4 Real and complex numbers

4.1 arb.h — real numbers
4.1.1 Types, macros and constants
4.1.2 Memory management
4.1.3 Assignment and rounding
4.1.4 Assignment of special values
4.1.5 Input and output
4.1.6 Random number generation
4.1.7 Radius and interval operations
4.1.8 Comparisons
4.1.9 Arithmetic
4.1.10 Powers and roots
4.1.11 Exponentials and logarithms
4.1.12 Trigonometric functions
4.1.13 Inverse trigonometric functions
4.1.14 Hyperbolic functions
4.1.15 Inverse hyperbolic functions
4.1.16 Constants
4.1.17 Gamma function and factorials
4.1.18 Zeta function
4.1.19 Bernoulli numbers and factorials
4.1.20 Polylogarithms
4.1.21 Other special functions
4.1.22 Internals for computing elementary functions
4.1.23 Vector functions

4.2 acb.h — complex numbers
4.2.1 Types, macros and constants
4.2.2 Memory management
4.2.3 Basic manipulation
4.2.4 Input and output
4.2.5 Random number generation
4.2.6 Precision and comparisons
4.2.7 Complex parts
4.2.8 Arithmetic
4.2.9 Mathematical constants
### 5 Polynomials and power series

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1</td>
<td>Types, macros and constants</td>
<td>73</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Memory management</td>
<td>73</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Basic manipulation</td>
<td>74</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Conversions</td>
<td>75</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Input and output</td>
<td>75</td>
</tr>
<tr>
<td>5.1.6</td>
<td>Random generation</td>
<td>75</td>
</tr>
<tr>
<td>5.1.7</td>
<td>Comparisons</td>
<td>75</td>
</tr>
<tr>
<td>5.1.8</td>
<td>Bounds</td>
<td>75</td>
</tr>
<tr>
<td>5.1.9</td>
<td>Arithmetic</td>
<td>76</td>
</tr>
<tr>
<td>5.1.10</td>
<td>Composition</td>
<td>78</td>
</tr>
<tr>
<td>5.1.11</td>
<td>Evaluation</td>
<td>79</td>
</tr>
<tr>
<td>5.1.12</td>
<td>Product trees</td>
<td>80</td>
</tr>
<tr>
<td>5.1.13</td>
<td>Multipoint evaluation</td>
<td>80</td>
</tr>
<tr>
<td>5.1.14</td>
<td>Interpolation</td>
<td>81</td>
</tr>
<tr>
<td>5.1.15</td>
<td>Differentiation</td>
<td>81</td>
</tr>
<tr>
<td>5.1.16</td>
<td>Transforms</td>
<td>81</td>
</tr>
<tr>
<td>5.1.17</td>
<td>Powers and elementary functions</td>
<td>82</td>
</tr>
<tr>
<td>5.1.18</td>
<td>Gamma function and factorials</td>
<td>85</td>
</tr>
<tr>
<td>5.1.19</td>
<td>Zeta function</td>
<td>86</td>
</tr>
<tr>
<td>5.1.20</td>
<td>Root-finding</td>
<td>87</td>
</tr>
<tr>
<td>5.1.21</td>
<td>Other special polynomials</td>
<td>87</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Types, macros and constants</td>
<td>88</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Memory management</td>
<td>88</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Basic properties and manipulation</td>
<td>88</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Input and output</td>
<td>89</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Random generation</td>
<td>90</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Comparisons</td>
<td>90</td>
</tr>
<tr>
<td>5.2.7</td>
<td>Conversions</td>
<td>90</td>
</tr>
<tr>
<td>5.2.8</td>
<td>Bounds</td>
<td>90</td>
</tr>
<tr>
<td>5.2.9</td>
<td>Arithmetic</td>
<td>91</td>
</tr>
<tr>
<td>5.2.10</td>
<td>Composition</td>
<td>92</td>
</tr>
<tr>
<td>5.2.11</td>
<td>Evaluation</td>
<td>94</td>
</tr>
<tr>
<td>5.2.12</td>
<td>Product trees</td>
<td>94</td>
</tr>
<tr>
<td>5.2.13</td>
<td>Multipoint evaluation</td>
<td>95</td>
</tr>
<tr>
<td>5.2.14</td>
<td>Interpolation</td>
<td>95</td>
</tr>
<tr>
<td>5.2.15</td>
<td>Differentiation</td>
<td>96</td>
</tr>
<tr>
<td>5.2.16</td>
<td>Transforms</td>
<td>96</td>
</tr>
<tr>
<td>5.2.17</td>
<td>Elementary functions</td>
<td>96</td>
</tr>
<tr>
<td>5.2.18</td>
<td>Gamma function</td>
<td>99</td>
</tr>
<tr>
<td>5.2.19</td>
<td>Power sums</td>
<td>100</td>
</tr>
<tr>
<td>5.2.20</td>
<td>Zeta function</td>
<td>100</td>
</tr>
</tbody>
</table>
# Matrices

## 6 Matrices

6.1 **arb_mat.h** – matrices over the real numbers
- 6.1.1 Types, macros and constants
- 6.1.2 Memory management
- 6.1.3 Conversions
- 6.1.4 Random generation
- 6.1.5 Input and output
- 6.1.6 Comparisons
- 6.1.7 Special matrices
- 6.1.8 Transpose
- 6.1.9 Norms
- 6.1.10 Arithmetic
- 6.1.11 Scalar arithmetic
- 6.1.12 Gaussian elimination and solving
- 6.1.13 Cholesky decomposition and solving
- 6.1.14 Characteristic polynomial
- 6.1.15 Special functions
- 6.1.16 Sparsity structure

6.2 **acb_mat.h** – matrices over the complex numbers
- 6.2.1 Types, macros and constants
- 6.2.2 Memory management
- 6.2.3 Conversions
- 6.2.4 Random generation
- 6.2.5 Input and output
- 6.2.6 Comparisons
- 6.2.7 Special matrices
- 6.2.8 Transpose
- 6.2.9 Norms
- 6.2.10 Arithmetic
- 6.2.11 Scalar arithmetic
- 6.2.12 Gaussian elimination and solving
- 6.2.13 Characteristic polynomial
- 6.2.14 Special functions

# Higher mathematical functions

## 7 Higher mathematical functions

7.1 **acb_hypgeom.h** – hypergeometric functions of complex variables
- 7.1.1 Convergent series
- 7.1.2 Asymptotic series
- 7.1.3 Generalized hypergeometric function
- 7.1.4 Confluent hypergeometric functions
- 7.1.5 Error functions and Fresnel integrals
- 7.1.6 Bessel functions
- 7.1.7 Airy functions
- 7.1.8 Incomplete gamma and beta functions
- 7.1.9 Exponential and trigonometric integrals
- 7.1.10 Gauss hypergeometric function
- 7.1.11 Orthogonal polynomials and functions

7.2 **arb_hypgeom.h** – hypergeometric functions of real variables
- 7.2.1 Generalized hypergeometric function
- 7.2.2 Confluent hypergeometric functions
- 7.2.3 Gauss hypergeometric function
- 7.2.4 Error functions and Fresnel integrals
- 7.2.5 Incomplete gamma and beta functions
- 7.2.6 Exponential and trigonometric integrals
- 7.2.7 Bessel functions
7.2.8 Airy functions ......................................................... 133
7.2.9 Orthogonal polynomials and functions ......................... 134
7.3 **acb_modular.h** – modular forms of complex variables ........ 134
  7.3.1 The modular group ............................................. 134
  7.3.2 Modular transformations ..................................... 135
  7.3.3 Addition sequences ........................................... 136
  7.3.4 Jacobi theta functions ..................................... 136
  7.3.5 The Dedekind eta function .................................. 139
  7.3.6 Modular forms ................................................. 139
  7.3.7 Elliptic functions .......................................... 140
  7.3.8 Elliptic integrals .......................................... 140
  7.3.9 Class polynomials ........................................... 141
7.4 **dirichlet.h** – Dirichlet characters ................................. 141
  7.4.1 Dirichlet characters ......................................... 141
  7.4.2 Multiplicative group modulo \( q \) ............................ 144
  7.4.3 Character type ................................................. 142
  7.4.4 Character properties ......................................... 143
  7.4.5 Character evaluation ....................................... 144
  7.4.6 Character operations ....................................... 144
7.5 **acb_dirichlet.h** – Dirichlet L-functions, zeta functions, and related functions ......................... 144
  7.5.1 Roots of unity .................................................. 145
  7.5.2 Truncated L-series and power sums ........................... 145
  7.5.3 Riemann zeta function and Riemann-Siegel formula ....... 146
  7.5.4 Hurwitz zeta function ....................................... 146
  7.5.5 Character evaluation ....................................... 147
  7.5.6 Gauss and Jacobi sums ..................................... 147
  7.5.7 Theta sums .................................................... 148
  7.5.8 L-functions ................................................... 149
  7.5.9 Hardy Z-functions ........................................... 150
7.6 **beroulli.h** – support for Bernoulli numbers ...................... 151
  7.6.1 Generation of Bernoulli numbers ............................. 151
  7.6.2 Caching .......................................................... 151
  7.6.3 Bounding ........................................................ 152
7.7 **hypergeom.h** – support for hypergeometric series ............. 152
  7.7.1 Strategy for error bounding .................................. 152
  7.7.2 Types, macros and constants ................................ 153
  7.7.3 Memory management ......................................... 153
  7.7.4 Error bounding ............................................... 154
  7.7.5 Summation ...................................................... 154
7.8 **partitions.h** – computation of the partition function .......... 154

8 Calculus ................................................................. 157
  8.1 **arb_calc.h** – calculus with real-valued functions .......... 157
    8.1.1 Types, macros and constants ............................... 157
    8.1.2 Debugging .................................................... 158
    8.1.3 Subdivision-based root finding ............................. 158
    8.1.4 Newton-based root finding ................................ 159
  8.2 **acb_calc.h** – calculus with complex-valued functions ..... 160
    8.2.1 Types, macros and constants ............................... 160
    8.2.2 Bounds ........................................................ 160
    8.2.3 Integration .................................................. 160

9 Extra utility modules ................................................. 163
  9.1 **fmpz_extras.h** – extra methods for FLINT integers ........... 163
    9.1.1 Memory-related methods .................................... 163
    9.1.2 Convenience methods ....................................... 163
    9.1.3 Inlined arithmetic ......................................... 163
    9.1.4 Low-level conversions ..................................... 164
## 9.2 bool_mat.h – matrices over booleans
- 9.2.1 Types, macros and constants ........................................... 164
- 9.2.2 Memory management ......................................................... 165
- 9.2.3 Conversions ................................................................. 165
- 9.2.4 Input and output ............................................................ 165
- 9.2.5 Value comparisons .......................................................... 165
- 9.2.6 Random generation .......................................................... 166
- 9.2.7 Special matrices .............................................................. 166
- 9.2.8 Transpose ................................................................. 166
- 9.2.9 Arithmetic ................................................................. 166
- 9.2.10 Special functions ........................................................... 166

## 9.3 dlog.h – discrete logarithms mod along primes
- 9.3.1 Types, macros and constants ........................................... 167
- 9.3.2 Single evaluation ............................................................ 168
- 9.3.3 Precomputations ............................................................. 168
- 9.3.4 Vector evaluations ........................................................... 168
- 9.3.5 Internal discrete logarithm strategies .................................. 169

## 9.4 fmp.r.h – Arb 1.x floating-point numbers (deprecated)
- 9.4.1 Types, macros and constants ........................................... 171
- 9.4.2 Memory management ........................................................ 172
- 9.4.3 Special values ................................................................. 172
- 9.4.4 Assignment, rounding and conversions .................................. 173
- 9.4.5 Comparisons ................................................................. 174
- 9.4.6 Random number generation ............................................. 175
- 9.4.7 Input and output ............................................................ 175
- 9.4.8 Arithmetic ................................................................. 176
- 9.4.9 Special functions ........................................................... 177

## 10 Supplementary algorithm notes

### 10.1 General formulas and bounds
- 10.1.1 Error propagation .......................................................... 179
- 10.1.2 Sums and series ............................................................. 180
- 10.1.3 Complex analytic functions ........................................... 180
- 10.1.4 Euler-Maclaurin formula ............................................... 181

### 10.2 Algorithms for mathematical constants
- 10.2.1 Pi ................................................................. 181
- 10.2.2 Logarithms of integers .................................................. 182
- 10.2.3 Euler’s constant ............................................................ 182
- 10.2.4 Catalan’s constant ........................................................ 182
- 10.2.5 Khinchin’s constant ....................................................... 182
- 10.2.6 Glaisher’s constant ......................................................... 183
- 10.2.7 Apery’s constant .......................................................... 183

### 10.3 Algorithms for the gamma function
- 10.3.1 The Stirling series ......................................................... 183
- 10.3.2 Rational arguments ....................................................... 183

### 10.4 Algorithms for the Hurwitz zeta function
- 10.4.1 Euler-Maclaurin summation ........................................... 184
- 10.4.2 Parameter Taylor series ................................................ 184

### 10.5 Algorithms for polylogarithms
- 10.5.1 Computation for small $z$ ............................................... 185
- 10.5.2 Expansion for general $z$ ................................................ 185

### 10.6 Algorithms for hypergeometric functions
- 10.6.1 Convergent series ........................................................ 186
- 10.6.2 Convergent series of power series ................................... 186
- 10.6.3 Asymptotic series for the confluent hypergeometric function .... 187
- 10.6.4 Asymptotic series for Airy functions ................................... 188
- 10.6.5 Corner case of the Gauss hypergeometric function ............... 189

### 10.7 Algorithms for the arithmetic-geometric mean

---

---
INTRODUCTION

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

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

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

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

This documentation is available in HTML format at http://arblib.org and in PDF format at http://arblib.org/arb.pdf. The version of the documentation you are currently reading was updated Dec 02, 2016 and describes Arb 2.9.0. Documentation for specific release versions is also available in PDF format.
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 \texttt{(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 \texttt{(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 \texttt{(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 \texttt{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 packages may not be up to date.

• Debian / Ubuntu / Linux Mint
  - https://packages.debian.org/source/sid/flint-arb

• Fedora
  - https://admin.fedoraproject.org/pkgdb/package/rpms/arb/

• Arch Linux
  - https://www.archlinux.org/packages/community/x86_64/arb/
  - https://www.archlinux.org/packages/community/i686/arb/

• Guix
  - https://www.gnu.org/software/guix/packages/

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 generally safe to use (the test suite should pass at all times), and recommended for keeping up with the latest improvements and bug fixes.

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 \texttt{--enable-gmpcompat} option.

• MPFR (http://www.mpfr.org) 3.0.0 or later.
FLINT (http://www.flintlib.org) version 2.4 or later. You may also use a git checkout of https://github.com/fredrik-johansson/flint2

2.2.4 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/MPRI, 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.

2.2.5 Installation as part of FLINT (deprecated)

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

With some versions of FLINT, Arb can be compiled as a FLINT extension package.

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

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

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

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
make . # configure
make --build . --config Release # build
make --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_TESTS=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:

```c
#include "arb.h"

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

Compile it with:

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

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

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

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

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

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

```
[3.1415926535897932384626433832795028841971693999999999999999751 +/- 6.28e-50]
Computed with arb-2.4.0
```

2.3 Using ball arithmetic

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

2.3.1 Ball semantics

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

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

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

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

General subsets of \( \mathbb{R} \) are not possible to represent on a computer. Instead, we work with subsets of the form \([m \pm r] = [m - r, m + r]\) where the midpoint \( m \) and radius \( r \) are binary floating-point numbers, i.e. numbers of the form \( u2^v \) with \( u, v \in \mathbb{Z} \) (to make this scheme complete, we also need to adjoin the special floating-point values \(-\infty, +\infty\) and NaN).
Given a ball \([m \pm r]\) with \(m \in \mathbb{R}\) (not necessarily a floating-point number), we can always round \(m\) to a nearby floating-point number that has at most most \(\text{prec}\) bits in the component \(u\), and add an upper bound for the rounding error to \(r\). In Arb, ball functions that take a \(\text{prec}\) argument as input (e.g. \texttt{arb_add()} ) always round their output to \(\text{prec}\) bits. Some functions are always exact (e.g. \texttt{arb_neg()}), and thus do not take a \(\text{prec}\) argument.

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

```c
#include "arb.h"

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

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

This snippet computes a 53-bit enclosure of \(\pi\) and prints it in three ways:

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

The output is:

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

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

A potential mistake is to create a ball from a \texttt{double} constant such as \(2.3\), when this actually represents \(2.299999999999982236431605997495353221893310546875\). To produce a ball containing the rational number \(23/10\), one of the following can be used:

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

### 2.3. Using ball arithmetic
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 \(\pi\)), then an error of order \(2^n\) can typically be overcome by working with \(n\) extra bits of precision, increasing the computation time by an amount that is polynomial in \(n\). In certain situations, however, overestimation leads to exponential slowdown or even failure of an algorithm to converge. For example, root-finding algorithms that refine the result iteratively may fail to converge in ball arithmetic, even if they do converge in plain floating-point arithmetic.

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

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:

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

Instead, the following can be used:

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

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

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

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

```c
if (arb_is_zero(x))
{
    /* do things assuming that x = 0 */
} #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 $|t| \leq 2^{-prec} < 1$. 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 $\sin(x)$.

```c
#include "arb.h"

void arb_sin_naive(arb_t res, const arb_t x, slong prec)
{
    arb_t s, t, u, tol;
    slong k;
    arb_init(s); arb_init(t); arb_init(u); arb_init(tol);
    arb_one(tol);
    arb_mul_2exp_si(tol, tol, -prec); /* tol = 2^-prec */
    for (k = 0; ; k++)
    {
        arb_pow_ui(t, x, 2 * k + 1, prec);
```
This algorithm is naive, because the Taylor series is slow to converge and suffers from catastrophic cancellation when $|x|$ is large (we could also improve the efficiency of the code slightly by computing the terms using recurrence relations instead of computing $x^k$ and $k!$ from scratch each iteration).

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

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

This example illustrates how ball arithmetic can be used to perform nontrivial calculations. To evaluate an infinite series, the user needs to know how to bound the tail of the series, but everything else is automatic. When evaluating a finite formula that can be expressed completely using built-in functions, all error bounding is automatic from the point of view of the user. In particular, the \texttt{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 \textit{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 \textit{Monte Carlo algorithm}. Using ball arithmetic to get a verified result effectively turns it into the analog of a \textit{Las Vegas algorithm}, which is a randomized algorithm that always gives a correct result if it terminates, but may fail to terminate (alternatively, instead of actually looping forever, it might signal failure after a certain number of iterations).

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

<table>
<thead>
<tr>
<th>Using</th>
<th>bits</th>
<th>( \sin(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using</td>
<td>64</td>
<td>([+/- 3.96e-18])</td>
</tr>
<tr>
<td>Using</td>
<td>128</td>
<td>([+/- 2.17e-37])</td>
</tr>
<tr>
<td>Using</td>
<td>256</td>
<td>([+/- 6.10e-76])</td>
</tr>
<tr>
<td>Using</td>
<td>512</td>
<td>([+/- 5.13e-153])</td>
</tr>
<tr>
<td>Using</td>
<td>1024</td>
<td>([+/- 4.01e-307])</td>
</tr>
<tr>
<td>Using</td>
<td>2048</td>
<td>([+/- 2.13e-615])</td>
</tr>
<tr>
<td>Using</td>
<td>4096</td>
<td>([+/- 6.85e-1232])</td>
</tr>
<tr>
<td>Using</td>
<td>8192</td>
<td>([+/- 4.66e-2465])</td>
</tr>
<tr>
<td>Using</td>
<td>16384</td>
<td>([+/- 5.09e-4931])</td>
</tr>
<tr>
<td>Using</td>
<td>32768</td>
<td>([+/- 5.41e-9863])</td>
</tr>
</tbody>
</table>

The sign of a nonzero real number can be decided by computing it to sufficiently high accuracy, but the sign of an expression that is exactly equal to zero cannot be decided by a numerical computation unless the entire computation happens to be exact (in this example, we could use the \texttt{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 \texttt{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 \texttt{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

\[2^{O(1) - \texttt{prec}}\]

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 \(|f(m) - f(m + \epsilon)|, |\epsilon| \leq r\). The working precision is automatically increased internally so that \(f(m)\) is computed to \texttt{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 \textit{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 \texttt{prec} bits. The sequence of operations might depend on \texttt{prec}; for example, an infinite series might be truncated so that the remainder is smaller than \(2^{-\texttt{prec}}\). The final result can be far from tight, and it is not guaranteed that the error converges to zero as \(\texttt{prec} \to \infty\), though in practice, it should do so in most cases.

In short, the \textit{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 \texttt{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 \(x = 2^2n\) 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 of the need to subtract an accurate multiple of \(2\pi\) 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 \( \text{prec} \). When the limit is exceeded, the output is set to a crude bound. For example, if \( x \) is too large, \( \text{arb\_sin()} \) will simply return \([\pm 1]\), and \( \text{arb\_exp()} \) will simply return \([\pm \infty] \) if \( x \) is positive or \([\pm 2^{-m}] \) if \( x \) is negative.

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

\[
\log(x) + \sin(x) \exp(-x).
\]

For example, the reflection formula of the digamma function has a similar structure. When \( x \) is large, the right term would be expensive to compute to high relative accuracy. Doing so is unnecessary, however, since a crude bound of \([\pm 1]\cdot [\pm 2^{-m}] \) 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 \( \alpha \) is about 0.3. Input with large \( |t| \) is most likely to be given deliberately by users with the explicit intent of evaluating the zeta function itself, so the evaluation is not cut off automatically.

## 2.4 Technical conventions and potential issues

### 2.4.1 Integer types

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

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

**slong**

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

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

**ulong**

The \( \text{ulong} \) 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 \( \text{ulong} \) parameter is denoted by \( ui \), for example \( \text{arb\_add\_ui()} \).

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

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

**mp\_limb\_t**

A single limb.
mp_ptr
   Pointer to a writable array of limbs.

mp_srcptr
   Pointer to a read-only array of limbs.

mp_size_t
   A limb count (always nonnegative).

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

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

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

fmpq_t
   FLINT multi-precision rational number.

fmpz_poly_t

fmpq_poly_t

fmpz_mat_t

fmpq_mat_t
   FLINT polynomials and matrices with integer and rational coefficients.

2.4.2 Integer overflow

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

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

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

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

2.4.3 Aliasing

As a rule, Arb allows aliasing of operands. For example, in the function call arb_add(z, x, y, prec), which performs $z \leftarrow x + y$, any two (or all three) of the variables $x$, $y$ and $z$ are allowed to be the same. Exceptions to this rule are documented explicitly.
The general rule that input and output variables can be aliased with each other only applies to variables of the same type (ignoring const qualifiers on input variables — a special case is that \texttt{arb_srcptr} is considered the const version of \texttt{arb_ptr}). This is a natural extension of the so-called strict aliasing rule in C.

For example, in \texttt{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 \_\texttt{arb_poly_evaluate()}\_: for the purposes of aliasing, \texttt{arb_srcptr} (the type of the coefficient array within \( f \)) and \texttt{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 \texttt{arb_ptr/arb_srcptr} variable pointing to a length 1 array would otherwise be interchangeable with an \texttt{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 \texttt{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 \texttt{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, \texttt{arb_add(z, x, x, prec)} and \texttt{arb_sub(z, x, x, prec)} currently do not implement this optimization. It is better to use \texttt{arb_mul_2exp_si(z, x, 1)} and \texttt{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 \texttt{flint_set_num_threads()} to set the number of threads that Arb is allowed to use internally for single computations (this is currently only exploited by a handful of operations). Please note that thread safety is only tested minimally, and extra caution when developing multithreaded code is therefore recommended.

Arb may cache some data (such as the value of \( \pi \) and Bernoulli numbers) to speed up various computations. In threadsafe mode, caches use thread-local storage. There is currently no way to save memory and avoid recomputation by having several threads share the same cache. Caches can be freed by calling the \texttt{flint_cleanup()} function. To avoid memory leaks, the user should call \texttt{flint_cleanup()} when exiting a thread. It is also recommended to call \texttt{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 \texttt{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 \texttt{flint_cleanup()}.

### 2.4.5 Use of hardware floating-point arithmetic

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

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

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

### 2.4.6 Interface changes

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

In general, Arb does not distinguish between “private” and “public” parts of the API. The implementation is meant to be transparent by design. All methods are intended to be fully documented and tested (exceptions to this are mainly due to lack of time on part of the author). The user should use common sense to determine whether a function is concerned with implementation details, making it likely to change as the implementation changes in the future. The interface of `arb_add()` is probably not going to change in the next version, but `_arb_get_mpn_fized_mod_pi4()` just might.

### 2.4.7 General note on correctness

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

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

All bug reports are highly appreciated.
2.5 Example programs

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

```
make examples
```

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

2.5.1 pi.c

This program computes \( \pi \) to an accuracy of roughly \( n \) decimal digits by calling the \texttt{arb_const_pi()} function with a working precision of roughly \( n \log_2(10) \) bits.

Sample output, computing \( \pi \) to one million digits:

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

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

2.5.2 hilbert_matrix.c

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

Sample output:

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

2.5.3 keiper_li.c

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

The program takes the following parameters:
keiper_li n [-prec prec] [-threads num_threads] [-out out_file]

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

Sample output:

```bash
> 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:  -0.693147180559945309417232121458176568075500134636026 +/- 6.5389e-347
1:  0.023095708966121033814310247906495291621932127152051 +/- 2.0924e-345
2:  0.04617286761402335192864243096033943387066108314123 +/- 1.674e-344
3:  0.06921297351810826793049734887726010689942120263932 +/- 5.0219e-344
4:  0.09219761987306049064967627824904391065514673490213 +/- 2.0089e-343
5:  0.1151085428223549048622182109857276671349132303596 +/- 1.0044e-342
6:  0.1379267687137298829041671370341666356138966078654 +/- 6.0264e-342
7:  0.1606315965299421290402427573585366292282442046163 +/- 2.1092e-341
8:  0.18321945964338257908193931774721858984998098273432 +/- 8.438e-341
9:  0.205657338707910746170289387421343304741236553410044 +/- 7.5931e-340
10: 0.2279393631915177436930340573684453380748385942738 +/- 7.5931e-339
991: 2.319661796161356792837389965699485256210143083341 +/- 2.461e-11
992: 2.32037662952684035498989518332550233162909717288 +/- 9.5363e-11
993: 2.32109206212973262811659116332326802034375592414 +/- 1.8485e-10
994: 2.32180735640188462110255826212502870112747188888893 +/- 3.5907e-10
995: 2.3225173928158572692870296122531402377358152533 +/- 6.978e-10
996: 2.3232444858146238733332360413703912358283071281 +/- 1.3574e-09
997: 2.3239447114886014522889542667580382003452609232475 +/- 2.6433e-09
998: 2.3246517591032700808344143240352605148856869322209 +/- 5.1524e-09
999: 2.325384827586138211981257605206526988544993162101 +/- 1.0053e-08
1000: 2.32605316168664664574065046940832238158044982041872 +/- 3.927e-08
virt/peak/res/peak(MB): 170.18 294.69 7.51 7.51
```

2.5.4 logistic.c

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

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

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

Sample output:

```bash
> 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]
```
2.5.5 real_roots.c

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

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

The following options are available:

- \texttt{--refine d}: If provided, after isolating the roots, attempt to refine the roots to \(d\) digits of accuracy using a few bisection steps followed by Newton’s method with adaptive precision, and then print them.
- \texttt{--verbose}: Print more information.
- \texttt{--maxdepth n}: Stop searching after \(n\) recursive subdivisions.
- \texttt{--maxeval n}: Stop searching after approximately \(n\) function evaluations (the actual number evaluations will be a small multiple of this).
- \texttt{--maxfound n}: Stop searching after having found \(n\) isolated roots.
- \texttt{--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 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, 21.049921875]
found isolated root in: [25, 25.048828125]
found isolated root in: [30.419921875, 30.443359375]
found isolated root in: [32.91015625, 32.958984375]
found isolated root in: [37.548828125, 37.59765625]
found isolated root in: [40.91796875, 40.96796875]
found isolated root in: [43.310546875, 43.339609375]
found isolated root in: [47.998046875, 48.0224609375]
found isolated root in: [49.755859375, 49.7802734375]
---------------------------------------------------------------
Found roots: 10
Subintervals possibly containing undetected roots: 0
Function evaluations: 3058
cpu/wall(s): 0.202 0.202
```

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

```
> build/examples/real_roots 0 0 50.0 -maxfound 1 -refine 75
interval: [0, 50]
maxdepth = 30, maxeval = 100000, maxfound = 1, low_prec = 30
refined root (0/8):
[14.134725141734693790457251983562470270784257115699243175685667460149963429809 +/- 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):
[-9.0226508533409803801581908398800892565246777535156083 +/- 4.85e-52]
refined root (1/6):
[-7.944133587120853123138280555798268532140674396972215 +/- 1.92e-52]
refined root (2/6):
[-6.7860708090071758998780246384496176966053882477393494 +/- 3.84e-52]
refined root (3/6):
[-5.520559828095551059129855512931293573797214280617525 +/- 1.05e-52]
refined root (4/6):
[-4.0879444130970616636988701457391060224764699108530 +/- 2.46e-52]
refined root (5/6):
[-2.338107410459767038489197252446735440638540145672388 +/- 1.48e-52]
---------------------------------------------------------------
Found roots: 6
Subintervals possibly containing undetected roots: 0
```
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/vall(s): 0.032 0.032
```

This does not miss any roots:

```
> build/examples/real_roots 2 1 100
interval: [1, 100]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
-----------------------------------------------
Found roots: 3183
Subintervals possibly containing undetected roots: 0
Function evaluations: 34039
cpu/vall(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/vall(s): 0.171 0.171
virt/peak/res/peak(MB): 28.39 30.38 4.05 4.05
```

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

### 2.5.6 poly_roots.c

This program finds the complex roots of an integer polynomial by calling \texttt{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 \(-\text{refine } d\) is passed, the roots are refined to an absolute tolerance better than \(10^{-d}\). By default, the roots are only computed to sufficient accuracy to isolate them. The refinement is not currently done efficiently.

If \(-\text{print } d\) 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:

```bash
> 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]
...
[96.00000000000000 +/- 1e-17]
[97.00000000000000 +/- 1e-17]
[98.00000000000000 +/- 3e-17]
[99.00000000000000 +/- 3e-17]
[100.00000000000000 +/- 3e-17]
cpu/wall(s): 1.664 1.664
```

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

```bash
> build/examples/poly_roots -refine 100 -print 20 b 16
computing squarefree factorization...
cpu/wall(s): 0.001 0.001
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!
[-0.9430870646605573383 +/- 2.02e-21]
[-0.75534059252067985752 +/- 2.70e-21]
[-0.2499975719077421009 +/- 4.27e-21]
[0.24999757152512726002 +/- 4.43e-21]
[0.7500024284787273998 +/- 4.43e-21]
[1.249997571907742101 +/- 1.43e-20]
[1.753405925206798575 +/- 1.74e-20]
```
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...
roots with multiplicity 1
searching for 6 roots, 3 deflated
prec=32: 3 isolated roots | cpu/wall(s): 0 0.001
done!
[-0.97493 +/- 2.10e-6]
[-0.78183 +/- 1.49e-6]
[-0.43388 +/- 3.75e-6]
[ 0.43388 +/- 3.75e-6]
[ 0.78183 +/- 1.49e-6]
[ 0.97493 +/- 2.10e-6]
roots with multiplicity 2
searching for 4 roots, 2 deflated
prec=32: 2 isolated roots | cpu/wall(s): 0 0
done!
[-0.90618 +/- 1.56e-7]
[-0.53847 +/- 6.91e-7]
[ 0.53847 +/- 6.91e-7]
[ 0.90618 +/- 1.56e-7]
roots with multiplicity 3
searching for 1 roots, 0 deflated
prec=32: 0 isolated roots | cpu/wall(s): 0 0
done!
0
roots with multiplicity 5
searching for 1 roots, 1 deflated
prec=32: 1 isolated roots | cpu/wall(s): 0 0
done!
-1.0000
cpu/wall(s): 0 0.001
```

2.5.7 complex_plot.c

This program plots one of the predefined functions over a complex interval \([x_a, x_b] + [y_a, y_b]i\) using domain coloring, at a resolution of \(x_n\) times \(y_n\) 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 \(x_n = y_n = 512\).

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

Function codes `<func>` are:

- `gamma` - Gamma function
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:

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

### 2.5.8 lvalue.c

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

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

Print value of Dirichlet L-function at \( s = a + bi \).

Default \( a = 0.5, b = 0, p = 53, (q, n) = (1, 0) \) (Riemann zeta)

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

Evaluating the Riemann zeta function and the Dirichlet beta function at \( s = 2 \):

```plaintext
> build/examples/lvalue -re 2 -prec 128
L(s) = [1.6449340668482264364724151664602518922 +/- 4.37e-39]
cpu/wall(s): 0.001 0.001
virt/peak/res/peak(MB): 26.86 26.88 2.05 2.05
```

```plaintext
> build/examples/lvalue -character 4 3 -re 2 -prec 128
L(s) = [0.9159655941772191050460351493238411077 +/- 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 \( s = 1 \):

```plaintext
> build/examples/lvalue -character 1009 101
L(s) = [-0.459256562383872 +/- 5.24e-16] + [1.346937111206009 +/- 3.03e-16]*I
cpu/wall(s): 0.012 0.012
virt/peak/res/peak(MB): 26.86 26.88 2.30 2.30
```
Computing the first few coefficients in the Laurent series of the Riemann zeta function at $s = 1$:

```plaintext
> 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
```

Evaluating the Riemann zeta function near the first nontrivial root:

```plaintext
> 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
```

2.5. Example programs

```plaintext
> build/examples/lvalue -z -re 14.134725 -prec 200
Z(s) = [-1.1241834983941753330111494358128257497862927935658e-7 +/- 4.62e-58]
cpu/wall(s): 0.001 0.001
virt/peak/res/peak(MB): 26.86 26.88 2.57 2.57
```

```plaintext
> 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) = [0.065164588492 +/- 5.39e-13]
[x^3] Z(s+x) = [0.020707782705 +/- 5.37e-13]
cpu/wall(s): 0.002 0.003
virt/peak/res/peak(MB): 26.86 26.88 2.57 2.57
```
FLOATING-POINT NUMBERS

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

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

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

3.1.1 Types, macros and constants

\texttt{mag\_struct}

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

\texttt{mag\_t}

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

3.1.2 Memory management

\begin{verbatim}
void mag_init(mag_t x)
    Initializes the variable \(x\) for use. Its value is set to zero.

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

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

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

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

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

void _mag_vec_clear(mag_ptr v, slong n)
    Clears a vector of length \(n\).
\end{verbatim}
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.

3.1.3 Special values

void mag_zero (mag_t x)
    Sets x to zero.

void mag_one (mag_t x)
    Sets x to one.

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

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

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

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

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

3.1.4 Comparisons

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

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

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

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

3.1.5 Input and output

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

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

3.1.6 Random generation

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

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

void \textbf{mag\_set\_d}(\textit{mag\_t}\ \textit{y}, \textit{double}\ \textit{x})

void \textbf{mag\_set\_fmp}\textbf{r}(\textit{mag\_t}\ \textit{y}, \textit{const fmp}\textbf{r}\textbf{t}\ \textit{x})

void \textbf{mag\_set\_ui}(\textit{mag\_t}\ \textit{y}, \textit{ulong}\ \textit{x})

void \textbf{mag\_set\_fmpz}(\textit{mag\_t}\ \textit{y}, \textit{const fmpz}\textbf{t}\ \textit{x})

Sets \textit{y} to an upper bound for $|\textit{x}|$.

void \textbf{mag\_set\_d\_2exp\_fmpz}(\textit{mag\_t}\ \textit{z}, \textit{double}\ \textit{x}, \textit{const fmpz}\textbf{t}\ \textit{y})

void \textbf{mag\_set\_fmpz\_2exp\_fmpz}(\textit{mag\_t}\ \textit{z}, \textit{const fmpz}\textbf{t}\ \textit{x}, \textit{const fmpz}\textbf{t}\ \textit{y})

void \textbf{mag\_set\_ui\_2exp\_si}(\textit{mag\_t}\ \textit{z}, \textit{ulong}\ \textit{x}, \textit{slong}\ \textit{y})

Sets \textit{z} to an upper bound for $|\textit{x}| \times 2^\textit{y}$.

void \textbf{mag\_get\_fmp}\textbf{r}(\textit{fmp}\textbf{r}\textbf{t}\ \textit{y}, \textit{const mag}\textbf{t}\ \textit{x})

Sets \textit{y} exactly to \textit{x}.

void \textbf{mag\_get\_fmp}\textbf{q}(\textit{fmp}\textbf{q}\textbf{t}\ \textit{y}, \textit{const mag}\textbf{t}\ \textit{x})

Sets \textit{y} exactly to \textit{x}. Assumes that no overflow occurs.

double \textbf{mag\_get\_d}(\textit{const mag}\textbf{t}\ \textit{x})

Returns a \textit{double} giving an upper bound for \textit{x}.

void \textbf{mag\_set\_ui\_lower}(\textit{mag}\textbf{t}\ \textit{z}, \textit{ulong}\ \textit{x})

void \textbf{mag\_set\_fmpz\_lower}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const fmpz}\textbf{t}\ \textit{x})

Sets \textit{y} to a lower bound for $|\textit{x}|$.

void \textbf{mag\_set\_fmpz\_2exp\_fmpz\_lower}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const fmpz}\textbf{t}\ \textit{x}, \textit{const fmpz}\textbf{t}\ \textit{y})

Sets \textit{z} to a lower bound for $|\textit{x}| \times 2^\textit{y}$.

3.1.8 Arithmetic

void \textbf{mag\_mul\_2exp\_si}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{slong}\ \textit{y})

void \textbf{mag\_mul\_2exp\_fmpz}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{const fmpz}\textbf{t}\ \textit{y})

Sets \textit{z} to $\textit{x} \times 2^\textit{y}$. This operation is exact.

void \textbf{mag\_mul}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{const mag}\textbf{t}\ \textit{y})

void \textbf{mag\_mul\_fmpz}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{const fmpz}\textbf{t}\ \textit{y})

Sets \textit{z} to an upper bound for $\textit{xy}$.

void \textbf{mag\_add}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{const mag}\textbf{t}\ \textit{y})

void \textbf{mag\_add\_ui}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{ulong}\ \textit{y})

Sets \textit{z} to an upper bound for $\textit{x} + \textit{y}$.

void \textbf{mag\_addmul}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{const mag}\textbf{t}\ \textit{y})

Sets \textit{z} to an upper bound for $\textit{z} + \textit{xy}$.

void \textbf{mag\_add\_2exp\_fmpz}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{const fmpz}\textbf{t}\ \textit{e})

Sets \textit{z} to an upper bound for $\textit{x} + 2^\textit{e}$.

void \textbf{mag\_add\_ui\_2exp\_si}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{ulong}\ \textit{y}, \textit{slong}\ \textit{e})

Sets \textit{z} to an upper bound for $\textit{x} + \textit{y}2^\textit{e}$.

void \textbf{mag\_div}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{const mag}\textbf{t}\ \textit{y})

void \textbf{mag\_div\_ui}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{ulong}\ \textit{y})

void \textbf{mag\_div\_fmpz}(\textit{mag}\textbf{t}\ \textit{z}, \textit{const mag}\textbf{t}\ \textit{x}, \textit{const fmpz}\textbf{t}\ \textit{y})

Sets \textit{z} to an upper bound for $\textit{x}/\textit{y}$. 
void \texttt{mag\_mul\_lower}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{const} \, \texttt{mag\_t} \, y) \;

void \texttt{mag\_mul\_ui\_lower}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{ulong} \, y) \;

void \texttt{mag\_mul\_fmpz\_lower}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{const} \, \texttt{fmpz\_t} \, y) \;

Sets \( z \) to a lower bound for \( xy \).

void \texttt{mag\_add\_lower}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{const} \, \texttt{mag\_t} \, y) \;

Sets \( z \) to a lower bound for \( x + y \).

void \texttt{mag\_sub}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{const} \, \texttt{mag\_t} \, y) \;

Sets \( z \) to an upper bound for \( \max(x - y, 0) \).

void \texttt{mag\_sub\_lower}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{const} \, \texttt{mag\_t} \, y) \;

Sets \( z \) to a lower bound for \( \max(x - y, 0) \).

3.1.9 Fast, unsafe arithmetic

The following methods assume that all inputs are finite and that all exponents (in all inputs as well as the final result) fit as \texttt{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 \texttt{mag\_fast\_init\_set}(\texttt{mag\_t} \, x, \texttt{const} \, \texttt{mag\_t} \, y) \;

Initialises \( x \) and sets it to the value of \( y \).

void \texttt{mag\_fast\_zero}(\texttt{mag\_t} \, x) \;

Sets \( x \) to zero.

int \texttt{mag\_fast\_is\_zero}(\texttt{const} \, \texttt{mag\_t} \, x) \;

Returns nonzero iff \( x \) is zero.

void \texttt{mag\_fast\_mul}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{const} \, \texttt{mag\_t} \, y) \;

Sets \( z \) to an upper bound for \( xy \).

void \texttt{mag\_fast\_addmul}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{const} \, \texttt{mag\_t} \, y) \;

Sets \( z \) to an upper bound for \( z + xy \).

void \texttt{mag\_fast\_add\_2exp\_si}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{slong} \, e) \;

Sets \( z \) to an upper bound for \( x + 2^e \).

void \texttt{mag\_fast\_mul\_2exp\_si}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{slong} \, e) \;

Sets \( z \) to an upper bound for \( x2^e \).

3.1.10 Powers and logarithms

void \texttt{mag\_pow\_ui}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{ulong} \, e) \;

void \texttt{mag\_pow\_fmpz}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{const} \, \texttt{fmpz\_t} \, e) \;

Sets \( z \) to an upper bound for \( x^e \). Requires \( e \geq 0 \).

void \texttt{mag\_pow\_ui\_lower}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{ulong} \, e) \;

Sets \( z \) to a lower bound for \( x^e \).

void \texttt{mag\_sqrt}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x) \;

Sets \( z \) to an upper bound for \( \sqrt{x} \).

void \texttt{mag\_rsqrt}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x) \;

Sets \( z \) to an upper bound for \( 1/\sqrt{x} \).

void \texttt{mag\_hypot}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{const} \, \texttt{mag\_t} \, y) \;

Sets \( z \) to an upper bound for \( \sqrt{x^2 + y^2} \).

void \texttt{mag\_root}(\texttt{mag\_t} \, z, \texttt{const} \, \texttt{mag\_t} \, x, \texttt{ulong} \, n) \;

Sets \( z \) to an upper bound for \( x^{1/n} \). We evaluate \( \exp(\log(1 + 2^{kn}x)/n)2^{-k} \), where \( k \) is chosen so that \( 2^{kn}x \approx 2^{30} \).
void \texttt{mag\_log1p}(\texttt{mag\_t} z, \texttt{const mag\_t} x)
Sets \( z \) to an upper bound for \( \log(1 + x) \). The bound is computed accurately for small \( x \).

void \texttt{mag\_log\_ui}(\texttt{mag\_t} z, \texttt{ulong} n)
Sets \( z \) to an upper bound for \( \log(n) \).

void \texttt{mag\_exp}(\texttt{mag\_t} z, \texttt{const mag\_t} x)
Sets \( z \) to an upper bound for \( \exp(x) \).

void \texttt{mag\_expinv}(\texttt{mag\_t} z, \texttt{const mag\_t} x)
Sets \( z \) to an upper bound for \( \exp(-x) \). As currently implemented, the bound is computed crudely by rounding \( x \) down to an integer before approximating the exponential.

void \texttt{mag\_exp\_m1}(\texttt{mag\_t} z, \texttt{const mag\_t} x)
Sets \( z \) to an upper bound for \( \exp(x) - 1 \). The bound is computed accurately for small \( x \).

void \texttt{mag\_exp\_tail}(\texttt{mag\_t} z, \texttt{const mag\_t} x, \texttt{ulong} N)
Sets \( z \) to an upper bound for \( \sum_{k=N}^{\infty} x^k / k! \).

void \texttt{mag\_binpow\_ui\_ui}(\texttt{mag\_t} z, \texttt{ulong} m, \texttt{ulong} n)
Sets \( z \) to an upper bound for \( (1 + 1/m)^n \).

void \texttt{mag\_geom\_series}(\texttt{mag\_t} res, \texttt{const mag\_t} x, \texttt{ulong} N)
Sets \( res \) to an upper bound for \( \sum_{k=N}^{\infty} x^k / k! \).

3.11 Special functions

void \texttt{mag\_const\_pi}(\texttt{mag\_t} z)
Sets \( z \) to an upper bound for \( \pi \).

void \texttt{mag\_fac\_ui}(\texttt{mag\_t} z, \texttt{ulong} n)
Sets \( z \) to an upper bound for \( n! \).

void \texttt{mag\_rfac\_ui}(\texttt{mag\_t} z, \texttt{ulong} n)
Sets \( z \) to an upper bound for \( 1/n! \).

void \texttt{mag\_bernoulli\_div\_fac\_ui}(\texttt{mag\_t} z, \texttt{ulong} n)
Sets \( z \) to an upper bound for \( |B_n|/n! \) where \( B_n \) denotes a Bernoulli number.

void \texttt{mag\_polylog\_tail}(\texttt{mag\_t} u, \texttt{const mag\_t} z, \texttt{ulong} s, \texttt{ulong} d, \texttt{ulong} N)
Sets \( u \) to an upper bound for

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

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

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

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

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

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

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

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

void \texttt{mag\_hurwitz\_zeta\_uiui}(mag_t res, ulong s, ulong a)

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

\[
\zeta(s, a) \leq \frac{1}{as} + \frac{1}{(s-1)a^{s-1}}
\]

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

### 3.2 \texttt{arf.h} – arbitrary-precision floating-point numbers

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

The exponent of a finite and nonzero floating-point number can be defined in different ways: for example, as the component \( y \) above, or as the unique integer \( e \) such that \( x \times 2^y = m \times 2^e \) where \( 1/2 \leq |m| < 1 \).

The internal representation of an \texttt{arf\_t} stores the exponent in the latter format.

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

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

The \texttt{arf\_t} type is almost identical semantically to the legacy \texttt{fmpr\_t} type, but uses a more efficient internal representation. The most significant differences that the user has to be aware of are:

- The mantissa is no longer represented as a FLINT \texttt{fmpz}, and the internal exponent points to the top of the binary expansion of the mantissa instead of of the bottom. Code designed to manipulate components of an \texttt{fmpr\_t} directly can be ported to the \texttt{arf\_t} type by making use of \texttt{arf\_get\_fmpz\_2exp()} and \texttt{arf\_set\_fmpz\_2exp()}.

- Some \texttt{arf\_t} functions return an \texttt{int} indicating whether a result is inexact, whereas the corresponding \texttt{fmpr\_t} functions return an \texttt{slong} encoding the relative exponent of the error.

#### 3.2.1 Types, macros and constants

\texttt{arf\_struct}

\texttt{arf\_t}

An \texttt{arf\_struct} contains four words: an \texttt{fmpz} exponent (\texttt{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 \texttt{arf\_struct} is used.
An \texttt{arf\_t} is defined as an array of length one of type \texttt{arf\_struct}, permitting an \texttt{arf\_t} to be passed by reference.

\texttt{arf\_rnd\_t}

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

\textbf{ARF\_RND\_DOWN}

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

\textbf{ARF\_RND\_UP}

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

\textbf{ARF\_RND\_FLOOR}

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

\textbf{ARF\_RND\_CEIL}

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

\textbf{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.

\textbf{ARF\_PREC\_EXACT}

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

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

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

### 3.2.2 Memory management

\textbf{void \texttt{arf\_init}\ (arf\_t x)}

Initializes the variable \( x \) for use. Its value is set to zero.

\textbf{void \texttt{arf\_clear}\ (arf\_t x)}

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

\textbf{slong \texttt{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 \texttt{sizeof(arf\_struct)} to get the size of the object as a whole.

### 3.2.3 Special values

\textbf{void \texttt{arf\_zero}\ (arf\_t x)}

\textbf{void \texttt{arf\_one}\ (arf\_t x)}

\textbf{void \texttt{arf\_pos\_inf}\ (arf\_t x)}
void arf_neg_inf(arf_t x)
void arf_nan(arf_t x)
    Sets x respectively to 0, 1, +∞, −∞, NaN.
int arf_is_zero(const arf_t x)
int arf_is_one(const arf_t x)
int arf_is_pos_inf(const arf_t x)
int arf_is_neg_inf(const arf_t x)
int arf_is_nan(const arf_t x)
    Returns nonzero iff x respectively equals 0, 1, +∞, −∞, NaN.
int arf_is_inf(const arf_t x)
    Returns nonzero iff x equals either +∞ or −∞.
int arf_is_normal(const arf_t x)
    Returns nonzero iff x is a finite, nonzero floating-point value, i.e. not one of the special values 0, +∞, −∞, NaN.
int arf_is_special(const arf_t x)
    Returns nonzero iff x is one of the special values 0, +∞, −∞, NaN, i.e. not a finite, nonzero floating-point value.
int arf_is_finite(arf_t x)
    Returns nonzero iff x is a finite floating-point value, i.e. not one of the values +∞, −∞, NaN. (Note that this is not equivalent to the negation of arf_is_inf().)

3.2.4 Assignment, rounding and conversions

void arf_set(arf_t y, const arf_t x)
void arf_set_mpz(arf_t y, const mpz_t x)
void arf_set_fmpz(arf_t y, const fmpz_t x)
void arf_set_ui(arf_t y, ulong x)
void arf_set_si(arf_t y, slong x)
void arf_set_mpfr(arf_t y, const mpfr_t x)
void arf_set_fmpfr(arf_t y, const fmpr_t x)
void arf_set_d(arf_t y, double x)
    Sets y exactly to x.
void arf_swap(arf_t y, arf_t x)
    Swaps y and x efficiently.
void arf_init_set_ui(arf_t y, ulong x)
void arf_init_set_si(arf_t y, slong x)
    Initialises y and sets it to x in a single operation.
int arf_set_round(arf_t y, const arf_t x, slong prec, arf_rnd_t rnd)
int arf_set_round_si(arf_t y, const arf_t x, slong v, slong prec, arf_rnd_t rnd)
int arf_set_round_ui(arf_t y, const arf_t x, ulong v, slong prec, arf_rnd_t rnd)
int arf_set_round_mpz(arf_t y, const mpz_t x, slong prec, arf_rnd_t rnd)
int arf_set_round_fmpz(arf_t y, const fmpz_t x, slong prec, arf_rnd_t rnd)
    Sets y to x, rounded to prec bits in the direction specified by rnd.
void arf_set_si_2exp_si(arf_t y, slong m, slong e)
void arf_set_ui_2exp_si(arf_t y, ulong m, slong e)
void arf_set_fmpz_2exp(arf_t y, const fmpz_t m, const fmpz_t e)

Sets y to $m \times 2^e$.

int arf_set_round_fmpz_2exp(arf_t y, const fmpz_t m, const fmpz_t e, slong prec, arf_rnd_t rnd)

Sets y to $x \times 2^e$, rounded to prec bits in the direction specified by rnd.

void arf_get_fmpz_2exp(fmpz_t m, fmpz_t e, const arf_t x)

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

void arf_frexp(arf_t m, fmpz_t e, const arf_t x)

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

double arf_get_d(const arf_t x, arf_rnd_t rnd)

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

void arf_get_fmpr(fmpr_t y, const arf_t x)

Sets y exactly to x.

int arf_get_mpfr(mpfr_t y, const arf_t x, mpfr_rnd_t rnd)

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

int arf_get_fmpz(fmpz_t z, const arf_t x, arf_rnd_t rnd)

Converts x to a mantissa with predetermined exponent, i.e. computes an integer y such that $y \times 2^e \approx x$, truncating if necessary. Returns 0 if exact and 1 if truncation occurred.

The warnings for arf_get_fmpz() apply.

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

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

3.2. arf.h – arbitrary-precision floating-point numbers
3.2.5 Comparisons and bounds

```c
int arf_equal(const arf_t x, const arf_t y)
int arf_equal_si(const arf_t x, slong y)
          Returns nonzero iff x and y are exactly equal. This function does not treat NaN specially, i.e. NaN compares as equal to itself.
int arf_cmp(const arf_t x, const arf_t y)
int arf_cmp_si(const arf_t x, slong y)
int arf_cmp_ui(const arf_t x, ulong y)
int arf_cmp_d(const arf_t x, double y)
          Returns negative, zero, or positive, depending on whether x is respectively smaller, equal, or greater compared to y. Comparison with NaN is undefined.
int arf_cmpabs(const arf_t x, const arf_t y)
int arf_cmpabs_ui(const arf_t x, ulong y)
int arf_cmpabs_d(const arf_t x, double y)
int arf_cmpabs_mag(const arf_t x, const mag_t y)
          Compares the absolute values of x and y.
int arf_cmp_2exp_si(const arf_t x, slong e)
int arf_cmpabs_2exp_si(const arf_t x, slong e)
          Compares x (respectively its absolute value) with \(2^e\).
int arf_sgn(const arf_t x)
          Returns \(-1\), 0 or \(+1\) according to the sign of x. The sign of NaN is undefined.
void arf_min(arf_t z, const arf_t a, const arf_t b)
void arf_max(arf_t z, const arf_t a, const arf_t b)
          Sets z respectively to the minimum and the maximum of a and b.
slong arf_bits(const arf_t x)
          Returns the number of bits needed to represent the absolute value of the mantissa of x, i.e. the minimum precision sufficient to represent x exactly. Returns 0 if x is a special value.
int arf_is_int(const arf_t x)
          Returns nonzero iff x is integer-valued.
int arf_is_int_2exp_si(const arf_t x, slong e)
          Returns nonzero iff x equals \(n2^e\) for some integer n.
void arf_abs_bound_lt_2exp_fmpz(fmpz_t b, const arf_t x)
          Sets b to the smallest integer such that |x| < 2^b. If x is zero, infinity or NaN, the result is undefined.
void arf_abs_bound_le_2exp_fmpz(fmpz_t b, const arf_t x)
          Sets b to the smallest integer such that |x| ≤ 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 \(-ARP_PREC_EXACT\) and \(ARP_PREC_EXACT\) inclusive. If x is zero, \(-ARP_PREC_EXACT\) is returned, and if x is infinity or NaN, \(ARP_PREC_EXACT\) is returned.
```

3.2.6 Magnitude functions

```c
void arf_get_mag(mag_t y, const arf_t x)
          Sets y to an upper bound for the absolute value of x.
void arf_get_mag_lower(mag_t y, const arf_t x)
          Sets y to a lower bound for the absolute value of x.
```
### 3.2.7 Shallow assignment

void **arf_init_set_shallow**(arf_t z, const arf_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 long as z is in use.

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

Initializes z shallowly to the negation of x.

### 3.2.8 Random number generation

void **arf_randtest**(arf_t x, flint_rand_t state, slong bits, slong mag_bits)

Generates a finite random number whose mantissa has precision at most bits and whose exponent has at most mag_bits bits. The values are distributed non-uniformly: special bit patterns are generated with high probability in order to allow the test code to exercise corner cases.

void **arf_randtest_not_zero**(arf_t x, flint_rand_t state, slong bits, slong mag_bits)

Identical to **arf_randtest()**, except that zero is never produced as an output.

void **arf_randtest_special**(arf_t x, flint_rand_t state, slong bits, slong mag_bits)

Identical to **arf_randtest()**, except that the output occasionally is set to an infinity or NaN.

### 3.2.9 Input and output

void **arf_debug**(const arf_t x)

Prints information about the internal representation of x.

void **arf_print**(const arf_t x)

Prints x as an integer mantissa and exponent.

void **arf_printd**(const arf_t y, slong d)

Prints x as a decimal floating-point number, rounding to d digits. This function is currently implemented using MPFR, and does not support large exponents.
3.2.10 Addition and multiplication

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

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

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

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

### void arf_mul_2exp_fmpz(arf_t y, const arf_t x, const fmpz_t e)
Sets $y = x \times 2^e$ exactly.

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

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

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

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

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

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

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

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

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

### int arf_addmul_2exp(arf_t z, const arf_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)
Sets $z = x + y \times 2^e$, rounded to $prec$ bits in the direction specified by $rnd$, returning nonzero iff the operation is inexact.

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

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

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

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

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

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

### int arf_addmul_mpz(arf_t z, const arf_t x, const mpz_t y, slong prec, arf_rnd_t rnd)
3.2.11 Summation

```c
int arf_sum(arf_t s, arf_srcptr terms, slong len, slong prec, arf_rnd_t rnd)
```
Sets to the sum of the array terms of length len, rounded to prec bits in the direction specified by rnd. The sum is computed as if done without any intermediate rounding error, with only a single rounding applied to the final result. Unlike repeated calls to `arf_add()` with infinite precision, this function does not overflow if the magnitudes of the terms are far apart. Warning: this function is implemented naively, and the running time is quadratic with respect to len in the worst case.

3.2.12 Division

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

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

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

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

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

```c
int arf_div_mpz(arf_t z, const arf_t x, const mpz_t y, slong prec, arf_rnd_t rnd)
```

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

Sets \( z = x/y \), rounded to prec bits in the direction specified by rnd, returning nonzero iff the operation is inexact. The result is NaN if y is zero.

3.2.13 Square roots

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

```c
int arf_sqrt_ui(arf_t z, const arf_t x, ulong k, slong prec, arf_rnd_t rnd)
```

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

```c
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);
```

```c
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.

```c
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 $e + f i = (a + bi)^2$. This function has identical semantics to `arf_complex_mul()` (with $c = a$, $b = d$), but is faster.

3.2.15 Low-level methods

```c
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.

```c
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.

