb_hypgeom_si_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_si_series**(*arb_poly_t res*, **const** *arb_poly_t z*, *slong len*, *slong prec*)
Computes the sine integral of the power series z, truncated to length *len*.

void **arb_hypgeom_ci**(*arb_t res*, **const** *arb_t z*, *slong prec*)
Computes the cosine integral $\operatorname{Ci}(z)$. The result is indeterminate if $z < 0$ since the value of the function would be complex.

void **_arb_hypgeom_ci_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_ci_series**(*arb_poly_t res*, **const** *arb_poly_t z*, *slong len*, *slong prec*)
Computes the cosine integral of the power series z, truncated to length *len*.

void **arb_hypgeom_shi**(*arb_t res*, **const** *arb_t z*, *slong prec*)
Computes the hyperbolic sine integral $\operatorname{Shi}(z) = -i \operatorname{Si}(iz)$.

void **_arb_hypgeom_shi_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_shi_series**(*arb_poly_t res*, **const** *arb_poly_t z*, *slong len*, *slong prec*)
Computes the hyperbolic sine integral of the power series z, truncated to length *len*.

void **arb_hypgeom_chi**(*arb_t res*, **const** *arb_t z*, *slong prec*)
Computes the hyperbolic cosine integral $\operatorname{Chi}(z)$. The result is indeterminate if $z < 0$ since the value of the function would be complex.

void **_arb_hypgeom_chi_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_chi_series**(*arb_poly_t res*, **const** *arb_poly_t z*, *slong len*, *slong prec*)
Computes the hyperbolic cosine integral of the power series z, truncated to length *len*.

void **arb_hypgeom_li**(*arb_t res*, **const** *arb_t z*, int *offset*, *slong prec*)
If *offset* is zero, computes the logarithmic integral $\operatorname{li}(z) = \operatorname{Ei}(\log(z))$.

If *offset* is nonzero, computes the offset logarithmic integral $\operatorname{Li}(z) = \operatorname{li}(z) - \operatorname{li}(2)$.

The result is indeterminate if $z < 0$ since the value of the function would be complex.

void **_arb_hypgeom_li_series**(*arb_ptr res*, *arb_srcptr z*, *slong zlen*, int *offset*, *slong len*, *slong prec*)

void **arb_hypgeom_li_series**(*arb_poly_t res*, **const** *arb_poly_t z*, int *offset*, *slong len*, *slong prec*)
Computes the logarithmic integral (optionally the offset version) of the power series z, truncated to length *len*.

9.2.7 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 $I_\nu(z) = z^\nu (iz)^{-\nu} J_\nu(iz)$.

void **arb_hypgeom_bessel_i_scaled**(*arb_t res*, **const** *arb_t nu*, **const** *arb_t z*, *slong prec*)
Computes the function $e^{-z} I_\nu(z)$.

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 $K_\nu(z)$.

void **arb_hypgeom_bessel_k_scaled**(*arb_t res*, **const** *arb_t nu*, **const** *arb_t z*, *slong prec*)
Computes the function $e^{z} K_\nu(z)$.

9.2.8 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 $(\mathrm{Ai}(z), \mathrm{Ai}'(z), \mathrm{Bi}(z), \mathrm{Bi}'(z))$ simultaneously. Any of the four function values can be omitted by passing *NULL* for the unwanted output variables, speeding up the evaluation.

void **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_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_airy_series**(*arb_poly_t ai*, *arb_poly_t ai_prime*, *arb_poly_t bi*, *arb_poly_t bi_prime*, **const** *arb_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*.

void **arb_hypgeom_airy_zero**(*arb_t a*, *arb_t a_prime*, *arb_t b*, *arb_t b_prime*, **const** *fmpz_t n*, *slong prec*)
Computes the n-th real zero a_n, a'_n, b_n, or b'_n for the respective Airy function or Airy function derivative. Any combination of the four output variables can be *NULL*. The zeros are indexed by increasing magnitude, starting with $n = 1$ to follow the convention in the literature. An index n that is not positive is invalid input. The implementation uses asymptotic expansions for the zeros [PS1991] together with the interval Newton method for refinement.

9.2.9 Coulomb wave functions

void **arb_hypgeom_coulomb**(*arb_t F*, *arb_t G*, **const** *arb_t l*, **const** *arb_t eta*, **const** *arb_t z*, *slong prec*)

Writes to F, G the values of the respective Coulomb wave functions $F_\ell(\eta, z)$ and $G_\ell(\eta, z)$. Either of the outputs can be *NULL*.

void **arb_hypgeom_coulomb_jet**(*arb_ptr F*, *arb_ptr G*, **const** *arb_t l*, **const** *arb_t eta*, **const** *arb_t z*, *slong len*, *slong prec*)

Writes to F, G the respective Taylor expansions of the Coulomb wave functions at the point z, truncated to length *len*. Either of the outputs can be *NULL*.

void **_arb_hypgeom_coulomb_series**(*arb_ptr F*, *arb_ptr G*, **const** *arb_t l*, **const** *arb_t eta*, *arb_srcptr z*, *slong zlen*, *slong len*, *slong prec*)

void **arb_hypgeom_coulomb_series**(*arb_poly_t F*, *arb_poly_t G*, **const** *arb_t l*, **const** *arb_t eta*, **const** *arb_poly_t z*, *slong len*, *slong prec*)

Computes the Coulomb wave functions evaluated at the power series z, truncated to length *len*. Either of the outputs can be *NULL*.

9.2.10 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 **arb_hypgeom_legendre_q**(*arb_t res*, **const** *arb_t n*, **const** *arb_t m*, **const** *arb_t z*, int *type*, *slong prec*)

Computes Legendre functions of the first and second kind. See *acb_hypgeom_legendre_p()* and *acb_hypgeom_legendre_q()* for definitions.

void **arb_hypgeom_legendre_p_ui_deriv_bound**(*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 $|x| \le 1$. 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 **arb_hypgeom_legendre_p_ui_zero**(*arb_t res*, *arb_t res_prime*, *ulong n*, **const** *arb_t x*, *slong K*, *slong prec*)

void **arb_hypgeom_legendre_p_ui_one**(*arb_t res*, *arb_t res_prime*, *ulong n*, **const** *arb_t x*, *slong K*, *slong prec*)

void **arb_hypgeom_legendre_p_ui_asymp**(*arb_t res*, *arb_t res_prime*, *ulong n*, **const** *arb_t x*, *slong K*, *slong prec*)

void **arb_hypgeom_legendre_p_rec**(*arb_t res*, *arb_t res_prime*, *ulong n*, **const** *arb_t x*, *slong prec*)

void **arb_hypgeom_legendre_p_ui**(*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 *zero*, *one* respectively use the hypergeometric series expansions at $x = 0$ and $x = 1$ while the *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 *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 **arb_hypgeom_legendre_p_ui_root**(*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 θ). If *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 $\lceil n/2 \rceil$ 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.

9.2.11 Dilogarithm

void **arb_hypgeom_dilog**(*arb_t res*, **const** *arb_t z*, *slong prec*)

Computes the dilogarithm $\mathrm{Li}_2(z)$.

9.2.12 Hypergeometric sequences

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

Computes the central binomial coefficient $\binom{2n}{n}$.

9.3 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.

9.3.1 Complete elliptic integrals

void **acb_elliptic_k**(*acb_t res*, **const** *acb_t m*, *slong prec*)
Computes the complete elliptic integral of the first kind

$$K(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m\sin^2 t}} = \int_0^1 \frac{dt}{\left(\sqrt{1-t^2}\right)\left(\sqrt{1-mt^2}\right)}$$

using the arithmetic-geometric mean: $K(m) = \pi/(2M(\sqrt{1-m}))$.

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. $K(m+x) \in \mathbb{C}[[x]]$.

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*.

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).

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.

9.3.2 Legendre incomplete elliptic integrals

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

$$F(\phi, m) = \int_0^{\phi} \frac{dt}{\sqrt{1 - m\sin^2 t}} = \int_0^{\sin\phi} \frac{dt}{\left(\sqrt{1-t^2}\right)\left(\sqrt{1-mt^2}\right)}$$

on the standard strip $-\pi/2 \le \mathrm{Re}(\phi) \le \pi/2$. 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 ϕ 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 **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

$$E(\phi, m) = \int_0^{\phi} \sqrt{1 - m \sin^2 t}\, dt = \int_0^{\sin \phi} \frac{\sqrt{1 - mt^2}}{\sqrt{1 - t^2}}\, dt$$

on the standard strip $-\pi/2 \le \operatorname{Re}(\phi) \le \pi/2$. 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 ϕ 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 **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 $-\pi/2 \le \operatorname{Re}(\phi) \le \pi/2$. 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 ϕ 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, $\Pi\left(n, \frac{\pi}{2}, m\right) = \Pi(n, m)$.

9.3.3 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, and 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 $x, y, z \notin (-\infty, 0)$, 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*)

void **acb_elliptic_rj_carlson**(*acb_t res*, **const** *acb_t x*, **const** *acb_t y*, **const** *acb_t z*, **const** *acb_t p*, int *flags*, *slong prec*)

void **acb_elliptic_rj_integration**(*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.

Three versions of this function are available: the *carlson* version applies one or more duplication steps 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 *integration* algorithm uses explicit numerical integration to translate the parameters to the right half-plane. This is reliable but can be slow.

The default method uses the *carlson* algorithm when it is certain to be correct, and otherwise falls back to the slow *integration* algorithm.

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 *carlson* 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 $R_C(1,1+x) = \operatorname{atan}(\sqrt{x})/\sqrt{x} = {}_2F_1(1,1/2,3/2,-x)$, which is needed in the evaluation of R_J.

9.3.4 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 ω_1, ω_2. We simplify by setting $2\omega_1 = 1, 2\omega_2 = \tau$ with $\operatorname{im}(\tau) > 0$. 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].

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

$$\wp(z,\tau) = \pi^2 \theta_2^2(0,\tau)\theta_3^2(0,\tau)\frac{\theta_4^2(z,\tau)}{\theta_1^2(z,\tau)} - \frac{\pi^2}{3}\left[\theta_2^4(0,\tau) + \theta_3^4(0,\tau)\right].$$

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 *len*. In particular, with $len = 2$, simultaneously computes $\wp(z,\tau), \wp'(z,\tau)$ which together generate the field of elliptic functions with periods 1 and τ.

void **_acb_elliptic_p_series**(*acb_ptr res*, *acb_srcptr z*, *slong zlen*, **const** *acb_t tau*, *slong len*, *slong prec*)

void **acb_elliptic_p_series**(*acb_poly_t res*, **const** *acb_poly_t z*, **const** *acb_t tau*, *slong len*, *slong prec*)
Sets *res* to the Weierstrass elliptic function of the power series z, with periods 1 and *tau*, truncated to length *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 *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 $4z^3 - g_2 z - g_3$.

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

$$\sigma(z,\tau) = z \prod_{n^2+m^2 \neq 0} \left[\left(1 - \frac{z}{m+n\tau}\right) \exp\left(\frac{z}{m+n\tau} + \frac{z^2}{2(m+n\tau)^2}\right) \right]$$

which is quasiperiodic with $\sigma(z+1,\tau) = -e^{2\zeta(1/2,\tau)(z+1/2)}\sigma(z,\tau)$ and $\sigma(z+\tau,\tau) = -e^{2\zeta(\tau/2,\tau)(z+\tau/2)}\sigma(z,\tau)$.

9.4 acb_modular.h – modular forms of complex variables

This module provides methods for numerical evaluation of modular forms and Jacobi theta functions. See *acb_elliptic.h* for the closely related elliptic functions and integrals.

In the context of this module, *tau* or τ always denotes an element of the complex upper half-plane $\mathbb{H} = \{z \in \mathbb{C} : \mathrm{Im}(z) > 0\}$. We also often use the variable q, variously defined as $q = e^{2\pi i\tau}$ (usually in relation to modular forms) or $q = e^{\pi i\tau}$ (usually in relation to theta functions) and satisfying $|q| < 1$. We will clarify the local meaning of q every time such a quantity appears as a function of τ.

As usual, the numerical functions in this module compute strict error bounds: if *tau* is represented by an `acb_t` whose content overlaps with the real line (or lies in the lower half-plane), and *tau* is passed to a function defined only on $\mathbb{H}$, then the output will have an infinite radius. The analogous behavior holds for functions requiring $|q| < 1$.

9.4.1 The modular group

type psl2z_struct

type psl2z_t
Represents an element of the modular group $\mathrm{PSL}(2, \mathbb{Z})$, namely an integer matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

with $ad - bc = 1$, and with signs canonicalized such that $c \geq 0$, and $d > 0$ if $c = 0$. The struct members a, b, c, d are of type *fmpz*.

void **psl2z_init**(*psl2z_t g*)
Initializes g and set it to the identity element.

void **psl2z_clear**(*psl2z_t g*)
Clears g.

void **psl2z_swap**(*psl2z_t f*, *psl2z_t g*)
Swaps f and g efficiently.

void **psl2z_set**(*psl2z_t f*, **const** *psl2z_t g*)
Sets f to a copy of g.

void **psl2z_one**(*psl2z_t g*)
Sets g to the identity element.

int **psl2z_is_one**(**const** *psl2z_t g*)
Returns nonzero iff g is the identity element.

void **psl2z_print**(**const** *psl2z_t g*)
Prints g to standard output.

void **psl2z_fprint**(FILE **file*, **const** *psl2z_t g*)
Prints g to the stream *file*.

int **psl2z_equal**(**const** *psl2z_t f*, **const** *psl2z_t g*)
Returns nonzero iff f and g are equal.

void **psl2z_mul**(*psl2z_t h*, **const** *psl2z_t f*, **const** *psl2z_t g*)
Sets h to the product of f and g, namely the matrix product with the signs canonicalized.

void **psl2z_inv**(*psl2z_t h*, **const** *psl2z_t g*)
Sets h to the inverse of g.

int **psl2z_is_correct**(**const** *psl2z_t g*)
Returns nonzero iff g contains correct data, i.e. satisfying $ad - bc = 1$, $c \geq 0$, and $d > 0$ if $c = 0$.

void **psl2z_randtest**(*psl2z_t g*, flint_rand_t *state*, *slong bits*)
Sets g to a random element of $\mathrm{PSL}(2,\mathbb{Z})$ with entries of bit length at most *bits* (or 1, if *bits* is not positive). We first generate a and d, compute their Bezout coefficients, divide by the GCD, and then correct the signs.

9.4.2 Modular transformations

void **acb_modular_transform**(*acb_t w*, **const** *psl2z_t g*, **const** *acb_t z*, *slong prec*)
Applies the modular transformation g to the complex number z, evaluating

$$w = gz = \frac{az+b}{cz+d}.$$

void **acb_modular_fundamental_domain_approx_d**(*psl2z_t g*, double *x*, double *y*, double *one_minus_eps*)

void **acb_modular_fundamental_domain_approx_arf**(*psl2z_t g*, **const** *arf_t x*, **const** *arf_t y*, **const** *arf_t one_minus_eps*, *slong prec*)
Attempts to determine a modular transformation g that maps the complex number $x+yi$ to the fundamental domain or just slightly outside the fundamental domain, where the target tolerance (not a strict bound) is specified by *one_minus_eps*.

The inputs are assumed to be finite numbers, with y positive.

Uses floating-point iteration, repeatedly applying either the transformation $z \leftarrow z+b$ or $z \leftarrow -1/z$. The iteration is terminated if $|x| \le 1/2$ and $x^2+y^2 \ge 1-\varepsilon$ where $1-\varepsilon$ is passed as *one_minus_eps*. It is also terminated if too many steps have been taken without convergence, or if the numbers end up too large or too small for the working precision.

The algorithm can fail to produce a satisfactory transformation. The output g is always set to *some* correct modular transformation, but it is up to the user to verify a posteriori that g maps $x+yi$ close enough to the fundamental domain.

void **acb_modular_fundamental_domain_approx**(*acb_t w*, *psl2z_t g*, **const** *acb_t z*, **const** *arf_t one_minus_eps*, *slong prec*)
Attempts to determine a modular transformation g that maps the complex number z to the fundamental domain or just slightly outside the fundamental domain, where the target tolerance (not a strict bound) is specified by *one_minus_eps*. It also computes the transformed value $w = gz$.

This function first tries to use *acb_modular_fundamental_domain_approx_d()* and checks if the result is acceptable. If this fails, it calls *acb_modular_fundamental_domain_approx_arf()* with higher precision. Finally, $w = gz$ is evaluated by a single application of g.

The algorithm can fail to produce a satisfactory transformation. The output g is always set to *some* correct modular transformation, but it is up to the user to verify a posteriori that w is close enough to the fundamental domain.

int **acb_modular_is_in_fundamental_domain**(**const** *acb_t z*, **const** *arf_t tol*, *slong prec*)
Returns nonzero if it is certainly true that $|z| \ge 1-\varepsilon$ and $|\operatorname{Re}(z)| \le 1/2+\varepsilon$ where ε is specified by *tol*. Returns zero if this is false or cannot be determined.

9.4.3 Addition sequences

void **acb_modular_fill_addseq**(*slong* *tab, *slong len*)
Builds a near-optimal addition sequence for a sequence of integers which is assumed to be reasonably dense.

As input, the caller should set each entry in *tab* to -1 if that index is to be part of the addition sequence, and to 0 otherwise. On output, entry i in *tab* will either be zero (if the number is not part of the sequence), or a value j such that both j and $i-j$ are also marked. The first two entries in *tab* are ignored (the number 1 is always assumed to be part of the sequence).

9.4.4 Jacobi theta functions

Unfortunately, there are many inconsistent notational variations for Jacobi theta functions in the literature. Unless otherwise noted, we use the functions

$$\theta_1(z,\tau) = -i \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i[(n+1/2)^2\tau + (2n+1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} (-1)^n q^{n(n+1)} \sin((2n+1)\pi z)$$

$$\theta_2(z,\tau) = \sum_{n=-\infty}^{\infty} \exp(\pi i[(n+1/2)^2\tau + (2n+1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos((2n+1)\pi z)$$

$$\theta_3(z,\tau) = \sum_{n=-\infty}^{\infty} \exp(\pi i[n^2\tau + 2nz]) = 1 + 2\sum_{n=1}^{\infty} q^{n^2} \cos(2n\pi z)$$

$$\theta_4(z,\tau) = \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i[n^2\tau + 2nz]) = 1 + 2\sum_{n=1}^{\infty} (-1)^n q^{n^2} \cos(2n\pi z)$$

where $q = \exp(\pi i \tau)$ and $q_{1/4} = \exp(\pi i \tau/4)$. Note that many authors write $q_{1/4}$ as $q^{1/4}$, but the principal fourth root $(q)^{1/4} = \exp(\frac{1}{4}\log q)$ differs from $q_{1/4}$ in general and some formulas are only correct if one reads "$q^{1/4} = \exp(\pi i \tau/4)$". To avoid confusion, we only write q^k when k is an integer.

void **acb_modular_theta_transform**(int *R, int *S, int *C, **const** *psl2z_t* g)

We wish to write a theta function with quasiperiod τ in terms of a theta function with quasiperiod $\tau' = g\tau$, given some $g = (a,b;c,d) \in \mathrm{PSL}(2,\mathbb{Z})$. For $i = 0,1,2,3$, this function computes integers R_i and S_i (R and S should be arrays of length 4) and $C \in \{0,1\}$ such that

$$\theta_{1+i}(z,\tau) = \exp(\pi i R_i/4) \cdot A \cdot B \cdot \theta_{1+S_i}(z',\tau')$$

where $z' = z$, $A = B = 1$ if $C = 0$, and

$$z' = \frac{-z}{c\tau+d}, \quad A = \sqrt{\frac{i}{c\tau+d}}, \quad B = \exp\left(-\pi i c \frac{z^2}{c\tau+d}\right)$$

if $C = 1$. Note that A is well-defined with the principal branch of the square root since $A^2 = i/(c\tau+d)$ lies in the right half-plane.

Firstly, if $c = 0$, we have $\theta_i(z,\tau) = \exp(-\pi i b/4)\theta_i(z,\tau+b)$ for $i = 1,2$, whereas θ_3 and θ_4 remain unchanged when b is even and swap places with each other when b is odd. In this case we set $C = 0$.

For an arbitrary g with $c > 0$, we set $C = 1$. The general transformations are given by Rademacher [Rad1973]. We need the function $\theta_{m,n}(z,\tau)$ defined for $m,n \in \mathbb{Z}$ by (beware of the typos in [Rad1973])

$$\theta_{0,0}(z,\tau) = \theta_3(z,\tau), \quad \theta_{0,1}(z,\tau) = \theta_4(z,\tau)$$

$$\theta_{1,0}(z,\tau) = \theta_2(z,\tau), \quad \theta_{1,1}(z,\tau) = i\theta_1(z,\tau)$$

$$\theta_{m+2,n}(z,\tau) = (-1)^n \theta_{m,n}(z,\tau)$$

$$\theta_{m,n+2}(z,\tau) = \theta_{m,n}(z,\tau).$$

Then we may write

$$\begin{aligned}
\theta_1(z,\tau) &= \varepsilon_1 AB\theta_1(z',\tau') \\
\theta_2(z,\tau) &= \varepsilon_2 AB\theta_{1-c,1+a}(z',\tau') \\
\theta_3(z,\tau) &= \varepsilon_3 AB\theta_{1+d-c,1-b+a}(z',\tau') \\
\theta_4(z,\tau) &= \varepsilon_4 AB\theta_{1+d,1-b}(z',\tau')
\end{aligned}$$

where ε_i is an 8th root of unity. Specifically, if we denote the 24th root of unity in the transformation formula of the Dedekind eta function by $\varepsilon(a,b,c,d) = \exp(\pi i R(a,b,c,d)/12)$ (see *acb_modular_epsilon_arg()*), then:

$$\begin{aligned}
\varepsilon_1(a,b,c,d) &= \exp(\pi i[R(-d,b,c,-a)+1]/4) \\
\varepsilon_2(a,b,c,d) &= \exp(\pi i[-R(a,b,c,d)+(5+(2-c)a)]/4) \\
\varepsilon_3(a,b,c,d) &= \exp(\pi i[-R(a,b,c,d)+(4+(c-d-2)(b-a))]/4) \\
\varepsilon_4(a,b,c,d) &= \exp(\pi i[-R(a,b,c,d)+(3-(2+d)b)]/4)
\end{aligned}$$

These formulas are easily derived from the formulas in [Rad1973] (Rademacher has the transformed/untransformed variables exchanged, and his "ε" differs from ours by a constant offset in the phase).

void **acb_modular_addseq_theta**(*slong* **exponents*, *slong* **aindex*, *slong* **bindex*, *slong num*)
Constructs an addition sequence for the first *num* squares and triangular numbers interleaved (excluding zero), i.e. 1, 2, 4, 6, 9, 12, 16, 20, 25, 30 etc.

void **acb_modular_theta_sum**(*acb_ptr theta1*, *acb_ptr theta2*, *acb_ptr theta3*, *acb_ptr theta4*, **const** *acb_t w*, int *w_is_unit*, **const** *acb_t q*, *slong len*, *slong prec*)
Simultaneously computes the first *len* coefficients of each of the formal power series

$$\begin{aligned}
\theta_1(z+x,\tau)/q_{1/4} &\in \mathbb{C}[[x]] \\
\theta_2(z+x,\tau)/q_{1/4} &\in \mathbb{C}[[x]] \\
\theta_3(z+x,\tau) &\in \mathbb{C}[[x]] \\
\theta_4(z+x,\tau) &\in \mathbb{C}[[x]]
\end{aligned}$$

given $w = \exp(\pi i z)$ and $q = \exp(\pi i \tau)$, by summing a finite truncation of the respective theta function series. In particular, with *len* equal to 1, computes the respective value of the theta function at the point z. We require *len* to be positive. If *w_is_unit* is nonzero, w is assumed to lie on the unit circle, i.e. z is assumed to be real.

