# Contents

1 Introduction ........................................ 1

2 General information .................................. 3

2.1 Feature overview .................................. 3

2.2 Setup ............................................ 4

2.2.1 Download .................................... 4

2.2.2 Dependencies .................................. 4

2.2.3 Installation as part of FLINT ................. 4

2.2.4 Standalone installation ....................... 4

2.2.5 Running code .................................. 5

2.3 Potential issues .................................... 5

2.3.1 Interface changes ................................ 5

2.3.2 Correctness .................................... 5

2.3.3 Aliasing ........................................ 6

2.3.4 Integer overflow ................................ 6

2.3.5 Thread safety and caches ..................... 7

2.3.6 Use of hardware floating-point arithmetic .... 7

2.4 History and changes .................................. 8

2.5 Example programs .................................... 17

2.5.1 pi.c ............................................ 17

2.5.2 hilbert_matrix.c ................................ 17

2.5.3 keiper_li.c ..................................... 18

2.5.4 real_roots.c ..................................... 19

2.5.5 poly_roots.c ..................................... 20

3 Module documentation (Arb 2.x types) ............. 23

3.1 mag.h – fixed-precision unsigned floating-point numbers for bounds .......... 23

3.1.1 Types, macros and constants .................. 23

3.1.2 Memory management ............................ 23

3.1.3 Special values .................................. 24

3.1.4 Comparisons .................................... 24

3.1.5 Input and output ................................ 24

3.1.6 Random generation ............................... 24

3.1.7 Conversions ..................................... 25

3.1.8 Arithmetic ...................................... 25

3.1.9 Fast, unsafe arithmetic .......................... 26

3.1.10 Powers and logarithms ......................... 26

3.1.11 Special functions ............................... 27

3.2 arf.h – arbitrary-precision floating-point numbers .................. 28

3.2.1 Types, macros and constants .................. 28

3.2.2 Memory management ............................ 29
### Contents

#### 3.3 arb.h – real numbers represented as floating-point balls

- **3.3.1** Types, macros and constants ........................................ 36
- **3.3.2** Memory management .................................................. 37
- **3.3.3** Assignment and rounding ............................................ 37
- **3.3.4** Assignment of special values ...................................... 38
- **3.3.5** Input and output ...................................................... 38
- **3.3.6** Random number generation ......................................... 39
- **3.3.7** Radius and interval operations .................................... 39
- **3.3.8** Comparisons .......................................................... 40
- **3.3.9** Arithmetic ............................................................ 41
- **3.3.10** Powers and roots ..................................................... 43
- **3.3.11** Exponentials and logarithms ...................................... 44
- **3.3.12** Trigonometric functions .......................................... 44
- **3.3.13** Inverse trigonometric functions ................................ 45
- **3.3.14** Hyperbolic functions .............................................. 45
- **3.3.15** Inverse hyperbolic functions .................................... 46
- **3.3.16** Constants ........................................................... 46
- **3.3.17** Gamma function and factorials ................................... 47
- **3.3.18** Zeta function ........................................................ 47
- **3.3.19** Bernoulli numbers .................................................... 49
- **3.3.20** Polylogarithms ....................................................... 49
- **3.3.21** Other special functions ............................................ 49
- **3.3.22** Internals for computing elementary functions ............... 49

#### 3.4 arb_poly.h – polynomials over the real numbers

- **3.4.1** Types, macros and constants ..................................... 51
- **3.4.2** Memory management ................................................ 52
- **3.4.3** Basic manipulation .................................................. 52
- **3.4.4** Conversions ............................................................ 53
- **3.4.5** Input and output ...................................................... 53
- **3.4.6** Random generation .................................................. 53
- **3.4.7** Comparisons .......................................................... 53
- **3.4.8** Arithmetic ............................................................ 53
- **3.4.9** Composition .......................................................... 55
- **3.4.10** Evaluation ............................................................ 56
- **3.4.11** Product trees ........................................................ 57
- **3.4.12** Multipoint evaluation .............................................. 57
- **3.4.13** Interpolation ........................................................ 58
- **3.4.14** Differentiation ....................................................... 58
- **3.4.15** Transforms ........................................................... 58
- **3.4.16** Powers and elementary functions ............................... 59
- **3.4.17** Gamma function and factorials .................................. 62
- **3.4.18** Zeta function ........................................................ 62
3.4.19 Root-finding .......................................................... 63
3.4.20 Other special polynomials ........................................... 64
3.5 \texttt{arb\_mat.h} – matrices over the real numbers ................. 64
  3.5.1 Types, macros and constants ...................................... 64
  3.5.2 Memory management .................................................. 64
  3.5.3 Conversions .............................................................. 64
  3.5.4 Random generation ...................................................... 65
  3.5.5 Input and output .......................................................... 65
  3.5.6 Comparisons .............................................................. 65
  3.5.7 Special matrices .......................................................... 65
  3.5.8 Norms ................................................................. 65
  3.5.9 Arithmetic ............................................................. 66
  3.5.10 Scalar arithmetic ......................................................... 66
  3.5.11 Gaussian elimination and solving .................................. 66
  3.5.12 Characteristic polynomial .......................................... 67
  3.5.13 Special functions ....................................................... 67
3.6 \texttt{arb\_calc.h} – calculus with real-valued functions ........... 67
  3.6.1 Types, macros and constants ...................................... 68
  3.6.2 Debugging ............................................................... 68
  3.6.3 Subdivision-based root finding ..................................... 68
  3.6.4 Newton-based root finding ......................................... 70
3.7 \texttt{acb.h} – complex numbers ........................................ 70
  3.7.1 Types, macros and constants ...................................... 71
  3.7.2 Memory management .................................................. 71
  3.7.3 Basic manipulation ..................................................... 71
  3.7.4 Input and output .......................................................... 72
  3.7.5 Random number generation ......................................... 72
  3.7.6 Precision and comparisons ......................................... 72
  3.7.7 Complex parts .......................................................... 73
  3.7.8 Arithmetic ............................................................. 73
  3.7.9 Elementary functions .................................................. 75
  3.7.10 Rising factorials .......................................................... 76
  3.7.11 Gamma function ......................................................... 76
  3.7.12 Zeta function ............................................................ 77
  3.7.13 Polylogarithms ........................................................... 77
  3.7.14 Arithmetic-geometric mean ........................................... 77
3.8 \texttt{acb\_poly.h} – polynomials over the complex numbers .......... 77
  3.8.1 Types, macros and constants ...................................... 77
  3.8.2 Memory management .................................................. 78
  3.8.3 Basic properties and manipulation ................................ 78
  3.8.4 Input and output .......................................................... 79
  3.8.5 Random generation ...................................................... 79
  3.8.6 Comparisons .............................................................. 79
  3.8.7 Conversions .............................................................. 79
  3.8.8 Arithmetic ............................................................. 80
  3.8.9 Composition ............................................................. 81
  3.8.10 Evaluation .............................................................. 82
  3.8.11 Product trees ........................................................... 83
  3.8.12 Multipoint evaluation .................................................. 83
  3.8.13 Interpolation ............................................................. 84
  3.8.14 Differentiation ............................................................ 84
  3.8.15 Elementary functions ................................................. 84
  3.8.16 Gamma function .......................................................... 87
  3.8.17 Power sums .............................................................. 87
4 Algorithms and proofs

4.1 Algorithms for mathematical constants

4.1.1 Pi

4.1.2 Logarithms of integers

4.1.3 Euler’s constant

4.1.4 Catalan’s constant

4.1.5 Khinchin’s constant
# 4.1 Algorithms for special numbers

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.6</td>
<td>Glaisher’s constant</td>
<td>112</td>
</tr>
<tr>
<td>4.1.7</td>
<td>Apery’s constant</td>
<td>112</td>
</tr>
</tbody>
</table>

## 4.2 Algorithms for gamma functions

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1</td>
<td>The Stirling series</td>
<td>112</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Rational arguments</td>
<td>113</td>
</tr>
</tbody>
</table>

## 4.3 Algorithms for polylogarithms

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1</td>
<td>Computation for small z</td>
<td>114</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Expansion for general z</td>
<td>114</td>
</tr>
</tbody>
</table>

# 5 Module documentation (Arb 1.x types)

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>fmpr.h – arbitrary-precision floating-point numbers</td>
<td>115</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Types, macros and constants</td>
<td>115</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Memory management</td>
<td>116</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Special values</td>
<td>116</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Assignment, rounding and conversions</td>
<td>117</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Comparisons</td>
<td>119</td>
</tr>
<tr>
<td>5.1.6</td>
<td>Random number generation</td>
<td>119</td>
</tr>
<tr>
<td>5.1.7</td>
<td>Input and output</td>
<td>120</td>
</tr>
<tr>
<td>5.1.8</td>
<td>Arithmetic</td>
<td>120</td>
</tr>
<tr>
<td>5.1.9</td>
<td>Special functions</td>
<td>122</td>
</tr>
<tr>
<td>5.2</td>
<td>fmprb.h – real numbers represented as floating-point balls</td>
<td>122</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Types, macros and constants</td>
<td>122</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Memory management</td>
<td>123</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Assignment and rounding</td>
<td>123</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Assignment of special values</td>
<td>123</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Input and output</td>
<td>124</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Random number generation</td>
<td>124</td>
</tr>
<tr>
<td>5.2.7</td>
<td>Radius and interval operations</td>
<td>124</td>
</tr>
<tr>
<td>5.2.8</td>
<td>Comparisons</td>
<td>125</td>
</tr>
<tr>
<td>5.2.9</td>
<td>Arithmetic</td>
<td>126</td>
</tr>
<tr>
<td>5.2.10</td>
<td>Powers and roots</td>
<td>128</td>
</tr>
</tbody>
</table>

# 6 Credits and references

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Credits and references</td>
<td>129</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Contributors</td>
<td>129</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Software</td>
<td>129</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Citing Arb</td>
<td>130</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Bibliography</td>
<td>130</td>
</tr>
</tbody>
</table>

## Bibliography

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>131</td>
</tr>
</tbody>
</table>

## Index

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>133</td>
</tr>
</tbody>
</table>
Arb is a C library for arbitrary-precision floating-point ball arithmetic, developed by Fredrik Johansson (fredrik.johansson@gmail.com). It supports real and complex numbers, polynomials, power series, matrices, and evaluation of many transcendental functions. All is done with automatic, rigorous error bounds.

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

The documentation website is http://fredrikj.net/arb/
2.1 Feature overview

Ball arithmetic, also known as mid-rad interval arithmetic, is an extension of floating-point arithmetic in which an error bound is attached to each variable. This allows doing rigorous computations over the real numbers, while avoiding the overhead of traditional (inf-sup) interval arithmetic at high precision, and eliminating much of the need for time-consuming and bug-prone manual error analysis associated with standard floating-point arithmetic. (See for example [Hoe2009].)

Other implementations of ball arithmetic include iRRAM and Mathemagix. In contrast to those systems, Arb is more focused on low-level arithmetic and computation of transcendental functions needed for number theory. Arb also differs in some technical aspects of the implementation.

Arb 2.x contains:

- A module \( \text{arf} \) 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 \( \text{mag} \) for representing magnitudes (error bounds) more efficiently than with an arbitrary-precision floating-point type.
- A module \( \text{arb} \) for real ball arithmetic, where a ball is implemented as an \( \text{arf} \) midpoint and a \( \text{mag} \) radius.
- A module \( \text{acb} \) for complex numbers in rectangular form, represented as pairs real balls.
- Functions for fast high-precision evaluation of various mathematical constants and special functions, implemented using ball arithmetic with rigorous error bounds.
- Modules \( \text{arb\_poly}, \text{acb\_poly} \) 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 \( \text{arb\_mat}, \text{acb\_mat} \) for matrices over the real and complex numbers, implemented using balls as coefficients. At the moment, only rudimentary linear algebra operations are provided.

Arb 1.x used a different set of numerical base types \( \text{fmpr}, \text{fmprb} \) and \( \text{fmpcb} \). These types had a slightly simpler internal representation, but generally had worse performance. Almost 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 2.3, only a small set of \( \text{fmpr} \) and \( \text{fmprb} \) methods are left for fallback and testing purposes.

Planned features include more transcendental functions and more extensive polynomial and matrix functionality, as well as further optimizations.

Arb uses GMP / MPIR and FLINT for the underlying integer arithmetic and other functions. The code conventions borrow from FLINT, and the project might get merged back into FLINT when the code stabilizes in the future. Arb also uses MPFR for testing purposes and for evaluation of some functions.
2.2 Setup

2.2.1 Download

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

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

2.2.2 Dependencies

Arb has the following dependencies:

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

2.2.3 Installation as part of FLINT

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

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

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

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

2.2.4 Standalone installation

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

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

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

Here is an example program to get started using Arb:

```c
#include "arb.h"

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

Compile it with:

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

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

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

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

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

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

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

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

2.3 Potential issues

2.3.1 Interface changes

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

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

2.3.2 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 contains bugs, so the usual precautions are advised:

- Perform sanity checks on the output (check known mathematical relations; recompute to another precision and compare)
- Compare against other mathematical software
- Read the source code to verify that it does what it is supposed to do

All bug reports are highly welcome!

### 2.3.3 Aliasing

As a rule, Arb allows aliasing of operands. For example, in the function call \texttt{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 \texttt{const} qualifiers on input variables – a special case is that \texttt{arb_srcptr} is considered the \texttt{const} version of \texttt{arb_ptr}). This is a natural extension of the so-called \textit{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 \textbf{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

\[
\{t^2 \mid t \in x\}
\]

and not the (generally larger) set

\[
\{tu \mid t \in x, u \in x\}.
\]

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, 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.

### 2.3.4 Integer overflow

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 checks are performed to verify that such quantities do not overflow. Precisions and lengths exceeding a small fraction of \texttt{LONG_MAX}, say \(2^{24} \sim 10^7\) 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. This caveat does not apply to exponents of floating-point numbers, which are represented as arbitrary-precision integers, nor to integers used as numerical scalars (e.g. `arb_mul_si()`). However, it still applies to conversions and operations where the result is requested exactly and sizes become an issue. For example, trying to convert the floating-point number $2^{100}$ to an integer could result in anything from a silent wrong value to thrashing followed by a crash, and it is the user’s responsibility not to attempt such a thing.

2.3.5 Thread safety and caches

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

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

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

2.3.6 Use of hardware floating-point arithmetic

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

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

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

- A `double` uses the standard IEEE 754 format (with a 53-bit significand, 11-bit exponent, encoding of infinities and NaNs, etc.)

- We assume that the compiler does not perform “unsafe” floating-point optimizations, such as reordering of operations. Unsafe optimizations are disabled by default in most modern C compilers, including GCC and Clang. The exception appears to be the Intel C++ compiler, which does some unsafe optimizations by default. These must be disabled by the user.

- We do not assume that floating-point operations are correctly rounded (a counterexample is the x87 FPU), or that rounding is done in any particular direction (the rounding mode may have been changed by the user). We assume that any floating-point operation is done with at most 1.1 ulp error.

- We do not assume that underflow or overflow behaves in a particular way (we only use doubles that fit in the regular exponent range, or explicit infinities).
• We do not use transcendental \texttt{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 \texttt{ldexp} and \texttt{sqrt}, which we assume to be correctly implemented.

### 2.4 History and changes

For more details, view the commit log in the git repository [https://github.com/fredrik-johansson/arb](https://github.com/fredrik-johansson/arb)

• 2015-01-28 - version 2.5.0
  
  – string conversion
    * added \texttt{arb_set_str}
    * added \texttt{arb_get_str} and \texttt{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 \texttt{atan} at very high precision, used instead of \texttt{mpfr_atan}
    * optimized elementary functions slightly for small input
    * added modified error functions \texttt{erfc} and \texttt{erfi}
    * added the generalized exponential integral
    * added the upper incomplete gamma function
    * implemented the complete elliptic integral of the first kind
    * implemented the arithmetic-geometric mean of complex numbers
    * optimized \texttt{arb_digamma} for small integers
    * made \texttt{mag_log_ui}, \texttt{mag_binpow_uiui} and \texttt{mag_polylog_tail} proper functions
    * added \texttt{pow}, \texttt{agm}, \texttt{erf}, \texttt{elliptic_k}, \texttt{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 \texttt{arb_sqrt1pm1}, \texttt{arb_atanh}, \texttt{arb_asinh}, \texttt{arb_atan}h
    * added \texttt{arb_log1p}, \texttt{acb_log1p}, \texttt{acb_atan}
    * added \texttt{arb_hurwitz_zeta}
    * improved parameter selection in the Hurwitz zeta function to try to avoid stalling when given enormous input
    * optimized \texttt{sqrt} and \texttt{rsqrt} of power series when given a binomial as input
    * made \texttt{arb_bernoulli_ui}(2^{64}-2) not crash
    * fixed \texttt{rgamma} of negative integers returning indeterminate
– polynomials and matrices
  * added characteristic polynomial computation for real and complex matrices
  * added polynomial set_round methods
  * added is_real methods for more types
  * added more get_unique_fmpz methods
  * added code for generating Swinnerton-Dyer polynomials
  * improved error bounding in det() and exp() of complex matrices to recognize when the result is real-valued
  * changed polynomial divrem to return success/fail instead of aborting on divide by zero

– miscellaneous
  * added logo to documentation
  * made inlined functions build as part of the library
  * silenced a clang warning
  * made _acb_vec_sort_pretty a library function

• 2014-11-15 - version 2.4.0
  – arithmetic and core functions
    * made evaluation of sin, cos and exp at medium precision faster using the sqrt trick
    * optimized arb_sinh and arb_sinh_cosh
    * optimized complex division with a small denominator
    * optimized cubing of complex numbers
    * added floor and ceil functions for the arf and arb types
    * added acb_poly powering functions
    * added acb_exp_pi_i
    * added functions for evaluation of Chebyshev polynomials
    * fixed arb_div to output nan for input containing nan
  – added a module acb_hypergeom for hypergeometric functions
    * evaluation of the generalized hypergeometric function in convergent cases
    * evaluation of confluent hypergeometric functions using asymptotic expansions
    * the Bessel function of the first kind for complex input
    * the error function for complex input
  – added a module acb_modular for modular forms and elliptic functions
    * support for working with modular transformations
    * mapping a point to the fundamental domain
    * evaluation of Jacobi theta functions and their series expansions
    * the Dedekind eta function
    * the j-invariant and the modular lambda and delta function
* Eisenstein series
* the Weierstrass elliptic function and its series expansion

miscellaneous
* fixed mag_print printing a too large exponent
* fixed printd methods to use a fallback instead of aborting when printing numbers too large for MPFR
* added version number string (arb_version)
* various additions to the documentation

• 2014-09-25 - version 2.3.0
* removed most of the legacy (Arb 1.x) modules
* updated build scripts, hopefully fixing various issues
* new implementations of arb_sin, arb_cos, arb_sin_cos, arb_atan, arb_log, arb_exp, arb_expm1, much faster up to a few thousand bits
* ported the bit-burst code for high-precision exponentials to the arb type
* speeded up arb_log_ui_from_prev
* added mag_exp, mag_expm1, mag_exp_tail, mag_pow_fmpz
* improved various mag functions
* added arb_get/set_interval_mpfr, arb_get_interval_arf, and improved arb_set_interval_arf
* improved arf_get_fmpz
* prettier printing of complex numbers with negative imaginary part
* changed some frequently-used functions from inline to non-inline to reduce code size

• 2014-08-01 - version 2.2.0
* added functions for computing polylogarithms and order expansions of polylogarithms, with support for real and complex s, z
* added a missing cast affecting C++ compatibility
* generalized powsum functions to allow a geometric factor
* improved powsum functions slightly when the exponent is an integer
* faster arb_log_ui_from_prev
* added mag_sqrt and mag_rsqrt functions
* fixed various minor bugs and added missing tests and documentation entries

• 2014-06-20 - version 2.1.0
* ported most of the remaining functions to the new arb/acb types, including:
  * elementary functions (log, atan, etc.)
  * hypergeometric series summation
  * the gamma function
  * the Riemann zeta function and related functions
  * Bernoulli numbers
  * the partition function
* the calculus modules (rigorous real root isolation, rigorous numerical integration of complex-valued functions)
* example programs
  – added several missing utility functions to the arf and mag modules

• 2014-05-27 - version 2.0.0
  – new modules mag, arf, arb, arb_poly, arb_mat, acb, acb_poly, acb_mat for higher-performance ball arithmetic
  – poly_roots2 and hilbert_matrix2 example programs
  – vector dot product and norm functions (contributed by Abhinav Baid)

• 2014-05-03 - version 1.1.0
  – faster and more accurate error bounds for polynomial multiplication (error bounds are now always as good as with classical multiplication, and multiplying high-degree polynomials with approximately equal coefficients now has proper quasilinear complexity)
  – faster and much less memory-hungry exponentials at very high precision
  – improved the partition function to support n bigger than a single word, and enabled the possibility to use two threads for the computation
  – fixed a bug in floating-point arithmetic that caused a too small bound for the rounding error to be reported when the result of an inexact operation was rounded up to a power of two (this bug did not affect the correctness of ball arithmetic, because operations on ball midpoints always round down)
  – minor optimizations to floating-point arithmetic
  – improved argument reduction of the digamma function and short series expansions of the rising factorial
  – removed the holonomic module for now, as it did not really do anything very useful

• 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
    * fmprb_trim, fmpcb_trim

2.4. History and changes
• 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
• 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

• 2013-08-07 - version 0.7

– floating-point and ball functions
• documented, added test code, and fixed bugs for various operations involving a ball containing an infinity or NaN
• added reciprocal square root functions (fmpr_rsqrt, fmprb_rsqrt) based on mprfr_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 (shiftin, 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

• 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 \texttt{fmpcb\_poly\_sub} and \texttt{fmpcb\_poly\_sub} functions
– faster exponentials (\texttt{fmpcb\_exp} and dependent functions) at low precision, using precomputation
– rewrote \texttt{fmpr\_add} and \texttt{fmpr\_sub} using \texttt{mpn} level code, improving efficiency at low precision
– ported the partition function implementation from flint (using ball arithmetic in all steps of the calculation to guarantee correctness)
– ported algorithm for computing the cosine minimal polynomial from flint (using ball arithmetic to guarantee correctness)
– support using gmp instead of mpir
– only use thread-local storage when enabled in flint
– slightly faster error bounding for the zeta function
– added some other helper functions

• 2013-03-28 - version 0.5
  – arithmetic and elementary functions
    * added \texttt{fmpr\_get\_fmpz}, \texttt{fmpr\_get\_si}
    * fixed accuracy problem with \texttt{fmpcb\_div\_2expm1}
    * special-cased squaring of complex numbers
    * added various \texttt{fmpcb} convenience functions (\texttt{addmul\_ui}, etc)
    * optimized \texttt{fmpr\_cmp\_2exp\_si} and \texttt{fmpr\_cmpabs\_2exp\_si}, and added test code for comparison functions
    * added \texttt{fmprb\_atan2}, also fixing a bug in \texttt{fmpcb\_arg}
    * added \texttt{fmprb\_sin\_pi}, \texttt{cos\_pi}, \texttt{sin\_cos\_pi} etc.
    * added \texttt{fmprb\_sin\_pi\_fmpq} (etc.) using algebraic methods for fast evaluation of roots of unity
    * faster \texttt{fmprb\_poly\_evaluate} and \texttt{evaluate\_fmpcb} using rectangular splitting
    * added \texttt{fmprb\_poly\_evaluate2}, \texttt{evaluate2\_fmpcb} for simultaneously evaluating the derivative
    * added \texttt{fmprb\_poly} root polishing code using near-optimal Newton steps (experimental)
    * added \texttt{fmpr\_root}, \texttt{fmpcb\_root} (currently based on MPFR)
    * added \texttt{fmpr\_min}, \texttt{fmpr\_max}
    * added \texttt{fmprb\_set\_interval\_fmp}, \texttt{fmprb\_union}
    * added \texttt{fmpr\_bits}, \texttt{fmpcb\_bits}, \texttt{fmpcb\_bits} for obtaining the mantissa width
    * added \texttt{fmprb\_hypot}
    * added complex square roots
    * improved \texttt{fmpr\_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

• 2013-01-26 - version 0.4
  – much faster fmpr_mul, fmprb_mul and set_round, resulting in general speed improvements
  – code for computing the complex Hurwitz zeta function with derivatives
  – fixed and documented error bounds for hypergeometric series
  – better algorithm for series evaluation of the gamma function at a rational point
– much faster generation of Bernoulli numbers
– complex log, exp, pow, trigonometric functions (currently based on MPFR)
– complex nth roots via Newton iteration
– added code for arithmetic on fmpcb_polys
– code for computing Khinchin’s constant
– code for rising factorials of polynomials or power series
– faster sin_cos
– better div_2expm1
– many other new helper functions
– improved thread safety
– more test code for core operations

• 2012-11-07 - version 0.3
  – converted documentation to sphinx
  – new module fmpcb for ball interval arithmetic over the complex numbers
    * conversions, utility functions and arithmetic operations
  – new module fmpcb_mat for matrices over the complex numbers
    * conversions, utility functions and arithmetic operations
    * multiplication, LU decomposition, solving, inverse and determinant
  – new module fmpcb_poly for polynomials over the complex numbers
    * root isolation for complex polynomials
  – new module fmpz_holonomic for functions/sequences defined by linear differential/difference equations with polynomial coefficients
    * functions for creating various special sequences and functions
    * some closure properties for sequences
    * Taylor series expansion for differential equations
    * computing the nth entry of a sequence using binary splitting
    * computing the nth entry mod p using fast multipoint evaluation
  – generic binary splitting code with automatic error bounding is now used for evaluating hypergeometric series
  – matrix powering
  – various other helper functions

• 2012-09-29 - version 0.2
  – code for computing the gamma function (Karatsuba, Stirling’s series)
  – rising factorials
  – fast exp_series using Newton iteration
  – improved multiplication of small polynomials by using classical multiplication
  – implemented error propagation for square roots
2.5 Example programs

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

```
make examples
```

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

2.5.1 pi.c

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

Sample output, computing π to one million digits:

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

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

2.5.2 hilbert_matrix.c

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

Sample output:
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:

\[
\text{keiper\_li} \ n \ [-\text{prec} \ prec] \ [-\text{threads} \ num\_threads] \ [-\text{out} \ out\_file]
\]

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

Sample output:

```
> build/examples/keiper_li 1000 -threads 2
zeta: cpu/wall(s): 0.4 0.244
virt/peak/res/peak(MB): 167.98 294.69 5.09 7.43
log: cpu/wall(s): 0.03 0.038
gamma: cpu/wall(s): 0.02 0.016
binomial transform: cpu/wall(s): 0.01 0.018
```

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:

\[
\text{keiper\_li} \ n \ [-\text{prec} \ prec] \ [-\text{threads} \ num\_threads] \ [-\text{out} \ out\_file]
\]

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

Sample output:

```
> build/examples/keiper_li 1000 -threads 2
zeta: cpu/wall(s): 0.4 0.244
virt/peak/res/peak(MB): 167.98 294.69 5.09 7.43
log: cpu/wall(s): 0.03 0.038
gamma: cpu/wall(s): 0.02 0.016
binomial transform: cpu/wall(s): 0.01 0.018
```
2.5.4 real_roots.c

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

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

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

- 0 - \(Z(x)\) (Riemann-Siegel Z-function)
- 1 - \(\sin(x)\)
- 2 - \(\sin(x^2)\)
- 3 - \(\sin(1/x)\)

The following options are available:

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

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

```
> build/examples/real_roots 0 0.0 50.0 -verbose
interval: 25 +/- 25
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
found isolated root in: 14.12353515625 +/- 0.012207
found isolated root in: 21.0205078125 +/- 0.024414
found isolated root in: 25.0244140625 +/- 0.024414
found isolated root in: 30.43212890625 +/- 0.012207
found isolated root in: 32.9345703125 +/- 0.024414
found isolated root in: 37.5732421875 +/- 0.024414
found isolated root in: 40.9423828125 +/- 0.024414
found isolated root in: 43.32275390625 +/- 0.012207
found isolated root in: 48.01025390625 +/- 0.012207
found isolated root in: 49.76806640625 +/- 0.012207
---------------------------------------------------------------
Found roots: 10
Subintervals possibly containing undetected roots: 0
Function evaluations: 3425
cpu/wall(s): 1.22 1.229
virt/peak/res/peak(MB): 20.63 20.66 2.23 2.23
```

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

\[
> \text{build/examples/real_roots 2 0 100} \\
\text{interval: 50 +/- 50} \\
\text{maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30} \\
\text{---------------------------------------------------------------} \\
\text{Found roots: 3183} \\
\text{Subintervals possibly containing undetected roots: 1} \\
\text{Function evaluations: 34058} \\
\text{cpu/wall(s): 0.26 0.263} \\
\text{virt/peak/res/peak(MB): 20.73 20.76 1.72 1.72} \\
\]

This does not miss any roots:

\[
> \text{build/examples/real_roots 2 1 100} \\
\text{interval: 50.5 +/- 49.5} \\
\text{maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30} \\
\text{---------------------------------------------------------------} \\
\text{Found roots: 3183} \\
\text{Subintervals possibly containing undetected roots: 0} \\
\text{Function evaluations: 34039} \\
\text{cpu/wall(s): 0.26 0.266} \\
\text{virt/peak/res/peak(MB): 20.73 20.76 1.70 1.70} \\
\]

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:

\[
> \text{build/examples/real_roots 3 0.0 1.0} \\
\text{interval: 0.5 +/- 0.5} \\
\text{maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30} \\
\text{---------------------------------------------------------------} \\
\text{Found roots: 10198} \\
\text{Subintervals possibly containing undetected roots: 24695} \\
\text{Function evaluations: 202587} \\
\text{cpu/wall(s): 1.73 1.731} \\
\text{virt/peak/res/peak(MB): 21.84 22.89 2.76 2.76} \\
\]

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

### 2.5.5 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. For convergence, the input polynomial is required to be squarefree.

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

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

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

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

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

```
> build/examples/poly_roots -print 15 w 100
prec=53: 0 isolated roots | cpu/wall(s): 0.42 0.426
prec=106: 0 isolated roots | cpu/wall(s): 1.37 1.368
prec=212: 0 isolated roots | cpu/wall(s): 1.48 1.485
prec=424: 100 isolated roots | cpu/wall(s): 0.61 0.611
done!
```

```
(1 + 1.7285178043492e-125j) +/- (7.2e-122, 7.2e-122j)
(2 + 5.1605530263601e-122j) +/- (3.77e-118, 3.77e-118j)
(3 + -2.5811555871665e-118j) +/- (5.72e-115, 5.72e-115j)
(4 + 1.02141628524271e-115j) +/- (4.38e-112, 4.38e-112j)
(5 + 1.61326834094948e-113j) +/- (2.6e-109, 2.6e-109j)
... 
(95 + 4.15294196875447e-62j) +/- (6.66e-59, 6.66e-59j)
(96 + 3.54502401922667e-64j) +/- (7.37e-60, 7.37e-60j)
(97 + -1.6775595325625e-65j) +/- (6.4e-61, 6.4e-61j)
(98 + 2.04638822325299e-65j) +/- (4e-62, 4e-62j)
(99 + -2.73425468028238e-66j) +/- (1.71e-63, 1.71e-63j)
(100 + -1.0095011302288e-68j) +/- (3.24e-65, 3.24e-65j)
``` 

cpu/wall(s): 3.88 3.893

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