```c
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 $x$.

```c
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.

```c
int _arf_set_round_mpn(arf_t z, slong * exp_shift, mp_srcptr xp, 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 $xp$, 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 $z$.  

---

40 Chapter 3. Floating-point numbers
CHAPTER
FOUR

REAL AND COMPLEX NUMBERS

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

4.1 arb.h – real numbers

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

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

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

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

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

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

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

4.1.1 Types, macros and constants

arb_struct

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

arb_ptr
Alias for \textit{arb_struct *}, used for vectors of numbers.

arb_srcptr
Alias for \textit{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 \textit{arf_t}.

arb_radref(x)
Macro returning a pointer to the radius of \(x\) as a \textit{mag_t}.

4.1.2 Memory management

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

void \textit{arb_clear}(\textit{arb_t }x)
Clears the variable \(x\), freeing or recycling its allocated memory.

\textit{arb_ptr }\textit{arb_vec_init}(\textit{slong }n)
Returns a pointer to an array of \(n\) initialized \textit{arb_struct} entries.

void \textit{arb_vec_clear}(\textit{arb_ptr }v, \textit{slong }n)
Clears an array of \(n\) initialized \textit{arb_struct} entries.

void \textit{arb_swap}(\textit{arb_t }x, \textit{arb_t }y)
Swaps \(x\) and \(y\) efficiently.

\textit{slong }\textit{arb_allocated_bytes}(\textit{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 \textit{sizeof(arb_struct)} to get the size of the object as a whole.

\textit{slong }\textit{arb_vec_allocated_bytes}(\textit{arb_srcptr }vec, \textit{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 \textit{arb_vec_estimate_allocated_bytes}(\textit{slong }len, \textit{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 \textit{double} to avoid overflow issues when both \textit{len} and \textit{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.
4.1.3 Assignment and rounding

```c
void arb_set(arb_t y, const arb_t x)
void arb_set_arf(arb_t y, const arf_t x)
void arb_set_si(arb_t y, slong x)
void arb_set_ui(arb_t y, ulong x)
void arb_set_d(arb_t y, double x)
void arb_set_fmpz(arb_t y, const fmpz_t x)
void arb_set_fmpz_2exp(arb_t y, const fmpz_t x, const fmpz_t e)
void arb_set_round(arb_t y, const arb_t x, slong prec)
void arb_set_round_fmpz(arb_t y, const fmpz_t x, slong prec)
void arb_set_round_fmpz_2exp(arb_t y, const fmpz_t x, const fmpz_t e, slong prec)
void arb_set_fmpq(arb_t y, const fmpq_t x, slong prec)
```

Sets `y` to the value of `x` without rounding.

```c
void 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.

```c
char * arb_get_str(const arb_t x, slong n, ulong flags)
```

Returns a nice human-readable representation of `x`, with at most `n` digits of the midpoint printed.

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

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

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

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

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

4.1.4 Assignment of special values

```c
void arb_zero(arb_t x)
```

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

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

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

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

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

4.1.5 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)
    Prints x in decimal. The printed value of the radius is not adjusted to compensate for the fact that
    the binary-to-decimal conversion of both the midpoint and the radius introduces additional error.

void arb_printn (const arb_t x, slong digits, ulong flags)
    Prints a nice decimal representation of x. By default, the output 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.

4.1.6 Random number generation

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

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

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

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

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

void arb_get_rand_fmpq (fmpq_t q, flint_rand_t state, const arb_t x, slong bits)
    Sets q to a random rational number from the interval represented by x. A denominator is chosen
    by multiplying the binary denominator of x by a random integer up to bits bits.
The outcome is undefined if the midpoint or radius of \( x \) is non-finite, or if the exponent of the midpoint or radius is so large or small that representing the endpoints as exact rational numbers would cause overflows.

### 4.1.7 Radius and interval operations

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

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

- **void arb_add_error_arf(arb_t x, const arf_t err)**  
  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)**  
  Adds \( 2^e \) to the radius of \( x \).

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

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

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

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

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

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

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

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

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

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

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

  This method aborts if \( x \) is infinite or NaN, or if the difference between the exponents of the midpoint and the radius is so large that allocating memory for the result fails.
Warning: this method will allocate a huge amount of memory to store the result if the exponent difference is huge. Memory allocation could succeed even if the required space is far larger than the physical memory available on the machine, resulting in swapping. It is recommended to check that the midpoint and radius of \( x \) both are within a reasonable range before calling this method.

```c
void arb_set_interval_arf(const arb_t x, const arf_t a, const arf_t b, slong prec)
void arb_set_interval_mpfr(const 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 \( a \leq b \).
```

```c
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.
```

```c
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 \( \text{ARF\_PREC\_EXACT} \).
```

```c
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 \( \text{arb\_rel\_error\_bits()} \).
```

```c
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.
```

```c
void arb_trim(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.
```

```c
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.
```

```c
void arb_floor(const arb_t y, const arb_t x, slong prec)
void arb ceil(const arb_t y, const arb_t x, slong prec)
Sets \( y \) to a ball containing \([x]\) and \([x]\) respectively, with the midpoint of \( y \) rounded to at most \( \text{prec} \) bits.
```

```c
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 \( \text{mid} \), \( \text{rad} \), \( \text{exp} \) such that \( x \in [\text{mid} - \text{rad}, \text{mid} + \text{rad}] \times 10^{\text{exp}} \) and such that the larger out of \( \text{mid} \) and \( \text{rad} \) has at least \( n \) digits plus a few guard digits. If \( x \) is infinite or exactly zero, the outputs are all set to zero.
```

```c
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 \( \text{prec} \) bits in the direction \( \text{rnd} \) 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 \texttt{arf_set_round()}, or \texttt{arf_get_mpfr()}, or \texttt{arf_get_d()} to the midpoint of \(x\) is guaranteed to return a correctly rounded \texttt{arf_t}, \texttt{mpfr_t} (provided that \texttt{prec} is the precision of the output variable), or \texttt{double} (provided that \texttt{prec} is 53). Moreover, \texttt{arf_get_mpfr()} is guaranteed to return the correct ternary value according to MPFR semantics.

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

This function may perform a fast, inexact test; that is, it may return zero in some cases even when correct rounding actually is possible. To be conservative, zero is returned when \(x\) is non-finite, even if it is an “exact” infinity.

4.1.8 Comparisons

\begin{verbatim}
int arb_is_zero(const arb_t x)  Returns nonzero iff the midpoint and radius of \(x\) are both zero.
int arb_is_nonzero(const arb_t x)  Returns nonzero iff zero is not contained in the interval represented by \(x\).
int arb_is_one(const arb_t f)  Returns nonzero iff \(x\) is exactly 1.
int arb_is_finite(const arb_t x)  Returns nonzero iff the midpoint and radius of \(x\) are both finite floating-point numbers, i.e. not infinities or NaN.
int arb_is_exact(const arb_t x)  Returns nonzero iff the radius of \(x\) is zero.
int arb_is_int(const arb_t x)  Returns nonzero iff \(x\) is an exact integer.
int arb_equal(const arb_t x, const arb_t y)  Returns nonzero iff \(x\) and \(y\) are equal as balls, i.e. have both the same midpoint and radius.
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)
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)
\end{verbatim}
int arb_contains_si(const arb_t x, slong y)

int arb_contains_mpfr(const arb_t x, const mpfr_t y)

int arb_contains(const arb_t x, const arb_t y)

Returns nonzero iff the given number (or ball) \( y \) is contained in the interval represented by \( x \).

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

int arb_contains_int(const arb_t x)

Returns nonzero iff the interval represented by \( x \) contains an integer.

int arb_contains_zero(const arb_t x)

int arb_contains_negative(const arb_t x)

int arb_contains_nonpositive(const arb_t x)

int arb_contains_positive(const arb_t x)

int arb_contains_nonnegative(const arb_t x)

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

int arb_eq(const arb_t x, const arb_t y)

int arb_ne(const arb_t x, const arb_t y)

int arb_lt(const arb_t x, const arb_t y)

int arb_le(const arb_t x, const arb_t y)

int arb_gt(const arb_t x, const arb_t y)

int arb_ge(const arb_t x, const arb_t y)

Respectively performs the comparison \( x = y \), \( x \neq y \), \( x < y \), \( x \leq y \), \( x > y \), \( x \geq y \) in a mathematically meaningful way. If the comparison \( t (op) u \) holds for all \( t \in x \) and all \( u \in y \), returns 1. Otherwise, returns 0.

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

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

### 4.1.9 Arithmetic

void arb_neg(arb_t y, const arb_t x)

void arb_neg_round(arb_t y, const arb_t x, slong prec)

Sets \( y \) to the negation of \( x \).

void arb_abs(arb_t y, const arb_t x)

Sets \( y \) to the absolute value of \( x \). No attempt is made to improve the interval represented by \( x \) if it contains zero.

void arb_sgn(arb_t y, const arb_t x)

Sets \( y \) to the sign function of \( x \). The result is \([0 \pm 1]\) if \( x \) contains both zero and nonzero numbers.

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

void arb_max(arb_t z, const arb_t x, const arb_t y, slong prec)

Sets \( z \) respectively to the minimum and the maximum of \( x \) and \( y \).

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

void arb_add_arf(arb_t z, const arb_t x, const arf_t y, slong prec)
void arb_add_ui(const arb_t z, arb_t x, ulong y, slong prec)
void arb_add_si(const arb_t z, arb_t x, slong y, slong prec)
void arb_add_fmpz(const arb_t z, arb_t x, fmpz_t y, slong prec)
  Sets \( z = x + y \), rounded to \( \text{prec} \) bits. The precision can be \text{ARF\_PREC\_EXACT} provided that the result fits in memory.
void arb_add_fmpz_2exp(const arb_t z, arb_t x, fmpz_t m, fmpz_t e, slong prec)
  Sets \( z = x + m \cdot 2^e \), rounded to \( \text{prec} \) bits. The precision can be \text{ARF\_PREC\_EXACT} provided that the result fits in memory.
void arb_sub(const arb_t z, arb_t x, arb_t y, slong prec)
void arb_sub_arf(const arb_t z, arb_t x, arf_t y, slong prec)
void arb_sub_si(const arb_t z, arb_t x, ulong y, slong prec)
void arb_sub_fmpz(const arb_t z, arb_t x, fmpz_t y, slong prec)
void arb_mul(const arb_t z, arb_t x, arb_t y, slong prec)
void arb_mul_arf(const arb_t z, arb_t x, arf_t y, slong prec)
void arb_mul_si(const arb_t z, arb_t x, slong y, slong prec)
void arb_mul_2exp(const arb_t z, arb_t x, slong y, slong prec)
void arb_mul_2exp_si(const arb_t y, arb_t x, slong e)
void arb_mul_2exp_fmpz(const arb_t y, arb_t x, fmpz_t e)
  Sets \( y \) to \( x \times 2^e \).
void arb_addmul(const arb_t z, arb_t x, arb_t y, slong prec)
void arb_addmul_arf(const arb_t z, arb_t x, arf_t y, slong prec)
void arb_addmul_si(const arb_t z, arb_t x, slong y, slong prec)
void arb_addmul_ui(const arb_t z, arb_t x, ulong y, slong prec)
void arb_addmul_fmpz(const arb_t z, arb_t x, fmpz_t y, slong prec)
  Sets \( z = x + y \), rounded to \( \text{prec} \) bits. The precision can be \text{ARF\_PREC\_EXACT} provided that the result fits in memory.
void arb_submul(const arb_t z, arb_t x, arb_t y, slong prec)
void arb_submul_arf(const arb_t z, arb_t x, arf_t y, slong prec)
void arb_submul_si(const arb_t z, arb_t x, slong y, slong prec)
void arb_submul_ui(const arb_t z, arb_t x, ulong y, slong prec)
void arb_submul_fmpz(const arb_t z, arb_t x, fmpz_t y, slong prec)
  Sets \( z = x - y \), rounded to \( \text{prec} \) bits. The precision can be \text{ARF\_PREC\_EXACT} provided that the result fits in memory.
void arb_inv(const arb_t z, arb_t x, slong prec)
void arb_div(const arb_t z, arb_t x, arb_t y, slong prec)
void arb_div_arf(const arb_t z, arb_t x, arf_t y, slong prec)
void arb_div_si(const arb_t z, 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 \( \text{prec} \) bits. If \( y \) contains zero, \( z \) is set to \( 0/\pm \infty \). Otherwise, error propagation uses the rule

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

where \( -1 \leq \xi_1, \xi_2 \leq 1 \), and where the triangle inequality has been applied to the numerator and the reverse triangle inequality has been applied to the denominator.

void arb_div_2expm1_ui(arb_t z, const arb_t x, ulong n, slong prec)

Sets \( z = x/(2^n - 1) \), rounded to \( \text{prec} \) bits.

### 4.1.10 Powers and roots

void arb_sqrt(arb_t z, const arb_t x, slong prec)
void arb_sqrt_arf(arb_t z, const arf_t x, slong prec)
void arb_sqrt_fmpz(arb_t z, const fmpz_t x, slong prec)

Sets \( z \) to the square root of \( x \), rounded to \( \text{prec} \) bits.

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

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 \( \text{prec} \) 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 \( \text{prec} \) bits. This function selects between different algorithms. For large \( k \), it evaluates \( \exp(\log(x)/k) \). For small \( k \), it uses \( \text{arf_root()} \) 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

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

This is evaluated using \( \text{mag_log1p()} \).

void arb_root(arb_t z, const arb_t x, ulong k, slong prec)

Alias for \( \text{arb_root_ui()} \), 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_ui_pow_ui(arb_t y, ulong b, ulong e, slong prec)
void arb_pow_fmpq(arb_t y, const arb_t x, const fmpq_t a, slong prec)
void arb_log_ui_from_prev(arb_t y, const arb_t x, slong prec)
void arb_log_fmpz(arb_t y, const arb_t x, const fmpz_t a, slong prec)
void arb_si_pow_ui(arb_t y, const arb_t x, slong prec)
void arb_log_arf(arb_t y, const arf_t x, slong prec)
void arb_log_arf(arb_t y, const arf_t x, slong prec)
void arb_log_fmpz(arb_t y, const arb_t x, const fmpz_t a, slong prec)
void arb_log_ui(arb_t y, const arb_t x, slong prec)
void arb_log1p(arb_t y, const arb_t x, slong prec)
    Computes log(1 + x), given log(k0) where \( k_0 < k_1 \). At high precision, this function uses the formula
    \[ \log(k_1) = \log(k_0) + 2 \text{atanh}((k_1 - k_0)/(k_1 + k_0)), \]
    evaluating the inverse hyperbolic tangent using binary splitting (for best efficiency, \( k_0 \) should be large and \( k_1 - k_0 \) should be small). Otherwise, it ignores \( \log(k_0) \) and evaluates the logarithm the usual way.

void arb_log1p(arb_t y, const arb_t x, slong prec)
    Computes log(1 + x), given log(k0) where \( k_0 < k_1 \). At high precision, this function uses the formula
    \[ \log(k_1) = \log(k_0) + 2 \text{atanh}((k_1 - k_0)/(k_1 + k_0)), \]
    evaluating the inverse hyperbolic tangent using binary splitting (for best efficiency, \( k_0 \) should be large and \( k_1 - k_0 \) should be small). Otherwise, it ignores \( \log(k_0) \) and evaluates the logarithm the usual way.

4.1.11 Exponentials and logarithms

void arb_log_ui(arb_t y, ulong x, slong prec)
void arb_log_fmpz(arb_t y, const fmpz_t x, slong prec)
void arb_log_arf(arb_t y, const arf_t x, slong prec)
void arb_log(arb_t y, const arb_t x, slong prec)
void arb_log_ui_from_prev(arb_t y, const arb_t x, ulong k1, ulong k0, ulong k0, slong prec)
void arb_log1p(arb_t y, const arb_t x, slong prec)
    Sets \( y = b^x \) using binary exponentiation (with an initial division if \( e < 0 \)). Provided that \( b \) and \( e \) are small enough and the exponent is positive, the exact power can be computed by setting the precision to \texttt{ARF\_PREC\_EXACT}.

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

void arb_pow_fmpq(arb_t y, const arb_t x, const fmpz_t a, slong prec)
void arb_log_base_ui(arb_t y, const arb_t x, ulong e, slong prec)
void arb_log1p(arb_t y, const arb_t x, slong prec)
    Computes \( \log(y) \) if \( y \) is a small exact half-integer, and generally as \( z = \exp(y \log x) \).

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

4.1. arb.h – real numbers
void \texttt{arb\_exp\_invexp}(\texttt{arb\_t} z, \texttt{arb\_t} w, \texttt{const arb\_t} x, \texttt{slong} prec)

Sets \(z = \exp(x)\) and \(w = \exp(-x)\). The second exponential is computed from the first using a division, but propagated error bounds are computed separately.

### 4.1.12 Trigonometric functions

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

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

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

Sets \(s = \sin(x)\), \(c = \cos(x)\). Error propagation uses the rule \(|\sin(m \pm r) - \sin(m)| \leq \min(r, 2)|\).

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

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

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

Sets \(s = \sin(\pi x)\), \(c = \cos(\pi x)\).

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

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

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

Sets \(z = \text{sinc}(x) = \frac{\sin(x)}{x}\).

### 4.1.13 Inverse trigonometric functions

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

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

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

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

The function \texttt{arb\_atan\_arf()} uses lookup tables if possible, and otherwise falls back to \texttt{arb\_atan\_arf\_bb()}.
void arb_asin(arb_t z, const arb_t x, slong prec)
    Sets \( z = \arcsin(x) = \arctan(x/\sqrt{1-x^2}) \). If \( x \) is not contained in the domain \([-1,1]\), the result is an indeterminate interval.

void arb_acos(arb_t z, const arb_t x, slong prec)
    Sets \( z = \arccos(x) = \pi/2 - \arcsin(x) \). If \( x \) is not contained in the domain \([-1,1]\), the result is an indeterminate interval.

### 4.1.14 Hyperbolic functions

void arb_sinh(arb_t s, const arb_t x, slong prec)
void arb_cosh(arb_t c, const arb_t x, slong prec)
void arb_sinh_cosh(arb_t s, arb_t c, const arb_t x, slong prec)
    Sets \( s = \sinh(x) \), \( c = \cosh(x) \). If the midpoint of \( x \) is close to zero and the hyperbolic sine is to be computed, evaluates \((e^{2x} \pm 1)/2e^x\) via \( \text{arb\_expm1()} \) 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 \( y = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} \), evaluated via \( \text{arb\_expm1()} \) 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 \( y = \coth(x) = \frac{\cosh(x)}{\sinh(x)} \), evaluated using the same strategy as \( \text{arb\_tanh()} \).

### 4.1.15 Inverse hyperbolic functions

void arb_atanh(arb_t z, const arb_t x, slong prec)
    Sets \( z = \tanh^{-1}(x) \).

void arb_asinh(arb_t z, const arb_t x, slong prec)
    Sets \( z = \sinh^{-1}(x) \).

void arb_acosh(arb_t z, const arb_t x, slong prec)
    Sets \( z = \cosh^{-1}(x) \). If \( x < 1 \), the result is an indeterminate interval.

### 4.1.16 Constants

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

void arb_const_pi(arb_t z, slong prec)
    Computes \( \pi \).

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 \texttt{arb\_const\_e}(\texttt{arb\_t z}, \texttt{slong prec})
Computes $e = \exp(1)$.

void \texttt{arb\_const\_khinchin}(\texttt{arb\_t z}, \texttt{slong prec})
Computes Khinchin’s constant $K_0$.

void \texttt{arb\_const\_glaisher}(\texttt{arb\_t z}, \texttt{slong prec})
Computes the Glaisher-Kinkelin constant $A = \exp(1/12 - \zeta'(-1))$.

void \texttt{arb\_const\_apery}(\texttt{arb\_t z}, \texttt{slong prec})
Computes Apery’s constant $\zeta(3)$.

\section*{4.1.17 Gamma function and factorials}

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

void \texttt{arb\_rising\_ui}(\texttt{arb\_t z}, \texttt{arb\_t x}, \texttt{ulong n}, \texttt{slong prec})
Computes the rising factorial $z = x(x+1)(x+2)\cdots(x+n-1)$.

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

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

void \texttt{arb\_rising\_fmpq\_ui}(\texttt{arb\_t z}, \texttt{fmpq\_t x}, \texttt{ulong n}, \texttt{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 \texttt{prec}, it is more efficient to convert $x$ to an approximation and use \texttt{arb\_rising\_ui()}.

void \texttt{arb\_rising2\_ui\_bs}(\texttt{arb\_t u}, \texttt{arb\_t v}, \texttt{arb\_t x}, \texttt{ulong n}, \texttt{slong prec})
void \texttt{arb\_rising2\_ui\_rs}(\texttt{arb\_t u}, \texttt{arb\_t v}, \texttt{arb\_t x}, \texttt{ulong n}, \texttt{ulong step}, \texttt{slong prec})

void \texttt{arb\_rising2\_ui}(\texttt{arb\_t u}, \texttt{arb\_t v}, \texttt{arb\_t x}, \texttt{ulong n}, \texttt{slong prec})
Letting $u(x) = x(x+1)(x+2)\cdots(x+n-1)$, simultaneously compute $u(x)$ and $v(x) = u'(x)$, respectively using binary splitting, rectangular splitting (with optional nonzero step length \texttt{step} to override the default choice), and an automatic algorithm choice.

void \texttt{arb\_fac\_ui}(\texttt{arb\_t z}, \texttt{ulong n}, \texttt{slong prec})
Computes the factorial $z = n!$ via the gamma function.

void \texttt{arb\_doublefac\_ui}(\texttt{arb\_t z}, \texttt{ulong n}, \texttt{slong prec})
Computes the double factorial $z = n!!$ via the gamma function.

void \texttt{arb\_bin\_ui}(\texttt{arb\_t z}, \texttt{arb\_t n}, \texttt{ulong k}, \texttt{slong prec})
void \texttt{arb\_bin\_uiui}(\texttt{arb\_t z}, \texttt{ulong n}, \texttt{ulong k}, \texttt{slong prec})
Computes the binomial coefficient $z = \binom{n}{k}$, via the rising factorial as $\binom{n}{k} = (n-k+1)k!/(k!k!)$.

void \texttt{arb\_gamma}(\texttt{arb\_t z}, \texttt{arb\_t x}, \texttt{slong prec})
void \texttt{arb\_gamma\_fmpq}(\texttt{arb\_t z}, \texttt{fmpq\_t x}, \texttt{slong prec})
void \texttt{arb\_gamma\_fmpz}(\texttt{arb\_t z}, \texttt{fmpz\_t x}, \texttt{slong prec})
Computes the gamma function $z = \Gamma(x)$.

void \texttt{arb\_lgamma}(\texttt{arb\_t z}, \texttt{arb\_t x}, \texttt{slong prec})
Computes the logarithmic gamma function $z = \log \Gamma(x)$. The complex branch structure is assumed, so if $x \leq 0$, the result is an indeterminate interval.
void \texttt{arb_rgamma} (\texttt{arb_t} \, z, \texttt{const arb_t} \, x, \texttt{slong} \, \texttt{prec})

Computes the reciprocal gamma function \( z = 1/\Gamma(x) \), avoiding division by zero at the poles of the gamma function.

void \texttt{arb_digamma} (\texttt{arb_t} \, y, \texttt{const arb_t} \, x, \texttt{slong} \, \texttt{prec})

Computes the digamma function \( y = \psi(x) = (\log \Gamma(x))' = \Gamma'(x)/\Gamma(x) \).

### 4.1.18 Zeta function

void \texttt{arb_zeta_ui_vec_borwein} (\texttt{arb_ptr} \, z, \texttt{ulong} \, \texttt{start}, \texttt{slong} \, \texttt{num}, \texttt{ulong} \, \texttt{step}, \texttt{slong} \, \texttt{prec})

Evaluates \( \zeta(s) \) at \texttt{num} consecutive integers \( s \) beginning with \texttt{start} and proceeding in increments of \texttt{step}. Uses Borwein’s formula ([Bor2000], [GS2003]), implemented to support fast multi-evaluation (but also works well for a single \( s \)).

Requires \( \texttt{start} \geq 2 \). For efficiency, the largest \( s \) should be at most about as large as \texttt{prec}. Arguments approaching \texttt{LONG_MAX} will cause overflows. One should therefore only use this function for \( s \) up to about \texttt{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 \( 2^{\texttt{prec}} \). 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 \( \texttt{step}^s \). 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 \texttt{arb_zeta_ui_asym} (\texttt{arb_t} \, x, \texttt{along} \, \texttt{s}, \texttt{slong} \, \texttt{prec})

void \texttt{arb_zeta_ui_euler_product} (\texttt{arb_t} \, z, \texttt{ulong} \, \texttt{s}, \texttt{slong} \, \texttt{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 \texttt{acb_dirichlet_euler_product_real_ui()}.

void \texttt{arb_zeta_ui_bernoulli} (\texttt{arb_t} \, x, \texttt{ulong} \, \texttt{s}, \texttt{slong} \, \texttt{prec})

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

void \texttt{arb_zeta_ui_borwein_bsplit} (\texttt{arb_t} \, x, \texttt{ulong} \, \texttt{s}, \texttt{slong} \, \texttt{prec})

Computes \( \zeta(s) \) for arbitrary \( s \geq 2 \) using a binary splitting implementation of Borwein’s algorithm. This has quasilinear complexity with respect to the precision (assuming that \( s \) is fixed).

void \texttt{arb_zeta_ui_vec} (\texttt{arb_ptr} \, x, \texttt{ulong} \, \texttt{start}, \texttt{slong} \, \texttt{num}, \texttt{slong} \, \texttt{prec})

void \texttt{arb_zeta_ui_vec_even} (\texttt{arb_ptr} \, x, \texttt{ulong} \, \texttt{start}, \texttt{slong} \, \texttt{num}, \texttt{slong} \, \texttt{prec})

void \texttt{arb_zeta_ui_vec_odd} (\texttt{arb_ptr} \, x, \texttt{ulong} \, \texttt{start}, \texttt{slong} \, \texttt{num}, \texttt{slong} \, \texttt{prec})

Computes \( \zeta(s) \) at \texttt{num} consecutive integers (respectively \texttt{num} even or \texttt{num} odd integers) beginning with \( s = \texttt{start} \geq 2 \), automatically choosing an appropriate algorithm.

void \texttt{arb_zeta_ui} (\texttt{arb_t} \, x, \texttt{ulong} \, \texttt{s}, \texttt{slong} \, \texttt{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 \texttt{arb_zeta} (\texttt{arb_t} \, z, \texttt{const arb_t} \, s, \texttt{slong} \, \texttt{prec})

Sets \( z \) to the value of the Riemann zeta function \( \zeta(s) \).

For computing derivatives with respect to \( s \), use \texttt{arb_poly_zeta_series()}.  

void \texttt{arb_hurwitz_zeta} (\texttt{arb_t} \, z, \texttt{const arb_t} \, s, \texttt{const arb_t} \, a, \texttt{slong} \, \texttt{prec})

Sets \( z \) to the value of the Hurwitz zeta function \( \zeta(s,a) \).

For computing derivatives with respect to \( s \), use \texttt{arb_poly_zeta_series()}.  

---

**4.1. arb.h — real numbers** 55
4.1.19 Bernoulli numbers and polynomials

void arb_bernoulli_ui(arb_t b, ulong n, slong prec)
void arb_bernoulli_fmpz(arb_t b, const fmpz_t n, slong prec)

Sets $b$ to the numerical value of the Bernoulli number $B_n$ approximated to $\text{prec}$ bits.

The internal precision is increased automatically to give an accurate result. Note that, with huge $\text{fmpz}$ 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 $\text{prec}$ 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 $\text{prec}$ bits, computed using the formula

$$B_{2n} = \frac{(-1)^{n+1} 2(2n) \zeta(2n) / (2\pi)^n}{n+1}.$$ 

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 $\text{res}$ 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 $\text{len} - 1$, sets entry $n$ in the output vector $\text{res}$ to

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

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

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

The computation uses the generating function for Bernoulli polynomials.

4.1.20 Polylogarithms

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

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

4.1.21 Other special functions

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

Computes the Fibonacci number $F_n$. Uses the binary squaring algorithm described in [Tak2000]. Provided that $n$ is small enough, an exact Fibonacci number can be computed by setting the precision to $\text{ARF_PREC_EXACT}$.

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 $y$. 

Chapter 4. Real and complex numbers
void arb_chebyshev_t_ui (arb_t x, ulong n, const arb_t x, slong prec)
void arb_chebyshev_t2_ui (arb_t a, arb_t b, ulong n, const arb_t x, slong prec)

void arb_partitions_fmpz (arb_t res, const fmpz_t n, const fmpz_t a, const fmpz_t b, const fmpz_t mmag, slong prec)
void arb_partitions_ui (arb_t res, const fmpz_t n, const fmpz_t a, const fmpz_t b, const fmpz_t mmag, slong prec)

void arb_partitions (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_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)

void arb_bell_fmpz (arb_t res, const fmpz_t n, const fmpz_t a, const fmpz_t b, const fmpz_t mmag, slong prec)
void arb_bell_ui (arb_t res, const fmpz_t n, const fmpz_t a, const fmpz_t b, const fmpz_t mmag, slong prec)

void arb_euler_number_fmpz (arb_t res, const fmpz_t n, const fmpz_t a, const fmpz_t b, const fmpz_t mmag, slong prec)
void arb_euler_number_ui (arb_t res, const fmpz_t n, const fmpz_t a, const fmpz_t b, const fmpz_t mmag, slong prec)

void arb_euler_number (arb_t res, const fmpz_t n, const fmpz_t a, const fmpz_t b, const fmpz_t mmag, slong 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)

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)

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

4.1.22 Internals for computing elementary functions

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

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

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

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

Computes an approximation of \( y = \sum_{k=0}^{N-1} x^k/k! \). Used internally for computing exponentials. The
naive version uses the forward recurrence, and the rs version uses a division-avoiding rectangular splitting scheme.

Requires $N \leq 287$, $0 \leq x \leq 1/16$, and $xn$ positive. The input $x$ is a fixed-point number with $xn$ fractional limbs, and the output $y$ is a fixed-point number with $xn$ fractional limbs plus one extra limb for the integer part of the result.

A bound for the ulp error is written to $error$.

```c
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)
```

```c
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 $N \leq 143$, $0 \leq x \leq 1/16$, 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 $sinonly$).

```c
int _arb_get_mpn_fixed_mod_log2(mp_ptr w, fmpz_t q, mp_limb_t * error, const arf_t x, mp_size_t wm)
```

Attempts to write $w = x - q \log(2)$ with $0 \leq w < \log(2)$, where $w$ is a fixed-point number with $wm$ limbs and ulp error $error$. Returns success.

```c
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 wm)
```

Attempts to write $w = |x| - q\pi/4$ with $0 \leq w < \pi/4$, where $w$ is a fixed-point number with $wm$ 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.

```c
slong _arb_taylor_bound(slong mag, slong prec)
```

Returns $n$ such that $\sum_{k=n}^{\infty} x^k/k! \leq 2^{-prec}$, assuming $|x| \leq 2^{mag} \leq 1/4$.

```c
void arbiof_bb(task_t x, const arf_t x, mp_limb_t prec, int m1)
```

Computes the exponential function using the bit-burst algorithm. If $m1$ is nonzero, the exponential function minus one is computed accurately.

Aborts if $x$ is extremely small or large (where another algorithm should be used).

For large $x$, repeated halving is used. In fact, we always do argument reduction until $|x|$ is smaller than about $2^{-d}$ where $d \approx 16$ to speed up convergence. If $|x| \approx 2^m$, we thus need about $m + d$ squarings.

Computing $\log(2)$ costs roughly 100-200 multiplications, so is not usually worth the effort at very high precision. However, this function could be improved by using log(2) based reduction at precision low enough that the value can be assumed to be cached.

```c
void _arb_exp_sum_bs_simple(fmpz_t T, fmpz_t Q, mp_bitcnt_t * Qexp, const fmpz_t x, mp_bitcnt_t r, slong N)
```

```c
void _arb_exp_sum_bs_powtab(fmpz_t T, fmpz_t Q, mp_bitcnt_t * Qexp, const fmpz_t x, mp_bitcnt_t r, slong N)
```

Computes $T$, $Q$ and $Qexp$ such that $T/(Q2^{Qexp}) = \sum_{k=1}^{N}(x/2^r)^k/k!$ 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.

```c
void _arb_atan_sum_bs_simple(fmpz_t T, fmpz_t Q, mp_bitcnt_t * Qexp, const fmpz_t x, mp_bitcnt_t r, slong N)
```
void _arb_atan_sum_bs_powtab(fmpz_t T, fmpz_t Q, mp_bitcnt_t *Qexp, const fmpz_t x, mp_bitcnt_t r, slong N)
Computes $T$, $Q$ and $Qexp$ such that $\frac{T}{Q^2Qexp} = \sum_{k=1}^{N} (-1)^k (x/2^r)^{2k}/(2k + 1)$ 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
\[
\tan(x) = 2 \tan \left( \frac{x}{1 + \sqrt{1 + x^2}} \right)
\]
is applied up to 8 times to get a small argument. Then a version of the bit-burst algorithm is used. The functional equation
\[
\tan(x) = \tan(p/q) + \tan(w), \quad w = \frac{qx - p}{px + q}, \quad p = \lfloor qx \rfloor
\]
is applied repeatedly instead of integrating a differential equation for the arctangent, as this appears to be more efficient.

4.1.23 Vector functions

void _arb_vec_zero(arb_ptr vec, slong n)
Sets all entries in $vec$ to zero.

int _arb_vec_is_zero(arb_srcptr vec, slong len)
Returns nonzero iff all entries in $x$ are zero.

int _arb_vec_is_finite(arb_srcptr x, slong len)
Returns nonzero iff all entries in $x$ certainly are finite.

void _arb_vec_set(arb_ptr res, arb_srcptr vec, slong len)
Sets $res$ to a copy of $vec$.

void _arb_vec_set_round(arb_ptr res, arb_srcptr vec, slong len, slong prec)
Sets $res$ to a copy of $vec$, rounding each entry to $prec$ bits.

void _arb_vec_swap(arb_ptr vec1, arb_ptr vec2, slong len)
Swaps the entries of $vec1$ and $vec2$.

void _arb_vec_neg(arb_ptr B, arb_srcptr A, slong n)
void _arb_vec_sub(arb_ptr C, arb_srcptr A, arb_srcptr B, slong n, slong prec)
void _arb_vec_add(arb_ptr C, arb_srcptr A, arb_srcptr B, slong n, slong prec)
void _arb_vec_scalar_mul(arb_ptr res, arb_srcptr vec, slong len, const arb_t c, slong prec)
void _arb_vec_scalar_div(arb_ptr res, arb_srcptr vec, slong len, const arb_t c, slong prec)
void _arb_vec_scalar_mul_fmpz(arb_ptr res, arb_srcptr vec, slong len, const fmpz_t c, slong prec)
void _arb_vec_scalar_mul_2exp_si(arb_ptr res, arb_srcptr src, slong len, slong c)
void _arb_vec_scalar_addmul(arb_ptr res, arb_srcptr vec, slong len, const arb_t c, slong prec)
Performs the respective scalar operation elementwise.

void _arb_vec_dot(arb_t res, arb_srcptr vec1, arb_srcptr vec2, slong len2, slong prec)
Sets $res$ to the dot product of $vec1$ and $vec2$.

void _arb_vec_get_mag(mag_t bound, arb_srcptr vec, slong len, slong prec)
Sets $bound$ to an upper bound for the entries in $vec$.

4.1. arb.h – real numbers 59
slong _arb_vec_bits(arb_srcptr x, slong len)
    Returns the maximum of \texttt{arb_bits()} for all entries in \texttt{vec}.

void _arb_vec_set_powers(arb_ptr xs, const arb_t x, slong len, slong prec)
    Sets \texttt{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 \texttt{err} to the radius of the corresponding entry in \texttt{res}.

void _arb_vec_indeterminate(arb_ptr vec, slong len)
    Applies \texttt{arb_indeterminate()} elementwise.

void _arb_vec_trim(arb_ptr res, arb_srcptr vec, slong len)
    Applies \texttt{arb_trim()} elementwise.

int _arb_vec_get_unique_fmpz_vec(fmpz * res, arb_srcptr vec, slong len)
    Calls \texttt{arb_get_unique_fmpz()} elementwise and returns nonzero if all entries can be rounded uniquely to integers. If any entry in \texttt{vec} cannot be rounded uniquely to an integer, returns zero.

4.2 acb.h – complex numbers

An \texttt{acb_t} represents a complex number with error bounds. An \texttt{acb_t} consists of a pair of real number balls of type \texttt{arbstruct}, representing the real and imaginary part with separate error bounds.

An \texttt{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 \texttt{acb_t} type is slightly less developed than that for the \texttt{arb_t} type. In many cases, the user can easily perform missing operations by directly manipulating the real and imaginary parts.

4.2.1 Types, macros and constants

\texttt{acbstruct}

\texttt{acb_t}
    An \texttt{acbstruct} consists of a pair of \texttt{arbstructs}. An \texttt{acb_t} is defined as an array of length one of type \texttt{acbstruct}, permitting an \texttt{acb_t} to be passed by reference.

\texttt{acb_ptr}
    Alias for \texttt{acbstruct *}, used for vectors of numbers.

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

\texttt{acb_realref(x)}
    Macro returning a pointer to the real part of \texttt{x} as an \texttt{arb_t}.

\texttt{acb_imagref(x)}
    Macro returning a pointer to the imaginary part of \texttt{x} as an \texttt{arb_t}.

4.2.2 Memory management

\texttt{void acb_init(acb_t x)}
    Initializes the variable \texttt{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_structs.

void _acb_vec_clear(acb_ptr v, slong n)
    Clears an array of \( n \) initialized acb_structs.

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().

4.2.3 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.
### 4.2.4 Input and output

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

```c
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.
```

### 4.2.5 Random number generation

```c
void acb_randtest(acb_t z, flint_rand_t state, slong prec, slong mag_bits)
    Generates a random complex number by generating separate random real and imaginary parts.
void acb_randtest_special(acb_t z, flint_rand_t state, slong prec, slong mag_bits)
    Generates a random complex number by generating separate random real and imaginary parts.
    Also generates NaNs and infinities.
void acb_randtest_precise(acb_t z, flint_rand_t state, slong prec, slong mag_bits)
    Generates a random complex number with precise real and imaginary parts.
void acb_randtest_param(acb_t z, flint_rand_t state, slong prec, slong mag_bits)
    Generates a random complex number, with very high probability of generating integers and half-
    integers.
```

### 4.2.6 Precision and comparisons

```c
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_equal(const acb_t x, const acb_t y)
Returns nonzero iff \( x \) and \( y \) are identical as sets, i.e. if the real and imaginary parts are equal as balls.

Note that this is not the same thing as testing whether both \( x \) and \( y \) certainly represent the same complex number, unless either \( x \) or \( y \) is exact (and neither contains NaN). To test whether both operands might represent the same mathematical quantity, use \( \text{acb_overlaps()} \) or \( \text{acb_contains()} \), depending on the circumstance.

int acb_equal_si(const acb_t x, slong y)
Returns nonzero iff \( x \) is equal to the integer \( y \).

int acb eq(const acb_t x, const acb_t y)
Returns nonzero iff \( x \) and \( y \) are certainly equal, as determined by testing that \( \text{arb_eq()} \) holds for both the real and imaginary parts.

int acb ne(const acb_t x, const acb_t y)
Returns nonzero iff \( x \) and \( y \) are certainly not equal, as determined by testing that \( \text{arb_ne()} \) holds for either the real or imaginary parts.

int acb_overlaps(const acb_t x, const acb_t y)
Returns nonzero iff \( x \) and \( y \) have some point in common.

void acb_get_abs_ubound_arf(arf_t u, const acb_t z, slong prec)
Sets \( u \) to an upper bound for the absolute value of \( z \), computed using a working precision of \( \text{prec} \) bits.

void acb_get_abs_lbound_arf(arf_t u, const acb_t z, slong prec)
Sets \( u \) to a lower bound for the absolute value of \( z \), computed using a working precision of \( \text{prec} \) bits.

void acb_get_rad_ubound_arf(arf_t u, const acb_t z, slong prec)
Sets \( u \) to an upper bound for the error radius of \( z \) (the value is currently not computed tightly).

void acb_get_mag(mag_t u, const acb_t x)
Sets \( u \) to an upper bound for the absolute value of \( x \).

void acb_get_mag_lower(mag_t u, const acb_t x)
Sets \( u \) to a lower bound for the absolute value of \( x \).

int acb_contains_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 \( x \) contains an integer.

slong acb_rel_error_bits(const acb_t x)
Returns the effective relative error of \( x \) measured in bits. This is computed as if calling \( \text{arb_rel_error_bits()} \) on the real ball whose midpoint is the larger out of the real and imaginary midpoints of \( x \), and whose radius is the larger out of the real and imaginary 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 \( \text{acb_rel_error_bits()} \).

slong acb_bits(const acb_t x)
Returns the maximum of \( \text{arb\_bits} \) 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 \texttt{acb\_trim}(acb\_t y, const acb\_t x)
  \[\text{Sets}\ y\ \text{to a copy of}\ x\ \text{with both the real and imaginary parts trimmed (see}\ \texttt{arb\_trim()}.\]

int \texttt{acb\_is\_real}(const acb\_t x)
  \[\text{Returns nonzero iff the imaginary part of}\ x\ \text{is zero. It does not test whether the real part of}\ x\ \text{also is finite.}\]

int \texttt{acb\_get\_unique\_fmpz}(fmpz\_t z, const acb\_t x)
  \[\text{If}\ x\ \text{contains a unique integer, sets}\ z\ \text{to that value and returns nonzero. Otherwise (if}\ x\ \text{represents no integers or more than one integer), returns zero.}\]

4.2.7 Complex parts

void \texttt{acb\_get\_real}(arb\_t re, const acb\_t z)
  \[\text{Sets}\ re\ \text{to the real part of}\ z.\]

void \texttt{acb\_get\_imag}(arb\_t im, const acb\_t z)
  \[\text{Sets}\ im\ \text{to the imaginary part of}\ z.\]

void \texttt{acb\_arg}(arb\_t r, const acb\_t z, slong prec)
  \[\text{Sets}\ r\ \text{to a real interval containing the complex argument (phase) of}\ z.\ \text{We define the complex argument have a discontinuity on}\ (-\infty, 0],\ \text{with the special value}\ arg(0) = 0,\ \text{and}\ arg(a + 0i) = \pi\ \text{for}\ a < 0. Equivalently, if}\ z = a + bi,\ \text{the argument is given by}\ \texttt{atan2}(b, a)\ \text{(see}\ \texttt{arb\_atan2()}.\]

void \texttt{acb\_abs}(arb\_t r, const acb\_t z, slong prec)
  \[\text{Sets}\ r\ \text{to the absolute value of}\ z.\]

void \texttt{acb\_sgn}(arb\_t r, const acb\_t z, slong prec)
  \[\text{Sets}\ r\ \text{to the complex sign of}\ z,\ \text{defined as} 0\ \text{if}\ z\ \text{is exactly zero and the projection onto the unit circle}\ z/|z| = \exp(i\arg(z))\ \text{otherwise.}\]

void \texttt{acb\_csgn}(arb\_t r, const acb\_t z)
  \[\text{Sets}\ r\ \text{to the extension of the real sign function taking the value 1 for}\ z\ \text{strictly in the right half plane, -1 for}\ z\ \text{strictly in the left half plane, and the sign of the imaginary part when}\ z\ \text{is on the imaginary axis. Equivalently,}\ csgn(z) = z/\sqrt{z^2}\ \text{except that the value is 0 when}\ z\ \text{is exactly zero.}\]

4.2.8 Arithmetic

void \texttt{acb\_neg}(acb\_t z, const acb\_t x)
  \[\text{Sets}\ z\ \text{to the negation of}\ x.\]

void \texttt{acb\_conj}(acb\_t z, const acb\_t x)
  \[\text{Sets}\ z\ \text{to the complex conjugate of}\ x.\]

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

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

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

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

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

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

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

void \texttt{acb\_sub\_arb}(acb\_t z, const acb\_t x, const arb\_t y, slong prec)
  \[\text{Sets}\ z\ \text{to the difference of}\ x\ \text{and}\ y.}\]
void acb_mul_onei(acb_t z, const acb_t x)
    Sets z to x multiplied by the imaginary unit.

void acb_div_onei(acb_t z, const acb_t x)
    Sets z to x divided by the imaginary unit.

void acb_mul_ui(acb_t z, const acb_t x, ulong y, slong prec)
void acb_mul_si(acb_t z, const acb_t x, slong y, slong prec)
void acb_mul_fmpz(acb_t z, const acb_t x, const fmpz_t y, slong prec)
void acb_mul_arb(acb_t z, const acb_t x, const arb_t y, slong prec)
    Sets z to the product of x and y.

void acb_mul(acb_t z, const acb_t x, const acb_t y, slong prec)
    Sets z to the product of x and y. If at least one part of x or y is zero, the operations is reduced to two real multiplications. If x and y are the same pointers, they are assumed to represent the same mathematical quantity and the squaring formula is used.

void acb_mul_2exp_si(acb_t z, const acb_t x, slong e)
void acb_mul_2exp_fmpz(acb_t z, const acb_t x, const fmpz_t e)
    Sets z to x multiplied by $2^e$, without rounding.

void acb_sqr(acb_t z, const acb_t x, slong prec)
    Sets z to x squared.

void acb_cube(acb_t z, const acb_t x, slong prec)
    Sets z to x cubed, computed efficiently using two real squarings, two real multiplications, and scalar operations.

void acb_addmul(acb_t z, const acb_t x, const acb_t y, slong prec)
void acb_addmul_ui(acb_t z, const acb_t x, ulong y, slong prec)
void acb_addmul_si(acb_t z, const acb_t x, slong y, slong prec)
void acb_addmul_fmpz(acb_t z, const acb_t x, const fmpz_t y, slong prec)
void acb_addmul_arb(acb_t z, const acb_t x, const arb_t y, slong prec)
    Sets z to z plus the product of x and y.

void acb_submul(acb_t z, const acb_t x, const acb_t y, slong prec)
void acb_submul_ui(acb_t z, const acb_t x, ulong y, slong prec)
void acb_submul_si(acb_t z, const acb_t x, slong y, slong prec)
void acb_submul_fmpz(acb_t z, const acb_t x, const fmpz_t y, slong prec)
void acb_submul_arb(acb_t z, const acb_t x, const arb_t y, slong prec)
    Sets z to z minus the product of x and y.

void acb_inv(acb_t z, const acb_t x, slong prec)
    Sets z to the multiplicative inverse of x.

void acb_div_ui(acb_t z, const acb_t x, ulong y, slong prec)
void acb_div_si(acb_t z, const acb_t x, slong y, slong prec)
void acb_div_fmpz(acb_t z, const acb_t x, const fmpz_t y, slong prec)
void acb_div_arb(acb_t z, const acb_t x, const arb_t y, slong prec)
void acb_div(acb_t z, const acb_t x, const acb_t y, slong prec)
    Sets z to the quotient of x and y.
4.2.9 Mathematical constants

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

4.2.10 Powers and roots

```c
void acb_sqrt(acb_t r, const acb_t z, slong prec)
  Sets r to the square root of z. If either the real or imaginary part is exactly zero, only a single real square root is needed. Generally, we use the formula \( \sqrt{a + bi} = u/2 + ib/u, u = \sqrt{2(|a + bi| + a)}, \) requiring two real square root extractions.

void acb_rsqrt(acb_t r, const acb_t z, slong prec)
  Sets r to the reciprocal square root of z. If either the real or imaginary part is exactly zero, only a single real reciprocal square root is needed. Generally, we use the formula \( 1/\sqrt{a + bi} = ((a + r) - bi)/v, r = |a + bi|, v = \sqrt{r(a + bi + r)^2}, \) requiring one real square root and one real reciprocal square root.

void acb_quadratic_roots_fmpz(acb_t r1, acb_t r2, const fmpz_t a, const fmpz_t b, const fmpz_t c, slong prec)
  Sets r1 and r2 to the roots of the quadratic polynomial ax^2 + bx + c. Requires that a is nonzero. This function is implemented so that both roots are computed accurately even when direct use of the quadratic formula would lose accuracy.

void acb_root_ui(acb_t r, const acb_t z, ulong k, slong prec)
  Sets r to the principal k-th root of z.

void acb_pow_fmpz(acb_t y, const acb_t b, const fmpz_t e, slong prec)

void acb_pow_ui(acb_t y, const acb_t b, ulong e, slong prec)

void acb_pow_si(acb_t y, const acb_t b, ulong 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 = a^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_unit_root(acb_t res, ulong order, slong prec)
  Sets res to \( \exp\left(\frac{2\pi i}{\text{order}}\right) \) to precision prec.
```

4.2.11 Exponentials and logarithms

```c
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).

void acb_exp_invexp(acb_t s, acb_t t, const acb_t z, slong prec)
  Sets v = exp(z) and w = exp(−z).

void acb_log(acb_t y, const acb_t z, slong prec)
  Sets y to the principal branch of the natural logarithm of z, computed as log(a + bi) = \( \frac{1}{2} \log(a^2 + b^2) + i \arg(a + bi) \).

void acb_loglp(acb_t z, const acb_t x, slong prec)
  Sets z = log(1 + x), computed accurately when x ≈ 0.
```
4.2.12 Trigonometric functions

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

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

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

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

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

4.2.13 Inverse trigonometric functions

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

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

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

4.2.14 Hyperbolic functions

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

void \texttt{acb\_sinh\_cosh}(acb\_t s, acb\_t c, const acb\_t z, slong prec)
void \texttt{acb\_tanh}(acb\_t s, const acb\_t z, slong prec)
void \texttt{acb\_coth}(acb\_t s, const acb\_t z, slong prec)
Respectively computes \( \sinh(z) = -i \sin(i z) \), \( \cosh(z) = \cos(i z) \), \( \tanh(z) = -i \tan(i z) \), \( \coth(z) = i \cot(i z) \).
4.2.15 Inverse hyperbolic functions

void acb_asinh(acb_t res, const acb_t z, slong prec)
Sets res to \( \text{asinh}(z) = -\text{asin}(iz) \).

void acb_acosh(acb_t res, const acb_t z, slong prec)
Sets res to \( \text{acosh}(z) = \log(z + \sqrt{z^2 - 1}) \).

void acb_atanh(acb_t res, const acb_t z, slong prec)
Sets res to \( \text{atanh}(z) = -i \text{atan}(iz) \).