Note that the factor $q_{1/4}$ is removed from θ_1 and θ_2. To get the true theta function values, the user has to multiply this factor back. This convention avoids unnecessary computations, since the user can compute $q_{1/4} = \exp(\pi i \tau/4)$ followed by $q = (q_{1/4})^4$, and in many cases when computing products or quotients of theta functions, the factor $q_{1/4}$ can be eliminated entirely.

This function is intended for $|q| \ll 1$. It can be called with any q, but will return useless intervals if convergence is not rapid. For general evaluation of theta functions, the user should only call this function after applying a suitable modular transformation.

We consider the sums together, alternatingly updating (θ_1, θ_2) or (θ_3, θ_4). For $k = 0, 1, 2, \ldots$, the powers of q are $\lfloor (k+2)^2/4 \rfloor = 1, 2, 4, 6, 9$ etc. and the powers of w are $\pm(k+2) = \pm 2, \pm 3, \pm 4, \ldots$ etc. The scheme is illustrated by the following table:

$$\begin{array}{rll}
 & \theta_1, \theta_2 & q^0 \quad (w^1 \pm w^{-1}) \\
k=0 & \theta_3, \theta_4 & q^1 \quad (w^2 \pm w^{-2}) \\
k=1 & \theta_1, \theta_2 & q^2 \quad (w^3 \pm w^{-3}) \\
k=2 & \theta_3, \theta_4 & q^4 \quad (w^4 \pm w^{-4}) \\
k=3 & \theta_1, \theta_2 & q^6 \quad (w^5 \pm w^{-5}) \\
k=4 & \theta_3, \theta_4 & q^9 \quad (w^6 \pm w^{-6}) \\
k=5 & \theta_1, \theta_2 & q^{12} \quad (w^7 \pm w^{-7})
\end{array}$$

For some integer $N \geq 1$, the summation is stopped just before term $k = N$. Let $Q = |q|$, $W = \max(|w|, |w^{-1}|)$, $E = \lfloor (N+2)^2/4 \rfloor$ and $F = \lfloor (N+1)/2 \rfloor + 1$. The error of the zeroth derivative can be bounded as

$$2Q^E W^{N+2} \left[1 + Q^F W + Q^{2F} W^2 + \ldots\right] = \frac{2Q^E W^{N+2}}{1 - Q^F W}$$

provided that the denominator is positive (otherwise we set the error bound to infinity). When *len* is greater than 1, consider the derivative of order r. The term of index k and order r picks up a

factor of magnitude $(k+2)^r$ from differentiation of w^{k+2} (it also picks up a factor π^r, but we omit this until we rescale the coefficients at the end of the computation). Thus we have the error bound

$$2Q^E W^{N+2}(N+2)^r \left[1 + Q^F W \frac{(N+3)^r}{(N+2)^r} + Q^{2F} W^2 \frac{(N+4)^r}{(N+2)^r} + \ldots\right]$$

which by the inequality $(1 + m/(N+2))^r \le \exp(mr/(N+2))$ can be bounded as

$$\frac{2Q^E W^{N+2}(N+2)^r}{1 - Q^F W \exp(r/(N+2))},$$

again valid when the denominator is positive.

To actually evaluate the series, we write the even cosine terms as $w^{2n} + w^{-2n}$, the odd cosine terms as $w(w^{2n} + w^{-2n-2})$, and the sine terms as $w(w^{2n} - w^{-2n-2})$. This way we only need even powers of w and w^{-1}. The implementation is not yet optimized for real z, in which case further work can be saved.

This function does not permit aliasing between input and output arguments.

void **acb_modular_theta_const_sum_basecase**(*acb_t theta2*, *acb_t theta3*, *acb_t theta4*, **const** *acb_t q*, *slong N*, *slong prec*)

void **acb_modular_theta_const_sum_rs**(*acb_t theta2*, *acb_t theta3*, *acb_t theta4*, **const** *acb_t q*, *slong N*, *slong prec*)

Computes the truncated theta constant sums $\theta_2 = \sum_{k(k+1)<N} q^{k(k+1)}$, $\theta_3 = \sum_{k^2<N} q^{k^2}$, $\theta_4 = \sum_{k^2<N} (-1)^k q^{k^2}$. The *basecase* version uses a short addition sequence. The *rs* version uses rectangular splitting. The algorithms are described in [EHJ2016].

void **acb_modular_theta_const_sum**(*acb_t theta2*, *acb_t theta3*, *acb_t theta4*, **const** *acb_t q*, *slong prec*)

Computes the respective theta constants by direct summation (without applying modular transformations). This function selects an appropriate N, calls either *acb_modular_theta_const_sum_basecase()* or *acb_modular_theta_const_sum_rs()* or depending on N, and adds a bound for the truncation error.

void **acb_modular_theta_notransform**(*acb_t theta1*, *acb_t theta2*, *acb_t theta3*, *acb_t theta4*, **const** *acb_t z*, **const** *acb_t tau*, *slong prec*)

Evaluates the Jacobi theta functions $\theta_i(z,\tau)$, $i = 1,2,3,4$ simultaneously. This function does not move τ to the fundamental domain. This is generally worse than *acb_modular_theta()*, but can be slightly better for moderate input.

void **acb_modular_theta**(*acb_t theta1*, *acb_t theta2*, *acb_t theta3*, *acb_t theta4*, **const** *acb_t z*, **const** *acb_t tau*, *slong prec*)

Evaluates the Jacobi theta functions $\theta_i(z,\tau)$, $i = 1,2,3,4$ simultaneously. This function moves τ to the fundamental domain and then also reduces z modulo τ before calling *acb_modular_theta_sum()*.

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 $\theta_i(z+x,\tau) \in \mathbb{C}[[x]]$ to each respective output variable. The *notransform* version does not move τ to the fundamental domain or reduce z during the computation.

void **_acb_modular_theta_series**(*acb_ptr theta1*, *acb_ptr theta2*, *acb_ptr theta3*, *acb_ptr theta4*, *acb_srcptr z*, *slong zlen*, **const** *acb_t tau*, *slong len*, *slong prec*)

void **acb_modular_theta_series**(*acb_poly_t theta1*, *acb_poly_t theta2*, *acb_poly_t theta3*, *acb_poly_t theta4*, **const** *acb_poly_t z*, **const** *acb_t tau*, *slong len*, *slong prec*)
Evaluates the respective Jacobi theta functions of the power series z, truncated to length *len*. Either of the output variables can be *NULL*.

9.4.5 Dedekind eta function

void **acb_modular_addseq_eta**(*slong *exponents*, *slong *aindex*, *slong *bindex*, *slong num*)
Constructs an addition sequence for the first *num* generalized pentagonal numbers (excluding zero), i.e. 1, 2, 5, 7, 12, 15, 22, 26, 35, 40 etc.

void **acb_modular_eta_sum**(*acb_t eta*, **const** *acb_t q*, *slong prec*)
Evaluates the Dedekind eta function without the leading 24th root, i.e.

$$\exp(-\pi i\tau/12)\eta(\tau) = \sum_{n=-\infty}^{\infty} (-1)^n q^{(3n^2-n)/2}$$

given $q = \exp(2\pi i\tau)$, by summing the defining series.

This function is intended for $|q| \ll 1$. It can be called with any q, but will return useless intervals if convergence is not rapid. For general evaluation of the eta function, the user should only call this function after applying a suitable modular transformation.

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].

int **acb_modular_epsilon_arg**(**const** *psl2z_t g*)
Given $g = (a, b; c, d)$, computes an integer R such that $\varepsilon(a, b, c, d) = \exp(\pi i R/12)$ is the 24th root of unity in the transformation formula for the Dedekind eta function,

$$\eta\left(\frac{a\tau+b}{c\tau+d}\right) = \varepsilon(a,b,c,d)\sqrt{c\tau+d}\eta(\tau).$$

void **acb_modular_eta**(*acb_t r*, **const** *acb_t tau*, *slong prec*)
Computes the Dedekind eta function $\eta(\tau)$ given τ in the upper half-plane. This function applies the functional equation to move τ to the fundamental domain before calling *acb_modular_eta_sum()*.

9.4.6 Modular forms

void **acb_modular_j**(*acb_t r*, **const** *acb_t tau*, *slong prec*)
Computes Klein's j-invariant $j(\tau)$ given τ in the upper half-plane. The function is normalized so that $j(i) = 1728$. We first move τ to the fundamental domain, which does not change the value of the function. Then we use the formula $j(\tau) = 32(\theta_2^8 + \theta_3^8 + \theta_4^8)^3/(\theta_2\theta_3\theta_4)^8$ where $\theta_i = \theta_i(0, \tau)$.

void **acb_modular_lambda**(*acb_t r*, **const** *acb_t tau*, *slong prec*)
Computes the lambda function $\lambda(\tau) = \theta_2^4(0,\tau)/\theta_3^4(0,\tau)$, which is invariant under modular transformations $(a, b; c, d)$ where a, d are odd and b, c are even.

void **acb_modular_delta**(*acb_t r*, **const** *acb_t tau*, *slong prec*)
Computes the modular discriminant $\Delta(\tau) = \eta(\tau)^{24}$, which transforms as

$$\Delta\left(\frac{a\tau+b}{c\tau+d}\right) = (c\tau+d)^{12}\Delta(\tau).$$

The modular discriminant is sometimes defined with an extra factor $(2\pi)^{12}$, which we omit in this implementation.

void **acb_modular_eisenstein**(*acb_ptr r*, **const** *acb_t tau*, *slong len*, *slong prec*)
Computes simultaneously the first *len* entries in the sequence of Eisenstein series $G_4(\tau), G_6(\tau), G_8(\tau), \ldots$, defined by

$$G_{2k}(\tau) = \sum_{m^2+n^2 \neq 0} \frac{1}{(m+n\tau)^{2k}}$$

and satisfying

$$G_{2k}\left(\frac{a\tau+b}{c\tau+d}\right) = (c\tau+d)^{2k} G_{2k}(\tau).$$

We first evaluate $G_4(\tau)$ and $G_6(\tau)$ on the fundamental domain using theta functions, and then compute the Eisenstein series of higher index using a recurrence relation.

9.4.7 Elliptic integrals and functions

See the *acb_elliptic.h* module for elliptic integrals and functions. The following wrappers are available for backwards compatibility.

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*)

9.4.8 Class polynomials

void **acb_modular_hilbert_class_poly**(*fmpz_poly_t res*, *slong D*)
Sets *res* to the Hilbert class polynomial of discriminant D, defined as

$$H_D(x) = \prod_{(a,b,c)} \left(x - j\left(\frac{-b+\sqrt{D}}{2a}\right)\right)$$

where (a, b, c) ranges over the primitive reduced positive definite binary quadratic forms of discriminant $b^2 - 4ac = D$.

The Hilbert class polynomial is only defined if $D < 0$ and D is congruent to 0 or 1 mod 4. If some other value of D is passed as input, *res* is set to the zero polynomial.

9.5 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 *acb_dirichlet.h – Dirichlet L-functions, Riemann zeta and related functions*.

9.5.1 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

$$\begin{array}{ccccc} (\mathbb{Z}/q\mathbb{Z})^\times \times (\mathbb{Z}/q\mathbb{Z})^\times & \to & \bigoplus_i \mathbb{Z}/\phi_i\mathbb{Z} \times \mathbb{Z}/\phi_i\mathbb{Z} & \to & \mathbb{C} \\ (m,n) & \mapsto & (a_i,b_i) & \mapsto & \chi_q(m,n) = \exp(2i\pi \sum \frac{a_i b_i}{\phi_i}) \end{array}$$

We call *number* a residue class m modulo q, and *log* the corresponding vector (a_i) of exponents of Conrey generators.

Going from a *log* to the corresponding *number* is a cheap operation we call exponential, while the converse requires computing discrete logarithms.

9.5.2 Multiplicative group modulo *q*

type dirichlet_group_struct

type 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 $h \mid q$. 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.

9.5.3 Character type

type dirichlet_char_struct

type 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* $[0, \ldots 0]$.

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* $[1, \ldots 1]$, or $[0, 1, \ldots 1]$ if $8 \mid q$.

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 $0 \ldots \varphi(q)$.

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.

9.5.4 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 $\chi_q(a,\cdot)$ can be obtained as a character mod 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* λ 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.

int **dirichlet_char_is_real**(**const** *dirichlet_group_t* *G*, **const** *dirichlet_char_t* *chi*)
Returns 1 if *chi* is a real character (iff it has order ≤ 2).

int **dirichlet_char_is_primitive**(**const** *dirichlet_group_t* *G*, **const** *dirichlet_char_t* *chi*)
Returns 1 if *chi* is primitive (iff its conductor is exactly q).

9.5.5 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*)
Compute the list of exponent values *v[k]* for $0 \leq k < nv$, as exponents modulo *G->expo*.

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 $0 \le k < nv$, as exponents modulo *order*, which is assumed to be a multiple of the order of *chi*.

9.5.6 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 χ_G and q_G, q_H are the moduli of G and H.

9.6 acb_dirichlet.h – Dirichlet L-functions, Riemann zeta and related functions

This module allows working with values of Dirichlet characters, Dirichlet L-functions, and related functions. 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 trivial character $\chi(k) = 1$ gives the Riemann zeta function. Working with Dirichlet characters is documented in *dirichlet.h – Dirichlet characters*.

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.

9.6.1 Roots of unity

type acb_dirichlet_roots_struct

type acb_dirichlet_roots_t

void **acb_dirichlet_roots_init**(*acb_dirichlet_roots_t* *roots*, *ulong* *n*, *slong* *num*, *slong* *prec*)
Initializes *roots* with precomputed data for fast evaluation of roots of unity $e^{2\pi ik/n}$ of a fixed order *n*. The precomputation is optimized for *num* evaluations.

For very small *num*, only the single root $e^{2\pi i/n}$ will be precomputed, which can then be raised to a power. For small *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 *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 *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 $e^{2\pi i k/n}$.

9.6.2 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 *res* to $k^{-(s+x)}$ as a power series in x truncated to length *len*. The flags *integer* and *critical_line* respectively specify optimizing for s being an integer or having real part $1/2$.

On input *log_prev* should contain the natural logarithm of the integer at *prev*. If *prev* is close to k, this can be used to speed up computations. If $\log(k)$ is computed internally by this function, then *log_prev* is overwritten by this value, and the integer at *prev* is overwritten by k, allowing *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 $(ij)^r$ 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 $2^{-(s+x)}$. 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.

9.6.3 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_jet**(*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 (43.3) in [Rad1973] 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_zeta_deriv_bound**(*mag_t der1*, *mag_t der2*, **const** *acb_t s*)
Sets *der1* to a bound for $|\zeta'(s)|$ and *der2* to a bound for $|\zeta''(s)|$. These bounds are mainly intended for use in the critical strip and will not be tight.

void **acb_dirichlet_eta**(*acb_t res*, **const** *acb_t s*, *slong prec*)
Sets *res* to the Dirichlet eta function $\eta(s) = \sum_{k=1}^{\infty} (-1)^{k+1}/k^s = (1 - 2^{1-s})\zeta(s)$, also known as the alternating zeta function. Note that the alternating character $\{1, -1\}$ is not itself a Dirichlet character.

void **acb_dirichlet_xi**(*acb_t res*, **const** *acb_t s*, *slong prec*)
Sets *res* to the Riemann xi function $\xi(s) = \frac{1}{2}s(s-1)\pi^{-s/2}\Gamma(\frac{1}{2}s)\zeta(s)$. The functional equation for xi is $\xi(1-s) = \xi(s)$.

9.6.4 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

$$\zeta(s) = \mathcal{R}(s) + X(s)\overline{\mathcal{R}}(1-s), \quad X(s) = \pi^{s-1/2}\frac{\Gamma((1-s)/2)}{\Gamma(s/2)}$$

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 $0 \le j < n$. Uses power series division. This method breaks down when $p = \pm 1/2$ (which is not problem if s is an exact floating-point number).

void **acb_dirichlet_zeta_rs_d_coeffs**(*arb_ptr d*, **const** *arb_t sigma*, *slong k*, *slong prec*)
Computes the coefficients $d_j^{(k)}$ for $0 \le j \le \lfloor 3k/2 \rfloor + 1$. 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 **acb_dirichlet_zeta_rs_bound**(*mag_t err*, **const** *acb_t s*, *slong K*)
Bounds the error term RS_K following Theorem 4.2 in Arias de Reyna.

void **acb_dirichlet_zeta_rs_r**(*acb_t res*, **const** *acb_t s*, *slong K*, *slong prec*)
Computes $\mathcal{R}(s)$ 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 **acb_dirichlet_zeta_rs**(*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 **acb_dirichlet_zeta_jet_rs**(*acb_t res*, **const** *acb_t s*, *slong len*, *slong prec*)
Computes the first *len* terms of the Taylor series of the Riemann zeta function at s using the Riemann siegel formula. This function currently only supports *len* = 1 or *len* = 2. A finite difference is used to compute the first derivative.

9.6.5 Hurwitz zeta function

void **acb_dirichlet_hurwitz**(*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.

9.6.6 Hurwitz zeta function precomputation

type acb_dirichlet_hurwitz_precomp_struct

type acb_dirichlet_hurwitz_precomp_t

void **acb_dirichlet_hurwitz_precomp_init**(*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 a, 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 *acb_dirichlet_hurwitz()* directly.

Otherwise, we require that A, K and N are all positive. For a finite error bound, we require $K + \mathrm{re}(s) > 1$. 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 *deflate* is set, the deflated Hurwitz zeta function is used, removing the pole at $s = 1$.

void **acb_dirichlet_hurwitz_precomp_init_num**(*acb_dirichlet_hurwitz_precomp_t pre*, **const** *acb_t s*, int *deflate*, double *num_eval*, *slong prec*)

Initializes *pre*, choosing the parameters A, K, and N automatically to minimize the cost of *num_eval* evaluations of the Hurwitz zeta function at argument s to precision *prec*.

void **acb_dirichlet_hurwitz_precomp_clear**(*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 $0 < a \le 1$. 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 $0 < p/q \le 1$.

9.6.7 Stieltjes constants

void **acb_dirichlet_stieltjes**(*acb_t res*, **const** *fmpz_t n*, **const** *acb_t a*, *slong prec*)

Given a nonnegative integer n, sets *res* to the generalized Stieltjes constant $\gamma_n(a)$ which is the coefficient in the Laurent series of the Hurwitz zeta function at the pole

$$\zeta(s, a) = \frac{1}{s-1} + \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \gamma_n(a)(s-1)^n.$$

With $a = 1$, this gives the ordinary Stieltjes constants for the Riemann zeta function.

This function uses an integral representation to permit fast computation for extremely large n [JB2018]. If n is moderate and the precision is high enough, it falls back to evaluating the Hurwitz zeta function of a power series and reading off the last coefficient.

Note that for computing a range of values $\gamma_0(a), \ldots, \gamma_n(a)$, it is generally more efficient to evaluate the Hurwitz zeta function series expansion once at $s = 1$ than to call this function repeatedly, unless n is extremely large (at least several hundred).

9.6.8 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.

9.6.9 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

$$G_q(a) = \sum_{x \bmod q} \chi_q(a, x) e^{\frac{2i\pi x}{q}}$$

- 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 $i^p\sqrt{q}$ where p is the parity of χ.
- 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.

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

$$J_q(a,b) = \sum_{x \bmod q} \chi_q(a,x)\chi_q(b,1-x)$$

- 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.

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 $x(t) = \exp(-\frac{\pi}{N}t^2)$ and define

$$\Theta_q(a,t) = \sum_{n \geq 0} \chi_q(a,n) x(t)^{n^2}.$$

ulong **acb_dirichlet_theta_length**(*ulong q*, **const** *arb_t t*, *slong prec*)

Compute the number of terms to be summed in the theta series of argument t so that the tail is less than $2^{-\text{prec}}$.

void **acb_dirichlet_qseries_powers_naive**(*acb_t res*, **const** *arb_t x*, int *p*, **const** *ulong* *a, **const** acb_dirichlet_powers_t *z*, *slong len*, *slong prec*)

void **acb_dirichlet_qseries_powers_smallorder**(*acb_t res*, **const** *arb_t x*, int *p*, **const** *ulong* **a*, **const** acb_dirichlet_powers_t *z*, *slong len*, *slong prec*)

Compute the series $\sum n^p z^{a_n} x^{n^2}$ for exponent list a, precomputed powers z and parity p (being 0 or 1).

The *naive* version sums the series as defined, while the *smallorder* variant evaluates the series on the quotient ring by a cyclotomic polynomial before evaluating at the root of unity, ignoring its argument z.

9.6.10 Discrete Fourier transforms

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

$$\hat{f}(\chi) = \sum_{x \bmod q} \overline{\chi(x)} f(x)$$

See the *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

$$g(a) = \sum_{b \bmod q} \overline{\chi_q(a,b)} f(b)$$

void **acb_dirichlet_dft_conrey**(*acb_ptr w*, *acb_srcptr v*, **const** *dirichlet_group_t G*, *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 *dirichlet_char_next()* or by *dirichlet_char_index()*).

For example, if $q = 15$, the Conrey elements are stored in following order

index	log = [e,f]	number = 7^e 11^f
0	[0, 0]	1
1	[0, 1]	7
2	[0, 2]	4
3	[0, 3]	13
4	[0, 4]	1
5	[1, 0]	11
6	[1, 1]	2
7	[1, 2]	14
8	[1, 3]	8
9	[1, 4]	11

void **acb_dirichlet_dft**(*acb_ptr w*, *acb_srcptr v*, **const** *dirichlet_group_t G*, *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.

9.6.11 Dirichlet L-functions

void **acb_dirichlet_root_number_theta**(*acb_t res*, **const** *dirichlet_group_t G*, **const** *dirichlet_char_t chi*, *slong prec*)

void **acb_dirichlet_root_number**(*acb_t res*, **const** *dirichlet_group_t G*, **const** *dirichlet_char_t chi*, *slong prec*)

Sets *res* to the root number $\epsilon(\chi)$ for a primitive character *chi*, which appears in the functional equation (where p is the parity of χ):

$$\left(\frac{q}{\pi}\right)^{\frac{s+p}{2}} \Gamma\left(\frac{s+p}{2}\right) L(s,\chi) = \epsilon(\chi) \left(\frac{q}{\pi}\right)^{\frac{1-s+p}{2}} \Gamma\left(\frac{1-s+p}{2}\right) L(1-s,\overline{\chi})$$

- The *theta* variant uses the evaluation at $t = 1$ of the Theta series.
- The default version computes it via the gauss sum.

void **acb_dirichlet_l_hurwitz**(*acb_t res*, **const** *acb_t s*, **const** *acb_dirichlet_hurwitz_precomp_t precomp*, **const** *dirichlet_group_t G*, **const** *dirichlet_char_t chi*, *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 χ is non-principal, the deflated Hurwitz zeta function is used to avoid poles.

If *precomp* is *NULL*, each Hurwitz zeta function value is computed directly. If a pre-initialized *precomp* object is provided, this will be used instead to evaluate the Hurwitz zeta function.

void **acb_dirichlet_l_euler_product**(*acb_t res*, **const** *acb_t s*, **const** *dirichlet_group_t G*, **const** *dirichlet_char_t chi*, *slong prec*)

void **_acb_dirichlet_euler_product_real_ui**(*arb_t res*, *ulong s*, **const** signed char **chi*, int *mod*, int *reciprocal*, *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 $\operatorname{re}(s) \geq 2$.