```
> build/examples/poly_roots -refine 100 -print 20 b 16
prec=53: 16 isolated roots | cpu/wall(s): 0 0.007
prec=106: 16 isolated roots | cpu/wall(s): 0 0.004
prec=212: 16 isolated roots | cpu/wall(s): 0 0.004
prec=424: 16 isolated roots | cpu/wall(s): 0 0.004
```
done!

(-0.94308706466055783383 + -5.5122726631684603e-128j) +/- (2.2e-125, 2.2e-125j)
(-0.75534059252067985752 + 1.937401283040249068e-128j) +/- (1.09e-125, 1.09e-125j)
(-0.24999757119077421009 + -4.5347924422246038692e-130j) +/- (3.6e-127, 3.6e-127j)
(0.24999757152512726002 + 4.2191300761823281708e-129j) +/- (4.98e-127, 4.98e-127j)
(0.75000242847487273998 + 9.0360649917413170142e-126j) +/- (8.88e-126, 8.88e-126j)
(1.249997571907742101 + 7.880412380107088267e-127j) +/- (2.66e-124, 2.66e-124j)
(1.7553405925206798575 + 5.432465269253967768e-126j) +/- (6.23e-123, 6.23e-123j)
(1.9430870646605578338 + 3.035377342500953239e-125j) +/- (7.05e-123, 7.05e-123j)
(-0.99509334829256233279 + 0.4454795815710360805j) +/- (5.5e-125, 5.5e-125j)
(-0.99509334829256233279 + -0.4454795815710360805j) +/- (5.46e-125, 5.46e-125j)
(1.9950933482925623328 + 0.4454795815710360805j) +/- (1.44e-122, 1.44e-122j)
(1.9950933482925623328 + -0.4454795815710360805j) +/- (1.43e-122, 1.43e-122j)
(-0.92177327714429290564 + -1.0954360955079385542j) +/- (9.31e-125, 9.31e-125j)
(-0.92177327714429290564 + 1.0954360955079385542j) +/- (1.02e-124, 1.02e-124j)
(1.9217732771442929056 + 1.0954360955079385542j) +/- (9.15e-123, 9.15e-123j)
(1.9217732771442929056 + -1.0954360955079385542j) +/- (8.12e-123, 8.12e-123j)
cpu/wall(s): 0.02 0.02
CHAPTER
THREE

MODULE DOCUMENTATION (ARB 2.X TYPES)

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

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

3.1.1 Types, macros and constants

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

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

3.1.2 Memory management

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

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

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

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

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

mag_ptr_mag_vec_init (long n)
Allocates a vector of length n. All entries are set to zero.
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, long 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.

3.1.6 Random generation

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

void mag_set_d (mag_t y, double x)
void mag_set_fmpr (mag_t y, const fmp_t x)
void mag_set_ui (mag_t y, ulong x)
void mag_set_fmpz (mag_t y, const fmpz_t x)

Sets y to an upper bound for $|x|$.

void mag_set_d_2exp_fmpz (mag_t z, double x, const fmpz_t y)
void mag_set_fmpz_2exp_fmpz (mag_t z, const fmpz_t x, const fmpz_t y)

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

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

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

void mag_set_ui_lower (mag_t z, ulong x)
void mag_set_fmpz_lower (mag_t z, const fmpz_t x)

Sets z to a lower bound for $|x|$.

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

### 3.1.8 Arithmetic

void mag_mul_2exp_si (mag_t z, const mag_t x, long y)
void mag_mul_2exp_fmpz (mag_t z, const mag_t x, const fmpz_t y)
Sets z to $x \times 2^y$. This operation is exact.

void mag_mul (mag_t z, const mag_t x, const mag_t y)
void mag_mul_ui (mag_t z, const mag_t x, ulong y)
void mag_mul_fmpz (mag_t z, const mag_t x, const fmpz_t y)

Sets z to an upper bound for $xy$.

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

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

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

void mag_div (mag_t z, const mag_t x, const mag_t y)
void mag_div_ui (mag_t z, const mag_t x, ulong y)
void mag_div_fmpz (mag_t z, const mag_t x, const fmpz_t y)
Sets z to an upper bound for $x/y$.

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

Sets z to a lower bound for $xy$.

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

Sets z to a lower bound for $xy$.

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

Sets z to a lower bound for $x + y$.

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

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

### 3.1.9 Fast, unsafe arithmetic

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

void **mag_fast_init_set** (mag_t x, const mag_t y)

Initialises x and sets it to the value of y.

void **mag_fast_zero** (mag_t x)

Sets x to zero.

int **mag_fast_is_zero** (const mag_t x)

Returns nonzero iff x is zero.

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

Sets z to an upper bound for $xy$.

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

Sets z to an upper bound for $z + xy$.

void **mag_fast_add_2exp_si** (mag_t z, const mag_t x, long e)

Sets z to an upper bound for $x + 2^e$.

void **mag_fast_mul_2exp_si** (mag_t z, const mag_t x, long e)

Sets z to an upper bound for $x2^e$.

### 3.1.10 Powers and logarithms

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

void **mag_pow_fmpz** (mag_t z, const mag_t x, const fmpz_t e)

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

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

Sets z to a lower bound for $x^e$.

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

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

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

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

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

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

void **mag_log1p** (mag_t z, const mag_t x)

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

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

void mag_expml (mag_t z, const mag_t x)
    Sets z to an upper bound for exp(x) − 1. The bound is computed accurately for small x.

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

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

3.1.11 Special functions

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

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

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

void mag_polylog_tail (mag_t u, const mag_t z, long s, ulong d, ulong N)
    Sets u to an upper bound for \(\sum_{k=N}^{\infty} \frac{z k \log^d(k)}{k^s}\).

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

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

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

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

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

In particular, we take

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

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

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

and

\[
\left( \frac{\log(k + 1)}{\log(k)} \right)^d \leq \left( 1 + \frac{1}{k \log(k)} \right)^d.
\]
3.2 arf.h – arbitrary-precision floating-point numbers

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

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

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

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

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

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

3.2.1 Types, macros and constants

**arf_struct**

**arf_t**

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

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

**arf_rnd_t**

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

**ARF_RND_DOWN**

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

**ARF_RND_UP**

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

**ARF_RND_FLOOR**

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

ARF_RND_NEAR
Specifies that the result of an operation should be rounded to the nearest representable number, rounding to 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).

ARF_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 with 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.

3.2.2 Memory management

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

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

3.2.3 Special values

void arf_zero (arf_t x)
void arf_one (arf_t x)
void arf_pos_inf (arf_t x)
void arf_neg_inf (arf_t x)
void arf_nan (arf_t x)
Sets x respectively to 0, 1, +∞, −∞, 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 \texttt{arf\_is\_finite} (\texttt{arf\_t} \textit{x})

Returns nonzero iff \textit{x} is a finite floating-point value, i.e. not one of the values $+\infty$, $-\infty$, NaN. (Note that this is not equivalent to the negation of \texttt{arf\_is\_inf()}.)

### 3.2.4 Assignment, rounding and conversions

void \texttt{arf\_set} (\texttt{arf\_t} \textit{y}, const \texttt{arf\_t} \textit{x})

void \texttt{arf\_set\_mpz} (\texttt{arf\_t} \textit{y}, const \texttt{mpz\_t} \textit{x})

void \texttt{arf\_set\_fmpz} (\texttt{arf\_t} \textit{y}, const \texttt{fmpz\_t} \textit{x})

void \texttt{arf\_set\_ui} (\texttt{arf\_t} \textit{y}, ulong \textit{x})

void \texttt{arf\_set\_si} (\texttt{arf\_t} \textit{y}, long \textit{x})

void \texttt{arf\_set\_mpfr} (\texttt{arf\_t} \textit{y}, const \texttt{mpfr\_t} \textit{x})

void \texttt{arf\_set\_fmpfr} (\texttt{arf\_t} \textit{y}, const \texttt{fmpr\_t} \textit{x})

void \texttt{arf\_set\_d} (\texttt{arf\_t} \textit{y}, double \textit{x})

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

void \texttt{arf\_swap} (\texttt{arf\_t} \textit{y}, \texttt{arf\_t} \textit{x})

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

void \texttt{arf\_init\_set\_ui} (\texttt{arf\_t} \textit{y}, ulong \textit{x})

void \texttt{arf\_init\_set\_si} (\texttt{arf\_t} \textit{y}, long \textit{x})

  Initialises \textit{y} and sets it to \textit{x} in a single operation.

int \texttt{arf\_set\_round} (\texttt{arf\_t} \textit{y}, const \texttt{arf\_t} \textit{x}, long \textit{prec}, \texttt{arf\_rnd\_t} \textit{rnd})

int \texttt{arf\_set\_round\_si} (\texttt{arf\_t} \textit{x}, long \textit{v}, long \textit{prec}, \texttt{arf\_rnd\_t} \textit{rnd})

int \texttt{arf\_set\_round\_mpz} (\texttt{arf\_t} \textit{y}, const \texttt{mpz\_t} \textit{x}, long \textit{prec}, \texttt{arf\_rnd\_t} \textit{rnd})

int \texttt{arf\_set\_round\_fmpz} (\texttt{arf\_t} \textit{y}, const \texttt{fmpz\_t} \textit{x}, long \textit{prec}, \texttt{arf\_rnd\_t} \textit{rnd})

Sets \textit{y} to \textit{x}, rounded to \textit{prec} bits in the direction specified by \textit{rnd}.

void \texttt{arf\_set\_si\_2exp\_si} (\texttt{arf\_t} \textit{y}, long \textit{m}, long \textit{e})

void \texttt{arf\_set\_ui\_2exp\_si} (\texttt{arf\_t} \textit{y}, ulong \textit{m}, long \textit{e})

void \texttt{arf\_set\_fmpz\_2exp} (\texttt{arf\_t} \textit{y}, const \texttt{fmpz\_t} \textit{m}, const \texttt{fmpz\_t} \textit{e})

  Sets \textit{y} to \textit{m} \times 2^{\textit{e}}.

int \texttt{arf\_set\_round\_fmpz\_2exp} (\texttt{arf\_t} \textit{y}, const \texttt{fmpz\_t} \textit{x}, const \texttt{fmpz\_t} \textit{e}, long \textit{prec}, \texttt{arf\_rnd\_t} \textit{rnd})

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

void \texttt{arf\_get\_fmpz\_2exp} (\texttt{fmpz\_t} \textit{m}, \texttt{fmpz\_t} \textit{e}, const \texttt{arf\_t} \textit{x})

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

double \texttt{arf\_get\_d} (const \texttt{arf\_t} \textit{x}, \texttt{arf\_rnd\_t} \textit{rnd})

  Returns \textit{x} rounded to a double in the direction specified by \textit{rnd}.

void \texttt{arf\_get\_mpfr} (\texttt{fmpr\_t} \textit{y}, const \texttt{arf\_t} \textit{x})

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

void *arf_get_fmpz* (fmpz_t z, const arf_t x, arf_rnd_t rnd)
Sets z to x rounded to the nearest integer in the direction specified by rnd. If rnd is ARF_RND_NEAR, rounds to the nearest even integer in case of a tie. Aborts if x is infinite, NaN or if the exponent is unreasonably large.

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

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

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

### 3.2.5 Comparisons and bounds

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

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

int *arf_cmpabs* (const arf_t x, const arf_t y)
int *arf_cmpabs_ui* (const arf_t x, ulong y)
int *arf_cmpabs_mag* (const arf_t x, const mag_t y)
int *arf_cmp_2exp_si* (const arf_t x, long e)
int *arf_cmpabs_2exp_si* (const arf_t x, long e)
Compares the absolute values of x and y.

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.

long *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, long e)
Returns nonzero iff \( x \) equals \( n2^e \) for some integer \( n \).

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

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

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

3.2.6 Magnitude functions

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

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

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

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

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

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

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

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

3.2.7 Shallow assignment

void arf_init_set_shallow (arf_t z, const arf_t x)

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

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

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

Initializes \( z \) shallowly to the negation of \( x \).

### 3.2.8 Random number generation

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

Indentical 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, long 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, long 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, long e)

Sets \( y = x \times 2^e \) exactly.

int **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, long prec, arf_rnd_t rnd)

int **arf_mul/ui** (arf_t z, const arf_t x, ulong y, long prec, arf_rnd_t rnd)

int **arf_mul/si** (arf_t z, const arf_t x, long y, long prec, arf_rnd_t rnd)

int **arf_mul/mpz** (arf_t z, const arf_t x, const mpz_t y, long prec, arf_rnd_t rnd)

int **arf_mul/fmpz** (arf_t z, const arf_t x, const fmpz_t y, long 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, long prec, arf_rnd_t rnd)

int **arf_add/si** (arf_t z, const arf_t x, long y, long prec, arf_rnd_t rnd)
3.2.11 Summation

int arf_add_ui (arf_t z, const arf_t x, ulong y, long prec, arf_rnd_t rnd)
int arf_add_fmpz (arf_t z, const arf_t x, const fmpz_t y, long prec, arf_rnd_t rnd)
    Sets \( z = x + y \), rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \), returning nonzero iff the operation is inexact.
int arf_add_fmpz_2exp (arf_t z, const arf_t x, const fmpz_t y, long prec, arf_rnd_t rnd)
    Sets \( z = x + y2^e \), rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \), returning nonzero iff the operation is inexact.
int arf_sub (arf_t z, const arf_t x, const arf_t y, long prec, arf_rnd_t rnd)
int arf_sub_si (arf_t z, const arf_t x, long y, long prec, arf_rnd_t rnd)
int arf_sub_ui (arf_t z, const arf_t x, ulong y, long prec, arf_rnd_t rnd)
int arf_sub_fmpz (arf_t z, const arf_t x, const fmpz_t y, long prec, arf_rnd_t rnd)
    Sets \( z = x - y \), rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \), returning nonzero iff the operation is inexact.
int arf_addmul (arf_t z, const arf_t x, const arf_t y, long prec, arf_rnd_t rnd)
int arf_addmul_ui (arf_t z, const arf_t x, ulong y, long prec, arf_rnd_t rnd)
int arf_addmul_si (arf_t z, const arf_t x, long y, long prec, arf_rnd_t rnd)
int arf_addmul_mpz (arf_t z, const arf_t x, const mpz_t y, long prec, arf_rnd_t rnd)
int arf_addmul_fmpz (arf_t z, const arf_t x, const fmpz_t y, long prec, arf_rnd_t rnd)
    Sets \( z = z + x \times y \), rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \), returning nonzero iff the operation is inexact.
int arf_submul (arf_t z, const arf_t x, const arf_t y, long prec, arf_rnd_t rnd)
int arf_submul_ui (arf_t z, const arf_t x, ulong y, long prec, arf_rnd_t rnd)
int arf_submul_si (arf_t z, const arf_t x, long y, long prec, arf_rnd_t rnd)
int arf_submul_mpz (arf_t z, const arf_t x, const mpz_t y, long prec, arf_rnd_t rnd)
int arf_submul_fmpz (arf_t z, const arf_t x, const fmpz_t y, long prec, arf_rnd_t rnd)
    Sets \( z = z - x \times y \), rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \), returning nonzero iff the operation is inexact.

3.2.12 Division

int arf_div (arf_t z, const arf_t x, const arf_t y, long prec, arf_rnd_t rnd)
int arf_div_ui (arf_t z, const arf_t x, ulong y, long prec, arf_rnd_t rnd)
int arf_ui_div (arf_t z, const arf_t x, ulong y, long prec, arf_rnd_t rnd)
int arf_div_si (arf_t z, const arf_t x, long y, long prec, arf_rnd_t rnd)
3.2.13 Square roots

int arf_sqrt (arf_t z, long x, const arf_t y, long prec, arf_rnd_t rnd)
int arf_sqrt_ui (arf_t z, ulong x, long prec, arf_rnd_t rnd)
int arf_sqrt_fmpz (arf_t z, const arf_t x, const fmpz_t y, long prec, arf_rnd_t rnd)
int arf_div_fmpz (arf_t z, const fmpz_t x, const arf_t y, long 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.

3.2.14 Complex arithmetic

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

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

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

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

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

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

3.3 arb.h – real numbers represented as floating-point balls

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 arb_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).

The \texttt{arb_t} type is almost identical semantically to the legacy \texttt{fmprb_t} type, but uses a more efficient internal representation. Whereas the midpoint and radius of an \texttt{fmprb_t} both have the same type, the \texttt{arb_t} type uses an \texttt{arf_t} for the midpoint and a \texttt{mag_t} for the radius. Code designed to manipulate the radius of an \texttt{fmprb_t} directly can be ported to the \texttt{arb_t} type by writing the radius to a temporary \texttt{arf_t} variable, manipulating that variable, and then converting back to the \texttt{mag_t} radius. Alternatively, \texttt{mag_t} methods can be used directly where available.

### 3.3.1 Types, macros and constants

**\texttt{arb_struct}**

**\texttt{arb_t}**

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

**\texttt{arb_ptr}**

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

**\texttt{arb_srcptr}**

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

**\texttt{arb_midref}(x)**

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

**\texttt{arb_radref}(x)**

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


3.3.2 Memory management

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

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

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

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

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

3.3.3 Assignment and rounding

void arb_set_fmprb (arb_t y, const fmprb_t x)
void arb_get_fmprb (fmprb_t y, const arb_t x)
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, long x)
void arb_set_ui (arb_t y, ulong x)
void arb_set_fmpz (arb_t y, const fmpz_t x)
Sets \( y \) to the value of \( x \) without rounding.

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

void arb_set_round (arb_t y, const arb_t x, long prec)
void arb_set_round_fmpz (arb_t y, const fmpz_t x, long prec)
Sets \( y \) to the value of \( x \), rounded to \( prec \) bits.

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

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

int arb_set_str (arb_t res, const char * inp, long 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 “-3.1459e-1”, and may also consist of two such literals separated by the symbol “+/-" and optionally enclosed in brackets, e.g. “[3.25 +/- 0.0001]”, or simply “[+/- 10]” with an implicit zero midpoint. The output is rounded to \( prec \) bits, and if the binary-to-decimal conversion is inexact, the resulting error is added to the radius.

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

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

char * arb_get_str (const arb_t x, long 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).

### 3.3.4 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.
```

### 3.3.5 Input and output

```c
void arb_print (const arb_t x)
    Prints the internal representation of $x$.

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

### 3.3.6 Random number generation

```c
void arb_randtest (arb_t x, flint_rand_t state, long prec, long 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, long prec, long mag_bits)
    Generates a random number with zero radius.

void arb_randtest_precise (arb_t x, flint_rand_t state, long prec, long 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, long prec, long mag_bits)
Generates a random number with midpoint and radius chosen independently, possibly giving a very large inter-
val.

void `arb_randtest_special` (arb_t x, flint_rand_t state, long prec, long 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, long 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.