4.2.16 Rising factorials

void acb_rising_ui_bs(acb_t z, const acb_t x, ulong 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_rec(acb_t z, const acb_t x, ulong n, slong prec)

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

void acb_rising_ui_get_mag(mag_t bound, const acb_t x, ulong n)
Computes an upper bound for the absolute value of the rising factorial \( z = x(x+1)(x+2)\cdots(x+n-1) \). Not currently optimized for large \( n \).

4.2.17 Gamma function

void acb_gamma(acb_t y, const acb_t x, slong prec)
Computes the gamma function \( y = \Gamma(x) \).

void acb_rgamma(acb_t y, const acb_t x, slong prec)
Computes the reciprocal gamma function \( y = 1/\Gamma(x) \), avoiding division by zero at the poles of the gamma function.

void acb_lgamma(acb_t y, const acb_t x, slong prec)
Computes the logarithmic gamma function \( y = \log \Gamma(x) \).

The branch cut of the logarithmic gamma function is placed on the negative half-axis, which means that \( \log \Gamma(z) + \log z = \log \Gamma(z+1) \) holds for all \( z \), whereas \( \log \Gamma(z) \neq \log \Gamma(z) \) in general. In the left half plane, the reflection formula with correct branch structure is evaluated via \( \text{acb_log_sin_pi()} \).

void acb_digamma(acb_t y, const acb_t x, slong prec)
Computes the digamma function \( y = \psi(x) = (\log \Gamma(x))' = \Gamma'(x)/\Gamma(x) \).
void \texttt{acb\_log\_sin\_pi}(\texttt{acb\_t res}, \texttt{const acb\_t z}, \texttt{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,

\[
S(z) = \log(\sin(\pi(z - n))) \mp n\pi i, \quad n = \lfloor \text{re}(z) \rfloor
\]

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 \(0 \leq \text{re}(z) < 1\). In this strip, we use use \(S(z) = \log(\sin(\pi(z)))\) if the imaginary part of \(z\) is small. Otherwise, we use \(S(z) = i\pi(z - 1/2) + \log((1 + e^{-2\pi i})/2)\) 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 \texttt{acb\_polygamma}(\texttt{acb\_t z}, \texttt{const acb\_t s}, \texttt{const acb\_t z}, \texttt{slong prec})

Sets \(\text{res}\) to the value of the generalized polygamma function \(\psi(s, z)\).

If \(s\) is a nonnegative order, this is simply the \(s\)-order derivative of the digamma function. If \(s = 0\), this function simply calls the digamma function internally. For integers \(s \geq 1\), it calls the Hurwitz zeta function. Note that for small integers \(s \geq 1\), it can be faster to use \texttt{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 \texttt{acb\_barnes\_g}(\texttt{acb\_t res}, \texttt{const acb\_t z}, \texttt{slong prec})
void \texttt{acb\_log\_barnes\_g}(\texttt{acb\_t res}, \texttt{const acb\_t z}, \texttt{slong prec})

Computes Barnes \(G\)-function or the logarithmic Barnes \(G\)-function, respectively. The logarithmic version has branch cuts on the negative real axis and is continuous elsewhere in the complex plane, in analogy with the logarithmic gamma function. The functional equation

\[
\log G(z + 1) = \log \Gamma(z) + \log G(z).
\]

holds for all \(z\).

For small integers, we directly use the recurrence relation \(G(z + 1) = \Gamma(z)G(z)\) together with the initial value \(G(1) = 1\). For general \(z\), we use the formula

\[
\log G(z) = (z - 1) \log \Gamma(z) - \zeta'(-1, z) + \zeta'(-1).
\]

### 4.2.18 Zeta function

void \texttt{acb\_zeta}(\texttt{acb\_t z}, \texttt{const acb\_t s}, \texttt{slong prec})

Sets \(z\) to the value of the Riemann zeta function \(\zeta(s)\). Note: for computing derivatives with respect to \(s\), use \texttt{acb\_poly\_zeta\_series()} or related methods.

void \texttt{acb\_hurwitz\_zeta}(\texttt{acb\_t z}, \texttt{const acb\_t s}, \texttt{const acb\_t a}, \texttt{slong prec})

Sets \(z\) to the value of the Hurwitz zeta function \(\zeta(s,a)\). Note: for computing derivatives with respect to \(s\), use \texttt{acb\_poly\_zeta\_series()} or related methods.
void acb_bernoulli_poly_ui(acb_t res, ulong n, const acb_t x, slong prec)
Sets res to the value of the Bernoulli polynomial \( B_n(x) \).

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

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

### 4.2.19 Polylogarithms

void acb_polylog(acb_t w, const acb_t s, const acb_t z, slong prec)

void acb_polylog_si(acb_t w, slong s, const acb_t z, slong prec)

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

### 4.2.20 Arithmetic-geometric mean

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

void acb_agm1(acb_t m, const acb_t z, slong prec)

Sets \( m \) to the arithmetic-geometric mean \( M(z) = \text{agm}(1, z) \), defined such that the function is continuous in the complex plane except for a branch cut along the negative half axis (where it is continuous from above). This corresponds to always choosing an “optimal” branch for the square root in the arithmetic-geometric mean iteration.

void acb_agm1_cpx(acb_ptr m, const acb_t z, slong len, slong prec)

Sets the coefficients in the array \( m \) to the power series expansion of the arithmetic-geometric mean at the point \( z \) truncated to length \( len \), i.e. \( M(z + x) \in \mathbb{C}[[x]] \).

### 4.2.21 Other special functions

void acb_chebyshev_t_ui(acb_t a, ulong n, const acb_t x, slong prec)

void acb_chebyshev_u_ui(acb_t a, ulong n, const acb_t x, slong prec)

Evaluates the Chebyshev polynomial of the first kind \( a = T_n(x) \) or the Chebyshev polynomial of the second kind \( a = U_n(x) \).

void acb_chebyshev_t2_ui(acb_t a, acb_t b, ulong n, const acb_t x, slong prec)

void acb_chebyshev_u2_ui(acb_t a, acb_t b, ulong n, const acb_t x, slong prec)

Simultaneously evaluates \( a = T_n(x) \), \( b = T_{n-1}(x) \) or \( a = U_n(x) \), \( b = U_{n-1}(x) \). Aliasing between \( a \), \( b \) and \( x \) is not permitted.

### 4.2.22 Vector functions

void _acb_vec_zero(acb_ptr A, slong n)

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

int _acb_vec_is_zero(acb_srcptr vec, slong len)

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

int _acb_vec_is_real(acb_srcptr v, slong len)

Returns nonzero iff all entries in \( x \) have zero imaginary part.

void _acb_vec_set(acb_ptr res, acb_srcptr vec, slong len)

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

void _acb_vec_set_round(acb_ptr res, acb_srcptr vec, slong len, slong prec)

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

void _acb_vec_neg(acb_ptr res, acb_srcptr vec, slong len)
void _acb_vec_add(acb_ptr res, acb_srcptr vec1, acb_srcptr vec2, slong len, slong prec)
void _acb_vec_sub(acb_ptr res, acb_srcptr vec1, acb_srcptr vec2, slong len, slong prec)
void _acb_vec_scalar_submul(acb_ptr res, acb_ptr vec1, acb_ptr vec2, slong len, const acb_t c, slong prec)
void _acb_vec_scalar_addmul(acb_ptr res, acb_ptr vec, slong len, const acb_t c, slong prec)
void _acb_vec_scalar_mul(acb_ptr res, acb_ptr vec, slong len, const acb_t c, slong prec)
void _acb_vec_scalar_mul_ui(acb_ptr res, acb_ptr vec, slong len, ulong c, slong prec)
void _acb_vec_scalar_mul_2exp_si(acb_ptr res, acb_ptr vec, slong len, slong c)
void _acb_vec_scalar_mul_onei(acb_ptr res, acb_ptr vec, slong len)
void _acb_vec_scalar_div_ui(acb_ptr res, acb_ptr vec, slong len, ulong c, slong prec)
void _acb_vec_scalar_div(acb_ptr res, acb_ptr vec, slong len, const acb_t c, slong prec)
void _acb_vec_scalar_mul_arb(acb_ptr res, acb_ptr vec, slong len, const arb_t c, slong prec)
void _acb_vec_scalar_div_arb(acb_ptr res, acb_ptr vec, slong len, const arb_t c, slong prec)
void _acb_vec_scalar_mul_fmpz(acb_ptr res, acb_ptr vec, slong len, const fmpz_t c, slong prec)
void _acb_vec_scalar_div_fmpz(acb_ptr res, acb_ptr 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, ..., x^{len-1}.

void _acb_vec_unit_roots(acb_ptr z, slong order, slong prec)
Sets z to the powers 1, z, z^2, ..., z^{order-1} where z = \exp(\frac{2\pi i}{\text{order}}) to precision prec.

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.

4.2. acb.h – complex numbers

71
These modules implement dense univariate polynomials with real and complex coefficients. Truncated power series are supported via methods acting on polynomials, without introducing a separate power series type.

5.1 arb_poly.h – polynomials over the real numbers

An `arb_poly_t` represents a polynomial over the real numbers, implemented as an array of coefficients of type `arb_struct`.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

5.1.1 Types, macros and constants

`arb_poly_struct` `arb_poly_t`

Contains a pointer to an array of coefficients (coeffs), the used length (length), and the allocated size of the array (alloc).

An `arb_poly_t` is defined as an array of length one of type `arb_poly_struct`, permitting an `arb_poly_t` to be passed by reference.

5.1.2 Memory management

```c
void arb_poly_init(arb_poly_t poly)
Initializes the polynomial for use, setting it to the zero polynomial.

void arb_poly_clear(arb_poly_t poly)
Clears the polynomial, deallocating all coefficients and the coefficient array.

void arb_poly_fit_length(arb_poly_t poly, slong len)
Makes sure that the coefficient array of the polynomial contains at least len initialized coefficients.

void _arb_poly_set_length(arb_poly_t poly, slong len)
Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void _arb_poly_normalise(arb_poly_t poly)
Strips any trailing coefficients which are identical to zero.

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

\section*{5.1.3 Basic manipulation}

\begin{verbatim}
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.
\end{verbatim}
5.1.4 Conversions

void \texttt{arb\_poly\_set\_fmpz\_poly}(\texttt{arb\_poly\_t poly}, \texttt{const fmpz\_poly\_t src}, \texttt{slong prec})

void \texttt{arb\_poly\_set\_fmpq\_poly}(\texttt{arb\_poly\_t poly}, \texttt{const fmpq\_poly\_t src}, \texttt{slong prec})

void \texttt{arb\_poly\_set\_si}(\texttt{arb\_poly\_t poly}, \texttt{slong src})

Sets \texttt{poly} to \texttt{src}, rounding the coefficients to \texttt{prec} bits.

5.1.5 Input and output

void \texttt{arb\_poly\_printd}(\texttt{const arb\_poly\_t poly}, \texttt{slong digits})

Prints the polynomial as an array of coefficients, printing each coefficient using \texttt{arb\_printd}.

void \texttt{arb\_poly\_fprintd}(\texttt{FILE \* file}, \texttt{const arb\_poly\_t poly}, \texttt{slong digits})

Prints the polynomial as an array of coefficients to the stream \texttt{file}, printing each coefficient using \texttt{arb\_fprintd}.

5.1.6 Random generation

void \texttt{arb\_poly\_randtest}(\texttt{arb\_poly\_t poly}, \texttt{flint\_rand\_t state}, \texttt{slong len}, \texttt{slong prec}, \texttt{slong mag\_bits})

Creates a random polynomial with length at most \texttt{len}.

5.1.7 Comparisons

int \texttt{arb\_poly\_contains}(\texttt{const arb\_poly\_t poly1}, \texttt{const arb\_poly\_t poly2})

int \texttt{arb\_poly\_contains\_fmpz\_poly}(\texttt{const arb\_poly\_t poly1}, \texttt{const fmpz\_poly\_t poly2})

int \texttt{arb\_poly\_contains\_fmpq\_poly}(\texttt{const arb\_poly\_t poly1}, \texttt{const fmpq\_poly\_t poly2})

Returns nonzero iff \texttt{poly1} contains \texttt{poly2}.

int \texttt{arb\_poly\_equal}(\texttt{const arb\_poly\_t A}, \texttt{const arb\_poly\_t B})

Returns nonzero iff \texttt{A} and \texttt{B} are equal as polynomial balls, i.e. all coefficients have equal midpoint and radius.

int \texttt{arb\_poly\_overlaps}(\texttt{arb\_srcptr poly1}, \texttt{slong len1}, \texttt{arb\_srcptr poly2}, \texttt{slong len2})

int \texttt{arb\_poly\_overlaps}(\texttt{const arb\_poly\_t poly1}, \texttt{const arb\_poly\_t poly2})

Returns nonzero iff \texttt{poly1} overlaps with \texttt{poly2}. The underscore function requires that \texttt{len1} ist at least as large as \texttt{len2}.

int \texttt{arb\_poly\_get\_unique\_fmpz\_poly}(\texttt{fmpz\_poly\_t z}, \texttt{const arb\_poly\_t x})

If \texttt{x} contains a unique integer polynomial, sets \texttt{z} to that value and returns nonzero. Otherwise (if \texttt{x} represents no integers or more than one integer), returns zero, possibly partially modifying \texttt{z}.

5.1.8 Bounds

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

void \texttt{arb\_poly\_majorant}(\texttt{arb\_poly\_t res}, \texttt{const arb\_poly\_t poly}, \texttt{slong prec})

Sets \texttt{res} to an exact real polynomial whose coefficients are upper bounds for the absolute values of the coefficients in \texttt{poly}, rounded to \texttt{prec} bits.
5.1.9 Arithmetic

void _arb_poly_add (arb_ptr C, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong prec)
Sets \( \{C, \max(\text{lenA}, \text{lenB})\} \) to the sum of \( \{A, \text{lenA}\} \) and \( \{B, \text{lenB}\} \). 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 \( \{C, \max(\text{lenA}, \text{lenB})\} \) to the difference of \( \{A, \text{lenA}\} \) and \( \{B, \text{lenB}\} \). 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 \( \{A, \text{lenA}\} \) and \( \{B, \text{lenB}\} \), truncated to length \( n \). The output is not allowed to be aliased with either of the inputs. We require \( \text{lenA} \geq \text{lenB} > 0, n > 0 \), \( \text{lenA} + \text{lenB} - 1 \geq n \).

The classical version uses a plain loop. 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.

```c
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, prec)
void _arb_poly_mullow_block(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong n, 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 \( n \).

```c
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.

```c
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 \( A \) truncated to length \( n \).

```c
void _arb_poly_inv_series(arb_ptr Q, arb_srcptr A, slong Alen, slong len, slong prec)
void _arb_poly_inv_series(arb_poly_t Q, const arb_poly_t A, slong n, slong prec)
void _arb_poly_div_series(arb_ptr Q, arb_srcptr A, slong Alen, arb_srcptr B, slong Blen, slong n, slong prec)
void _arb_poly_div_series(arb_poly_t Q, const arb_poly_t A, const arb_poly_t B, slong n, slong prec)
```
Sets \( \{ Q, len \} \) to the power series inverse of \( \{ A, Alen \} \). Uses Newton iteration.

Sets \( Q \) to the power series inverse of \( A \), truncated to length \( n \).

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

```c
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)
```
Performs polynomial division with remainder, computing a quotient \( Q \) and a remainder \( R \) such that \( A = BQ + R \). The implementation reverses the inputs and performs power series division.

If the leading coefficient of \( B \) contains zero (or if \( B \) is identically zero), returns 0 indicating failure without modifying the outputs. Otherwise returns nonzero.

```c
int _arb_poly_divrem(arb_poly_t Q, arb_poly_t R, const arb_poly_t A, const arb_poly_t B, slong prec)
```
Divides \( A \) by the polynomial \( x - c \), computing the quotient \( Q \) as well as the remainder \( R = f(c) \).

5.1. `arb_poly.h` – polynomials over the real numbers
5.1.10 Composition

void _arb_poly_taylor_shift_horner(arb_ptr g, const arb_t c, slong n, slong prec)
void arb_poly_taylor_shift_horner(arb_poly_t g, const arb_poly_t f, const arb_t c, slong prec)
void _arb_poly_taylor_shift_divconquer(arb_ptr g, const arb_t c, slong n, slong prec)
void arb_poly_taylor_shift_divconquer(arb_poly_t g, const arb_poly_t f, const arb_t c, slong prec)
void _arb_poly_taylor_shift_convolution(arb_ptr g, const arb_t c, slong n, slong prec)
void arb_poly_taylor_shift_convolution(arb_poly_t g, const arb_poly_t f, const arb_t c, slong prec)
void _arb_poly_taylor_shift(arb_ptr g, const arb_t c, slong n, slong prec)

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 \( \text{res} \) to the composition \( h(x) = f(g(x)) \) where \( f \) is given by \( \text{poly1} \) and \( g \) is given by \( \text{poly2} \), respectively using Horner’s rule, divide-and-conquer, and an automatic choice between the two algorithms.

The default algorithm also handles special-form input \( g = ax^n + c \) efficiently by performing a Taylor shift followed by a rescaling.

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

We require that the constant term in \( g(x) \) is exactly zero. The underscore methods do not support aliasing of the output with either input polynomial.

```c
void _arb_poly_revert_series_lagrange(arb_ptr h, arb_srcptr f, slong flen, slong n, slong prec)
void arb_poly_revert_series_lagrange(arb_poly_t h, const arb_poly_t f, slong n, slong prec)
void _arb_poly_revert_series_lagrange_fast(arb_ptr h, arb_srcptr f, slong flen, slong n, slong prec)
void arb_poly_revert_series_lagrange_fast(arb_poly_t h, const arb_poly_t f, slong n, slong prec)
void _arb_poly_revert_series(arb_ptr h, arb_srcptr f, slong flen, slong n, slong prec)
void arb_poly_revert_series(arb_poly_t h, const arb_poly_t f, slong n, slong prec)
```

Sets \( h \) to the power series reversion of \( f \), i.e. the expansion of the compositional inverse function \( f^{-1}(x) \), truncated to order \( O(x^n) \), using respectively Lagrange inversion, Newton iteration, fast Lagrange inversion, and a default algorithm choice.

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

### 5.1.11 Evaluation

```c
void _arb_poly_evaluate_horner(arb_t y, arb_srcptr f, slong len, const arb_t x, slong prec)
void arb_poly_evaluate_horner(arb_t y, const arb_poly_t f, const arb_t x, slong prec)
void _arb_poly_evaluate_rectangular(arb_t y, arb_srcptr f, slong len, const arb_t x, slong prec)
void arb_poly_evaluate_rectangular(arb_t y, const arb_poly_t f, const arb_t x, slong prec)
void _arb_poly_evaluate(acb_t y, arb_srcptr f, slong len, const acb_t x, slong prec)
void arb_poly_evaluate(acb_t y, const arb_poly_t f, const acb_t x, slong prec)
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)
```

5.1. arb_poly.h – polynomials over the real numbers
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(const arb_t y, arb_t z, const arb_poly_t f, const arb_t x, slong prec)

Sets \( y = f(x), z = f'(x) \), evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

When Horner's rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.

void _arb_poly_evaluate2_acb_horner(acb_t y, acb_t z, arb_srcptr f, slong len, const acb_t x, slong prec)

void arb_poly_evaluate2_acb_horner(acb_t y, acb_t z, const arb_poly_t f, const acb_t x, slong prec)

void _arb_poly_evaluate2_acb_rectangular(acb_t y, acb_t z, arb_srcptr f, slong len, const acb_t x, slong prec)

void arb_poly_evaluate2_acb_rectangular(acb_t y, acb_t z, const arb_poly_t f, const acb_t x, slong prec)

void _arb_poly_evaluate2_acb(acb_t y, acb_t z, arb_srcptr f, slong len, const acb_t x, slong prec)

void arb_poly_evaluate2_acb(acb_t y, acb_t z, const arb_poly_t f, const acb_t x, slong prec)

Sets \( y = f(x), z = f'(x) \), evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

5.1.12 Product trees

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

void arb_poly_product_roots(arb_poly_t poly, arb_srcptr xs, slong n, slong prec)

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

arb_ptr * _arb_poly_tree_alloc(slong len)

Returns an initialized data structured capable of representing a remainder tree (product tree) of \( n \) 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 \( n \) roots. The tree structure must be pre-allocated to the specified length using _arb_poly_tree_alloc().

5.1.13 Multipoint evaluation

void _arb_poly_evaluate_vec_iter(arb_ptr ys, arb_srcptr poly, slong plen, arb_srcptr xs, slong n, slong prec)

void arb_poly_evaluate_vec_iter(arb_ptr ys, const arb_poly_t poly, arb_srcptr xs, slong n, slong prec)

Evaluates the polynomial simultaneously at \( n \) given points, calling _arb_poly_evaluate() repeatedly.

void _arb_poly_evaluate_vec_fast_precomp(arb_ptr vs, arb_srcptr poly, slong plen, arb_ptr * tree, slong len, slong prec)
void _arb_poly_evaluate_vec_fast(\(\text{arb_ptr } ys, \text{arb_srcptr } poly, \text{slong } plen, \text{arb_srcptr } xs, \text{slong } n, \text{slong } prec\))

void arb_poly_evaluate_vec_fast(\(\text{arb_ptr } ys, \text{const arbi_pol} \text{y_t } poly, \text{arb_srcptr } xs, \text{slong } n, \text{slong } prec\))

Evaluates the polynomial simultaneously at \(n\) given points, using fast multipoint evaluation.

5.1.14 Interpolation

void _arb_poly_interpolate_newton(\(\text{arb_ptr } poly, \text{arb_srcptr } xs, \text{arb_srcptr } ys, \text{slong } n, \text{slong } prec\))

void arb_poly_interpolate_newton(\(\text{arb_poly_t } poly, \text{arb_srcptr } xs, \text{slong } n, \text{slong } prec\))

Recover the unique polynomial of length at most \(n\) that interpolates the given \(x\) and \(y\) values.

This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

void _arb_poly_interpolate_barycentric(\(\text{arb_ptr } poly, \text{arb_srcptr } xs, \text{arb_srcptr } ys, \text{slong } n, \text{slong } prec\))

void arb_poly_interpolate_barycentric(\(\text{arb_poly_t } poly, \text{arb_srcptr } xs, \text{arb_srcptr } ys, \text{slong } n, \text{slong } prec\))

Recover the unique polynomial of length at most \(n\) that interpolates the given \(x\) and \(y\) values.

This implementation uses the barycentric form of Lagrange interpolation.

void _arb_poly_interpolation_weights(\(\text{arb_ptr } w, \text{arb_ptr } * \text{tree, slong } len, \text{slong } prec\))

void _arb_poly_interpolate_fast_precomp(\(\text{arb_ptr } poly, \text{arb_srcptr } ys, \text{arb_ptr } * \text{tree, arb_srcptr } poly, \text{slong } len, \text{slong } prec\))

void _arb_poly_interpolate_fast(\(\text{arb_ptr } poly, \text{arb_srcptr } xs, \text{arb_srcptr } ys, \text{slong } len, \text{slong } prec\))

void arb_poly_interpolate_fast(\(\text{arb_poly_t } poly, \text{arb_srcptr } xs, \text{arb_srcptr } ys, \text{slong } n, \text{slong } prec\))

Recover the unique polynomial of length at most \(n\) that interpolates the given \(x\) and \(y\) values, using fast Lagrange interpolation. The precomp function takes a precomputed product tree over the \(x\) values and a vector of interpolation weights as additional inputs.

5.1.15 Differentiation

void _arb_poly_derivative(\(\text{arb_ptr } res, \text{arb_srcptr } poly, \text{slong } len, \text{slong } prec\))

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

void arb_poly_derivative(\(\text{arb_poly_t } res, \text{const arbi_pol} \text{y_t } poly, \text{slong } prec\))

Sets \(res\) to the derivative of \(poly\).

void _arb_poly_integral(\(\text{arb_ptr } res, \text{arb_srcptr } poly, \text{slong } len, \text{slong } prec\))

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

void arb_poly_integral(\(\text{arb_poly_t } res, \text{const arbi_pol} \text{y_t } poly, \text{slong } prec\))

Sets \(res\) to the integral of \(poly\).

5.1.16 Transforms

void _arb_poly_borel_transform(\(\text{arb_ptr } res, \text{arb_srcptr } poly, \text{slong } len, \text{slong } prec\))

void arb_poly_borel_transform(\(\text{arb_poly_t } res, \text{const arbi_pol} \text{y_t } poly, \text{slong } prec\))

Computes the Borel transform of the input polynomial, mapping \(\sum_k a_k x^k\) to \(\sum_k \frac{a_k}{k!} x^k\). The underscore method allows aliasing.

void _arb_poly_inv_borel_transform(\(\text{arb_ptr } res, \text{arb_srcptr } poly, \text{slong } len, \text{slong } prec\))
void \texttt{arb\_poly\_inv\_borel\_transform}(\texttt{arb\_poly\_t res}, \texttt{const arb\_poly\_t poly}, \texttt{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 \_\texttt{arb\_poly\_binomial\_transform\_basecase}(\texttt{arb\_ptr b}, \texttt{arb\_srcptr a}, \texttt{slong alen}, \texttt{slong len},
\texttt{slong prec})
void \texttt{arb\_poly\_binomial\_transform\_basecase}(\texttt{arb\_poly\_t b}, \texttt{const arb\_poly\_t a}, \texttt{slong len},
\texttt{slong prec})
void \_\texttt{arb\_poly\_binomial\_transform\_convolution}(\texttt{arb\_ptr b}, \texttt{arb\_srcptr a}, \texttt{slong alen},
\texttt{slong len}, \texttt{slong prec})
void \texttt{arb\_poly\_binomial\_transform\_convolution}(\texttt{arb\_poly\_t b}, \texttt{const arb\_poly\_t a}, \texttt{slong len},
\texttt{slong prec})

void \texttt{arb\_poly\_binomial\_transform}(\texttt{arb\_poly\_t b}, \texttt{const arb\_poly\_t a}, \texttt{slong len}, \texttt{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 \( b_k = \sum_{a=0}^n (-1)^a \binom{n}{a} a_k \). The binomial transform is equivalent to the power series composition \( f(x) \rightarrow (1 - x)^{-1} f(x/(x - 1)) \), and is its own inverse.

The \texttt{basecase} version evaluates coefficients one by one from the definition, generating the binomial coefficients by a recurrence relation.

The \texttt{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.

\subsection*{5.1.17 Powers and elementary functions}

void \_\texttt{arb\_poly\_pow\_ui\_trunc\_binexp}(\texttt{arb\_ptr res}, \texttt{arb\_srcptr f}, \texttt{slong flen}, \texttt{ulong exp}, \texttt{slong len},
\texttt{slong prec})
Sets \{res, len\} to \{f, flen\} raised to the power \texttt{exp}, truncated to length \texttt{len}. Requires that \texttt{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 \texttt{flen} and \texttt{len} are positive. Uses binary exponentiation.

void \texttt{arb\_poly\_pow\_ui\_trunc\_binexp}(\texttt{arb\_poly\_t res}, \texttt{const arb\_poly\_t poly}, \texttt{ulong exp},
\texttt{slong len}, \texttt{slong prec})
Sets \texttt{res} to \texttt{poly} raised to the power \texttt{exp}, truncated to length \texttt{len}. Uses binary exponentiation.

void \_\texttt{arb\_poly\_pow\_ui}(\texttt{arb\_ptr res}, \texttt{arb\_srcptr f}, \texttt{slong flen}, \texttt{ulong exp}, \texttt{slong prec})
Sets \texttt{res} to \{f, \texttt{flen}\} raised to the power \texttt{exp}. Does not support aliasing of the input and output, and requires that \texttt{flen} is positive.

void \texttt{arb\_poly\_pow\_ui}(\texttt{arb\_poly\_t res}, \texttt{const arb\_poly\_t poly}, \texttt{ulong exp}, \texttt{slong prec})
Sets \texttt{res} to \texttt{poly} raised to the power \texttt{exp}.

void \_\texttt{arb\_poly\_pow\_series}(\texttt{arb\_ptr h}, \texttt{arb\_srcptr f}, \texttt{slong flen}, \texttt{arb\_srcptr g}, \texttt{slong glen},
\texttt{slong len}, \texttt{slong prec})
Sets \{h, \texttt{len}\} to the power series \( f(x)^g(x) = \exp(g(x) \log f(x)) \) truncated to length \texttt{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 \texttt{flen} and \texttt{glen} do not exceed \texttt{len}.

void \texttt{arb\_poly\_pow\_series}(\texttt{arb\_poly\_t h}, \texttt{const arb\_poly\_t f}, \texttt{const arb\_poly\_t g}, \texttt{slong len},
\texttt{slong prec})
Sets h to the power series \( f(x)^g(x) = \exp(g(x) \log f(x)) \) truncated to length \texttt{len}. This function
detects special cases such as $g$ being an exact small integer or $\pm 1/2$, and computes such powers more efficiently.

```c
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 $len$.

```c
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 $len$.

```c
void _arb_poly_sqrt_series(arb_ptr g, arb_srcptr h, slong hlen, 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.

```c
void arb_poly_sqrt_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.

```c
void _arb_poly_rsqrt_series(arb_ptr g, arb_srcptr h, slong hlen, 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.

```c
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 supports aliasing of the input and output arrays. It requires that $flen$ and $n$ are greater than zero.

```c
void _arb_poly_log_series(arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec)
```

Sets $res$ to the power series logarithm of $f$, truncated to length $n$. Uses the formula $\log(f(x)) = \int (x' / f(x)) dx$, 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.

```c
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 $\log(f(x)) = \int (x' / f(x)) dx$, 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.

```c
void _arb_poly_atan_series(arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec)
```

Sets $res$ respectively to the power series inverse tangent, inverse sine and inverse cosine of $f$, truncated to length $n$.

Uses the formulas

$$
\tan^{-1}(f(x)) = \int \frac{x'}{1 + f(x)^2} dx,
$$

$$
\sin^{-1}(f(x)) = \int \frac{x'}{(1 - f(x)^2)^{1/2}} dx,
$$

$$
\cos^{-1}(f(x)) = -\int \frac{x'}{(1 - f(x)^2)^{1/2}} dx,
$$

adding the inverse function of the constant term in $f$ as the constant of integration.

The underscore method supports aliasing of the input and output arrays. They require that $flen$ and $n$ are greater than zero.

5.1. arb_poly.h – polynomials over the real numbers 83
Chapter 5. Polynomials and power series

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.

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 $\pi$.

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

Respectively evaluates the power series sine or cosine. These functions simply wrap \_arb_poly_sin_series(). The underscore methods support aliasing and require the lengths to be nonzero.

Sets $g$ to the power series tangent of $h$. 

\begin{verbatim}
void _arb_poly_exp_series_basecase(const arb_poly_t h, slong n, slong prec)
void _arb_poly_exp_series(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_ptr f, arb_srcptr h, slong hlen, slong n, slong prec)

void _arb_poly_sin_series_basecase(const arb_poly_t h, slong n, slong prec)
void _arb_poly_sin_series(const arb_poly_t h, slong n, slong prec)
void _arb_poly_sin_series(arb_ptr s, arb_ptr c, arb_srcptr h, slong hlen, slong n, slong prec)
void arb_poly_sin_series(arb_ptr s, arb_ptr c, arb_srcptr h, slong hlen, slong n, slong prec)

void _arb_poly_tan_series_basecase(const arb_poly_t h, slong n, slong prec)
void _arb_poly_tan_series(const arb_poly_t h, slong n, slong prec)
void _arb_poly_tan_series(arb_ptr t, arb_ptr c, arb_srcptr h, slong hlen, slong n, slong prec)
void arb_poly_tan_series(arb_ptr t, arb_ptr c, arb_srcptr h, slong hlen, slong n, slong prec)
\end{verbatim}
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.

```c
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, 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, arb_poly_t h, slong n, slong prec)
void _arb_poly_sinh_cosh_series_basecase(arb_ptr c, arb_ptr s, arb_srcptr h, slong hlen, slong n, slong prec)
void arb_poly_sinh_cosh_series_basecase(arb_poly_t c, arb_poly_t s, const arb_poly_t h, slong n, slong prec)
void _arb_poly_sinh_cosh_series_exponential(arb_ptr c, arb_ptr s, arb_srcptr h, slong hlen, slong n, slong prec)
void arb_poly_sinh_cosh_series_exponential(arb_poly_t c, arb_poly_t s, const arb_poly_t h, slong n, slong prec)
void _arb_poly_sinh_series(arb_ptr c,arb_ptr s,arb_srcptr h,slong hlen, slong n, slong prec)
void arb_poly_sinh_series(arb_poly_t c, arb_poly_t s, const arb_poly_t h, slong n, slong prec)
void _arb_poly_cosh_series(arb_ptr c,arb_ptr s,arb_srcptr h,slong hlen, slong n, slong prec)
void arb_poly_cosh_series(arb_poly_t c, arb_poly_t s, const arb_poly_t h, slong n, slong prec)
void _arb_poly_sinc_series(arb_ptr c,arb_ptr s,arb_srcptr h,slong hlen, slong n, slong prec)
void arb_poly_sinc_series(arb_poly_t c, arb_poly_t s, const arb_poly_t h, slong n, slong prec)
void _arb_poly_rgamma_series(arb_ptr c, arb_ptr s, arb_srcptr h, slong hlen, slong n, slong prec)
void arb_poly_rgamma_series(arb_poly_t c, arb_poly_t s, const arb_poly_t h, slong n, slong prec)
void _arb_poly_lgamma_series(arb_ptr c, arb_ptr s, arb_srcptr h, slong hlen, slong n, slong prec)
void arb_poly_lgamma_series(arb_poly_t c, arb_poly_t s, const arb_poly_t h, slong n, slong prec)
```

### 5.1.18 Gamma function and factorials

```c
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)
```

5.1. **arb_poly.h** – polynomials over the real numbers
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 \( \Gamma(h(x)), 1/\Gamma(h(x)), \text{or} \log \Gamma(h(x)), \psi(h(x)) \), truncated to length \( n \).

These functions first generate the Taylor series at the constant term of \( h \), and then call _arb_poly_compose_series(). The Taylor coefficients are generated using the Riemann zeta function if the constant term of \( h \) is a small integer, and with Stirling's series otherwise.

The underscore methods support aliasing of the input and output arrays, and require that \( hlen \) and \( n \) are greater than zero.

void _arb_poly_rising_ui_series(arb_ptr res, arb_srcptr f, slong flen, ulong r, slong trunc, slong prec)
void arb_poly_rising_ui_series(arb_poly_t res, const arb_poly_t f, ulong r, slong trunc, slong prec)

Sets res to the rising factorial \((f)(f+1)(f+2)\cdots(f+r-1)\), truncated to length \( trunc \). The underscore method assumes that \( flen, r \) and \( trunc \) are at least 1, and does not support aliasing.

Uses binary splitting.

### 5.1.19 Zeta function

void arb_poly_zeta_series(arb_poly_t res, const arb_poly_t s, const arb_t a, int deflate, slong n, slong prec)

Sets res to the Hurwitz zeta function \( \zeta(s,a) \) where \( s \) a power series and \( a \) is a constant, truncated to length \( n \). To evaluate the usual Riemann zeta function, set \( a = 1 \).

If \( deflate \) is nonzero, evaluates \( \zeta(s,a)+1/(1-s) \), which is well-defined as a limit when the constant term of \( s \) is 1. In particular, expanding \( \zeta(s,a)+1/(1-s) \) with \( s = 1+x \) gives the Stieltjes constants

\[
\sum_{k=0}^{n-1} \frac{(-1)^k}{k!} \gamma_k(a)x^k.
\]

If \( a = 1 \), this implementation uses the reflection formula if the midpoint of the constant term of \( s \) is negative.

void _arb_poly_riemann_siegel_theta_series(arb_ptr res, arb_srcptr h, slong hlen, slong n, slong prec)
void arb_poly_riemann_siegel_theta_series(arb_poly_t res, const arb_poly_t h, slong n, slong prec)

Sets res to the series expansion of the Riemann-Siegel theta function

\[
\theta(h) = \arg \left( \Gamma \left( \frac{2ih + 1}{4} \right) \right) - \frac{\log \pi}{2} h
\]

where the argument of the gamma function is chosen continuously as the imaginary part of the log gamma function.

The underscore method does not support aliasing of the input and output arrays, and requires that the lengths are greater than zero.

void _arb_poly_riemann_siegel_z_series(arb_ptr res, arb_srcptr h, slong hlen, slong n, slong prec)
void arb_poly_riemann_siegel_z_series(arb_poly_t res, const arb_poly_t h, slong n, slong prec)

Sets res to the series expansion of the Riemann-Siegel Z-function

\[
Z(h) = e^{i\theta(h)}\zeta(1/2 + ih).
\]
The zeros of the Z-function on the real line precisely correspond to the imaginary parts of the zeros of the Riemann zeta function on the critical line.

The underscore method supports aliasing of the input and output arrays, and requires that the lengths are greater than zero.

### 5.1.20 Root-finding

```c
void _arb_poly_root_bound_fujiwara(mag_t bound, arb_srcptr poly, slong len)
```

Sets `bound` to an upper bound for the magnitude of all the complex roots of `poly`. Uses Fujiwara’s bound

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

where \(a_0, \ldots, a_n\) are the coefficients of `poly`.

```c
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 \(C = \sup_{t, u \in I} \frac{1}{2} \left| f'(t)/f'(u) \right|\), 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.

```c
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 \(C = \sup_{t, u \in I} \frac{1}{2} \left| f''(t)/f'(u) \right|\), 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.

```c
void _arb_poly_newton_refine_root(arb_t r, arb_srcptr poly, slong len, const arb_t start, const arb_t convergence_interval, const arf_t convergence_factor, slong eval_extra_prec, slong prec)
```

Refines a precise estimate of a polynomial root to high precision by performing several Newton steps, using nearly optimally chosen doubling precision steps.

The inputs are defined as for `_arb_poly_newton_step`, except for the precision parameters: `prec` is the target accuracy and `eval_extra_prec` is the estimated number of guard bits that need to be added to evaluate the polynomial accurately close to the root (typically, if the polynomial has large coefficients of alternating signs, this needs to be approximately the bit size of the coefficients).

### 5.1.21 Other special polynomials

```c
void _arb_poly_swinnerton_dyer_ui(arb_ptr poly, ulong n, slong trunc, 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.

5.1. `arb_poly.h` – polynomials over the real numbers 87
If \( \text{prec} \) is set to zero, a precision is chosen automatically such that \( \text{arb\_poly\_get\_unique\_fmpz\_poly}() \) should be successful. Otherwise a working precision of \( \text{prec} \) bits is used.

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

### 5.2 \texttt{acb\_poly.h} – polynomials over the complex numbers

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

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

#### 5.2.1 Types, macros and constants

\texttt{acb\_poly\_struct}

\texttt{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 \texttt{acb\_poly\_t} is defined as an array of length one of type \texttt{acb\_poly\_struct}, permitting an \texttt{acb\_poly\_t} to be passed by reference.

#### 5.2.2 Memory management

\texttt{void \_acb\_poly\_init}\( (\texttt{acb\_poly\_t \ poly}) \)

Initializes the polynomial for use, setting it to the zero polynomial.

\texttt{void \_acb\_poly\_clear}\( (\texttt{acb\_poly\_t \ poly}) \)

Clears the polynomial, deallocating all coefficients and the coefficient array.

\texttt{void \_acb\_poly\_fit\_length}\( (\texttt{acb\_poly\_t \ poly, slong \ len}) \)

Makes sure that the coefficient array of the polynomial contains at least \( \text{len} \) initialized coefficients.

\texttt{void \_acb\_poly\_set\_length}\( (\texttt{acb\_poly\_t \ poly, slong \ len}) \)

Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

\texttt{void \_acb\_poly\_normalise}\( (\texttt{acb\_poly\_t \ poly}) \)

Strips any trailing coefficients which are identical to zero.

\texttt{void \_acb\_poly\_swap}\( (\texttt{acb\_poly\_t \ poly1, acb\_poly\_t \ poly2}) \)

Swaps \texttt{poly1} and \texttt{poly2} efficiently.

\texttt{slong \_acb\_poly\_allocated\_bytes}\( (\texttt{const acb\_poly\_t \ x}) \)

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

#### 5.2.3 Basic properties and manipulation

\texttt{slong \_acb\_poly\_length}\( (\texttt{const acb\_poly\_t \ poly}) \)

Returns the length of \texttt{poly}, i.e. zero if \texttt{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)
    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)
    Sets the coefficient with index n in poly to the value c. We require that n is nonnegative.

void acb_poly_set_coeff_acb(acb_poly_t poly, slong n, const acb_t c)
    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.

5.2.4 Input and output

void acb_poly_printd(const acb_poly_t poly, slong digits)
    Prints the polynomial as an array of coefficients, printing each coefficient using acb_printd.

void acb_poly_fprintd(FILE * file, const acb_poly_t poly, slong digits)
    Prints the polynomial as an array of coefficients to the stream file, printing each coefficient using
acb_fprintd.
5.2.5 Random generation

```c
void acb_poly_randtest(acb_poly_t poly, flint_rand_t state, slong len, slong prec,
                     slong mag_bits)
```
Creates a random polynomial with length at most `len`.

5.2.6 Comparisons

```c
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.

```c
int acb_poly_contains(const acb_poly_t poly1, const acb_poly_t poly2)
```
Returns nonzero iff `poly2` is contained in `poly1`.

```c
int acb_poly_contains_fmpz_poly(const acb_poly_t poly1, const fmpz_poly_t poly2)
```
Returns nonzero iff `poly2` is contained in `poly1`.

```c
int acb_poly_contains_fmpq_poly(const acb_poly_t poly1, const fmpq_poly_t poly2)
```
Returns nonzero iff `poly2` is contained in `poly1`.

```c
int _acb_poly_overlaps(acb_srcptr poly1, slong len1, acb_srcptr poly2, slong len2)
```
Returns nonzero iff `poly1` overlaps with `poly2`. The underscore function requires that `len1` ist at least as large as `len2`.

```c
int acb_poly_is_real(const acb_poly_t poly)
```
Returns nonzero iff all coefficients in `poly` have zero imaginary part.

5.2.7 Conversions

```c
void acb_poly_set_fmpz_poly(acb_poly_t poly, const fmpz_poly_t re, slong prec)
```
Sets `poly` to the given real part `re` plus the imaginary part, both rounded to `prec` bits.

```c
void acb_poly_set2_fmpz_poly(acb_poly_t poly, const fmpz_poly_t re, const fmpz_poly_t im, slong prec)
```
Sets `poly` to the given real part `re` plus the imaginary part `im`, both rounded to `prec` bits.

```c
void acb_poly_set_arb_poly(acb_poly_t poly, const arb_poly_t re)
```
Sets `poly` to the given real part `re`, plus the imaginary part.

```c
void acb_poly_set2_arb_poly(acb_poly_t poly, const arb_poly_t re, const arb_poly_t im)
```
Sets `poly` to the given real part `re` and the imaginary part `im`, both rounded to `prec` bits.

```c
void acb_poly_set_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.

```c
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.

```c
void acb_poly_set_acb(acb_poly_t poly, slong src)
```
Sets `poly` to `src`.