An error bound is computed via *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 χ is a real Dirichlet character given by the explicit list *chi* of character values at 0, 1, ..., *mod* - 1. If *reciprocal* is set, it computes $1/L(s,\chi)$ (this is faster if the reciprocal can be used directly).

void **acb_dirichlet_l**(*acb_t res*, **const** *acb_t s*, **const** *dirichlet_group_t G*, **const** *dirichlet_char_t chi*, *slong prec*)

Computes $L(s,\chi)$ using a default choice of algorithm.

void **acb_dirichlet_l_vec_hurwitz**(*acb_ptr res*, **const** *acb_t s*, **const** *acb_dirichlet_hurwitz_precomp_t precomp*, **const** *dirichlet_group_t G*, *slong prec*)

Compute all values $L(s,\chi)$ for χ mod q, using the Hurwitz zeta function and a discrete Fourier transform. The output *res* is assumed to have length *G->phi_q* and values are stored by lexicographically ordered Conrey logs. See *acb_dirichlet_dft_conrey()*.

If *precomp* is *NULL*, each Hurwitz zeta function value is computed directly. If a pre-initialized *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

$$L(s,\chi) - \frac{\sum_{k=1}^{q} \chi(k)}{(s-1)q}$$

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 $L(s,\chi)$ where s is a given power series, truncating the result to length *len*. See *acb_dirichlet_l_jet()* for the meaning of the *deflate* flag.

9.6.12 Hardy Z-functions

For convenience, setting both G and *chi* to *NULL* in the following methods selects the Riemann zeta function.

Currently, these methods require *chi* to be a primitive character.

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

$$\theta(t) = -\frac{t}{2}\log(\pi/q) - \frac{i\log(\epsilon)}{2} + \frac{\log\Gamma((s+\delta)/2) - \log\Gamma((1-s+\delta)/2)}{2i}$$

where $s = 1/2 + it$, δ is the parity of *chi*, and ϵ 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 $Z(t) = e^{i\theta(t)}L(1/2+it)$, 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*.

9.6.13 Gram points

void **acb_dirichlet_gram_point**(*arb_t res*, **const** *fmpz_t n*, **const** *dirichlet_group_t G*, **const** *dirichlet_char_t chi*, *slong prec*)
Sets *res* to the n-th Gram point g_n, defined as the unique solution in $[7, \infty)$ of $\theta(g_n) = \pi n$. Currently only the Gram points corresponding to the Riemann zeta function are supported and G and *chi* must both be set to *NULL*. Requires $n \geq -1$.

9.6.14 Riemann zeta function zeros

The following functions for counting and isolating zeros of the Riemann zeta function use the ideas from the implementation of Turing's method in mpmath [Joh2018b] by Juan Arias de Reyna, described in [Ari2012].

ulong **acb_dirichlet_turing_method_bound**(**const** *fmpz_t p*)
Computes an upper bound B for the minimum number of consecutive good Gram blocks sufficient to count nontrivial zeros of the Riemann zeta function using Turing's method [Tur1953] as updated by [Leh1970], [Bre1979], and [Tru2011].

Let $N(T)$ denote the number of zeros (counted according to their multiplicities) of $\zeta(s)$ in the region $0 < \mathrm{Im}(s) \leq T$. If at least B consecutive Gram blocks with union $[g_n, g_p)$ satisfy Rosser's rule, then $N(g_n) \leq n + 1$ and $N(g_p) \geq p + 1$.

int **_acb_dirichlet_definite_hardy_z**(*arb_t res*, **const** *arf_t t*, *slong* *pprec)
Sets *res* to the Hardy Z-function $Z(t)$. The initial precision (* *pprec*) is increased as necessary to determine the sign of $Z(t)$. The sign is returned.

void **_acb_dirichlet_isolate_gram_hardy_z_zero**(*arf_t a*, *arf_t b*, **const** *fmpz_t n*)
Uses Gram's law to compute an interval (a, b) that contains the n-th zero of the Hardy Z-function and no other zero. Requires $1 \leq n \leq 126$.

void **_acb_dirichlet_isolate_rosser_hardy_z_zero**(*arf_t a*, *arf_t b*, **const** *fmpz_t n*)
Uses Rosser's rule to compute an interval (a, b) that contains the n-th zero of the Hardy Z-function and no other zero. Requires $1 \leq n \leq 13999526$.

void **_acb_dirichlet_isolate_turing_hardy_z_zero**(*arf_t a*, *arf_t b*, **const** *fmpz_t n*)
Computes an interval (a, b) that contains the n-th zero of the Hardy Z-function and no other zero, following Turing's method. Requires $n \geq 2$.

void **acb_dirichlet_isolate_hardy_z_zero**(*arf_t a*, *arf_t b*, **const** *fmpz_t n*)
Computes an interval (a, b) that contains the n-th zero of the Hardy Z-function and contains no other zero, using the most appropriate underscore version of this function. Requires $n \geq 1$.

void **_acb_dirichlet_refine_hardy_z_zero**(*arb_t res*, **const** *arf_t a*, **const** *arf_t b*, *slong prec*)
Sets *res* to the unique zero of the Hardy Z-function in the interval (a, b).

void **acb_dirichlet_hardy_z_zero**(*arb_t res*, **const** *fmpz_t n*, *slong prec*)
Sets *res* to the n-th zero of the Hardy Z-function, requiring $n \geq 1$.

void **acb_dirichlet_hardy_z_zeros**(*arb_ptr res*, **const** *fmpz_t n*, *slong len*, *slong prec*)
Sets the entries of *res* to *len* consecutive zeros of the Hardy Z-function, beginning with the n-th zero. Requires positive n.

void **acb_dirichlet_zeta_zero**(*acb_t res*, **const** *fmpz_t n*, *slong prec*)
Sets *res* to the n-th nontrivial zero of $\zeta(s)$, requiring $n \geq 1$.

void **acb_dirichlet_zeta_zeros**(*acb_ptr res*, **const** *fmpz_t n*, *slong len*, *slong prec*)
Sets the entries of *res* to *len* consecutive nontrivial zeros of $\zeta(s)$ beginning with the n-th zero. Requires positive n.

void **_acb_dirichlet_exact_zeta_nzeros**(*fmpz_t res*, **const** *arf_t t*)

void **acb_dirichlet_zeta_nzeros**(*arb_t res*, **const** *arb_t t*, *slong prec*)
Compute the number of zeros (counted according to their multiplicities) of $\zeta(s)$ in the region $0 < \operatorname{Im}(s) \le t$.

void **acb_dirichlet_backlund_s**(*arb_t res*, **const** *arb_t t*, *slong prec*)
Compute $S(t) = \frac{1}{\pi} \arg \zeta(\frac{1}{2} + it)$ where the argument is defined by continuous variation of s in $\zeta(s)$ starting at $s = 2$, then vertically to $s = 2 + it$, then horizontally to $s = \frac{1}{2} + it$. In particular arg in this context is not the principal value of the argument, and it cannot be computed directly by *acb_arg()*. In practice $S(t)$ is computed as $S(t) = N(t) - \frac{1}{\pi}\theta(t) - 1$ where $N(t)$ is *acb_dirichlet_zeta_nzeros()* and $\theta(t)$ is *acb_dirichlet_hardy_theta()*.

void **acb_dirichlet_backlund_s_bound**(*mag_t res*, **const** *arb_t t*)
Compute an upper bound for $|S(t)|$ quickly. Theorem 1 and the bounds in (1.2) in [Tru2014] are used.

void **acb_dirichlet_zeta_nzeros_gram**(*fmpz_t res*, **const** *fmpz_t n*)
Compute $N(g_n)$. That is, compute the number of zeros (counted according to their multiplicities) of $\zeta(s)$ in the region $0 < \operatorname{Im}(s) \le g_n$ where g_n is the n-th Gram point. Requires $n \ge -1$.

slong **acb_dirichlet_backlund_s_gram**(**const** *fmpz_t n*)
Compute $S(g_n)$ where g_n is the n-th Gram point. Requires $n \ge -1$.

9.6.15 Riemann zeta function zeros (Platt's method)

The following functions related to the Riemann zeta function use the ideas and formulas described by David J. Platt in [Pla2017].

void **acb_dirichlet_platt_scaled_lambda**(*arb_t res*, **const** *arb_t t*, *slong prec*)
Compute $\Lambda(t)e^{\pi t/4}$ where

$$\Lambda(t) = \pi^{-\frac{it}{2}} \Gamma\left(\frac{\frac{1}{2} + it}{2}\right) \zeta\left(\frac{1}{2} + it\right)$$

is defined in the beginning of section 3 of [Pla2017]. As explained in [Pla2011] this function has the same zeros as $\zeta(1/2 + it)$ and is real-valued by the functional equation, and the exponential factor is designed to counteract the decay of the gamma factor as t increases.

void **acb_dirichlet_platt_scaled_lambda_vec**(*arb_ptr res*, **const** *fmpz_t T*, *slong A*, *slong B*, *slong prec*)

void **acb_dirichlet_platt_multieval**(*arb_ptr res*, **const** *fmpz_t T*, *slong A*, *slong B*, **const** *arb_t h*, *slong J*, *slong K*, *slong sigma*, *slong prec*)
Compute *acb_dirichlet_platt_scaled_lambda()* at $N = AB$ points on a grid, following the notation of [Pla2017]. The first point on the grid is $T - B/2$ and the distance between grid points is $1/A$. The product $N = AB$ must be an even integer. The multieval variant evaluates the function at all points on the grid simultaneously using forward and inverse discrete Fourier transforms, and it requires the four additional tuning parameters *h*, *J*, *K*, and *sigma*.

void **acb_dirichlet_platt_ws_interpolation**(*arb_t res*, *arf_t deriv*, **const** *arb_t t0*, *arb_srcptr p*, **const** *fmpz_t T*, *slong A*, *slong B*, *slong Ns_max*, **const** *arb_t H*, *slong sigma*, *slong prec*)
Compute *acb_dirichlet_platt_scaled_lambda()* at *t0* by Gaussian-windowed Whittaker-Shannon interpolation of points evaluated by *acb_dirichlet_platt_scaled_lambda_vec()*. The derivative is also approximated if the output parameter *deriv* is not *NULL*. *Ns_max* defines the maximum number of supporting points to be used in the interpolation on either side of *t0*. *H* is the standard deviation of the Gaussian window centered on *t0* to be applied before the interpolation. *sigma* is an odd positive integer tuning parameter $\sigma \in 2\mathbb{Z}_{>0} + 1$ used in computing error bounds.

slong **_acb_dirichlet_platt_local_hardy_z_zeros**(*arb_ptr res*, **const** *fmpz_t n*, *slong len*, **const** *fmpz_t T*, *slong A*, *slong B*, **const** *arb_t h*, *slong J*, *slong K*, *slong sigma_grid*, *slong Ns_max*, **const** *arb_t H*, *slong sigma_interp*, *slong prec*)

slong **acb_dirichlet_platt_local_hardy_z_zeros**(*arb_ptr res*, **const** *fmpz_t n*, *slong len*, *slong prec*)

Sets the entries of *res* to at most *len* consecutive zeros of the Hardy Z-function, beginning with the n-th zero. Requires positive n. The number of isolated zeros is returned. Internally this function uses a single call to Platt's grid evaluation of the scaled Lambda function. The final several parameters of the underscored variant have the same meanings as in the functions *acb_dirichlet_platt_multieval()* and *acb_dirichlet_platt_ws_interpolation()*. The non-underscored variant currently requires $10^4 \le n \le 3 \times 10^22$.

9.7 bernoulli.h – support for Bernoulli numbers

This module provides helper functions for exact or approximate calculation of the Bernoulli numbers, which are defined by the exponential generating function

$$\frac{x}{e^x - 1} = \sum_{n=0}^{\infty} B_n \frac{x^n}{n!}.$$

Efficient algorithms are implemented for both multi-evaluation and calculation of isolated Bernoulli numbers. A global (or thread-local) cache is also provided, to support fast repeated evaluation of various special functions that depend on the Bernoulli numbers (including the gamma function and the Riemann zeta function).

9.7.1 Generation of Bernoulli numbers

type 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*.

9.7.2 Caching

slong **bernoulli_cache_num**

fmpq ***bernoulli_cache**
Cache of Bernoulli numbers. Uses thread-local storage if enabled in FLINT.

void **bernoulli_cache_compute**(*slong* *n*)
Makes sure that the Bernoulli numbers up to at least B_{n-1} are cached. Calling `flint_cleanup()` frees the cache.

9.7.3 Bounding

slong **bernoulli_bound_2exp_si**(*ulong* *n*)
Returns an integer b such that $|B_n| \le 2^b$. Uses a lookup table for small n, and for larger n uses the inequality $|B_n| < 4n!/(2\pi)^n < 4(n+1)^{n+1}e^{-n}/(2\pi)^n$. Uses integer arithmetic throughout, with the bound for the logarithm being looked up from a table. If $|B_n| = 0$, returns *LONG_MIN*. Otherwise, the returned exponent b is never more than one percent larger than the true magnitude.

This function is intended for use when n small enough that one might comfortably compute B_n exactly. It aborts if n is so large that internal overflow occurs.

void **_bernoulli_fmpq_ui_zeta**(*fmpz_t* *num*, *fmpz_t* *den*, *ulong* *n*)
Sets *num* and *den* to the reduced numerator and denominator of the Bernoulli number B_n.

This function computes the denominator d using von Staudt-Clausen theorem, numerically approximates B_n using *arb_bernoulli_ui_zeta()*, and then rounds dB_n to the correct numerator. If the working precision is insufficient to determine the numerator, the function prints a warning message and retries with increased precision (this should not be expected to happen).

void **_bernoulli_fmpq_ui**(*fmpz_t* *num*, *fmpz_t* *den*, *ulong* *n*)

void **bernoulli_fmpq_ui**(*fmpq_t* *b*, *ulong* *n*)
Computes the Bernoulli number B_n as an exact fraction, for an isolated integer n. This function reads B_n from the global cache if the number is already cached, but does not automatically extend the cache by itself.

9.8 hypgeom.h – support for hypergeometric series

This module provides functions for high-precision evaluation of series of the form

$$\sum_{k=0}^{n-1} \frac{A(k)}{B(k)} \prod_{j=1}^{k} \frac{P(k)}{Q(k)} z^k$$

where A, B, P, Q are polynomials. The present version only supports $A, B, P, Q \in \mathbb{Z}[k]$ (represented using the FLINT *fmpz_poly_t* type). This module also provides functions for high-precision evaluation of infinite series ($n \to \infty$), with automatic, rigorous error bounding.

Note that we can standardize to $A = B = 1$ by setting $\tilde{P}(k) = P(k)A(k)B(k-1), \tilde{Q}(k) = Q(k)A(k-1)B(k)$. However, separating out A and B is convenient and improves efficiency during evaluation.

9.8.1 Strategy for error bounding

We wish to evaluate $S(z) = \sum_{k=0}^{\infty} T(k) z^k$ where $T(k)$ satisfies $T(0) = 1$ and

$$T(k) = R(k)T(k-1) = \left(\frac{P(k)}{Q(k)}\right) T(k-1)$$

for given polynomials

$$P(k) = a_p k^p + a_{p-1} k^{p-1} + \ldots a_0$$
$$Q(k) = b_q k^q + b_{q-1} k^{q-1} + \ldots b_0.$$

For convergence, we require $p < q$, or $p = q$ with $|z||a_p| < |b_q|$. We also assume that $P(k)$ and $Q(k)$ have no roots among the positive integers (if there are positive integer roots, the sum is either finite or undefined). With these conditions satisfied, our goal is to find a parameter $n \geq 0$ such that

$$\left| \sum_{k=n}^{\infty} T(k) z^k \right| \leq 2^{-d}.$$

We can rewrite the hypergeometric term ratio as

$$zR(k) = z\frac{P(k)}{Q(k)} = z\left(\frac{a_p}{b_q}\right)\frac{1}{k^{q-p}} F(k)$$

where

$$F(k) = \frac{1 + \tilde{a}_1/k + \tilde{a}_2/k^2 + \ldots + \tilde{a}_q/k^p}{1 + \tilde{b}_1/k + \tilde{b}_2/k^2 + \ldots + \tilde{b}_q/k^q} = 1 + O(1/k)$$

and where $\tilde{a}_i = a_{p-i}/a_p$, $\tilde{b}_i = b_{q-i}/b_q$. Next, we define

$$C = \max_{1 \leq i \leq p} |\tilde{a}_i|^{(1/i)}, \quad D = \max_{1 \leq i \leq q} |\tilde{b}_i|^{(1/i)}.$$

Now, if $k > C$, the magnitude of the numerator of $F(k)$ is bounded from above by

$$1 + \sum_{i=1}^{p} \left(\frac{C}{k}\right)^i \leq 1 + \frac{C}{k-C}$$

and if $k > 2D$, the magnitude of the denominator of $F(k)$ is bounded from below by

$$1 - \sum_{i=1}^{q} \left(\frac{D}{k}\right)^i \geq 1 + \frac{D}{D-k}.$$

Putting the inequalities together gives the following bound, valid for $k > K = \max(C, 2D)$:

$$|F(k)| \leq \frac{k(k-D)}{(k-C)(k-2D)} = \left(1 + \frac{C}{k-C}\right)\left(1 + \frac{D}{k-2D}\right).$$

Let $r = q - p$ and $\tilde{z} = |z a_p / b_q|$. 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).

9.8.2 Types, macros and constants

type hypgeom_struct

type hypgeom_t
Stores polynomials A, B, P, Q and precomputed bounds, representing a fixed hypergeometric series.

9.8.3 Memory management

void **hypgeom_init**(*hypgeom_t hyp*)

void **hypgeom_clear**(*hypgeom_t hyp*)

9.8.4 Error bounding

slong **hypgeom_estimate_terms**(**const** *mag_t z*, int *r*, *slong d*)
Computes an approximation of the largest n such that $|z|^n/(n!)^r = 2^{-d}$, giving a first-order estimate of the number of terms needed to approximate the sum of a hypergeometric series of weight $r \geq 0$ and argument z to an absolute precision of $d \geq 0$ bits. If $r = 0$, the direct solution of the equation is given by $n = (\log(1-z) - d\log 2)/\log z$. If $r > 0$, using $\log n! \approx n\log n - n$ gives an equation that can be solved in terms of the Lambert W-function as $n = (d\log 2)/(r\,W(t))$ where $t = (d\log 2)/(erz^{1/r})$.

The evaluation is done using double precision arithmetic. The function aborts if the computed value of n is greater than or equal to LONG_MAX / 2.

slong **hypgeom_bound**(*mag_t error*, int *r*, *slong C*, *slong D*, *slong K*, **const** *mag_t TK*, **const** *mag_t z*, *slong prec*)
Computes a truncation parameter sufficient to achieve *prec* bits of absolute accuracy, according to the strategy described above. The input consists of r, C, D, K, precomputed bound for $T(K)$, and $\tilde{z} = z(a_p/b_q)$, such that for $k > K$, the hypergeometric term ratio is bounded by

$$\frac{\tilde{z}}{k^r}\frac{k(k-D)}{(k-C)(k-2D)}.$$

Given this information, we compute a ε and an integer n such that $|\sum_{k=n}^{\infty} T(k)| \leq \varepsilon \leq 2^{-\mathrm{prec}}$. The output variable *error* is set to the value of ε, and n is returned.

void **hypgeom_precompute**(*hypgeom_t hyp*)
Precomputes the bounds data C, D, K and an upper bound for $T(K)$.

9.8.5 Summation

void **arb_hypgeom_sum**(*arb_t P*, *arb_t Q*, **const** *hypgeom_t hyp*, **const** *slong n*, *slong prec*)
Computes P, Q such that $P/Q = \sum_{k=0}^{n-1} T(k)$ where $T(k)$ is defined by *hyp*, using binary splitting and a working precision of *prec* bits.

void **arb_hypgeom_infsum**(*arb_t P*, *arb_t Q*, *hypgeom_t hyp*, *slong tol*, *slong prec*)
Computes P, Q such that $P/Q = \sum_{k=0}^{\infty} T(k)$ where $T(k)$ is defined by *hyp*, using binary splitting and working precision of *prec* bits. The number of terms is chosen automatically to bound the truncation error by at most $2^{-\mathrm{tol}}$. The bound for the truncation error is included in the output as part of P.

9.9 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.

void **partitions_hrr_sum_arb**(*arb_t* *x*, **const** *fmpz_t* *n*, *slong* *N0*, *slong* *N*, int *use_doubles*)
Evaluates the partial sum $\sum_{k=N_0}^{N} t(n,k)$ of the Hardy-Ramanujan-Rademacher series.

If *use_doubles* is nonzero, doubles and the system's standard library math functions are used to evaluate the smallest terms. This significantly speeds up evaluation for small n (e.g. $n < 10^6$), and gives a small speed improvement for larger n, but the result is not guaranteed to be correct. In practice, the error is estimated very conservatively, and unless the system's standard library is broken, use of doubles can be considered safe. Setting *use_doubles* to zero gives a fully guaranteed bound.

void **partitions_fmpz_fmpz**(*fmpz_t* *p*, **const** *fmpz_t* *n*, int *use_doubles*)
Computes the partition function $p(n)$ using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing $p(n)$ and verifies that the ball contains a unique integer.

If n is sufficiently large and a number of threads greater than 1 has been selected with `flint_set_num_threads()`, the computation time will be reduced by using two threads.

See *partitions_hrr_sum_arb()* for an explanation of the *use_doubles* option.

void **partitions_fmpz_ui**(*fmpz_t* *p*, *ulong* *n*)
Computes the partition function $p(n)$ using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing $p(n)$ and verifies that the ball contains a unique integer.

void **partitions_fmpz_ui_using_doubles**(*fmpz_t* *p*, *ulong* *n*)
Computes the partition function $p(n)$, enabling the use of doubles internally. This significantly speeds up evaluation for small n (e.g. $n < 10^6$), but the error bounds are not certified (see remarks for *partitions_hrr_sum_arb()*).

void **partitions_leading_fmpz**(*arb_t* *res*, **const** *fmpz_t* *n*, *slong* *prec*)
Sets *res* to the leading term in the Hardy-Ramanujan series for $p(n)$ (without Rademacher's correction of this term, which is vanishingly small when n is large), that is, $\sqrt{12}(1-1/t)e^t/(24n-1)$ where $t = \pi\sqrt{24n-1}/6$.

CHAPTER

TEN

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.

10.1 arb_calc.h – calculus with real-valued functions

This module provides functions for operations of calculus over the real numbers (intended to include root-finding, optimization, integration, and so on). It is planned that the module will include two types of algorithms:

- Interval algorithms that give provably correct results. An example would be numerical integration on an interval by dividing the interval into small balls and evaluating the function on each ball, giving rigorous upper and lower bounds.
- Conventional numerical algorithms that use heuristics to estimate the accuracy of a result, without guaranteeing that it is correct. An example would be numerical integration based on pointwise evaluation, where the error is estimated by comparing the results with two different sets of evaluation points. Ball arithmetic then still tracks the accuracy of the function evaluations.

Any algorithms of the second kind will be clearly marked as such.

10.1.1 Types, macros and constants

type 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

10.1.2 Debugging

int **arb_calc_verbose**
If set, enables printing information about the calculation to standard output.

10.1.3 Subdivision-based root finding

type arf_interval_struct

type 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.

type arf_interval_ptr
Alias for `arf_interval_struct *`, used for vectors of intervals.

type arf_interval_srcptr
Alias for `const arf_interval_struct *`, used for vectors of intervals.

void **arf_interval_init**(*arf_interval_t v*)

void **arf_interval_clear**(*arf_interval_t v*)

arf_interval_ptr **_arf_interval_vec_init**(*slong n*)

void **_arf_interval_vec_clear**(*arf_interval_ptr v*, *slong n*)

void **arf_interval_set**(*arf_interval_t v*, **const** *arf_interval_t u*)

void **arf_interval_swap**(*arf_interval_t v*, *arf_interval_t u*)

void **arf_interval_get_arb**(*arb_t x*, **const** *arf_interval_t v*, *slong prec*)

void **arf_interval_printd**(**const** *arf_interval_t v*, *slong n*)
Helper functions for endpoint-based intervals.

void **arf_interval_fprintd**(FILE **file*, **const** *arf_interval_t v*, *slong n*)
Helper functions for endpoint-based intervals.

slong **arb_calc_isolate_roots**(*arf_interval_ptr *found*, int **flags*, *arb_calc_func_t func*, void *param*, **const** *arf_interval_t interval*, *slong maxdepth*, *slong maxeval*, *slong maxfound*, *slong prec*)
Rigorously isolates single roots of a real analytic function on the interior of an interval.