### 3.3.7 Radius and interval operations

void `arb_add_error_arf` (arb_t x, const arf_t err)
Adds err, which is assumed to be nonnegative, to the radius of x.

void `arb_add_error_2exp_si` (arb_t x, long e)
void `arb_add_error_2exp_fmpz` (arb_t x, const fmpz_t e)
Adds $2^e$ to the radius of x.

void `arb_add_error` (arb_t x, const arb_t error)
Adds the supremum of err, which is assumed to be nonnegative, to the radius of x.

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

void `arb_get_abs_ubound_arf` (arf_t u, const arb_t x, long 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, long 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_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.

`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. $x = [a, b] \times 2^{\exp}$.

The outcome is undefined if the midpoint or radius of x is non-finite, or if the difference in magnitude between
the midpoint and radius is so large that representing the endpoints exactly would cause overflows.

void `arb_set_interval_arf` (arb_t x, const arf_t a, const arf_t b, long prec)
void `arb_set_interval_mpfr` (arb_t x, const mpfr_t a, const mpfr_t b, long prec)
Sets x to a ball containing the interval $[a, b]$. We require that $a \leq b$.

void `arb_get_interval_arf` (arf_t a, arf_t b, const arb_t x, long 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.

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

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

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

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

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

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

void **arb_get_fmpz_mid_rad_10exp** (fmpz_t mid, fmpz_t rad, fmpz_t exp, const arb_t x, long 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.

### 3.3.8 Comparisons

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

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

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

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

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

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

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

Note that this is not the same thing as testing whether both \(x\) and \(y\) certainly represent the same real number, unless either \(x\) or \(y\) is exact (and neither contains NaN). To test whether both operands might represent the same mathematical quantity, use \(arb_overlaps()\) or \(arb_contains()\), depending on the circumstance.
int \texttt{arb\_is\_positive} (const \texttt{arb\_t} \texttt{x})

int \texttt{arb\_is\_nonnegative} (const \texttt{arb\_t} \texttt{x})

int \texttt{arb\_is\_negative} (const \texttt{arb\_t} \texttt{x})

int \texttt{arb\_is\_nonpositive} (const \texttt{arb\_t} \texttt{x})

Returns nonzero iff all points \( p \) in the interval represented by \( x \) satisfy, respectively, \( p > 0 \), \( p \geq 0 \), \( p < 0 \), \( p \leq 0 \).

If \( x \) contains NaN, returns zero.

int \texttt{arb\_overlaps} (const \texttt{arb\_t} \texttt{x}, const \texttt{arb\_t} \texttt{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 \texttt{arb\_contains\_arf} (const \texttt{arb\_t} \texttt{x}, const \texttt{arf\_t} \texttt{y})

int \texttt{arb\_contains\_fmpq} (const \texttt{arb\_t} \texttt{x}, const \texttt{fmpq\_t} \texttt{y})

int \texttt{arb\_contains\_fmpz} (const \texttt{arb\_t} \texttt{x}, const \texttt{fmpz\_t} \texttt{y})

int \texttt{arb\_contains\_si} (const \texttt{arb\_t} \texttt{x}, \texttt{long} \texttt{y})

int \texttt{arb\_contains\_mpfr} (const \texttt{arb\_t} \texttt{x}, const \texttt{mpfr\_t} \texttt{y})

int \texttt{arb\_contains} (const \texttt{arb\_t} \texttt{x}, const \texttt{arb\_t} \texttt{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 \texttt{arb\_contains\_zero} (const \texttt{arb\_t} \texttt{x})

int \texttt{arb\_contains\_negative} (const \texttt{arb\_t} \texttt{x})

int \texttt{arb\_contains\_nonpositive} (const \texttt{arb\_t} \texttt{x})

int \texttt{arb\_contains\_positive} (const \texttt{arb\_t} \texttt{x})

int \texttt{arb\_contains\_nonnegative} (const \texttt{arb\_t} \texttt{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.

\section{Arithmetic}

\texttt{void \texttt{arb\_neg}} (\texttt{arb\_t} \texttt{y}, \texttt{const arb\_t} \texttt{x})

\texttt{void \texttt{arb\_neg\_round}} (\texttt{arb\_t} \texttt{y}, \texttt{const arb\_t} \texttt{x}, \texttt{long} \texttt{prec})

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

\texttt{void \texttt{arb\_abs}} (\texttt{arb\_t} \texttt{x}, \texttt{const arb\_t} \texttt{y})

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

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

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

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

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

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

Sets \( z = x + y \), rounded to \texttt{prec} bits. The precision can be \texttt{ARF\_PREC\_EXACT} provided that the result fits in memory.
void `arb_add_fmpz_2exp`(arb_t z, const arb_t x, const fmpz_t m, const fmpz_t e, long prec)
   Sets \( z = x + m \cdot 2^e \), rounded to \( \text{prec} \) bits. The precision can be \textit{ARF\_PREC\_EXACT} provided that the result fits in memory.

void `arb_sub` (arb_t z, const arb_t x, const arb_t y, long prec)

void `arb_sub_arf` (arb_t z, const arb_t x, const arf_t y, long prec)

void `arb_sub_ui` (arb_t z, const arb_t x, ulong y, long prec)

void `arb_sub_si` (arb_t z, const arb_t x, long y, long prec)

void `arb_sub_fmpz` (arb_t z, const arb_t x, const fmpz_t y, long prec)
   Sets \( z = x - y \), rounded to \( \text{prec} \) bits. The precision can be \textit{ARF\_PREC\_EXACT} provided that the result fits in memory.

void `arb_mul` (arb_t z, const arb_t x, const arb_t y, long prec)

void `arb_mul_arf` (arb_t z, const arb_t x, const arf_t y, long prec)

void `arb_mul_si` (arb_t z, const arb_t x, long y, long prec)

void `arb_mul_fmpz` (arb_t z, const arb_t x, const fmpz_t y, long prec)
   Sets \( z = x \cdot y \), rounded to \( \text{prec} \) bits. The precision can be \textit{ARF\_PREC\_EXACT} provided that the result fits in memory.

void `arb_mul_2exp_si` (arb_t y, const arb_t x, long e)

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

void `arb_addmul` (arb_t z, const arb_t x, const arb_t y, long prec)

void `arb_addmul_arf` (arb_t z, const arb_t x, const arf_t y, long prec)

void `arb_addmul_si` (arb_t z, const arb_t x, long y, long prec)

void `arb_addmul_ui` (arb_t z, const arb_t x, ulong y, long prec)

void `arb_addmul_fmpz` (arb_t z, const arb_t x, const fmpz_t y, long prec)
   Sets \( z = z + x \cdot y \), rounded to \( \text{prec} \) bits. The precision can be \textit{ARF\_PREC\_EXACT} provided that the result fits in memory.

void `arb_submul` (arb_t z, const arb_t x, const arb_t y, long prec)

void `arb_submul_arf` (arb_t z, const arb_t x, const arf_t y, long prec)

void `arb_submul_si` (arb_t z, const arb_t x, long y, long prec)

void `arb_submul_ui` (arb_t z, const arb_t x, ulong y, long prec)

void `arb_submul_fmpz` (arb_t z, const arb_t x, const fmpz_t y, long prec)
   Sets \( z = z - x \cdot y \), rounded to \( \text{prec} \) bits. The precision can be \textit{ARF\_PREC\_EXACT} provided that the result fits in memory.

void `arb_inv` (arb_t y, const arb_t x, long prec)
   Sets \( z = 1/x \).

void `arb_div` (arb_t z, const arb_t x, const arb_t y, long prec)

void `arb_div_arf` (arb_t z, const arb_t x, const arf_t y, long prec)

void `arb_div_si` (arb_t z, const arb_t x, long y, long prec)

void `arb_div_ui` (arb_t z, const arb_t x, ulong y, long prec)
void \texttt{arb\_div\_fmpz} (arb\_t z, const arb\_t x, const fmpz\_t y, long prec)

void \texttt{arb\_fmpz\_div\_fmpz} (arb\_t z, const fmpz\_t x, const fmpz\_t y, long prec)

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

Sets $z = x/y$, rounded to \textit{prec} bits. If $y$ contains zero, $z$ is set to 0 ± \infty. Otherwise, error propagation uses the rule

$$|\frac{x}{y} - x\xi_1 y + \frac{x\xi_2 b - y\xi_2 a}{y(y + \xi_2 b)}| \leq \frac{|xb| + |ya|}{|y||y - b|}$$

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 \texttt{arb\_div\_2expml\_ui} (arb\_t z, const arb\_t x, ulong n, long prec)

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

### 3.3.10 Powers and roots

void \texttt{arb\_sqrt} (arb\_t z, const arb\_t x, long prec)

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

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

void \texttt{arb\_sqrt\_ui} (arb\_t z, ulong x, long prec)

Sets $z$ to the square root of $x$, rounded to \textit{prec} bits.

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

void \texttt{arb\_sqrtpos} (arb\_t z, const arb\_t x, long 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), and producing an output interval not containing any negative numbers (unless the radius is infinite).

void \texttt{arb\_hypot} (arb\_t z, const arb\_t x, const arb\_t y, long prec)

Sets $z$ to $\sqrt{x^2 + y^2}$.

void \texttt{arb\_rsqrt} (arb\_t z, const arb\_t x, long prec)

void \texttt{arb\_rsqrt\_ui} (arb\_t z, ulong x, long prec)

Sets $z$ to the reciprocal square root of $x$, rounded to \textit{prec} bits. At high precision, this is faster than computing a square root.

void \texttt{arb\_sqrtpm1} (arb\_t z, const arb\_t x, long prec)

Sets $z = \sqrt{1 + x} - 1$, computed accurately when $x \approx 0$.

void \texttt{arb\_root} (arb\_t z, const arb\_t x, ulong k, long prec)

Sets $z$ to the $k$-th root of $x$, rounded to \textit{prec} bits. As currently implemented, this function is only fast for small $k$.

For large $k$ it is better to use \texttt{arb\_pow\_fmpz()} or \texttt{arb\_pow()}.

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

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

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

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

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

Sets $y = b^e$ using binary exponentiation (with an initial division if $e < 0$). Provided that $b$ and $e$ are small enough and the exponent is positive, the exact power can be computed by setting the precision to \textit{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).

\textbf{3.3.11 Exponentials and logarithms}

void \textbf{arb_log\_ui\_from\_prev} (arb\_t z, const arb\_t x, long prec)  
Sets \( y = b^x \), computed as \( y = (b^{1/q})^p \) if the denominator of \( e = p/q \) is small, and generally as \( y = \exp(e \log b) \).  
Note that this function can get slow if the exponent is extremely large (in such cases \texttt{arb_log()} may be superior).

void \textbf{arb_log\_pow\_fmpq} (arb\_t y, const arb\_t x, const fmpq\_t a, long prec)  
Sets \( y = b^x \), computed as \( y = (b^{1/q})^p \) if the denominator of \( e = p/q \) is small, and generally as \( y = \exp(e \log b) \).  
Note that this function can get slow if the exponent is extremely large (in such cases \texttt{arb_log()} may be superior).

void \textbf{arb_log\_ui\_from\_prev} (arb\_t logk1, ulong k1, arb\_t logk0, ulong k0, long prec)  
Computes \( \log(k_1) \), given \( \log(k_0) \) where \( k_0 < k_1 \). At high precision, this function uses the formula \( \log(k_1) = \log(k_0) + 2 \tanh((k_1 - k_0)/(k_1 + k_0)) \), 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 \textbf{arb_log\_lp} (arb\_t z, const arb\_t x, long prec)  
Sets \( z = \log(1 + x) \), computed accurately when \( x \approx 0 \).

void \textbf{arb\_exp} (arb\_t z, const arb\_t x, long prec)  
Sets \( z = \exp(x) \). Error propagation is done using the following rule: assuming \( x = m \pm r \), the error is largest at \( m + r \), and we have \( \exp(m + r) - \exp(m) = \exp(m)(\exp(r) - 1) \leq r \exp(m + r) \).

void \textbf{arb\_exp\_m1} (arb\_t z, const arb\_t x, long prec)  
Sets \( z = \exp(x) - 1 \), computed accurately when \( x \approx 0 \).

\textbf{3.3.12 Trigonometric functions}

void \textbf{arb\_sin} (arb\_t z, const arb\_t x, long prec)  
void \textbf{arb\_cos} (arb\_t c, const arb\_t x, long prec)

void \textbf{arb\_sin\_cos} (arb\_t s, arb\_t c, const arb\_t x, long prec)  
Sets \( s = \sin(x) \), \( c = \cos(x) \). Error propagation uses the rule \( |\sin(m \pm r) - \sin(m)| \leq \min(r, 2) \).

void \textbf{arb\_sin\_pi} (arb\_t x, const arb\_t x, long prec)
void \textbf{arb\_cos\_pi} (arb\_t c, const arb\_t x, long prec)
void \textbf{arb\_sin\_cos\_pi} (arb\_t s, arb\_t c, const arb\_t x, long prec)  
Sets \( s = \sin(\pi x) \), \( c = \cos(\pi x) \).
void **arb_tan**( arb_t y, const arb_t x, long prec)
Sets \( y = \tan(x) = \sin(x) / \cos(y) \).

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

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

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

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

3.3.13 Inverse trigonometric functions

void **arb_atan_arf**( arb_t z, const arf_t x, long prec)
void **arb_atan**( arb_t z, const arb_t x, long prec)
Sets \( z = \text{atan}(x) \).

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

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

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

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

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

3.3.14 Hyperbolic functions

void **arb_sinh**( arb_t s, const arb_t x, long prec)
void **arb_cosh**( arb_t c, const arb_t x, long prec)
void \texttt{arb\_sinh\_cosh}(\texttt{arb\_t} s, \texttt{arb\_t} c, \texttt{arb\_t} x, \texttt{long} 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 \texttt{arb\_expm1()} to avoid loss of accuracy. Otherwise evaluates $(e^x \pm e^{-x})/2$.

void \texttt{arb\_tanh}(\texttt{arb\_t} y, \texttt{arb\_t} x, \texttt{long} prec)

Sets $y = \tanh(x) = \sinh(x)/\cosh(x)$, evaluated via \texttt{arb\_expm1()} as $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$ if the midpoint of $x$ is negative and as $\tanh(x) = (1 - e^{-2x})/(1 + e^{-2x})$ otherwise.

void \texttt{arb\_coth}(\texttt{arb\_t} y, \texttt{arb\_t} x, \texttt{long} prec)

Sets $y = \coth(x) = \cosh(x)/\sinh(x)$, evaluated using the same strategy as \texttt{arb\_tanh()}.

### 3.3.15 Inverse hyperbolic functions

void \texttt{arb\_atanh}(\texttt{arb\_t} z, \texttt{arb\_t} x, \texttt{long} prec)

Sets $z = \operatorname{atanh}(x)$.

void \texttt{arb\_asinh}(\texttt{arb\_t} z, \texttt{arb\_t} x, \texttt{long} prec)

Sets $z = \operatorname{asinh}(x)$.

void \texttt{arb\_acosh}(\texttt{arb\_t} z, \texttt{arb\_t} x, \texttt{long} prec)

Sets $z = \operatorname{acosh}(x)$. If $x < 1$, the result is an indeterminate interval.