```c
void acb_poly_set_si(acb_poly_t poly, slong src)
```
Sets `poly` to `src`.

5.2.8 Bounds

```c
void _acb_poly_majorant(arb_ptr res, acb_srcptr poly, slong len, 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.

```c
void acb_poly_majorant(arb_poly_t res, const acb_poly_t poly, slong prec)
```
Sets `res` to an exact real polynomial whose coefficients are upper bounds for the absolute values of the coefficients in `poly`, rounded to `prec` bits.
5.2.9 Arithmetic

```c
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 lenA ≥ lenB > 0, \(n > 0\),
    lenA + lenB − 1 ≥ \(n\).

    The \textit{classical} version uses a plain loop.

    The \textit{transpose} version evaluates the product using four real polynomial multiplications (via
    \texttt{arb_poly_mullow()}).

    The \textit{transpose\_gauss} version evaluates the product using three real polynomial multiplications.
    This is almost always faster than \textit{transpose}, but has worse numerical stability when the coefficients
    vary in magnitude.

    The default function \texttt{acb\_poly\_mullow()} automatically switches between \textit{classical} and \textit{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 \texttt{acb\_poly\_mullow\_classical}(acb\_poly\_t C, \texttt{const} acb\_poly\_t A, \texttt{const} acb\_poly\_t B, \\
slong n, \texttt{slong prec})

void \texttt{acb\_poly\_mullow\_transpose}(acb\_poly\_t C, \texttt{const} acb\_poly\_t A, \texttt{const} acb\_poly\_t B, \\
slong n, \texttt{slong prec})

void \texttt{acb\_poly\_mullow\_transpose\_gauss}(acb\_poly\_t C, \texttt{const} acb\_poly\_t A, \texttt{const} acb\_poly\_t B, \\
\texttt{slong n}, \texttt{slong prec})

void \texttt{acb\_poly\_mullow}(acb\_poly\_t C, \texttt{const} acb\_poly\_t A, \texttt{const} acb\_poly\_t B, \texttt{slong n}, \\
\texttt{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 \texttt{\_acb\_poly\_mul}(acb\_ptr C, acb\_srcptr A, \texttt{slong} lenA, acb\_srcptr B, \texttt{slong} lenB, \texttt{slong prec})

Sets \( \{C, \text{len}A + \text{len}B - 1\} \) to the product of \( \{A, \text{len}A\} \) and \( \{B, \text{len}B\} \). The output is not allowed to be aliased with either of the inputs. We require \( \text{len}A \geq \text{len}B > 0 \). This function is implemented as a simple wrapper for \texttt{\_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 \texttt{acb\_poly\_mul}(acb\_poly\_t C, \texttt{const} acb\_poly\_t A, \texttt{const} acb\_poly\_t B2, \texttt{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 \texttt{\_acb\_poly\_inv\_series}(acb\_ptr Qinv, acb\_srcptr Q, \texttt{slong} Qlen, \texttt{slong len}, \texttt{slong prec})

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

void \texttt{acb\_poly\_inv\_series}(acb\_poly\_t Qinv, \texttt{const} acb\_poly\_t Q, \texttt{slong n}, \texttt{slong prec})

Sets \( Qinv \) to the power series inverse of \( Q \).

void \texttt{\_acb\_poly\_div\_series}(acb\_ptr Q, acb\_srcptr A, \texttt{slong} Alen, acb\_srcptr B, \texttt{slong} Blen, \\
\texttt{slong n}, \texttt{slong prec})

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

void \texttt{acb\_poly\_div\_series}(acb\_poly\_t Q, \texttt{const} acb\_poly\_t A, \texttt{const} acb\_poly\_t B, \texttt{slong n}, \\
\texttt{slong prec})

Sets \( Q \) to the power series quotient \( A \) divided by \( B \), truncated to length \( n \).

void \texttt{\_acb\_poly\_div}(acb\_ptr Q, acb\_srcptr A, \texttt{slong} lenA, acb\_srcptr B, \texttt{slong} lenB, \texttt{slong prec})

void \texttt{\_acb\_poly\_rem}(acb\_ptr R, acb\_srcptr A, \texttt{slong} lenA, acb\_srcptr B, \texttt{slong} lenB, \texttt{slong prec})

void \texttt{\_acb\_poly\_divrem}(acb\_ptr Q, acb\_ptr R, acb\_srcptr A, \texttt{slong} lenA, acb\_srcptr B, \\
\texttt{slong lenB}, \texttt{slong prec})

int \texttt{acb\_poly\_divrem}(acb\_poly\_t Q, acb\_poly\_t R, \texttt{const} acb\_poly\_t A, \texttt{const} acb\_poly\_t B, \\
\texttt{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 \texttt{\_acb\_poly\_div\_root}(acb\_ptr Q, \texttt{acb\_t} R, acb\_srcptr A, \texttt{slong} len, \texttt{const} \texttt{acb\_t} c, \\
\texttt{slong prec})

Divides \( A \) by the polynomial \( x - c \), computing the quotient \( Q \) as well as the remainder \( R = f(c) \).

### 5.2.10 Composition

void \texttt{\_acb\_poly\_taylor\_shift\_horner}(acb\_ptr g, \texttt{const} \texttt{acb\_t} \texttt{c}, \texttt{slong n}, \texttt{slong prec})

void \texttt{\_acb\_poly\_taylor\_shift\_horner}(acb\_poly\_t g, \texttt{const} acb\_poly\_t f, \texttt{const} \texttt{acb\_t} \texttt{c}, \\
\texttt{slong prec})

void \texttt{\_acb\_poly\_taylor\_shift\_divconquer}(acb\_ptr g, \texttt{const} \texttt{acb\_t} \texttt{c}, \texttt{slong n}, \texttt{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_divconquer(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, 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_series_horner(acb_ptr res, acb_srcptr poly1, slong len1, acb_srcptr poly2, slong len2, slong prec)
void acb_poly_compose_series_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 \( \text{res} \) to the composition \( h(x) = f(g(x)) \) where \( f \) is given by \( \text{poly1} \) and \( g \) is given by \( \text{poly2} \), respectively using Horner’s rule, divide-and-conquer, and an automatic choice between the two algorithms.

The default algorithm also handles special-form input \( g = ax^n + c \) efficiently by performing a Taylor shift followed by a rescaling.

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

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

### 5.2.11 Evaluation

void _acb_poly_evaluate_horner(acb_t y, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate_horner(acb_t y, const acb_poly_t f, const acb_t x, slong prec)
void _acb_poly_evaluate_rectangular(acb_t y, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate_rectangular(acb_t y, const acb_poly_t f, const acb_t x, slong prec)
void _acb_poly_evaluate(acb_t y, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate(acb_t y, const acb_poly_t f, const acb_t x, slong prec)
void _acb_poly_evaluate(acb_t y, const acb_poly_t f, const acb_t x, slong prec)

Sets \( y = f(x) \), evaluated respectively using Horner’s rule, rectangular splitting, and an automatic algorithm choice.

void _acb_poly_evaluate2_horner(acb_t y, acb_t z, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate2_horner(acb_t y, acb_t z, const acb_poly_t f, const acb_t x, slong prec)
void _acb_poly_evaluate2_rectangular(acb_t y, acb_t z, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate2_rectangular(acb_t y, acb_t z, const acb_poly_t f, const acb_t x, slong prec)
void _acb_poly_evaluate2(acb_t y, acb_t z, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate2(acb_t y, acb_t z, const acb_poly_t f, const acb_t x, slong prec)

Sets \( y = f(x) \), \( z = f'(x) \), evaluated respectively using Horner’s rule, rectangular splitting, and an automatic algorithm choice.

When Horner’s rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.

### 5.2.12 Product trees

void _acb_poly_product_roots(acb_ptr poly, acb_srcptr xs, slong n, slong prec)
void acb_poly_product_roots(acb_poly_t poly, acb_srcptr xs, slong n, slong prec)

Generates the polynomial \((x - x_0)(x - x_1)\cdots(x - x_{n-1})\).
acb_ptr * _acb_poly_tree_alloc(slong len)
    Returns an initialized data structure capable of representing a remainder tree (product tree) of len roots.

void _acb_poly_tree_free(acb_ptr * tree, slong len)
    Deallocates a tree structure as allocated using _acb_poly_tree_alloc.

void _acb_poly_tree_build(acb_ptr * tree, acb_srcptr roots, slong len, slong prec)
    Constructs a product tree from a given array of len roots. The tree structure must be pre-allocated to the specified length using _acb_poly_tree_alloc().

5.2.13 Multipoint evaluation

void _acb_poly_evaluate_vec_iter(acb_ptr ys, acb_srcptr poly, slong plen, acb_srcptr xs, slong n, slong prec)
void acb_poly_evaluate_vec_iter(acb_ptr ys, const acb_poly_t poly, acb_srcptr xs, slong n, slong prec)
    Evaluates the polynomial simultaneously at n given points, calling _acb_poly_evaluate() repeatedly.

void _acb_poly_evaluate_vec_fast_precomp(acb_ptr vs, acb_srcptr poly, slong plen, acb_ptr * tree, slong len, slong prec)
void _acb_poly_evaluate_vec_fast(acb_ptr ys, acb_srcptr poly, slong plen, acb_ptr * tree, slong len, slong prec)
void acb_poly_evaluate_vec_fast(acb_ptr ys, const acb_poly_t poly, acb_srcptr xs, slong n, slong prec)
    Evaluates the polynomial simultaneously at n given points, using fast multipoint evaluation.

5.2.14 Interpolation

void _acb_poly_interpolate_newton(acb_ptr poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
void acb_poly_interpolate_newton(acb_poly_t poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
    Recovers the unique polynomial of length at most n that interpolates the given x and y values.
    This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

void _acb_poly_interpolate_barycentric(acb_ptr poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
void acb_poly_interpolate_barycentric(acb_poly_t poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
    Recovers the unique polynomial of length at most n that interpolates the given x and y values.
    This implementation uses the barycentric form of Lagrange interpolation.

void _acb_poly_interpolation_weights(acb_ptr w, acb_ptr * tree, slong len, slong prec)
void _acb_poly_interpolate_fast_precomp(acb_ptr poly, acb_srcptr ys, acb_ptr * tree, acb_srcptr weights, slong len, slong prec)
void _acb_poly_interpolate_fast(acb_ptr poly, acb_srcptr xs, acb_srcptr ys, slong len, slong prec)
void acb_poly_interpolate_fast(acb_poly_t poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
    Recovers the unique polynomial of length at most n that interpolates the given x and y values, using fast Lagrange interpolation. The precomp function takes a precomputed product tree over the x values and a vector of interpolation weights as additional inputs.
5.2.15 Differentiation

void _acb_poly_derivative(acb_ptr res, acb_srcptr poly, slong len, slong prec)
Sets \{res, len - 1\} to the derivative of \{poly, len\}. Allows aliasing of the input and output.

void acb_poly_derivative(acb_poly_t res, const acb_poly_t poly, slong prec)
Sets res to the derivative of poly.

void _acb_poly_integral(acb_ptr res, acb_srcptr poly, slong len, slong prec)
Sets \{res, len\} to the integral of \{poly, len - 1\}. Allows aliasing of the input and output.

void acb_poly_integral(acb_poly_t res, const acb_poly_t poly, slong prec)
Sets res to the integral of poly.

5.2.16 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 \[\sum_k \frac{a_k}{k!} x^k\]. 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 \texttt{arb\_poly\_binomial\_transform()} for details.

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

5.2.17 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 exponentiation.

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, slong 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_ptr res, acb_srcptr g, slong glen, slong prec)
    Sets \( \{h, \text{len}\} \) to the power series \( f(x)^{g(x)} = \exp(g(x) \log f(x)) \) truncated to length \( \text{len} \). This function detects special cases such as \( g \) being an exact small integer or \( \pm 1/2 \), and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that \( \text{flen} \) and \( \text{glen} \) do not exceed \( \text{len} \).

void acb_poly_pow_acb_series(acb_poly_t h, const acb_poly_t f, const acb_poly_t g, slong len, slong prec)
    Sets \( h \) to the power series \( f(x)^{g(x)} = \exp(g(x) \log f(x)) \) truncated to length \( \text{len} \). This function detects special cases such as \( g \) being an exact small integer or \( \pm 1/2 \), and computes such powers more efficiently.

void _acb_poly_pow_acb_series(acb_ptr h, acb_srcptr f, slong flen, const acb_t g, slong len, slong prec)
    Sets \( \{h, \text{len}\} \) to the power series \( f(x)^{g(x)} = \exp(g(x) \log f(x)) \) truncated to length \( \text{len} \). This function detects special cases such as \( g \) being an exact small integer or \( \pm 1/2 \), and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that \( \text{flen} \) does not exceed \( \text{len} \).

void acb_poly_pow_acb_series(acb_poly_t h, const acb_poly_t f, const acb_t g, slong len, slong prec)
    Sets \( h \) to the power series \( f(x)^{g(x)} = \exp(g(x) \log f(x)) \) truncated to length \( \text{len} \).

void _acb_poly_sqrt_series(acb_ptr h, acb_ptr f, slong flen, slong prec)
void acb_poly_sqrt_series(acb_poly_t h, const acb_poly_t f, slong n, slong prec)
    Sets \( g \) to the power series square root of \( h \), truncated to length \( n \). Uses division-free Newton iteration for the reciprocal square root, followed by a multiplication.
    The underscore method does not support aliasing of the input and output arrays. It requires that \( \text{hlen} \) and \( n \) are greater than zero.

void _acb_poly_rsqrt_series(acb_ptr h, acb_ptr f, slong flen, slong prec)
void acb_poly_rsqrt_series(acb_poly_t h, const acb_poly_t f, slong n, slong prec)
    Sets \( g \) to the reciprocal power series square root of \( h \), truncated to length \( n \). Uses division-free Newton iteration.
    The underscore method does not support aliasing of the input and output arrays. It requires that \( \text{hlen} \) and \( n \) are greater than zero.

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

void _acb_poly_atan_series(acb_ptr res, acb_srcptr f, slong flen, slong n, slong prec)
void acb_poly_atan_series(acb_poly_t res, const acb_poly_t f, slong n, slong prec)
    Sets res the power series inverse tangent of \( f \), truncated to length \( n \).
    Uses the formula
    \[
    \tan^{-1}(f(x)) = \int f'(x)/(1 + f(x)^2)dx,
    \]
    adding the function of the constant term in \( f \) as the constant of integration.
The underscore method supports aliasing of the input and output arrays. It requires that \( \text{flen} \) and \( n \) are greater than zero.

```c
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.

```c
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 \( \text{times}_\pi \) specifying that the input is to be multiplied by \( \pi \).

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

```c
void _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.

```c
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, acb_poly_t h, slong n, slong prec)

void acb_poly_cot_pi_series(acb_poly_t c, acb_srcptr h, slong hlen, slong n, slong prec)

void _acb_poly_cot_pi_series(acb_poly_t c, slong n, acb_srcptr h)

void acb_poly_cos_pi_series(acb_poly_t s, acb_poly_t h, slong n, slong prec)

void _acb_poly_sinh_cosh_series_basecase(acb_ptr s, acb_ptr c, 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, 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_basecase(acb_ptr s, acb_ptr c, 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)

Compute the respective trigonometric functions of the input multiplied by $\pi$.

void _acb_poly_sinh_cosh_series_basecase(acb_ptr s, acb_ptr c, acb_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_sinc_series(acb_ptr s, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_sinc_series(acb_poly_t s, acb_poly_t h, slong n, slong prec)

Sets $s$ to the sinc function 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_rgamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_rgamma_series(acb_ptr res, 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_ptr res, acb_poly_t h, slong n, slong prec)

5.2.18 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_ptr res, acb_poly_t h, slong n, slong prec)
void _acb_poly_lgamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)
void acb_poly_lgamma_series(acb_poly_t res, const acb_poly_t h, slong n, slong prec)
void _acb_poly_digamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)
void acb_poly_digamma_series(acb_poly_t res, const acb_poly_t h, slong n, slong prec)

Sets res to the series expansion of $\Gamma(h(x))$, $1/\Gamma(h(x))$, or $\log \Gamma(h(x))$, $\psi(h(x))$, truncated to length $n$.

These functions first generate the Taylor series at the constant term of $h$, and then call _acb_poly_compose_series(). The Taylor coefficients are generated using Stirling’s series.

The underscore methods support aliasing of the input and output arrays, and require that $hlen$ and $n$ are greater than zero.

void _acb_poly_rising_ui_series(acb_ptr res, acb_srcptr f, slong flen, ulong r, slong trunc, slong prec)
void acb_poly_rising_ui_series(acb_poly_t res, const acb_poly_t f, ulong r, slong trunc, slong prec)

Sets res to the rising factorial $(f)(f+1)(f+2)\cdots(f+r-1)$, truncated to length trunc. The underscore method assumes that $flen$, $r$ and $trunc$ are at least 1, and does not support aliasing. Uses binary splitting.

5.2.19 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

$$z = S(s, a, n) = \sum_{k=0}^{n-1} \frac{q^k}{(k+a)^{s+1}}$$

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

$$z = S(s, 1, n) \sum_{k=1}^{n} \frac{1}{k^{s+1}}$$

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 $2^{-(s+1)}$. This scheme requires about $n/\log n$ powers, $n/2$ multiplications, and temporary storage of $n/6$ power series. Due to the extra power series multiplications, it is only faster than the naive algorithm when len is small.

5.2.20 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.
5.2.21 Other special functions

void _acb_poly_zeta_em_bound1(acb_t bound, const acb_t s, const acb_t a, slong N, slong M, slong d, slong wp)

void _acb_poly_zeta_em_bound(acb_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, ulong 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, ulong 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 $\text{prec}$ bits. With $a=1$, this gives the usual Riemann zeta function.

If $\text{deflate}$ is nonzero, $\zeta(s,a) - 1/(s-1)$ is evaluated (which permits series expansion at $s=1$).

void _acb_poly_zeta_cpx_series(acb_ptr z, const acb_t s, const acb_t a, int deflate, slong d, slong prec)

Computes the series expansion of $\zeta(s+x,a)$ (or $\zeta(s+x,a) - 1/(s+x-1)$ if $\text{deflate}$ is nonzero) to order $d$.

This function wraps \_acb\_poly\_zeta\_em\_sum(), automatically choosing default values for $N,M$ using \_acb\_poly\_zeta\_em\_choose\_param() to target an absolute truncation error of $2^{-\text{prec}}$.

void _acb_poly_zeta_series(acb_ptr res, acb_srcptr h, slong hlen, const acb_t a, int deflate, slong len, slong prec)

void _acb_poly_zeta_series(acb_poly_t res, const acb_poly_t f, const acb_t a, int deflate, slong n, slong prec)

Sets $\text{res}$ to the Hurwitz zeta function $\zeta(s,a)$ where $s$ a power series and $a$ is a constant, truncated to length $n$. To evaluate the usual Riemann zeta function, set $a=1$.

If $\text{deflate}$ is nonzero, evaluates $\zeta(s,a) + 1/(1-s)$, which is well-defined as a limit when the constant term of $s$ is 1. In particular, expanding $\zeta(s,a) + 1/(1-s)$ with $s = 1+x$ gives the Stieltjes constants

$$\sum_{k=0}^{n-1} \frac{(-1)^k}{k!} \gamma_k(a)x^k.$$  

If $a=1$, this implementation uses the reflection formula if the midpoint of the constant term of $s$ is negative.

5.2.21 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 $\text{Li}_{s+z}(z)$, where $s$ and $z$ are given complex constants. The output is computed to length $\text{len}$ which must be positive. Aliasing between $w$ and $s$ or $z$ is not permitted.

The \texttt{small} version uses the standard power series expansion with respect to $z$, convergent when $|z|<1$. The \texttt{zeta} version evaluates the polylogarithm as a sum of two Hurwitz zeta functions. The default version automatically delegates to the \texttt{small} version when $z$ is close to zero, and the \texttt{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 \( \text{Li}_s(z) \) 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 \( \text{acb_hypgeom_erf_series}() \), \( \text{acb_hypgeom_erfc_series}() \), \( \text{acb_hypgeom_erfi_series}() \).

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

void acb_poly_agm1_series(acb_poly_t res, const acb_poly_t z, slong n, slong prec)

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

void _acb_poly_elliptic_k_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)

void acb_poly_elliptic_k_series(acb_poly_t res, const acb_poly_t z, slong n, slong prec)

Sets \( res \) to the complete elliptic integral of the first kind of the power series \( z \), truncated to length \( n \).

void _acb_poly_elliptic_p_series(acb_ptr res, acb_srcptr z, slong zlen, const acb_t tau, slong len, slong prec)

void acb_poly_elliptic_p_series(acb_poly_t res, const acb_poly_t z, const acb_t tau, slong n, slong prec)

Sets \( res \) to the Weierstrass elliptic function of the power series \( z \), with periods \( 1 \) and \( \tau \), truncated to length \( n \).

5.2.22 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

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

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

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

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

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.

```c
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).

```c
slong _acb_poly_find_roots(acb_ptr roots, acb_srcptr poly, acb_srcptr initial, slong len, 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.

```c
int _acb_poly_validate_real_roots(acb_srcptr roots, acb_srcptr poly, slong len, 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`. 

---

5.2. acb_poly.h – polynomials over the complex numbers
These modules implement dense matrices with real and complex coefficients. Rudimentary linear algebra is supported.

6.1 arb_mat.h – matrices over the real numbers

An \texttt{arb\_mat\_t} represents a dense matrix over the real numbers, implemented as an array of entries of type \texttt{arb\_struct}.

The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

6.1.1 Types, macros and constants

\texttt{arb\_mat\_struct}

\texttt{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 \texttt{arb\_mat\_t} is defined as an array of length one of type \texttt{arb\_mat\_struct}, permitting an \texttt{arb\_mat\_t} to be passed by reference.

\texttt{arb\_mat\_entry(mat, i, j)}

Macro giving a pointer to the entry at row \textit{i} and column \textit{j}.

\texttt{arb\_mat\_nrows(mat)}

Returns the number of rows of the matrix.

\texttt{arb\_mat\_ncols(mat)}

Returns the number of columns of the matrix.

6.1.2 Memory management

\texttt{void arb\_mat\_init(arb\_mat\_t mat, slong r, slong c)}

Initializes the matrix, setting it to the zero matrix with \textit{r} rows and \textit{c} columns.

\texttt{void arb\_mat\_clear(arb\_mat\_t mat)}

Clears the matrix, deallocating all entries.

\texttt{slong arb\_mat\_allocated\_bytes(const arb\_mat\_t z)}

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

void arb_mat_set(arb_mat_t dest, const arb_mat_t src)
void arb_mat_set_fmpz_mat(arb_mat_t dest, const fmpz_mat_t src)
void arb_mat_set_round_fmpz_mat(arb_mat_t dest, const fmpz_mat_t src, slong prec)
void arb_mat_set_fmpq_mat(arb_mat_t dest, const fmpq_mat_t src, slong prec)

Sets dest to src. The operands must have identical dimensions.

6.1.4 Random generation

void arb_mat_randtest(arb_mat_t mat, flint_rand_t state, slong prec, slong mag_bits)

Sets mat to a random matrix with up to prec bits of precision and with exponents of width up to mag_bits.

6.1.5 Input and output

void arb_mat_printd(const arb_mat_t mat, slong digits)
Prints each entry in the matrix with the specified number of decimal digits.

void arb_mat_fprintd(FILE * file, const arb_mat_t mat, slong digits)
Prints each entry in the matrix with the specified number of decimal digits to the stream file.

6.1.6 Comparisons

int arb_mat_equal(const arb_mat_t mat1, const arb_mat_t mat2)
Returns nonzero iff the matrices have the same dimensions and identical entries.

int arb_mat_overlaps(const arb_mat_t mat1, const arb_mat_t mat2)
Returns nonzero iff the matrices have the same dimensions and each entry in mat1 overlaps with the corresponding entry in mat2.

int arb_mat_contains(const arb_mat_t mat1, const arb_mat_t mat2)
int arb_mat_contains_fmpz_mat(const arb_mat_t mat1, const fmpz_mat_t mat2)
int arb_mat_contains_fmpq_mat(const arb_mat_t mat1, const fmpq_mat_t mat2)

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

int arb_mat_eq(const arb_mat_t mat1, const arb_mat_t mat2)
Returns nonzero iff mat1 and mat2 certainly represent the same matrix.

int arb_mat_ne(const arb_mat_t mat1, const arb_mat_t mat2)
Returns nonzero iff mat1 and mat2 certainly do not represent the same matrix.

int arb_mat_is_empty(const arb_mat_t mat)
Returns nonzero iff the number of rows or the number of columns in mat is zero.

int arb_mat_is_square(const arb_mat_t mat)
Returns nonzero iff the number of rows is equal to the number of columns in mat.

6.1.7 Special matrices

void arb_mat_zero(arb_mat_t mat)
Sets all entries in mat to zero.

void arb_mat_one(arb_mat_t mat)
Sets the entries on the main diagonal to ones, and all other entries to zero.
6.1.8 Transpose

void arb_mat_transpose(arb_mat_t dest, const arb_mat_t src)
    Sets dest to the exact transpose src. The operands must have compatible dimensions. Aliasing is allowed.

6.1.9 Norms

void arb_mat_bound_inf_norm(mag_t b, const arb_mat_t A)
    Sets b to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of A.

void arb_mat_frobenius_norm(arb_t res, const arb_mat_t A, slong prec)
    Sets res to the Frobenius norm (i.e. the square root of the sum of squares of entries) of A.

void arb_mat_bound_frobenius_norm(mag_t res, const arb_mat_t A)
    Sets res to an upper bound for the Frobenius norm of A.

6.1.10 Arithmetic

void arb_mat_neg(arb_mat_t dest, const arb_mat_t src)
    Sets dest to the exact negation of src. The operands must have the same dimensions.

void arb_mat_add(arb_mat_t res, const arb_mat_t mat1, const arb_mat_t mat2, slong prec)
    Sets res to the sum of mat1 and mat2. The operands must have the same dimensions.

void arb_mat_sub(arb_mat_t res, const arb_mat_t mat1, const arb_mat_t mat2, slong prec)
    Sets res to the difference of mat1 and mat2. The operands must have the same dimensions.

void arb_mat_mul_classical(arb_mat_t C, const arb_mat_t A, const arb_mat_t B, slong prec)
void arb_mat_mul_threaded(arb_mat_t C, const arb_mat_t A, const arb_mat_t B, slong prec)
void arb_mat_mul(arb_mat_t res, const arb_mat_t mat1, const arb_mat_t mat2, slong prec)
    Sets res to the matrix product of mat1 and mat2. The operands must have compatible dimensions for matrix multiplication.
    The threaded version splits the computation over the number of threads returned by flint_get_num_threads(). The default version automatically calls the threaded version if the matrices are sufficiently large and more than one thread can be used.

void arb_mat_mul_entrywise(arb_mat_t C, const arb_mat_t A, const arb_mat_t B, slong prec)
    Sets C to the entrywise product of A and B. The operands must have the same dimensions.

void arb_mat_sqr_classical(arb_mat_t B, const arb_mat_t A, slong prec)
void arb_mat_sqr(arb_mat_t res, const arb_mat_t mat, slong prec)
    Sets res to the matrix square of mat. The operands must both be square with the same dimensions.

void arb_mat_pow_ui(arb_mat_t res, const arb_mat_t mat, ulong exp, slong prec)
    Sets res to mat raised to the power exp. Requires that mat is a square matrix.

6.1.11 Scalar arithmetic

void arb_mat_scalar_mul_2exp_si(arb_mat_t B, const arb_mat_t A, slong c)
    Sets B to A multiplied by $2^c$.

void arb_mat_scalar_addmul_si(arb_mat_t B, const arb_mat_t A, slong c, slong prec)
void arb_mat_scalar_addmul_fmpz(arb_mat_t B, const arb_mat_t A, const fmpz_t c, slong prec)
void arb_mat_scalar_addmul_arb(arb_mat_t B, const arb_mat_t A, const arb_t c, slong prec)
    Sets $B$ to $B + A \times c$.

void arb_mat_scalar_mul_si(arb_mat_t B, const arb_mat_t A, slong c, slong prec)

void arb_mat_scalar_mul_fmpz(arb_mat_t B, const arb_mat_t A, const fmpz_t c, slong prec)
    Sets $B$ to $A \times c$.

void arb_mat_scalar_mul_arb(arb_mat_t B, const arb_mat_t A, const arb_t c, slong prec)

void arb_mat_scalar_div_si(arb_mat_t B, const arb_mat_t A, slong c, slong prec)

void arb_mat_scalar_div_fmpz(arb_mat_t B, const arb_mat_t A, const fmpz_t c, slong prec)

void arb_mat_scalar_div_arb(arb_mat_t B, const arb_mat_t A, const arb_t c, slong prec)
    Sets $B$ to $A/c$.

6.1.12 Gaussian elimination and solving

int arb_mat_lu(slong * perm, arb_mat_t LU, const arb_mat_t A, slong prec)
    Given an $n \times n$ matrix $A$, computes an LU decomposition $PLU = A$ using Gaussian elimination with partial pivoting. The input and output matrices can be the same, performing the decomposition in-place.

Entry $i$ in the permutation vector perm is set to the row index in the input matrix corresponding to row $i$ in the output matrix.

The algorithm succeeds and returns nonzero if it can find $n$ invertible (i.e. not containing zero) pivot entries. This guarantees that the matrix is invertible.

The algorithm fails and returns zero, leaving the entries in $P$ and $LU$ undefined, if it cannot find $n$ invertible pivot elements. In this case, either the matrix is singular, the input matrix was computed to insufficient precision, or the LU decomposition was attempted at insufficient precision.

void arb_mat_solve_lu_precomp(arb_mat_t X, const slong * perm, const arb_mat_t LU, const arb_mat_t B, slong prec)
    Solves $AX = B$ given the precomputed nonsingular LU decomposition $A = PLU$. The matrices $X$ and $B$ are allowed to be aliased with each other, but $X$ is not allowed to be aliased with $LU$.

int arb_mat_solve(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, slong prec)
    Solves $AX = B$ where $A$ is a nonsingular $n \times n$ matrix and $X$ and $B$ are $n \times m$ matrices, using LU decomposition.

If $m > 0$ and $A$ cannot be inverted numerically (indicating either that $A$ is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that $A$ is invertible and that the exact solution matrix is contained in the output.

int arb_mat_inv(arb_mat_t X, const arb_mat_t A, slong prec)
    Sets $X = A^{-1}$ where $A$ is a square matrix, computed by solving the system $AX = I$.

If $A$ cannot be inverted numerically (indicating either that $A$ is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the matrix is invertible and that the exact inverse is contained in the output.

void arb_mat_det(arb_t det, const arb_mat_t A, slong prec)
    Computes the determinant of the matrix, using Gaussian elimination with partial pivoting. If at some point an invertible pivot element cannot be found, the elimination is stopped and the magnitude of the determinant of the remaining submatrix is bounded using Hadamard’s inequality.

6.1.13 Cholesky decomposition and solving

int _arb_mat_cholesky_banachiewicz(arb_mat_t A, slong prec)
int arb_mat_cho(arb_mat_t L, const arb_mat_t A, slong prec)

Computes the Cholesky decomposition of $A$, returning nonzero iff the symmetric matrix defined by the lower triangular part of $A$ is certainly positive definite.

If a nonzero value is returned, then $L$ is set to the lower triangular matrix such that $A = L \times L^T$.

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 $L \times D \times L^T$ 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 $L$.  

6.1. _arb_mat.h – matrices over the real numbers
void \texttt{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 \texttt{arb_mat_ldl()}. The inverse is calculated using the method of \cite{Kri2013} which is more efficient than solving $AX = I$ with \texttt{arb_mat_solve_ldl_precomp()}.  

6.1.14 Characteristic polynomial

void _arb_mat_charpoly\,(arb\_ptr cp, const arb\_mat\_t mat, slong prec)
void arb_mat_charpoly\,(arb\_poly\_t cp, const arb\_mat\_t mat, slong prec)
Sets $cp$ to the characteristic polynomial of $mat$ which must be a square matrix. If the matrix has $n$ rows, the underscore method requires space for $n + 1$ output coefficients. Employs a division-free algorithm using $O(n^4)$ operations.

6.1.15 Special functions

void arb_mat_exp_taylor_sum\,(arb\_mat\_t S, const arb\_mat\_t A, slong N, slong prec)
Sets $S$ to the truncated exponential Taylor series $S = \sum_{k=0}^{N-1} \frac{A^k}{k!}$. Uses rectangular splitting to compute the sum using $O(\sqrt{N})$ matrix multiplications. The recurrence relation for factorials is used to get scalars that are small integers instead of full factorials. As in \cite{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 $\exp(A/2^r)^2$, where $r$ is chosen to give rapid convergence.

The elementwise error when truncating the Taylor series after $N$ terms is bounded by the error in the infinity norm, for which we have

$$\left\| \exp(2^{-r}A) - \sum_{k=0}^{N-1} \frac{(2^{-r}A)^k}{k!} \right\|_{\infty} \leq \sum_{k=N}^{\infty} \frac{(2^{-r}\|A\|_{\infty})^k}{k!}.$$  

We bound the sum on the right using \texttt{mag_exp_tail()}. Truncation error is not added to entries whose values are determined by the sparsity structure of $A$.

void arb_mat_trace\,(arb\_t trace, const arb\_mat\_t mat, slong prec)
Sets $trace$ to the trace of the matrix, i.e. the sum of entries on the main diagonal of $mat$. The matrix is required to be square.

6.1.16 Sparsity structure

void arb_mat_entrywise_is_zero\,(fmpz\_mat\_t dest, const arb\_mat\_t src)
Sets each entry of $dest$ to indicate whether the corresponding entry of $src$ is certainly zero. If the entry of $src$ at row $i$ and column $j$ is zero according to \texttt{arb_is_zero()} then the entry of $dest$ at that row and column is set to one, otherwise that entry of $dest$ is set to zero.

void 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 \texttt{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 \texttt{arb_is_zero()}.  

slong arb_mat_count_not_is_zero\,(const arb\_mat\_t mat)
Returns the number of entries of $mat$ that are not certainly zero.
6.2 acb_mat.h – matrices over the complex numbers

An `acb_mat_t` represents a dense matrix over the complex numbers, implemented as an array of entries of type `acb_struct`.

The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

6.2.1 Types, macros and constants

`acb_mat_struct`

`acb_mat_t`

Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An `acb_mat_t` is defined as an array of length one of type `acb_mat_struct`, permitting an `acb_mat_t` to be passed by reference.

`acb_mat_entry(mat, i, j)`

Macro giving a pointer to the entry at row `i` and column `j`.

`acb_mat_nrows(mat)`

Returns the number of rows of the matrix.

`acb_mat_ncols(mat)`

Returns the number of columns of the matrix.

6.2.2 Memory management

`void acb_mat_init(acb_mat_t mat, slong r, slong c)`

Initializes the matrix, setting it to the zero matrix with `r` rows and `c` columns.

`void acb_mat_clear(acb_mat_t mat)`

Clears the matrix, deallocating all entries.

`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.

6.2.3 Conversions

`void acb_mat_set(acb_mat_t dest, const acb_mat_t src)`

`void acb_mat_set_fmpz_mat(acb_mat_t dest, const fmpz_mat_t src)`

`void acb_mat_set_round_fmpz_mat(acb_mat_t dest, const fmpz_mat_t src, slong prec)`

`void acb_mat_set_fmpq_mat(acb_mat_t dest, const fmpq_mat_t src, slong prec)`

`void acb_mat_set_arb_mat(acb_mat_t dest, const arb_mat_t src)`

`void acb_mat_set_round_arb_mat(acb_mat_t dest, const arb_mat_t src, slong prec)`

Sets `dest` to `src`. The operands must have identical dimensions.

6.2.4 Random generation

`void acb_mat_randtest(acb_mat_t mat, flint_rand_t state, slong prec, slong mag_bits)`

Sets `mat` to a random matrix with up to `prec` bits of precision and with exponents of width up to `mag_bits`. 
6.2.5 Input and output

void \texttt{acb\_mat\_printd}(\texttt{const acb\_mat\_t mat, slong digits})

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

void \texttt{acb\_mat\_fprintd}(\texttt{FILE * file, const acb\_mat\_t mat, slong digits})

Prints each entry in the matrix with the specified number of decimal digits to the stream \texttt{file}.

6.2.6 Comparisons

\begin{itemize}
  \item \texttt{int acb\_mat\_equal}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})
    \begin{itemize}
      \item Returns nonzero iff the matrices have the same dimensions and identical entries.
    \end{itemize}
  \item \texttt{int acb\_mat\_overlaps}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})
    \begin{itemize}
      \item Returns nonzero iff the matrices have the same dimensions and each entry in \texttt{mat1} overlaps with the corresponding entry in \texttt{mat2}.
    \end{itemize}
  \item \texttt{int acb\_mat\_contains}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})
  \item \texttt{int acb\_mat\_contains\_fmpz\_mat}(\texttt{const acb\_mat\_t mat1, const fmpz\_mat\_t mat2})
  \item \texttt{int acb\_mat\_contains\_fmpq\_mat}(\texttt{const acb\_mat\_t mat1, const fmpq\_mat\_t mat2})
    \begin{itemize}
      \item Returns nonzero iff the matrices have the same dimensions and each entry in \texttt{mat2} is contained in the corresponding entry in \texttt{mat1}.
    \end{itemize}
  \item \texttt{int acb\_mat\_eq}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})
    \begin{itemize}
      \item Returns nonzero iff \texttt{mat1} and \texttt{mat2} certainly represent the same matrix.
    \end{itemize}
  \item \texttt{int acb\_mat\_ne}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})
    \begin{itemize}
      \item Returns nonzero iff \texttt{mat1} and \texttt{mat2} certainly do not represent the same matrix.
    \end{itemize}
  \item \texttt{int acb\_mat\_is\_real}(\texttt{const acb\_mat\_t mat})
    \begin{itemize}
      \item Returns nonzero iff all entries in \texttt{mat} have zero imaginary part.
    \end{itemize}
  \item \texttt{int acb\_mat\_is\_empty}(\texttt{const acb\_mat\_t mat})
    \begin{itemize}
      \item Returns nonzero iff the number of rows or the number of columns in \texttt{mat} is zero.
    \end{itemize}
  \item \texttt{int acb\_mat\_is\_square}(\texttt{const acb\_mat\_t mat})
    \begin{itemize}
      \item Returns nonzero iff the number of rows is equal to the number of columns in \texttt{mat}.
    \end{itemize}
\end{itemize}

6.2.7 Special matrices

\begin{itemize}
  \item \texttt{void acb\_mat\_zero}(\texttt{acb\_mat\_t mat})
    \begin{itemize}
      \item Sets all entries in \texttt{mat} to zero.
    \end{itemize}
  \item \texttt{void acb\_mat\_one}(\texttt{acb\_mat\_t mat})
    \begin{itemize}
      \item Sets the entries on the main diagonal to ones, and all other entries to zero.
    \end{itemize}
\end{itemize}

6.2.8 Transpose

\begin{itemize}
  \item \texttt{void acb\_mat\_transpose}(\texttt{acb\_mat\_t dest, const acb\_mat\_t src})
    \begin{itemize}
      \item Sets \texttt{dest} to the exact transpose \texttt{src}. The operands must have compatible dimensions. Aliasing is allowed.
    \end{itemize}
\end{itemize}

6.2.9 Norms

\begin{itemize}
  \item \texttt{void acb\_mat\_bound\_inf\_norm}(\texttt{mag\_t b, const acb\_mat\_t A})
    \begin{itemize}
      \item Sets \texttt{b} to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of \texttt{A}.
    \end{itemize}
  \item \texttt{void acb\_mat\_frobenius\_norm}(\texttt{acb\_t res, const acb\_mat\_t A, slong prec})
    \begin{itemize}
      \item Sets \texttt{res} to the Frobenius norm (i.e. the square root of the sum of squares of entries) of \texttt{A}.
    \end{itemize}
\end{itemize}
void acb_mat_bound_frobenius_norm(mag_t res, const acb_mat_t A)
    Sets res to an upper bound for the Frobenius norm of A.

6.2.10 Arithmetic

void acb_mat_neg(acb_mat_t dest, const acb_mat_t src)
    Sets dest to the exact negation of src. The operands must have the same dimensions.

void acb_mat_add(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)
    Sets res to the sum of mat1 and mat2. The operands must have the same dimensions.

void acb_mat_sub(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)
    Sets res to the difference of mat1 and mat2. The operands must have the same dimensions.

void acb_mat_mul(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)
    Sets res to the matrix product of mat1 and mat2. The operands must have compatible dimensions
    for matrix multiplication.

void acb_mat_mul_entrywise(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)
    Sets res to the entrywise product of mat1 and mat2. The operands must have the same dimensions.

void acb_mat_sqr(acb_mat_t res, const acb_mat_t mat, slong prec)
    Sets res to the matrix square of mat. The operands must both be square with the same dimensions.

void acb_mat_pow_ui(acb_mat_t res, const acb_mat_t mat, ulong exp, slong prec)
    Sets res to mat raised to the power exp. Requires that mat is a square matrix.

6.2.11 Scalar arithmetic

void acb_mat_scalar_mul_2exp_si(acb_mat_t B, const acb_mat_t A, slong c)
    Sets B to A multiplied by $2^c$.

void acb_mat_scalar_addmul_si(acb_mat_t B, const acb_mat_t A, slong c, slong prec)
void acb_mat_scalar_addmul_fmpz(acb_mat_t B, const acb_mat_t A, const fmpz_t c, slong prec)
void acb_mat_scalar_addmul_arb(acb_mat_t B, const acb_mat_t A, const arb_t c, slong prec)
void acb_mat_scalar_addmul_acb(acb_mat_t B, const acb_mat_t A, const acb_t c, slong prec)
    Sets B to $B + A \times c$.

void acb_mat_scalar_mul_si(acb_mat_t B, const acb_mat_t A, slong c, slong prec)
void acb_mat_scalar_mul_fmpz(acb_mat_t B, const acb_mat_t A, const fmpz_t c, slong prec)
void acb_mat_scalar_mul_arb(acb_mat_t B, const acb_mat_t A, const arb_t c, slong prec)
void acb_mat_scalar_mul_acb(acb_mat_t B, const acb_mat_t A, const acb_t c, slong prec)
    Sets B to $B \times c$.

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 $B/A/c$. 

6.2. acb_mat.h – matrices over the complex numbers
6.2.12 Gaussian elimination and solving

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

```c
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 \( LU \).

```c
int acb_mat_solve(acb_mat_t X, const acb_mat_t A, const acb_mat_t B, slong prec)
```

Solves \( AX = B \) where \( A \) is a nonsingular \( n \times n \) matrix and \( X \) and \( B \) are \( n \times m \) matrices, using LU decomposition.

If \( m > 0 \) and \( A \) cannot be inverted numerically (indicating either that \( A \) is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that \( A \) is invertible and that the exact solution matrix is contained in the output.

```c
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.

```c
void acb_mat_det(acb_t det, const acb_mat_t A, slong prec)
```

Computes the determinant of the matrix, using Gaussian elimination with partial pivoting. If at some point an invertible pivot element cannot be found, the elimination is stopped and the magnitude of the determinant of the remaining submatrix is bounded using Hadamard’s inequality.

6.2.13 Characteristic polynomial

```c
void _acb_mat_charpoly(acb_ptr cp, const acb_mat_t mat, slong prec)
```

```c
void acb_mat_charpoly(acb_poly_t cp, const acb_mat_t mat, slong prec)
```

Sets \( cp \) to the characteristic polynomial of \( mat \) which must be a square matrix. If the matrix has \( n \) rows, the underscore method requires space for \( n + 1 \) output coefficients. Employs a division-free algorithm using \( O(n^4) \) operations.

6.2.14 Special functions

```c
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 \(\exp(A/2^r)^2\), where \(r\) is chosen to give rapid convergence of the Taylor series. Error bounds are computed as for \texttt{arb_mat_exp}.

void *acb_mat_trace(*acb_t trace, const *acb_mat_t mat, slong prec)

Sets trace to the trace of the matrix, i.e. the sum of entries on the main diagonal of mat. The matrix is required to be square.
These modules implement mathematical functions with complexity that goes beyond the basics covered directly in the \texttt{arb} and \texttt{acb} modules.

### 7.1 \texttt{acb_hypgeom.h} – hypergeometric functions of complex variables

The generalized hypergeometric function is formally defined by

\[
pFq(a_1, \ldots, a_p; b_1, \ldots, b_q; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \cdots (a_p)_k}{(b_1)_k \cdots (b_q)_k} \frac{z^k}{k!}.
\]

It can be interpreted using analytic continuation or regularization when the sum does not converge. In a looser sense, we understand “hypergeometric functions” to be linear combinations of generalized hypergeometric functions with prefactors that are products of exponentials, powers, and gamma functions.

#### 7.1.1 Convergent series

In this section, we define $T(k) = \prod_{i=0}^{p-1}(a_i)_k z^k / \prod_{i=0}^{q-1}(b_i)_k$ and

\[
pFq(a_0, \ldots, a_{p-1}; b_0, \ldots, b_{q-1}; z) = pFq(a_0, \ldots, a_{p-1}, 1; b_0 \ldots b_{q-1}; z) = \sum_{k=0}^{\infty} T(k)
\]

For the conventional generalized hypergeometric function $pFq$, compute $pFq+1$ with the explicit parameter $b_q = 1$, or remove a 1 from the $a_i$ parameters if there is one.

```c
void acb_hypgeom_pfq_bound_factor(mag_t C, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, ulong n)
```

Computes a factor $C$ such that $|\sum_{k=n}^{\infty} T(k)| \leq C|T(n)|$. See \textit{Convergent series}. As currently implemented, the bound becomes infinite when $n$ is too small, even if the series converges.

```c
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 $\texttt{prec}$ 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 $\texttt{prec}$ 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_inz(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_inz(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 \( w = \frac{1}{z} \) 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

\[
p_f(z) = \sum_{k=0}^{\infty} T(k) = \sum_{k=0}^{n-1} T(k) + \varepsilon
\]

directly from the defining series, including a rigorous bound for the truncation error \( \varepsilon \) 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 \( \text{len} \) is positive.
If \texttt{regularized} is set, the regularized sum is computed, avoiding division by zero at the poles of the gamma function.

The \texttt{forward}, \texttt{bs}, \texttt{rs} and default versions use forward recurrence, binary splitting, rectangular splitting, and an automatic algorithm choice.

```c
void acb_hypgeom_pfq_series_direct(acb_poly_t res, const acb_poly_struct * a, slong p, const acb_poly_struct * b, slong q, const acb_poly_t z, int regularized, slong n, slong len, slong prec)
```

Computes $p \, _q F_q(z)$ directly using the defining series, given parameters and argument that are power series. The result is a power series of length \texttt{len}. We require that \texttt{len} is positive.

An error bound is computed automatically as a function of the number of terms \texttt{n}. If \texttt{n} < 0, the number of terms is chosen automatically.

If \texttt{regularized} is set, the regularized hypergeometric function is computed instead.

### 7.1.2 Asymptotic series

$U(a, b, z)$ is the confluent hypergeometric function of the second kind with the principal branch cut, and $U^* = z^a U(a, b, z)$. For details about how error bounds are computed, see \textit{Asymptotic series for the confluent hypergeometric function}.

```c
void acb_hypgeom_u_asympt(acb_t res, const acb_t a, const acb_t b, const acb_t z, slong n, slong prec)
```

Sets \texttt{res} to $U^*(a, b, z)$ computed using \texttt{n} terms of the asymptotic series, with a rigorous bound for the error included in the output. We require \texttt{n} $\geq$ 0.

```c
int acb_hypgeom_u_usage_asympt(const acb_t z, slong prec)
```

Heuristically determines whether the asymptotic series can be used to evaluate $U(a, b, z)$ to \texttt{prec} accurate bits (assuming that \texttt{a} and \texttt{b} are small).

### 7.1.3 Generalized hypergeometric function

```c
void acb_hypgeom_pfq(acb_poly_t res, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, int regularized, slong len, slong prec)
```

Computes the generalized hypergeometric function $p \, _q F_q(z)$, or the regularized version if \texttt{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 \texttt{a} and \texttt{b}. This can be done ahead of time by the user in applications where duplicate parameters are likely to occur.

### 7.1.4 Confluent hypergeometric functions

```c
void acb_hypgeom_u_if1_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 \texttt{len}, given $a, b, z \in \mathbb{C}[[x]]$. If $b[0] \in \mathbb{Z}$, it computes one extra derivative and removes the singularity (it is then assumed that $b[1] \neq 0$). As currently implemented, the output is indeterminate if $b$ is nonexact and contains an integer.

```c
void acb_hypgeom_u_if1(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 \texttt{acb_hypgeom_u_if1_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(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 \( M(a, b, z) = \frac{1}{\Gamma(b)} {}_1F_1(a, b, z) \) 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_asym(acb_t res, const acb_t a, const acb_t t, int regularized, slong prec)
void acb_hypgeom_0f1_direct(acb_t res, const acb_t a, const acb_t t, int regularized, slong prec)
void acb_hypgeom_0f1(acb_t res, const acb_t a, const acb_t t, 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

\[
\frac{a F_1(a, z)}{\Gamma(a)} = (-z)^{(1-a)/2} I_{a-1}(2\sqrt{-z}) = z^{(1-a)/2} I_{a-1}(2\sqrt{z}).
\]

The Bessel-\( J \) function is used in the left half-plane and the Bessel-\( I \) function is used in the right half-plane, to avoid loss of accuracy due to evaluating the square root on the branch cut.

7.1.5 Error functions and Fresnel integrals

void acb_hypgeom_erf_propagated_error(mag_t re, mag_t im, const acb_t z)
Sets \( \text{re and im} \) 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_asym(acb_t res, const acb_t z, int complementary, slong prec, slong prec2)
Computes the error function respectively using

\[
\text{erf}(z) = \frac{2z}{\sqrt{\pi}} {}_1F_1\left(\frac{1}{2}, \frac{3}{2}, -z^2\right)
\]

\[
\text{erf}(z) = \frac{2ze^{-z^2}}{\sqrt{\pi}} {}_1F_1(1, \frac{3}{2}, z^2)
\]

\[
\text{erf}(z) = \frac{z}{\sqrt{2\pi}} \left( 1 - e^{-z^2} U\left(\frac{1}{2}, \frac{1}{2}, z^2\right) \right) = \frac{z}{\sqrt{2\pi}} e^{-z^2} U^\ast\left(\frac{1}{2}, \frac{1}{2}, z^2\right).
\]

The \( asymp \) version takes a second precision to use for the \( U \) term. It also takes an extra flag \( \text{complementary} \), 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

\texttt{acb_hypgeom_erf_propagated_error()}.

\texttt{void _acb_hypgeom_erf_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)}

Computes the error function of the power series \( z \), truncated to length \( \text{len} \).

\texttt{void acb_hypgeom_erf(acb_t res, const acb_t z, slong len, slong prec)}

Computes the complementary error function \( \text{erfc}(z) = 1 - \text{erf}(z) \). This function avoids catastrophic cancellation for large positive \( z \).

\texttt{void _acb_hypgeom_erfc_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)}

Computes the complementary error function of the power series \( z \), truncated to length \( \text{len} \).

\texttt{void acb_hypgeom_erfi(acb_t res, const acb_t z, slong len, slong prec)}

Computes the imaginary error function \( \text{erfi}(z) = -i\text{erf}(iz) \). This is a trivial wrapper of \texttt{acb_hypgeom_erf()}. 

\texttt{void _acb_hypgeom_erfi_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)}

Computes the imaginary error function of the power series \( z \), truncated to length \( \text{len} \).

\texttt{void acb_hypgeom_fresnel(acb_t res1, acb_t res2, const acb_t z, int normalized, slong prec)}

Sets \( \text{res1} \) to the Fresnel sine integral \( S(z) \) and \( \text{res2} \) to the Fresnel cosine integral \( C(z) \). Optionally, just a single function can be computed by passing \texttt{NULL} as the other output variable. The definition \( S(z) = \frac{1}{\sqrt{2\pi}} \int_0^z \sin(t^2)dt \) is used if \texttt{normalized} is 0, and \( S(z) = \frac{1}{\sqrt{2\pi}} \int_0^z \sin(\frac{1}{2}\pi t^2)dt \) is used if \texttt{normalized} is 1 (the latter is the Abramowitz & Stegun convention). \( C(z) \) is defined analogously.

\texttt{void _acb_hypgeom_fresnel_series(acb_ptr res1, acb_ptr res2, acb_srcptr z, slong zlen, int normalized, slong len, slong prec)}

Sets \( \text{res1} \) to the Fresnel sine integral and \( \text{res2} \) to the Fresnel cosine integral of the power series \( z \), truncated to length \( \text{len} \). Optionally, just a single function can be computed by passing \texttt{NULL} as the other output variable.

### 7.1.6 Bessel functions

\texttt{void acb_hypgeom_bessel_j_asym(acb_t res, const acb_t nu, const acb_t z, slong prec)}

Computes the Bessel function of the first kind via \texttt{acb_hypgeom_u_asym()}. For all complex \( \nu, z \), we have

\[
J_{\nu}(z) = \frac{z^{\nu}}{2^\nu e^{i\pi \nu/2} \Gamma(\nu + 1)} F_1(\nu + \frac{1}{2}, 2\nu + 1, 2iz) = A_+B_+ + A_-B_- 
\]

where

\[
A_{\pm} = z^\nu (\pm i z)^{-1/2 -\nu} U(\nu + 1/2, 2\nu + 1, \mp i z)^{-1/2} = (\pm iz)^{-1/2 -\nu} z^\nu (2\pi)^{-1/2} 
\]

\[
B_{\pm} = e^{\pm iz} U(\nu + 1/2, 2\nu + 1, \mp i z). 
\]

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 \( \text{Re}(z) > 0 \), we have \( A_{\pm} = \exp(\mp i(2\nu + 1)/4\pi)(2\pi z)^{-1/2} \).

\texttt{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 \texttt{acb\_hypgeom\_bessel\_j}(\texttt{acb\_t res}, \texttt{acb\_t nu}, \texttt{acb\_t z}, \texttt{slong prec})
Computes the Bessel function of the first kind $J_{\nu}(z)$ using an automatic algorithm choice.

void \texttt{acb\_hypgeom\_bessel\_y}(\texttt{acb\_t res}, \texttt{acb\_t nu}, \texttt{acb\_t z}, \texttt{slong prec})
Computes the Bessel function of the second kind $Y_{\nu}(z)$ from the formula
\[ Y_{\nu}(z) = \frac{\cos(\nu \pi)J_{\nu}(z) - J_{-\nu}(z)}{\sin(\nu \pi)} \]
unless $\nu = n$ is an integer in which case the limit value
\[ Y_n(z) = -\frac{2}{\pi} \left( i^n K_n(iz) + [\log(iz) - \log(z)] J_n(z) \right) \]
is computed. As currently implemented, the output is indeterminate if $\nu$ is nonexact and contains an integer.

void \texttt{acb\_hypgeom\_bessel\_jy}(\texttt{acb\_t res1}, \texttt{acb\_t res2}, \texttt{acb\_t nu}, \texttt{acb\_t z}, \texttt{slong prec})
Sets \texttt{res1} to $J_{\nu}(z)$ and \texttt{res2} to $Y_{\nu}(z)$, computed simultaneously. From these values, the user can easily construct the Bessel functions of the third kind (Hankel functions) $H^{(1)}_{\nu}(z), H^{(2)}_{\nu}(z) = J_{\nu}(z) \pm iY_{\nu}(z)$.

void \texttt{acb\_hypgeom\_bessel\_i\_asymp}(\texttt{acb\_t res}, \texttt{acb\_t nu}, \texttt{acb\_t z}, \texttt{slong prec})
void \texttt{acb\_hypgeom\_bessel\_i\_0f1}(\texttt{acb\_t res}, \texttt{acb\_t nu}, \texttt{acb\_t z}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_bessel\_i}(\texttt{acb\_t res}, \texttt{acb\_t nu}, \texttt{acb\_t z}, \texttt{slong prec})
Computes the modified Bessel function of the first kind $I_{\nu}(z)$ as a power series truncated to length \texttt{len}, given $\nu, z \in \mathbb{C}[\llbracket x \rrbracket]$. Uses the formula
\[ K_{\nu}(z) = \frac{1}{\Gamma(\nu + 1)} \left( \frac{z}{2} \right)^{\nu} \frac{\pi}{\sin(\pi \nu)} \left( e^{-\frac{i\pi}{2}} U^{\nu}(\nu + \frac{1}{2}, 2\nu + 1, 2z) \right) \]

If $\nu[0] \in \mathbb{Z}$, it computes one extra derivative and removes the singularity (it is then assumed that $\nu[1] \neq 0$). As currently implemented, the output is indeterminate if $\nu[0]$ is nonexact and contains an integer.

void \texttt{acb\_hypgeom\_bessel\_k\_0f1\_series}(\texttt{acb\_poly\_t res}, \texttt{acb\_poly\_t nu}, \texttt{acb\_poly\_t z}, \texttt{slong len}, \texttt{slong prec})
Computes the modified Bessel function of the second kind $K_{\nu}(z)$ as a power series truncated to length \texttt{len}, given $\nu, z \in \mathbb{C}[\llbracket x \rrbracket]$. Uses the formula
\[ K_{\nu}(z) = \frac{1}{2} \frac{\pi}{\sin(\pi \nu)} \left[ \left( \frac{z}{2} \right)^{\nu} \Phi_1 \left( 1 - \nu, \frac{z^2}{4} \right) - \left( \frac{z}{2} \right)^{\nu} \Phi_1 \left( 1 + \nu, \frac{z^2}{4} \right) \right] \]

If $\nu[0] \in \mathbb{Z}$, it computes one extra derivative and removes the singularity (it is then assumed that $\nu[1] \neq 0$). As currently implemented, the output is indeterminate if $\nu[0]$ is nonexact and contains an integer.

void \texttt{acb\_hypgeom\_bessel\_k\_0f1}(\texttt{acb\_t res}, \texttt{acb\_t nu}, \texttt{acb\_t z}, \texttt{slong prec})
Computes the modified Bessel function of the second kind $K_{\nu}(z)$ using an automatic algorithm choice.
7.1.7 Airy functions

The Airy functions are linearly independent solutions of the differential equation $y'' - zy = 0$. All solutions are entire functions. The standard solutions are denoted $Ai(z), Bi(z)$. For negative $z$, both functions are oscillatory. For positive $z$, the first function decreases exponentially while the second increases exponentially.

The Airy functions can be expressed in terms of Bessel functions of fractional order, but this is inconvenient since such formulas only hold piecewise (due to the Stokes phenomenon). Computation of the Airy functions can also be optimized more than Bessel functions in general. We therefore provide a dedicated interface for evaluating Airy functions.

The following methods optionally compute $(Ai(z), Ai'(z), Bi(z), Bi'(z))$ simultaneously. Any of the four function values can be omitted by passing NULL for the unwanted output variables, speeding up the evaluation.

```c
void acb_hypgeom_airy_direct(acb_t ai, acb_t ai_prime, acb_t bi, acb_t bi_prime, const acb_t z, slong n, slong prec)
```
Computes the Airy functions using direct series expansions truncated at $n$ terms. Error bounds are included in the output.

```c
void acb_hypgeom_airy_asympt(acb_t ai, acb_t ai_prime, acb_t bi, acb_t bi_prime, const acb_t z, slong n, slong prec)
```
Computes the Airy functions using asymptotic expansions truncated at $n$ terms. Error bounds are included in the output. For details about how the error bounds are computed, see Asymptotic series for Airy functions.

```c
void acb_hypgeom_airy_bound(mag_t ai, mag_t ai_prime, mag_t bi, mag_t bi_prime, const acb_t z)
```
Computes bounds for the Airy functions using first-order asymptotic expansions together with error bounds. This function uses some shortcuts to make it slightly faster than calling acb_hypgeom_airy_asympt() with $n = 1$.

```c
void acb_hypgeom_airy(acb_t ai, acb_t ai_prime, acb_t bi, acb_t bi_prime, const acb_t z, slong prec)
```
Computes Airy functions using an automatic algorithm choice.