This routine writes an array of n interesting subintervals of *interval* to *found* and corresponding flags to *flags*, returning the integer n. The output has the following properties:

- The function has no roots on *interval* outside of the output subintervals.
- Subintervals are sorted in increasing order (with no overlap except possibly starting and ending with the same point).
- Subintervals with a flag of 1 contain exactly one (single) root.
- Subintervals with any other flag may or may not contain roots.

If no flags other than 1 occur, all roots of the function on *interval* have been isolated. If there are output subintervals on which the existence or nonexistence of roots could not be determined, the user may attempt further searches on those subintervals (possibly with increased precision and/or increased bounds for the breaking criteria). Note that roots of multiplicity higher than one and roots located exactly at endpoints cannot be isolated by the algorithm.

The following breaking criteria are implemented:

- At most *maxdepth* recursive subdivisions are attempted. The smallest details that can be distinguished are therefore about $2^{-\text{maxdepth}}$ times the width of *interval*. A typical, reasonable value might be between 20 and 50.

- If the total number of tested subintervals exceeds *maxeval*, the algorithm is terminated and any untested subintervals are added to the output. The total number of calls to *func* is thereby restricted to a small multiple of *maxeval* (the actual count can be slightly higher depending on implementation details). A typical, reasonable value might be between 100 and 100000.
- The algorithm terminates if *maxfound* roots have been isolated. In particular, setting *maxfound* to 1 can be used to locate just one root of the function even if there are numerous roots. To try to find all roots, *LONG_MAX* may be passed.

The argument *prec* denotes the precision used to evaluate the function. It is possibly also used for some other arithmetic operations performed internally by the algorithm. Note that it probably does not make sense for *maxdepth* to exceed *prec*.

Warning: it is assumed that subdivision points of *interval* can be represented exactly as floating-point numbers in memory. Do not pass $1 \pm 2^{-10^{100}}$ as input.

int **arb_calc_refine_root_bisect**(*arf_interval_t r*, *arb_calc_func_t func*, void **param*, **const** *arf_interval_t start*, *slong iter*, *slong prec*)

Given an interval *start* known to contain a single root of *func*, refines it using *iter* bisection steps. The algorithm can return a failure code if the sign of the function at an evaluation point is ambiguous. The output r is set to a valid isolating interval (possibly just *start*) even if the algorithm fails.

10.1.4 Newton-based root finding

void **arb_calc_newton_conv_factor**(*arf_t conv_factor*, *arb_calc_func_t func*, void **param*, **const** *arb_t conv_region*, *slong prec*)

Given an interval I specified by *conv_region*, evaluates a bound for $C = \sup_{t,u \in I} \frac{1}{2}|f''(t)|/|f'(u)|$, where f is the function specified by *func* and *param*. The bound is obtained by evaluating $f'(I)$ and $f''(I)$ directly. If f is ill-conditioned, I may need to be extremely precise in order to get an effective, finite bound for C.

int **arb_calc_newton_step**(*arb_t xnew*, *arb_calc_func_t func*, void **param*, **const** *arb_t x*, **const** *arb_t conv_region*, **const** *arf_t conv_factor*, *slong prec*)

Performs a single step with an interval version of Newton's method. The input consists of the function f specified by *func* and *param*, a ball $x = [m - r, m + r]$ known to contain a single root of f, a ball I (*conv_region*) containing x with an associated bound (*conv_factor*) for $C = \sup_{t,u \in I} \frac{1}{2}|f''(t)|/|f'(u)|$, and a working precision *prec*.

The Newton update consists of setting $x' = [m' - r', m' + r']$ where $m' = m - f(m)/f'(m)$ and $r' = Cr^2$. The expression $m - f(m)/f'(m)$ is evaluated using ball arithmetic at a working precision of *prec* bits, and the rounding error during this evaluation is accounted for in the output. We now check that $x' \in I$ and $r' < r$. If both conditions are satisfied, we set *xnew* to x' and return *ARB_CALC_SUCCESS*. If either condition fails, we set *xnew* to x and return *ARB_CALC_NO_CONVERGENCE*, indicating that no progress is made.

int **arb_calc_refine_root_newton**(*arb_t r*, *arb_calc_func_t func*, void **param*, **const** *arb_t start*, **const** *arb_t conv_region*, **const** *arf_t conv_factor*, *slong eval_extra_prec*, *slong prec*)

Refines a precise estimate of a single root of a function to high precision by performing several Newton steps, using nearly optimally chosen doubling precision steps.

The inputs are defined as for *arb_calc_newton_step*, except for the precision parameters: *prec* is the target accuracy and *eval_extra_prec* is the estimated number of guard bits that need to be added to evaluate the function accurately close to the root (for example, if the function is a polynomial with large coefficients of alternating signs and Horner's rule is used to evaluate it, the extra precision should typically be approximately the bit size of the coefficients).

This function returns *ARB_CALC_SUCCESS* if all attempted Newton steps are successful (note that this does not guarantee that the computed root is accurate to *prec* bits, which has to be verified by the user), only that it is more accurate than the starting ball.

On failure, *ARB_CALC_IMPRECISE_INPUT* or *ARB_CALC_NO_CONVERGENCE* may be returned. In this case, r is set to a ball for the root which is valid but likely does have full accuracy (it can possibly just be equal to the starting ball).

10.2 acb_calc.h – calculus with complex-valued functions

This module provides functions for operations of calculus over the complex numbers (intended to include root-finding, integration, and so on). The numerical integration code is described in [Joh2018a].

10.2.1 Types, macros and constants

type acb_calc_func_t

Typedef for a pointer to a function with signature:

```
int func(acb_ptr out, const acb_t inp, void * param, slong order, slong prec)
```

implementing a univariate complex function $f(z)$. 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 builtin 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 an 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:

```
/* 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 *acb_real_abs()*, *acb_real_sgn()*, *acb_real_heaviside()*, *acb_real_floor()*, *acb_real_ceil()*, *acb_real_max()*, *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):

```
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, *acb_real_sqrtpos()* may be slightly better if it is known that z will be nonnegative on the path.)

See the demo program `examples/integrals.c` for more examples.

10.2.2 Integration

int **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 (*func*, *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 *func* will only be called with $order = 0$ or $order = 1$; that is, derivatives are not required.

- The integrand will be called with $order = 0$ 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 $order = 1$ 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, *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:

- *rel_goal* - relative accuracy goal as a number of bits, i.e. target a relative error less than $\varepsilon_{rel} = 2^{-r}$ where $r = rel\_goal$ (note the sign: *rel_goal* should be nonnegative).
- *abs_tol* - absolute accuracy goal as a *mag_t* describing the error tolerance, i.e. target an absolute error less than $\varepsilon_{abs} = abs\_tol$.
- *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 *rel_goal* = *prec* and *abs_tol* = 2^{-prec}. It usually only makes sense to have *rel_goal* between 0 and *prec*.

The algorithm attempts to achieve an error of $\max(\varepsilon_{abs}, M\varepsilon_{rel})$ 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 $\varepsilon_{abs} \approx M\varepsilon_{rel}$ where M is a known estimate of the magnitude. Setting ε_{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 ε_{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 $\varepsilon_{abs} \approx M\varepsilon_{rel}$ 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 *options* parameter (documented below). To use all defaults, *NULL* can be passed for *options*.

Options for integration

type acb_calc_integrate_opt_struct

type acb_calc_integrate_opt_t

This structure contains several fields, explained below. An *acb_calc_integrate_opt_t* is defined as an array of *acb_calc_integrate_opt_struct* of length 1, permitting it to be passed by reference. An *acb_calc_integrate_opt_t* must be initialized before use, which sets all fields to 0 or *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.

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 $0.5 \cdot \min(prec, rel\_goal) + 60$ 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.

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 $1000 \cdot prec + prec^2$. 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.

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.

10.2.3 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

$$\left| I - \sum_{k=0}^{n-1} w_k f(x_k) \right| \le \frac{64M}{15(\rho-1)\rho^{2n-1}}$$

if f is holomorphic with $|f(z)| \le M$ inside the ellipse E with foci ± 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 $\Delta f(\Delta I + m)$ 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^{2^k}. 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.

10.2.4 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

$$C(x, r) = \frac{1}{2\pi r} \oint_{|z-x|=r} |f(z)|dz = \int_0^1 |f(x + re^{2\pi it})|dt$$

where f is specified by (*func*, *param*) and r is given by *radius*. The integral is computed using a simple step sum. The integration range is subdivided until the order of magnitude of b can be determined (i.e. its error bound is smaller than its midpoint), or until the step length has been cut in half *maxdepth* times. This function is currently implemented completely naively, and repeatedly subdivides the whole integration range instead of performing adaptive subdivisions.

int **acb_calc_integrate_taylor**(*acb_t res*, *acb_calc_func_t func*, void **param*, **const** *acb_t a*, **const** *acb_t b*, **const** *arf_t inner_radius*, **const** *arf_t outer_radius*, *slong accuracy_goal*, *slong prec*)

Computes the integral

$$I = \int_a^b f(t)dt$$

where f is specified by (*func*, *param*), following a straight-line path between the complex numbers a and b which both must be finite.

The integral is approximated by piecewise centered Taylor polynomials. Rigorous truncation error bounds are calculated using the Cauchy integral formula. More precisely, if the Taylor series of f centered at the point m is $f(m+x) = \sum_{n=0}^{\infty} a_n x^n$, then

$$\int f(m+x) = \left(\sum_{n=0}^{N-1} a_n \frac{x^{n+1}}{n+1}\right) + \left(\sum_{n=N}^{\infty} a_n \frac{x^{n+1}}{n+1}\right).$$

For sufficiently small x, the second series converges and its absolute value is bounded by

$$\sum_{n=N}^{\infty} \frac{C(m,R)}{R^n} \frac{|x|^{n+1}}{N+1} = \frac{C(m,R)Rx}{(R-x)(N+1)} \left(\frac{x}{R}\right)^N.$$

It is required that any singularities of f are isolated from the path of integration by a distance strictly greater than the positive value *outer_radius* (which is the integration radius used for the Cauchy bound). Taylor series step lengths are chosen so as not to exceed *inner_radius*, which must be strictly smaller than *outer_radius* for convergence. A smaller *inner_radius* gives more rapid convergence of each Taylor series but means that more series might have to be used. A reasonable choice might be to set *inner_radius* to half the value of *outer_radius*, giving roughly one accurate bit per term.

The truncation point of each Taylor series is chosen so that the absolute truncation error is roughly 2^{-p} where p is given by *accuracy_goal* (in the future, this might change to a relative accuracy). Arithmetic operations and function evaluations are performed at a precision of *prec* bits. Note that due to accumulation of numerical errors, both values may have to be set higher (and the endpoints may have to be computed more accurately) to achieve a desired accuracy.

This function chooses the evaluation points uniformly rather than implementing adaptive subdivision.

CHAPTER

ELEVEN

EXTRA UTILITY MODULES

Mainly for internal use.

11.1 fmpz_extras.h – extra methods for FLINT integers

This module implements a few utility methods for the FLINT multiprecision integer type (*fmpz_t*). It is mainly intended for internal use.

11.1.1 Memory-related methods

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.

11.1.2 Convenience methods

void **fmpz_add_si**(*fmpz_t z*, **const** *fmpz_t x*, *slong y*)

void **fmpz_sub_si**(*fmpz_t z*, **const** *fmpz_t x*, *slong y*)
Sets z to the sum (respectively difference) of x and y.

void **fmpz_adiv_q_2exp**(*fmpz_t z*, **const** *fmpz_t x*, *flint_bitcnt_t exp*)
Sets z to $x/2^{exp}$, rounded away from zero.

void **fmpz_ui_mul_ui**(*fmpz_t x*, *ulong a*, *ulong b*)
Sets x to a times b.

void **fmpz_ui_pow_ui**(*fmpz_t x*, *ulong b*, *ulong e*)
Sets x to b raised to the power e.

void **fmpz_max**(*fmpz_t z*, **const** *fmpz_t x*, **const** *fmpz_t y*)

void **fmpz_min**(*fmpz_t z*, **const** *fmpz_t x*, **const** *fmpz_t y*)
Sets z to the maximum (respectively minimum) of x and y.

11.1.3 Inlined arithmetic

The *fmpz_t* bignum type uses an immediate representation for small integers, specifically when the absolute value is at most $2^{62} - 1$ (on 64-bit machines) or $2^{30} - 1$ (on 32-bit machines). The following methods completely inline the case where all operands (and possibly some intermediate values in the calculation) are known to be small. This is faster in code where all values *almost certainly will be much smaller than a full word*. In particular, these methods are used within Arb for manipulating exponents of floating-point numbers. Inlining slows down the general case, and increases code size, so these methods should not be used gratuitously.

void **fmpz_add_inline**(*fmpz_t* *z*, **const** *fmpz_t* *x*, **const** *fmpz_t* *y*)

void **fmpz_add_si_inline**(*fmpz_t* *z*, **const** *fmpz_t* *x*, *slong* *y*)

void **fmpz_add_ui_inline**(*fmpz_t* *z*, **const** *fmpz_t* *x*, *ulong* *y*)
Sets z to the sum of x and y.

void **fmpz_sub_si_inline**(*fmpz_t* *z*, **const** *fmpz_t* *x*, *slong* *y*)
Sets z to the difference of x and y.

void **fmpz_add2_fmpz_si_inline**(*fmpz_t* *z*, **const** *fmpz_t* *x*, **const** *fmpz_t* *y*, *slong* *c*)
Sets z to the sum of x, y, and c.

mp_size_t **_fmpz_size**(**const** *fmpz_t* *x*)
Returns the number of limbs required to represent x.

slong **_fmpz_sub_small**(**const** *fmpz_t* *x*, **const** *fmpz_t* *y*)
Computes the difference of x and y and returns the result as an *slong*. The result is clamped between -*WORD_MAX* and *WORD_MAX*, i.e. between $\pm(2^{63} - 1)$ inclusive on a 64-bit machine.

slong **_fmpz_set_si_small**(*fmpz_t* *x*, *slong* *v*)
Sets x to the integer v which is required to be a value between *COEFF_MIN* and *COEFF_MAX* so that promotion to a bignum cannot occur.

11.1.4 Low-level conversions

void **fmpz_set_mpn_large**(*fmpz_t* *z*, *mp_srcptr* *src*, *mp_size_t* *n*, int *negative*)
Sets z to the integer represented by the n limbs in the array *src*, or minus this value if *negative* is 1. Requires $n \geq 2$ and that the top limb of *src* is nonzero. Note that *fmpz_set_ui*, *fmpz_neg_ui* can be used for single-limb integers.

FMPZ_GET_MPN_READONLY(*zsign*, *zn*, *zptr*, *ztmp*, *zv*)
Given an *fmpz_t* *zv*, this macro sets *zptr* to a pointer to the limbs of *zv*, *zn* to the number of limbs, and *zsign* to a sign bit (0 if nonnegative, 1 if negative). The variable *ztmp* must be a single *mp_limb_t*, which is used as a buffer. If *zv* is a small value, *zv* itself contains no limb array that *zptr* could point to, so the single limb is copied to *ztmp* and *zptr* is set to point to *ztmp*. The case where *zv* is zero is not handled specially, and *zn* is set to 1.

void **fmpz_lshift_mpn**(*fmpz_t* *z*, *mp_srcptr* *src*, *mp_size_t* *n*, int *negative*, *flint_bitcnt_t* *shift*)
Sets z to the integer represented by the n limbs in the array *src*, or minus this value if *negative* is 1, shifted left by *shift* bits. Requires $n \geq 1$ and that the top limb of *src* is nonzero.

11.2 bool_mat.h – matrices over booleans

A *bool_mat_t* represents a dense matrix over the boolean semiring $\langle \{0, 1\}, \vee, \wedge \rangle$, implemented as an array of entries of type `int`.

The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

11.2.1 Types, macros and constants

type bool_mat_struct

type bool_mat_t
Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

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

int **bool_mat_get_entry**(**const** *bool_mat_t mat*, *slong i*, *slong j*)
Returns the entry of matrix *mat* at row i and column j.

void **bool_mat_set_entry**(*bool_mat_t mat*, *slong i*, *slong j*, int x)
Sets the entry of matrix *mat* at row i and column j to x.

bool_mat_nrows(*mat*)
Returns the number of rows of the matrix.

bool_mat_ncols(*mat*)
Returns the number of columns of the matrix.

11.2.2 Memory management

void **bool_mat_init**(*bool_mat_t mat*, *slong r*, *slong c*)
Initializes the matrix, setting it to the zero matrix with r rows and c columns.

void **bool_mat_clear**(*bool_mat_t mat*)
Clears the matrix, deallocating all entries.

int **bool_mat_is_empty**(**const** *bool_mat_t mat*)
Returns nonzero iff the number of rows or the number of columns in *mat* is zero. Note that this does not depend on the entry values of *mat*.

int **bool_mat_is_square**(**const** *bool_mat_t mat*)
Returns nonzero iff the number of rows is equal to the number of columns in *mat*.

11.2.3 Conversions

void **bool_mat_set**(*bool_mat_t dest*, **const** *bool_mat_t src*)
Sets *dest* to *src*. The operands must have identical dimensions.

11.2.4 Input and output

void **bool_mat_print**(**const** *bool_mat_t mat*)
Prints each entry in the matrix.

void **bool_mat_fprint**(FILE **file*, **const** *bool_mat_t mat*)
Prints each entry in the matrix to the stream *file*.

11.2.5 Value comparisons

int **bool_mat_equal**(**const** *bool_mat_t mat1*, **const** *bool_mat_t mat2*)
Returns nonzero iff the matrices have the same dimensions and identical entries.

int **bool_mat_any**(**const** *bool_mat_t mat*)
Returns nonzero iff *mat* has a nonzero entry.

int **bool_mat_all**(**const** *bool_mat_t mat*)
Returns nonzero iff all entries of *mat* are nonzero.

int **bool_mat_is_diagonal**(**const** *bool_mat_t A*)
Returns nonzero iff $i \neq j \implies \bar{A}_{ij}$.

int **bool_mat_is_lower_triangular**(**const** *bool_mat_t A*)
Returns nonzero iff $i < j \implies \bar{A}_{ij}$.

int **bool_mat_is_transitive**(**const** *bool_mat_t mat*)
Returns nonzero iff $A_{ij} \wedge A_{jk} \implies A_{ik}$.

int **bool_mat_is_nilpotent**(**const** *bool_mat_t A*)
Returns nonzero iff some positive matrix power of A is zero.

11.2.6 Random generation

void **bool_mat_randtest**(*bool_mat_t mat*, flint_rand_t *state*)
Sets *mat* to a random matrix.

void **bool_mat_randtest_diagonal**(*bool_mat_t mat*, flint_rand_t *state*)
Sets *mat* to a random diagonal matrix.

void **bool_mat_randtest_nilpotent**(*bool_mat_t mat*, flint_rand_t *state*)
Sets *mat* to a random nilpotent matrix.

11.2.7 Special matrices

void **bool_mat_zero**(*bool_mat_t mat*)
Sets all entries in mat to zero.

void **bool_mat_one**(*bool_mat_t mat*)
Sets the entries on the main diagonal to ones, and all other entries to zero.

void **bool_mat_directed_path**(*bool_mat_t A*)
Sets A_{ij} to $j = i + 1$. Requires that A is a square matrix.

void **bool_mat_directed_cycle**(*bool_mat_t A*)
Sets A_{ij} to $j = (i + 1) \mod n$ where n is the order of the square matrix A.

11.2.8 Transpose

void **bool_mat_transpose**(*bool_mat_t dest*, **const** *bool_mat_t src*)
Sets *dest* to the transpose of *src*. The operands must have compatible dimensions. Aliasing is allowed.

11.2.9 Arithmetic

void **bool_mat_complement**(*bool_mat_t B*, **const** *bool_mat_t A*)
Sets B to the logical complement of A. That is B_{ij} is set to $\bar{A}_{ij}$. The operands must have the same dimensions.

void **bool_mat_add**(*bool_mat_t res*, **const** *bool_mat_t mat1*, **const** *bool_mat_t mat2*)
Sets *res* to the sum of *mat1* and *mat2*. The operands must have the same dimensions.

void **bool_mat_mul**(*bool_mat_t res*, **const** *bool_mat_t mat1*, **const** *bool_mat_t mat2*)
Sets *res* to the matrix product of *mat1* and *mat2*. The operands must have compatible dimensions for matrix multiplication.

void **bool_mat_mul_entrywise**(*bool_mat_t res*, **const** *bool_mat_t mat1*, **const** *bool_mat_t mat2*)
Sets *res* to the entrywise product of *mat1* and *mat2*. The operands must have the same dimensions.

void **bool_mat_sqr**(*bool_mat_t B*, **const** *bool_mat_t A*)
Sets B to the matrix square of A. The operands must both be square with the same dimensions.

void **bool_mat_pow_ui**(*bool_mat_t B*, **const** *bool_mat_t A*, *ulong exp*)
Sets B to A raised to the power *exp*. Requires that A is a square matrix.

11.2.10 Special functions

int **bool_mat_trace**(**const** *bool_mat_t mat*)
Returns the trace of the matrix, i.e. the sum of entries on the main diagonal of *mat*. The matrix is required to be square. The sum is in the boolean semiring, so this function returns nonzero iff any entry on the diagonal of *mat* is nonzero.

slong **bool_mat_nilpotency_degree**(**const** *bool_mat_t A*)
Returns the nilpotency degree of the $n \times n$ matrix A. It returns the smallest positive k such that $A^k = 0$. If no such k exists then the function returns -1 if n is positive, and otherwise it returns 0.

void **bool_mat_transitive_closure**(*bool_mat_t B*, **const** *bool_mat_t A*)
Sets B to the transitive closure $\sum_{k=1}^{\infty} A^k$. The matrix A is required to be square.

slong **bool_mat_get_strongly_connected_components**(*slong* *p, **const** *bool_mat_t A*)
Partitions the n row and column indices of the $n \times n$ matrix A according to the strongly connected components (SCC) of the graph for which A is the adjacency matrix. If the graph has k SCCs then the function returns k, and for each vertex $i \in [0, n-1]$, p_i is set to the index of the SCC to which the vertex belongs. The SCCs themselves can be considered as nodes in a directed acyclic graph (DAG), and the SCCs are indexed in postorder with respect to that DAG.

slong **bool_mat_all_pairs_longest_walk**(*fmpz_mat_t B*, **const** *bool_mat_t A*)
Sets B_{ij} to the length of the longest walk with endpoint vertices i and j in the graph whose adjacency matrix is A. The matrix A must be square. Empty walks with zero length which begin and end at the same vertex are allowed. If j is not reachable from i then no walk from i to j exists and B_{ij} is set to the special value -1. If arbitrarily long walks from i to j exist then B_{ij} is set to the special value -2.

The function returns -2 if any entry of B_{ij} is -2, and otherwise it returns the maximum entry in B, except if A is empty in which case -1 is returned. Note that the returned value is one less than that of `nilpotency_degree()`.