### 3.3.16 Constants

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

void \texttt{arb\_const\_pi}(\texttt{arb\_t} z, \texttt{long} prec)

Computes $\pi$.

void \texttt{arb\_const\_sqrt\_pi}(\texttt{arb\_t} z, \texttt{long} prec)

Computes $\sqrt{\pi}$.

void \texttt{arb\_const\_log\_sqrt2pi}(\texttt{arb\_t} z, \texttt{long} prec)

Computes $\log\sqrt{2\pi}$.

void \texttt{arb\_const\_log2}(\texttt{arb\_t} z, \texttt{long} prec)

Computes $\log(2)$.

void \texttt{arb\_const\_log10}(\texttt{arb\_t} z, \texttt{long} prec)

Computes $\log(10)$.

void \texttt{arb\_const\_euler}(\texttt{arb\_t} z, \texttt{long} prec)

Computes Euler's constant $\gamma = \lim_{k \to \infty}(H_k - \log k)$ where $H_k = 1 + 1/2 + \ldots + 1/k$.

void \texttt{arb\_const\_catalan}(\texttt{arb\_t} z, \texttt{long} 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{long} prec)

Computes $e = \exp(1)$.

void \texttt{arb\_const\_khinchin}(\texttt{arb\_t} z, \texttt{long} prec)

Computes Khinchin's constant $K_0$.

void \texttt{arb\_const\_glaisher}(\texttt{arb\_t} z, \texttt{long} prec)

Computes the Glaisher-Kinkelin constant $A = \exp(1/12 - \zeta'(-1))$.

void \texttt{arb\_const\_apery}(\texttt{arb\_t} z, \texttt{long} prec)

Computes Apery's constant $\zeta(3)$.
3.3.17 Gamma function and factorials

void arb_rising_ui bs (arb_t z, const arb_t x, ulong n, long prec)
void arb_rising_ui rs (arb_t z, const arb_t x, ulong n, ulong step, long prec)
void arb_rising_ui rec (arb_t z, const arb_t x, ulong n, long prec)

void arb_rising2_ui bs (arb_t u, arb_t v, const arb_t x, ulong n, long prec)
void arb_rising2_ui rs (arb_t u, arb_t v, const arb_t x, ulong n, ulong step, long prec)

void arb_fac_ui (arb_t z, ulong n, long prec)
void arb_bin_ui (arb_t z, const arb_t n, ulong k, long prec)
void arb_bin_uiui (arb_t z, ulong n, ulong k, long prec)

void arb_gamma (arb_t z, const arb_t x, long prec)
void arb_gamma_fmpq (arb_t z, const fmpq_t x, long prec)
void arb_gamma_fmpz (arb_t z, const fmpz_t x, long prec)

void arb_lgamma (arb_t z, const arb_t x, long prec)
void arb_rgamma (arb_t z, const arb_t x, long prec)

3.3.18 Zeta function

void arb_zeta_ui_vec_borwein (arb_ptr z, ulong start, long num, ulong step, long prec)

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

Requires \( \text{start} \geq 2 \). For efficiency, the largest \( s \) should be at most about as large as \( \text{prec} \). Arguments approaching \( \text{LONG}_\text{MAX} \) will cause overflows. One should therefore only use this function for \( s \) up to about \( \text{prec} \), and then switch to the Euler product.

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

void \textbf{arb_zeta_ui_asym} (arb_t x, ulong \( s \), long \( \text{prec} \))

Assuming \( s \geq 2 \), approximates \( \zeta(s) \) by \( 1 + 2^{-s} \) along with a correct error bound. We use the following bounds:

for \( s > b \), \( \zeta(s) - 1 < 2^{-b} \), and generally, \( \zeta(s) - (1 + 2^{-s}) < 2^{2-3s/2} \).

void \textbf{arb_zeta_ui_euler_product} (arb_t z, ulong \( s \), long \( \text{prec} \))

Computes \( \zeta(s) \) using the Euler product. This is fast only if \( s \) is large compared to the precision.

Writing \( P(a, b) = \prod_{a \leq p \leq b} (1 - p^{-s}) \), we have \( 1/\zeta(s) = P(a, M) P(M + 1, \infty) \).

To bound the error caused by truncating the product at \( M \), we write \( P(M + 1, \infty) = 1 - \epsilon(s, M) \). Since \( 0 < P(a, M) \leq 1 \), the absolute error for \( \zeta(s) \) is bounded by \( \epsilon(s, M) \).

According to the analysis in [Fil1992], it holds for all \( s \geq 6 \) and \( M \geq 1 \) that \( 1/P(M + 1, \infty) - 1 \leq f(s, M) \equiv 2M^{1-s}/(s/2 - 1) \). Thus, we have \( 1/(1 - \epsilon(s, M)) - 1 \leq f(s, M) \), and expanding the geometric series allows us to conclude that \( \epsilon(M) \leq f(s, M) \).

void \textbf{arb_zeta_ui_bernoulli} (arb_t x, ulong \( s \), long \( \text{prec} \))

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

void \textbf{arb_zeta_ui_borwein_bsplit} (arb_t x, ulong \( s \), long \( \text{prec} \))

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

void \textbf{arb_zeta_ui_vec} (arb_ptr x, ulong \( \text{start} \), long \( \text{num} \), long \( \text{prec} \))

void \textbf{arb_zeta_ui_vec_even} (arb_ptr x, ulong \( \text{start} \), long \( \text{num} \), long \( \text{prec} \))

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

void \textbf{arb_zeta_ui} (arb_t x, ulong \( s \), long \( \text{prec} \))

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

void \textbf{arb_zeta} (arb_t z, const arb_t s, long \( \text{prec} \))

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

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

void \textbf{arb_hurwitz_zeta} (arb_t z, const arb_t s, const arb_t a, long \( \text{prec} \))

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

For computing derivatives with respect to \( s \), use \textbf{arb_poly_zeta_series}().
3.3.19 Bernoulli numbers

void \texttt{arb\_bernoulli\_ui} (\texttt{arb\_t b}, ulong \texttt{n}, long \texttt{prec})

Sets \(b\) to the numerical value of the Bernoulli number \(B_n\), accurate to \(prec\) bits, computed by a division of the exact fraction if \(B_n\) is in the global cache or the exact numerator roughly is larger than \(prec\) bits, and using \texttt{arb\_bernoulli\_ui\_zeta()} otherwise. 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 \texttt{arb\_bernoulli\_ui\_zeta} (\texttt{arb\_t b}, ulong \texttt{n}, long \texttt{prec})

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

To avoid potential infinite recursion, we explicitly call the Euler product implementation of the zeta function. We therefore assume that the precision is small enough and \(n\) large enough for the Euler product to converge rapidly (otherwise this function will effectively hang).

3.3.20 Polylogarithms

void \texttt{arb\_polylog} (\texttt{arb\_t w}, const \texttt{arb\_t s}, const \texttt{arb\_t z}, long \texttt{prec})

void \texttt{arb\_polylog\_si} (\texttt{arb\_t w}, long \texttt{s}, const \texttt{arb\_t z}, long \texttt{prec})

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

3.3.21 Other special functions

void \texttt{arb\_fib\_fmpz} (\texttt{arb\_t z}, const \texttt{fmpz\_t n}, long \texttt{prec})

void \texttt{arb\_fib\_ui} (\texttt{arb\_t z}, ulong \texttt{n}, long \texttt{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 \texttt{ARF\_PREC\_EXACT}.

void \texttt{arb\_agm} (\texttt{arb\_t z}, const \texttt{arb\_t x}, const \texttt{arb\_t y}, long \texttt{prec})

Sets \(z\) to the arithmetic-geometric mean of \(x\) and \(y\).

void \texttt{arb\_chebyshev\_t\_ui} (\texttt{arb\_t a}, ulong \texttt{n}, const \texttt{arb\_t x}, long \texttt{prec})

void \texttt{arb\_chebyshev\_u\_ui} (\texttt{arb\_t a}, ulong \texttt{n}, const \texttt{arb\_t x}, long \texttt{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 \texttt{arb\_chebyshev\_t\_2\_ui} (\texttt{arb\_t a}, \texttt{arb\_t b}, ulong \texttt{n}, const \texttt{arb\_t x}, long \texttt{prec})

void \texttt{arb\_chebyshev\_u\_2\_ui} (\texttt{arb\_t a}, \texttt{arb\_t b}, ulong \texttt{n}, const \texttt{arb\_t x}, long \texttt{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.

3.3.22 Internals for computing elementary functions

void \texttt{arb\_atan\_taylor\_naive} (mp\_ptr y, mp\_limb\_t\* error, mp\_srcptr x, mp\_size\_t xn, ulong \texttt{N}, int alternating)

void \texttt{arb\_atan\_taylor\_rs} (mp\_ptr y, mp\_limb\_t\* error, mp\_srcptr x, mp\_size\_t xn, ulong \texttt{N}, int alternating)

Computes an approximation of \(y = \sum_{k=0}^{N-1} x^{2k+1}/(2k+1)\) (if \texttt{alternating} is 0) or \(y = \sum_{k=0}^{N-1} (-1)^k x^{2k+1}/(2k+1)\) (if \texttt{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 \( error \).

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

Computes an approximation of \( y = \sum_{k=0}^{N-1} \frac{x^k}{k!} \). 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_exp_taylor_rs (mp_ptr y, mp_limb_t * error, mp_srcptr x, mp_size_t xn, ulong N)
```

Used internally for computing exponentials. The naive version uses the forward recurrence, and the rs version uses a division-avoiding rectangular splitting scheme.

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

Attempts to write \( w = x - q \log(2) \) with \( 0 \leq w < \log(2) \), where \( w \) is a fixed-point number with \( wn \) 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 wn)
```

Attempts to write \( w = |x| - q \pi/4 \) with \( 0 \leq w < \pi/4 \), where \( w \) is a fixed-point number with \( wn \) limbs and ulp error \( error \). Returns success.

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

```c
long _arb_exp_taylor_bound (long mag, long prec)
```

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

```c
void arb_exp_arf_bb (arb_t z, const arf_t x, long prec, int ml)
```

Computes the exponential function using the bit-burst algorithm. If \( ml \) is nonzero, the exponential function minus one is computed accurately.

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

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

Computing \( \log(2) \) costs roughly 100-200 multiplications, so is not usually worth the effort at very high precision. However, this function could be improved by using \( \log(2) \) based reduction at precision low enough that the value can be assumed to be cached.
void _arb_exp_sum_bs_simple (fmpz_t T, fmpz_t Q, mp_bitcnt_t *Qexp, const fmpz_t x, mp_bitcnt_t r, long N)
void _arb_exp_sum_bs_powtab (fmpz_t T, fmpz_t Q, mp_bitcnt_t *Qexp, const fmpz_t x, mp_bitcnt_t r, long N)

Computes $T$, $Q$ and $Qexp$ such that $T/(Q^2Qexp) = \sum_{k=1}^N (x/2^r)^k/k!$ using binary splitting. Note that the sum is taken to $N$ inclusive and omits the constant term.

The powtab version precomputes a table of powers of $x$, resulting in slightly higher memory usage but better speed. For best efficiency, $N$ should have many trailing zero bits.

void _arb_atan_sum_bs_simple (fmpz_t T, fmpz_t Q, mp_bitcnt_t *Qexp, const fmpz_t x, mp_bitcnt_t r, long N)
void _arb_atan_sum_bs_powtab (fmpz_t T, fmpz_t Q, mp_bitcnt_t *Qexp, const fmpz_t x, mp_bitcnt_t r, long N)

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

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, long prec)

Computes the arctangent of $x$. Initially, the argument-halving formula

$$\tan(x) = 2 \atan\left(\frac{x}{1 + \sqrt{1 + x^2}}\right)$$

is applied up to 8 times to get a small argument. Then a version of the bit-burst algorithm is used. The functional equation

$$\tan(x) = \atan(p/q) + \atan(w), \quad w = \frac{ qx - p }{ px + q }, \quad p = \lfloor qx \rfloor$$

is applied repeatedly instead of integrating a differential equation for the arctangent, as this appears to be more efficient.

### 3.4 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.

#### 3.4.1 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.

3.4. arb_poly.h – polynomials over the real numbers
3.4.2 Memory management

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

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

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

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

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

3.4.3 Basic manipulation

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, long prec)
   Sets dest to a copy of src, rounded to prec bits.

void arb_poly_set_coeff_si (arb_poly_t poly, long n, long c)

void arb_poly_set_coeff_arb (arb_poly_t poly, long 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, long 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 ≥ 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, long len, long n)

void _arb_poly_set_shift_right (arb_poly_t res, const arb_poly_t poly, long 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, long len, long n)

void _arb_poly_shift_left (arb_poly_t res, const arb_poly_t poly, long n)
   Sets res to poly multiplied by x^n. We require that n is nonnegative.

void arb_poly_truncate (arb_poly_t poly, long n)
   Truncates poly to have length at most n, i.e. degree strictly smaller than n.

long 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.

long 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 \texttt{poly}, so the return value of this function is effectively an upper bound.

### 3.4.4 Conversions

```c
void \texttt{arb_poly_set_fmpz_poly}(\texttt{arb_poly_t} \texttt{poly}, \texttt{const fmpz_poly_t} \texttt{src}, \texttt{long} \texttt{prec})
void \texttt{arb_poly_set_fmpq_poly}(\texttt{arb_poly_t} \texttt{poly}, \texttt{const fmpq_poly_t} \texttt{src}, \texttt{long} \texttt{prec})
void \texttt{arb_poly_set_si}(\texttt{arb_poly_t} \texttt{poly}, \texttt{long} \texttt{src})
```

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

### 3.4.5 Input and output

```c
void \texttt{arb_poly_printd}(\texttt{const arb_poly_t} \texttt{poly}, \texttt{long} \texttt{digits})
```

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

### 3.4.6 Random generation

```c
void \texttt{arb_poly_randtest}(\texttt{arb_poly_t} \texttt{poly}, \texttt{flint_rand_t} \texttt{state}, \texttt{long} \texttt{len}, \texttt{long} \texttt{prec}, \texttt{long} \texttt{mag_bits})
```

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

### 3.4.7 Comparisons

```c
int \texttt{arb_poly_contains}(\texttt{const arb_poly_t} \texttt{poly1}, \texttt{const arb_poly_t} \texttt{poly2})
int \texttt{arb_poly_contains_fmpz_poly}(\texttt{const arb_poly_t} \texttt{poly1}, \texttt{const fmpz_poly_t} \texttt{poly2})
int \texttt{arb_poly_contains_fmpq_poly}(\texttt{const arb_poly_t} \texttt{poly1}, \texttt{const fmpq_poly_t} \texttt{poly2})
```

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

```c
int \texttt{arb_poly_equal}(\texttt{const arb_poly_t} \texttt{A}, \texttt{const arb_poly_t} \texttt{B})
```

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

```c
int \_arb_poly_overlaps(\texttt{arb_srcptr} \texttt{poly1}, \texttt{long} \texttt{len1}, \texttt{arb_srcptr} \texttt{poly2}, \texttt{long} \texttt{len2})
int \texttt{arb_poly_overlaps}(\texttt{const arb_poly_t} \texttt{poly1}, \texttt{const arb_poly_t} \texttt{poly2})
```

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

```c
int \texttt{arb_poly_get_unique_fmpz_poly}(\texttt{fmpz_poly_t} \texttt{z}, \texttt{const arb_poly_t} \texttt{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}.

### 3.4.8 Arithmetic