We use acb_hypgeom_airy_asympt() whenever this gives full accuracy and acb_hypgeom_airy_direct() otherwise. In the latter case, we first use hardware double precision arithmetic to determine an accurate estimate of the working precision needed to compute the Airy functions accurately for given $z$. This estimate is obtained by comparing the leading-order asymptotic estimate of the Airy functions with the magnitude of the largest term in the power series. The estimate is generic in the sense that it does not take into account vanishing near the roots of the functions. We subsequently evaluate the power series at the midpoint of $z$ and bound the propagated error using derivatives. Derivatives are bounded using acb_hypgeom_airy_bound().

```c
void acb_hypgeom_airy_jet(acb_ptr ai, acb_ptr bi, const acb_t z, slong len, slong prec)
```
Writes to $ai$ and $bi$ the respective Taylor expansions of the Airy functions at the point $z$, truncated to length $len$. Either of the outputs can be NULL to avoid computing that function. The variable $z$ is not allowed to be aliased with the outputs. To simplify the implementation, this method does not compute the series expansions of the primed versions directly; these are easily obtained by computing one extra coefficient and differentiating the output with acb_poly_derivative().

```c
void _acb_hypgeom_airy_series(acb_ptr ai, acb_ptr ai_prime, acb_ptr bi, acb_ptr bi_prime, acb_srcptr z, slong zlen, slong len, slong prec)
```

```c
void acb_hypgeom_airy_series(acb_poly_t ai, acb_poly_t ai_prime, acb_poly_t bi, acb_poly_t bi_prime, const acb_poly_t z, slong len, slong prec)
```
Computes the Airy functions evaluated at the power series $z$, truncated to length $len$. As with the other Airy methods, any of the outputs can be NULL.
7.1.8 Incomplete gamma and beta functions

```c
void acb_hypgeom_gamma_upper_asympt(acb_t res, const acb_t s, const acb_t z, int regularized,
                                     slong prec)
void acb_hypgeom_gamma_upper_ifia(acb_t res, const acb_t s, const acb_t z, int regularized,
                                     slong prec)
void acb_hypgeom_gamma_upper_ifib(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

\[
\Gamma(s, z) = e^{-z}U(1-s, 1-s, z)
\]

\[
\Gamma(s, z) = \Gamma(s) - \frac{z^s}{s} F_1(s, s+1, -z)
\]

\[
\Gamma(s, z) = \Gamma(s) - \frac{z^s e^{-z}}{s} F_1(1, s+1, z)
\]

\[
\Gamma(s, z) = \frac{(-1)^n}{n!} (\psi(n+1) - \log(z)) + \frac{(-1)^n}{(n+1)!} z^2 F_2(1, 1, 2 + n, -z) - z^{-n} \sum_{k=0}^{n-1} \frac{(-z)^k}{(k-n)!}, \quad n = -s \in \mathbb{Z}_{\geq 0}
\]

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.

```c
void _acb_hypgeom_gamma_upper_series(acb_ptr res, const acb_t s, acb_srcptr z, slong zlen,
                                      int regularized, slong n, slong prec)
void acb_hypgeom_gamma_upper_series(acb_poly_t res, const acb_t s, const acb_poly_t z,
                                    int regularized, slong n, slong prec)
```

Sets `res` to an upper incomplete gamma function where $s$ is a constant and $z$ is a power series, truncated to length $n$. The `regularized` argument has the same interpretation as in `acb_hypgeom_gamma_upper()`.

```c
void acb_hypgeom_gamma_lower(acb_t res, const acb_t s, const acb_t z, int regularized,
                           slong prec)
void _acb_hypgeom_gamma_lower_series(acb_ptr res, const acb_t s, acb_srcptr z, slong zlen,
                                        int regularized, slong n, slong prec)
```

If `regularized` is 0, computes the lower incomplete gamma function $\gamma(s, z) = \frac{z^s}{s} F_1(s, s+1, -z)$.

If `regularized` is 1, computes the regularized lower incomplete gamma function $P(s, z) = \gamma(s, z) / \Gamma(s)$.

If `regularized` is 2, computes a further regularized lower incomplete gamma function $\gamma^*(s, z) = z^{-s} P(s, z)$.
void \texttt{acb\_hypgeom\_gamma\_lower\_series}(\texttt{acb\_poly\_t res}, \texttt{acb\_t s}, \texttt{acb\_poly\_t z},
\texttt{int regularized}, \texttt{slong n}, \texttt{slong prec})

Sets \texttt{res} to an lower incomplete gamma function where \texttt{s} is a constant and \texttt{z} is a power series, truncated to length \texttt{n}. The \texttt{regularized} argument has the same interpretation as in \texttt{acb\_hypgeom\_gamma\_lower()}.

void \texttt{acb\_hypgeom\_beta\_lower}(\texttt{acb\_t res}, \texttt{acb\_t a}, \texttt{acb\_t b}, \texttt{acb\_t z}, \texttt{int regularized}, \texttt{slong prec})

Computes the (lower) incomplete beta function, defined by \( I(a, b; z) = \frac{\Gamma(a + b)}{\Gamma(b)} \int_0^z t^{a-1}(1-t)^{b-1} dt \), optionally the regularized incomplete beta function \( I(a, b; z) = B(a, b; z)/B(a, b; 1) \).

In general, the integral must be interpreted using analytic continuation. The precise definitions for all parameter values are

\[
B(a, b; z) = \frac{z^a}{a} {}_2F_1(a, 1 - b, a + 1, z)
\]

\[
I(a, b; z) = \frac{\Gamma(a + b)}{\Gamma(b)} z^a {}_2\tilde{F}_1(a, 1 - b, a + 1, z).
\]

Note that both functions with this definition are undefined for nonpositive integer \( a \), and \( I \) is undefined for nonpositive integer \( a + b \).

void \texttt{acb\_hypgeom\_beta\_lower\_series}(\texttt{acb\_ptr res}, \texttt{acb\_t a}, \texttt{acb\_t b}, \texttt{acb\_srcptr z},
\texttt{slong zlen}, \texttt{int regularized}, \texttt{slong n}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_beta\_lower\_series}(\texttt{acb\_poly\_t res}, \texttt{acb\_t a}, \texttt{acb\_t b}, \texttt{acb\_poly\_t z},
\texttt{int regularized}, \texttt{slong n}, \texttt{slong prec})

Sets \texttt{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 \texttt{n}. The underscore method requires positive lengths and does not support aliasing.

### 7.1.9 Exponential and trigonometric integrals

The branch cut conventions of the following functions match Mathematica.

void \texttt{acb\_hypgeom\_expint}(\texttt{acb\_t res}, \texttt{acb\_t s}, \texttt{acb\_t z}, \texttt{slong prec})

Computes the generalized exponential integral \( E_a(z) \). This is a trivial wrapper of \texttt{acb\_hypgeom\_gamma\_upper()}.

void \texttt{acb\_hypgeom\_ei\_asymp}(\texttt{acb\_t res}, \texttt{acb\_t z}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_ei\_2f2}(\texttt{acb\_t res}, \texttt{acb\_t z}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_ei}(\texttt{acb\_t res}, \texttt{acb\_t z}, \texttt{slong prec})

Computes the exponential integral \( \text{Ei}(z) \), respectively using

\[
\text{Ei}(z) = -e^z U(1, 1, -z) - \log(-z) + \frac{1}{2} \left( \log(z) - \log \left( \frac{1}{z} \right) \right)
\]

\[
\text{Ei}(z) = z {}_2F_2(1, 1; 2, 2; z) + \gamma \left( \log(z) - \log \left( \frac{1}{z} \right) \right)
\]

and an automatic algorithm choice.

void \texttt{acb\_hypgeom\_ei\_series}(\texttt{acb\_ptr res}, \texttt{acb\_srcptr z}, \texttt{slong zlen}, \texttt{slong len}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_ei\_series}(\texttt{acb\_poly\_t res}, \texttt{acb\_poly\_t z}, \texttt{slong len}, \texttt{slong prec})

Computes the exponential integral of the power series \texttt{z}, truncated to length \texttt{len}.

void \texttt{acb\_hypgeom\_si\_asymp}(\texttt{acb\_t res}, \texttt{acb\_t z}, \texttt{slong prec})

void \texttt{acb\_hypgeom\_si\_1f2}(\texttt{acb\_t res}, \texttt{acb\_t z}, \texttt{slong prec})
void \texttt{acb_hypgeom_si}(\texttt{acb} \_ \texttt{t} \texttt{res}, \texttt{const} \ \texttt{acb} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{prec})
Computes the sine integral Si(z), respectively using
\[
Si(z) = \frac{i}{2} \left[ e^{iz}U(1,1,-iz) - e^{-iz}U(1,1,iz) + \log(-iz) - \log(iz) \right]
\]
and an automatic algorithm choice.

void \_acb_hypgeom_si_series(\texttt{acb} \_ \texttt{ptr} \ \texttt{res}, \ \texttt{acb} \_ \texttt{srcptr} \ \texttt{z}, \ \texttt{slong} \ \texttt{zlen}, \ \texttt{slong} \ \texttt{len}, \ \texttt{slong} \ \texttt{prec})

void \texttt{acb_hypgeom_si_series}(\texttt{acb} \_ \texttt{poly} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{poly} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{len}, \ \texttt{slong} \ \texttt{prec})
Computes the sine integral of the power series z, truncated to length len.

void \texttt{acb_hypgeom_ci_asym}(\texttt{acb} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{prec})

void \texttt{acb_hypgeom_ci_2f3}(\texttt{acb} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{prec})

void \texttt{acb_hypgeom_ci}(\texttt{acb} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{prec})
Computes the cosine integral Ci(z), respectively using
\[
Ci(z) = \log(z) - \frac{1}{2} \left[ e^{iz}U(1,1,-iz) + e^{-iz}U(1,1,iz) + \log(-iz) + \log(iz) \right]
\]
and an automatic algorithm choice.

void \_acb_hypgeom_ci_series(\texttt{acb} \_ \texttt{ptr} \ \texttt{res}, \ \texttt{acb} \_ \texttt{srcptr} \ \texttt{z}, \ \texttt{slong} \ \texttt{zlen}, \ \texttt{slong} \ \texttt{len}, \ \texttt{slong} \ \texttt{prec})

void \texttt{acb_hypgeom_ci_series}(\texttt{acb} \_ \texttt{poly} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{poly} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{len}, \ \texttt{slong} \ \texttt{prec})
Computes the cosine integral of the power series z, truncated to length len.

void \texttt{acb_hypgeom_shi}(\texttt{acb} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{prec})
Computes the hyperbolic sine integral Shi(z) = -iSi(iz). This is a trivial wrapper of \texttt{acb_hypgeom_si}().

void \_acb_hypgeom_shi_series(\texttt{acb} \_ \texttt{ptr} \ \texttt{res}, \ \texttt{acb} \_ \texttt{srcptr} \ \texttt{z}, \ \texttt{slong} \ \texttt{zlen}, \ \texttt{slong} \ \texttt{len}, \ \texttt{slong} \ \texttt{prec})

void \texttt{acb_hypgeom_shi_series}(\texttt{acb} \_ \texttt{poly} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{poly} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{len}, \ \texttt{slong} \ \texttt{prec})
Computes the hyperbolic sine integral of the power series z, truncated to length len.

void \texttt{acb_hypgeom_chiaasyn}(\texttt{acb} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{prec})

void \texttt{acb_hypgeom_chiaasyn}(\texttt{acb} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{prec})

void \texttt{acb_hypgeom_chi}(\texttt{acb} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{prec})
Computes the hyperbolic cosine integral Chi(z), respectively using
\[
\text{Chi}(z) = -\frac{1}{2} \left[ e^zU(1,1,-z) + e^{-z}U(1,1,z) + \log(-z) - \log(z) \right]
\]
and an automatic algorithm choice.

void \_acb_hypgeom_chi_series(\texttt{acb} \_ \texttt{ptr} \ \texttt{res}, \ \texttt{acb} \_ \texttt{srcptr} \ \texttt{z}, \ \texttt{slong} \ \texttt{zlen}, \ \texttt{slong} \ \texttt{len}, \ \texttt{slong} \ \texttt{prec})

void \texttt{acb_hypgeom_chi_series}(\texttt{acb} \_ \texttt{poly} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{poly} \_ \texttt{t} \ \texttt{z}, \ \texttt{slong} \ \texttt{len}, \ \texttt{slong} \ \texttt{prec})
Computes the hyperbolic cosine integral of the power series z, truncated to length len.

void \texttt{acb_hypgeom_li}(\texttt{acb} \_ \texttt{t} \ \texttt{res}, \ \texttt{const} \ \texttt{acb} \_ \texttt{t} \ \texttt{z}, \ \texttt{int} \ \texttt{offset}, \ \texttt{slong} \ \texttt{prec})
If \texttt{offset} is zero, computes the logarithmic integral li(z) = Ei(log(z)).
If \texttt{offset} is nonzero, computes the offset logarithmic integral Li(z) = li(z) - li(2).
The following methods compute the Gauss hypergeometric function

\[ F(z) = _2F_1(a, b, c, z) = \sum_{k=0}^{\infty} \frac{(a)_k(b)_k}{(c)_k} \frac{z^k}{k!} \]

or the regularized version \( F(z) = _2F_1(a, b, c, z) = _2F_1(a, b, c, z)/\Gamma(c) \) if the flag \texttt{regularized} is set.

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

### 7.1.10 Gauss hypergeometric function

The following methods compute the Gauss hypergeometric function

\[ F(z) = _2F_1(a, b, c, z) = \sum_{k=0}^{\infty} \frac{(a)_k(b)_k}{(c)_k} \frac{z^k}{k!} \]

Computes \( F(z) \) of the given power series truncated to length \( \text{len} \), using direct summation of the hypergeometric series.

```c
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 \( \text{res0} \) and \( \text{res1} \) 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.

```c
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 \( \text{len} \), using direct summation of the hypergeometric series.

```c
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.

```c
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 \texttt{which}. Legal values are 1 for \( 1/(z-1) \), 2 for \( 1/z \), 3 for \( 1/(1-z) \), 4 for \( 1-z \), and 5 for \( 1-1/z \).

The \texttt{transform_limit} version assumes that \texttt{which} is not 1. If \texttt{which} is 2 or 3, it assumes that \( b-a \) represents an exact integer. If \texttt{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 \texttt{acb_hypgeom_2f1()} for the meaning of \texttt{flags}.

```c
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.

```c
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 \texttt{acb_hypgeom_2f1_transform()} should be used. If the return value is 6, the corner case algorithm should be used.

```c
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 \( F(z) \) using an automatic algorithm choice.

The following bit fields can be set in \texttt{flags}:
• **ACB\_HYPGEOM\_2F1\_REGULARIZED** - computes the regularized hypergeometric function $F(z)$. Setting flags to 1 is the same as just toggling this option.

• **ACB\_HYPGEOM\_2F1\_AB** - $a - b$ is an integer.

• **ACB\_HYPGEOM\_2F1\_ABC** - $a + b - c$ is an integer.

• **ACB\_HYPGEOM\_2F1\_AC** - $a - c$ is an integer.

• **ACB\_HYPGEOM\_2F1\_BC** - $b - c$ is an integer.

The last four flags can be set to indicate that the respective parameter differences are known to represent exact integers, even if the input intervals are inexact. This allows the correct limits to be evaluated when applying transformation formulas. For example, to evaluate $2F1(\sqrt{2}, 1/2, \sqrt{2} + 3/2, 9/10)$, the **ABC** flag should be set. If not set, the result will be an indeterminate interval due to internally dividing by an interval containing zero. If the parameters are exact floating-point numbers (including exact integers or half-integers), then the limits are computed automatically, and setting these flags is unnecessary.

Currently, only the **AB** and **ABC** flags are used this way; the **AC** and **BC** flags might be used in the future.

### 7.1.11 Orthogonal polynomials and functions

**void acb_hypgeom_chebyshev_t(acb_t res, const acb_t n, const acb_t z, slong prec)**

Computes the Chebyshev polynomial (or Chebyshev function) of first or second kind

$$T_n(z) = 2F1(-n, n, 1/2, 1 - z/2)$$

$$U_n(z) = (n+1)2F1(-n, n+2, 3/2, 1 - z/2).$$

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)

$$P_n^{(a,b)}(z) = \frac{(a+1)_n}{\Gamma(n+1)} 2F1(-n, n+a+b+1, a+1, 1 - z/2).$$

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)

$$C_n^m(z) = \frac{(2m)_n}{\Gamma(n+1)} 2F1(-n, 2m+n, m+1, 1 - z/2).$$

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)
  \[ L_n^m(z) = \frac{(m + 1)^n}{(n + 1)!} F_1 \left( -n, m + 1, z \right). \]
  For nonnegative integer \( n \), this is a polynomial in \( m \) and \( z \), even when the parameters are such
  that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct
  methods.
  There are at least two incompatible ways to define the Laguerre function when \( n \) is a negative
  integer. One possibility when \( m = 0 \) is to define \( L_n^0(z) = e^z L_n^0(-z) \). Another possibility is to
  cover this case with the recurrence relation \( L_n^m(z) + L_n^{-m}(z) = L_n^0(z) \). 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)
  \[ H_n(z) = 2^n \sqrt{\pi} \frac{1}{\Gamma((1-n)/2)} F_1 \left( -\frac{n}{2} \cdot \frac{1}{2}, z^2 \right) - \frac{2z}{\Gamma((1-n)/2)} F_1 \left( \frac{1}{2} \cdot \frac{3}{2}, \frac{3}{2}, z^2 \right). \]

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
  \[ P_n^m(z) = \frac{(1 + z)^{m/2}}{(1 - z)^{m/2}} F \left( -n, n + 1, 1 - m, \frac{1 - z}{2} \right) \]
  is computed, and if \( type \) is 1, the alternative version
  \[ P_n^m(z) = \frac{(z + 1)^{m/2}}{(z - 1)^{m/2}} F \left( -n, n + 1, 1 - m, \frac{1 - z}{2} \right). \]
  is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in \texttt{Mathematica} and
  \texttt{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
  \[ Q_n^m(z) = \frac{\pi}{2 \sin(\pi m)} \left( \cos(\pi m) P_n^m(z) - \Gamma(1 + m + n) P_n^{-m}(z) \right) \]
  is computed, and if \( type \) is 1, the alternative version
  \[ Q_n^m(z) = \frac{\pi}{2 \sin(\pi m)} \left( P_n^m(z) - \Gamma(1 + m + n) P_n^{-m}(z) \right) \]
  is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in \texttt{Mathematica} and
  \texttt{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 \texttt{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 \texttt{acb\_hypgeom\_legendre\_p\_uui\_rec()}. 

\section*{7.2 \texttt{arb\_hypgeom.h} – hypergeometric functions of real variables}

See \texttt{acb\_hypgeom.h} – hypergeometric functions of complex variables for the implementation of hypergeometric functions.

For convenience, this module provides corresponding functions for direct use with the real types \texttt{arb\_t} and \texttt{arb\_poly\_t}. Most methods are simple wrappers around the complex versions, with a tiny amount of extra overhead for conversions, but in some cases the functions in this module will be faster. In the future, code that is further optimized specifically for real variables might be added to this module.

\subsection*{7.2.1 Generalized hypergeometric function}

void \texttt{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 \texttt{regularized} is set.

\subsection*{7.2.2 Confluent hypergeometric functions}

void \texttt{arb\_hypgeom\_0f1}(arb\_t res, const arb\_t a, const arb\_t b, 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 \texttt{regularized} is set.

void \texttt{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 \( \frac{1}{\Gamma(b)}_1F_1(a, b, z) \) if \texttt{regularized} is set.

void \texttt{arb\_hypgeom\_1f1}(arb\_t res, const arb\_t a, const arb\_t b, const arb\_t z, int regularized, slong prec)

Alias for \texttt{arb\_hypgeom\_m()}. 

void \texttt{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 \( U(a, b, z) \).

\subsection*{7.2.3 Gauss hypergeometric function}

void \texttt{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 \( \frac{1}{\Gamma(c)}_2F_1(a, b, c, z) \) if \texttt{regularized} is set.

Additional evaluation flags can be passed via the \texttt{regularized} argument; see \texttt{arb\_hypgeom\_2f1()} for documentation.
### 7.2.4 Error functions and Fresnel integrals

```c
void arb_hypgeom_erf (arb_t res, const arb_t z, slong prec)
```

Computes the error function \( \text{erf}(z) \).

```c
void _arb_hypgeom_erf_series (arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

Computes the error function of the power series \( z \), truncated to length \( len \).

```c
void arb_hypgeom_erf_series (arb_poly_t res, const arb_poly_t z, slong len, slong prec)
```

Computes the error function \( \text{erf}(z) \) of the power series \( z \), truncated to length \( len \).

```c
void arb_hypgeom_erfc (arb_t res, const arb_t z, slong prec)
```

Computes the complementary error function \( \text{erfc}(z) = 1 - \text{erf}(z) \). This function avoids catastrophic cancellation for large positive \( z \).

```c
void _arb_hypgeom_erfc_series (arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

Computes the complementary error function of the power series \( z \), truncated to length \( len \).

```c
void arb_hypgeom_erfc_series (arb_poly_t res, const arb_poly_t z, slong len, slong prec)
```

Computes the complementary error function \( \text{erfc}(z) \) of the power series \( z \), truncated to length \( len \).

```c
void arb_hypgeom_erfi (arb_t res, const arb_t z, slong prec)
```

Computes the imaginary error function \( \text{erfi}(z) = -i \text{erf}(iz) \).

```c
void _arb_hypgeom_erfi_series (arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

Computes the imaginary error function \( \text{erfi}(z) \) of the power series \( z \), truncated to length \( len \).

```c
void arb_hypgeom_erfi_series (arb_poly_t res, const arb_poly_t z, slong len, slong prec)
```

Computes the imaginary error function \( \text{erfi}(z) \) of the power series \( z \), truncated to length \( len \).

```c
void arb_hypgeom_fresnel (arb_t res1, arb_t res2, const arb_t z, int normalized, slong prec)
```

Sets \( \text{res1} \) to the Fresnel sine integral \( S(z) \) and \( \text{res2} \) 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 \( \text{normalized} \) is 0, and
\[
S(z) = \int_0^z \sin \left( \frac{1}{2} \pi t^2 \right) dt
\]
is used if \( \text{normalized} \) is 1 (the latter is the Abramowitz & Stegun convention). \( C(z) \) is defined analogously.

```c
void _arb_hypgeom_fresnel_series (arb_ptr res1, arb_ptr res2, arb_srcptr z, slong zlen, int normalized, slong len, slong prec)
```

Sets \( \text{res1} \) to the Fresnel sine integral and \( \text{res2} \) to the Fresnel cosine integral of the power series \( z \), truncated to length \( len \). Optionally, just a single function can be computed by passing \( NULL \) as the other output variable.

### 7.2.5 Incomplete gamma and beta functions

```c
void arb_hypgeom_gamma_upper (arb_t res, const arb_t s, const arb_t z, int regularized, slong prec)
```

If \( \text{regularized} \) is 0, computes the upper incomplete gamma function \( \Gamma(s, z) \).

If \( \text{regularized} \) is 1, computes the regularized upper incomplete gamma function \( Q(s, z) = \frac{\Gamma(s, z)}{\Gamma(s)} \).

If \( \text{regularized} \) 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; \( \text{arb_hypgeom_expint}() \) is the intended interface for computing the exponential integral).

```c
void _arb_hypgeom_gamma_upper_series (arb_ptr res, const arb_t s, arb_srcptr z, slong zlen, int regularized, slong n, slong prec)
```

Sets \( \text{res} \) to an upper incomplete gamma function where \( s \) is a constant and \( z \) is a power series, truncated to length \( n \). The \( \text{regularized} \) argument has the same interpretation as in \( \text{arb_hypgeom_gamma_upper}() \).

```c
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 \( \text{res} \) to an upper incomplete gamma function where \( s \) is a constant and \( z \) is a power series, truncated to length \( n \). The \( \text{regularized} \) argument has the same interpretation as in \( \text{arb_hypgeom_gamma_upper}() \).
void \texttt{arb\_hypgeom\_gamma\_lower}(\texttt{arb\_t res}, \texttt{const arb\_t s}, \texttt{const arb\_t z}, \texttt{int regularized}, \texttt{slong prec})

If \texttt{regularized} is 0, computes the lower incomplete gamma function $\gamma(s, z) = \frac{1}{\Gamma(s)} \int_0^z t^{s-1} e^{-t} \, dt$.

If \texttt{regularized} is 1, computes the regularized lower incomplete gamma function $P(s, z) = \frac{\gamma(s, z)}{\Gamma(s)}$.

If \texttt{regularized} is 2, computes a further regularized lower incomplete gamma function $\gamma^*(s, z) = z^{-s} P(s, z)$.

void \texttt{arb\_hypgeom\_gamma\_lower\_series}(\texttt{arb\_ptr res}, \texttt{const arb\_t s}, \texttt{arb\_srcreptr z}, \texttt{slong zlen}, \texttt{int regularized}, \texttt{slong n}, \texttt{slong prec})

Sets \texttt{res} to an lower incomplete gamma function, truncated to length \texttt{n}. The \texttt{regularized} argument has the same interpretation as in \texttt{arb\_hypgeom\_gamma\_lower(\ldots)}.

void \texttt{arb\_hypgeom\_beta\_lower}(\texttt{arb\_t res}, \texttt{const arb\_t a}, \texttt{const arb\_t b}, \texttt{const arb\_t z}, \texttt{int regularized}, \texttt{slong prec})

Computes the (lower) incomplete beta function, defined by $B(a, b; z) = \int_0^z t^{a-1} (1-t)^{b-1} \, dt$.

void \texttt{arb\_hypgeom\_beta\_lower\_series}(\texttt{arb\_ptr res}, \texttt{const arb\_t a}, \texttt{const arb\_t b}, \texttt{arb\_srcreptr z}, \texttt{slong zlen}, \texttt{int regularized}, \texttt{slong n}, \texttt{slong prec})

void \texttt{arb\_hypgeom\_beta\_lower\_series}(\texttt{arb\_poly\_t res}, \texttt{const arb\_t a}, \texttt{const arb\_t b}, \texttt{const arb\_poly\_t z}, \texttt{int regularized}, \texttt{slong n}, \texttt{slong prec})

Sets \texttt{res} to the lower incomplete beta function, truncated to length \texttt{n}. The underscore method requires positive lengths and does not support aliasing.

### 7.2.6 Exponential and trigonometric integrals

void \texttt{arb\_hypgeom\_expint}(\texttt{arb\_t res}, \texttt{const arb\_t s}, \texttt{const arb\_t z}, \texttt{slong prec})

Computes the generalized exponential integral $E_a(z)$.

void \texttt{arb\_hypgeom\_ei}(\texttt{arb\_t res}, \texttt{const arb\_t z}, \texttt{slong prec})

Computes the exponential integral $Ei(z)$.

void \texttt{arb\_hypgeom\_ei\_series}(\texttt{arb\_ptr res}, \texttt{arb\_srcreptr z}, \texttt{slong zlen}, \texttt{slong len}, \texttt{slong prec})

void \texttt{arb\_hypgeom\_ei\_series}(\texttt{arb\_poly\_t res}, \texttt{const arb\_poly\_t z}, \texttt{slong len}, \texttt{slong prec})

Computes the exponential integral of the power series, truncated to length \texttt{len}.

void \texttt{arb\_hypgeom\_si}(\texttt{arb\_t res}, \texttt{const arb\_t z}, \texttt{slong prec})

Computes the sine integral $Si(z)$.

void \texttt{arb\_hypgeom\_si\_series}(\texttt{arb\_ptr res}, \texttt{arb\_srcreptr z}, \texttt{slong zlen}, \texttt{slong len}, \texttt{slong prec})

void \texttt{arb\_hypgeom\_si\_series}(\texttt{arb\_poly\_t res}, \texttt{const arb\_poly\_t z}, \texttt{slong len}, \texttt{slong prec})

Computes the sine integral of the power series, truncated to length \texttt{len}.

void \texttt{arb\_hypgeom\_ci}(\texttt{arb\_t res}, \texttt{const arb\_t z}, \texttt{slong prec})

Computes the cosine integral $Ci(z)$. The result is indeterminate if $z < 0$ since the value of the function would be complex.

void \texttt{arb\_hypgeom\_ci\_series}(\texttt{arb\_ptr res}, \texttt{arb\_srcreptr z}, \texttt{slong zlen}, \texttt{slong len}, \texttt{slong prec})

void \texttt{arb\_hypgeom\_ci\_series}(\texttt{arb\_poly\_t res}, \texttt{const arb\_poly\_t z}, \texttt{slong len}, \texttt{slong prec})

Computes the cosine integral of the power series, truncated to length \texttt{len}.

void \texttt{arb\_hypgeom\_shi}(\texttt{arb\_t res}, \texttt{const arb\_t z}, \texttt{slong prec})

Computes the hyperbolic sine integral $Shi(z) = -i Si(iz)$.

void \texttt{arb\_hypgeom\_shi\_series}(\texttt{arb\_ptr res}, \texttt{arb\_srcreptr z}, \texttt{slong zlen}, \texttt{slong len}, \texttt{slong prec})

132 Chapter 7. Higher mathematical functions
void \texttt{arb\_hypgeom\_shifser\_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 \texttt{arb\_hypgeom\_chi}(arb\_t res, const arb\_t z, slong prec)
   Computes the hyperbolic cosine integral \( \text{Chi}(z) \). The result is indeterminate if \( z < 0 \) since the value of the function would be complex.

void \texttt{\_arb\_hypgeom\_chi\_series}(arb\_ptr res, arb\_srcptr z, slong zlen, slong len, slong prec)

void \texttt{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 \texttt{arb\_hypgeom\_li}(arb\_t res, const arb\_t z, int offset, slong prec)
   If \( offset \) is zero, computes the logarithmic integral \( \text{li}(z) = \text{Ei}(\log(z)) \).
   If \( offset \) is nonzero, computes the offset logarithmic integral \( \text{Li}(z) = \text{li}(z) - \text{li}(2) \).
   The result is indeterminate if \( z < 0 \) since the value of the function would be complex.

void \texttt{\_arb\_hypgeom\_li\_series}(arb\_ptr res, arb\_srcptr z, slong zlen, int offset, slong len, slong prec)

void \texttt{arb\_hypgeom\_li\_series}(arb\_poly\_t res, const arb\_poly\_t z, int offset, slong len, slong prec)
   Computes the logarithmic integral (optionally the offset version) of the power series \( z \), truncated to length \( len \).

7.2.7 Bessel functions

void \texttt{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 \texttt{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 \texttt{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 \texttt{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 \( I_\nu(z) = z^\nu(i z)^{-\nu} J_\nu(iz) \).

void \texttt{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 \( K_\nu(z) \).

7.2.8 Airy functions

void \texttt{arb\_hypgeom\_airy}(arb\_t ai, arb\_t ai\_prime, arb\_t bi, arb\_t bi\_prime, const arb\_t z, slong prec)
   Computes the Airy functions \( (Ai(z), Ai'(z), Bi(z), Bi'(z)) \) simultaneously. Any of the four function values can be omitted by passing \texttt{NULL} for the unwanted output variables, speeding up the evaluation.

void \texttt{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 \texttt{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 \texttt{\_arb\_poly\_derivative()}.

void \texttt{\_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.

### 7.2.9 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.

### 7.3 acb_modular.h – modular forms of complex variables

This module provides methods for numerical evaluation of modular forms, Jacobi theta functions, and elliptic functions.

In the context of this module, $\tau$ or $\tau$ always denotes an element of the complex upper half-plane $\mathbb{H} = \{ z \in \mathbb{C} : \text{Im}(z) > 0 \}$. We also often use the variable $q$, variously defined as $q = e^{2\pi \tau}$ (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 $\tau$.

As usual, the numerical functions in this module compute strict error bounds: if $\tau$ is represented by an `acb_t` whose content overlaps with the real line (or lies in the lower half-plane), and $\tau$ is passed to a function defined only on $\mathbb{H}$, then the output will have an infinite radius. The analogous behavior holds for functions requiring $|q| < 1$.

#### 7.3.1 The modular group

**psl2z_struct**

**psl2z_t**

Represents an element of the modular group $\text{PSL}(2, \mathbb{Z})$, 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 PSL(2, \( \mathbb{Z} \)) with entries of bit length at most bits (or 1, if bits is not positive). We first generate \( a \) and \( d \), compute their Bezout coefficients, divide by the GCD, and then correct the signs.

7.3.2 Modular transformations

void acb_modular_transform(acb_t w, const psl2z_t g, const acb_t z, slong prec)
    Applies the modular transformation \( \gamma \) to the complex number \( z \), evaluating

\[
    w = gz = \frac{az + b}{cz + d},
\]

void acb_modular_fundamental_domain_approx_d(psl2z_t g, double x, double y, double one_minus_eps)
    Attempts to determine a modular transformation \( \gamma \) 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.

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 \texttt{acb_modular\_fundamental\_domain\_approx}(\texttt{acb\_t} w, \texttt{psl2\_z\_t} g, \texttt{const} \texttt{acb\_t} z, \texttt{const} \texttt{arf\_t} one\_minus\_eps, \texttt{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 \texttt{one\_minus\_eps}. It also computes the transformed value \( w = gz \).

This function first tries to use \texttt{acb_modular\_fundamental\_domain\_approx\_d()} and checks if the result is acceptable. If this fails, it calls \texttt{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 \texttt{acb_modular\_is\_in\_fundamental\_domain}(\texttt{const} \texttt{acb\_t} z, \texttt{const} \texttt{arf\_t} tol, \texttt{slong} prec)

Returns nonzero if it is certainly true that \(|z| \geq 1 - \varepsilon \) and \(|\text{Re}(z)| \leq 1/2 + \varepsilon \) where \( \varepsilon \) is specified by \texttt{tol}. Returns zero if this is false or cannot be determined.

### 7.3.3 Addition sequences

void \texttt{acb_modular\_fill\_addseq}(\texttt{slong} * tab, \texttt{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 \texttt{tab} to \(-1\) if that index is to be part of the addition sequence, and to 0 otherwise. On output, entry \( i \) in \texttt{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 \texttt{tab} are ignored (the number 1 is always assumed to be part of the sequence).

### 7.3.4 Jacobi theta functions

Unfortunately, there are many inconsistent notational variations for Jacobi theta functions in the literature. Unless otherwise noted, we use the functions

\[
\theta_1(z, \tau) = -i \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i [(n + 1/2)^2 \tau + (2n + 1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} (-1)^n q^{n(n+1)} \sin((2n + 1)\pi z)
\]

\[
\theta_2(z, \tau) = \sum_{n=-\infty}^{\infty} \exp(\pi i [(n + 1/2)^2 \tau + (2n + 1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos((2n + 1)\pi z)
\]

\[
\theta_3(z, \tau) = \sum_{n=-\infty}^{\infty} \exp(\pi i [n^2 \tau + 2nz]) = 1 + 2 \sum_{n=1}^{\infty} q^{n^2} \cos(2n\pi z)
\]

\[
\theta_4(z, \tau) = \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i [n^2 \tau + 2nz]) = 1 + 2 \sum_{n=1}^{\infty} (-1)^n q^{n^2} \cos(2n\pi z)
\]

where \( q = \exp(\pi i \tau) \) and \( q_{1/4} = \exp(\pi i \tau/4) \). Note that many authors write \( q_{1/4} \) as \( q^{1/4} \), but the principal fourth root \( (q)^{1/4} = \exp(\frac{1}{4} \log q) \) differs from \( q_{1/4} \) in general and some formulas are only correct if one reads “\( q^{1/4} = \exp(\pi i \tau/4) \)”. To avoid confusion, we only write \( q^k \) when \( k \) is an integer.

void \texttt{acb_modular\_theta\_transform}(\texttt{int} * R, \texttt{int} * S, \texttt{int} * C, \texttt{const} \texttt{psl2\_z\_t} g)

We wish to write a theta function with quasiperiod \( \tau' \) in terms of a theta function with quasiperiod \( \tau' = g\tau \), given some \( g = (a, b, c, d) \in \text{PSL}(2, \mathbb{Z}) \). For \( i = 0, 1, 2, 3 \), this function computes integers \( R_i \) and \( S_i \) (\( R \) and \( S \) should be arrays of length 4) and \( C \in \{0, 1\} \) such that

\[
\theta_{1+i}(z, \tau) = \exp(\pi i R_i/4) \cdot A \cdot B \cdot \theta_{1+S_i}(z', \tau')
\]
where $z' = z, A = B = 1$ if $C = 0$, and

$$z' = \frac{-z}{c\tau + d}, \quad A = \sqrt{\frac{i}{c\tau + d}}, \quad B = \exp\left(-\pi ic\frac{z^2}{c\tau + d}\right)$$

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 $\theta_i(z, \tau) = \exp(-\pi ib/4)\theta_i(z, \tau + b)$ for $i = 1, 2$, whereas $\theta_3$ and $\theta_4$ remain unchanged when $b$ is even and swap places with each other when $b$ is odd. In this case we set $C = 0$.

For an arbitrary $g$ with $c > 0$, we set $C = 1$. The general transformations are given by Rademacher [Rad1973]. We need the function $\theta_{m,n}(z, \tau)$ defined for $m, n \in \mathbb{Z}$ by (beware of the typos in [Rad1973])

$$\theta_{0,0}(z, \tau) = \theta_3(z, \tau), \quad \theta_{0,1}(z, \tau) = \theta_4(z, \tau)$$

$$\theta_{1,0}(z, \tau) = \theta_2(z, \tau), \quad \theta_{1,1}(z, \tau) = i\theta_1(z, \tau)$$

$$\theta_{m+2,n}(z, \tau) = (-1)^n\theta_{m,n}(z, \tau)$$

$$\theta_{m,n+2}(z, \tau) = \theta_{m,n}(z, \tau).$$

Then we may write

$$\theta_1(z, \tau) = \varepsilon_1 AB\theta_1(z', \tau')$$

$$\theta_2(z, \tau) = \varepsilon_2 AB\theta_1_{c-1+a}(z', \tau')$$

$$\theta_3(z, \tau) = \varepsilon_3 AB\theta_1_{d-b+1-c}(z', \tau')$$

$$\theta_4(z, \tau) = \varepsilon_4 AB\theta_1_{d+1-b}(z', \tau')$$

where $\varepsilon_i$ is an 8th root of unity. Specifically, if we denote the 24th root of unity in the transformation formula of the Dedekind eta function by $\varepsilon(a, b, c, d) = \exp(\pi i R(a, b, c, d)/12)$ (see \texttt{acb_modular_epson_4}(\_)), then:

$$\varepsilon_1(a, b, c, d) = \exp(\pi i [R(-d, b, c, -a) + 1]/4)$$

$$\varepsilon_2(a, b, c, d) = \exp(\pi i [-R(a, b, c, d) + (5 + (2 - c)a)]/4)$$

$$\varepsilon_3(a, b, c, d) = \exp(\pi i [-R(a, b, c, d) + (4 + (c - d - 2)(b - a))])/4)$$

$$\varepsilon_4(a, b, c, d) = \exp(\pi i [-R(a, b, c, d) + (3 - (2 + d)b)]/4)$$

These formulas are easily derived from the formulas in [Rad1973] (Rademacher has the transformed/untransformed variables exchanged, and his “$e$” differs from ours by a constant offset in the phase).

\textbf{void \texttt{acb_modular_addseq_theta}(slong \* \texttt{exponents}, slong \* \texttt{aindex}, slong \* \texttt{bindex}, slong \* \texttt{num})}

Constructs an addition sequence for the first \texttt{num} squares and triangular numbers interleaved (excluding zero), i.e. 1, 2, 4, 6, 9, 12, 16, 20, 25, 30 etc.

\textbf{void \texttt{acb_modular_theta_sum}(acb_ptr \texttt{theta1}, acb_ptr \texttt{theta2}, acb_ptr \texttt{theta3}, acb_ptr \texttt{theta4},

\texttt{const acb_t \texttt{w}, int \texttt{w_is_unit}, const acb_t \texttt{q}, slong \texttt{len}, slong \texttt{prec})}

Simultaneously computes the first \texttt{len} coefficients of each of the formal power series

$$\theta_1(z + x, \tau)/q^{1/4} \in \mathbb{C}[x]$$

$$\theta_2(z + x, \tau)/q^{1/4} \in \mathbb{C}[x]$$

$$\theta_3(z + x, \tau) \in \mathbb{C}[x]$$

$$\theta_4(z + x, \tau) \in \mathbb{C}[x]$$
given \( w = \exp(\pi i z) \) and \( q = \exp(\pi i r) \), by summing a finite truncation of the respective theta function series. In particular, with \( \text{len} \) equal to 1, computes the respective value of the theta function at the point \( z \). We require \( \text{len} \) to be positive. If \( \text{w\_is\_unit} \) is nonzero, \( w \) is assumed to lie on the unit circle, i.e. \( z \) is assumed to be real.

Note that the factor \( q_{1/4} \) is removed from \( \theta_1 \) and \( \theta_2 \). To get the true theta function values, the user has to multiply this factor back. This convention avoids unnecessary computations, since the user can compute \( q_{1/4} = \exp(\pi i r / 4) \) followed by \( q = (q_{1/4})^4 \), and in many cases when computing products or quotients of theta functions, the factor \( q_{1/4} \) can be eliminated entirely.

This function is intended for \( |q| \ll 1 \). It can be called with any \( q \), but will return useless intervals if convergence is not rapid. For general evaluation of theta functions, the user should only call this function after applying a suitable modular transformation.

We consider the sums together, alternatingly updating \( (\theta_1, \theta_2) \) or \( (\theta_3, \theta_4) \). For \( k = 0, 1, 2, \ldots \), the powers of \( q \) are \([k^2 + 4]/4\) = 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:

\[
\begin{array}{lcc}
  \theta_1, \theta_2 & q^0 & (w^1 \pm w^{-1}) \\
  k = 0 & \theta_3, \theta_4 & q^1 & (w^2 \pm w^{-2}) \\
  k = 1 & \theta_1, \theta_2 & q^2 & (w^3 \pm w^{-3}) \\
  k = 2 & \theta_3, \theta_4 & q^4 & (w^4 \pm w^{-4}) \\
  k = 3 & \theta_1, \theta_2 & q^6 & (w^5 \pm w^{-5}) \\
  k = 4 & \theta_3, \theta_4 & q^9 & (w^6 \pm w^{-6}) \\
  k = 5 & \theta_1, \theta_2 & q^{12} & (w^7 \pm w^{-7}) \\
\end{array}
\]

For some integer \( N \geq 1 \), the summation is stopped just before term \( k = N \). Let \( Q = |q| \), \( W = \max(|w|, |w^{-1}|) \), \( E = [(N + 2)^2/4] \) and \( F = [(N + 1)/2] + 1 \). 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 \( \text{len} \) is greater than 1, consider the derivative of order \( r \). The term of index \( k \) and order \( r \) picks up a factor of magnitude \((k + 2)^r\) from differentiation of \( w^{k+2} \) (it also picks up a factor \( \pi^r \), but we omit this until we rescale the coefficients at the end of the computation). Thus we have the error bound

\[
2Q^E W^{N+2} (N + 2)^r \left[ 1 + Q^F W \frac{(N + 3)^r}{(N + 2)^r} + Q^{2F} W^2 \frac{(N + 4)^r}{(N + 2)^r} + \ldots \right]
\]

which by the inequality \((1 + m/(N + 2))^r \leq \exp(mr/(N + 2))\) can be bounded as

\[
\frac{2Q^E W^{N+2} (N + 2)^r}{1 - Q^F W \exp(r/(N + 2))}
\]

again valid when the denominator is positive.

To actually evaluate the series, we write the even cosine terms as \( w^{2n} + w^{-2n} \), the odd cosine terms as \( w(w^{2n} - w^{-2n-2}) \), and the sine terms as \( w(w^{2n} - w^{-2n-2}) \). This way we only need even powers of \( w \) and \( w^{-1} \). The implementation is not yet optimized for real \( z \), in which case further work can be saved.

This function does not permit aliasing between input and output arguments.

```c
void acb_modular_theta_const_sum_basecase(acb_t theta2, acb_t theta3, acb_t theta4, const acb_t q, slong N, slong prec)

void acb_modular_theta_const_sum_rs(acb_t theta2, acb_t theta3, acb_t theta4, const acb_t q, slong N, slong prec)
```

Computes the truncated theta constant sums \( \theta_2 = \sum_{k(k+1) < N} q^{k(k+1)} \), \( \theta_3 = \sum_{k^2 < N} q^k \), \( \theta_4 = \sum_{k^3 < N} (-1)^k q^{k^2} \). 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 \texttt{acb_modular_theta_const_sum_basecase()} or \texttt{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 \( \tau \) to the fundamental domain. This is generally worse than \texttt{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 \( \tau \) to the fundamental domain before calling \texttt{acb_modular_theta_sum()}.

### 7.3.5 The Dedekind eta function

void acb_modular_addeq_eta(slong * exponents, slong * aindex, slong * bindex, slong num)

Constructs an addition sequence for the first \( \text{num} \) 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.

\[
\exp(-\pi \tau / 12) \eta(\tau) = \sum_{n=-\infty}^{\infty} (-1)^n q^{(3n^2-n)/2}
\]

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 \( \tau \) in the upper half-plane. This function applies the functional equation to move \( \tau \) to the fundamental domain before calling \texttt{acb_modular_eta_sum()}.

### 7.3.6 Modular forms

void acb_modular_j(acb_t r, const acb_t tau, slong prec)

Computes Klein’s j-invariant \( j(\tau) \) given \( \tau \) in the upper half-plane. The function is normalized so that \( j(i) = 1728 \). We first move \( \tau \) to the fundamental domain, which does not change the value of the function. Then we use the formula

\[
j(\tau) = 32(\theta_2^4 + \theta_3^4 + \theta_4^4)/(\theta_2 \theta_3 \theta_4)^4
\]

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 \texttt{acb\_modular\_delta}(\texttt{acb\_t r}, \texttt{const acb\_t tau}, \texttt{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 \texttt{acb\_modular\_eisenstein}(\texttt{acb\_ptr r}, \texttt{const acb\_t tau}, \texttt{slong len}, \texttt{slong prec})

Computes simultaneously the first $\text{len}$ entries in the sequence of Eisenstein series $G_4(\tau), G_6(\tau), G_8(\tau), \ldots$, defined by

$$
G_{2k}(\tau) = \sum_{m^2+n^2 \neq 0} \frac{1}{(m+n\tau)^{2k}}
$$

and satisfying

$$
G_{2k} \left( \frac{a\tau + b}{c\tau + d} \right) = (c\tau + d)^{2k} G_{2k}(\tau).
$$

We first evaluate $G_4(\tau)$ and $G_6(\tau)$ on the fundamental domain using theta functions, and then compute the Eisenstein series of higher index using a recurrence relation.

### 7.3.7 Elliptic functions

void \texttt{acb\_modular\_elliptic\_p}(\texttt{acb\_t wp}, \texttt{const acb\_t z}, \texttt{const acb\_t tau}, \texttt{slong prec})

Computes Weierstrass’s elliptic function

$$
\wp(z, \tau) = \frac{1}{z^2} + \sum_{n^2+m^2 \neq 0} \left[ \frac{1}{(z+m+n\tau)^2} - \frac{1}{(m+n\tau)^2} \right]
$$

which satisfies $\wp(z, \tau) = \wp(z+1, \tau) = \wp(z+\tau, \tau)$. To evaluate the function efficiently, we use the formula

$$
\wp(z, \tau) = \pi^2 \theta_2^4(0, \tau) \theta_3^4(0, \tau) \frac{\theta_2^4(z, \tau)}{\theta_3^4(z, \tau)} - \frac{\pi^2}{3} \left[ \theta_3^4(0, \tau) + \theta_4^4(0, \tau) \right].
$$

void \texttt{acb\_modular\_elliptic\_p\_zpx}(\texttt{acb\_ptr wp}, \texttt{const acb\_t z}, \texttt{const acb\_t tau}, \texttt{slong len}, \texttt{slong prec})

Computes the formal power series $\wp(z+x, \tau) \in \mathbb{C}[[x]]$, truncated to length $\text{len}$. In particular, with $\text{len} = 2$, simultaneously computes $\wp(z, \tau), \wp'(z, \tau)$ which together generate the field of elliptic functions with periods 1 and $\tau$.

### 7.3.8 Elliptic integrals

void \texttt{acb\_modular\_elliptic\_k}(\texttt{acb\_t w}, \texttt{const acb\_t m}, \texttt{slong prec})

Computes the complete elliptic integral of the first kind $K(m)$, using the arithmetic-geometric mean: $K(m) = \pi/(2M(\sqrt{1-m}))$.

void \texttt{acb\_modular\_elliptic\_k\_cpx}(\texttt{acb\_ptr w}, \texttt{const acb\_t m}, \texttt{slong len}, \texttt{slong prec})

Sets the coefficients in the array $\text{w}$ to the power series expansion of the complete elliptic integral of the first kind at the point $m$ truncated to length $\text{len}$, i.e. $K(m+x) \in \mathbb{C}[[x]]$.

void \texttt{acb\_modular\_elliptic\_e}(\texttt{acb\_t w}, \texttt{const acb\_t m}, \texttt{slong prec})

Computes the complete elliptic integral of the second kind $E(m)$, which is given by $E(m) = (1-m)(2mK'(m) + K(m))$ (where the prime denotes a derivative, not a complementary integral).
7.3.9 Class polynomials

void acb_modular_hilbert_class_poly(fmpz_poly_t res, slong D)
Sets res to the Hilbert class polynomial of discriminant D, defined as

\[ H_D(x) = \prod_{(a,b,c)} \left( x - j \left( -b + \sqrt{D} \over 2a \right) \right) \]

where \((a,b,c)\) ranges over the primitive reduced positive definite binary quadratic forms of discriminant \(b^2 - 4ac = D\).

The Hilbert class polynomial is only defined if \(D < 0\) and \(D\) is congruent to 0 or 1 mod 4. If some other value of \(D\) is passed as input, \(res\) is set to the zero polynomial.

7.4 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, zeta functions, and related functions.

7.4.1 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

\[(\mathbb{Z}/q\mathbb{Z})^\times \times (\mathbb{Z}/q\mathbb{Z})^\times \mapsto \bigoplus_{i} \mathbb{Z}/\phi_i\mathbb{Z} \times \mathbb{Z}/\phi_i\mathbb{Z} \mapsto \chi_q(m,n) = \exp(2i\pi \sum a_i b_i)\]

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.

7.4.2 Multiplicative group modulo \(q\)

dirichlet_group_struct

dirichlet_group_t
Represents the group of Dirichlet characters mod \(q\).

An dirichlet_group_t is defined as an array of dirichlet_group_struct of length 1, permitting it to be passed by reference.

void dirichlet_group_init(dirichlet_group_t G, ulong q)
Initializes \(G\) to the group of Dirichlet characters mod \(q\).

This method computes a canonical decomposition of \(G\) in terms of cyclic groups, which are the mod \(p^e\) subgroups for \(p^e \parallel q\), plus the specific generator described by Conrey for each subgroup.

In particular \(G\) contains:

- the number num of components
- the generators
• the exponent \( \text{expo} \) 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.

```c
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.

```c
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).

```c
ulong dirichlet_group_size(const dirichlet_group_t G)
```

Returns the number of elements in \( G \), i.e. \( \varphi(q) \).

```c
ulong dirichlet_group_num_primitive(const dirichlet_group_t G)
```

Returns the number of primitive elements in \( G \).

```c
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 \( \text{num} \) calls to discrete logarithms.

If \( \text{num} \) gets very large, the entire group may be indexed.

```c
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.

### 7.4.3 Character type

```c
dirichlet_char_struct
```

```c
dirichlet_char_t
```

Represents a Dirichlet character. This structure contains both a \( \text{number} \) (residue class) and the corresponding \( \text{log} \) (exponents on the group generators).

An \( \text{dirichlet_char_t} \) is defined as an array of \( \text{dirichlet_char_struct} \) of length 1, permitting it to be passed by reference.

```c
void dirichlet_char_init(dirichlet_char_t chi, const dirichlet_group_t G)
```

Initializes \( \text{chi} \) to an element of the group \( G \) and sets its value to the principal character.

```c
void dirichlet_char_clear(dirichlet_char_t chi)
```

Clears \( \text{chi} \).

```c
void dirichlet_char_print(const dirichlet_group_t G, const dirichlet_char_t chi)
```

Prints the array of exponents representing this character.

```c
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 \( G \).

```c
ulong dirichlet_char_exp(const dirichlet_group_t G, const dirichlet_char_t x)
```

Returns the number \( m \) corresponding to exponents in \( x \).