This function can help quantify entrywise errors in a truncated evaluation of a matrix power series. If A is an indictor matrix with the same sparsity pattern as a matrix M over the real or complex numbers, and if B_{ij} does not take the special value -2, then the tail $\left[\sum_{k=N}^{\infty} a_k M^k\right]_{ij}$ vanishes when $N > B_{ij}$.

11.3 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.

11.3.1 Types, macros and constants

DLOG_NONE
Return value when the discrete logarithm does not exist

type dlog_precomp_struct

type 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.

11.3.2 Single evaluation

ulong **dlog_once**(*ulong* *b*, *ulong* *a*, **const** nmod_t *mod*, *ulong* *n*)
Return x such that $b = a^x$ in $(\mathbb{Z}/mod\mathbb{Z})^\times$, where a is known to have order n.

11.3.3 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 pe 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.

11.3.4 Vector evaluations

These functions compute all logarithms of successive integers $1 \dots n$.

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 value *va* for $0 \le k < nv$, 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 $\log(k, a) \times v_a$ 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*)

void **dlog_vec_eratos_add**(*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_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 the present implementation, the full index-calculus method is not implemented.

11.3.5 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

type dlog_table_struct

type 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

type dlog_bsgs_struct

type 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 rought measure of the cost of each logarithm using this table. The user should take $m \approx \sqrt{kn}$ 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

type dlog_modpe_struct

type 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

type dlog_crt_struct

type 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

type dlog_power_struct

type 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

type dlog_rho_struct

type 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.

11.4 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.

11.4.1 Types, macros and constants

type 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.

type 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.

type 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.

11.4.2 Memory management

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

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

11.4.3 Special values

void **fmpr_zero**(*fmpr_t* *x*)

void **fmpr_one**(*fmpr_t* *x*)

void **fmpr_pos_inf**(*fmpr_t* *x*)

void **fmpr_neg_inf**(*fmpr_t* *x*)

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

int **fmpr_is_zero**(const *fmpr_t* *x*)

int **fmpr_is_one**(const *fmpr_t* *x*)

int **fmpr_is_pos_inf**(const *fmpr_t* *x*)

int **fmpr_is_neg_inf**(const *fmpr_t* *x*)

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

int **fmpr_is_inf**(const *fmpr_t* *x*)
: Returns nonzero iff x equals either $+\infty$ or $-\infty$.

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

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

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

11.4.4 Assignment, rounding and conversions

slong **_fmpr_normalise**(*fmpz_t man*, *fmpz_t exp*, *slong prec*, *fmpr_rnd_t rnd*)
Rounds the mantissa and exponent in-place.

void **fmpr_set**(*fmpr_t y*, **const** *fmpr_t x*)
Sets y to a copy of x.

void **fmpr_swap**(*fmpr_t x*, *fmpr_t y*)
Swaps x and y efficiently.

slong **fmpr_set_round**(*fmpr_t y*, **const** *fmpr_t x*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_set_round_fmpz**(*fmpr_t y*, **const** *fmpz_t x*, *slong prec*, *fmpr_rnd_t rnd*)
Sets y to a copy of x rounded in the direction specified by rnd to the number of bits specified by prec.

slong **_fmpr_set_round_mpn**(*slong* *shift, *fmpz_t man*, *mp_srcptr x*, *mp_size_t xn*, int *negative*, *slong prec*, *fmpr_rnd_t rnd*)
Given an integer represented by a pointer x to a raw array of xn limbs (negated if *negative* is nonzero), sets *man* to the corresponding floating-point mantissa rounded to *prec* bits in direction *rnd*, sets *shift* to the exponent, and returns the error bound. We require that xn is positive and that the leading limb of x is nonzero.

slong **fmpr_set_round_ui_2exp_fmpz**(*fmpr_t z*, *mp_limb_t lo*, **const** *fmpz_t exp*, int *negative*, *slong prec*, *fmpr_rnd_t rnd*)
Sets z to the unsigned integer *lo* times two to the power *exp*, negating the value if *negative* is nonzero, and rounding the result to *prec* bits in direction *rnd*.

slong **fmpr_set_round_uiui_2exp_fmpz**(*fmpr_t z*, *mp_limb_t hi*, *mp_limb_t lo*, **const** *fmpz_t exp*, int *negative*, *slong prec*, *fmpr_rnd_t rnd*)
Sets z to the unsigned two-limb integer *{hi, lo}* times two to the power *exp*, negating the value if *negative* is nonzero, and rounding the result to *prec* bits in direction *rnd*.

void **fmpr_set_error_result**(*fmpr_t err*, **const** *fmpr_t result*, *slong rret*)
Given the return value *rret* and output variable *result* from a function performing a rounding (e.g. *fmpr_set_round* or *fmpr_add*), sets *err* to a bound for the absolute error.

void **fmpr_add_error_result**(*fmpr_t err*, **const** *fmpr_t err_in*, **const** *fmpr_t result*, *slong rret*, *slong prec*, *fmpr_rnd_t rnd*)
Like *fmpr_set_error_result*, but adds *err_in* to the error.

void **fmpr_ulp**(*fmpr_t u*, **const** *fmpr_t x*, *slong prec*)
Sets u to the floating-point unit in the last place (ulp) of x. The ulp is defined as in the MPFR documentation and satisfies $2^{-n}|x| < u \leq 2^{-n+1}|x|$ for any finite nonzero x. If x is a special value, u is set to the absolute value of x.

int **fmpr_check_ulp**(**const** *fmpr_t x*, *slong r*, *slong prec*)
Assume that r is the return code and x is the floating-point result from a single floating-point rounding. Then this function returns nonzero iff x and r define an error of exactly 0 or 1 ulp. In other words, this function checks that *fmpr_set_error_result()* gives exactly 0 or 1 ulp as expected.

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

void **fmpr_set_mpfr**(*fmpr_t x*, **const** mpfr_t *y*)
Sets x to the exact value of the MPFR variable y.

double **fmpr_get_d**(**const** *fmpr_t x*, *fmpr_rnd_t rnd*)
Returns x rounded to a *double* in the direction specified by *rnd*.

void **fmpr_set_d**(*fmpr_t x*, double *v*)
Sets x the the exact value of the argument v of type *double*.

void **fmpr_set_ui**(*fmpr_t* *x*, *ulong* *c*)

void **fmpr_set_si**(*fmpr_t* *x*, *slong* *c*)

void **fmpr_set_fmpz**(*fmpr_t* *x*, **const** *fmpz_t* *c*)
Sets x exactly to the integer c.

void **fmpr_get_fmpz**(*fmpz_t* *z*, **const** *fmpr_t* *x*, *fmpr_rnd_t* *rnd*)
Sets z to x rounded to the nearest integer in the direction specified by *rnd*. If *rnd* is *FMPR_RND_NEAR*, rounds to the nearest even integer in case of a tie. Aborts if x is infinite, NaN or if the exponent is unreasonably large.

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

void **fmpr_get_fmpq**(*fmpq_t* *y*, **const** *fmpr_t* *x*)
Sets y to the exact value of x. The result is undefined if x is not a finite fraction.

slong **fmpr_set_fmpq**(*fmpr_t* *x*, **const** *fmpq_t* *y*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets x to the value of y, rounded according to *prec* and *rnd*.

void **fmpr_set_fmpz_2exp**(*fmpr_t* *x*, **const** *fmpz_t* *man*, **const** *fmpz_t* *exp*)

void **fmpr_set_si_2exp_si**(*fmpr_t* *x*, *slong* *man*, *slong* *exp*)

void **fmpr_set_ui_2exp_si**(*fmpr_t* *x*, *ulong* *man*, *slong* *exp*)
Sets x to $\text{man} \times 2^{\text{exp}}$.

slong **fmpr_set_round_fmpz_2exp**(*fmpr_t* *x*, **const** *fmpz_t* *man*, **const** *fmpz_t* *exp*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets x to $\text{man} \times 2^{\text{exp}}$, rounded according to *prec* and *rnd*.

void **fmpr_get_fmpz_2exp**(*fmpz_t* *man*, *fmpz_t* *exp*, **const** *fmpr_t* *x*)
Sets *man* and *exp* to the unique integers such that $x = \text{man} \times 2^{\text{exp}}$ and *man* is odd, provided that x is a nonzero finite fraction. If x is zero, both *man* and *exp* are set to zero. If x is infinite or NaN, the result is undefined.

int **fmpr_get_fmpz_fixed_fmpz**(*fmpz_t* *y*, **const** *fmpr_t* *x*, **const** *fmpz_t* *e*)

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

11.4.5 Comparisons

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

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

int **fmpr_cmpabs**(**const** *fmpr_t* *x*, **const** *fmpr_t* *y*)

int **fmpr_cmpabs_ui**(**const** *fmpr_t* *x*, *ulong* *y*)
Compares the absolute values of x and y.

int **fmpr_cmp_2exp_si**(**const** *fmpr_t* *x*, *slong* *e*)

int **fmpr_cmpabs_2exp_si**(**const** *fmpr_t* *x*, *slong* *e*)
Compares x (respectively its absolute value) with 2^e.

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

void **fmpr_min**(*fmpr_t* *z*, **const** *fmpr_t* *a*, **const** *fmpr_t* *b*)

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

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

int **fmpr_is_int**(**const** *fmpr_t* *x*)
Returns nonzero iff x is integer-valued.

int **fmpr_is_int_2exp_si**(**const** *fmpr_t* *x*, *slong* *e*)
Returns nonzero iff x equals $n2^e$ for some integer n.

11.4.6 Random number generation

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

void **fmpr_randtest_not_zero**(*fmpr_t* *x*, flint_rand_t *state*, *slong* *bits*, *slong* *mag_bits*)
Identical to *fmpr_randtest*, except that zero is never produced as an output.

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

11.4.7 Input and output

void **fmpr_print**(**const** *fmpr_t* *x*)
Prints the mantissa and exponent of x as integers, precisely showing the internal representation.

void **fmpr_printd**(**const** *fmpr_t* *x*, *slong* *digits*)
Prints x as a decimal floating-point number, rounding to the specified number of digits. This function is currently implemented using MPFR, and does not support large exponents.

11.4.8 Arithmetic

void **fmpr_neg**(*fmpr_t* *y*, **const** *fmpr_t* *x*)
Sets y to the negation of x.

slong **fmpr_neg_round**(*fmpr_t* *y*, **const** *fmpr_t* *x*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets y to the negation of x, rounding the result.

void **fmpr_abs**(*fmpr_t* *y*, **const** *fmpr_t* *x*)
Sets y to the absolute value of x.

slong **fmpr_add**(*fmpr_t* *z*, **const** *fmpr_t* *x*, **const** *fmpr_t* *y*, *slong* *prec*, *fmpr_rnd_t* *rnd*)

slong **fmpr_add_ui**(*fmpr_t* *z*, **const** *fmpr_t* *x*, *ulong* *y*, *slong* *prec*, *fmpr_rnd_t* *rnd*)

slong **fmpr_add_si**(*fmpr_t* *z*, **const** *fmpr_t* *x*, *slong* *y*, *slong* *prec*, *fmpr_rnd_t* *rnd*)

slong **fmpr_add_fmpz**(*fmpr_t* *z*, **const** *fmpr_t* *x*, **const** *fmpz_t* *y*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets $z = x + y$, rounded according to *prec* and *rnd*. The precision can be *FMPR_PREC_EXACT* to perform an exact addition, provided that the result fits in memory.

slong **_fmpr_add_eps**(*fmpr_t* *z*, **const** *fmpr_t* *x*, int *sign*, *slong* *prec*, *fmpr_rnd_t* *rnd*)
Sets z to the value that results by adding an infinitesimal quantity of the given sign to x, and rounding. The result is undefined if x is zero.

slong **fmpr_sub**(*fmpr_t* *z*, **const** *fmpr_t* *x*, **const** *fmpr_t* *y*, *slong* *prec*, *fmpr_rnd_t* *rnd*)

slong **fmpr_sub_ui**(*fmpr_t z*, **const** *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_sub_si**(*fmpr_t z*, **const** *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_sub_fmpz**(*fmpr_t z*, **const** *fmpr_t x*, **const** *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = x - y$, rounded according to *prec* and *rnd*. The precision can be *FMPR_PREC_EXACT* to perform an exact addition, provided that the result fits in memory.

slong **fmpr_mul**(*fmpr_t z*, **const** *fmpr_t x*, **const** *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_mul_ui**(*fmpr_t z*, **const** *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_mul_si**(*fmpr_t z*, **const** *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_mul_fmpz**(*fmpr_t z*, **const** *fmpr_t x*, **const** *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = x \times y$, rounded according to prec and rnd. The precision can be *FMPR_PREC_EXACT* to perform an exact multiplication, provided that the result fits in memory.

void **fmpr_mul_2exp_si**(*fmpr_t y*, **const** *fmpr_t x*, *slong e*)

void **fmpr_mul_2exp_fmpz**(*fmpr_t y*, **const** *fmpr_t x*, **const** *fmpz_t e*)
Sets y to x multiplied by 2^e without rounding.

slong **fmpr_div**(*fmpr_t z*, **const** *fmpr_t x*, **const** *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_div_ui**(*fmpr_t z*, **const** *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_ui_div**(*fmpr_t z*, *ulong x*, **const** *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_div_si**(*fmpr_t z*, **const** *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_si_div**(*fmpr_t z*, *slong x*, **const** *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_div_fmpz**(*fmpr_t z*, **const** *fmpr_t x*, **const** *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_fmpz_div**(*fmpr_t z*, **const** *fmpz_t x*, **const** *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_fmpz_div_fmpz**(*fmpr_t z*, **const** *fmpz_t x*, **const** *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = x/y$, rounded according to *prec* and *rnd*. If y is zero, z is set to NaN.

slong **fmpr_addmul**(*fmpr_t z*, **const** *fmpr_t x*, **const** *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_addmul_ui**(*fmpr_t z*, **const** *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_addmul_si**(*fmpr_t z*, **const** *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_addmul_fmpz**(*fmpr_t z*, **const** *fmpr_t x*, **const** *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = z + x \times y$, rounded according to *prec* and *rnd*. The intermediate multiplication is always performed without roundoff. The precision can be *FMPR_PREC_EXACT* to perform an exact addition, provided that the result fits in memory.

slong **fmpr_submul**(*fmpr_t z*, **const** *fmpr_t x*, **const** *fmpr_t y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_submul_ui**(*fmpr_t z*, **const** *fmpr_t x*, *ulong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_submul_si**(*fmpr_t z*, **const** *fmpr_t x*, *slong y*, *slong prec*, *fmpr_rnd_t rnd*)

slong **fmpr_submul_fmpz**(*fmpr_t z*, **const** *fmpr_t x*, **const** *fmpz_t y*, *slong prec*, *fmpr_rnd_t rnd*)
Sets $z = z - x \times y$, rounded according to *prec* and *rnd*. The intermediate multiplication is always performed without roundoff. The precision can be *FMPR_PREC_EXACT* to perform an exact subtraction, provided that the result fits in memory.

slong **fmpr_sqrt**(*fmpr_t y*, **const** *fmpr_t x*, *slong prec*, *fmpr_rnd_t rnd*)
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 $b \geq 0$.

11.4.9 Special functions

slong **fmpr_log**(*fmpr_t y*, **const** *fmpr_t x*, *slong prec*, *fmpr_rnd_t rnd*)
Sets y to $\log(x)$, rounded according to *prec* and *rnd*. The result is NaN if x is negative. This function is currently implemented using MPFR and does not support large exponents.

slong **fmpr_log1p**(*fmpr_t y*, **const** *fmpr_t x*, *slong prec*, *fmpr_rnd_t rnd*)
Sets y to $\log(1+x)$, rounded according to *prec* and *rnd*. This function computes an accurate value when x is small. The result is NaN if $1 + x$ is negative. This function is currently implemented using MPFR and does not support large exponents.

slong **fmpr_exp**(*fmpr_t y*, **const** *fmpr_t x*, *slong prec*, *fmpr_rnd_t rnd*)
Sets y to $\exp(x)$, rounded according to *prec* and *rnd*. This function is currently implemented using MPFR and does not support large exponents.

slong **fmpr_expm1**(*fmpr_t y*, **const** *fmpr_t x*, *slong prec*, *fmpr_rnd_t rnd*)
Sets y to $\exp(x) - 1$, rounded according to *prec* and *rnd*. This function computes an accurate value when x is small. This function is currently implemented using MPFR and does not support large exponents.

CHAPTER

TWELVE

SUPPLEMENTARY ALGORITHM NOTES

Here, we give extra proofs, error bounds, and formulas that would be too lengthy to reproduce in the documentation for each module.

12.1 General formulas and bounds

This section collects some results from real and complex analysis that are useful when deriving error bounds. Beware of typos.

12.1.1 Error propagation

We want to bound the error when $f(x+a)$ is approximated by $f(x)$. Specifically, the goal is to bound $f(x+a)-f(x)$ in terms of r for the set of values a with $|a| \le r$. Most bounds will be monotone increasing with $|a|$ (assuming that x is fixed), so for brevity we simply express the bounds in terms of $|a|$.

Theorem (generic first-order bound):

$$|f(x+a) - f(x)| \le \min(2C_0, C_1|a|)$$

where

$$C_0 = \sup_{|t| \le |a|} |f(x+t)|, \quad C_1 = \sup_{|t| \le |a|} |f'(x+t)|.$$

The statement is valid with either $a, t \in \mathbb{R}$ or $a, t \in \mathbb{C}$.

Theorem (product): For $x, y \in \mathbb{C}$ and $a, b \in \mathbb{C}$,

$$|(x+a)(y+b) - xy| \le |xb| + |ya| + |ab|.$$

Theorem (quotient): For $x, y \in \mathbb{C}$ and $a, b \in \mathbb{C}$ with $|b| < |y|$,

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

Theorem (square root): For $x, a \in \mathbb{R}$ with $0 \le |a| \le x$,

$$\left|\sqrt{x+a} - \sqrt{x}\right| \le \sqrt{x}\left(1 - \sqrt{1 - \frac{|a|}{x}}\right) \le \frac{\sqrt{x}}{2}\left(\frac{|a|}{x} + \frac{|a|^2}{x^2}\right)$$

where the first inequality is an equality if $a \le 0$. (When $x = a = 0$, the limiting value is 0.)

Theorem (reciprocal square root): For $x, a \in \mathbb{R}$ with $0 \le |a| < x$,

$$\left|\frac{1}{\sqrt{x+a}} - \frac{1}{\sqrt{x}}\right| \le \frac{|a|}{2(x - |a|)^{3/2}}.$$

Theorem (k-th root): For $k > 1$ and $x, a \in \mathbb{R}$ with $0 \le |a| \le x$,

$$\left|(x+a)^{1/k} - x^{1/k}\right| \le x^{1/k} \min\left(1, \frac{1}{k} \log\left(1 + \frac{|a|}{x-|a|}\right)\right).$$

Proof: The error is largest when $a = -r$ is negative, and

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

Theorem (sine, cosine): For $x, a \in \mathbb{R}$, $|\sin(x+a) - \sin(x)| \le \min(2, |a|)$.

Theorem (logarithm): For $x, a \in \mathbb{R}$ with $0 \le |a| < x$,

$$|\log(x+a) - \log(x)| \le \log\left(1 + \frac{|a|}{x-|a|}\right),$$

with equality if $a \le 0$.

Theorem (exponential): For $x, a \in \mathbb{R}$, $|e^{x+a} - e^x| = e^x(e^a - 1) \le e^x(e^{|a|} - 1)$, with equality if $a \ge 0$.

Theorem (inverse tangent): For $x, a \in \mathbb{R}$,

$$|\operatorname{atan}(x+a) - \operatorname{atan}(x)| \le \min(\pi, C_1|a|).$$

where

$$C_1 = \sup_{|t| \le |a|} \frac{1}{1+(x+t)^2}.$$

If $|a| < |x|$, then $C_1 = (1 + (|x| - |a|)^2)^{-1}$ gives a monotone bound.

An exact bound: if $|a| < |x|$ or $|x(x+a)| < 1$, then

$$|\operatorname{atan}(x+a) - \operatorname{atan}(x)| = \operatorname{atan}\left(\frac{|a|}{1+x(x+a)}\right).$$

In the last formula, a case distinction has to be made depending on the signs of x and a.

12.1.2 Sums and series

Theorem (geometric bound): If $|c_k| \le C$ and $|z| \le D < 1$, then

$$\left|\sum_{k=N}^{\infty} c_k z^k\right| \le \frac{CD^N}{1-D}.$$

Theorem (integral bound): If $f(x)$ is nonnegative and monotone decreasing, then

$$\int_N^{\infty} f(x) \le \sum_{k=N}^{\infty} f(k) \le f(N) + \int_N^{\infty} f(x)dx.$$

12.1.3 Complex analytic functions

Theorem (Cauchy's integral formula): If $f(z) = \sum_{k=0}^{\infty} c_k z^k$ is analytic (on an open subset of $\mathbb{C}$ containing the disk $D = \{z : |z| \le R\}$ in its interior, where $R > 0$), then

$$c_k = \frac{1}{2\pi i} \int_{|z|=R} \frac{f(z)}{z^{k+1}}\, dz.$$

Corollary (derivative bound):

$$|c_k| \le \frac{C}{R^k}, \quad C = \max_{|z|=R} |f(z)|.$$

Corollary (Taylor series tail): If $0 \le r < R$ and $|z| \le r$, then

$$\left| \sum_{k=N}^{\infty} c_k z^k \right| \le \frac{CD^N}{1-D}, \quad D = \left|\frac{r}{R}\right|.$$

12.1.4 Euler-Maclaurin formula

Theorem (Euler-Maclaurin): If $f(t)$ is $2M$-times differentiable, then

$$\sum_{k=L}^{U} f(k) = S + I + T + R$$

$$S = \sum_{k=L}^{N-1} f(k), \quad I = \int_N^U f(t)dt,$$

$$T = \frac{1}{2}\left(f(N) + f(U)\right) + \sum_{k=1}^{M} \frac{B_{2k}}{(2k)!}\left(f^{(2k-1)}(U) - f^{(2k-1)}(N)\right),$$

$$R = -\int_N^U \frac{B_{2M}(t - \lfloor t \rfloor)}{(2M)!} f^{(2M)}(t)dt.$$

Lemma (Bernoulli polynomials): $|B_n(t - \lfloor t \rfloor)| \le 4n!/(2\pi)^n$.

Theorem (remainder bound):

$$|R| \le \frac{4}{(2\pi)^{2M}} \int_N^U \left| f^{(2M)}(t) \right| dt.$$

Theorem (parameter derivatives): If $f(t) = f(t,x) = \sum_{k=0}^{\infty} a_k(t)x^k$ and $R = R(x) = \sum_{k=0}^{\infty} c_k x^k$ are analytic functions of x, then

$$|c_k| \le \frac{4}{(2\pi)^{2M}} \int_N^U |a_k^{(2M)}(t)|dt.$$

12.2 Algorithms for mathematical constants

Most mathematical constants are evaluated using the generic hypergeometric summation code.

12.2.1 Pi

π is computed using the Chudnovsky series

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}}$$

which is hypergeometric and adds roughly 14 digits per term. Methods based on the arithmetic-geometric mean seem to be slower by a factor three in practice.

A small trick is to compute $1/\sqrt{640320}$ instead of $\sqrt{640320}$ at the end.