```c
void \_arb_poly_add(\texttt{arb_ptr} \texttt{C}, \texttt{arb_srcptr} \texttt{A}, \texttt{long} \texttt{lenA}, \texttt{arb_srcptr} \texttt{B}, \texttt{long} \texttt{lenB}, \texttt{long} \texttt{prec})
void \texttt{arb_poly_add}(\texttt{arb_poly_t} \texttt{C}, \texttt{const arb_poly_t} \texttt{A}, \texttt{const arb_poly_t} \texttt{B}, \texttt{long} \texttt{prec})
```

Sets \texttt{C} to the sum of \texttt{A} and \texttt{B}.
void _arb_poly_sub (arb_ptr C, arb_srcptr A, long lenA, arb_srcptr B, long lenB, long prec)
Sets \( C, \max(lenA, \text{lenB}) \) to the difference of \( \{A, \text{lenA}\} \) and \( \{B, \text{lenB}\} \). Allows aliasing of the input and output operands.

void _arb_poly_mullow (arb_poly_t C, const arb_poly_t A, const arb_poly_t B, long prec)
Sets \( C \) to the difference of \( A \) and \( B \).

void _arb_poly_neg (arb_poly_t C, const arb_poly_t A)
Sets \( C \) to the negation of \( A \).

void arb_poly_scalar_mul_2exp_si (arb_ptr C, const arb_poly_t A, long c)
Sets \( C \) to \( A \) multiplied by \( 2^c \).

void _arb_poly_mullow_classical (arb_ptr C, arb_srcptr A, long lenA, arb_srcptr B, long lenB, long n, long prec)
void _arb_poly_mullow_block (arb_ptr C, arb_srcptr A, long lenA, arb_srcptr B, long lenB, long n, long prec)
void _arb_poly_mullow (arb_ptr C, arb_srcptr A, long lenA, arb_srcptr B, long lenB, long n, long 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:

\[
c = \left\lfloor \left( \frac{(e_2 - e_1) + (f_2 + f_1)}{(a_2 - a_1) + (b_2 - b_1)} + \frac{1}{2} \right) \right\rfloor.
\]

This strategy is used because it is simple. It is not optimal in all cases, but will typically give good performance when multiplying two power series with a similar decay rate.

The default algorithm chooses the classical algorithm for short polynomials and the block algorithm for long polynomials.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void _arb_poly_mullow_classical (arb_poly_t C, const arb_poly_t A, const arb_poly_t B, long n, long prec)
void _arb_poly_mullow_ztrunc (arb_poly_t C, const arb_poly_t A, const arb_poly_t B, long n, long prec)
void _arb_poly_mullow_block (arb_poly_t C, const arb_poly_t A, const arb_poly_t B, long n, long prec)
void _arb_poly_mullow (arb_poly_t C, const arb_poly_t A, const arb_poly_t B, long n, long prec)
Sets \( C \) to the product of \( A \) and \( B \), truncated to length \( n \). If the same variable is passed for \( A \) and \( B \), sets \( C \) to the square of \( A \) truncated to length \( n \).

void _arb_poly_mullow_ztrunc (arb_ptr C, arb_srcptr A, long lenA, arb_srcptr B, long lenB, long prec)
Sets \( C, \text{lenA} + \text{lenB} - 1 \) to the product of \( \{A, \text{lenA}\} \) and \( \{B, \text{lenB}\} \). The output is not allowed to be aliased with either of the inputs. We require \( \text{lenA} \geq \text{lenB} > 0 \). This function is implemented as a simple wrapper for _arb_poly_mullow().
If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void arb_poly_mul (arb_poly_t C, const arb_poly_t A, const arb_poly_t B, long prec)
Sets C to the product of A and B. If the same variable is passed for A and B, sets C to the square of A.

void _arb_poly_inv_series (arb_ptr Q, arb_srcptr A, long Alen, long len, long prec)
Sets Q to the power series inverse of [A, Alen]. Uses Newton iteration.

void arb_poly_inv_series (arb_poly_t Q, const arb_poly_t A, long n, long prec)
Sets Q to the power series inverse of A, truncated to length n.

void _arb_poly_div_series (arb_ptr Q, arb_srcptr A, long Alen, arb_srcptr B, long Blen, long n, long prec)

void arb_poly_div_series (arb_poly_t Q, const arb_poly_t A, const arb_poly_t B, long n, long prec)
Sets Q to the power series quotient of A divided by B, truncated to length n.

void _arb_poly_div (arb_ptr Q, arb_srcptr A, long lenA, arb_srcptr B, long lenB, long prec)

void _arb_poly_rem (arb_ptr R, arb_srcptr A, long lenA, arb_srcptr B, long lenB, long prec)

void _arb_poly_divrem (arb_ptr Q, arb_ptr R, arb_srcptr A, long lenA, arb_srcptr B, long lenB, long prec)

int arb_poly_divrem (arb_poly_t Q, arb_poly_t R, const arb_poly_t A, const arb_poly_t B, long prec)
Performs polynomial division with remainder, computing a quotient Q and a remainder R such that \( A = BQ + R \). The implementation reverses the inputs and performs power series division.

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

void _arb_poly_div_root (arb_ptr Q, arb_t R, arb_srcptr A, long len, const arb_t c, long prec)
Divides \( A \) by the polynomial \( x - c \), computing the quotient Q as well as the remainder \( R = f(c) \).

### 3.4.9 Composition

void _arb_poly_compose_horner (arb_ptr res, arb_srcptr poly1, long len1, arb_srcptr poly2, long len2, long prec)

void arb_poly_compose_horner (arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, long prec)

void _arb_poly_compose_divconquer (arb_ptr res, arb_srcptr poly1, long len1, arb_srcptr poly2, long len2, long prec)

void arb_poly_compose_divconquer (arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, long len2, long prec)

void _arb_poly_compose (arb_ptr res, arb_srcptr poly1, long len1, arb_srcptr poly2, long len2, long prec)

void arb_poly_compose (arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, long len2, long prec)
Sets res to the composition \( h(x) = f(g(x)) \) where \( f \) is given by poly1 and \( g \) is given by poly2, respectively using Horner’s rule, divide-and-conquer, and an automatic choice between the two algorithms. The 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, long len1, arb_srcptr poly2, long len2, long n, long prec)

void arb_poly_compose_series_horner (arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, long n, long prec)
arb_poly_compose_series_brent_kung (arb_ptr res, arb_srcptr poly1, long len1, arb_srcptr poly2, long len2, long n, long prec)

arb_poly_revert_series_lagrange (arb_ptr h, arb_srcptr f, long flen, long n, long prec)

arb_poly_revert_series_lagrange (arb_poly_t h, const arb_poly_t f, long n, long prec)

arb_poly_revert_series_newton (arb_poly_t h, arb_srcptr f, long flen, long n, long prec)

arb_poly_revert_series_lagrange_fast (arb_ptr h, arb_srcptr f, long flen, long n, long prec)

arb_poly_revert_series (arb_poly_t h, const arb_poly_t f, long n, long prec)

void _arb_poly_compose_series_brent_kung (arb_ptr res, arb_srcptr poly1, long len1, arb_srcptr poly2, long len2, long n, long prec)

void _arb_poly_revert_series_lagrange (arb_ptr h, arb_srcptr f, long flen, long n, long prec)

void _arb_poly_revert_series_lagrange (arb_poly_t h, const arb_poly_t f, long n, long prec)

void _arb_poly_revert_series_newton (arb_poly_t h, arb_srcptr f, long flen, long n, long prec)

void _arb_poly_revert_series_lagrange_fast (arb_ptr h, arb_srcptr f, long flen, long n, long prec)

void _arb_poly_revert_series (arb_poly_t h, const arb_poly_t f, long n, long prec)

void arb_poly_compose_series (arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, long n, long prec)

void arb_poly_revert_series (arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, long n, long prec)

void arb_poly_evaluate (arb_t y, const arb_poly_t poly1, const arb_poly_t poly2, long n, long prec)

Sets res to the power series composition \( h(x) = f(g(x)) \) truncated to order \( O(x^n) \) where \( f \) is given by \( \text{poly}1 \) and \( g \) is given by \( \text{poly}2 \), respectively using Horner’s rule, the Brent-Kung baby step-giant step algorithm, and an automatic choice between the two algorithms. We require that the constant term in \( g(x) \) is exactly zero. The underscore methods do not support aliasing of the output with either input polynomial.

3.4.10 Evaluation

void _arb_poly_evaluate_horner (arb_t y, arb_srcptr f, long len, const arb_t x, long prec)

void _arb_poly_evaluate_horner (arb_t y, const arb_poly_t f, const arb_t x, long prec)

void _arb_poly_evaluate_rectangular (arb_t y, arb_srcptr f, long len, const arb_t x, long prec)

void _arb_poly_evaluate_rectangular (arb_t y, const arb_poly_t f, const arb_t x, long prec)

void _arb_poly_evaluate (arb_t y, arb_srcptr f, long len, const arb_t x, long prec)

void _arb_poly_evaluate (arb_t y, const arb_poly_t f, const arb_t x, long prec)

void arb_poly_evaluate (arb_t y, const arb_poly_t f, const arb_t x, long prec)

void arb_poly_evaluate (arb_t y, arb_srcptr f, long len, const acb_t x, long prec)

void arb_poly_evaluate (arb_t y, const arb_poly_t f, const acb_t x, long prec)

void arb_poly_evaluate (arb_t y, arb_srcptr f, long len, const acb_t x, long prec)

void arb_poly_evaluate (arb_t y, const arb_poly_t f, const acb_t x, long prec)

void arb_poly_evaluate_acb (arb_t y, arb_srcptr f, long len, const acb_t x, long prec)

void arb_poly_evaluate_acb (arb_t y, const arb_poly_t f, const acb_t x, long prec)

void arb_poly_evaluate_acb (arb_t y, arb_srcptr f, long len, const acb_t x, long prec)

void arb_poly_evaluate_acb (arb_t y, const arb_poly_t f, const acb_t x, long prec)

void arb_poly_evaluate_acb (arb_t y, arb_srcptr f, long len, const acb_t x, long prec)
3.4.11 Product trees

void _arb_poly_product_roots (arb_ptr poly, arb_srcptr xs, long n, long prec)
Generates the polynomial \((\sum_{i=0}^{n-1} r_i (x - x_0)) \cdots (x - x_{n-1})\).

void _arb_poly_tree_alloc (long len)
Returns an initialized data structure capable of representing a remainder tree (product tree) of \(\text{len}\) roots.

void _arb_poly_tree_free (arb_ptr * tree, long len)
Deallocates a tree structure as allocated using _arb_poly_tree_alloc.

void _arb_poly_tree_build (arb_ptr * tree, arb_srcptr roots, long len, long prec)
Constructs a product tree from a given array of \(\text{len}\) roots. The tree structure must be pre-allocated to the specified length using _arb_poly_tree_alloc().

3.4.12 Multipoint evaluation

void _arb_poly_evaluate2_acb (arb_t y, const arb_poly_t f, const arb_t x, long prec)
Sets \(y = f(x)\) where \(x\) is a complex number, evaluating the polynomial respectively using Horner’s rule, rectangular splitting, and an automatic algorithm choice.

void _arb_poly_evaluate2_acb_horner (arb_t y, arb_t z, const arb_poly_t f, const arb_t x, long 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.

3.4. arb_poly.h – polynomials over the real numbers
void `arb_poly_evaluate_vec_iter` (arb_ptr ys, const arb_poly_t poly, arb_srcptr xs, long n, long 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, long plen, arb_ptr * tree, long len, long prec)
void `_arb_poly_evaluate_vec_fast` (arb_ptr ys, arb_srcptr poly, long plen, arb_srcptr xs, long n, long prec)
Evaluates the polynomial simultaneously at \( n \) given points, using fast multipoint evaluation.

3.4.13 Interpolation

void `_arb_poly_interpolate_newton` (arb_ptr poly, arb_srcptr xs, arb_srcptr ys, long n, long prec)
void `arb_poly_interpolate_newton` (arb_poly_t poly, arb_srcptr xs, arb_srcptr ys, long n, long prec)
Recovers the unique polynomial of length at most \( n \) that interpolates the given \( x \) and \( y \) values. This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

void `_arb_poly_interpolate_barycentric` (arb_ptr poly, arb_srcptr xs, arb_srcptr ys, long n, long prec)
void `arb_poly_interpolate_barycentric` (arb_poly_t poly, arb_srcptr xs, arb_srcptr ys, long n, long prec)
Recovers the unique polynomial of length at most \( n \) that interpolates the given \( x \) and \( y \) values. This implementation uses the barycentric form of Lagrange interpolation.

void `_arb_poly_interpolation_weights` (arb_ptr w, arb_ptr * tree, long len, long prec)
void `_arb_poly_interpolate_fast_precomp` (arb_ptr poly, arb_srcptr ys, arb_ptr * tree, arb_srcptr weights, long len, long prec)
void `_arb_poly_interpolate_fast` (arb_ptr poly, arb_srcptr xs, arb_srcptr ys, long len, long prec)
void `arb_poly_interpolate_fast` (arb_poly_t poly, arb_srcptr xs, arb_srcptr ys, long n, long 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.

3.4.14 Differentiation

void `_arb_poly_derivative` (arb_ptr res, arb_srcptr poly, long len, long prec)
Sets \( \{res, len - 1\} \) to the derivative of \( \{poly, len\} \). Allows aliasing of the input and output.

void `arb_poly_derivative` (arb_poly_t res, const arb_poly_t poly, long prec)
Sets \( res \) to the derivative of \( poly \).

void `_arb_poly_integral` (arb_ptr res, arb_srcptr poly, long len, long prec)
Sets \( \{res, len\} \) to the integral of \( \{poly, len - 1\} \). Allows aliasing of the input and output.

void `arb_poly_integral` (arb_poly_t res, const arb_poly_t poly, long prec)
Sets \( res \) to the integral of \( poly \).

3.4.15 Transforms

void `_arb_poly_borel_transform` (arb_ptr res, arb_srcptr poly, long len, long prec)
3.4. tyr_poly.h – polynomials over the real numbers

void arb_poly_borel_transform (arb_poly_t res, const arb_poly_t poly, long prec)
Computes the Borel transform of the input polynomial, mapping \( \sum_k a_k x^k \) to \( \sum_k (a_k / k!) x^k \). The underscore method allows aliasing.

void _arb_poly_inv_borel_transform (arb_ptr res, arb_srcptr poly, long len, long prec)
void arb_poly_inv_borel_transform (arb_poly_t res, const arb_poly_t poly, long prec)
Computes the inverse Borel transform of the input polynomial, mapping \( \sum_k a_k x^k \) to \( \sum_k a_k k! x^k \). The underscore method allows aliasing.

void _arb_poly_binomial_transform_basecase (arb_ptr b, arb_srcptr a, long alen, long len, long prec)
void arb_poly_binomial_transform_basecase (arb_poly_t b, const arb_poly_t a, long len, long prec)
void _arb_poly_binomial_transform_convolution (arb_ptr b, arb_srcptr a, long alen, long len, long prec)
void arb_poly_binomial_transform_convolution (arb_poly_t b, const arb_poly_t a, long len, long prec)
void _arb_poly_binomial_transform (arb_ptr b, arb_srcptr a, long alen, long len, long prec)
void arb_poly_binomial_transform (arb_poly_t b, const arb_poly_t a, long len, long prec)

Computes the binomial transform of the input polynomial, truncating the output to length \( \text{len} \). The binomial transform maps the coefficients \( a_k \) in the input polynomial to the coefficients \( b_k \) in the output polynomial via \( b_n = \sum_{k=0}^{n} (-1)^k \binom{n}{k} a_k \). 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 basecase version evaluates coefficients one by one from the definition, generating the binomial coefficients by a recurrence relation.

The convolution version uses the identity \( T(f(x)) = B^{-1}(e^x B(f(-x))) \) where \( T \) denotes the binomial transform operator and \( B \) denotes the Borel transform operator. This only costs a single polynomial multiplication, plus some scalar operations.

The default version automatically chooses an algorithm.

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

3.4.16 Powers and elementary functions

void _arb_poly_pow_ui_trunc_binexp (arb_ptr res, arb_srcptr f, long flen, ulong exp, long len, long prec)
Sets \( \{ \text{res}, \text{len} \} \) to \( \{ f, \text{flen} \} \) raised to the power \( \text{exp} \), truncated to length \( \text{len} \). Requires that \( \text{len} \) is no longer than the length of the power as computed without truncation (i.e. no zero-paddings is performed). Does not support aliasing of the input and output, and requires that \( \text{flen} \) and \( \text{len} \) are positive. Uses binary exponentiation.

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

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

void arb_poly_pow_ui (arb_poly_t res, const arb_poly_t poly, ulong exp, long prec)
Sets \( \text{res} \) to \( \text{poly} \) raised to the power \( \text{exp} \).
void _arb_poly_pow_series (arb_ptr h, arb_srcptr f, longflen, arb_srcptr g, long glen, long len, long prec)
Sets \( \{ h, len \} \) to the power series \( f(x)^g(x) = \exp(g(x) \log f(x)) \) truncated to length \( len \). This function detects
special cases such as \( g \) being an exact small integer or \( \pm \frac{1}{2} \), and computes such powers more efficiently. This
function does not support aliasing of the output with either of the input operands. It requires that all lengths are
positive, and assumes that \( flen \) and \( glen \) do not exceed \( len \).

void arb_poly_pow_series (arb_poly_t h, const arb_poly_t f, const arb_poly_t g, long len, long prec)
Sets \( h \) to the power series \( f(x)^g(x) = \exp(g(x) \log f(x)) \) truncated to length \( len \). This function detects special
cases such as \( g \) being an exact small integer or \( \pm \frac{1}{2} \), and computes such powers more efficiently.

void _arb_poly_pow_arb_series (arb_ptr h, arb_srcptr f, longflen, const arb_t g, long len, long 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 \( \pm \frac{1}{2} \), and computes such powers more efficiently. This function
does not support aliasing of the output with either of the input operands. It requires that all lengths are positive,
and assumes that \( flen \) does not exceed \( len \).

void arb_poly_pow_arb_series (arb_poly_t h, const arb_poly_t f, const arb_t g, long len, long prec)
Sets \( h \) to the power series \( f(x)^g = \exp(g \log f(x)) \) truncated to length \( len \).

void _arb_poly_rsqrt_series (arb_ptr g, arb_srcptr h, long hlen, long n, long prec)
void arb_poly_rsqrt_series (arb_poly_t g, const arb_poly_t h, long n, long prec)
Sets \( g \) to the power series square root of \( h \), truncated to length \( n \). Uses division-free Newton iteration for the
reciprocal square root, followed by a multiplication.

The underscore method does not support aliasing of the input and output arrays. It requires that \( hlen \) and \( n \) are
greater than zero.

void _arb_poly_rsqrt_series (arb_ptr g, arb_srcptr h, long hlen, long n, long prec)
void arb_poly_rsqrt_series (arb_poly_t g, const arb_poly_t h, long n, long prec)
Sets \( g \) to the reciprocal power series square root of \( h \), truncated to length \( n \). Uses division-free Newton iteration.

The underscore method does not support aliasing of the input and output arrays. It requires that \( hlen \) and \( n \) are
greater than zero.

void _arb_poly_log_series (arb_ptr res, arb_srcptr f, longflen, long n, long prec)
void arb_poly_log_series (arb_poly_t res, const arb_poly_t f, long n, long prec)
Sets \( res \) to the power series logarithm of \( f \), truncated to length \( n \). Uses the formula \( \log(f(x)) = \int f'(x)/f(x)dx \),
adding the logarithm of the constant term in \( f \) as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that \( flen \) and \( n \) are
greater than zero.

void _arb_poly_atan_series (arb_ptr res, arb_srcptr f, longflen, long n, long prec)
void arb_poly_atan_series (arb_poly_t res, const arb_poly_t f, long n, long prec)
void _arb_poly_asin_series (arb_ptr res, arb_srcptr f, longflen, long n, long prec)
void arb_poly_asin_series (arb_poly_t res, const arb_poly_t f, long n, long prec)
void _arb_poly_acos_series (arb_ptr res, arb_srcptr f, longflen, long n, long prec)
void arb_poly_acos_series (arb_poly_t res, const arb_poly_t f, long n, long prec)
Sets \( res \) respectively to the power series inverse tangent, inverse sine and inverse cosine of \( f \), truncated to length \( n \).
Uses the formulas
\[
\begin{align*}
\tan^{-1}(f(x)) &= \int f'(x)/(1 + f(x)^2)\,dx, \\
\sin^{-1}(f(x)) &= \int f'(x)/(1 - f(x)^2)^{1/2}\,dx, \\
\cos^{-1}(f(x)) &= -\int f'(x)/(1 - f(x)^2)^{1/2}\,dx,
\end{align*}
\]
adding the inverse function of the constant term in \(f\) as the constant of integration.

The underscore methods supports aliasing of the input and output arrays. They require that \(flen \) and \(n\) are greater than zero.