```c
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.

```c
void dirichlet_char_one(dirichlet_char_t x, const dirichlet_group_t G)
```

Sets \( x \) to the principal character in \( G \), having \( \text{log} [0, \ldots 0] \).

```c
void dirichlet_char_first_primitive(dirichlet_char_t x, const dirichlet_group_t G)
```

Sets \( x \) to the first primitive character of \( G \), having \( \text{log} [1, \ldots 1] \), 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);
```

7.4.4 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 \( \lambda \) in \{0, 1\} of \( \chi_q(a, \cdot) \), such that \( \chi_q(a, -1) = (-1)^\lambda \).

ulong dirichlet_order_ui(const dirichlet_group_t G, ulong a)
ulong dirichlet_order_char(const dirichlet_group_t G, const dirichlet_char_t x)
Returns the order of \( \chi_q(a, \cdot) \) which is the order of a mod q.

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 \( \leq 2 \)).

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).
7.4.5 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.

```c
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*.

```c
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*.

```c
void dirichlet_chi_vec(ulong *v, const dirichlet_group_t G, const dirichlet_char_t chi, slong nv)
void dirichlet_chi_vec_order(ulong *v, const dirichlet_group_t G, const dirichlet_char_t chi, ulong order, slong nv)
```

Compute the list of exponent values $v[k]$ for $0 \leq k < nv$, as exponents modulo *G->expo*.

7.4.6 Character operations

```c
void dirichlet_char_mul(dirichlet_char_t chi12, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2)
void dirichlet_char_pow(dirichlet_char_t c, const dirichlet_group_t G, const dirichlet_char_t a, ulong n)
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)
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)
```

Multiply two characters of the same group *G*.

Take the power of a character.

If *H* is a subgroup of *G*, computes the character in *G* corresponding to *chi_H* in *H*.

If *chi_G* is a character of *G* which factors through *H*, sets *chi_H* to the corresponding restriction in *H*.

This requires $c(\chi_G) \mid q_H \mid q_G$, where $c(\chi_G)$ is the conductor of $\chi_G$ and $q_G, q_H$ are the moduli of *G* and *H*.

7.5 acb_dirichlet.h – Dirichlet L-functions, zeta functions, and related functions

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

This module allows working with values of Dirichlet characters, Dirichlet L-functions, and related functions. Working with Dirichlet characters is documented in [dirichlet.h – Dirichlet characters](#).

A Dirichlet L-function is the analytic continuation of an L-series

$$L(s, \chi) = \sum_{k=1}^{\infty} \frac{\chi(k)}{k^s}$$
where $\chi(k)$ is a Dirichlet character.

The code in other modules for computing the Riemann zeta function, Hurwitz zeta function and polylogarithm will possibly be migrated to this module in the future.

### 7.5.1 Roots of unity

**acb_dirichlet_roots_struct**

**acb_dirichlet_roots_t**

```c
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 $e^{2\pi i/k}$ 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.

```c
void acb_dirichlet_roots_clear(acb_dirichlet_roots_t roots)
```

Clears the structure.

```c
void acb_dirichlet_root(acb_t res, const acb_dirichlet_roots_t roots, ulong k, slong prec)
```

Computes $e^{2\pi ik/n}$.

### 7.5.2 Truncated L-series and power sums

```c
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 $k^{-1}(s+x)$ 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.

```c
void acb_dirichlet_powsum_sieved(acb_ptr res, const acb_t s, ulong n, slong len, slong prec)
```

Sets `res` to $\sum_{k=1}^{n} k^{-1}(s+x)$ 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.

```c
void acb_dirichlet_powsum_smooth(acb_ptr res, const acb_t s, ulong n, slong len, slong prec)
```

Sets `res` to $\sum_{k=1}^{n} k^{-1}(s+x)$ 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.
7.5.3 Riemann zeta function and 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

$$
\zeta(s) = R(s) + X(s)\overline{R}(1-s), \quad X(s) = \pi^{s-1/2} \frac{\Gamma((1-s)/2)}{\Gamma(s/2)}
$$

where

$$
R(s) = \sum_{k=1}^{N} \frac{1}{k^s} + (-1)^{N-1} U a^{-\sigma} \left[ \sum_{k=0}^{K} \frac{C_k(p)}{a^k} + RS_K \right]
$$

$$
U = \exp \left( -i \left[ \frac{t}{2} \log \left( \frac{t}{2\pi} \right) - \frac{t}{2} - \frac{\pi}{8} \right] \right), \quad a = \sqrt{\frac{t}{2\pi}}, \quad N = \lfloor a \rfloor, \quad p = 1 - 2(a - N).
$$

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.

```c
void acb_dirichlet_zeta_rs_f_coeffs(acb_ptr f, const arb_t p, slong n, slong prec);
```

Computes the coefficients $F^{(j)}(p)$ for $0 \leq j < n$. Uses power series division. This method breaks down when $p = \pm 1/2$ (which is not problem if $s$ is an exact floating-point number).

```c
void acb_dirichlet_zeta_rs_d_coeffs(acb_ptr d, const arb_t sigma, slong k, slong prec);
```

Computes the coefficients $d_j^{(k)}$ for $0 \leq j \leq [3k/2] + 1$. On input, the array $d$ must contain the coefficients for $d_j^{(k-1)}$ unless $k = 0$, and these coefficients will be updated in-place.

```c
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.

```c
void acb_dirichlet_zeta_rs_r(acb_t res, const acb_t s, slong K, slong prec);
```

Computes $R(s)$ in the upper half plane. Uses precisely $K$ asymptotic terms in the RS formula if this input parameter is positive; otherwise chooses the number of terms automatically based on $s$ and the precision.

```c
void acb_dirichlet_zeta(acb_t res, const acb_t s, 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.

```c
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.

```c
void acb_dirichlet_eta(acb_t res, const acb_t s, slong prec);
```

Sets res to the Dirichlet eta function $\eta(s) = \sum_{k=1}^{\infty} (-1)^k/k^s = (1 - 2^{1-s})\zeta(s)$, also known as the alternating zeta function. Note that the alternating character $\{1, -1\}$ is not itself a Dirichlet character.

7.5.4 Hurwitz zeta function

acb_dirichlet_hurwitz_precomp_struct
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 $O(1/(2AN)^K)$.

We require that $A$, $K$ and $N$ are all positive. Moreover, for a finite error bound, we require $K + \Re(s) > 1$. To avoid an initial “bump” that steals precision and slows convergence, $AN$ should be at least roughly as large as $|s|$, e.g. it is a good idea to have at least $AN > 0.5|s|$.

If `deflate` is set, the deflated Hurwitz zeta function is used, removing the pole at $s = 1$.

void acb_dirichlet_hurwitz_precomp_clear(acb_dirichlet_hurwitz_precomp_t pre)

Clears the precomputed data.

void acb_dirichlet_hurwitz_precomp_choose_param(ulong * A, ulong * K, ulong * N, const acb_t s, double num_eval, slong prec)

Chooses precomputation parameters $A$, $K$ and $N$ to minimize the cost of `num_eval` evaluations of the Hurwitz zeta function at argument $s$ to precision `prec`. If it is estimated that evaluating each Hurwitz zeta function from scratch would be better than performing a precomputation, $A$, $K$ and $N$ are all set to 0.

void acb_dirichlet_hurwitz_precomp_bound(mag_t res, const acb_t s, ulong A, ulong K, ulong N)

Computes an upper bound for the truncation error (not accounting for roundoff error) when evaluating $\zeta(s, a)$ with precomputation parameters $A$, $K$, $N$, assuming that $0 < a \leq 1$. For details, see Algorithms for the Hurwitz zeta function.

void acb_dirichlet_hurwitz_precomp_eval(acb_t res, const acb_dirichlet_hurwitz_precomp_t pre, ulong p, ulong q, slong prec)

Evaluates $\zeta(s, p/q)$ using precomputed data, assuming that $0 < p/q \leq 1$.

### 7.5.5 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 res, 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.

### 7.5.6 Gauss and Jacobi 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)

---

7.5. acb_dirichlet.h – Dirichlet L-functions, zeta functions, and related functions 147
void **acb_dirichlet_gauss_sum** (acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)

void **acb_dirichlet_gauss_sum_ui** (acb_t res, const dirichlet_group_t G, ulong a, slong prec)

Sets res to the Gauss sum

\[ G_q(a) = \sum_{x \mod q} \chi_q(a, x)e^{\frac{2\pi i x}{q}} \]

- the **naive** version computes the sum as defined.
- the **factor** version writes it as a product of local Gauss sums by chinese remainder theorem.
- the **order2** version assumes chi is real and primitive and returns \( i^{\frac{p}{2}} \sqrt{q} \) where \( p \) is the parity of \( \chi \).
- the **theta** version assumes that \( \chi \) is primitive to obtain the Gauss sum by functional equation of the theta series at \( t = 1 \). An abort will be raised if the theta series vanishes at \( t = 1 \). Only 4 exceptional characters of conductor 300 and 600 are known to have this particularity, and none with primepower modulus.
- the default version automatically combines the above methods.
- the **ui** version only takes the Conrey number \( a \) as parameter.

void **acb_dirichlet_jacobi_sum_naive** (acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2, slong prec)

void **acb_dirichlet_jacobi_sum_factor** (acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2, slong prec)

void **acb_dirichlet_jacobi_sum_gauss** (acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2, slong prec)

void **acb_dirichlet_jacobi_sum** (acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2, slong prec)

void **acb_dirichlet_jacobi_sum_ui** (acb_t res, const dirichlet_group_t G, ulong a, ulong b, slong prec)

Computes the Jacobi sum

\[ J_q(a, b) = \sum_{x \mod q} \chi_q(a, x)\chi_q(b, 1 - x) \]

- the **naive** version computes the sum as defined.
- the **factor** version writes it as a product of local Jacobi sums
- the **gauss** version assumes \( ab \) is primitive and uses the formula \( J_q(a, b)G_q(ab) = G_q(a)G_q(b) \)
- the default version automatically combines the above methods.
- the **ui** version only takes the Conrey numbers \( a \) and \( b \) as parameters.

### 7.5.7 Theta sums

We call **theta series** of a Dirichlet character the quadratic series

\[ \Theta_q(a) = \sum_{n \geq 0} \chi_q(a, n)n^p x^n \]

where \( p \) is the parity of the character \( \chi_q(a, \cdot) \).
For $\Re(t) > 0$ we write $x(t) = \exp(-\frac{\pi}{t} t^2)$ and define

$$\Theta_q(a,t) = \sum_{n \geq 0} \chi_q(a,n)x(t)n^2.$$ 

void `acb_dirichlet_ch_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 $\theta(a,t) = \epsilon(\chi) \theta\left(\frac{1}{a}, \frac{1}{t}\right)$ should be used, which is not done automatically (to avoid recomputing the Gauss sum).

### 7.5.8 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 $\chi$):

$$\left(\frac{q}{\pi}\right)^{\frac{s}{2}} \Gamma\left(s + \frac{p}{2}\right) L(s, \chi) = \epsilon(\chi) \left(\frac{q}{\pi}\right)^{\frac{1-s+p}{2}} \Gamma\left(1 - \frac{s + p}{2}\right) L(1 - s, \chi)$$

• 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 $\chi$ 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 $\text{re}(s) \geq 2$.

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

$$\prod_{p<N} \left(1 - \frac{\chi(p)}{p^s}\right) - \frac{1}{L(s, \chi)} \leq \sum_{k=N}^{\infty} \frac{1}{k^s} = \zeta(s, N).$$

The underscore version specialized for integer $s$ assumes that $\chi$ is a real Dirichlet character given by the explicit list $\chi$ of character values at 0, 1, ..., $\mod -1$. If $\text{reciprocal}$ is set, it computes $1/L(s, \chi)$ (this is faster if the reciprocal can be used directly).

```c
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.

```c
void acb_dirichlet_l_jet(acb_ptr res, const acb_t t, 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 $\text{len}$, i.e. $L(s), L'(s), \ldots, L^{(\text{len}-1)}(s)/(\text{len}-1)!$. If $\text{deflate}$ is set, computes the expansion of

$$L(s, \chi) - \sum_{k=1}^{\text{len}} \frac{\chi(k)}{(s-1)^k}$$

instead. If $\chi$ is a principal character, then this has the effect of subtracting the pole with residue $\sum_{k=1}^{\text{len}} \chi(k) = \phi(q)/q$ 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, $\text{deflate}$ has no effect.

```c
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)
```

Sets $\text{res}$ to the power series $L(s, \chi)$ where $s$ is a given power series, truncating the result to length $\text{len}$. See $\text{acb_dirichlet_l_jet()}$ for the meaning of the $\text{deflate}$ flag.

### 7.5.9 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.

```c
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

$$\theta(t) = -\frac{t}{2} \log(\pi/q) - \frac{i \log(\epsilon)}{2} + \log \Gamma((s + \delta)/2) - \log \Gamma((1 - s + \delta)/2)$$

where $s = 1/2 + it$, $\delta$ is the parity of $\chi$, and $\epsilon$ is the root number as computed by $\text{acb_dirichlet_root_number()}$. The first $\text{len}$ terms in the Taylor expansion are written to the output.

```c
void acb_dirichlet_hardy_z(acb_t res, const acb_t t, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)
```

Computes the Hardy Z-function, also known as the Riemann-Siegel Z-function $Z(t) = e^{\theta(t)}L(1/2 + it)$, which is real-valued for real $t$. The first $\text{len}$ terms in the Taylor expansion are written to the output.

```c
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)
```

```c
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 $\text{res}$ to the power series $\theta(t)$ where $t$ is a given power series, truncating the result to length $\text{len}$. 

---

**Chapter 7. Higher mathematical functions**
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.

### 7.6 bernoulli.h – support for Bernoulli numbers

This module provides helper functions for exact or approximate calculation of the Bernoulli numbers, which are defined by the exponential generating function

$$
\frac{x}{e^x - 1} = \sum_{n=0}^{\infty} B_n \frac{x^n}{n!}.
$$

Efficient algorithms are implemented for both multi-evaluation and calculation of isolated Bernoulli numbers. A global (or thread-local) cache is also provided, to support fast repeated evaluation of various special functions that depend on the Bernoulli numbers (including the gamma function and the Riemann zeta function).

#### 7.6.1 Generation of Bernoulli numbers

bernoulli_rev_t

An iterator object for generating a range of even-indexed Bernoulli numbers exactly in reverse order, i.e. computing the exact fractions $B_n, B_{n-2}, B_{n-4}, \ldots, B_0$. The Bernoulli numbers are generated from scratch, i.e. no caching is performed.

The Bernoulli numbers are computed by direct summation of the zeta series. This is made fast by storing a table of powers (as done by [Blo2009]). As an optimization, we only include the odd powers, and use fixed-point arithmetic.

The reverse iteration order is preferred for performance reasons, as the powers can be updated using multiplications instead of divisions, and we avoid having to periodically recompute terms to higher precision. To generate Bernoulli numbers in the forward direction without having to store all of them, one can split the desired range into smaller blocks and compute each block with a single reverse pass.

void bernoulli_rev_init(bernoulli_rev_t iter, ulong n)

Initializes the iterator iter. The first Bernoulli number to be generated by calling bernoulli_rev_next() is $B_n$. It is assumed that $n$ is even.

void bernoulli_rev_next(fmpz_t numer, fmpz_t denom, bernoulli_rev_t iter)

Sets numer and denom to the exact, reduced numerator and denominator of the Bernoulli number $B_k$ and advances the state of iter so that the next invocation generates $B_{k-2}$.

void bernoulli_rev_clear(bernoulli_rev_t iter)

Frees all memory allocated internally by iter.

#### 7.6.2 Caching

slong bernoulli_cache_num

fmpz * bernoulli_cache

Cache of Bernoulli numbers. Uses thread-local storage if enabled in FLINT.
void bernoulli_cache_compute(slong n)
    Makes sure that the Bernoulli numbers up to at least \( B_{n-1} \) are cached. Calling flint_cleanup() frees the cache.

7.6.3 Bounding

slong bernoulli_bound_2exp_si(ulong n)
    Returns an integer \( b \) such that \( |B_n| \leq 2^b \). Uses a lookup table for small \( n \), and for larger \( n \) uses the inequality \( |B_n| < 4n!/(2\pi)^n < 4(n+1)^{n+1}e^{-n}/(2\pi)^n \). Uses integer arithmetic throughout, with the bound for the logarithm being looked up from a table. If \( |B_n| = 0 \), returns LONG_MIN. Otherwise, the returned exponent \( b \) is never more than one percent larger than the true magnitude.

This function is intended for use when \( n \) small enough that one might comfortably compute \( B_n \) exactly. It aborts if \( n \) is so large that internal overflow occurs.

void _bernoulli_fmpq_ui_zeta(fmpz_t num, fmpz_t den, ulong n)
    Sets \( num \) and \( den \) to the reduced numerator and denominator of the Bernoulli number \( B_n \). This function computes the denominator \( d \) using von Staudt-Clausen theorem, numerically approximates \( B_n \) using arb_bernoulli_ui_zeta(), and then rounds \( dB_n \) to the correct numerator. If the working precision is insufficient to determine the numerator, the function prints a warning message and retries with increased precision (this should not be expected to happen).

void _bernoulli_fmpq_ui(fmpz_t num, fmpz_t den, ulong n)
void bernoulli_fmpq_ui(fmpq_t b, ulong n)
    Computes the Bernoulli number \( B_n \) as an exact fraction, for an isolated integer \( n \). This function reads \( B_n \) from the global cache if the number is already cached, but does not automatically extend the cache by itself.

7.7 hypgeom.h – support for hypergeometric series

This module provides functions for high-precision evaluation of series of the form

\[
\sum_{k=0}^{n-1} \frac{A(k)}{B(k)} \prod_{j=1}^{k} \frac{P(j)}{Q(j)} z^k
\]

where \( A, B, P, Q \) are polynomials. The present version only supports \( A, B, P, Q \in \mathbb{Z}[k] \) (represented using the FLINT fmpz_poly_t type). This module also provides functions for high-precision evaluation of infinite series \( (n \to \infty) \), with automatic, rigorous error bounding.

Note that we can standardize to \( A = B = 1 \) by setting \( \tilde{P}(k) = P(k)A(k)B(k-1) \), \( \tilde{Q}(k) = Q(k)A(k-1)B(k) \). However, separating out \( A \) and \( B \) is convenient and improves efficiency during evaluation.

7.7.1 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

$$zR(k) = \frac{P(k)}{Q(k)} = z \left( \frac{a_p}{b_q} \right) \frac{1}{k^{q-p}} F(k)$$

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).

### 7.7.2 Types, macros and constants

**hypgeom_struct**

**hypgeom_t**

Stores polynomials $A$, $B$, $P$, $Q$ and precomputed bounds, representing a fixed hypergeometric series.

### 7.7.3 Memory management

```c
void hypgeom_init(hypgeom_t hyp)
void hypgeom_clear(hypgeom_t hyp)
```
7.7.4 Error bounding

```c
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.

```c
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 $\text{prec}$ 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

$$\frac{k(k-D)}{K^r(k-C)(k-2D)}$$

Given this information, we compute a $\varepsilon$ and an integer $n$ such that $|\sum_{k=n}^\infty T(k)| \leq \varepsilon \leq 2^{-\text{prec}}$. The output variable `error` is set to the value of $\varepsilon$, and $n$ is returned.

```c
void hypgeom_precompute(hypgeom_t hyp)
```
Precomputes the bounds data $C$, $D$, $K$ and an upper bound for $T(K)$.

7.7.5 Summation

```c
void arb_hypgeom_sum(arb_t P, arb_t Q, const hypgeom_t hyp, const slong n, slong prec)
```
Computes $P/Q = \sum_{k=0}^{n-1} T(k)$ where $T(k)$ is defined by `hyp`, using binary splitting and a working precision of $\text{prec}$ bits.

```c
void arb_hypgeom_infsum(arb_t P, arb_t Q, hypgeom_t hyp, slong tol, slong prec)
```
Computes $P/Q = \sum_{k=0}^{\infty} T(k)$ where $T(k)$ is defined by `hyp`, using binary splitting and working precision of $\text{prec}$ bits. The number of terms is chosen automatically to bound the truncation error by at most $2^{-\text{tol}}$. The bound for the truncation error is included in the output as part of $P$.

7.8 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. $n < 10^6$).

```c
void partitions_rademacher_bound(arf_t b, const fmpz_t n, ulong N)
```
Sets $b$ to an upper bound for

$$M(n,N) = \frac{44\pi^2}{225\sqrt{3}} N^{-1/2} + \frac{\pi\sqrt{2}}{75} \left(\frac{N}{n-1}\right)^{1/2} \sinh \left(\frac{\pi}{N} \sqrt{\frac{2n}{3}}\right).$$

This formula gives an upper bound for the truncation error in the Hardy-Ramanujan-Rademacher formula when the series is taken up to the term $t(n,N)$ inclusive.
partitions_hrr_sum_arb\((\text{arb}_t \ x, \text{const } \text{fmpz}_t n, \text{slong } N_0, \text{slong } N, \text{int use_doubles})\)

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\((\text{fmpz}_t p, \text{const } \text{fmpz}_t n, \text{int use_doubles})\)

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\((\text{fmpz}_t p, \text{ulong } n)\)

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\((\text{fmpz}_t p, \text{ulong } n)\)

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\((\text{arb}_t \text{res }, \text{const } \text{fmpz}_t n, \text{slong prec})\)

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 $t = \pi \sqrt{24n-1}/6$. 

7.8. partitions.h – computation of the partition function 155
Using ball arithmetic, it is possible to do rigorous root-finding and integration (among other operations) with generic functions. This code should be considered experimental.

8.1 arb_calc.h – calculus with real-valued functions

This module provides functions for operations of calculus over the real numbers (intended to include root-finding, optimization, integration, and so on). It is planned that the module will include two types of algorithms:

- Interval algorithms that give provably correct results. An example would be numerical integration on an interval by dividing the interval into small balls and evaluating the function on each ball, giving rigorous upper and lower bounds.
- Conventional numerical algorithms that use heuristics to estimate the accuracy of a result, without guaranteeing that it is correct. An example would be numerical integration based on pointwise evaluation, where the error is estimated by comparing the results with two different sets of evaluation points. Ball arithmetic then still tracks the accuracy of the function evaluations.

Any algorithms of the second kind will be clearly marked as such.

8.1.1 Types, macros and constants

arb_calc_func_t

Typedef for a pointer to a function with signature:

```
int func(arb_ptr out, const arb_t inp, void * param, slong order, slong prec)
```

implementing a univariate real function \( f(x) \). When called, \( func \) should write to \( out \) the first \( order \) coefficients in the Taylor series expansion of \( f(x) \) at the point \( inp \), evaluated at a precision of \( prec \) bits. The \( param \) argument may be used to pass through additional parameters to the function. The return value is reserved for future use as an error code. It can be assumed that \( out \) and \( inp \) are not aliased and that \( order \) is positive.

ARB_CALC_SUCCESS

Return value indicating that an operation is successful.

ARB_CALC_IMPRECISE_INPUT

Return value indicating that the input to a function probably needs to be computed more accurately.

ARB_CALC_NO_CONVERGENCE

Return value indicating that an algorithm has failed to convergence, possibly due to the problem not having a solution, the algorithm not being applicable, or the precision being insufficient.
8.1.2 Debugging

```c
int arb_calc_verbose
```
If set, enables printing information about the calculation to standard output.

8.1.3 Subdivision-based root finding

```c
arf_interval_struct
arf_interval_t
```
An `arf_interval_struct` consists of a pair of `arf_struct`, representing an interval used for subdivision-based root-finding. An `arf_interval_t` is defined as an array of length one of type `arf_interval_struct`, permitting an `arf_interval_t` to be passed by reference.

```c
arf_interval_ptr
```
Alias for `arf_interval_struct *`, used for vectors of intervals.

```c
arf_interval_srcptr
```
Alias for `const arf_interval_struct *`, used for vectors of intervals.

```c
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.

```c
void arf_interval_fprintd(FILE * file, const arf_interval_t v, slong n)
```
Helper functions for endpoint-based intervals.

```c
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.
8.1.4 Newton-based root finding

void arb_calc_newton_conv_factor (arb_t conv_factor, arb_calc_func_t func, void * param, const arb_t conv_region, slong prec)

Given an interval \( I \) specified by \( \text{conv} \_\text{region} \), evaluates a bound for \( C = \sup_{t,u \in I} \frac{1}{2} |f''(t)|/|f'(u)| \), where \( f \) is the function specified by \( \text{func} \) and \( \text{param} \). 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_refine_root_newton (arb_t r, arb_calc_func_t func, void * param, const arb_t start, const arb_t conv_region, const arb_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 \( \text{arb} \_\text{calc} \_\text{newton} \_\text{step} \), except for the precision parameters: \( \text{prec} \) is the target accuracy and \( \text{eval} \_\text{extra} \_\text{prec} \) 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 \( \text{ARB} \_\text{CALC} \_\text{SUCCESS} \) if all attempted Newton steps are successful (note that this does not guarantee that the computed root is accurate to \( \text{prec} \) bits, which has to be verified by the user), only that it is more accurate than the starting ball.
On failure, `ARB_CALC_LMPRECISE_INPUT` or `ARB_CALC_NO_CONVERGENCE` may be returned. In this case, \( r \) is set to a ball for the root which is valid but likely does have full accuracy (it can possibly just be equal to the starting ball).

8.2 `acb_calc.h` – calculus with complex-valued functions

This module provides functions for operations of calculus over the complex numbers (intended to include root-finding, integration, and so on).

8.2.1 Types, macros and constants

`acb_calc_func_t`

TYPEDef for a pointer to a function with signature:

\[
\text{int func(acb_ptr out, const acb_t inp, void * param, slong order, slong prec)}
\]

implementing a univariate complex function \( f(x) \). When called, \( \text{func} \) should write to \( \text{out} \) the first \( \text{order} \) coefficients in the Taylor series expansion of \( f(x) \) at the point \( \text{inp} \), evaluated at a precision of \( \text{prec} \) bits. The \( \text{param} \) 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 \( \text{out} \) and \( \text{inp} \) are not aliased and that \( \text{order} \) is positive.

8.2.2 Bounds

\[
\text{void acb_calc_cauchy bound (arb_t bound, acb_calc_func_t func, void * param, const acb_t x, const arb_t radius, slong maxdepth, slong prec)}
\]

Sets \( \text{bound} \) 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 \( (\text{func, param}) \) and \( r \) is given by \( \text{radius} \). 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 \( \text{maxdepth} \) times. This function is currently implemented completely naively, and repeatedly subdivides the whole integration range instead of performing adaptive subdivisions.

8.2.3 Integration

\[
\text{int acb_calc_integrate_taylor (acb_t res, acb_calc_func_t func, void * param, const acb_t a, const acb_t b, const arf_t inner_radius, const arf_t outer_radius, slong accuracy_goal, slong prec)}
\]

Computes the integral

\[
I = \int_a^b f(t) dt
\]

where \( f \) is specified by \( (\text{func, param}) \), 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

$$\sum_{n=N}^{\infty} \frac{C(m, R) |x|^{n+1}}{R^n (N+1)} = \frac{C(m, R)Rx}{(R-x)(N+1)} \left( \frac{x}{R} \right)^N.$$  

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.
Mainly for internal use.

9.1 fmpz_extras.h – extra methods for FLINT integers

This module implements a few utility methods for the FLINT multiprecision integer type (fmpz_t). It is mainly intended for internal use.

9.1.1 Memory-related methods

\textit{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 \texttt{sizeof(fmpz)} to get the size of the object as a whole.

9.1.2 Convenience methods

\texttt{void fmpz_add_si(fmpz_t z, const fmpz_t x, slong y)}

Sets \( z \) to the sum (respectively difference) of \( x \) and \( y \).

\texttt{void fmpz_sub_si(fmpz_t z, const fmpz_t x, slong y)}

Sets \( z \) to \( x - y \), rounded away from zero.

\texttt{void fmpz_adiv_q_2exp(fmpz_t z, const fmpz_t x, mp_bitcnt_t exp)}

Sets \( z \) to \( x/2^\text{exp} \), rounded away from zero.

\texttt{void fmpz_ui_pow_ui(fmpz_t x, ulong b, ulong e)}

Sets \( x \) to \( b \) raised to the power \( e \).

\texttt{void fmpz_max(fmpz_t z, const fmpz_t x, const fmpz_t y)}

Sets \( z \) to the maximum (respectively minimum) of \( x \) and \( y \).

9.1.3 Inlined arithmetic

The \texttt{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 \textit{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.

\texttt{void fmpz_add_inline(fmpz_t z, const fmpz_t x, const fmpz_t y)}

\texttt{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 -\texttt{WORD\_MAX} and \texttt{WORD\_MAX}, i.e. between \pm(2^{63}−1) inclusive on a 64-bit machine.

### 9.1.4 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 \texttt{fmpz\_set\_ui}, \texttt{fmpz\_neg\_ui} can be used for single-limb integers.

void FMPZ\_GET\_MPN\_READONLY(zsign, zn, zptr, ztmp, zv)
Given an fmpz_t zv, this macro sets zptr to a pointer to the limbs of zv, zn to the number of limbs, and zsign to a sign bit (0 if nonnegative, 1 if negative). The variable ztmp must be a single mp_limb_t, which is used as a buffer. If zv is a small value, zv itself contains no limb array that zptr could point to, so the single limb is copied to ztmp and zptr is set to point to ztmp. The case where zv is zero is not handled specially, and zn is set to 1.

void fmpz_lshift_mpn(fmpz_t z, mp_srcptr src, mp_size_t n, int negative, mp_bitcnt_t shift)
Sets z to the integer represented by the n limbs in the array src, or minus this value if negative is 1, shifted left by shift bits. Requires \(n \geq 1\) and that the top limb of src is nonzero.

### 9.2 bool_mat.h – matrices over booleans

A bool_mat_t represents a dense matrix over the boolean semiring \(\langle\{0,1\}, \lor, \land\rangle\), implemented as an array of entries of type int.

The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

#### 9.2.1 Types, macros and constants

bool_mat_struct

bool_mat_t
Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An bool_mat_t is defined as an array of length one of type bool_mat_struct, permitting an bool_mat_t to be passed by reference.

int bool_mat_get_entry(const bool_mat_t mat, slong i, slong j)
Returns the entry of matrix mat at row i and column j.

void bool_mat_set_entry(bool_mat_t mat, slong i, slong j, int x)
Sets the entry of matrix mat at row i and column j to x.
bool_mat_nrows(mat)
Returns the number of rows of the matrix.

bool_mat_ncols(mat)
Returns the number of columns of the matrix.

9.2.2 Memory management

void bool_mat_init(bool_mat_t mat, long r, long c)
Initializes the matrix, setting it to the zero matrix with \( r \) rows and \( c \) columns.

void bool_mat_clear(bool_mat_t mat)
Clears the matrix, deallocating all entries.

int bool_mat_is_empty(const bool_mat_t mat)
Returns nonzero iff the number of rows or the number of columns in \( mat \) is zero. Note that this does not depend on the entry values of \( mat \).

int bool_mat_is_square(const bool_mat_t mat)
Returns nonzero iff the number of rows is equal to the number of columns in \( mat \).

9.2.3 Conversions

void bool_mat_set(bool_mat_t dest, const bool_mat_t src)
Sets \( dest \) to \( src \). The operands must have identical dimensions.

9.2.4 Input and output

void bool_mat_print(const bool_mat_t mat)
Prints each entry in the matrix.

void bool_mat_fprint(FILE * file, const bool_mat_t mat)
Prints each entry in the matrix to the stream \( file \).

9.2.5 Value comparisons

int bool_mat_equal(const bool_mat_t mat1, const bool_mat_t mat2)
Returns nonzero iff the matrices have the same dimensions and identical entries.

int bool_mat_any(const bool_mat_t mat)
Returns nonzero iff \( mat \) has a nonzero entry.

int bool_mat_all(const bool_mat_t mat)
Returns nonzero iff all entries of \( mat \) are nonzero.

int bool_mat_is_diagonal(const bool_mat_t A)
Returns nonzero iff \( i \neq j \implies A_{ij} \).

int bool_mat_is_lower_triangular(const bool_mat_t A)
Returns nonzero iff \( i < j \implies A_{ij} \).

int bool_mat_is_transitive(const bool_mat_t mat)
Returns nonzero iff \( A_{ij} \land A_{jk} \implies A_{ik} \).

int bool_mat_is_nilpotent(const bool_mat_t A)
Returns nonzero iff some positive matrix power of \( A \) is zero.
9.2.6 Random generation

void bool_mat_randtest(bool_mat_t mat, flint_rand_t state)
Sets mat to a random matrix.

void bool_mat_randtest_diagonal(bool_mat_t mat, flint_rand_t state)
Sets mat to a random diagonal matrix.

void bool_mat_randtest_nilpotent(bool_mat_t mat, flint_rand_t state)
Sets mat to a random nilpotent matrix.

9.2.7 Special matrices

void bool_mat_zero(bool_mat_t mat)
Sets all entries in mat to zero.

void bool_mat_one(bool_mat_t mat)
Sets the entries on the main diagonal to ones, and all other entries to zero.

void bool_mat_directed_path(bool_mat_t A)
Sets $A_{ij}$ to $j = i + 1$. Requires that $A$ is a square matrix.

void bool_mat_directed_cycle(bool_mat_t A)
Sets $A_{ij}$ to $j = (i + 1) \mod n$ where $n$ is the order of the square matrix $A$.

9.2.8 Transpose

void bool_mat_transpose(bool_mat_t dest, const bool_mat_t src)
Sets dest to the transpose of src. The operands must have compatible dimensions. Aliasing is allowed.

9.2.9 Arithmetic

void bool_mat_complement(bool_mat_t B, const bool_mat_t A)
Sets B to the logical complement of A. That is $B_{ij}$ is set to $\overline{A_{ij}}$. The operands must have the same dimensions.

void bool_mat_add(bool_mat_t res, const bool_mat_t mat1, const bool_mat_t mat2)
Sets res to the sum of mat1 and mat2. The operands must have the same dimensions.

void bool_mat_mul(bool_mat_t res, const bool_mat_t mat1, const bool_mat_t mat2)
Sets res to the matrix product of mat1 and mat2. The operands must have compatible dimensions for matrix multiplication.

void bool_mat_mul_entrywise(bool_mat_t res, const bool_mat_t mat1, const bool_mat_t mat2)
Sets res to the entrywise product of mat1 and mat2. The operands must have the same dimensions.

void bool_mat_sqr(bool_mat_t B, const bool_mat_t A)
Sets B to the matrix square of A. The operands must both be square with the same dimensions.

void bool_mat_pow_ui(bool_mat_t B, const bool_mat_t A, ulong exp)
Sets B to $A$ raised to the power exp. Requires that $A$ is a square matrix.

9.2.10 Special functions

int bool_mat_trace(const bool_mat_t mat)
Returns the trace of the matrix, i.e. the sum of entries on the main diagonal of mat. The matrix is required to be square. The sum is in the boolean semiring, so this function returns nonzero iff any entry on the diagonal of mat is nonzero.
### 9.3 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.

#### 9.3.1 Types, macros and constants

**DLOG_NONE**
- Return value when the discrete logarithm does not exist

**dlog_precomp_struct**

**dlog_precomp_t**
- Structure for discrete logarithm precomputed data.

A `dlog_precomp_t` is defined as an array of length one of type `dlog_precomp_struct`, permitting a `dlog_precomp_t` to be passed by reference.
9.3.2 Single evaluation

ulong dlog_once(ulong b, ulong a, const nmod_t mod, ulong n)

Return $x$ such that $b = a^x \pmod{\mathbb{Z}/mod\mathbb{Z}}$, where $a$ is known to have order $n$.

9.3.3 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 $pe$.

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.

9.3.4 Vector evaluations

These functions compute all logarithms of successive integers $1 \ldots n$.

void dlog_vec_fill(ulong * v, ulong nv, ulong x)

Sets values $v[k]$ to $x$ for all $k$ less than $nv$.

void dlog_vec_set_not_found(ulong * v, ulong nv, nmod_t mod)

Sets values $v[k]$ to $DLOG_NONE$ for all $k$ not coprime to $mod$.

void dlog_vec(ulong * v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)

Sets $v[k]$ to $\log(k, a) \times va$ for $0 \leq k < nv$, where $a$ has order $na$. $va$ should be 1 for usual log computation.

void dlog_vec_add(ulong * v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)

Same parameters as before, but adds $\log(k, a) \times va$ to $v[k]$ and reduce modulo $order$ instead of replacing the value. Indices $k$ such that $v[k] = DLOG_NONE$ 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.

9.3.5 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**

*\textbf{dlog\_table}*

*\textbf{dlog\_table\_t}*

Structure for complete lookup table.

*\textbf{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 \( mod \).

*\textbf{void dlog\_table\_clear(dlog\_table\_t t)}*

Clears \( t \).

*\textbf{ulong dlog\_table(dlog\_table\_t t, ulong b)}*

Return \( \log(b, a) \) using the precomputed data \( t \).

**Baby-step giant-step table**

*\textbf{dlog\_bsgs\_struct}*

*\textbf{dlog\_bsgs\_t}*

Structure for Baby-Step Giant-Step decomposition.

*\textbf{ulong dlog\_bsgs\_init(dlog\_bsgs\_t t, ulong a, ulong mod, ulong n, ulong m)}*

Initialize \( t \) and store the first \( m \) powers of \( a \) in a sorted array. The return value is a rough measure of the cost of each logarithm using this table. The user should take \( m \approx \sqrt{k n} \) to compute \( k \) logarithms in a group of size \( n \).

*\textbf{void dlog\_bsgs\_clear(dlog\_bsgs\_t t)}*

Clears \( t \).```
ulong dlog_bsgs(dlog_bsgs_t t, ulong b)
    Return log\(b, a\) using the precomputed data \(t\).

Prime-power modulus decomposition

dlog_modpe_struct

dlog_modpe_t
    Structure for discrete logarithm modulo primepower \(p^e\).

    A dlog_modpe_t is defined as an array of length one of type dlog_modpe_struct, permitting a
dlog_modpe_t to be passed by reference.

ulong dlog_modpe_init(dlog_modpe_t t, ulong a, ulong p, ulong e, ulong pe, ulong num)
    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_modpe_clear(dlog_modpe_t t)
    Clears \(t\).

ulong dlog_modpe(dlog_modpe_t t, ulong b)
    Return log\(b, a\) using the precomputed data \(t\).

CRT decomposition

dlog_crt_struct

dlog_crt_t
    Structure for discrete logarithm for groups of composite order. A dlog_crt_t is defined as an
    array of length one of type dlog_crt_struct, permitting a dlog_crt_t to be passed by reference.

ulong dlog_crt_init(dlog_crt_t t, ulong a, ulong mod, ulong n, ulong num)
    Precompute data for \(num\) evaluations of discrete logarithms in base \(a\) modulo \(mod\), where \(a\) has
    composite order \(n\), using chinese remainder decomposition.

void dlog_crt_clear(dlog_crt_t t)
    Clears \(t\).

ulong dlog_crt(dlog_crt_t t, ulong b)
    Return log\(b, a\) using the precomputed data \(t\).

padic decomposition

dlog_power_struct

dlog_power_t
    Structure for discrete logarithm for groups of primepower order. A dlog_power_t is defined as an
    array of length one of type dlog_power_struct, permitting a dlog_power_t to be passed by reference.

ulong dlog_power_init(dlog_power_t t, ulong a, ulong mod, ulong p, ulong e, ulong num)
    Precompute data for \(num\) evaluations of discrete logarithms in base \(a\) modulo \(mod\), where \(a\) has
    prime power order \(pe\) equals \(p^e\), using decomposition in base \(p\).

void dlog_power_clear(dlog_power_t t)
    Clears \(t\).

ulong dlog_power(dlog_power_t t, ulong b)
    Return log\(b, a\) using the precomputed data \(t\).
Pollard rho method

dlog_rho Struct

dlog_rho_t

Structure for discrete logarithm using Pollard rho. A \texttt{dlog_rho_t} is defined as an array of length one of type \texttt{dlog_rho_struct}, permitting a \texttt{dlog_rho_t} to be passed by reference.

\texttt{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 $n$.

\texttt{void dlog_rho_clear(dlog_rho_t t)}

Clears $t$.

\texttt{ulong dlog_rho(dlog_rho_t t, ulong b)}

Return $\log(b, a)$ by the rho method in the group described by $t$.

9.4 \texttt{fmpfr.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 \texttt{arf_t} type that was introduced in Arb 2.0. Please use \texttt{arf_t} instead of the \texttt{fmpfr_t} type.

A variable of type \texttt{fmpfr_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 of these might be added in the future.

An \texttt{fmpfr} number is exact and has no inherent “accuracy”. We use the term \textit{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 $\sqrt{2} = 1.01101010100\ldots_2$).

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

9.4.1 Types, macros and constants

\texttt{fmpfr_struct}

An \texttt{fmpfr_struct} holds a mantissa and an exponent. If the mantissa and exponent are sufficiently small, their values are stored as immediate values in the \texttt{fmpfr_struct}; large values are represented by pointers to heap-allocated arbitrary-precision integers. Currently, both the mantissa and exponent are implemented using the FLINT \texttt{fmpz} type. Special values are currently encoded by the mantissa being set to zero.

\texttt{fmpfr_t}

An \texttt{fmpfr_t} is defined as an array of length one of type \texttt{fmpfr_struct}, permitting an \texttt{fmpfr_t} to be passed by reference.
FMPR_rnd_t
   Specifies the rounding mode for the result of an approximate operation.

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

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

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

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

FMPR_RND_NEAR
   Specifies that the result of an operation should be rounded to the nearest representable number,
   rounding to an odd mantissa if there is a tie between two values. Warning: this rounding mode is
   currently not implemented (except for a few conversions functions where this stated explicitly).

FMPR_PREC_EXACT
   If passed as the precision parameter to a function, indicates that no rounding is to be performed.
   This must only be used when it is known that the result of the operation can be represented exactly
   and fits in memory (the typical use case is working small integer values). Note that, for example,
   adding two numbers whose exponents are far apart can easily produce an exact result that is far
   too large to store in memory.

9.4.2 Memory management

void fmpfr_init(fmpr_t x)
   Initializes the variable x for use. Its value is set to zero.

void fmpfr_clear(fmpr_t x)
   Clears the variable x, freeing or recycling its allocated memory.

9.4.3 Special values

void fmpfr_zero(fmpr_t x)
void fmpfr_one(fmpr_t x)
void fmpfr_pos_inf(fmpr_t x)
void fmpfr_neg_inf(fmpr_t x)
void fmpfr_nan(fmpr_t x)
   Sets x respectively to 0, 1, +\infty, -\infty, NaN.

int fmpfr_is_zero(const fmpr_t x)
int fmpfr_is_one(const fmpr_t x)
int fmpfr_is_pos_inf(const fmpr_t x)
int fmpfr_is_neg_inf(const fmpr_t x)
int fmpfr_is_nan(const fmpr_t x)
   Returns nonzero iff x respectively equals 0, 1, +\infty, -\infty, NaN.

int fmpfr_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, \(+\infty\), \(-\infty\), \(\text{NaN}\).

int fmpr_is_special(const fmpr_t x)
Returns nonzero iff \( x \) is one of the special values 0, \(+\infty\), \(-\infty\), \(\text{NaN}\), 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 \(+\infty\), \(-\infty\), \(\text{NaN}\).
(Note that this is not equivalent to the negation of \( \text{fmpr_is_infin()} \).)

### 9.4.4 Assignment, rounding and conversions

**slong fmpr_normalise(fmpr_t man, fmpz_t exp, slong prec, fmpr_rnd_t rnd)**

Rounds the mantissa and exponent in-place.

**void fmpr_set(fmpr_t y, const fmpr_t x)**
Sets \( y \) to a copy of \( x \).

**void fmpr_swap(fmpr_t x, fmpr_t y)**
Swaps \( x \) and \( y \) efficiently.

**slong fmpr_set_round(fmpr_t y, const fmpr_t x, slong prec, fmpr_rnd_t rnd)**
Sets \( y \) to a copy of \( x \) rounded in the direction specified by \( \text{rnd} \) to the number of bits specified by \( \text{prec} \).

**slong fmpr_set_round_mpn(slong * shift, fmpz_t man, mp_srcptr x, mp_size_t xn, int negative, slong prec, fmpr_rnd_t rnd)**

Given an integer represented by a pointer \( x \) to a raw array of \( xn \) limbs (negated if \( \text{negative} \) is nonzero), sets \( \text{man} \) to the corresponding floating-point mantissa rounded to \( \text{prec} \) bits in direction \( \text{rnd} \), sets \( \text{shift} \) to the exponent, and returns the error bound. We require that \( xn \) is positive and that the leading limb of \( x \) is nonzero.

**slong fmpr_set_round_ui_2exp_fmpz(fmpr_t z, mp_limb_t lo, const fmpz_t exp, int negative, slong prec, fmpr_rnd_t rnd)**
Sets \( z \) to the unsigned integer \( lo \) times two to the power \( exp \), negating the value if \( \text{negative} \) is nonzero, and rounding the result to \( \text{prec} \) bits in direction \( \text{rnd} \).

**slong fmpr_set_round_uiui_2exp_fmpz(fmpr_t z, mp_limb_t hi, mp_limb_t lo, const fmpz_t exp, int negative, slong prec, fmpr_rnd_t rnd)**

Sets \( z \) to the unsigned two-limb integer \( \{ hi, lo \} \) times two to the power \( exp \), negating the value if \( \text{negative} \) is nonzero, and rounding the result to \( \text{prec} \) bits in direction \( \text{rnd} \).

**void fmpr_set_error_result(fmpr_t err, const fmpr_t result, slong rret)**
Given the return value \( \text{rret} \) and output variable \( \text{result} \) from a function performing a rounding (e.g. \( \text{fmpr_set_round} \) or \( \text{fmpr_add} \)), sets \( \text{err} \) to a bound for the absolute error.

**void fmpr_add_error_result(fmpr_t err, const fmpr_t err_in, const fmpr_t result, slong rret, slong prec, fmpr_rnd_t rnd)**

Like \( \text{fmpr_set_error_result} \), but adds \( \text{err_in} \) to the error.

**void fmpr_ulp(fmpr_t u, const fmpr_t x, slong prec)**
Sets \( u \) to the floating-point unit in the last place (ulp) of \( x \). The ulp is defined as in the MPFR documentation and satisfies \( 2^{-n}|x| < u \leq 2^{-n+1}|x| \) for any finite nonzero \( x \). If \( x \) is a special value, \( u \) is set to the absolute value of \( x \).

**int fmpr_check_ulp(const fmpr_t x, slong r, slong prec)**
Assume that \( r \) is the return code and \( x \) is the floating-point result from a single floating-point rounding. Then this function returns nonzero iff \( x \) and \( r \) define an error of exactly 0 or 1 ulp. In other words, this function checks that \( \text{fmpr_set_error_result()} \) gives exactly 0 or 1 ulp as expected.

9.4. fmpr.h – Arb 1.x floating-point numbers (deprecated)
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, mpfr_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 $x$ to 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_set_fmpz(fmpz_t y, const fmpr_t x, mpfr_rnd_t rnd)
Sets $y$ to the exact value of the argument $x$ of type double.

void fmpr_set_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.

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 $man \times 2^{exp}$.

slong fmpr_set_round_fmpz_2exp(fmpr_t x, const fmpz_t man, const fmpz_t exp, slong prec, mpfr_rnd_t rnd)
Sets $x$ to $man \times 2^{exp}$, rounded according to $prec$ and $rnd$.

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.

9.4.5 Comparisons
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 $n \cdot 2^e$ for some integer `n`.

void `fmpr_abs_bound_le_2exp_fmpz`(fmpz_t `b`, const `fmpr_t x`)

Sets `b` to the smallest integer such that $|x| \leq 2^b$. If `x` is zero, infinity or NaN, the result is undefined.

void `fmpr_abs_bound_lt_2exp_fmpz`(fmpz_t `b`, const `fmpr_t x`)

Sets `b` to the smallest integer such that $|x| < 2^b$. If `x` is zero, infinity or NaN, the result is undefined.

`slong` `fmpr_abs_bound_lt_2exp_si`(const `fmpr_t x`)

Returns the smallest integer `b` such that $|x| < 2^b$, clamping the result to lie between `-FMPR_PREC_EXACT` and `FMPR_PREC_EXACT` inclusive. If `x` is zero, `-FMPR_PREC_EXACT` is returned, and if `x` is infinity or NaN, `FMPR_PREC_EXACT` is returned.

### 9.4.6 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.

### 9.4.7 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 \texttt{fmpr\_printd} (const \texttt{fmpr\_t} \(x\), \texttt{slong} \(\text{digits}\))

Prints \(x\) as a decimal floating-point number, rounding to the specified number of digits. This function is currently implemented using MPFR, and does not support large exponents.

### 9.4.8 Arithmetic

void \texttt{fmpr\_neg} (const \texttt{fmpr\_t} \(y\), const \texttt{fmpr\_t} \(x\))

Sets \(y\) to the negation of \(x\).

\texttt{slong} \texttt{fmpr\_neg\_round} (const \texttt{fmpr\_t} \(x\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

Sets \(y\) to the negation of \(x\), rounding the result.

void \texttt{fmpr\_abs} (const \texttt{fmpr\_t} \(y\), const \texttt{fmpr\_t} \(x\))

Sets \(y\) to the absolute value of \(x\).

\texttt{slong} \texttt{fmpr\_add} (const \texttt{fmpr\_t} \(y\), const \texttt{fmpr\_t} \(x\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

Sets \(z = x + y\), rounded according to \(\text{prec}\) and \(\text{rnd}\). The precision can be \texttt{FMPR\_PREC\_EXACT} to perform an exact addition, provided that the result fits in memory.

\texttt{slong} \texttt{fmpr\_add\_eps} (const \texttt{fmpr\_t} \(y\), const \texttt{fmpr\_t} \(x\), \texttt{int} \(\text{sign}\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

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.

\texttt{slong} \texttt{fmpr\_sub} (const \texttt{fmpr\_t} \(y\), const \texttt{fmpr\_t} \(x\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

\texttt{slong} \texttt{fmpr\_sub\_ui} (const \texttt{fmpr\_t} \(y\), \texttt{ulong} \(y\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

\texttt{slong} \texttt{fmpr\_sub\_si} (const \texttt{fmpr\_t} \(y\), \texttt{slong} \(y\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

\texttt{slong} \texttt{fmpr\_sub\_fmpz} (const \texttt{fmpr\_t} \(y\), const \texttt{fmpz\_t} \(y\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

Sets \(z = x - y\), rounded according to \(\text{prec}\) and \(\text{rnd}\). The precision can be \texttt{FMPR\_PREC\_EXACT} to perform an exact subtraction, provided that the result fits in memory.

\texttt{slong} \texttt{fmpr\_sum} (const \texttt{fmpr\_struct} * \(\text{terms}\), \texttt{slong len}, \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

Sets \(s\) to the sum of the array \(\text{terms}\) of length \(\text{len}\), rounded to \(\text{prec}\) bits in the direction \(\text{rnd}\). 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 \texttt{fmpr\_add}, this function does not overflow if the magnitudes of the terms are far apart. Warning: this function is implemented naively, and the running time is quadratic with respect to \(\text{len}\) in the worst case.

\texttt{slong} \texttt{fmpr\_mul} (const \texttt{fmpr\_t} \(y\), const \texttt{fmpr\_t} \(x\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

\texttt{slong} \texttt{fmpr\_mul\_ui} (const \texttt{fmpr\_t} \(y\), \texttt{ulong} \(y\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

\texttt{slong} \texttt{fmpr\_mul\_si} (const \texttt{fmpr\_t} \(y\), \texttt{slong} \(y\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

\texttt{slong} \texttt{fmpr\_mul\_fmpz} (const \texttt{fmpr\_t} \(y\), const \texttt{fmpz\_t} \(y\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

Sets \(z = x \times y\), rounded according to \(\text{prec}\) and \(\text{rnd}\). The precision can be \texttt{FMPR\_PREC\_EXACT} to perform an exact multiplication, provided that the result fits in memory.

\texttt{void} \texttt{fmpr\_mul\_2exp\_si} (const \texttt{fmpr\_t} \(y\), const \texttt{fmpz\_t} \(e\))

Sets \(y\) to \(x\) multiplied by \(2^e\) without rounding.