12.2.2 Logarithms of integers

We use the formulas

$$\log(2) = \frac{3}{4}\sum_{k=0}^{\infty}\frac{(-1)^k(k!)^2}{2^k(2k+1)!}$$

$$\log(10) = 46\,\operatorname{atanh}(1/31) + 34\,\operatorname{atanh}(1/49) + 20\,\operatorname{atanh}(1/161)$$

12.2.3 Euler's constant

Euler's constant γ is computed using the Brent-McMillan formula ([BM1980], [MPFR2012])

$$\gamma = \frac{S_0(2n) - K_0(2n)}{I_0(2n)} - \log(n)$$

in which n is a free parameter and

$$S_0(x) = \sum_{k=0}^{\infty}\frac{H_k}{(k!)^2}\left(\frac{x}{2}\right)^{2k}, \quad I_0(x) = \sum_{k=0}^{\infty}\frac{1}{(k!)^2}\left(\frac{x}{2}\right)^{2k}$$

$$2xI_0(x)K_0(x) \sim \sum_{k=0}^{\infty}\frac{[(2k)!]^3}{(k!)^4 8^{2k}x^{2k}}.$$

All series are evaluated using binary splitting. The first two series are evaluated simultaneously, with the summation taken up to $k = N - 1$ inclusive where $N \geq \alpha n + 1$ and $\alpha \approx 4.9706257595442318644$ satisfies $\alpha(\log\alpha - 1) = 3$. The third series is taken up to $k = 2n - 1$ inclusive. With these parameters, it is shown in [BJ2013] that the error is bounded by $24e^{-8n}$.

12.2.4 Catalan's constant

Catalan's constant is computed using the hypergeometric series

$$C = \frac{1}{64}\sum_{k=1}^{\infty}\frac{256^k(580k^2 - 184k + 15)}{k^3(2k-1)\binom{6k}{3k}\binom{6k}{4k}\binom{4k}{2k}}$$

given in [PP2010].

12.2.5 Khinchin's constant

Khinchin's constant K_0 is computed using the formula

$$\log K_0 = \frac{1}{\log 2}\left[\sum_{k=2}^{N-1}\log\left(\frac{k-1}{k}\right)\log\left(\frac{k+1}{k}\right) + \sum_{n=1}^{\infty}\frac{\zeta(2n,N)}{n}\sum_{k=1}^{2n-1}\frac{(-1)^{k+1}}{k}\right]$$

where $N \geq 2$ is a free parameter that can be used for tuning [BBC1997]. If the infinite series is truncated after $n = M$, the remainder is smaller in absolute value than

$$\begin{aligned}\sum_{n=M+1}^{\infty}\zeta(2n,N) &= \sum_{n=M+1}^{\infty}\sum_{k=0}^{\infty}(k+N)^{-2n} \leq \sum_{n=M+1}^{\infty}\left(N^{-2n} + \int_0^{\infty}(t+N)^{-2n}dt\right) \\ &= \sum_{n=M+1}^{\infty}\frac{1}{N^{2n}}\left(1 + \frac{N}{2n-1}\right) \leq \sum_{n=M+1}^{\infty}\frac{N+1}{N^{2n}} = \frac{1}{N^{2M}(N-1)} \leq \frac{1}{N^{2M}}.\end{aligned}$$

Thus, for an error of at most 2^{-p} in the series, it is sufficient to choose $M \geq p/(2\log_2 N)$.

12.2.6 Glaisher's constant

Glaisher's constant $A = \exp(1/12 - \zeta'(-1))$ is computed directly from this formula. We don't use the reflection formula for the zeta function, as the arithmetic in Euler-Maclaurin summation is faster at $s = -1$ than at $s = 2$.

12.2.7 Apery's constant

Apery's constant $\zeta(3)$ is computed using the hypergeometric series

$$\zeta(3) = \frac{1}{64} \sum_{k=0}^{\infty} (-1)^k (205k^2 + 250k + 77) \frac{(k!)^{10}}{[(2k+1)!]^5}.$$

12.3 Algorithms for the gamma function

12.3.1 The Stirling series

In general, the gamma function is computed via the Stirling series

$$\log \Gamma(z) = \left(z - \frac{1}{2}\right) \log z - z + \frac{\ln 2\pi}{2} + \sum_{k=1}^{n-1} \frac{B_{2k}}{2k(2k-1)z^{2k-1}} + R(n, z)$$

where ([Olv1997] pp. 293-295) the remainder term is exactly

$$R_n(z) = \int_0^{\infty} \frac{B_{2n} - \tilde{B}_{2n}(x)}{2n(x+z)^{2n}} dx.$$

To evaluate the gamma function of a power series argument, we substitute $z \to z + t \in \mathbb{C}[[t]]$.

Using the bound for $|x + z|$ given by [Olv1997] and the fact that the numerator of the integrand is bounded in absolute value by $2|B_{2n}|$, the remainder can be shown to satisfy the bound

$$|[t^k] R_n(z+t)| \le 2|B_{2n}| \frac{\Gamma(2n+k-1)}{\Gamma(k+1)\Gamma(2n+1)} \, |z| \, \left(\frac{b}{|z|}\right)^{2n+k}$$

where $b = 1/\cos(\arg(z)/2)$. Note that by trigonometric identities, assuming that $z = x + yi$, we have $b = \sqrt{1+u^2}$ where

$$u = \frac{y}{\sqrt{x^2+y^2} + x} = \frac{\sqrt{x^2+y^2} - x}{y}.$$

To use the Stirling series at p-bit precision, we select parameters r, n such that the remainder $R(n, z)$ approximately is bounded by 2^{-p}. If $|z|$ is too small for the Stirling series to give sufficient accuracy directly, we first translate to $z + r$ using the formula $\Gamma(z) = \Gamma(z+r)/(z(z+1)(z+2)\cdots(z+r-1))$.

To obtain a remainder smaller than 2^{-p}, we must choose an r such that, in the real case, $z + r > \beta p$, where $\beta > \log(2)/(2\pi) \approx 0.11$. In practice, a slightly larger factor $\beta \approx 0.2$ more closely balances n and r. A much larger β (e.g. $\beta = 1$) could be used to reduce the number of Bernoulli numbers that have to be precomputed, at the expense of slower repeated evaluation.

12.3.2 Rational arguments

We use efficient methods to compute $y = \Gamma(p/q)$ where q is one of $1, 2, 3, 4, 6$ and p is a small integer.

The cases $\Gamma(1) = 1$ and $\Gamma(1/2) = \sqrt{\pi}$ are trivial. We reduce all remaining cases to $\Gamma(1/3)$ or $\Gamma(1/4)$ using the following relations:

$$\Gamma(2/3) = \frac{2\pi}{3^{1/2}\Gamma(1/3)}, \qquad \Gamma(3/4) = \frac{2^{1/2}\pi}{\Gamma(1/4)},$$

$$\Gamma(1/6) = \frac{\Gamma(1/3)^2}{(\pi/3)^{1/2}2^{1/3}}, \qquad \Gamma(5/6) = \frac{2\pi(\pi/3)^{1/2}2^{1/3}}{\Gamma(1/3)^2}.$$

We compute $\Gamma(1/3)$ and $\Gamma(1/4)$ rapidly to high precision using

$$\Gamma(1/3) = \left(\frac{12\pi^4}{\sqrt{10}} \sum_{k=0}^{\infty} \frac{(6k)!(-1)^k}{(k!)^3(3k)!3^k 160^{3k}}\right)^{1/6}, \qquad \Gamma(1/4) = \sqrt{\frac{(2\pi)^{3/2}}{\operatorname{agm}(1, \sqrt{2})}}.$$

An alternative formula which could be used for $\Gamma(1/3)$ is

$$\Gamma(1/3) = \frac{2^{4/9}\pi^{2/3}}{3^{1/12}\left(\operatorname{agm}\left(1, \frac{1}{2}\sqrt{2+\sqrt{3}}\right)\right)^{1/3}},$$

but this appears to be slightly slower in practice.

12.4 Algorithms for the Hurwitz zeta function

12.4.1 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].

12.4.2 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 $\sigma = \operatorname{re}(s)$ with $K + \sigma > 1$. The tail is bounded by

$$\sum_{k=K}^{\infty} |x|^k \frac{|(s)_k|}{k!} \zeta(\sigma+k, a) \leq \sum_{k=K}^{\infty} |x|^k \frac{|(s)_k|}{k!} \left[\frac{1}{a^{\sigma+k}} + \frac{1}{(\sigma+k-1)a^{\sigma+k-1}}\right].$$

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 $C < 1$,

$$\sum_{k=K}^{\infty} T(k) \leq \frac{T(K)}{1-C}.$$

12.5 Algorithms for polylogarithms

The polylogarithm is defined for $s, z \in \mathbb{C}$ with $|z| < 1$ by

$$\mathrm{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}$$

and for $|z| \geq 1$ by analytic continuation, except for the singular point $z = 1$.

12.5.1 Computation for small z

The power sum converges rapidly when $|z| \ll 1$. To compute the series expansion with respect to s, we substitute $s \to s + x \in \mathbb{C}[[x]]$ and obtain

$$\mathrm{Li}_{s+x}(z) = \sum_{d=0}^{\infty} x^d \frac{(-1)^d}{d!} \sum_{k=1}^{\infty} T(k)$$

where

$$T(k) = \frac{z^k \log^d(k)}{k^s}.$$

The remainder term $|\sum_{k=N}^{\infty} T(k)|$ is bounded via the following strategy, implemented in *mag_polylog_tail()*.

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

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

and

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

12.5.2 Expansion for general z

For general complex s, z, we write the polylogarithm as a sum of two Hurwitz zeta functions

$$\mathrm{Li}_s(z) = \frac{\Gamma(v)}{(2\pi)^v} \left[i^v \zeta\left(v, \frac{1}{2} + \frac{\log(-z)}{2\pi i}\right) + i^{-v} \zeta\left(v, \frac{1}{2} - \frac{\log(-z)}{2\pi i}\right)\right]$$

in which $s = 1 - v$. With the principal branch of $\log(-z)$, we obtain the conventional analytic continuation of the polylogarithm with a branch cut on $z \in (1, +\infty)$.

To compute the series expansion with respect to v, we substitute $v \to v + x \in \mathbb{C}[[x]]$ in this formula (at the end of the computation, we map $x \to -x$ to obtain the power series for $\mathrm{Li}_{s+x}(z)$). The right hand side becomes

$$\Gamma(v+x)[E_1 Z_1 + E_2 Z_2]$$

where $E_1 = (i/(2\pi))^{v+x}$, $Z_1 = \zeta(v+x,\ldots)$, $E_2 = (1/(2\pi i))^{v+x}$, $Z_2 = \zeta(v+x,\ldots)$.

When $v = 1$, the Z_1 and Z_2 terms become Laurent series with a leading $1/x$ term. In this case, we compute the deflated series $\tilde{Z}_1, \tilde{Z}_2 = \zeta(x,\ldots) - 1/x$. Then

$$E_1 Z_1 + E_2 Z_2 = (E_1 + E_2)/x + E_1 \tilde{Z}_1 + E_2 \tilde{Z}_2.$$

Note that $(E_1 + E_2)/x$ is a power series, since the constant term in $E_1 + E_2$ is zero when $v = 1$. So we simply compute one extra derivative of both E_1 and E_2, and shift them one step. When $v = 0, -1, -2, \ldots$, the $\Gamma(v+x)$ prefactor has a pole. In this case, we proceed analogously and formally multiply $x\,\Gamma(v+x)$ with $[E_1 Z_1 + E_2 Z_2]/x$.

Note that the formal cancellation only works when the order s (or v) is an exact integer: it is not currently possible to use this method when s is a small ball containing any of $0, 1, 2, \ldots$ (then the result becomes indeterminate).

The Hurwitz zeta method becomes inefficient when $|z| \to 0$ (it gives an indeterminate result when $z = 0$). This is not a problem since we just use the defining series for the polylogarithm in that region. It also becomes inefficient when $|z| \to \infty$, for which an asymptotic expansion would better.

12.6 Algorithms for hypergeometric functions

The algorithms used to compute hypergeometric functions are described in [Joh2016]. Here, we state the most important error bounds.

12.6.1 Convergent series

Let

$$T(k) = \frac{\prod_{i=0}^{p-1} (a_i)_k}{\prod_{i=0}^{q-1} (b_i)_k} z^k.$$

We compute a factor C such that

$$\left| \sum_{k=n}^{\infty} T(k) \right| \le C |T(n)|.$$

We check that $\mathrm{Re}(b+n) > 0$ for all lower parameters b. If this does not hold, C is set to infinity. Otherwise, we cancel out pairs of parameters a and b against each other. We have

$$\left| \frac{a+k}{b+k} \right| = \left| 1 + \frac{a-b}{b+k} \right| \le 1 + \frac{|a-b|}{|b+n|}$$

and

$$\left| \frac{1}{b+k} \right| \le \frac{1}{|b+n|}$$

for all $k \ge n$. This gives us a constant D such that $T(k+1) \le DT(k)$ for all $k \ge n$. If $D \ge 1$, we set C to infinity. Otherwise, we take $C = \sum_{k=0}^{\infty} D^k = (1-D)^{-1}$.

12.6.2 Convergent series of power series

The same principle is used to get tail bounds for with $a_i, b_i, z \in \mathbb{C}[[x]]$, or more precisely, bounds for each coefficient in $\sum_{k=N}^{\infty} T(k) \in \mathbb{C}[[x]]/\langle x^n \rangle$ given $a_i, b_i, z \in \mathbb{C}[[x]]/\langle x^n \rangle$. First, we fix some notation, assuming that A and B are power series:

- $A_{[k]}$ denotes the coefficient of x^k in A, and $A_{[m:n]}$ denotes the power series $\sum_{k=m}^{n-1} A_{[k]} x^k$.
- $|A|$ denotes $\sum_{k=0}^{\infty} |A_{[k]}| x^k$ (this can be viewed as an element of $\mathbb{R}_{\geq 0}[[x]]$).
- $A \leq B$ signifies that $|A|_{[k]} \leq |B|_{[k]}$ holds for all k.
- We define $\mathcal{R}(B) = |B_{[0]}| - |B_{[1:\infty]}|$.

Using the formulas

$$(AB)_{[k]} = \sum_{j=0}^{k} A_{[j]} B_{[k-j]}, \quad (1/B)_{[k]} = \frac{1}{B_{[0]}} \sum_{j=1}^{k} -B_{[j]} (1/B)_{[k-j]},$$

it is easy prove the following bounds for the coefficients of sums, products and quotients of formal power series:

$$|A + B| \leq |A| + |B|, \quad |AB| \leq |A||B|, \quad |A/B| \leq |A|/\mathcal{R}(B).$$

If $p \leq q$ and $\operatorname{Re}(b_{i\,[0]} + N) > 0$ for all b_i, then we may take

$$D = |z| \prod_{i=1}^{p} \left(1 + \frac{|a_i - b_i|}{\mathcal{R}(b_i + N)}\right) \prod_{i=p+1}^{q} \frac{1}{\mathcal{R}(b_i + N)}.$$

If $D_{[0]} < 1$,then $(1 - D)^{-1} |T(n)|$ gives the error bound.

Note when adding and multiplying power series with (complex) interval coefficients, we can use point-valued upper bounds for the absolute values instead of performing interval arithmetic throughout. For $\mathcal{R}(B)$, we must then pick a lower bound for $|B_{[0]}|$ and upper bounds for the coefficients of $|B_{[1:\infty]}|$.

12.6.3 Asymptotic series for the confluent hypergeometric function

Let $U(a, b, z)$ denote the confluent hypergeometric function of the second kind with the principal branch cut, and let $U^* = z^a U(a, b, z)$. For all $z \neq 0$ and $b \notin \mathbb{Z}$ (but valid for all b as a limit), we have (DLMF 13.2.42)

$$U(a, b, z) = \frac{\Gamma(1 - b)}{\Gamma(a - b + 1)} M(a, b, z) + \frac{\Gamma(b - 1)}{\Gamma(a)} z^{1-b} M(a - b + 1, 2 - b, z).$$

Moreover, for all $z \neq 0$ we have

$$\frac{{}_1F_1(a, b, z)}{\Gamma(b)} = \frac{(-z)^{-a}}{\Gamma(b - a)} U^*(a, b, z) + \frac{z^{a-b} e^z}{\Gamma(a)} U^*(b - a, b, -z)$$

which is equivalent to DLMF 13.2.41 (but simpler in form).

We have the asymptotic expansion

$$U^*(a, b, z) \sim {}_2F_0(a, a - b + 1, -1/z)$$

where ${}_2F_0(a, b, z)$ denotes a formal hypergeometric series, i.e.

$$U^*(a, b, z) = \sum_{k=0}^{n-1} \frac{(a)_k (a - b + 1)_k}{k! (-z)^k} + \varepsilon_n(z).$$

The error term $\varepsilon_n(z)$ is bounded according to DLMF 13.7. A case distinction is made depending on whether z lies in one of three regions which we index by R. Our formula for the error bound increases

with the value of R, so we can always choose the larger out of two indices if z lies in the union of two regions.

Let $r = |b - 2a|$. If $\operatorname{Re}(z) \geq r$, set $R = 1$. Otherwise, if $\operatorname{Im}(z) \geq r$ or $\operatorname{Re}(z) \geq 0 \wedge |z| \geq r$, set $R = 2$. Otherwise, if $|z| \geq 2r$, set $R = 3$. Otherwise, the bound is infinite. If the bound is finite, we have

$$|\varepsilon_n(z)| \leq 2\alpha C_n \left| \frac{(a)_n (a-b+1)_n}{n! z^n} \right| \exp(2\alpha\rho C_1/|z|)$$

in terms of the following auxiliary quantities

$$\sigma = |(b-2a)/z|$$

$$C_n = \begin{cases} 1 & \text{if } R = 1 \\ \chi(n) & \text{if } R = 2 \\ (\chi(n) + \sigma\nu^2 n)\nu^n & \text{if } R = 3 \end{cases}$$

$$\nu = \left(\tfrac{1}{2} + \tfrac{1}{2}\sqrt{1-4\sigma^2}\right)^{-1/2} \leq 1 + 2\sigma^2$$

$$\chi(n) = \sqrt{\pi}\,\Gamma(\tfrac{1}{2}n + 1)/\Gamma(\tfrac{1}{2}n + \tfrac{1}{2})$$

$$\sigma' = \begin{cases} \sigma & \text{if } R \neq 3 \\ \nu\sigma & \text{if } R = 3 \end{cases}$$

$$\alpha = (1-\sigma')^{-1}$$

$$\rho = \tfrac{1}{2}|2a^2 - 2ab + b| + \sigma'(1 + \tfrac{1}{4}\sigma')(1-\sigma')^{-2}$$

12.6.4 Asymptotic series for Airy functions

Error bounds are based on Olver (DLMF section 9.7). For $\arg(z) < \pi$ and $\zeta = (2/3)z^{3/2}$, we have

$$\operatorname{Ai}(z) = \frac{e^{-\zeta}}{2\sqrt{\pi} z^{1/4}} \left[S_n(\zeta) + R_n(z)\right], \quad \operatorname{Ai}'(z) = -\frac{z^{1/4} e^{-\zeta}}{2\sqrt{\pi}} \left[(S_n'(\zeta) + R_n'(z)\right]$$

$$S_n(\zeta) = \sum_{k=0}^{n-1} (-1)^k \frac{u(k)}{\zeta^k}, \quad S_n'(\zeta) = \sum_{k=0}^{n-1} (-1)^k \frac{v(k)}{\zeta^k}$$

$$u(k) = \frac{(1/6)_k (5/6)_k}{2^k k!}, \quad v(k) = \frac{6k+1}{1-6k} u(k).$$

Assuming that n is positive, the error terms are bounded by

$$|R_n(z)| \leq C|u(n)||\zeta|^{-n}, \quad |R_n'(z)| \leq C|v(n)||\zeta|^{-n}$$

where

$$C = \begin{cases} 2\exp(7/(36|\zeta|)) & |\arg(z)| \leq \pi/3 \\ 2\chi(n)\exp(7\pi/(72|\zeta|)) & \pi/3 \leq |\arg(z)| \leq 2\pi/3 \\ 4\chi(n)\exp(7\pi/(36|\operatorname{re}(\zeta)|))|\cos(\arg(\zeta))|^{-n} & 2\pi/3 \leq |\arg(z)| < \pi. \end{cases}$$

For computing Bi when z is roughly in the positive half-plane, we use the connection formulas

$$\operatorname{Bi}(z) = -i(2w^{+1}\operatorname{Ai}(zw^{-2}) - \operatorname{Ai}(z))$$
$$\operatorname{Bi}(z) = +i(2w^{-1}\operatorname{Ai}(zw^{+2}) - \operatorname{Ai}(z))$$

where $w = \exp(\pi i/3)$. Combining roots of unity gives

$$\operatorname{Bi}(z) = \frac{1}{2\sqrt{\pi} z^{1/4}} [2X + iY]$$

$$\operatorname{Bi}(z) = \frac{1}{2\sqrt{\pi} z^{1/4}} [2X - iY]$$

$$X = \exp(+\zeta)[S_n(-\zeta) + R_n(z w^{\mp 2})], \quad Y = \exp(-\zeta)[S_n(\zeta) + R_n(z)]$$

where the upper formula is valid for $-\pi/3 < \arg(z) < \pi$ and the lower formula is valid for $-\pi < \arg(z) < \pi/3$. We proceed analogously for the derivative of Bi.

In the negative half-plane, we use the connection formulas

$$\operatorname{Ai}(z) = e^{+\pi i/3} \operatorname{Ai}(z_1) + e^{-\pi i/3} \operatorname{Ai}(z_2)$$

$$\operatorname{Bi}(z) = e^{-\pi i/6} \operatorname{Ai}(z_1) + e^{+\pi i/6} \operatorname{Ai}(z_2)$$

where $z_1 = -z e^{+\pi i/3}$, $z_2 = -z e^{-\pi i/3}$. Provided that $|\arg(-z)| < 2\pi/3$, we have $|\arg(z_1)|, |\arg(z_2)| < \pi$, and thus the asymptotic expansion for Ai can be used. As before, we collect roots of unity to obtain

$$\operatorname{Ai}(z) = A_1[S_n(i\zeta) + R_n(z_1)] + A_2[S_n(-i\zeta) + R_n(z_2)]$$

$$\operatorname{Bi}(z) = A_3[S_n(i\zeta) + R_n(z_1)] + A_4[S_n(-i\zeta) + R_n(z_2)]$$

where $\zeta = (2/3)(-z)^{3/2}$ and

$$A_1 = \frac{\exp(-i(\zeta - \pi/4))}{2\sqrt{\pi}(-z)^{1/4}}, \quad A_2 = \frac{\exp(+i(\zeta - \pi/4))}{2\sqrt{\pi}(-z)^{1/4}}, \quad A_3 = -iA_1, \quad A_4 = +iA_2.$$

The differentiated formulas are analogous.

12.6.5 Corner case of the Gauss hypergeometric function

In the corner case where z is near $\exp(\pm \pi i/3)$, none of the linear fractional transformations is effective. In this case, we use Taylor series to analytically continue the solution of the hypergeometric differential equation from the origin. The function $f(z) = {}_2F_1(a, b, c, z_0 + z)$ satisfies

$$f''(z) = -\frac{((z_0 + z)(a + b + 1) - c)}{(z_0 + z)(z_0 - 1 + z)} f'(z) - \frac{ab}{(z_0 + z)(z_0 - 1 + z)} f(z).$$

Knowing $f(0), f'(0)$, we can compute the consecutive derivatives recursively, and evaluating the truncated Taylor series allows us to compute $f(z), f'(z)$ to high accuracy for sufficiently small z. Some experimentation showed that two continuation steps

$$0 \quad \to \quad 0.375 \pm 0.625i \quad \to \quad 0.5 \pm 0.8125i \quad \to \quad z$$

gives good performance. Error bounds for the truncated Taylor series are obtained using the Cauchy-Kovalevskaya majorant method, following the outline in [Hoe2001]. The differential equation is majorized by

$$g''(z) = \frac{N + 1}{2} \left(\frac{\nu}{1 - \nu z} \right) g'(z) + \frac{(N + 1)N}{2} \left(\frac{\nu}{1 - \nu z} \right)^2 g(z)$$

provided that N and $\nu \geq \max(1/|z_0|, 1/|z_0 - 1|)$ are chosen sufficiently large. It follows that we can compute explicit numbers A, N, ν such that the simple solution $g(z) = A(1 - \nu z)^{-N}$ of the differential equation provides the bound

$$|f_{[k]}| \leq g_{[k]} = A \binom{N + k}{k} \nu^k.$$

12.7 Algorithms for the arithmetic-geometric mean

With complex variables, it is convenient to work with the univariate function $M(z) = \operatorname{agm}(1, z)$. The general case is given by $\operatorname{agm}(a, b) = aM(1, b/a)$.