```c
void __arb_poly_exp_series_basecase (arb_ptr f, arb_srcptr h, long hlen, long n, long 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 __arb_poly_sin_cos_series_basecase (arb_ptr s, arb_ptr c, arb_srcptr h, long hlen, long n, long prec)
```

Sets \(s\) and \(c\) to the power series sine and cosine of \(h\), computed simultaneously.

The basecase version uses a simple recurrence for the coefficients, requiring \(O(nm)\) operations where \(m\) is the length of \(h\).

The tangent version uses the tangent half-angle formulas to compute the sine and cosine via `__arb_poly_tan_series()`. This requires \(O(M(n))\) operations. When \(h = h_0 + h_1\) where the constant term \(h_0\) is nonzero, the evaluation is done as
\[
\begin{align*}
sin(h_0 + h_1) &= \cos(h_0) \sin(h_1) + \sin(h_0) \cos(h_1), \\
\cos(h_0 + h_1) &= \cos(h_0) \cos(h_1) - \sin(h_0) \sin(h_1),
\end{align*}
\]
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 underscore methods support aliasing and require the lengths to be nonzero.

```c
void __arb_poly_sin_series (arb_ptr s, arb_srcptr h, long hlen, long n, long prec)
```
void arb_poly_sin_series (arb_poly_t s, const arb_poly_t h, long n, long prec)
void _arb_poly_sin_series (arb_ptr s, arb_srcptr h, long hlen, long n, long prec)

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.

void arb_poly_cos_series (arb_poly_t c, const arb_poly_t h, long n, long prec)
void _arb_poly_cos_series (arb_ptr c, arb_srcptr h, long hlen, long n, long prec)

Sets c 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.

3.4.17 Gamma function and factorials

void _arb_poly_gamma_series (arb_ptr res, arb_srcptr h, long hlen, long n, long prec)
void arb_poly_gamma_series (arb_poly_t res, const arb_poly_t h, long n, long prec)
void _arb_poly_rgamma_series (arb_ptr res, arb_srcptr h, long hlen, long n, long prec)
void arb_poly_rgamma_series (arb_poly_t res, const arb_poly_t h, long n, long prec)
void _arb_poly_lgamma_series (arb_ptr res, arb_srcptr h, long hlen, long n, long prec)
void arb_poly_lgamma_series (arb_poly_t res, const arb_poly_t h, long n, long prec)

Sets res to the series expansion of \( \Gamma(h(x)), 1/\Gamma(h(x)), \) or \( \log \Gamma(h(x)) \), truncated to length n.

These functions first generate the Taylor series at the constant term of h, and then call
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, long flen, ulong r, long trunc, long prec)
void arb_poly_rising_ui_series (arb_poly_t res, const arb_poly_t f, ulong r, long trunc, long 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.

3.4.18 Zeta function

void arb_poly_zeta_series (arb_poly_t res, const arb_poly_t s, const arb_t a, int deflate, long n,
                          long 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, long hlen, long n, long prec)

void _arb_poly_riemann_siegel_z_series (arb_ptr res, arb_srcptr h, long hlen, long n, long prec)

Sets res to the series expansion of the Riemann-Siegel theta function

$$\theta(h) = \arg \left( \Gamma \left( \frac{2ih + 1}{4} \right) \right) - \log \frac{\pi}{2} h$$

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, long hlen, long n, long 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.

### 3.4.19 Root-finding

void _arb_poly_newton_convergence_factor (arf_t convergence_factor, arb_srcptr poly, long len, const arb_t convergence_interval, long prec)

Given an interval $I$ specified by $\text{convergence\_interval}$, evaluates a bound for $C = \sup_{t,u \in I} \frac{1}{2} |f''(t)|/|f'(u)|$, where $f$ is the polynomial defined by the coefficients $\{\text{poly}, \text{len}\}$. The bound is obtained by evaluating $f'(I)$ and $f''(I)$ directly. If $f$ has large coefficients, $I$ must be extremely precise in order to get a finite factor.

int _arb_poly_newton_step (arb_t xnew, arb_srcptr poly, long len, const arb_t x, const arb_t convergence_interval, const arf_t convergence_factor, long prec)

Performs a single step with Newton’s method.

The input consists of the polynomial $f$ specified by the coefficients $\{\text{poly}, \text{len}\}$, an interval $x = [m - r, m + r]$ known to contain a single root of $f$, an interval $I$ ($\text{convergence\_interval}$) containing $x$ with an associated bound ($\text{convergence\_factor}$) for $C = \sup_{t,u \in I} \frac{1}{2} |f''(t)|/|f'(u)|$, and a working precision $\text{prec}$.

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 $\text{prec}$ bits, and the rounding error during this evaluation is accounted for in the output. We now check that $x' \in I$ and $m' < m$. If both conditions are satisfied, we set $xnew$ to $x'$ and return nonzero. If either condition fails, we set $xnew$ to $x$ and return zero, indicating that no progress was made.

void _arb_poly_newton_refine_root (arb_t r, arb_srcptr poly, long len, const arb_t start, const arb_t convergence_interval, const arf_t convergence_factor, long eval_extra_prec, long 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: $\text{prec}$ is the target accuracy and $\text{eval\_extra\_prec}$ 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).
3.4.20 Other special polynomials

void _arb_poly_swinnerton_dyer_ui (arb_ptr poly, ulong n, long trunc, long prec)

void arb_poly_swinnerton_dyer_ui (arb_poly_t poly, ulong n, long prec)

Computes the Swinnerton-Dyer polynomial $S_n$, which has degree $2^n$ and is the rational minimal polynomial of the sum of the square roots of the first $n$ prime numbers.

If prec is set to zero, a precision is chosen automatically such that arb_poly_get_unique_fmpz_poly() should be successful. Otherwise a working precision of prec bits is used.

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

3.5 arb_mat.h – matrices over the real numbers

An arb_mat_t represents a dense matrix over the real numbers, implemented as an array of entries of type arb_struct.

The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

3.5.1 Types, macros and constants

arb_mat_struct

arb_mat_t

Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An arb_mat_t is defined as an array of length one of type arb_mat_struct, permitting an arb_mat_t to be passed by reference.

arb_mat_entry (mat, i, j)

Macro giving a pointer to the entry at row i and column j.

arb_mat_nrows (mat)

Returns the number of rows of the matrix.

arb_mat_ncols (mat)

Returns the number of columns of the matrix.

3.5.2 Memory management

void arb_mat_init (arb_mat_t mat, long r, long c)

Initializes the matrix, setting it to the zero matrix with r rows and c columns.

void arb_mat_clear (arb_mat_t mat)

Clears the matrix, deallocating all entries.

3.5.3 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_fmpq_mat** (arb_mat_t dest, const fmpq_mat_t src, long prec)

Sets dest to src. The operands must have identical dimensions.

### 3.5.4 Random generation

void **arb_mat_randtest** (arb_mat_t mat, flint_rand_t state, long prec, long mag_bits)

Sets mat to a random matrix with up to prec bits of precision and with exponents of width up to mag_bits.

### 3.5.5 Input and output

void **arb_mat_printd** (const arb_mat_t mat, long digits)

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

### 3.5.6 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.

### 3.5.7 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.

### 3.5.8 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.

### 3.5.9 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, long 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, long 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, long prec)

void **arb_mat_mul_threaded** (arb_mat_t C, const arb_mat_t A, const arb_mat_t B, long 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** (arb_mat_t res, const arb_mat_t mat1, const arb_mat_t mat2, long prec)

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

3.5.10 Scalar arithmetic

void **arb_mat_scalar_mul_2exp_si** (arb_mat_t B, const arb_mat_t A, long c)

Sets B to A multiplied by $2^c$.

void **arb_mat_scalar_addmul_si** (arb_mat_t B, const arb_mat_t A, long c, long prec)

void **arb_mat_scalar_addmul_fmpz** (arb_mat_t B, const arb_mat_t A, const fmpz_t c, long prec)

void **arb_mat_scalar_addmul_arb** (arb_mat_t B, const arb_mat_t A, const arb_t c, long prec)

Sets B to $B + A \times c$.

void **arb_mat_scalar_mul_si** (arb_mat_t B, const arb_mat_t A, long c, long prec)

void **arb_mat_scalar_mul_fmpz** (arb_mat_t B, const arb_mat_t A, const fmpz_t c, long prec)

void **arb_mat_scalar_mul_arb** (arb_mat_t B, const arb_mat_t A, const arb_t c, long prec)

Sets B to $A \times c$.

void **arb_mat_scalar_div_si** (arb_mat_t B, const arb_mat_t A, long c, long prec)

void **arb_mat_scalar_div_fmpz** (arb_mat_t B, const arb_mat_t A, const fmpz_t c, long prec)

void **arb_mat_scalar_div_arb** (arb_mat_t B, const arb_mat_t A, const arb_t c, long prec)

Sets B to $A/c$.

3.5.11 Gaussian elimination and solving

int **arb_mat_lu** (long * perm, arb_mat_t LU, const arb_mat_t A, long 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 \texttt{arb\_mat\_solve\_lu\_precomp} (\texttt{arb\_mat\_t} \texttt{X}, \texttt{const long} \* \texttt{perm}, \texttt{const arb\_mat\_t} \texttt{LU}, \texttt{const arb\_mat\_t} \texttt{B}, \texttt{long} \texttt{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 \( LU \).

\begin{verbatim}
int \texttt{arb\_mat\_solve} (\texttt{arb\_mat\_t} \texttt{X}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{const arb\_mat\_t} \texttt{B}, \texttt{long} \texttt{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.
\end{verbatim}

\begin{verbatim}
int \texttt{arb\_mat\_inv} (\texttt{arb\_mat\_t} \texttt{X}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{long} \texttt{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.
\end{verbatim}

\begin{verbatim}
void \texttt{arb\_mat\_det} (\texttt{arb\_t} \texttt{det}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{long} \texttt{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.
\end{verbatim}

### 3.5.12 Characteristic polynomial

\begin{verbatim}
void \_\texttt{arb\_mat\_charpoly} (\texttt{arb\_ptr} \texttt{cp}, \texttt{const arb\_mat\_t} \texttt{mat}, \texttt{long} \texttt{prec})
\end{verbatim}

\begin{verbatim}
void \texttt{arb\_mat\_charpoly} (\texttt{arb\_poly\_t} \texttt{cp}, \texttt{const arb\_mat\_t} \texttt{mat}, \texttt{long} \texttt{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.
\end{verbatim}

### 3.5.13 Special functions

\begin{verbatim}
void \texttt{arb\_mat\_exp} (\texttt{arb\_mat\_t} \texttt{B}, \texttt{const arb\_mat\_t} \texttt{A}, \texttt{long} \texttt{prec})
Sets \( B \) to the exponential of the matrix \( A \), defined by the Taylor series

\[
\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.
\]

The function is evaluated as \( \exp(A/2^r)^{2^r} \), where \( r \) is chosen to give rapid convergence of the Taylor series. The series is evaluated using rectangular splitting. If \( \|A/2^r\| \leq c \) and \( N \geq 2c \), we bound the entrywise error when truncating the Taylor series before term \( N \) by \( 2c^N/N! \).
\end{verbatim}

### 3.6 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.

### 3.6.1 Types, macros and constants

**arb_calc_func_t**

Typedef for a pointer to a function with signature:

```c
int func(arb_ptr out, const arb_t inp, void * param, long order, long 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.

### 3.6.2 Debugging

**int arb_calc_verbose**

If set, enables printing information about the calculation to standard output.

### 3.6.3 Subdivision-based root finding

**arf_interval_struct**

**arf_interval_interval_t**

An `arf_interval_struct` consists of a pair of `arf_struct`, representing an interval used for subdivision-based root-finding. An `arf_interval_t` is defined as an array of length one of type `arf_interval_struct`, permitting an `arf_interval_t` to be passed by reference.

**arf_interval_ptr**

Alias for `arf_interval_struct *`, used for vectors of intervals.

**arf_interval_ssrcptr**

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 (long n)
void arf_interval_vec_clear (arf_interval_ptr v, long 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, long prec)
void arf_interval_printd (const arf_interval_t v, long n)

Helper functions for endpoint-based intervals.

long arb_calc_isolate_roots (arf_interval_ptr * found, int ** flags, arb_calc_func_t func, void * param, const arf_interval_t interval, long maxdepth, long maxeval, long maxfound, long 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 \( \text{interval} \) to \( \text{found} \) and corresponding flags to \( \text{flags} \), returning the integer \( n \). The output has the following properties:

- The function has no roots on \( \text{interval} \) 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 \( \text{interval} \) 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 \( \text{maxdepth} \) recursive subdivisions are attempted. The smallest details that can be distinguished are therefore about \( 2^{-\text{maxdepth}} \) times the width of \( \text{interval} \). A typical, reasonable value might be between 20 and 50.
- If the total number of tested subintervals exceeds \( \text{maxeval} \), the algorithm is terminated and any untested subintervals are added to the output. The total number of calls to \( \text{func} \) is thereby restricted to a small multiple of \( \text{maxeval} \) (the actual count can be slightly higher depending on implementation details). A typical, reasonable value might be between 100 and 100000.
- The algorithm terminates if \( \text{maxfound} \) roots have been isolated. In particular, setting \( \text{maxfound} \) to 1 can be used to locate just one root of the function even if there are numerous roots. To try to find all roots, \( \text{LONG\_MAX} \) may be passed.

The argument \( \text{prec} \) denotes the precision used to evaluate the function. It is possibly also used for some other arithmetic operations performed internally by the algorithm. Note that it probably does not make sense for \( \text{maxdepth} \) to exceed \( \text{prec} \).

Warning: it is assumed that subdivision points of \( \text{interval} \) can be represented exactly as floating-point numbers in memory. Do not pass \( 1 \pm 2^{-100} \) as input.

int arb_calc_refine_root_bisect (arf_interval_t r, arb_calc_func_t func, void * param, const arf_interval_t start, long iter, long prec)

Given an interval \( \text{start} \) known to contain a single root of \( \text{func} \), refines it using \( \text{iter} \) bisection steps. The algorithm can return a failure code if the sign of the function at an evaluation point is ambiguous. The output \( r \) is set to a valid isolating interval (possibly just \( \text{start} \)) even if the algorithm fails.
3.6.4 Newton-based root finding

```c
void arb_calc_newton_conv_factor (arf_t conv_factor, arb_calc_func_t func, void * param, const arb_t conv_region, long prec)
```

Given an interval $I$ specified by `conv_region`, evaluates a bound for $C = \sup_{t,u \in I} \left\{ \frac{1}{2} \mid f''(t) \mid / \mid f'(u) \mid \right\}$, where $f$ is the function specified by `func` and `param`. The bound is obtained by evaluating $f'(I)$ and $f''(I)$ directly. If $f$ is ill-conditioned, $I$ may need to be extremely precise in order to get an effective, finite bound for $C$.

```c
int arb_calc_newton_step (arb_t xnew, arb_calc_func_t func, void * param, const arb_t x, const arb_t conv_region, const arb_t conv_factor, long prec)
```

Performs a single step with an interval version of Newton’s method. The input consists of the function $f$ specified by `func` and `param`, a ball $x = [m - r, m + r]$ known to contain a single root of $f$, a ball $I$ (`conv_region`) containing $x$ with an associated bound (`conv_factor`) for $C = \sup_{t,u \in I} \frac{1}{2} \mid f''(t) \mid / \mid f'(u) \mid$, and a working precision `prec`.