\texttt{slong} \texttt{fmpr\_div} (const \texttt{fmpr\_t} \(y\), const \texttt{fmpr\_t} \(x\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

\texttt{slong} \texttt{fmpr\_div\_ui} (const \texttt{fmpr\_t} \(y\), \texttt{ulong} \(y\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

\texttt{slong} \texttt{fmpr\_ui\_div} (const \texttt{fmpr\_t} \(y\), \texttt{ulong} \(y\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))

\texttt{slong} \texttt{fmpr\_div\_si} (const \texttt{fmpr\_t} \(y\), \texttt{slong} \(y\), \texttt{slong prec}, \texttt{fmpr\_rnd\_t} \(\text{rnd}\))
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)

Sets $z = x/y$, rounded according to prec and rnd. If $y$ is zero, $z$ is set to NaN.

void fmpr_divappr_abs_ubound(fmpr_t z, const fmpr_t x, const fmpr_t y, slong prec)
Sets $z$ to an upper bound for $|x|/|y|$, computed to a precision of approximately prec bits. The error can be a few ulp.

slong fmpr_addmul(fmpr_t z, const fmpr_t x, const fmpr_t y, slong prec, fmpr_rnd_t rnd)
slong fmpr_addmul_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_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_root(fmpr_t z, const fmpr_t x, ulong k, slong prec, fmpr_rnd_t rnd)
Sets $z$ to the $k$-th root of $x$, rounded to prec bits in the direction rnd. Warning: this function wraps MPFR, and is currently only fast for small $k$.

void fmpr_pow_sloppy_fmpz(fmpr_t y, const fmpr_t b, const fmpz_t e, slong prec, fmpr_rnd_t rnd)
void fmpr_pow_sloppy_ui(fmpr_t y, const fmpr_t b, ulong e, slong prec, fmpr_rnd_t rnd)
void fmpr_pow_sloppy_si(fmpr_t y, const fmpr_t b, slong e, slong prec, fmpr_rnd_t rnd)

Sets $y = b^e$, computed using without guaranteeing correct (optimal) rounding, but guaranteeing that the result is a correct upper or lower bound if the rounding is directional. Currently requires $b \geq 0$.

### 9.4.9 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.
Arb Documentation, Release 2.9.0

\texttt{slong fmpfr\_log1p(fmpr\_t y, const fmpr\_t x, slong prec, fmpr\_rnd\_t rnd)}

Sets $y$ to $\log(1 + x)$, rounded according to $\text{prec}$ and $\text{rnd}$. This function computes an accurate value when $x$ is small. The result is NaN if $1 + x$ is negative. This function is currently implemented using MPFR and does not support large exponents.

\texttt{slong fmpfr\_exp(fmpr\_t y, const fmpr\_t x, slong prec, fmpr\_rnd\_t rnd)}

Sets $y$ to $\exp(x)$, rounded according to $\text{prec}$ and $\text{rnd}$. This function is currently implemented using MPFR and does not support large exponents.

\texttt{slong fmpfr\_expm1(fmpr\_t y, const fmpr\_t x, slong prec, fmpr\_rnd\_t rnd)}

Sets $y$ to $\exp(x) - 1$, rounded according to $\text{prec}$ and $\text{rnd}$. This function computes an accurate value when $x$ is small. This function is currently implemented using MPFR and does not support large exponents.
SUPPLEMENTARY ALGORITHM NOTES

Here, we give extra proofs, error bounds, and formulas that would be too lengthy to reproduce in the documentation for each module.

10.1 General formulas and bounds

This section collects some results from real and complex analysis that are useful when deriving error bounds. Beware of typos.

10.1.1 Error propagation

We want to bound the error when \( f(x + a) \) is approximated by \( f(x) \). Specifically, the goal is to bound \( f(x + a) - f(x) \) in terms of \( r \) for the set of values \( a \) with \( |a| \leq r \). Most bounds will be monotone increasing with \( |a| \) (assuming that \( x \) is fixed), so for brevity we simply express the bounds in terms of \( |a| \).

**Theorem (generic first-order bound):**

\[
|f(x + a) - f(x)| \leq \min(2C_0, C_1|a|)
\]

where

\[
C_0 = \sup_{|t| \leq |a|} |f(x + t)|, \quad C_1 = \sup_{|t| \leq |a|} |f'(x + t)|.
\]

The statement is valid with either \( a, t \in \mathbb{R} \) or \( a, t \in \mathbb{C} \).

**Theorem (product):** For \( x, y \in \mathbb{C} \) and \( a, b \in \mathbb{C} \),

\[
|(x + a)(y + b) - xy| \leq |xb| + |ya| + |ab|.
\]

**Theorem (quotient):** For \( x, y \in \mathbb{C} \) and \( a, b \in \mathbb{C} \) with \( |b| < |y| \),

\[
\left| \frac{x}{y} - \frac{x + a}{y + b} \right| \leq \frac{|xb| + |ya|}{|y|(|y| - |b|)}.
\]

**Theorem (square root):** For \( x, a \in \mathbb{R} \) with \( 0 \leq |a| \leq x \),

\[
|\sqrt{x + a} - \sqrt{x}| \leq \sqrt{x} \left( 1 - \sqrt{1 - \frac{|a|}{x}} \right) \leq \frac{\sqrt{x}}{2} \left( \frac{|a|}{x} + \frac{|a|^2}{x^2} \right)
\]

where the first inequality is an equality if \( a \leq 0 \). (When \( x = a = 0 \), the limiting value is 0.)

**Theorem (reciprocal square root):** For \( x, a \in \mathbb{R} \) with \( 0 \leq |a| < x \),

\[
\left| \frac{1}{\sqrt{x + a}} - \frac{1}{\sqrt{x}} \right| \leq \frac{|a|}{2(x - |a|)^{3/2}}.
\]
Theorem (k-th root): For $k > 1$ and $x, a \in \mathbb{R}$ with $0 \leq |a| \leq x$,

$$
|(x + a)^{1/k} - x^{1/k}| \leq x^{1/k} \min \left(1, \frac{1}{k} \log \left(1 + \frac{|a|}{x - |a|}\right)\right).
$$

Proof: The error is largest when $a = -r$ is negative, and

$$
x^{1/k} - (x - r)^{1/k} = x^{1/k}[1 - (1 - r/x)^{1/k}]
= x^{1/k}[1 - \exp(\log(1 - r/x)/k)] \leq x^{1/k} \min(1, - \log(1 - r/x)/k)
= x^{1/k} \min(1, \log(1 + r/(x - r))/k).
$$

Theorem (sine, cosine): For $x, a \in \mathbb{R}$, $|\sin(x + a) - \sin(x)| \leq \min(2, |a|)$.

Theorem (logarithm): For $x, a \in \mathbb{R}$ with $0 \leq |a| < x$,

$$
|\log(x + a) - \log(x)| \leq \log \left(1 + \frac{|a|}{x - |a|}\right),
$$

with equality if $a \leq 0$.

Theorem (exponential): For $x, a \in \mathbb{R}$,

$$
|e^{x+a} - e^x| = e^x(e^a - 1) \leq e^x(e^{|a|} - 1),
$$

with equality if $a \geq 0$.

Theorem (inverse tangent): For $x, a \in \mathbb{R}$,

$$
|\tan(x + a) - \tan(x)| \leq \min(\pi, C_1 |a|).
$$

where

$$
C_1 = \sup_{|t| \leq |a|} \frac{1}{1 + (x + t)^2}.
$$

If $|a| < |x|$, then $C_1 = \left(1 + (|x| - |a|)^2\right)^{-1}$ gives a monotone bound.

An exact bound: if $|a| < |x|$ or $|x(x + a)| < 1$, then

$$
|\tan(x + a) - \tan(x)| = \tan \left(\frac{|a|}{1 + x(x + a)}\right).
$$

In the last formula, a case distinction has to be made depending on the signs of $x$ and $a$.

10.1.2 Sums and series

Theorem (geometric bound): If $|c_k| \leq C$ and $|z| \leq D < 1$, then

$$
\left| \sum_{k=N}^{\infty} c_k z^k \right| \leq \frac{CD^N}{1 - D}.
$$

Theorem (integral bound): If $f(x)$ is nonnegative and monotone decreasing, then

$$
\int_N^{\infty} f(x) \leq \sum_{k=N}^{\infty} f(k) \leq f(N) + \int_N^{\infty} f(x) dx.
$$

10.1.3 Complex analytic functions

Theorem (Cauchy’s integral formula): If $f(z) = \sum_{k=0}^{\infty} c_k z^k$ is analytic (on an open subset of $\mathbb{C}$ containing the disk $D = \{z : |z| \leq R\}$ in its interior, where $R > 0$), then

$$
c_k = \frac{1}{2\pi i} \int_{|z|=R} \frac{f(z)}{z^{k+1}} dz.
$$
Corollary (derivative bound):
\[ |c_k| \leq \frac{C}{R^k}, \quad C = \max_{|z|=R} |f(z)|. \]

Corollary (Taylor series tail): If \( 0 \leq r < R \) and \(|z| \leq r\), then
\[ \left| \sum_{k=N}^{\infty} c_k z^k \right| \leq \frac{CD^N}{1-D}, \quad D = \frac{r}{R}. \]

### 10.1.4 Euler-Maclaurin formula

**Theorem (Euler-Maclaurin):** If \( f(t) \) is \( 2M \)-times differentiable, then
\[
\begin{align*}
U & \sum_{k=L}^{U} f(k) = S + I + T + R \\
S &= \sum_{k=L}^{N-1} f(k), \quad I = \int_{N}^{U} f(t)dt, \\
T &= \frac{1}{2} (f(N) + f(U)) + \sum_{k=1}^{M} \frac{B_{2k}}{(2k)!} \left( f^{(2k-1)}(U) - f^{(2k-1)}(N) \right), \\
R &= - \int_{N}^{U} \frac{B_{2M}(t-\lfloor t \rfloor)}{(2M)!} f^{(2M)}(t)dt.
\end{align*}
\]

**Lemma (Bernoulli polynomials):** \(|B_n(t - \lfloor t \rfloor)| \leq 4n!/(2\pi)^n\).

**Theorem (remainder bound):**
\[ |R| \leq \frac{4}{(2\pi)^{2M}} \int_{N}^{U} |f^{(2M)}(t)| dt. \]

**Theorem (parameter derivatives):** If \( f(t) = f(t,x) = \sum_{k=0}^{\infty} a_k(t)x^k \) and \( R = R(x) = \sum_{k=0}^{\infty} c_k x^k \) are analytic functions of \( x \), then
\[ |c_k| \leq \frac{4}{(2\pi)^{2M}} \int_{N}^{U} |a_k^{(2M)}(t)| dt. \]

### 10.2 Algorithms for mathematical constants

Most mathematical constants are evaluated using the generic hypergeometric summation code.

#### 10.2.1 Pi

\( \pi \) is computed using the Chudnovsky series
\[
\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k(6k)!((13591409 + 545140134k)(3k))}{(3k)!k!^{3}640320^{3k+3/2}}
\]

which is hypergeometric and adds roughly 14 digits per term. Methods based on the arithmetic-geometric mean seem to be slower by a factor three in practice.

A small trick is to compute \( 1/\sqrt{640320} \) instead of \( \sqrt{640320} \) at the end.
10.2.2 Logarithms of integers

We use the formulas

$$\log(2) = 3 \sum_{k=0}^{\infty} \frac{(-1)^k (k)^2}{2k(2k + 1)!}$$

$$\log(10) = 46 \text{atanh}(1/31) + 34 \text{atanh}(1/49) + 20 \text{atanh}(1/161)$$

10.2.3 Euler’s constant

Euler’s constant $\gamma$ is computed using the Brent-McMillan formula ([BM1980], [MPFR2012])

$$\gamma = \frac{S_0(2n) - K_0(2n)}{I_0(2n)} - \log(n)$$

in which $n$ is a free parameter and

$$S_0(x) = \sum_{k=0}^{\infty} \frac{H_k}{(k!)^2} \left(\frac{x}{2}\right)^{2k}, \quad I_0(x) = \sum_{k=0}^{\infty} \frac{1}{(k!)^2} \left(\frac{x}{2}\right)^{2k}$$

$$2xI_0(x)K_0(x) \sim \sum_{k=0}^{\infty} \frac{[(2k)!!]^3}{(k!)^4 8^{k+2k} 2^k}.$$ 

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 $N \geq 4n + 1$ and $\alpha \approx 4.9706257595442318644$ satisfies $\alpha(\log \alpha - 1) = 3$. The third series is taken up to $k = 2n - 1$ inclusive. With these parameters, it is shown in [BJ2013] that the error is bounded by $24e^{-8n}$.

10.2.4 Catalan’s constant

Catalan’s constant is computed using the hypergeometric series

$$C = \sum_{k=0}^{\infty} \frac{(-1)^k 4^{4k+1} (40k^2 + 56k + 19) [(k+1)]^2[(2k+2)]^3}{(k+1)^3(2k+1)(4k+4)!^2}$$

10.2.5 Khinchin’s constant

Khinchin’s constant $K_0$ is computed using the formula

$$\log K_0 = \frac{1}{2} \log 2 \left[ \sum_{k=2}^{N-1} \log \left( \frac{k-1}{k} \right) \log \left( \frac{k+1}{k} \right) + \sum_{n=1}^{\infty} \frac{\zeta(2n, N)}{n} \sum_{k=1}^{2n-1} \frac{(-1)^{k+1}}{k} \right]$$

where $N \geq 2$ is a free parameter that can be used for tuning [BBC1997]. If the infinite series is truncated after $n = M$, the remainder is smaller in absolute value than

$$\sum_{n=M+1}^{\infty} \zeta(2n, N) = \sum_{n=M+1}^{\infty} \sum_{k=0}^{\infty} (k+N)^{-2n} \leq \sum_{n=M+1}^{\infty} \left( N^{-2n} + \int_0^{\infty} (t+N)^{-2n} dt \right)$$

$$= \sum_{n=M+1}^{\infty} \frac{1}{N^{2n}} \left( 1 + \frac{N}{2n-1} \right) \leq \sum_{n=M+1}^{\infty} \frac{N+1}{N^{2n}} = \frac{1}{N^{2M}} \frac{1}{2N-1} \leq \frac{1}{N^{2M}}.$$ 

Thus, for an error of at most $2^{-p}$ in the series, it is sufficient to choose $M \geq p/2 \log_2 N$. 

182 Chapter 10. Supplementary algorithm notes
10.2.6 Glaisher’s constant

Glaisher’s constant $A = \exp(1/12 - \zeta'(-1))$ is computed directly from this formula. We don’t use the reflection formula for the zeta function, as the arithmetic in Euler-Maclaurin summation is faster at $s = -1$ than at $s = 2$.

10.2.7 Apery’s constant

Apery’s constant $\zeta(3)$ is computed using the hypergeometric series

$$\zeta(3) = \frac{1}{64} \sum_{k=0}^{\infty} (-1)^k (205k^2 + 250k + 77) \frac{(2k)!^{10}}{[(2k + 1)!]^5}.$$  

10.3 Algorithms for the gamma function

10.3.1 The Stirling series

In general, the gamma function is computed via the Stirling series

$$\log \Gamma(z) = \left( z - \frac{1}{2} \right) \log z - z + \frac{\ln 2\pi}{2} + \sum_{k=1}^{n-1} \frac{B_{2k}}{2k(2k - 1)} z^{2k-1} + R(n, z)$$

where ([Ole1997] 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 [Ole1997] and the fact that the numerator of the integrand is bounded in absolute value by $2|B_{2n}|$, the remainder can be shown to satisfy the bound

$$|[t^k]R_n(z + t)| \leq 2|B_{2n}| \frac{\Gamma(2n + k - 1)}{\Gamma(k + 1)\Gamma(2n + 1)} |z| \left( \frac{b}{|z|} \right)^{2n+k}$$

where $b = 1 / \cos(\arg(z)/2)$. 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 $\Gamma(z) = \Gamma(z + r)/(z + r)(z + r + 1) \cdots (z + r - 1))$.

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 $\beta$ (e.g. $\beta = 1$) could be used to reduce the number of Bernoulli numbers that have to be precomputed, at the expense of slower repeated evaluation.

10.3.2 Rational arguments

We use efficient methods to compute $y = \Gamma(p/q)$ where $q$ is one of 1, 2, 3, 4, 6 and $p$ is a small integer.
The cases \( \Gamma(1) = 1 \) and \( \Gamma(1/2) = \sqrt{\pi} \) are trivial. We reduce all remaining cases to \( \Gamma(1/3) \) or \( \Gamma(1/4) \) using the following relations:

\[
\Gamma(2/3) = \frac{2\pi}{3^{1/2}\Gamma(1/3)}, \quad \Gamma(3/4) = \frac{2^{1/2}\pi}{\Gamma(1/4)}, \\
\Gamma(1/6) = \frac{\Gamma(1/3)^2}{(\pi/3)^{1/2}2^{1/3}}, \quad \Gamma(5/6) = \frac{2\pi(\pi/3)^{1/2}2^{1/3}}{(\Gamma(1/3)^2).}
\]

We compute \( \Gamma(1/3) \) and \( \Gamma(1/4) \) rapidly to high precision using

\[
\Gamma(1/3) = \left(\frac{12\pi^4}{\sqrt{10}} \sum_{k=0}^{\infty} \frac{(6k)!(-1)^k}{(k!)^3(3k)^3}160^{3k}\right)^{1/6}, \quad \Gamma(1/4) = \sqrt{\frac{(2\pi)^{3/2}}{\text{agm}(1, \sqrt{2})}}.
\]

An alternative formula which could be used for \( \Gamma(1/3) \) is

\[
\Gamma(1/3) = \frac{2^{4/9}\pi^{2/3}}{3^{1/12}\left(\text{agm}\left(1, \frac{1}{2}\sqrt{2 + \sqrt{3}}\right)\right)^{1/3},}
\]

but this appears to be slightly slower in practice.

## 10.4 Algorithms for the Hurwitz zeta function

### 10.4.1 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].

### 10.4.2 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 \( \sigma = \text{re}(s) \) with \( K + \sigma > 1 \). The tail is bounded by

\[
\sum_{k=K}^{\infty} \frac{|x|^k k! (s)_k}{k!} |\zeta(s + k, a)| \leq \sum_{k=K}^{\infty} \frac{|x|^k k! (s)_k}{k!} \left[ \frac{1}{a^{\sigma+k}} + \frac{1}{(\sigma+k-1)a^{\sigma+k-1}} \right].
\]

Denote the term on the right by \( T(k) \). Then

\[
\left| \frac{T(k+1)}{T(k)} \right| = \frac{|x|}{a} \frac{(k + \sigma - 1)(k + \sigma + a)}{(k + \sigma + a - 1)(k + 1)} \leq \frac{|x|}{a} \left( 1 + \frac{1}{K + \sigma + a - 1} \right) \left( 1 + \frac{|s - 1|}{K + 1} \right) = C
\]

and if \( C < 1 \),

\[
\sum_{k=K}^{\infty} T(k) \leq \frac{T(K)}{1 - C}.
\]

## 10.5 Algorithms for polylogarithms

The polylogarithm is defined for \( s, z \in \mathbb{C} \) with \( |z| < 1 \) by

\[
\text{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}
\]

and for \( |z| \geq 1 \) by analytic continuation, except for the singular point \( z = 1 \).
10.5.1 Computation for small $z$

The power sum converges rapidly when $|z| \ll 1$. To compute the series expansion with respect to $s$, we substitute $s \rightarrow s + x \in \mathbb{C}[[x]]$ and obtain

$$\text{Li}_{s+x}(z) = \sum_{d=0}^{\infty} x^d \frac{(-1)^d}{d!} \sum_{k=1}^{\infty} T(k)$$

where

$$T(k) = \frac{z^k \log^d(k)}{k^s}.$$ 

The remainder term $|\sum_{k=N}^{\infty} T(k)|$ is bounded via $\text{mag\_polylog\_tail()}$.

10.5.2 Expansion for general $z$

For general complex $s, z$, we write the polylogarithm as a sum of two Hurwitz zeta functions

$$\text{Li}_s(z) = \frac{\Gamma(v)}{(2\pi)^v} \left[ i^v \zeta \left( v, \frac{1}{2} + \log(-z) \right) + i^{-v} \zeta \left( v, \frac{1}{2} - \log(-z) \right) \right]$$

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 $v \rightarrow v + x \in \mathbb{C}[[x]]$ in this formula (at the end of the computation, we map $x \rightarrow -x$ to obtain the power series for $\text{Li}_{s+x}(z)$). The right hand side becomes

$$\Gamma(v + x)[E_1Z_1 + E_2Z_2]$$

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

$$E_1Z_1 + E_2Z_2 = (E_1 + E_2)/x + E_1\tilde{Z}_1 + E_2\tilde{Z}_2.$$ 

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 $[E_1Z_1 + E_2Z_2]/x$.

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 $|z| \rightarrow 0$ (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 $|z| \rightarrow \infty$, for which an asymptotic expansion would better.

10.6 Algorithms for hypergeometric functions

The algorithms used to compute hypergeometric functions are described in [Joh2016]. Here, we state the most important error bounds.
10.6.1 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

\[ \left| \sum_{k=n}^\infty T(k) \right| \leq C|T(n)|. \]

We check that \( \text{Re}(b + n) > 0 \) 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

\[ \left| \frac{a + k}{b + k} \right| = \frac{1 + \frac{a - b}{b + k}}{1 + \frac{a - b}{b + k}} \leq 1 + \frac{|a - b|}{|b + n|} \]

and

\[ \left| \frac{1}{b + k} \right| \leq \frac{1}{|b + n|} \]

for all \( k \geq n \). This gives us a constant \( D \) such that \( T(k + 1) \leq DT(k) \) for all \( k \geq n \). If \( D \geq 1 \), we set \( C \) to infinity. Otherwise, we take \( C = \sum_{k=0}^\infty D^k = (1 - D)^{-1} \).

10.6.2 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 \( \sum_{k=N}^\infty T(k) \in \mathbb{C}[x]/(x^n) \) given \( a_i, b_i, z \in \mathbb{C}[x]/(x^n) \). First, we fix some notation, assuming that \( A \) and \( B \) are power series:

- \( A_{[k]} \) denotes the coefficient of \( x^k \) in \( A \), and \( A_{[m:n]} \) denotes the power series \( \sum_{k=m}^{n-1} A_{[k]} x^k \).
- \( |A| \) denotes \( \sum_{k=0}^\infty |A_{[k]}| x^k \) (this can be viewed as an element of \( \mathbb{R}_{\geq 0}[[x]] \)).
- \( A \leq B \) signifies that \( |A_{[k]}| \leq |B_{[k]}| \) holds for all \( k \).
- We define \( \mathcal{R}(B) = |B_{[0]}| - |B_{[1:\infty]}| \).

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 \( \text{Re}(b_{[i]} + N) > 0 \) for all \( b_i \), then we may take

\[ D = |z| \prod_{i=1}^{p} \left( 1 + \frac{|a_i - b_i|}{\mathcal{R}(b_i + N)} \right) \prod_{i=p+1}^{q} \frac{1}{\mathcal{R}(b_i + N)}. \]

If \( D_{[0]} < 1 \), then \( (1 - D)^{-1}|T(n)| \) gives the error bound.

Note when adding and multiplying power series with (complex) interval coefficients, we can use point-valued upper bounds for the absolute values instead of performing interval arithmetic throughout. For \( \mathcal{R}(B) \), we must then pick a lower bound for \( |B_{[0]}| \) and upper bounds for the coefficients of \( |B_{[1:\infty]}| \).
10.6.3 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 \( U^* = z^b U(a, b, z) \). 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

\[
\frac{1}{\gamma(b)} = \frac{(-z)^{-a}}{\Gamma(b - a)} U^*(a, b, z) + \frac{z^{a-b} e^z}{\Gamma(a)} U^*(b - a, b, -z)
\]

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 \( \Re(z) \geq r \), set \( R = 1 \). Otherwise, if \( \Im(z) \geq r \) or \( \Re(z) \geq 0 \) and \( |z| \geq r \), set \( R = 2 \). Otherwise, if \( |z| \geq 2r \), set \( R = 3 \). Otherwise, the bound is infinite. If the bound is finite, we have

\[
|\varepsilon_n(z)| \leq 2 \alpha C_n \left| \frac{(a)_n(a-b+1)_n}{n!z^n} \right| \exp(2\alpha \rho C_1/|z|)
\]

in terms of the following auxiliary quantities

\[
\sigma = |(b - 2a)/z|
\]

\[
C_n = \begin{cases} 
1 & \text{if } R = 1 \\
\chi(n) & \text{if } R = 2 \\
(\chi(n) + \sigma \nu^2 n) \nu^n & \text{if } R = 3
\end{cases}
\]

\[
\nu = \left( \frac{1}{2} + \frac{1}{2} \sqrt{1 - 4\sigma^2} \right)^{-1/2} \leq 1 + 2\sigma^2
\]

\[
\chi(n) = \sqrt{\pi} \Gamma\left(\frac{1}{2}n + 1\right)/\Gamma\left(\frac{1}{2}n + \frac{1}{2}\right)
\]

\[
\sigma' = \begin{cases} 
\sigma & \text{if } R \neq 3 \\
\nu \sigma & \text{if } R = 3
\end{cases}
\]

\[
\alpha = (1 - \sigma')^{-1}
\]

\[
\rho = \frac{1}{2} |2a^2 - 2ab + b| + \sigma'(1 + \frac{1}{2} \sigma')(1 - \sigma')^{-2}
\]
10.6.4 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

\[
\Ai(z) = \frac{e^{-\zeta}}{2\sqrt{\pi}z^{1/4}} \left[S_n(\zeta) + R_n(z)\right], \quad \Ai'(z) = -\frac{z^{1/4}e^{-\zeta}}{2\sqrt{\pi}} \left[\left(S'_n(\zeta) + R'_n(z)\right)\right]
\]

\[
S_n(\zeta) = \sum_{k=0}^{n-1} (-1)^k \frac{u(k)}{\zeta^k}, \quad S'_n(\zeta) = \sum_{k=0}^{n-1} (-1)^k \frac{v(k)}{\zeta^k}
\]

\[
u(k) = \frac{(1/6)k(5/6)_k}{2^k k!}, \quad v(k) = \frac{6k+1}{1-6k} u(k).
\]

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

\[
C = \begin{cases}
2 \exp(7/36|\zeta|) & |\arg(z)| \leq \pi/3 \\
2X(n) \exp(7\pi/72|\zeta|) & \pi/3 \leq |\arg(z)| \leq 2\pi/3 \\
4X(n) \exp(7\pi/36 \cos(\arg(\zeta))) \cos(\arg(\zeta))^{-n} & 2\pi/3 \leq |\arg(z)| < \pi.
\end{cases}
\]

For computing \( \Bi(z) \) when \( z \) is roughly in the positive half-plane, we use the connection formulas

\[
\Bi(z) = -i(2w+1) \Ai(zw^{-2}) - \Ai(z)) \\
\Bi(z) = +i(2w^{-1} \Ai(zw^{-2}) - \Ai(z))
\]

where \( w = \exp(\pi i/3) \). Combining roots of unity gives

\[
\Bi(z) = \frac{1}{2\sqrt{\pi}z^{1/4}} \left[2X + iY\right]
\]

\[
\Bi(z) = \frac{1}{2\sqrt{\pi}z^{1/4}} \left[2X - iY\right]
\]

\[
X = \exp(+\zeta)[S_n(-\zeta) + R_n(zw^{-2})], \quad Y = \exp(-\zeta)[S_n(\zeta) + R_n(z)]
\]

where the upper formula is valid for \(-\pi/3 < \arg(z) < \pi \) and the lower formula is valid for \(-\pi < \arg(z) < \pi/3 \). We proceed analogously for the derivative of \( \Bi(z) \).

In the negative half-plane, we use the connection formulas

\[
\Ai(z) = e^{+\pi i/3} \Ai(z_1) + e^{-\pi i/3} \Ai(z_2)
\]

\[
\Bi(z) = e^{-\pi i/6} \Ai(z_1) + e^{+\pi i/6} \Ai(z_2)
\]

where \( z_1 = -ze^{+\pi i/3}, z_2 = -ze^{-\pi i/3} \). Provided that |\arg(-z)| < 2\pi/3, we have |\arg(z_1)|, |\arg(z_2)| < \pi, and thus the asymptotic expansion for \( \Ai(z) \) can be used. As before, we collect roots of unity to obtain

\[
\Ai(z) = A_1[S_n(i\zeta) + R_n(z_1)] + A_2[S_n(-i\zeta) + R_n(z_2)]
\]

\[
\Bi(z) = A_3[S_n(i\zeta) + R_n(z_1)] + A_4[S_n(-i\zeta) + R_n(z_2)]
\]

where \( \zeta = (2/3)(-z)^{3/2} \) and

\[
A_1 = \frac{\exp(-i(\zeta - \pi/4))}{2\sqrt{\pi}(-z)^{1/4}}, \quad A_2 = \frac{\exp(i(\zeta - \pi/4))}{2\sqrt{\pi}(-z)^{1/4}}, \quad A_3 = -IA_1, \quad A_4 = IA_2.
\]

The differentiated formulas are analogous.
10.6.5 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

$$0 \rightarrow 0.375 \pm 0.625i \rightarrow 0.5 \pm 0.8125i \rightarrow z$$

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 $A, N, \nu$ such that the simple solution $g(z) = A(1 - \nu z)^{-N}$ of the differential equation provides the bound

$$|f[k]| \leq g[k] = A\left(\frac{N + k}{k}\right)\nu^k.$$

10.7 Algorithms for the arithmetic-geometric mean

With complex variables, it is convenient to work with the univariate function $M(z) = \text{agm}(1, z)$. The general case is given by $\text{agm}(a, b) = aM(1, b/a)$.

10.7.1 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 $M'(z) = [M(u) - (z - 1)M'(u)]/((1 + z)\sqrt{z})]/2$.

10.7.2 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 $\sqrt{ab}, i\sqrt{-ab}, -i\sqrt{-ab}, \sqrt{ab}$ respectively if both $a$ and $b$ have positive real part, nonnegative imaginary part, nonpositive imaginary part, or otherwise.

It is shown in [Dup2006], p. 87 that, for $z$ with nonnegative real part, $|M(z) - a_n| \leq |a_n - b_n|$.

A small improvement would be to switch to a series expansion for the last one or two steps.
10.7.3 First derivative

Assuming that \( z \) is exact and that \( |\arg(z)| \leq 3\pi/4 \), we compute \((M(z), M'(z))\) simultaneously using a finite difference.

The basic inequality we need is \( |M(z)| \leq \max(1, |z|) \), which is an immediate consequence of the AGM iteration.

By Cauchy’s integral formula, \( |M^{(k)}(z)/k!| \leq CD^k \) where \( C = \max(1, |z| + r) \) and \( D = 1/r \), for any \( 0 < r < |z| \) (we choose \( r \) to be of the order \( |z|/4 \)). Taylor expansion now gives

\[
\frac{|M(z+h) - M(z) - M'(z)h|}{h} \leq \frac{CD^2h}{1 - Dh}
\]

assuming that \( h \) is chosen so that it satisfies \( hD < 1 \).

The forward finite difference requires two function evaluations at doubled precision. It would be more efficient to use a central difference (this could be implemented in the future).

When \( z \) is not exact, we evaluate at the midpoint as above and bound the propagated error using derivatives. Again by Cauchy’s integral formula, we have

\[
|M'(z + \epsilon)| \leq \frac{\max(1, |z| + |\epsilon| + r)}{r}
\]

\[
|M''(z + \epsilon)| \leq \frac{2\max(1, |z| + |\epsilon| + r)}{r^2}
\]

assuming that the circle centered on \( z \) with radius \( |\epsilon| + r \) does not cross the negative half axis. We choose \( r \) of order \( |z|/2 \) and verify that all assumptions hold.

10.7.4 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

\[-2z(z^2 - 1)c_2 = (3z^2 - 1)c_1 + zc_0,\]

\[-(k + 2)(k + 3)z(z^2 - 1)c_{k+3} = (k + 2)^2(3z^2 - 1)c_{k+2} + (3k(k + 3) + 7)zc_{k+1} + (k + 1)^2c_k\]

in general, and

\[-(k + 2)^2c_{k+2} = (3k(k + 3) + 7)c_{k+1} + (k + 1)^2c_k\]

when \( z = 1 \).
11.1 Credits and references

11.1.1 License

Arb is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License (LGPL) as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

Arb is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Arb (see the LICENSE file in the root of the Arb source directory). If not, see http://www.gnu.org/licenses/.

Versions of Arb up to and including 2.8 were distributed under the GNU General Public License (GPL), not the LGPL. The switch to the LGPL applies retroactively; i.e. users may redistribute older versions of Arb under the LGPL if they prefer.

11.1.2 Authors

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

The following authors have developed major new features.

- Pascal Molin - 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

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
• Marc Mezzarobba - work on Arb interface in Sage, bug reports, feedback
• Ricky Farr - convenience functions, feedback
• Marcello Seri - fix for static builds on OS X
• Tommy Hofmann - matrix transpose, comparison, other utility methods, Julia interface
• Alexander Kobel - documentation and code cleanup patches
• Hrvoje Abraham - patches for MinGW compatibility
• Julien Puydt - soname versioning support
• Jeroen Demeyer - patch for major bug on PPC64
• Isuru Fernando - continuous integration setup, support for cmake and MSVC builds

11.1.3 Funding

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

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

11.1.4 Software

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

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

If you wish to cite Arb in a scientific paper, the following reference can be used (you may also cite the manual or the website directly):


In BibTeX format:

```bibtex
@article{Johansson2013arb,  
title={Arb: a C library for ball arithmetic},  
author={F. Johansson},  
journal={ACM Communications in Computer Algebra},  
volume={47},  
number={4},  
pages={166--169},  
year={2013},  
publisher={ACM}
}
```

11.1.6 Bibliography

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

11.2 History and changes

For more details, view the commit log in the git repository https://github.com/fredrik-johansson/arb

Old releases of the code can be accessed from https://github.com/fredrik-johansson/arb/releases

11.2.1 Old versions of the documentation

- http://arblib.org/arb-2.7.0.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

11.2.2 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.
• 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_fmpq 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).

Polyomials
- 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_mul and acb_poly_mul 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.

11.2.3 2015-12-31 - version 2.8.1

• Fixed 32-bit test failure for the Laguerre function.
• Made the Laguerre function indeterminate at negative integer orders, to be consistent with the test code.

11.2.4 2015-12-29 - version 2.8.0

• Compatibility and build system
  – Windows64 support (contributed by Bill Hart).
  – Fixed a bug that broke basic arithmetic on targets where FLINT uses fallback code instead of assembly code, such as PPC64 (contributed by Jeroen Demeyer).
  – Fixed configure to use EXTRA_SHARED_FLAGS/LDFLAGS, and other build system fixes (contributed by Tommy Hofmann, Bill Hart).
  – Added soname versioning (contributed by Julien Puydt).
  – Fixed test code on MinGW (contributed by Hrvoje Abraham).
  – Miscellaneous fixes to simplify interfacing Arb from Julia.

• Arithmetic and elementary functions
  – Fixed arf_get_d to handle underflow/overflow correctly and to support round-to-nearest.
  – Added more complex inverse hyperbolic functions (acb_asin, acb_acos, acb_asinh, acb_acosh, acb_atanh).
  – Added arb_contains_int and acb_contains_int for testing whether an interval contains any integer.
  – Added acb_quadratic_roots_fmpz.
  – Improved arb_sinh to use a more accurate formula for x < 0.
  – Added sinc function (arb_sinc) (contributed by Alex Griffing).
  – Fixed bug in arb_exp affecting convergence for huge input.
  – Faster implementation of arb_div_2expm1_ui.
  – Added mag_root, mag_geom_series.
  – Improved and added test code for arb_add_error functions.
  – Changed arb_pow and acb_pow to make pow(0,positive) = 0 instead of nan.
  – Improved acb_sqrt to return finite output for finite input straddling the branch cut.
  – Improved arb_set_interval_arf so that [inf,inf] = inf instead of an infinite interval.
  – Added computation of Bell numbers (arb_bell_fmpz).
- Added `arb_power_sum_vec` for computing power sums using Bernoulli numbers.
- Added computation of the Fujiwara root bound for `acb_poly`.
- Added code to identify all the real roots of a real polynomial (`acb_poly_validate_real_roots`).
- Added several convenient assignment functions, including `arb_set_d`, `acb_set_d`, `acb_set_d_d`, `acb_set_fmpz_fmpz` (contributed by Ricky Farr).
- Added many accessor functions (`_arb/acb_vec_entry_ptr`, `arb_get_mid/rad_arb`, `acb_real/imag_ptr`, `arb_mid/rad_ptr`, `acb_get_real/imag`).
- Added missing functions `acb_add_si`, `acb_sub_si`.
- Renamed `arb_root` to `arb_root_ui` (keeping alias) and added `acb_root_ui`.

- **Special functions**
  - Implemented the Gauss hypergeometric function 2F1 and its regularized version.
  - Fixed two bugs in `acb_hypgeom_pfq_series_direct` discovered while implementing 2F1. In rare cases, these could lead to incorrect values for functions depending on parameter derivatives of hypergeometric series.
    * The first bug involved incorrect handling of negative integer parameters. The bug only affected 2F1 and higher functions; it did not affect correctness of any previously implemented functions that relied on `acb_hypgeom_pfq_series_direct` (such as Bessel Y and K functions of integer order).
    * The second bug involved a too small bound being computed for the sum of a geometric series. The geometric series bound is nearly tight for 2F1, and the incorrect version caused immediate test failures for that function. Theoretically, this bug affected correctness of some previously-implemented functions that relied on `acb_hypgeom_pfq_series_direct` (such as Bessel Y and K functions of integer order), but since the geometric bound is not as tight in those cases, those functions were still reliable in practice (no failing test case has been found).
  - Implemented Airy functions and their derivatives (`acb_hypgeom_airy`).
  - Implemented the confluent hypergeometric function 0F1 (`acb_hypgeom_0f1`).
  - Implemented associated Legendre functions P and Q.
  - Implemented Chebyshev, Jacobi, Gegenbauer, Laguerre, Hermite functions.
  - Implemented spherical harmonics.
  - Added function for computing Bessel J and Y functions simultaneously.
  - Added rising factorials for non-integer n (`arb_rising`, `acb_rising`).
  - Made rising factorials use gamma function for large integer n.
  - Faster algorithm for theta constants and Dedekind eta function at very high precision.
  - Fixed erf to give finite values instead of +/-inf for big imaginary input.
  - Improved `acb_zeta` (and `arb_zeta`) to automatically use fast code for integer zeta values.
  - Added double factorial (`arb_doublefac_ui`).
  - Added code for generating Hilbert class polynomials (`acb_modular_hilbert_class_poly`).

- **Matrices**
  - Added faster matrix squaring (`arb/acb_mat_sqr`) (contributed by Alex Griffing).
  - Added matrix trace (`arb/acb_mat_trace`) (contributed by Alex Griffing).
  - Added `arb/acb_mat_set_round_fmpz_mat`, `acb_mat_set(_round)_arb_mat` (contributed by Tommy Hofmann).
  - Added `arb/acb_mat_transpose` (contributed by Tommy Hofmann).
- Added comparison methods arb/acb_mat_eq/ne (contributed by Tommy Hofmann).

**Other**

- Added complex_plot example program.
- Added Airy functions to real_roots example program.
- Other minor patches were contributed by Alexander Kobel, Marc Mezzarobba, Julien Puydt.
- Removed obsolete file config.h.

### 11.2.5 2015-07-14 - version 2.7.0

**Hypergeometric functions**

- Implemented Bessel I and Y functions (acb_hypgeom_bessel_i, acb_hypgeom_bessel_y).
- Fixed bug in Bessel K function giving the wrong branch for negative real arguments.
- Added code for evaluating complex hypergeometric series binary splitting.
- Added code for evaluating complex hypergeometric series using fast multipoint evaluation.

**Gamma related functions**

- Implemented the Barnes G-function and its continuous logarithm (acb_barnes_g, acb_log_barnes_g).
- Implemented the generalized polygamma function (acb_polygamma).
- Implemented the reflection formula for the logarithmic gamma function (acb_lgamma, acb_poly_lgamma_series).
- Implemented the digamma function of power series (arb_poly_digamma_series, acb_poly_digamma_series).
- Improved acb_poly_zeta_series to produce exact zero imaginary parts in most cases when the result should be real-valued.
- Made the real logarithmic gamma function (arb_lgamma, arb_poly_lgamma_series) abort more quickly for negative input.

**Elementary functions**

- Added arb_exp_expinv and acb_exp_expinv functions for simultaneously computing exp(x), exp(-x).
- Improved acb_tan, acb_tan_pi, acb_cot and acb_cot_pi for input with large imaginary parts.
- Added complex hyperbolic functions (acb_sinh, acb_cosh, acb_sinh_cosh, acb_tanh, acb_coth).
- Added acb_log_sin_pi for computing the logarithmic sine function without branch cuts away from the real line.
- Added arb_poly_cot_pi_series, acb_poly_cot_pi_series.
- Added arf_root and improved speed of arb_root.
- Tuned algorithm selection in arb_pow_fmpq.

**Other**

- Added documentation for arb and acb vector functions.
11.2.6 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 $e_i, s_i, shi, ci, chi, li$.
  - Added the complete elliptic integral of the second kind $E$.
  - Added support for computing hypergeometric functions with power series as parameters.
  - Fixed special cases in Bessel J function returning useless output.
  - Fixed precision of zeta function accidentally being capped at 7000 digits (bug in 2.5).
  - Special-cased real input in the gamma functions for complex types.
  - Fixed exp of huge numbers outputting unnecessarily useless intervals.
  - Fixed broken code in erf that sometimes gave useless output.
  - Made selection of number of terms in hypergeometric series more robust.

- Polynomials and power series.
  - Added $\sin_{\pi}, \cos_{\pi}$ and $\sin_{\cos_{\pi}}$ for real and complex power series.
  - Speeded up series reciprocal and division for length = 2.
  - Added $add_{\_si}$ methods for polynomials.
  - Made $inv_{\_series}$ and $div_{\_series}$ with zero input produce indeterminates instead of aborting.
  - Added $arb_{\_poly_{\_majorant}}, acb_{\_poly_{\_majorant}}$.

- Basic functions
  - Added comparison methods $arb_{\_eq}, arb_{\_ne}, arb_{\_lt}, arb_{\_le}, arb_{\_gt}, arb_{\_ge}, acb_{\_eq}, acb_{\_ne}$.
  - Added $acb_{\_rel_{\_accuracy_{\_bits}}}$ and improved the real version.
  - Fixed precision of constants like $\pi$ behaving more nondeterministically than necessary.
  - Fixed $arf_{\_get_{\_mag_{\_lower}}}(nan)$ to output 0 instead of inf.

- Other
  - Removed call to $fmpq_{\_dedekind_{\_sum}}$ which only exists in the git version of flint.
  - Fixed a test code bug that could cause crashes on some systems.
  - Added fix for static build on OS X (thanks Marcello Seri).
  - Miscellaneous corrections to the documentation.

11.2.7 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 exponential integral.
  – Implemented the complete elliptic integral of the first kind.
  – Implemented the arithmetic-geometric mean of complex numbers.
  – Optimized arb_digamma for small integers.
  – Made mag_log_ui, mag_binpow_uiui and mag_polylog_tail proper functions.
  – Added pow, agm, erf, elliptic_k, elliptic_p as functions of complex power series.
  – Added incomplete gamma function of complex power series.
  – Improved code for bounding complex rising factorials (the old code could potentially have
given wrong results in degenerate cases).
  – Added arb_sqrt1pm1, arb_atanh, arb_asinh, arb_atanh.
  – Added arb_log1p, acb_log1p, acb_atan.
  – Added arb_hurwitz_zeta.
  – Improved parameter selection in the Hurwitz zeta function to try to avoid stalling when given
enormous input.
  – Optimized sqrt and rsqrt of power series when given a binomial as input.
  – Made arb_bernoulli_ui(2^64-2) not crash.
  – Fixed rgamma of negative integers returning indeterminate.

• Polynomials and matrices
  – Added characteristic polynomial computation for real and complex matrices.
  – Added polynomial set_round methods.
  – Added is_real methods for more types.
  – Added more get_unique_fmpz methods.
  – Added code for generating Swinnerton-Dyer polynomials.
  – Improved error bounding in det() and exp() of complex matrices to recognize when the result
is real-valued.
  – Changed polynomial divrem to return success/fail instead of aborting on divide by zero.

• Miscellaneous
  – Added logo to documentation.
  – Made inlined functions build as part of the library.
  – Silenced a clang warning.
  – Made _acb_vec_sort_pretty a library function.
11.2.8 2014-11-15 - version 2.4.0

- Arithmetic and core functions
  - Made evaluation of sin, cos and exp at medium precision faster using the sqrt trick.
  - Optimized arb_sinh and arb_sinh_cosh.
  - Optimized complex division with a small denominator.
  - Optimized cubing of complex numbers.
  - Added floor and ceil functions for the arf and arb types.
  - Added acb_poly powering functions.
  - Added acb_exp_pi_i.
  - Added functions for evaluation of Chebyshev polynomials.
  - Fixed arb_div to output nan for input containing nan.
- Added a module acb_hypgeom for hypergeometric functions
  - Evaluation of the generalized hypergeometric function in convergent cases.
  - Evaluation of confluent hypergeometric functions using asymptotic expansions.
  - The Bessel function of the first kind for complex input.
  - The error function for complex input.
- Added a module acb_modular for modular forms and elliptic functions
  - Support for working with modular transformations.
  - Mapping a point to the fundamental domain.
  - Evaluation of Jacobi theta functions and their series expansions.
  - The Dedekind eta function.
  - The j-invariant and the modular lambda and delta function.
  - Eisenstein series.
  - The Weierstrass elliptic function and its series expansion.
- Miscellaneous
  - Fixed mag_print printing a too large exponent.
  - Fixed printd methods to use a fallback instead of aborting when printing numbers too large for MPFR.
  - Added version number string (arb_version).
  - Various additions to the documentation.

11.2.9 2014-09-25 - version 2.3.0

- Removed most of the legacy (Arb 1.x) modules.
- Updated build scripts, hopefully fixing various issues.
- New implementations of arb_sin, arb_cos, arb_sin_cos, arb_atan, arb_log, arb_exp, arb_expm1, much faster up to a few thousand bits.
- Ported the bit-burst code for high-precision exponentials to the arb type.
- Speeded up arb_log_ui_from_prev.
- Added mag_exp, mag_expm1, mag_exp_tail, mag_pow_fmpz.
• Improved various mag functions.
• Added arb_get/set_interval_mpfr, arb_get_interval_arf, and improved arb_set_interval_arf.
• Improved arf_get_fmpz.
• Prettier printing of complex numbers with negative imaginary part.
• Changed some frequently-used functions from inline to non-inline to reduce code size.

11.2.10 2014-08-01 - version 2.2.0

• Added functions for computing polylogarithms and order expansions of polylogarithms, with support for real and complex s, z.
• Added a missing cast affecting C++ compatibility.
• Generalized powsum functions to allow a geometric factor.
• Improved powsum functions slightly when the exponent is an integer.
• Faster arb_log__ui_from__prev.
• Added mag_sqrt and mag_rsqrt functions.
• Fixed various minor bugs and added missing tests and documentation entries.

11.2.11 2014-06-20 - version 2.1.0

• Ported most of the remaining functions to the new arb/acb types, including:
  – Elementary functions (log, atan, etc.).
  – Hypergeometric series summation.
  – The gamma function.
  – The Riemann zeta function and related functions.
  – Bernoulli numbers.
  – The partition function.
  – The calculus modules (rigorous real root isolation, rigorous numerical integration of complex-valued functions).
  – Example programs.
• Added several missing utility functions to the arf and mag modules.

11.2.12 2014-05-27 - version 2.0.0

• New modules mag, arf, arb, arb_poly, arb_mat, acb, acb_poly, acb_mat for higher-performance ball arithmetic.
• Poly_roots2 and hilbert_matrix2 example programs.
• Vector dot product and norm functions (contributed by Abhinav Baid).

11.2.13 2014-05-03 - version 1.1.0

• Faster and more accurate error bounds for polynomial multiplication (error bounds are now always as good as with classical multiplication, and multiplying high-degree polynomials with approximately equal coefficients now has proper quasilinear complexity).
• Faster and much less memory-hungry exponentials at very high precision.
• Improved the partition function to support \( n \) bigger than a single word, and enabled the possibility to use two threads for the computation.

• Fixed a bug in floating-point arithmetic that caused a too small bound for the rounding error to be reported when the result of an inexact operation was rounded up to a power of two (this bug did not affect the correctness of ball arithmetic, because operations on ball midpoints always round down).

• Minor optimizations to floating-point arithmetic.

• Improved argument reduction of the digamma function and short series expansions of the rising factorial.

• Removed the holonomic module for now, as it did not really do anything very useful.

11.2.14 2013-12-21 - version 1.0.0

• New example programs directory
  – poly_roots example program.
  – real_roots example program.
  – pi_digits example program.
  – hilbert_matrix example program.
  – keiper_li example program.

• New fmprb_calc module for calculus with real functions
  – Bisection-based root isolation.
  – Asymptotically fast Newton root refinement.

• New fmpcb_calc module for calculus with complex functions
  – Numerical integration using Taylor series.

• Scalar functions
  – Simplified fmprb_const_euler using published error bound.
  – Added fmprb_inv.
  – Added fmprb_trim, fmpcb_trim.
  – Added fmpcb_rsqrt (complex reciprocal square root).
  – Fixed bug in fmprb_sqrtpos with nonfinite input.
  – Slightly improved fmprb powering code.
  – Added various functions for bounding fmprs by powers of two.
  – Added fmpr_is_int.

• Polynomials and power series
  – Implemented scaling to speed up blockwise multiplication.
  – Slightly faster basecase power series exponentials.
  – Improved sin/cos/tan/exp for short power series.
  – Added complex sqrt_series, rsqrt_series.
  – Implemented the Riemann-Siegel Z and theta functions for real power series.
  – Added fmprb_poly_pow_series, fmprb_poly_pow_ui and related methods.
  – Added fmprb/fmpcb_poly_contains_fmpz_poly.
• Faster composition by monomials.
• Implemented Borel transform and binomial transform for real power series.

• Matrices
  • Implemented matrix exponentials.
  • Multithreaded fmprb_mat_mul.
  • Added matrix infinity norm functions.
  • Added some more matrix-scalar functions.
  • Added matrix contains and overlaps methods.

• Zeta function evaluation
  • Multithreaded power sum evaluation.
  • Faster parameter selection when computing many derivatives.
  • Implemented binary splitting to speed up computing many derivatives.

• Miscellaneous
  • Corrections for C++ compatibility (contributed by Jonathan Bober).
  • Several minor bugfixes and test code enhancements.

11.2.15 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_rsqr, fmprb_rsqr) based on mpfr_rec_sqrt.
  • Faster high-precision division by not computing an explicit remainder.
  • Slightly faster computation of pi by using new reciprocal square root and division code.
  • Added an fmpr function for approximate division to speed up certain radius operations.
  • Added fmpr_set_d for conversion from double.
  • Allow use of doubles to optionally compute the partition function faster but without an error bound.
  • Bypass mpfr overflow when computing the exponential function to extremely high precision (approximately 1 billion digits).
  • Made fmprb_exp faster for large numbers at extremely high precision by skipping the log(2) removal.
  • Made fmpcb_lgamma faster at high precision by speeding up the argument reduction branch computation.
  • Added fmprb_asin, fmprb_acos.
  • Added various other utility functions to the fmprb module.
  • Added a function for computing the Glaisher constant.
  • Optimized evaluation of the Riemann zeta function at high precision.

• Polynomials and power series
  • Made squaring of polynomials faster than generic multiplication.
  • Implemented power series reversion (various algorithms) for the fmprb_poly type.
– Added many fmprb_poly utility functions (shifting, truncating, setting/getting coefficients, etc.).
– Improved power series division when either operand is short
– Improved power series logarithm when the input is short.
– Improved power series exponential to use the basecase algorithm for short input regardless of the output size.
– Added power series square root and reciprocal square root.
– Added atan, tan, sin, cos, sin_cos, asin, acos fmprb_poly power series functions.
– Added Newton iteration macros to simplify various functions.
– Added gamma functions of real and complex power series ([fmprb/fmpcb]_poly_[gamma/rgamma/lgamma]_series).
– Added wrappers for computing the Hurwitz zeta function of a power series ([fmprb/fmpcb]_poly_zeta_series).
– Implemented sieving and other optimizations to improve performance for evaluating the zeta function of a short power series.
– Improved power series composition when the inner series is linear.
– Added many fmpcb_poly versions of nearly all fmprb_poly functions.
– Improved speed and stability of series composition/reversion by balancing the power table exponents.

• Other
  – Added support for freeing all cached data by calling flint_cleanup().
  – Introduced fmprb_ptr, fmprb_srcptr, fmpcb_ptr, fmpcb_srcptr typedefs for cleaner function signatures.
  – Various bug fixes and general cleanup.