12.7.1 Functional equation

If the real part of z initially is not completely nonnegative, we apply the functional equation $M(z) = (z+1)M(u)/2$ where $u = \sqrt{z}/(z+1)$.

Note that u has nonnegative real part, absent rounding error. It is not a problem for correctness if rounding makes the interval contain negative points, as this just inflates the final result.

For the derivative, the functional equation becomes $M'(z) = [M(u) - (z-1)M'(u)/((1+z)\sqrt{z})]/2$.

12.7.2 AGM iteration

Once z is in the right half plane, we can apply the AGM iteration ($2a_{n+1} = a_n + b_n, b_{n+1}^2 = a_n b_n$) directly. The correct square root is given by $\sqrt{a}\sqrt{b}$, which is computed as $\sqrt{ab}, i\sqrt{-ab}, -i\sqrt{-ab}, \sqrt{a}\sqrt{b}$ respectively if both a and b have positive real part, nonnegative imaginary part, nonpositive imaginary part, or otherwise.

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, $|M(z) - a_n| \le |a_n - b_n|$, 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

$$\operatorname{agm}(a, b) = \frac{(a+b)\pi}{4K(z^2)},$$

valid at least when $|z| < 1$ and a, b have nonnegative real part, and

$$\frac{\pi}{4K(z^2)} = \tfrac{1}{2} - \tfrac{1}{8}z^2 - \tfrac{5}{128}z^4 - \tfrac{11}{512}z^6 - \tfrac{469}{32768}z^8 + \ldots$$

where the tail is bounded by $\sum_{k=10}^{\infty} |z|^k/64$.

12.7.3 First derivative

Assuming that z is exact and that $|\arg(z)| \le 3\pi/4$, we compute $(M(z), M'(z))$ simultaneously using a finite difference.

The basic inequality we need is $|M(z)| \le \max(1, |z|)$, which is an immediate consequence of the AGM iteration.

By Cauchy's integral formula, $|M^{(k)}(z)/k!| \le CD^k$ where $C = \max(1, |z| + r)$ and $D = 1/r$, for any $0 < r < |z|$ (we choose r to be of the order $|z|/4$). Taylor expansion now gives

$$\left|\frac{M(z+h) - M(z)}{h} - M'(z)\right| \le \frac{CD^2h}{1-Dh}$$
$$\left|\frac{M(z+h) - M(z-h)}{2h} - M'(z)\right| \le \frac{CD^3h^2}{1-Dh}$$
$$\left|\frac{M(z+h) + M(z-h)}{2} - M(z)\right| \le \frac{CD^2h^2}{1-Dh}$$

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

$$|M'(z+\varepsilon)| \le \frac{\max(1, |z| + |\varepsilon| + r)}{r}$$
$$|M''(z+\varepsilon)| \le \frac{2\max(1, |z| + |\varepsilon| + r)}{r^2}$$

assuming that the circle centered on z with radius $|\varepsilon| + r$ does not cross the negative half axis. We choose r of order $|z|/2$ and verify that all assumptions hold.

12.7.4 Higher derivatives

The function $W(z) = 1/M(z)$ is D-finite. The coefficients of $W(z+x) = \sum_{k=0}^{\infty} c_k x^k$ satisfy

$$-2z(z^2-1)c_2 = (3z^2-1)c_1 + zc_0,$$

$$-(k+2)(k+3)z(z^2-1)c_{k+3} = (k+2)^2(3z^2-1)c_{k+2} + (3k(k+3)+7)zc_{k+1} + (k+1)^2 c_k$$

in general, and

$$-(k+2)^2 c_{k+2} = (3k(k+3)+7)c_{k+1} + (k+1)^2 c_k$$

when $z = 1$.

CHAPTER

THIRTEEN

VERSION HISTORY

13.1 History and changes

For more details, view the commit log in the git repository https://github.com/fredrik-johansson/arb

Old releases of the code can be accessed from https://github.com/fredrik-johansson/arb/releases

13.1.1 Old versions of the documentation

- http://arblib.org/arb-2.18.1.pdf
- http://arblib.org/arb-2.18.0.pdf
- http://arblib.org/arb-2.17.0.pdf
- http://arblib.org/arb-2.16.0.pdf
- http://arblib.org/arb-2.15.0.pdf
- http://arblib.org/arb-2.14.0.pdf
- http://arblib.org/arb-2.13.0.pdf
- http://arblib.org/arb-2.12.0.pdf
- http://arblib.org/arb-2.11.1.pdf
- http://arblib.org/arb-2.11.0.pdf
- http://arblib.org/arb-2.10.0.pdf
- http://arblib.org/arb-2.9.0.pdf
- http://arblib.org/arb-2.8.1.pdf
- http://arblib.org/arb-2.8.0.pdf
- http://arblib.org/arb-2.7.0.pdf
- http://arblib.org/arb-2.6.0.pdf
- http://arblib.org/arb-2.5.0.pdf
- http://arblib.org/arb-2.4.0.pdf
- http://arblib.org/arb-2.3.0.pdf

13.1.2 2020-06-25 – version 2.18.1

- Support MinGW64.
- Added version numbers (___ARB_VERSION, ___ARB_RELEASE, ARB_VERSION) to arb.h.

13.1.3 2020-06-09 – version 2.18.0

- General
  - Flint 2.6 support.
  - Several build system improvements (contributed by Isuru Fernando).
  - Changed arf_get_mpfr to return an MPFR underflow/overflow result (rounding to 0 or infinity with the right sign and MPFR overflow flags) instead of throwing flint_abort() if the exponent is out of bounds for MPFR.
  - Documentation and type corrections (contributed by Joel Dahne).
- Arithmetic
  - The number of iterations per precision level in arb_fmpz_poly_complex_roots has been tweaked to avoid extreme slowdown for some polynomials with closely clustered roots.
  - Added arb_contains_interior, acb_contains_interior.
- Special functions
  - Fixed unsafe shifts causing Dirichlet characters for certain moduli exceeding 32 bits to crash.
  - Added acb_agm for computing the arithmetic-geometric mean of two complex numbers.
  - acb_elliptic_rj now uses a slow fallback algorithm in cases where Carlson's algorithm is not known to be valid. This fixes instances where acb_elliptic_pi, acb_elliptic_pi_inc and acb_elliptic_rj previously ended up on the wrong branch. Users should be cautioned that the new version can give worse enclosures and sometimes fails to converge in some cases where the old algorithm did (the pi flag for acb_elliptic_pi_inc is useful as a workaround).
  - Optimized some special cases in acb_hurwitz_zeta.

13.1.4 2019-10-16 – version 2.17.0

- General
  - Added exact serialization methods (arb_dump_str, arb_load_str, arb_dump_file, arb_load_file, arf_dump_str, arf_load_str, arf_dump_file, arf_load_file, mag_dump_str, mag_load_str, mag_dump_file, mag_load_file) (contributed by Julian Rüth).
  - Removed many obsolete fmpr methods and de-inlined several helper functions to slightly improve compile time and library size.
  - Fixed a namespace clash for an internal function (contributed by Julian Rüth).
  - Added the helper function arb_sgn_nonzero.
  - Added the helper function acb_rel_one_accuracy_bits.
- Riemann zeta function
  - Added a function for efficiently computing individual zeros of the Riemann zeta function using Turing's method (acb_dirichlet_zeta_zero) (contributed by D.H.J. Polymath).
  - Added a function for counting zeros of the Riemann zeta function up to given height using Turing's method (acb_dirichlet_zeta_nzeros) (contributed by D.H.J. Polymath).
  - Added the Backlund S function (acb_dirichlet_backlund_s).

  - Added a function for computing Gram points (acb_dirichlet_gram_point).
  - Added acb_dirichlet_zeta_deriv_bound for quickly bounding the derivative of the Riemann zeta function.
  - Fast multi-evaluation of the Riemann zeta function using Platt's algorithm (acb_dirichlet_platt_multieval) (contributed by D.H.J. Polymath).
- Other special functions
  - Improved the algorithm in acb_hypgeom_u to estimate precision loss more accurately.
  - Implemented Coulomb wave functions (acb_hypgeom_coulomb, acb_hypgeom_coulomb_series and other functions).
  - Faster algorithm for Catalan's constant.
  - Added acb_modular_theta_series.
  - Added arb_poly_sinc_pi_series (contributed by D.H.J. Polymath).
  - Improved tuning in acb_hypgeom_pfq_series_sum for higher derivatives at high precision (reported by Mark Watkins).

13.1.5 2018-12-07 – version 2.16.0

- Linear algebra and arithmetic
  - Added acb_mat_approx_eig_qr for approximate computation of eigenvalues and eigenvectors of complex matrices.
  - Added acb_mat_eig_enclosure_rump implementing Rump's algorithm for certification of eigenvalue-eigenvector pairs as well as clusters.
  - Added acb_mat_eig_simple_rump for certified diagonalization of matrices with simple eigenvalues.
  - Added acb_mat_eig_simple_vdhoeven_mourrain, acb_mat_eig_simple for fast certified diagonalization of matrices with simple eigenvalues.
  - Added acb_mat_eig_multiple_rump, acb_mat_eig_multiple for certified computation of eigenvalues with possible overlap.
  - Added acb_mat_eig_global_enclosure for fast global inclusion of eigenvalues without isolation.
  - Added arb_mat_companion, acb_mat_companion for constructing companion matrices.
  - Added several arb_mat and acb_mat helper functions: indeterminate, is_exact, is_zero, is_finite, is_triu, is_tril, is_diag, diag_prod.
  - Added arb_mat_approx_inv, acb_mat_approx_inv.
  - Optimized arb_mat_mul_block by using arb_dot when the blocks are small.
  - Added acb_get_mid.
  - Updated hilbert_matrix example program.

13.1.6 2018-10-25 – version 2.15.1

- Fixed precision issue leading to spurious NaN results in incomplete elliptic integrals

13.1.7 2018-09-18 – version 2.15.0

- Arithmetic
  - Added arb_dot and acb_dot for efficient evaluation of dot products.
  - Added arb_approx_dot and acb_approx_dot for efficient evaluation of dot products without error bounds.
  - Converted loops to arb_dot and acb_dot in the arb_poly and acb_poly methods mullow_classical, inv_series, div_series, exp_series_basecase, sin_cos_series_basecase, sinh_cosh_series_basecase, evaluate_rectangular, evaluate2_rectangular, revert_series_lagrange_fast. Also changed the algorithm cutoffs for mullow, exp_series, sin_cos_series, sinh_cosh_series.
  - Converted loops to arb_dot and acb_dot in the arb_mat and acb_mat methods mul_classical, mul_threaded, solve_tril, solve_triu, charpoly. Also changed the algorithm cutoffs for mul, solve_tril, solve_triu.
  - Converted loops to arb_approx_dot and acb_approx_dot in the arb_mat and acb_mat methods approx_solve_tril, approx_solve_triu. Also changed the algorithm cutoffs.
  - Added arb_mat_approx_mul and acb_mat_approx_mul for matrix multiplication without error bounds.
- Miscellaneous
  - Added arb_hypgeom_airy_zero for computing zeros of Airy functions.
  - Added arb_hypgeom_dilog wrapper.
  - Optimized arb_const_pi and arb_const_log2 by using a static table at low precision, giving a small speedup and avoiding common recomputation when starting threads.
  - Optimized mag_set_ui_2exp_si.
  - Remove obsolete and unused function _arb_vec_dot.
  - Converted some inline functions to ordinary functions to reduce library size.
  - Fixed acb_dirichlet_stieltjes to use the integration algorithm also when a != 1.
  - Fixed test failure for acb_dirichlet_stieltjes on ARM64 (reported by Gianfranco Costamagna). Special thanks to Julien Puydt for assistance with debugging.
  - Fixed crash in acb_dft_bluestein with zero length (reported by Gianfranco Costamagna).

13.1.8 2018-07-22 – version 2.14.0

- Linear algebra
  - Faster and more accurate real matrix multiplication using block decomposition, scaling, and multiplying via FLINT integer matrices in combination with safe use of doubles for radius matrix multiplications.
  - Faster and more accurate complex matrix multiplication by reordering and taking advantage of real matrix multiplication.
  - The new multiplication algorithm methods (arb_mat_mul_block, acb_mat_mul_reorder) are used automatically by the main multiplication methods.

  - Faster and more accurate LU factorization by using a block recursive algorithm that takes advantage of matrix multiplication. Added separate algorithm methods (arb/acb)_mat_lu_(recursive/classical) with an automatic algorithm choice in the default lu methods.
  - Added methods (arb/acb)_mat_solve_(tril/triu) (and variants) for solving upper or lower triangular systems using a block recursive algorithm taking advantage of matrix multiplication.
  - Improved linear solving and inverse for large well-conditioned matrices by using a preconditioning algorithm. Added separate solving algorithm methods (arb/acb)_mat_solve_(lu/precond) with an automatic algorithm choice in the default solve methods (contributed by anonymous user arbguest).
  - Improved determinants using a preconditioning algorithm. Added separate determinant algorithm methods (arb/acb)_mat_det_(lu/precond) with an automatic algorithm choice in the default det methods.
  - Added automatic detection of triangular matrices in arb_mat_det and acb_mat_det.
  - Added arb_mat_solve_preapprox which allows certifying a precomputed approximate solution (contributed by anonymous user arbguest).
  - Added methods for constructing various useful test matrices: arb_mat_ones, arb_mat_hilbert, arb_mat_pascal, arb_mat_stirling, arb_mat_dct, acb_mat_ones, acb_mat_dft.
  - Added support for window matrices (arb/acb_mat_window_init/clear).
  - Changed random test matrix generation (arb/acb_mat_randtest) to produce sparse matrices with higher probability.
  - Added acb_mat_conjugate and acb_mat_conjugate_transpose.

- Arithmetic and elementary functions
  - Improved arb_sin_cos, arb_sin and arb_cos to produce more accurate enclosures for wide input intervals. The working precision is also reduced automatically based on the accuracy of the input to improve efficiency.
  - Improved arb_sinh_cosh, arb_sinh and arb_cosh to produce more accurate enclosures for wide input intervals. The working precision is also reduced automatically based on the accuracy of the input to improve efficiency.
  - Improved arb_exp_invexp and arb_expm1 to produce more accurate enclosures for wide input intervals. The working precision is also reduced automatically based on the accuracy of the input to improve efficiency.
  - Improved acb_rsqrt to produce more accurate enclosures for wide intervals.
  - Made mag_add_ui_lower public.
  - Added mag_sinh, mag_cosh, mag_sinh_lower, mag_cosh_lower.
  - Fixed minor precision loss near -1 in arb_log_hypot and acb_log.
  - Return imaginary numbers with exact zero real part when possible in acb_acos and acb_acosh (contributed by Ralf Stephan).
  - Improved special cases in arb_set_interval_arf (reported by Marc Mezzarobba).

- Special functions
  - Added a function for computing isolated generalized Stieltjes constants (acb_dirichlet_stieltjes).
  - Added scaled versions of Bessel functions (acb_hypgeom_bessel_i_scaled, acb_hypgeom_bessel_k_scaled).
  - The interface for the internal methods computing Bessel functions (i_asymp, k_asymp, etc.) has been changed to accommodate computing scaled versions.

  - Added Riemann xi function (acb_dirichlet_xi) (contributed by D.H.J Polymath).
  - Fixed infinite error bounds in the Riemann zeta function when evaluating at a ball containing zero centered in the left plane (contributed by D.H.J Polymath).
  - Fixed precision loss in Airy functions with huge input and high precision.
  - Legendre functions of the first kind (legendre_p): handle inexact integer a+b-c in 2F1 better (contributed by Joel Dahne).
- Example programs and documentation
  - Added more color functions to complex_plot.c.
  - Added more example integrals suggested by Nicolas Brisebarre and Bruno Salvy to integrals.c
  - Changed Sphinx style and redesigned the documentation front page.
  - Miscellaneous documentation cleanups.
  - Added documentation page about contributing.
- Other
  - Fixed a crash on some systems when calling acb_dft methods with a length of zero.
  - Fixed issue with setting rpath in configure (contributed by Vincent Delecroix).

13.1.9 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 recurrence 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" compilation 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 Mezzarobba).

  - Fixed Travis builds on Mac OS again (contributed by Isuru Fernando).
  - Added missing declaration for arb_bell_ui() (reported by numsys).

13.1.10 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).

13.1.11 2017-07-10 - version 2.11.1

- Avoid use of a function that was unavailable in the latest public FLINT release

13.1.12 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.

13.1.13 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 add acb_poly_exp_pi_i_series.
  - Added arb_poly_log1p_series, acb_poly_log1p_series.

13.1.14 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 fprint 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 arb/acb_poly_inv_series.
  - Added functions for fast polynomial Taylor shift (arb_poly_taylor_shift, acb_poly_taylor_shift and variants).
  - Fast handling of special cases in polynomial composition.
  - Added acb_poly scalar mul and div convenience methods (contributed by Alex Griffing).
  - Added set_trunc, set_trunc_round convenience methods.
  - Added add_series, sub_series methods for truncating addition.
  - Added polynomial is_zero, is_one, is_x, valuation convenience methods.
  - Added hack to arb_poly_mullow and acb_poly_mullow to avoid overhead when doing an in-place multiplication with length at most 2.
  - Added binomial and Borel transform methods for acb_poly.
- Matrices
  - Added Cholesky decomposition plus solving and inverse for positive definite matrices (arb_mat_cho, arb_mat_spd_solve, 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 (arb_mat_ldl, arb_mat_solve_ldl_precomp, 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 (arb_mat_exp_taylor_sum, acb_mat_exp_taylor_sum).
  - Added functions related to sparsity structure (arb_mat_entrywise_is_zero, arb_mat_count_is_zero, etc.) (contributed by Alex Griffing).
  - Entrywise multiplication (arb_mat_mul_entrywise, acb_mat_mul_entrywise) (contributed by Alex Griffing).
  - Added is_empty and is_square convenience methods (contributed by Alex Griffing).

  - Added the 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 (arb_bernoulli_poly_ui, acb_bernoulli_poly_ui).
  - Added convenience function for evaluation of huge Bernoulli numbers (arb_bernoulli_fmpz).
  - Added Euler numbers (arb_euler_number_ui, arb_euler_number_fmpz).
  - Added fast approximate partition function (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 arb_atan.
  - Added arb_log_base_ui.
  - Added complex sinc function (acb_sinc).
  - Special handling of z = 1 when computing polylogarithms.
  - Fixed 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 (contributed 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 representation (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.

13.1.15 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.

13.1.16 2015-12-29 - version 2.8.0

- Compatibility and build system
  - Windows64 support (contributed by Bill Hart).
  - Fixed a bug that broke basic arithmetic on targets where FLINT uses fallback code instead of assembly code, such as PPC64 (contributed by Jeroen Demeyer).
  - Fixed configure to use EXTRA_SHARED_FLAGS/LDFLAGS, and other build system fixes (contributed by Tommy Hofmann, Bill Hart).
  - Added soname versioning (contributed by Julien Puydt).
  - Fixed test code on MinGW (contributed by Hrvoje Abraham).
  - Miscellaneous fixes to simplify interfacing Arb from Julia.
- Arithmetic and elementary functions
  - Fixed arf_get_d to handle underflow/overflow correctly and to support round-to-nearest.
  - Added more complex inverse hyperbolic functions (acb_asin, acb_acos, acb_asinh, acb_acosh, acb_atanh).
  - Added arb_contains_int and acb_contains_int for testing whether an interval contains any integer.
  - Added acb_quadratic_roots_fmpz.
  - Improved arb_sinh to use a more accurate formula for $x < 0$.
  - Added sinc function (arb_sinc) (contributed by Alex Griffing).
  - Fixed bug in arb_exp affecting convergence for huge input.
  - Faster implementation of arb_div_2expm1_ui.

  - Added mag_root, mag_geom_series.
  - Improved and added test code for arb_add_error functions.
  - Changed arb_pow and acb_pow to make pow(0,positive) = 0 instead of nan.
  - Improved acb_sqrt to return finite output for finite input straddling the branch cut.
  - Improved arb_set_interval_arf so that [inf,inf] = inf instead of an infinite interval.
  - Added computation of Bell numbers (arb_bell_fmpz).
  - Added arb_power_sum_vec for computing power sums using Bernoulli numbers.
  - Added computation of the Fujiwara root bound for acb_poly.
  - Added code to identify all the real roots of a real polynomial (acb_poly_validate_real_roots).
  - Added several convenient assignment functions, including arb_set_d, acb_set_d, acb_set_d_d, acb_set_fmpz_fmpz (contributed by Ricky Farr).
  - Added many accessor functions (_arb/acb_vec_entry_ptr, arb_get_mid/rad_arb, acb_real/imag_ptr, arb_mid/rad_ptr, acb_get_real/imag).
  - Added missing functions acb_add_si, acb_sub_si.
  - Renamed arb_root to arb_root_ui (keeping alias) and added acb_root_ui.

- Special functions
  - Implemented the Gauss hypergeometric function 2F1 and its regularized version.
  - Fixed two bugs in acb_hypgeom_pfq_series_direct discovered while implementing 2F1. In rare cases, these could lead to incorrect values for functions depending on parameter derivatives of hypergeometric series.
    * The first bug involved incorrect handling of negative integer parameters. The bug only affected 2F1 and higher functions; it did not affect correctness of any previously implemented functions that relied on acb_hypgeom_pfq_series_direct (such as Bessel Y and K functions of integer order).
    * The second bug involved a too small bound being computed for the sum of a geometric series. The geometric series bound is nearly tight for 2F1, and the incorrect version caused immediate test failures for that function. Theoretically, this bug affected correctness of some previously-implemented functions that relied on acb_hypgeom_pfq_series_direct (such as Bessel Y and K functions of integer order), but since the geometric bound is not as tight in those cases, those functions were still reliable in practice (no failing test case has been found).
  - Implemented Airy functions and their derivatives (acb_hypgeom_airy).
  - Implemented the confluent hypergeometric function 0F1 (acb_hypgeom_0f1).
  - Implemented associated Legendre functions P and Q.
  - Implemented Chebyshev, Jacobi, Gegenbauer, Laguerre, Hermite functions.
  - Implemented spherical harmonics.
  - Added function for computing Bessel J and Y functions simultaneously.
  - Added rising factorials for non-integer n (arb_rising, acb_rising).
  - Made rising factorials use gamma function for large integer n.
  - Faster algorithm for theta constants and Dedekind eta function at very high precision.
  - Fixed erf to give finite values instead of +/-inf for big imaginary input.
  - Improved acb_zeta (and arb_zeta) to automatically use fast code for integer zeta values.
  - Added double factorial (arb_doublefac_ui).

  - Added code for generating Hilbert class polynomials (acb_modular_hilbert_class_poly).
- Matrices
  - Added faster matrix squaring (arb/acb_mat_sqr) (contributed by Alex Griffing).
  - Added matrix trace (arb/acb_mat_trace) (contributed by Alex Griffing).
  - Added arb/acb_mat_set_round_fmpz_mat, acb_mat_set(_round)_arb_mat (contributed by Tommy Hofmann).
  - Added arb/acb_mat_transpose (contributed by Tommy Hofmann).
  - Added comparison methods arb/acb_mat_eq/ne (contributed by Tommy Hofmann).
- Other
  - Added complex_plot example program.
  - Added Airy functions to real_roots example program.
  - Other minor patches were contributed by Alexander Kobel, Marc Mezzarobba, Julien Puydt.
  - Removed obsolete file config.h.