11.2.16 2013-05-31 - version 0.6

• Made fast polynomial multiplication over the reals numerically stable by using a blockwise algorithm.
• Disabled default use of the Gauss formula for multiplication of complex polynomials, to improve numerical stability.
• Added division and remainder for complex polynomials.
• Added fast multipoint evaluation and interpolation for complex polynomials.
• Added missing fmprb_poly_sub and fmpcb_poly_sub functions.
• Faster exponentials (fmprb_exp and dependent functions) at low precision, using precomputation.
• Rewrote fmpr_add and fmpr_sub using mpn level code, improving efficiency at low precision.
• Ported the partition function implementation from flint (using ball arithmetic in all steps of the calculation to guarantee correctness).
• Ported algorithm for computing the cosine minimal polynomial from flint (using ball arithmetic to guarantee correctness).
• Support using GMP instead of MPIR.
• Only use thread-local storage when enabled in flint.
• Slightly faster error bounding for the zeta function.
• Added some other helper functions.

11.2.17 2013-03-28 - version 0.5

• Arithmetic and elementary functions
  – Added fmpr_get_fmpz, fmpr_get_si.
  – Fixed accuracy problem with fmprb_div_2expm1.
  – Special-cased squaring of complex numbers.
  – Added various fmpcb convenience functions (addmul_ui, etc).
  – Optimized fmpr_cmp_2exp_si and fmpr_cmpabs_2exp_si, and added test code for comparison functions.
  – Added fmprb_atan2, also fixing a bug in fmpcb_arg.
  – Added fmprb_sin_pi, cos_pi, sin_cos_pi, etc.
  – Added fmprb_sin_pi_fmpq (etc.) using algebraic methods for fast evaluation of roots of unity.
  – Faster fmprb_poly_evaluate and evaluate_fmpcb using rectangular splitting.
  – Added fmprb_poly_evaluate2, evaluate2_fmpcb for simultaneously evaluating the derivative.
  – Added fmprb_poly root polishing code using near-optimal Newton steps (experimental).
  – Added fmpr_root, fmprb_root (currently based on MPFR).
  – Added fmpr_min, fmpr_max.
  – Added fmprb_set_interval_fmpr, fmprb_union.
  – Added fmpr_bits, fmprb_bits, fmpcb_bits for obtaining the mantissa width.
  – Added fmprb_hypot.
  – Added complex square roots.
  – Improved fmprb_log to slightly improve speed, and properly support huge arguments.
  – Fixed exp, cosh, sinh to work with huge arguments.
  – Added fmprb_expm1.
  – Fixed sin, cos, atan to work with huge arguments.
  – Improved fmprb_pow and fmpcb_pow, including automatic detection of small integer and half-integer exponents.
  – Added many more elementary functions: fmprb_tan/cot/tanh/coth, fmpcb_tan/cot, and pi versions.
  – Added fmprb_const_e, const_log2, const_log10, const_catalan.
  – Fixed ball containment/overlap checking to work operate efficiently and correctly with huge exponents.
  – Strengthened test code for many core operations.

• Special functions
  – Reorganized zeta function related code.
  – Faster evaluation of the Riemann zeta function via sieving.
  – Documented and improved efficiency of the zeta constant binary splitting code.
  – Calculate error bound in Borwein’s algorithm with fmprs instead of using doubles.
- Optimized divisions in zeta evaluation via the Euler product.
- Use functional equation for Riemann zeta function of a negative argument.
- Compute single Bernoulli numbers using ball arithmetic instead of relying on the floating-point code in flint.
- Initial code for evaluating the gamma function using its Taylor series.
- Much faster rising factorials at high precision, using difference polynomials.
- Much faster gamma function at high precision.
- Added complex gamma function, log gamma function, and other versions.
- Added fmprb_agm (real arithmetic-geometric mean).
- Added fmprb_gamma_fmpq, supporting rapid computation of gamma(p/q) for q = 1,2,3,4,6.
- Added real and complex digamma function.
- Fixed unnecessary recomputation of Bernoulli numbers.
- Optimized computation of Euler’s constant, and added proper error bounds.
- Avoid reliance on doubles in the hypergeometric series tail bound.
- Cleaned up factorials and binomials, computing factorials via gamma.
  
- Other
  - Added an fmpz_extras module to collect various internal fmpz helper functions.
  - Fixed detection of flint header files.
  - Fixed various other small bugs.

**11.2.18 2013-01-26 - version 0.4**

- Much faster fmpr_mul, fmprb_mul and set_round, resulting in general speed improvements.
- Code for computing the complex Hurwitz zeta function with derivatives.
- Fixed and documented error bounds for hypergeometric series.
- Better algorithm for series evaluation of the gamma function at a rational point.
- Much faster generation of Bernoulli numbers.
- Complex log, exp, pow, trigonometric functions (currently based on MPFR).
- Complex nth roots via Newton iteration.
- Added code for arithmetic on fmpcb_polys.
- Code for computing Khinchin’s constant.
- Code for rising factorials of polynomials or power series
- Faster sin_cos.
- Better div_2expm1.
- Many other new helper functions.
- Improved thread safety.
- More test code for core operations.
11.2.19 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 fnmpz_holonomic for functions/sequences defined by linear differential/difference equations with polynomial coefficients
  - Functions for creating various special sequences and functions.
  - Some closure properties for sequences.
  - Taylor series expansion for differential equations.
  - Computing the nth entry of a sequence using binary splitting.
  - Computing the nth entry mod p using fast multipoint evaluation.
- Generic binary splitting code with automatic error bounding is now used for evaluating hypergeometric series.
- Matrix powering.
- Various other helper functions.

11.2.20 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 fnmprb_mat module for matrices of balls of real numbers.
- Matrix multiplication.
- Interval-aware LU decomposition, solving, inverse and determinant.
- Many helper functions and small bugfixes.

11.2.21 2012-09-14 - version 0.1

- 2012-08-05 - Began simplified rewrite.
- 2012-04-05 - Experimental ball and polynomial code (first commit).
[WQ3b] http://functions.wolfram.com/07.12.27.0014.01

[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


of Computation 66 (1997) 417-431


personal/pborwein/PAPERS/P155.pdf


Euler_CARMA_10.pdf


[Dup2006] R. Dupont. “Moyenne arithmético-géométrique, suites de Borchardt et applications.” These
Labo/Regis.Dupont/these_soutenance.pdf


Special Functions (2004), 101-115.


Symbols

acb_dirichlet_euler_product_real_ui (C function), 149
acb_dirichlet_hardy_theta_series (C function), 150
acb_dirichlet_hardy_z_series (C function), 151
acb_dirichlet_l_series (C function), 150
acb_hypgeom_airy_series (C function), 123
acb_hypgeom_beta_lower_series (C function), 125
acb_hypgeom_chi_series (C function), 126
acb_hypgeom_ci_series (C function), 126
acb_hypgeom_erf_series (C function), 121
acb_hypgeom_erfc_series (C function), 121
acb_hypgeom_erfi_series (C function), 121
acb_hypgeom_fresnel_series (C function), 121
acb_hypgeom_gamma_lower_series (C function), 124
acb_hypgeom_gamma_upper_series (C function), 124
acb_hypgeom_li_series (C function), 126
acb_hypgeom_shi_series (C function), 126
acb_hypgeom_si_series (C function), 126
acb_mat_charpoly (C function), 114
acb_poly_add (C function), 91
acb_poly_derivative (C function), 96
acb_poly_digamma_series (C function), 100
acb_poly_div (C function), 92
acb_poly_div_root (C function), 92
acb_poly_divrem (C function), 92
acb_poly_elliptic_k_series (C function), 102
acb_poly_elliptic_p_series (C function), 102
acb_poly_erf_series (C function), 102
acb_poly_evaluate (C function), 94
acb_poly_evaluate2 (C function), 94
acb_poly_evaluate2_horner (C function), 94
acb_poly_evaluate2_rectangular (C function), 94
acb_poly_evaluate_horner (C function), 94
acb_poly_evaluate_rectangular (C function), 94
acb_poly_evaluate_vec_fast (C function), 95
acb_poly_evaluate_vec_fast_precomp (C function), 95
acb_poly_evaluate_vec_iter (C function), 95
acb_poly_exp_series (C function), 98
acb_poly_exp_series_basecase (C function), 98
acb_poly_find_roots (C function), 103
acb_poly_gamma_series (C function), 99
acb_poly_integral (C function), 96
acb_poly_interpolate_barycentric (C function), 95
acb_poly_interpolate_fast (C function), 95
acb_poly_interpolate_fast_precomp (C function), 95
acb_poly_interpolate_newton (C function), 95
acb_poly_interpolation_weights (C function), 95
acb_poly_inverse_borel_transform (C function), 96
acb_poly_inv_borel_transform (C function), 96
acb_poly_lgamma_series (C function), 100
acb_poly_majorant (C function), 90
acb_poly_mul (C function), 92
acb_poly_mullow (C function), 91
acb_poly_mullow_classical (C function), 91
acb_poly_mullow_transpose (C function), 91
acb_poly_mullow_transpose_gauss (C function), 91
acb_poly_normalise (C function), 88
acb_poly_overlaps (C function), 90
acb_poly_polylog_cpx (C function), 101

215
_arb_exp_taylor_naive (C function), 57
_arb_exp_taylor_rs (C function), 57
_arb_get_mpn_fixed_mod_log2 (C function), 58
_arb_get_mpn_fixed_mod_pi4 (C function), 58
_arb_hypgeom_airy_series (C function), 133
_arb_hypgeom_beta_lower_series (C function), 132
_arb_hypgeom_chi_series (C function), 133
_arb_hypgeom_ci_series (C function), 132
_arb_hypgeom_ei_series (C function), 132
_arb_hypgeom_erf_series (C function), 131
_arb_hypgeom_erfc_series (C function), 131
_arb_hypgeom_erfi_series (C function), 131
_arb_hypgeom_fresnel_series (C function), 131
_arb_hypgeom_gamma_lower_series (C function), 132
_arb_hypgeom_gamma_upper_series (C function), 131
_arb_hypgeom_li_series (C function), 133
_arb_hypgeom_shi_series (C function), 132
_arb_hypgeom_si_series (C function), 132

_arb_mat_charpoly (C function), 110
_arb_mat_cholesky_banachiewicz (C function), 108
_arb_mat_cholesky_banachiewicz (C function), 108
_arb_mat_ldl_golub_and_van_loan (C function), 109
_arb_mat_lldl_inplace (C function), 109

_arb_poly_acos_series (C function), 83
_arb_poly_add (C function), 76
_arb_poly_asin_series (C function), 83
_arb_poly_atan_series (C function), 83
_arb_poly_binomial_transform (C function), 82
_arb_poly_binomial_transform_basecase (C function), 82
_arb_poly_binomial_transform_convolution (C function), 82

_arb_poly_borel_transform (C function), 81
_arb_poly_compose (C function), 78
_arb_poly_compose_divconquer (C function), 78
_arb_poly_compose_horner (C function), 78
_arb_poly_compose_series (C function), 78
_arb_poly_compose_series_brent_kung (C function), 78

_arb_poly_cos_pi_series (C function), 85
_arb_poly_cos_series (C function), 84
_arb_poly_cosh_series (C function), 85
_arb_poly_derivative (C function), 81
_arb_poly_digamma_series (C function), 86
_arb_poly_div (C function), 77
_arb_poly_div_root (C function), 77

Index 217
acb_calc_func_t (C type), 160
acb_calc_integrate_taylor (C function), 160
acb_chebyshev_t2_ui (C function), 70
acb_chebyshev_t_ui (C function), 70
acb_chebyshev_u2_ui (C function), 70
acb_chebyshev_u_ui (C function), 70
acb_clear (C function), 60
acb_conj (C function), 64
acb_conjugate (C function), 64
acb_const_pi (C function), 66
acb_contains (C function), 63
acb_contains_fmpq (C function), 63
acb_contains_fmpz (C function), 63
acb_contains_int (C function), 63
acb_contains_zero (C function), 63
acb_cos (C function), 67
acb_cosh (C function), 67
acb_cos Pi (C function), 67
acb_cot (C function), 67
acb_coth (C function), 67
acb_csgn (C function), 64
acb_cube (C function), 65
acb_digamma (C function), 68
acb_dirichler_hurwitz_precomp_choose_param (C function), 147
acb_dirichlet_chi (C function), 147
acb_dirichlet_chi_theta_arb (C function), 149
acb_dirichlet_chirvec (C function), 147
acb_dirichlet_eta (C function), 146
acb_dirichlet_gauss_sum (C function), 147
acb_dirichlet_gauss_sum_factor (C function), 147
acb_dirichlet_gauss_sum_naive (C function), 147
acb_dirichlet_gauss_sum_order2 (C function), 147
acb_dirichlet_gauss_sum_theta (C function), 147
acb_dirichlet_gauss_sum ui (C function), 148
acb_dirichlet_hardy_theta (C function), 150
acb_dirichlet_hardy_theta_series (C function), 150
acb_dirichlet_hardy_z (C function), 150
acb_dirichlet_hardy_z_series (C function), 151
acb_dirichlet_hurwitz_precomp_bound (C function), 147
acb_dirichlet_hurwitz_precomp_clear (C function), 147
acb_dirichlet_hurwitz_precomp_eval (C function), 147
acb_dirichlet_hurwitz_precomp_init (C function), 146
acb_dirichlet_hurwitz_precomp_struct (C type), 146
acb_dirichlet_hurwitz_precomp_t (C type), 146
acb_dirichlet_jacobi_sum (C function), 148
acb_dirichlet_jacobi_sum_factor (C function), 148
acb_dirichlet_jacobi_sum_gauss (C function), 148
acb_dirichlet_jacobi_sum_naive (C function), 148
acb_dirichlet_jacobi_sum_ui (C function), 148
acb_dirichlet_1 (C function), 150
acb_dirichlet_1_euler_product (C function), 149
acb_dirichlet_1_hurwitz (C function), 149
acb_dirichlet_1_jet (C function), 150
acb_dirichlet_1_series (C function), 150
acb_dirichlet_pairing (C function), 147
acb_dirichlet_pairing_char (C function), 147
acb_dirichlet_powsum_sieved (C function), 145
acb_dirichlet_powsum_smooth (C function), 145
acb_dirichlet_powsum_term (C function), 145
acb_dirichlet_root (C function), 145
acb_dirichlet_root_number (C function), 149
acb_dirichlet_root_number_theta (C function), 149
acb_dirichlet_root_number_theta (C function), 149
acb_dirichlet_roots_clear (C function), 145
acb_dirichlet_roots_init (C function), 145
acb_dirichlet_roots_struct (C type), 145
acb_dirichlet_root_t (C type), 145
acb_dirichlet_ui_theta_arb (C function), 149
acb_dirichlet_zeta (C function), 146
acb_dirichlet_zeta_bound (C function), 146
acb_dirichlet_zeta RS (C function), 146
acb_dirichlet_zeta RS_bound (C function), 146
acb_dirichlet_zeta_RS_d_coeffs (C function), 146
acb_dirichlet_zeta_RS_f_coeffs (C function), 146
acb_dirichlet_zeta_RS_r (C function), 146
acb_div (C function), 65
acb_div_arb (C function), 65
acb_div_fmpz (C function), 65
acb_div_oneui (C function), 65
acb_div_ui (C function), 65
acb_div_ui (C function), 65
acb_div_ui (C function), 65
acb_eq (C function), 63
acb_equal (C function), 62
acb_equal_si (C function), 63
acb_equal_si (C function), 63
acb_exp (C function), 66
acb_exp_invexp (C function), 66
acb_exp_pi_i (C function), 66
acb_fprintf (C function), 62
acb_fprintf (C function), 62
acb_fprintf (C function), 62
acb_get_abs_lbound_arf (C function), 63
acb_get_abs_ubound_arf (C function), 63
acb_get_abs_ubound_arf (C function), 63
acb_get_imag (C function), 64
acb_get_imag (C function), 64
acb_get_imag (C function), 64
acb_get_mag (C function), 63
acb_get_mag (C function), 63
acb_get_mag (C function), 63
acb_get_mag_lower (C function), 63
acb_get_rad_ubound_arf (C function), 63
acb_get_rad_ubound_arf (C function), 63
acb_get_real (C function), 64
acb_get_real (C function), 64
acb_get_unique_fmpz (C function), 69
acb_hurwitz_zeta (C function), 69
acb_hypgeom_0f1 (C function), 120
acb_hypgeom_0f1_asym (C function), 120
acb_hypgeom_0f1_direct (C function), 120
acb_hypgeom_1f1 (C function), 120

Index 219
acb_hypgeom_2f1 (C function), 127
acb_hypgeom_2f1_choose (C function), 127
acb_hypgeom_2f1_continuation (C function), 127
acb_hypgeom_2f1_corner (C function), 127
acb_hypgeom_2f1_direct (C function), 127
acb_hypgeom_2f1_series_direct (C function), 127
acb_hypgeom_2f1_transform (C function), 127
acb_hypgeom_2f1_transform_limit (C function), 127
acb_hypgeom_airy (C function), 123
acb_hypgeom_airy_asymp (C function), 123
acb_hypgeom_airy_bound (C function), 123
acb_hypgeom_airy_direct (C function), 123
acb_hypgeom_airy_jet (C function), 123
acb_hypgeom_airy_series (C function), 123
acb_hypgeom_bessel_i (C function), 122
acb_hypgeom_bessel_i_0f1 (C function), 122
acb_hypgeom_bessel_i_asymp (C function), 122
acb_hypgeom_bessel_j (C function), 122
acb_hypgeom_bessel_j_0f1 (C function), 122
acb_hypgeom_bessel_j_0f1_series (C function), 122
acb_hypgeom_bessel_j_asymp (C function), 122
acb_hypgeom_bessel_k (C function), 122
acb_hypgeom_bessel_k_0f1 (C function), 122
acb_hypgeom_bessel_k_0f1_series (C function), 122
acb_hypgeom_bessel_k_asymp (C function), 122
acb_hypgeom_bessel_y (C function), 122
acb_hypgeom_beta_lower (C function), 125
acb_hypgeom_beta_lower_series (C function), 125
acb_hypgeom_beta_lower_singular (C function), 125
acb_hypgeom_chebyshev_t (C function), 128
acb_hypgeom_chebyshev_u (C function), 128
acb_hypgeom_chi (C function), 126
acb_hypgeom_chi_2f3 (C function), 126
acb_hypgeom_chi_asymp (C function), 126
acb_hypgeom_chi_series (C function), 126
acb_hypgeom_ci (C function), 126
acb_hypgeom_ci_2f3 (C function), 126
acb_hypgeom_ci_asymp (C function), 126
acb_hypgeom_ci_series (C function), 126
acb_hypgeom_e (C function), 125
acb_hypgeom_e_2f2 (C function), 125
acb_hypgeom_e_0f1 (C function), 125
acb_hypgeom_e_0f1_series (C function), 125
acb_hypgeom_e_asymp (C function), 125
acb_hypgeom_erf (C function), 120
acb_hypgeom_erf_1f1a (C function), 120
acb_hypgeom_erf_1f1b (C function), 120
acb_hypgeom_erf_asymp (C function), 120
acb_hypgeom_erf_propagated_error (C function), 120
acb_hypgeom_erf_series (C function), 121
acb_hypgeom_erfc (C function), 121
acb_hypgeom_erfc_series (C function), 121
acb_hypgeom_erfi (C function), 121
acb_hypgeom_erfi_series (C function), 121
acb_hypgeom_expint (C function), 125
acb_hypgeom_fresnel (C function), 121
acb_hypgeom_fresnel_series (C function), 121
acb_hypgeom_gamma_lower (C function), 124
acb_hypgeom_gamma_lower_series (C function), 124
acb_hypgeom_gamma_upper (C function), 124
acb_hypgeom_gamma_upper_1f1a (C function), 124
acb_hypgeom_gamma_upper_1f1b (C function), 124
acb_hypgeom_gamma_upper_asympt (C function), 124
acb_hypgeom_gamma_upper_series (C function), 124
acb_hypgeom_gamma_upper_singular (C function), 124
acb_hypgeom_gegenbauer_c (C function), 128
acb_hypgeom_hermite_h (C function), 129
acb_hypgeom_jacobi_p (C function), 128
acb_hypgeom_krigeure_r (C function), 128
acb_hypgeom_legendre_p (C function), 129
acb_hypgeom_legendre_p_1f1 (C function), 129
acb_hypgeom_legendre_p_0f1_0f1 (C function), 129
acb_hypgeom_li (C function), 126
acb_hypgeom_li_series (C function), 127
acb_hypgeom_m (C function), 120
acb_hypgeom_m_1f1 (C function), 120
acb_hypgeom_m_asymp (C function), 120
acb_hypgeom_pfq (C function), 119
acb_hypgeom_pfq_bound_factor (C function), 117
acb_hypgeom_pfq_choose_n (C function), 117
acb_hypgeom_pfq_direct (C function), 118
acb_hypgeom_pfq_series_direct (C function), 119
acb_hypgeom_pfq_series_sum (C function), 118
acb_hypgeom_pfq_series_sum_bs (C function), 118
acb_hypgeom_pfq_series_sum_forward (C function), 118
acb_hypgeom_pfq_series_sum_rs (C function), 118
acb_hypgeom_pfq_sum (C function), 118
acb_hypgeom_pfq_sum_bs (C function), 118
acb_hypgeom_pfq_sum_bs_invz (C function), 118
acb_hypgeom_pfq_sum_fme (C function), 118
acb_hypgeom_pfq_sum_invz (C function), 118
acb_hypgeom_pfq_sum_rs (C function), 118
acb_hypgeom_pfq_sum_forward (C function), 117
acb_hypgeom_pfq_sum_inz (C function), 118
acb_hypgeom_pfq_sum_rs (C function), 118
acb_hypgeom_shi (C function), 126
acb_hypgeom_shi_series (C function), 126
acb_hypgeom_si (C function), 125
acb_hypgeom_si_1f2 (C function), 125
acb_hypgeom_si_asymp (C function), 125
acb_hypgeom_si_series (C function), 126
acb_hypgeom_spherical_y (C function), 129
acb_modular_theta_notransform (C function), 139
acb_modular_theta_sum (C function), 137
acb_modular_theta_transform (C function), 136
acb_modular_transform (C function), 135
acb_mul (C function), 65
acb_mul_2exp_fmpz (C function), 65
acb_mul_2exp_si (C function), 65
acb_mul_arb (C function), 65
acb_mul_fmpz (C function), 65
acb_mul_onei (C function), 64
acb_mul_si (C function), 65
acb_mul_ui (C function), 65
acb_ne (C function), 63
acb_neg (C function), 64
acb_one (C function), 61
acb_onei (C function), 61
acb_overlaps (C function), 63
acb_poly_add (C function), 91
acb_poly_add_series (C function), 91
acb_poly_add_si (C function), 91
acb_poly_agm1_series (C function), 102
acb_poly_allocated_bytes (C function), 88
acb_poly_atan_series (C function), 97
acb_poly_binomial_transform (C function), 96
acb_poly_binomial_transform_basecase (C function), 96
acb_poly_binomial_transform_convolution (C function), 96
acb_poly_borel_transform (C function), 96
acb_poly_clear (C function), 88
acb_poly_compose (C function), 93
acb_poly_compose_divconquer (C function), 93
acb_poly_compose_horner (C function), 93
acb_poly_compose_series (C function), 93
acb_poly_compose_series_brent_kung (C function), 93
acb_poly_compose_series_horner (C function), 93
acb_poly_complexify (C function), 89
acb_poly_complexify2 (C function), 91
acb_poly_compose (C function), 93
acb_polycontains (C function), 90
acb_polycontains_fmpq_poly (C function), 90
acb_polycos_pi_series (C function), 99
acb_polycos_series (C function), 98
acb_polycosh_series (C function), 99
acb_polycot_pi_series (C function), 99
acb_polydegree (C function), 88
acb_polyderivative (C function), 96
acb_polydigamma_series (C function), 100
acb_polydiv_series (C function), 92
acb_polydivrem (C function), 92
acb_polyelliptic_k_series (C function), 102
acb_polyelliptic_p_series (C function), 102
acb_polyequal (C function), 90
acb_polyerf_series (C function), 102
acb_polyevaluate (C function), 94
acb_polyevaluate2 (C function), 94
acb_polyevaluate2_horner (C function), 94
acb_polyevaluate_rectangular (C function), 94
acb_polyevaluate_vec_fast (C function), 94
acb_polyevaluate_vec_iter (C function), 95
acb_polyexp_series (C function), 98
acb_polyexp_series_basecase (C function), 98
acb_polyfind_series (C function), 103
acb_polyfit_length (C function), 88
acb_polyfprintd (C function), 89
acb_polygamma_series (C function), 99
acb_polyget_coeff_acb (C function), 89
acb_polyget_coeff_ptr (C function), 90
acb_polyget_unique_fmpz_poly (C function), 90
acb_polyinit (C function), 88
acb_polyintegral (C function), 96
acb_polyinterpolate_barycentric (C function), 95
acb_polyinterpolate_fast (C function), 95
acb_polyinterpolate_newton (C function), 95
acb_polyinv_borel_transform (C function), 96
acb_polyinv_series (C function), 92
acb_polyis_one (C function), 89
acb_polyis_real (C function), 90
acb_polyis_x (C function), 89
acb_polyis_zero (C function), 89
acb_polylength (C function), 88
acb_polylgamma_series (C function), 100
acb_polylog_series (C function), 97
acb_polymajorant (C function), 90
acb_plynmul (C function), 92
acb_plynmullow (C function), 92
acb_plynmullow_classical (C function), 91
acb_plynmullow_transpose (C function), 92
acb_plynmullow_transpose_gauss (C function), 92
acb_poly_neg (C function), 91
acb_polyone (C function), 89
acb_polyoverlaps (C function), 90
acb_polypolylog_series (C function), 102
acb_polypow_acb_series (C function), 97
acb_polypow_series (C function), 97
acb_polypow_ui (C function), 96
acb_polypow_ui_trunc_bineexp (C function), 96
acb_polyprintf (C function), 89
acb_polyproduct_roots (C function), 94
acb_polyrandtest (C function), 90
acb_polyrevert_series (C function), 94
acb_polyrevert_series_lagrange (C function), 93
acb_polyrevert_series_lagrange_fast (C function), 94
acb_polyrevert_series_newton (C function), 94
acb_plyrgamma_series (C function), 99
acb_polyrising_ui_series (C function), 100
acb_polyroot_bound_fujiwara (C function), 102
acb_polyrsqrt_series (C function), 97
acb_poly_scalar_div (C function), 91
acb_submul_fmpz (C function), 65
acb_submul_si (C function), 65
acb_submul_ui (C function), 65
acb_swap (C function), 61
acb_t (C type), 60
acb_tan (C function), 67
acb_tan_pi (C function), 67
acb_trim (C function), 64
acb_unit_root (C function), 66
acb_zero (C function), 61
acb_zeta (C function), 69
acb_tan (C function), 67
acb_tanh (C function), 67
acb_trim (C function), 64
acb_unit_root (C function), 66
acb_zero (C function), 61
acb_zeta (C function), 69
acb_tan (C function), 67
acb_tan_pi (C function), 67
acb_trim (C function), 64
acb_unit_root (C function), 66
acb_zero (C function), 61
acb_zeta (C function), 69
acb_tan (C function), 67
acb_tanh (C function), 67
arb_mat_bound_frobenius_norm (C function), 107
arb_mat_bound_inf_norm (C function), 107
arb_mat_charpoly (C function), 110
arb_mat_cho (C function), 108
arb_mat_contains (C function), 106
arb_mat_contains_fmpq_mat (C function), 106
arb_mat_count_is_zero (C function), 110
arb_mat_count_not_is_zero (C function), 110
arb_mat_det (C function), 108
arb_mat_entry (C macro), 105
arb_mat_entrywise_is_zero (C function), 110
arb_mat_entrywise_not_is_zero (C function), 110
arb_mat_eq (C function), 106
arb_mat_equal (C function), 106
arb_mat_exp (C function), 110
arb_mat_exp_taylor_sum (C function), 110
arb_mat_fprintd (C function), 106
arb_mat_frobenius_norm (C function), 107
arb_mat_init (C function), 105
arb_mat_inv (C function), 108
arb_mat_inv_cho_precomp (C function), 109
arb_mat_inv_ldl_precomp (C function), 109
arb_mat_is_empty (C function), 106
arb_mat_is_square (C function), 106
arb_mat_ldl (C function), 109
arb_mat_lu (C function), 108
arb_mat_mul (C function), 107
arb_mat_mul_classical (C function), 107
arb_mat_mul_entrywise (C function), 107
arb_mat_mul_threaded (C function), 107
arb_mat_ncols (C macro), 105
arb_mat_ne (C function), 106
arb_mat_neg (C function), 107
arb_mat_nrows (C macro), 105
arb_mat_one (C function), 106
arb_mat_overlaps (C function), 106
arb_mat_pow_ui (C function), 107
arb_mat_printd (C function), 106
arb_mat_randtest (C function), 106
arb_mat_scalar_addmul_arb (C function), 107
arb_mat_scalar_addmul_fmpz (C function), 107
arb_mat_scalar_addmul_si (C function), 107
arb_mat_scalar_div_arb (C function), 108
arb_mat_scalar_div_fmpz (C function), 108
arb_mat_scalar_div_si (C function), 108
arb_mat_scalar_mul_2exp_si (C function), 107
arb_mat_scalar_mul_arb (C function), 108
arb_mat_scalar_mul_fmpz (C function), 108
arb_mat_scalar_mul_si (C function), 108
arb_mat_set (C function), 106
arb_mat_set_fmpq_mat (C function), 106
arb_mat_set_fmpz_mat (C function), 106
arb_mat_set_round_fmpz_mat (C function), 106
arb_mat_solve (C function), 108
arb_mat_solve_cho_precomp (C function), 109
arb_mat_solve_ldl_precomp (C function), 109
arb_mat_solve_lu_precomp (C function), 109
arb_mat_sdp_inv (C function), 109
arb_mat_sdp_solve (C function), 109
arb_mat_sq (C function), 107
arb_mat_sq_classical (C function), 107
arb_mat_struct (C type), 105
arb_mat_sub (C function), 107
arb_mat_t (C type), 105
arb_mat_trace (C function), 107
arb_mat_transpose (C function), 107
arb_mat_zero (C function), 106
arb_max (C function), 48
arb_midref (C macro), 42
arb_min (C function), 48
arb_mul (C function), 49
arb_mul_2exp_fmpz (C function), 49
arb_mul_2exp_si (C function), 49
arb_mul_arf (C function), 49
arb_mul_fmpz (C function), 49
arb_mul_si (C function), 49
arb_mul_ui (C function), 49
arb_ne (C function), 48
arb_neg (C function), 48
arb_neg_inf (C function), 44
arb_neg_round (C function), 48
arb_one (C function), 43
arb_overlaps (C function), 47
arb_partitions_fmpz (C function), 57
arb_partitions_ui (C function), 57
arb_poly_acos_series (C function), 83
arb_poly_add (C function), 76
arb_poly_add_series (C function), 76
arb_poly_allocated_bytes (C function), 73
arb_poly_compose (C function), 78
arb_poly_compose_divconquer (C function), 78
arb_poly_compose_horner (C function), 78
arb_poly_compose_series (C function), 78
arb_poly_compose_series_brent_kung (C function), 78
arb_poly_binomial_transform (C function), 82
arb_poly_binomial_transform_basecase (C function), 82
arb_poly_binomial_transform_convolution (C function), 82
arb_poly_borel_transform (C function), 81
arb_poly_clear (C function), 73
arb_poly_contains (C function), 75
arb_poly_contains_fmpq_poly (C function), 75
arb_poly_contains_fmpz_poly (C function), 75
arb_poly_cos_pi_series (C function), 84
arb_poly_taylor_shift_divconquer (C function), 78
arb_poly_taylor_shift_horner (C function), 78
arb_poly_truncate (C function), 74
arb_poly_valuation (C function), 74
arb_poly_zero (C function), 74
arb_poly_zeta_series (C function), 86
arb_polylog (C function), 56
arb_polylog_si (C function), 56
arb_pos_inf (C function), 44
arb_pow (C function), 51
arb_pow_fmpq (C function), 51
arb_pow_fmpz (C function), 51
arb_pow_fmpz_binexp (C function), 51
arb_pow_ui (C function), 51
arb_power_sum_vec (C function), 56
arb_print (C function), 44
arb_printd (C function), 44
arb_printn (C function), 44
arb_ptr (C type), 42
arb_radref (C macro), 42
arb_randtest (C function), 44
arb_randtest_exact (C function), 44
arb_randtest_precise (C function), 44
arb_randtest_special (C function), 44
arb_randtest_wide (C function), 44
arb_rel_accuracy_bits (C function), 46
arb_rel_error_bits (C function), 46
arb_rgamma (C function), 54
arb_rising (C function), 54
arb_rising2_ui (C function), 54
arb_rising2_ui_bs (C function), 54
arb_rising2_ui_rs (C function), 54
arb_rising_fmpq_ui (C function), 54
arb_rising_ui (C function), 54
arb_rising_ui_bs (C function), 54
arb_rising_ui_rec (C function), 54
arb_rising_ui_rs (C function), 54
arb_root (C function), 50
arb_root_ui (C function), 50
arb_rsqrt (C function), 50
arb_rsqrt_ui (C function), 50
arb_set (C function), 43
arb_set_arf (C function), 43
arb_set_d (C function), 43
arb_set_fmpq (C function), 43
arb_set_fmpz (C function), 43
arb_set_fmpz_2exp (C function), 43
arb_set_round (C function), 43
arb_set_xml (C function), 43
arb_set_si (C function), 43
arb_set_str (C function), 43
arb_set_ui (C function), 43
arb_sgn (C function), 48
arb_si_pow_ui (C function), 51
arb_sin (C function), 52
arb_sin_cos (C function), 52
arb_sin_cos_pi (C function), 52
arb_sin_cos_pi_fmpq (C function), 52
arb_sin_pi (C function), 52
arb_sin_pi_fmpq (C function), 52
arb_sinc (C function), 52
arb_sinh (C function), 53
arb_sinh_cosh (C function), 53
arb_sqrt (C function), 50
arb_sqrt1pm1 (C function), 50
arb_sqrt_arf (C function), 50
arb_sqrt_fmpz (C function), 50
arb_sqrt_ui (C function), 50
arb_sqrtpos (C function), 50
arb_strcptr (C type), 42
arb_union (C function), 45
arb_zero (C function), 43
arb_zero_pm_inf (C function), 44
arb_zeta (C function), 55
arb_zeta_ui (C function), 55
arb_zeta_ui_asympt (C function), 55
arb_zeta_ui_bernoulli (C function), 55
arb_zeta_ui_borwein_bsplit (C function), 55
arb_zeta_ui_euler_product (C function), 55
arb_zeta_ui_vec (C function), 55
arb_zeta_ui_vec_borwein (C function), 55
arb_zeta_ui_vec_even (C function), 55
arf_abs (C function), 38
arf_abs_bound_le_2exp_fmpz (C function), 36
arf_abs_bound_less_2exp_fmpz (C function), 36
arf_abs_bound_less_2exp_si (C function), 36
arf_add (C function), 38
arf_add_fmpz (C function), 38
arf_add_fmpz_2exp (C function), 38
arf_add_si (C function), 38
arf_add_ui (C function), 38
arf_addmul (C function), 38
arf_addmul_fmpz (C function), 38
arf_addmul_mpz (C function), 38
arf_addmul_si (C function), 38
arf_addmul_ui (C function), 38
arf_allocated_bytes (C function), 33
arf_bits (C function), 36
arf_clear (C function), 33
arf_cmp (C function), 36
arf_cmp_2exp_si (C function), 36
arf_cmp_d (C function), 36
arf_cmp_si (C function), 36
arf_cmp_ui (C function), 36
arf_cmpabs (C function), 36
arf_cmpabs_2exp_si (C function), 36
arf_cmpabs_d (C function), 36
arf_cmpabs_mag (C function), 36
arf_cmpabs_ui (C function), 36
arf_complex_mul (C function), 40
arf_complex_mul_fallback (C function), 40
arf_complex_sqr (C function), 40
arf_debug (C function), 37
arf_div (C function), 39
arf_div_fmpz (C function), 39
arf_div_si (C function), 39
arf_div_ui (C function), 39
arf_equal (C function), 36
arf_equal_si (C function), 36
arf_floor (C function), 35
arf_fmpz_div (C function), 39
arf_fmpz_div_fmpz (C function), 39
arf_fprint (C function), 37
arf_fprintf (C function), 38
arf_frexp (C function), 35
arf_get_d (C function), 35
arf_get_fmpr (C function), 35
arf_get_fmpz (C function), 35
arf_get_fmpz_2exp (C function), 35
arf_get_fmpz_fixed_fmpz (C function), 35
arf_get_fmpz_fixed_si (C function), 35
arf_get_mag (C function), 36
arf_get_mag_lower (C function), 36
arf_get_mpfr (C function), 35
arf_get_si (C function), 35
arf_init (C function), 33
arf_init_neg_mag_shallow (C function), 37
arf_init_neg_shallow (C function), 37
arf_init_set_mag_shallow (C function), 37
arf_init_set_shallow (C function), 37
arf_init_set_si (C function), 34
arf_init_set_ui (C function), 34
arf_interval_clear (C function), 158
arf_interval_get_arb (C function), 158
arf_interval_srcptr (C function), 158
arf_interval_set (C function), 158
arf_interval_srcptr (C type), 158
arf_interval_struct (C type), 158
arf_interval_swap (C function), 158
arf_interval_t (C type), 158
arf_is_finite (C function), 34
arf_is_inf (C function), 34
arf_is_int (C function), 36
arf_is_int_2exp_si (C function), 36
arf_is_nan (C function), 34
arf_is_neg_inf (C function), 34
arf_is_normal (C function), 34
arf_is_one (C function), 34
arf_is_pos_inf (C function), 34
arf_is_special (C function), 34
arf_is_zero (C function), 34
arf_mag_add_ulp (C function), 37
arf_mag_fast_add_ulp (C function), 37
arf_mag_set_ulp (C function), 37
arf_max (C function), 36
arf_min (C function), 36
arf_mul (C function), 38
arf_mul_2exp_fmpz (C function), 38
arf_mul_2exp_si (C function), 38
arf_mul_fmpz (C function), 38
arf_mul_mpz (C function), 38
arf_mul_si (C function), 38
arf_mul_ui (C function), 38
arf_nan (C function), 34
arf_neg (C function), 38
arf_neg_inf (C function), 33
arf_neg_round (C function), 38
arf_one (C function), 33
arf_pos_inf (C function), 33
ARF_PREC_EXACT (C macro), 33
arf_print (C function), 37
arf_printd (C function), 37
arf_randtest (C function), 37
arf_randtest_not_zero (C function), 37
arf_randtest_special (C function), 37
ARF_RND_CEIL (C macro), 33
ARF_RND_DOWN (C macro), 33
ARF_RND_FLOOR (C macro), 33
ARF_RND_NEAR (C macro), 33
ARF_RND_UP (C macro), 33
arf_root (C function), 39
arf_rsqrt (C function), 39
arf_set (C function), 34
arf_set_d (C function), 34
arf_set_fmpr (C function), 34
arf_set_fmpz (C function), 34
arf_set_fmpz_2exp (C function), 35
arf_set_mag (C function), 37
arf_set_mpfr (C function), 34
arf_set_mpz (C function), 34
arf_set_round (C function), 34
arf_set_round_fmpz (C function), 34
Index

arl_set_round_fmpz_2exp (C function), 35
arl_set_round_mpz (C function), 34
arl_set_round_si (C function), 34
arl_set_si (C function), 34
arl_set_si_2exp_si (C function), 34
arl_set_ui (C function), 34
arl_set_ui_2exp_si (C function), 34
arl_sgn (C function), 36
arl_si_div (C function), 39
arl_sqrt (C function), 39
arl_sqrt_fmpz (C function), 39
arl_sqrt_ui (C function), 39
arl_sub (C function), 38
arl_sub_fmpz (C function), 38
arl_sub_si (C function), 38
arl_sub_ui (C function), 38
arl_submul (C function), 39
arl_submul_fmpz (C function), 39
arl_submul_mpz (C function), 39
arl_submul_si (C function), 39
arl_submul_ui (C function), 39
arl_sum (C function), 39
arl_swap (C function), 34
arl_t (C type), 32
arl_sub (C function), 38
arl_sub_fmpz (C function), 38
arl_sub_si (C function), 38
arl_sub_ui (C function), 38
arl_submul (C function), 39
arl_submul_fmpz (C function), 39
arl_submul_mpz (C function), 39
arl_submul_si (C function), 39
arl_submul_ui (C function), 39
arl_sum (C function), 39
arl_swap (C function), 34
arl_t (C type), 32
arl_sub (C function), 38
arl_sub_fmpz (C function), 38
arl_sub_si (C function), 38
arl_sub_ui (C function), 38
arl_submul (C function), 39
arl_submul_fmpz (C function), 39
arl_submul_mpz (C function), 39
arl_submul_si (C function), 39
arl_submul_ui (C function), 39
arl_sum (C function), 39
arl_swap (C function), 34
arl_t (C type), 32
arniui_div (C function), 39
arl_zero (C function), 33

B
bernoulli_bound_2exp_si (C function), 152
bernoulli_cache (C variable), 151
bernoulli_cache_compute (C function), 151
bernoulli_cache_num (C variable), 151
bernoulli_fmpz_ui (C function), 152
bernoulli_rev_clear (C function), 151
bernoulli_rev_init (C function), 151
bernoulli_rev_next (C function), 151
bernoulli_rev_t (C type), 151
bool_mat_add (C function), 166
bool_mat_all (C function), 165
bool_mat_all_pairs_longest_walk (C function), 167
bool_mat_any (C function), 165
bool_mat_clear (C function), 165
bool_mat_complement (C function), 166
bool_mat_directed_cycle (C function), 166
bool_mat_directed_path (C function), 166
bool_mat_equal (C function), 165
bool_mat_fprint (C function), 165
bool_mat_get_entry (C function), 164
bool_mat_get_strongly_connected_components (C function), 167
bool_mat_init (C function), 165
bool_mat_is_diagonal (C function), 165
bool_mat_is_empty (C function), 165
bool_mat_is_lower_triangular (C function), 165
bool_mat_is_nilpotent (C function), 165
bool_mat_is_square (C macro), 165
bool_mat_is_transitive (C function), 165
bool_mat_mul (C function), 166
bool_mat_mul_entrywise (C function), 166
bool_mat_nilpotency_degree (C function), 166
bool_mat_nrows (C macro), 164
bool_mat_one (C function), 166
bool_mat_pow_ui (C function), 166
bool_mat_print (C function), 165
bool_mat_randtest (C function), 166
bool_mat_randtest_diagonal (C function), 166
bool_mat_randtest_nilpotent (C function), 166
bool_mat_set (C function), 165
bool_mat_set_entry (C function), 164
bool_mat_sqr (C macro), 166
bool_mat_struct (C type), 164
bool_mat_t (C type), 164
bool_mat_trace (C function), 166
bool_mat_transpose (C function), 166
bool_mat_transitive_closure (C function), 167
bool_mat_zero (C function), 166

D
dirichlet_char_clear (C function), 142
dirichlet_char_eq (C function), 143
dirichlet_char_eq_deep (C function), 143
dirichlet_char_exp (C function), 142
dirichlet_char_first_primitive (C function), 142
dirichlet_char_index (C function), 143
dirichlet_char_init (C function), 142
dirichlet_char_is_primitive (C function), 143
dirichlet_char_is_principal (C function), 143
dirichlet_char_is_real (C function), 143
dirichlet_char_lifting (C function), 144
dirichlet_char_log (C function), 142
dirichlet_char_lower (C function), 144
dirichlet_char_mul (C function), 144
dirichlet_char_next (C function), 143
dirichlet_char_next_primitive (C function), 143
dirichlet_char_one (C function), 142
dirichlet_char_pow (C function), 144
dirichlet_char_print (C function), 142
dirichlet_char_set (C function), 143
dirichlet_char_struct (C type), 142
dirichlet_char_t (C type), 142
dirichlet_chi (C function), 144
dirichlet_chi_vec (C function), 144
dirichlet_chi_vec_order (C function), 144
dirichlet_conductor_char (C function), 143
dirichlet_conductor_ui (C function), 143
dirichlet_group_clear (C function), 142
dirichlet_group_dlog_clear (C function), 142
dirichlet_group_dlog_precompute (C function), 142
dirichlet_group_init (C function), 142
dirichlet_group_num_primitive (C function), 142
dirichlet_group_size (C function), 142

230
dirichlet_group_struct (C type), 141
dirichlet_group_t (C type), 141
dirichlet_index_char (C function), 143
dirichlet_order_char (C function), 143
dirichlet_order_ui (C function), 143
dirichlet_pairing (C function), 144
dirichlet_pairing_char (C function), 144
dirichlet_parity_char (C function), 143
dirichlet_parity_ui (C function), 143
dirichlet_subgroup_init (C function), 142
dlog bsgs (C function), 169
dlog bsgs_clear (C function), 169
dlog bsgs_init (C function), 169
dlog bsgs struct (C type), 169
dlog bsgs_t (C type), 169
dlog crt (C function), 170
dlog crt_clear (C function), 170
dlog crt_init (C function), 170
dlog crt struct (C type), 170
dlog crt_t (C type), 170
dlog modpe (C function), 170
dlog modpe_clear (C function), 170
dlog modpe_init (C function), 170
dlog modpe struct (C type), 170
dlog modpe_t (C type), 170
DLOGNONE (C macro), 167
dlog once (C function), 168
dlog power (C function), 170
dlog power_clear (C function), 170
dlog power_init (C function), 170
dlog power struct (C type), 170
dlog power_t (C type), 170
dlog precomp (C function), 168
dlog precomp_clear (C function), 168
dlog precomp_modpe_init (C function), 168
dlog precomp_n_init (C function), 168
dlog precomp_p_init (C function), 168
dlog precomp_pe_init (C function), 168
dlog precomp_small_init (C function), 168
dlog precomp struct (C type), 167
dlog precomp_t (C type), 167
dlog rho (C function), 171
dlog rho_clear (C function), 171
dlog rho_init (C function), 171
dlog rho struct (C type), 171
dlog rho_t (C type), 171
dlog table (C function), 169
dlog table (C type), 169
dlog table_clear (C function), 169
dlog table_init (C function), 169
dlog table_t (C type), 169
dlog vec (C function), 168
dlog vec_add (C function), 168
dlog vec_eratos (C function), 168
dlog vec_eratos_add (C function), 169
dlog vec_fill (C function), 168
dlog vec_loop (C function), 168
dlog vec_loop_add (C function), 168
dlog_vec_set_not_found (C function), 168
dlog_vec_sieve (C function), 169
dlog_vec_sieve_add (C function), 169
F
fmpz_mat_t (C type), 14
fmpz_poly_t (C type), 14
fmpz_t (C type), 14
fmpfr_abs (C function), 176
fmpfr_abs_bound_le_2exp_fmpz (C function), 175
fmpfr_abs_bound_lt_2exp_fmpz (C function), 175
fmpfr_abs_bound_lt_2exp_si (C function), 175
fmpfr_add (C function), 176
fmpfr_add_error_result (C function), 173
fmpfr_add_fmpz (C function), 176
fmpfr_add_si (C function), 176
fmpfr_add_ui (C function), 176
fmpfr_addmul (C function), 177
fmpfr_addmul_fmpz (C function), 177
fmpfr_addmul_si (C function), 177
fmpfr_addmul_ui (C function), 177
fmpfr_bits (C function), 175
fmpfr_check_ulp (C function), 173
fmpfr_clear (C function), 172
fmpfr_cmp (C function), 174
fmpfr_cmp_2exp_si (C function), 175
fmpfr_cmpabs (C function), 175
fmpfr_cmpabs_2exp_si (C function), 175
fmpfr_cmpabs_ui (C function), 175
fmpfr_div (C function), 176
fmpfr_div_fmpz (C function), 176
fmpfr_div_precomp (C function), 168
fmpfr_divprecomp_abs_ubound (C function), 177
fmpfr_equal (C function), 174
fmpfr_exp (C function), 178
fmpfr_expm1 (C function), 178
fmpfr_fmpz_div (C function), 177
fmpfr_fmpz_div_fmpz (C function), 177
fmpfr_fmpz_div_si (C function), 177
fmpfr_get_d (C function), 174
fmpfr_get_fmpz (C function), 174
fmpfr_get_fmpz_2exp_si (C function), 175
fmpfr_getfmpz (C function), 174
fmpfr_getfmpz_2exp_si (C function), 174
fmpfr_getfmpz_fixed_fmpz (C function), 174
fmpfr_getfmpz_fixed_si (C function), 174
fmpfr_getfmpfr (C function), 173
fmpfr_getfmpfr_2exp (C function), 174
fmpfr_getfmpz (C function), 174
fmpfr_getfmpz_2exp (C function), 174
fmpfr_getfmpz_fixed (C function), 174
fmpfr_getfmpz_fixed_si (C function), 174
fmpfr_getmpfr (C function), 173
fmpfr_getmpfr_2exp (C function), 174
fmpfr_is_equal (C function), 174
fmpfr_is_finite (C function), 173
fmpfr_is_inf (C function), 172
fmpfr_is_nan (C function), 172
fmpfr_is_normal (C function), 172
fmpfr_is_one (C function), 172

Index
231
fmpr_is_pos_inf (C function), 172
fmpr_is_special (C function), 173
fmpr_is_zero (C function), 172
fmpr_log (C function), 177
fmpr_log1p (C function), 177
fmpr_max (C function), 175
fmpr_min (C function), 175
fmpr_mul (C function), 176
fmpr_mul_2exp_fmpz (C function), 176
fmpr_mul_2exp_si (C function), 176
fmpr_mul_fmpz (C function), 176
fmpr_mul_si (C function), 176
fmpr_mul_ui (C function), 176
fmpr_nan (C function), 172
fmpr_neg (C function), 176
fmpr_neg_inf (C function), 172
fmpr_neg_round (C function), 176
fmpr_one (C function), 172
fmpr_pos_inf (C function), 172
fmpr_pow_sloppy_fmpz (C function), 177
fmpr_pow_sloppy_si (C function), 177
fmpr_pow_sloppy_ui (C function), 177
FMPR_PREC_EXACT (C macro), 172
fmpr_print (C function), 175
fmpr_printd (C function), 175
fmpr_randtest (C function), 175
fmpr_randtest_not_zero (C function), 175
fmpr_randtest_special (C function), 175
FMPR_RND_CEIL (C macro), 172
FMPR_RND_DOWN (C macro), 172
FMPR_RND_FLOOR (C macro), 172
FMPR_RND_NEAR (C macro), 172
FMPR_RND_UP (C macro), 172
fmpr_root (C function), 177
fmpr_rnd_t (C type), 172
fmpr_set (C function), 173
fmpr_set_d (C function), 174
fmpr_set_error_result (C function), 173
fmpr_set_fmpz (C function), 174
fmpr_set_fmpq (C function), 174
fmpr_set_mpfr (C function), 174
fmpr_set_round (C function), 173
fmpr_set_round_fmpz (C function), 173
fmpr_set_round_fmpz_2exp (C function), 174
fmpr_set_round_si_2exp_fmpz (C function), 173
fmpr_set_round_ui_2exp_fmpz (C function), 173
fmpr_set_si (C function), 174
fmpr_set_si_2exp_si (C function), 174
fmpr_set_ui (C function), 174
fmpr_sign (C function), 175
fmpr_sgn (C function), 175
fmpr_sqr (C function), 177
fmpr_sqrt (C function), 177
fmpr_sqrt_fmpz (C function), 177
fmpr_sqrt_ui (C function), 177
H
hypgeom_bound (C function), 154
hypgeom_clear (C function), 153
hypgeom_estimate_terms (C function), 154
hypgeom_init (C function), 153
hypgeom_precompute (C function), 154
hypgeom_struct (C type), 153
hypgeom_t (C type), 153
M
mag_add (C function), 29
mag_add_2exp_fmpz (C function), 29
mag_add_lower (C function), 30
mag_add_ui (C function), 29
mag_add_ui_2exp_si (C function), 29
mag_addmul (C function), 29
mag_allocated_bytes (C function), 27
mag_bernoulli_div_fac_ui (C function), 31
mag_binpow_uiui (C function), 31
mag_clear (C function), 27
mag_cmp (C function), 28
mag_cmp_2exp_si (C function), 28
mag_const_pi (C function), 31
Index

P
partitions_fmpz_fmpz (C function), 155
partitions_fmpz_ui (C function), 155
partitions_fmpz_ui_using_doubles (C function), 155
partitions_rhr_sum_arb (C function), 155
partitions_rademacher_bound (C function), 154
psl2z_clear (C function), 134
psl2z_equal (C function), 135
psl2z_fprint (C function), 135
psl2z_init (C function), 134
psl2z_inv (C function), 135
psl2z_is_correct (C function), 135
psl2z_is_one (C function), 135
psl2z_mul (C function), 135
psl2z_one (C function), 135
psl2z_randtest (C function), 135
psl2z_randtest_correct (C function), 135
psl2z_randtest_special (C function), 135
psl2z_set (C function), 135
psl2z_struct (C function), 134
psl2z_swap (C function), 135
psl2z_t (C type), 134

S
slong (C type), 13

U
ulong (C type), 13