13.1.17 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.

13.1.18 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.

13.1.19 2015-01-28 - version 2.5.0

- String conversion
  - Added arb_set_str.
  - Added arb_get_str and arb_printn for pretty-printed rigorous decimal output.
  - Added helper functions for binary to decimal conversion.
- Core arithmetic
  - Improved speed of division when using GMP instead of MPIR.
  - Improved complex division with a small denominator.
  - Removed a little bit of overhead for complex squaring.
- Special functions
  - Faster code for atan at very high precision, used instead of mpfr_atan.
  - Optimized elementary functions slightly for small input.
  - Added modified error functions erfc and erfi.
  - Added the generalized exponential integral.
  - Added the upper incomplete gamma function.
  - Implemented the complete elliptic integral of the first kind.
  - Implemented the arithmetic-geometric mean of complex numbers.
  - Optimized arb_digamma for small integers.
  - Made mag_log_ui, mag_binpow_uiui and mag_polylog_tail proper functions.
  - Added pow, agm, erf, elliptic_k, elliptic_p as functions of complex power series.
  - Added incomplete gamma function of complex power series.
  - Improved code for bounding complex rising factorials (the old code could potentially have given wrong results in degenerate cases).
  - Added arb_sqrt1pm1, arb_atanh, arb_asinh, arb_atanh.
  - Added arb_log1p, acb_log1p, acb_atan.
  - Added arb_hurwitz_zeta.
  - Improved parameter selection in the Hurwitz zeta function to try to avoid stalling when given enormous input.
  - Optimized sqrt and rsqrt of power series when given a binomial as input.
  - Made arb_bernoulli_ui(2^64-2) not crash.
  - Fixed rgamma of negative integers returning indeterminate.
- Polynomials and matrices
  - Added characteristic polynomial computation for real and complex matrices.
  - Added polynomial set_round methods.
  - Added is_real methods for more types.
  - Added more get_unique_fmpz methods.
  - Added code for generating Swinnerton-Dyer polynomials.
  - Improved error bounding in det() and exp() of complex matrices to recognize when the result is real-valued.
  - Changed polynomial divrem to return success/fail instead of aborting on divide by zero.

- Miscellaneous
  - Added logo to documentation.
  - Made inlined functions build as part of the library.
  - Silenced a clang warning.
  - Made _acb_vec_sort_pretty a library function.

13.1.20 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.

13.1.21 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.

13.1.22 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.

13.1.23 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.

13.1.24 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).

13.1.25 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.

13.1.26 2013-12-21 - version 1.0.0

- New example programs directory
  - poly_roots example program.
  - real_roots example program.
  - pi_digits example program.
  - hilbert_matrix example program.
  - keiper_li example program.
- New fmprb_calc module for calculus with real functions
  - Bisection-based root isolation.
  - Asymptotically fast Newton root refinement.
- New fmpcb_calc module for calculus with complex functions
  - Numerical integration using Taylor series.
- Scalar functions
  - Simplified fmprb_const_euler using published error bound.
  - Added fmprb_inv.
  - Added fmprb_trim, fmpcb_trim.
  - Added fmpcb_rsqrt (complex reciprocal square root).
  - Fixed bug in fmprb_sqrtpos with nonfinite input.
  - Slightly improved fmprb powering code.

  - Added various functions for bounding fmprs by powers of two.
  - Added fmpr_is_int.
- Polynomials and power series
  - Implemented scaling to speed up blockwise multiplication.
  - Slightly faster basecase power series exponentials.
  - Improved sin/cos/tan/exp for short power series.
  - Added complex sqrt_series, rsqrt_series.
  - Implemented the Riemann-Siegel Z and theta functions for real power series.
  - Added fmprb_poly_pow_series, fmprb_poly_pow_ui and related methods.
  - Added fmprb/fmpcb_poly_contains_fmpz_poly.
  - Faster composition by monomials.
  - Implemented Borel transform and binomial transform for real power series.
- Matrices
  - Implemented matrix exponentials.
  - Multithreaded fmprb_mat_mul.
  - Added matrix infinity norm functions.
  - Added some more matrix-scalar functions.
  - Added matrix contains and overlaps methods.
- Zeta function evaluation
  - Multithreaded power sum evaluation.
  - Faster parameter selection when computing many derivatives.
  - Implemented binary splitting to speed up computing many derivatives.
- Miscellaneous
  - Corrections for C++ compatibility (contributed by Jonathan Bober).
  - Several minor bugfixes and test code enhancements.

13.1.27 2013-08-07 - version 0.7

- Floating-point and ball functions
  - Documented, added test code, and fixed bugs for various operations involving a ball containing an infinity or NaN.
  - Added reciprocal square root functions (fmpr_rsqrt, fmprb_rsqrt) based on mpfr_rec_sqrt.
  - Faster high-precision division by not computing an explicit remainder.
  - Slightly faster computation of pi by using new reciprocal square root and division code.
  - Added an fmpr function for approximate division to speed up certain radius operations.
  - Added fmpr_set_d for conversion from double.
  - Allow use of doubles to optionally compute the partition function faster but without an error bound.
  - Bypass mpfr overflow when computing the exponential function to extremely high precision (approximately 1 billion digits).

  - Made fmprb_exp faster for large numbers at extremely high precision by skipping the log(2) removal.
  - Made fmpcb_lgamma faster at high precision by speeding up the argument reduction branch computation.
  - Added fmprb_asin, fmprb_acos.
  - Added various other utility functions to the fmprb module.
  - Added a function for computing the Glaisher constant.
  - Optimized evaluation of the Riemann zeta function at high precision.
- Polynomials and power series
  - Made squaring of polynomials faster than generic multiplication.
  - Implemented power series reversion (various algorithms) for the fmprb_poly type.
  - Added many fmprb_poly utility functions (shifting, truncating, setting/getting coefficients, etc.).
  - Improved power series division when either operand is short
  - Improved power series logarithm when the input is short.
  - Improved power series exponential to use the basecase algorithm for short input regardless of the output size.
  - Added power series square root and reciprocal square root.
  - Added atan, tan, sin, cos, sin_cos, asin, acos fmprb_poly power series functions.
  - Added Newton iteration macros to simplify various functions.
  - Added gamma functions of real and complex power series ([fmprb/fmpcb]_poly_[gamma/rgamma/lgamma]_series).
  - Added wrappers for computing the Hurwitz zeta function of a power series ([fmprb/fmpcb]_poly_zeta_series).
  - Implemented sieving and other optimizations to improve performance for evaluating the zeta function of a short power series.
  - Improved power series composition when the inner series is linear.
  - Added many fmpcb_poly versions of nearly all fmprb_poly functions.
  - Improved speed and stability of series composition/reversion by balancing the power table exponents.
- Other
  - Added support for freeing all cached data by calling flint_cleanup().
  - Introduced fmprb_ptr, fmprb_srcptr, fmpcb_ptr, fmpcb_srcptr typedefs for cleaner function signatures.
  - Various bug fixes and general cleanup.

13.1.28 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.

13.1.29 2013-03-28 - version 0.5

- Arithmetic and elementary functions
  - Added fmpr_get_fmpz, fmpr_get_si.
  - Fixed accuracy problem with fmprb_div_2expm1.
  - Special-cased squaring of complex numbers.
  - Added various fmpcb convenience functions (addmul_ui, etc).
  - Optimized fmpr_cmp_2exp_si and fmpr_cmpabs_2exp_si, and added test code for comparison functions.
  - Added fmprb_atan2, also fixing a bug in fmpcb_arg.
  - Added fmprb_sin_pi, cos_pi, sin_cos_pi, etc.
  - Added fmprb_sin_pi_fmpq (etc.) using algebraic methods for fast evaluation of roots of unity.
  - Faster fmprb_poly_evaluate and evaluate_fmpcb using rectangular splitting.
  - Added fmprb_poly_evaluate2, evaluate2_fmpcb for simultaneously evaluating the derivative.
  - Added fmprb_poly root polishing code using near-optimal Newton steps (experimental).
  - Added fmpr_root, fmprb_root (currently based on MPFR).
  - Added fmpr_min, fmpr_max.
  - Added fmprb_set_interval_fmpr, fmprb_union.
  - Added fmpr_bits, fmprb_bits, fmpcb_bits for obtaining the mantissa width.
  - Added fmprb_hypot.
  - Added complex square roots.

  - Improved fmprb_log to slightly improve speed, and properly support huge arguments.
  - Fixed exp, cosh, sinh to work with huge arguments.
  - Added fmprb_expm1.
  - Fixed sin, cos, atan to work with huge arguments.
  - Improved fmprb_pow and fmpcb_pow, including automatic detection of small integer and half-integer exponents.
  - Added many more elementary functions: fmprb_tan/cot/tanh/coth, fmpcb_tan/cot, and pi versions.
  - Added fmprb const_e, const_log2, const_log10, const_catalan.
  - Fixed ball containment/overlap checking to work operate efficiently and correctly with huge exponents.
  - Strengthened test code for many core operations.
- Special functions
  - Reorganized zeta function related code.
  - Faster evaluation of the Riemann zeta function via sieving.
  - Documented and improved efficiency of the zeta constant binary splitting code.
  - Calculate error bound in Borwein's algorithm with fmprs instead of using doubles.
  - Optimized divisions in zeta evaluation via the Euler product.
  - Use functional equation for Riemann zeta function of a negative argument.
  - Compute single Bernoulli numbers using ball arithmetic instead of relying on the floating-point code in flint.
  - Initial code for evaluating the gamma function using its Taylor series.
  - Much faster rising factorials at high precision, using difference polynomials.
  - Much faster gamma function at high precision.
  - Added complex gamma function, log gamma function, and other versions.
  - Added fmprb_agm (real arithmetic-geometric mean).
  - Added fmprb_gamma_fmpq, supporting rapid computation of gamma(p/q) for q = 1,2,3,4,6.
  - Added real and complex digamma function.
  - Fixed unnecessary recomputation of Bernoulli numbers.
  - Optimized computation of Euler's constant, and added proper error bounds.
  - Avoid reliance on doubles in the hypergeometric series tail bound.
  - Cleaned up factorials and binomials, computing factorials via gamma.
- Other
  - Added an fmpz_extras module to collect various internal fmpz helper functions.
  - Fixed detection of flint header files.
  - Fixed various other small bugs.

13.1.30 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.

13.1.31 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.

13.1.32 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.

13.1.33 2012-09-14 - version 0.1

- 2012-08-05 - Began simplified rewrite.
- 2012-04-05 - Experimental ball and polynomial code (first commit).

BIBLIOGRAPHY

[Ari2011] J. Arias de Reyna, "High precision computation of Riemann's zeta function by the Riemann-Siegel formula, I", Mathematics of Computation 80 (2011), 995-1009

[Ari2012] J. Arias de Reyna, "Programs for Riemann's zeta function", (J. A. J. van Vonderen, Ed.) *Leven met getallen : liber amicorum ter gelegenheid van de pensionering van Herman te Riele* CWI (2012) 102-112, https://ir.cwi.nl/pub/19724

[Arn2010] J. Arndt, *Matters Computational*, Springer (2010), http://www.jjj.de/fxt/#fxtbook

[BBC1997] D. H. Bailey, J. M. Borwein and R. E. Crandall, "On the Khintchine constant", Mathematics of Computation 66 (1997) 417-431

[Blo2009] R. Bloemen, "Even faster zeta(2n) calculation!", https://web.archive.org/web/20141101133659/http://xn–2umb.com/09/11/even-faster-zeta-calculation

[BBC2000] J. Borwein, D. M. Bradley and R. E. Crandall, "Computational strategies for the Riemann zeta function", Journal of Computational and Applied Mathematics 121 (2000) 247-296

[BZ1992] J. Borwein and I. Zucker, "Fast evaluation of the gamma function for small rational fractions using complete elliptic integrals of the first kind", IMA Journal of Numerical Analysis 12 (1992) 519-526

[Bog2012] I. Bogaert, B. Michiels and J. Fostier, "O(1) computation of Legendre polynomials and Gauss-Legendre nodes and weights for parallel computing", SIAM Journal on Scientific Computing 34:3 (2012), C83-C101

[Bor1987] P. Borwein, "Reduced complexity evaluation of hypergeometric functions", Journal of Approximation Theory 50:3 (1987)

[Bor2000] P. Borwein, "An Efficient Algorithm for the Riemann Zeta Function", Constructive experimental and nonlinear analysis, CMS Conference Proc. 27 (2000) 29-34, http://www.cecm.sfu.ca/personal/pborwein/PAPERS/P155.pdf

[BM1980] R. P. Brent and E. M. McMillan, "Some new algorithms for high-precision computation of Euler's constant", Mathematics of Computation 34 (1980) 305-312.

[Bre1978] R. P. Brent, "A Fortran multiple-precision arithmetic package", ACM Transactions on Mathematical Software, 4(1):57–70, March 1978.

[Bre1979] R. P. Brent, "On the Zeros of the Riemann Zeta Function in the Critical Strip", Mathematics of Computation 33 (1979), 1361-1372, https://doi.org/10.1090/S0025-5718-1979-0537983-2

[Bre2010] R. P. Brent, "Ramanujan and Euler's Constant", http://wwwmaths.anu.edu.au/~brent/pd/Euler_CARMA_10.pdf

[BJ2013] R. P. Brent and F. Johansson, "A bound for the error term in the Brent-McMillan algorithm", preprint (2013), http://arxiv.org/abs/1312.0039

[BZ2011] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*, Cambridge University Press (2011), http://www.loria.fr/~zimmerma/mca/pub226.html

[Car1995] B. C. Carlson, "Numerical computation of real or complex elliptic integrals". Numerical Algorithms, 10(1):13-26 (1995).

[CP2005] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*, second edition, Springer (2005).

[CGHJK1996] R. M. Corless, G. H. Gonnet, D. E. Hare, D. J. Jeffrey and D. E. Knuth, "On the Lambert W function", Advances in Computational Mathematics, 5(1) (1996), 329-359

[Dup2006] R. Dupont. "Moyenne arithmético-géométrique, suites de Borchardt et applications." These de doctorat, École polytechnique, Palaiseau (2006). http://http://www.lix.polytechnique.fr/Labo/Regis.Dupont/these_soutenance.pdf

[DYF1999] A. Dzieciol, S. Yngve and P. O. Fröman, "Coulomb wave functions with complex values of the variable and the parameters", J. Math. Phys. 40, 6145 (1999), https://doi.org/10.1063/1.533083

[EHJ2016] A. Enge, W. Hart and F. Johansson, "Short addition sequences for theta functions", preprint (2016), https://arxiv.org/abs/1608.06810

[EM2004] O. Espinosa and V. Moll, "A generalized polygamma function", Integral Transforms and Special Functions (2004), 101-115.

[Fil1992] S. Fillebrown, "Faster Computation of Bernoulli Numbers", Journal of Algorithms 13 (1992) 431-445

[Gas2018] D. Gaspard, "Connection formulas between Coulomb wave functions" (2018), https://arxiv.org/abs/1804.10976

[GG2003] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, second edition, Cambridge University Press (2003)

[GVL1996] G. H. Golub and C. F. Van Loan, *Matrix Computations*, third edition, Johns Hopkins University Press (1996).

[GS2003] X. Gourdon and P. Sebah, "Numerical evaluation of the Riemann Zeta-function" (2003), http://numbers.computation.free.fr/Constants/Miscellaneous/zetaevaluations.pdf

[HS1967] E. Hansen and R. Smith, "Interval Arithmetic in Matrix Computations, Part II", SIAM Journal of Numerical Analysis, 4(1):1-9 (1967). https://doi.org/10.1137/0704001

[HZ2004] G. Hanrot and P. Zimmermann, "Newton Iteration Revisited" (2004), http://www.loria.fr/~zimmerma/papers/fastnewton.ps.gz

[Hoe2009] J. van der Hoeven, "Ball arithmetic", Technical Report, HAL 00432152 (2009), http://www.texmacs.org/joris/ball/ball-abs.html

[Hoe2001] J. van der Hoeven. "Fast evaluation of holonomic functions near and in regular singularities", Journal of Symbolic Computation, 31(6):717-743 (2001).

[HM2017] J. van der Hoeven and B. Mourrain. "Efficient certification of numeric solutions to eigenproblems", MACIS 2017, 81-94, (2017), https://hal.archives-ouvertes.fr/hal-01579079

[JB2018] F. Johansson and I. Blagouchine. "Computing Stieltjes constants using complex integration", preprint (2018), https://arxiv.org/abs/1804.01679

[Joh2012] F. Johansson, "Efficient implementation of the Hardy-Ramanujan-Rademacher formula", LMS Journal of Computation and Mathematics, Volume 15 (2012), 341-359, http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=8710297

[Joh2013] F. Johansson, "Rigorous high-precision computation of the Hurwitz zeta function and its derivatives", Numerical Algorithms, http://arxiv.org/abs/1309.2877 http://dx.doi.org/10.1007/s11075-014-9893-1

[Joh2014a] F. Johansson, *Fast and rigorous computation of special functions to high precision*, PhD thesis, RISC, Johannes Kepler University, Linz, 2014. http://fredrikj.net/thesis/

[Joh2014b] F. Johansson, "Evaluating parametric holonomic sequences using rectangular splitting", ISSAC 2014, 256-263. http://dx.doi.org/10.1145/2608628.2608629

[Joh2014c] F. Johansson, "Efficient implementation of elementary functions in the medium-precision range", http://arxiv.org/abs/1410.7176

[Joh2015] F. Johansson, "Computing Bell numbers", http://fredrikj.net/blog/2015/08/computing-bell-numbers/

[Joh2016] F. Johansson, "Computing hypergeometric functions rigorously", preprint (2016), https://arxiv.org/abs/1606.06977

[Joh2017a] F. Johansson. "Arb: efficient arbitrary-precision midpoint-radius interval arithmetic", IEEE Transactions on Computers, 66(8):1281-1292 (2017). https://doi.org/10.1109/TC.2017.2690633

[Joh2017b] F. Johansson, "Computing the Lambert W function in arbitrary-precision complex interval arithmetic", preprint (2017), https://arxiv.org/abs/1705.03266

[Joh2018a] F. Johansson, "Numerical integration in arbitrary-precision ball arithmetic", preprint (2018), https://arxiv.org/abs/1802.07942

[Joh2018b] F. Johansson and others, "mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.1.0)", December 2018. http://mpmath.org/

[JM2018] F. Johansson and M. Mezzarobba, "Fast and rigorous arbitrary-precision computation of Gauss-Legendre quadrature nodes and weights", preprint (2018), https://arxiv.org/abs/1802.03948

[Kar1998] E. A. Karatsuba, "Fast evaluation of the Hurwitz zeta function and Dirichlet L-series", Problems of Information Transmission 34:4 (1998), 342-353, http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=ppi&paperid=425&option_lang=eng

[Kob2010] A. Kobel, "Certified Complex Numerical Root Finding", Seminar on Computational Geometry and Geometric Computing (2010), http://www.mpi-inf.mpg.de/departments/d1/teaching/ss10/Seminar_CGGC/Slides/02_Kobel_NRS.pdf

[Kri2013] A. Krishnamoorthy and D. Menon, "Matrix Inversion Using Cholesky Decomposition" Proc. of the International Conference on Signal Processing Algorithms, Architectures, Arrangements, and Applications (SPA-2013), pp. 70-72, 2013.

[Leh1970] R. S. Lehman, "On the Distribution of Zeros of the Riemann Zeta-Function", Proc. of the London Mathematical Society 20(3) (1970), 303-320, https://doi.org/10.1112/plms/s3-20.2.303

[Mic2007] N. Michel, "Precise Coulomb wave functions for a wide range of complex l, eta and z", Computer Physics Communications, Volume 176, Issue 3, (2007), 232-249, https://doi.org/10.1016/j.cpc.2006.10.004

[Miy2010] S. Miyajima, "Fast enclosure for all eigenvalues in generalized eigenvalue problems", Journal of Computational and Applied Mathematics, 233 (2010), 2994-3004, https://dx.doi.org/10.1016/j.cam.2009.11.048

[MPFR2012] The MPFR team, "MPFR Algorithms" (2012), http://www.mpfr.org/algo.html

[NIST2012] National Institute of Standards and Technology, *Digital Library of Mathematical Functions* (2012), http://dlmf.nist.gov/

[Olv1997] F. Olver, *Asymptotics and special functions*, AKP Classics, AK Peters Ltd., Wellesley, MA, 1997. Reprint of the 1974 original.

[Rad1973] H. Rademacher, *Topics in analytic number theory*, Springer, 1973.

[Pet1999] K. Petras, "On the computation of the Gauss-Legendre quadrature formula with a given precision", Journal of Computational and Applied Mathematics 112 (1999), 253-267

[Pla2011] D. J. Platt, "Computing degree 1 L-functions rigorously", Ph.D. Thesis, University of Bristol (2011), https://people.maths.bris.ac.uk/~madjp/thesis5.pdf

[Pla2017] D. J. Platt, "Isolating some non-trivial zeros of zeta", Mathematics of Computation 86 (2017), 2449-2467, https://doi.org/10.1090/mcom/3198

[PP2010] K. H. Pilehrood and T. H. Pilehrood. "Series acceleration formulas for beta values", Discrete Mathematics and Theoretical Computer Science, DMTCS, 12 (2) (2010), 223-236, https://hal.inria.fr/hal-00990465/

[PS1973] M. S. Paterson and L. J. Stockmeyer, "On the number of nonscalar multiplications necessary to evaluate polynomials", SIAM J. Comput (1973)

[PS1991] G. Pittaluga and L. Sacripante, "Inequalities for the zeros of the Airy functions", SIAM J. Math. Anal. 22:1 (1991), 260-267.

[Rum2010] S. M. Rump, "Verification methods: Rigorous results using floating-point arithmetic", Acta Numerica 19 (2010), 287-449.

[Smi2001] D. M. Smith, "Algorithm: Fortran 90 Software for Floating-Point Multiple Precision Arithmetic, Gamma and Related Functions", Transactions on Mathematical Software 27 (2001) 377-387, http://myweb.lmu.edu/dmsmith/toms2001.pdf

[Tak2000] D. Takahashi, "A fast algorithm for computing large Fibonacci numbers", Information Processing Letters 75 (2000) 243-246, http://www.ii.uni.wroc.pl/~lorys/IPL/article75-6-1.pdf

[Tre2008] L. N. Trefethen, "Is Gauss Quadrature Better than Clenshaw-Curtis?", SIAM Review, 50:1 (2008), 67-87, https://doi.org/10.1137/060659831

[Tru2011] T. S. Trudgian, "Improvements to Turing's method", Mathematics of Computation 80 (2011), 2259-2279, https://doi.org/10.1090/S0025-5718-2011-02470-1

[Tru2014] T. S. Trudgian, "An improved upper bound for the argument of the Riemann zeta-function on the critical line II", Journal of Number Theory 134 (2014), 280-292, https://doi.org/10.1016/j.jnt.2013.07.017

[Tur1953] A. M. Turing, "Some Calculations of the Riemann Zeta-Function", Proc. of the London Mathematical Society 3(3) (1953), 99-117, https://doi.org/10.1112/plms/s3-3.1.99

[WQ3a] http://functions.wolfram.com/07.11.26.0033.01

[WQ3b] http://functions.wolfram.com/07.12.27.0014.01

[WQ3c] http://functions.wolfram.com/07.12.26.0003.01

[WQ3d] http://functions.wolfram.com/07.12.26.0088.01

INDEX

Symbols

A

B

D

F

H

M

P

S

U