The Newton update consists of setting $x' = [m' - r', m' + r']$ where $m' = m - f(m)/f'(m)$ and $r' = Cr^2$. The expression $m - f(m)/f'(m)$ is evaluated using ball arithmetic at a working precision of `prec` bits, and the rounding error during this evaluation is accounted for in the output. We now check that $x' \in I$ and $r' < r$. If both conditions are satisfied, we set `xnew` to $x'$ and return `ARB_CALC_SUCCESS`. If either condition fails, we set `xnew` to $x$ and return `ARB_CALC_NO_CONVERGENCE`, indicating that no progress is made.

```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 arf_t conv_factor, long eval_extra_prec, long prec)
```

Refines a precise estimate of a single root of a function to high precision by performing several Newton steps, using nearly optimally chosen doubling precision steps.

The inputs are defined as for `arb_calc_newton_step`, except for the precision parameters: `prec` is the target accuracy and `eval_extra_prec` is the estimated number of guard bits that need to be added to evaluate the function accurately close to the root (for example, if the function is a polynomial with large coefficients of alternating signs and Horner’s rule is used to evaluate it, the extra precision should typically be approximately the bit size of the coefficients).

This function returns `ARB_CALC_SUCCESS` if all attempted Newton steps are successful (note that this does not guarantee that the computed root is accurate to `prec` bits, which has to be verified by the user), only that it is more accurate than the starting ball.

On failure, `ARB_CALC_IMPRECISE_INPUT` or `ARB_CALC_NO_CONVERGENCE` may be returned. In this case, `r` is set to a ball for the root which is valid but likely does have full accuracy (it can possibly just be equal to the starting ball).

3.7 acb.h – complex numbers

An `acb_t` represents a complex number with error bounds. An `acb_t` consists of a pair of real number balls of type `arb_struct`, representing the real and imaginary part with separate error bounds.

An `acb_t` thus represents a rectangle $[m_1 - r_1, m_1 + r_1] + [m_2 - r_2, m_2 + r_2]i$ in the complex plane. This is used instead of a disk or square representation (consisting of a complex floating-point midpoint with a single radius), since it allows implementing many operations more conveniently by splitting into ball operations on the real and imaginary parts. It also allows tracking when complex numbers have an exact (for example exactly zero) real part and an inexact imaginary part, or vice versa.

The interface for the `acb_t` type is slightly less developed than that for the `arb_t` type. In many cases, the user can easily perform missing operations by directly manipulating the real and imaginary parts.
3.7.1 Types, macros and constants

**acb_struct**

**acb_t**
An `acb_struct` consists of a pair of `arb_struct`s. An `acb_t` is defined as an array of length one of type `acb_struct`, permitting an `acb_t` to be passed by reference.

**acb_ptr**
Alias for `acb_struct *`, used for vectors of numbers.

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

**acb_realref** (x)
Macro returning a pointer to the real part of x as an `arb_t`.

**acb_imagref** (x)
Macro returning a pointer to the imaginary part of x as an `arb_t`.

3.7.2 Memory management

void **acb_init** (acb_t x)
Initializes the variable x for use, and sets its value to zero.

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

acb_ptr **acb_vec_init** (long n)
Returns a pointer to an array of n initialized `acb_struct`s.

void **acb_vec_clear** (acb_ptr v, long n)
Clears an array of n initialized `acb_struct`s.

3.7.3 Basic manipulation

int **acb_is_zero** (const acb_t z)
Returns nonzero iff z is zero.

int **acb_is_one** (const acb_t z)
Returns nonzero iff z is exactly 1.

int **acb_is_exact** (const acb_t z)
Returns nonzero iff z is exact.

int **acb_is_int** (const acb_t z)
Returns nonzero iff z is an exact integer.

void **acb_zero** (acb_t z)

void **acb_one** (acb_t z)

void **acb_onei** (acb_t z)
Sets z respectively to 0, 1, i = √−1.

void **acb_set** (acb_t z, const acb_t x)

void **acb_set_ui** (acb_t z, long x)

void **acb_set_si** (acb_t z, long x)
 void \texttt{acb\_set\_fmpz} (\texttt{acb\_t} \texttt{z}, \texttt{const fmpz\_t} \texttt{x})
 void \texttt{acb\_set\_arb} (\texttt{acb\_t} \texttt{z}, \texttt{const arb\_t} \texttt{c})
 \hspace{1em} Sets \texttt{z} to the value of \texttt{x}.
 void \texttt{acb\_set\_fmpq} (\texttt{acb\_t} \texttt{z}, \texttt{const fmpq\_t} \texttt{x}, \texttt{long} \texttt{prec})
 void \texttt{acb\_set\_round} (\texttt{acb\_t} \texttt{z}, \texttt{const acb\_t} \texttt{x}, \texttt{long} \texttt{prec})
 void \texttt{acb\_set\_round\_fmpz} (\texttt{acb\_t} \texttt{z}, \texttt{const fmpz\_t} \texttt{x}, \texttt{long} \texttt{prec})
 void \texttt{acb\_set\_round\_arb} (\texttt{acb\_t} \texttt{z}, \texttt{const arb\_t} \texttt{x}, \texttt{long} \texttt{prec})
 \hspace{1em} Sets \texttt{z} to \texttt{x}, rounded to \texttt{prec} bits.
 void \texttt{acb\_swap} (\texttt{acb\_t} \texttt{z}, \texttt{acb\_t} \texttt{x})
 \hspace{1em} Swaps \texttt{z} and \texttt{x} efficiently.

3.7.4 Input and output

 void \texttt{acb\_print} (\texttt{const acb\_t} \texttt{x})
 \hspace{1em} Prints the internal representation of \texttt{x}.
 void \texttt{acb\_printd} (\texttt{const acb\_t} \texttt{z}, \texttt{long} \texttt{digits})
 \hspace{1em} Prints \texttt{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.

3.7.5 Random number generation

 void \texttt{acb\_randtest} (\texttt{acb\_t} \texttt{z}, \texttt{flint\_rand\_t} \texttt{state}, \texttt{long} \texttt{prec}, \texttt{long} \texttt{mag\_bits})
 \hspace{1em} Generates a random complex number by generating separate random real and imaginary parts.
 void \texttt{acb\_randtest\_special} (\texttt{acb\_t} \texttt{z}, \texttt{flint\_rand\_t} \texttt{state}, \texttt{long} \texttt{prec}, \texttt{long} \texttt{mag\_bits})
 \hspace{1em} Generates a random complex number by generating separate random real and imaginary parts. Also generates NaNs and infinities.
 void \texttt{acb\_randtest\_precise} (\texttt{acb\_t} \texttt{z}, \texttt{flint\_rand\_t} \texttt{state}, \texttt{long} \texttt{prec}, \texttt{long} \texttt{mag\_bits})
 \hspace{1em} Generates a random complex number with precise real and imaginary parts.

3.7.6 Precision and comparisons

 int \texttt{acb\_equal} (\texttt{const acb\_t} \texttt{x}, \texttt{const acb\_t} \texttt{y})
 \hspace{1em} Returns nonzero iff \texttt{x} and \texttt{y} are identical as sets, i.e. if the real and imaginary parts are equal as balls.
 \hspace{1em} Note that this is not the same thing as testing whether both \texttt{x} and \texttt{y} certainly represent the same complex number, unless either \texttt{x} or \texttt{y} is exact (and neither contains NaN). To test whether both operands might represent the same mathematical quantity, use \texttt{acb\_overlaps()} or \texttt{acb\_contains()}, depending on the circumstance.
 int \texttt{acb\_overlaps} (\texttt{const acb\_t} \texttt{x}, \texttt{const acb\_t} \texttt{y})
 \hspace{1em} Returns nonzero iff \texttt{x} and \texttt{y} have some point in common.
 void \texttt{acb\_get\_abs\_ubound\_arf} (\texttt{arf\_t} \texttt{u}, \texttt{const acb\_t} \texttt{z}, \texttt{long} \texttt{prec})
 \hspace{1em} Sets \texttt{u} to an upper bound for the absolute value of \texttt{z}, computed using a working precision of \texttt{prec} bits.
 void \texttt{acb\_get\_abs\_lbound\_arf} (\texttt{arf\_t} \texttt{u}, \texttt{const acb\_t} \texttt{z}, \texttt{long} \texttt{prec})
 \hspace{1em} Sets \texttt{u} to a lower bound for the absolute value of \texttt{z}, computed using a working precision of \texttt{prec} bits.
 void \texttt{acb\_get\_rad\_ubound\_arf} (\texttt{arf\_t} \texttt{u}, \texttt{const acb\_t} \texttt{z}, \texttt{long} \texttt{prec})
 \hspace{1em} Sets \texttt{u} to an upper bound for the error radius of \texttt{z} (the value is currently not computed tightly).
void \texttt{acb\_get\_mag} (mag\_t \texttt{u}, \texttt{const acb\_t} \texttt{x})
\hspace{1em}Sets \texttt{u} to an upper bound for the absolute value of \texttt{x}.

void \texttt{acb\_get\_mag\_lower} (mag\_t \texttt{u}, \texttt{const acb\_t} \texttt{x})
\hspace{1em}Sets \texttt{u} to a lower bound for the absolute value of \texttt{x}.

int \texttt{acb\_contains\_fmpq} (\texttt{const acb\_t} \texttt{x}, \texttt{const fmpq\_t} \texttt{y})
\hspace{1em}Returns nonzero iff \texttt{y} is contained in \texttt{x}.

int \texttt{acb\_contains\_fmpz} (\texttt{const acb\_t} \texttt{x}, \texttt{const fmpz\_t} \texttt{y})
\hspace{1em}Returns nonzero iff \texttt{y} is contained in \texttt{x}.

int \texttt{acb\_contains\_zero} (\texttt{const acb\_t} \texttt{x})
\hspace{1em}Returns nonzero iff zero is contained in \texttt{x}.

long \texttt{acb\_bits} (\texttt{const acb\_t} \texttt{x})
\hspace{1em}Returns the maximum of \texttt{arb\_bits} applied to the real and imaginary parts of \texttt{x}, i.e. the minimum precision sufficient to represent \texttt{x} exactly.

void \texttt{acb\_trim} (\texttt{acb\_t} \texttt{y}, \texttt{const acb\_t} \texttt{x})
\hspace{1em}Sets \texttt{y} to a copy of \texttt{x} with both the real and imaginary parts trimmed (see \texttt{arb\_trim()}).

int \texttt{acb\_is\_real} (\texttt{const acb\_t} \texttt{x})
\hspace{1em}Returns nonzero iff the imaginary part of \texttt{x} is zero. It does not test whether the real part of \texttt{x} also is finite.

int \texttt{acb\_get\_unique\_fmpz} (\texttt{fmpz\_t} \texttt{z}, \texttt{const acb\_t} \texttt{x})
\hspace{1em}If \texttt{x} contains a unique integer, sets \texttt{z} to that value and returns nonzero. Otherwise (if \texttt{x} represents no integers or more than one integer), returns zero.

3.7.7 Complex parts

void \texttt{acb\_arg} (arb\_t \texttt{r}, \texttt{const acb\_t} \texttt{z}, \texttt{long} \texttt{prec})
\hspace{1em}Sets \texttt{r} to a real interval containing the complex argument (phase) of \texttt{z}. We define the complex argument have a discontinuity on $(-\infty, 0]$, with the special value $\arg(0) = 0$, and $\arg(a + 0i) = \pi$ for $a < 0$. Equivalently, if $z = a + bi$, the argument is given by $\text{atan2}(b, a)$ (see \texttt{arb\_atan2()}).

void \texttt{acb\_abs} (arb\_t \texttt{r}, \texttt{const acb\_t} \texttt{z}, \texttt{long} \texttt{prec})
\hspace{1em}Sets \texttt{r} to the absolute value of \texttt{z}.

3.7.8 Arithmetic

void \texttt{acb\_neg} (acb\_t \texttt{z}, \texttt{const acb\_t} \texttt{x})
\hspace{1em}Sets \texttt{z} to the negation of \texttt{x}.

void \texttt{acb\_conj} (acb\_t \texttt{z}, \texttt{const acb\_t} \texttt{x})
\hspace{1em}Sets \texttt{z} to the complex conjugate of \texttt{x}.

void \texttt{acb\_add\_ui} (acb\_t \texttt{z}, \texttt{const acb\_t} \texttt{x}, ulong \texttt{y}, \texttt{long} \texttt{prec})
void \texttt{acb\_add\_fmpz} (acb\_t \texttt{z}, \texttt{const acb\_t} \texttt{x}, \texttt{const fmpz\_t} \texttt{y}, \texttt{long} \texttt{prec})
void \texttt{acb\_add\_arb} (acb\_t \texttt{z}, \texttt{const acb\_t} \texttt{x}, \texttt{const arb\_t} \texttt{y}, \texttt{long} \texttt{prec})
void \texttt{acb\_add} (acb\_t \texttt{z}, \texttt{const acb\_t} \texttt{x}, \texttt{const acb\_t} \texttt{y}, \texttt{long} \texttt{prec})
\hspace{1em}Sets \texttt{z} to the sum of \texttt{x} and \texttt{y}.

void \texttt{acb\_sub\_ui} (acb\_t \texttt{z}, \texttt{const acb\_t} \texttt{x}, ulong \texttt{y}, \texttt{long} \texttt{prec})
void \texttt{acb\_sub\_fmpz} (acb\_t \texttt{z}, \texttt{const acb\_t} \texttt{x}, \texttt{const fmpz\_t} \texttt{y}, \texttt{long} \texttt{prec})
void \texttt{acb\_sub\_arb} (acb\_t \textit{z}, const acb\_t \textit{x}, const arb\_t \textit{y}, long \textit{prec})

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

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

void \texttt{acb\_mul\_ui} (acb\_t \textit{z}, const acb\_t \textit{x}, ulong \textit{y}, long \textit{prec})
void \texttt{acb\_mul\_si} (acb\_t \textit{z}, const acb\_t \textit{x}, long \textit{y}, long \textit{prec})
void \texttt{acb\_mul\_fmpz} (acb\_t \textit{z}, const acb\_t \textit{x}, const fmpz\_t \textit{y}, long \textit{prec})
void \texttt{acb\_mul\_arb} (acb\_t \textit{z}, const acb\_t \textit{x}, const arb\_t \textit{y}, long \textit{prec})
   Sets \textit{z} to the product of \textit{x} and \textit{y}.

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

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

void \texttt{acb\_addmul\_ui} (acb\_t \textit{z}, const acb\_t \textit{x}, ulong \textit{y}, long \textit{prec})
void \texttt{acb\_addmul\_si} (acb\_t \textit{z}, const acb\_t \textit{x}, long \textit{y}, long \textit{prec})
void \texttt{acb\_addmul\_fmpz} (acb\_t \textit{z}, const acb\_t \textit{x}, const fmpz\_t \textit{y}, long \textit{prec})
void \texttt{acb\_addmul\_arb} (acb\_t \textit{z}, const acb\_t \textit{x}, const arb\_t \textit{y}, long \textit{prec})
   Sets \textit{z} to \textit{z} plus the product of \textit{x} and \textit{y}.

void \texttt{acb\_submul\_ui} (acb\_t \textit{z}, const acb\_t \textit{x}, ulong \textit{y}, long \textit{prec})
void \texttt{acb\_submul\_si} (acb\_t \textit{z}, const acb\_t \textit{x}, long \textit{y}, long \textit{prec})
void \texttt{acb\_submul\_fmpz} (acb\_t \textit{z}, const acb\_t \textit{x}, const fmpz\_t \textit{y}, long \textit{prec})
void \texttt{acb\_submul\_arb} (acb\_t \textit{z}, const acb\_t \textit{x}, const arb\_t \textit{y}, long \textit{prec})
   Sets \textit{z} to \textit{z} minus the product of \textit{x} and \textit{y}.

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

void \texttt{acb\_div\_ui} (acb\_t \textit{z}, const acb\_t \textit{x}, ulong \textit{y}, long \textit{prec})
void \texttt{acb\_div\_si} (acb\_t \textit{z}, const acb\_t \textit{x}, long \textit{y}, long \textit{prec})
void \texttt{acb\_div\_fmpz} (acb\_t \textit{z}, const acb\_t \textit{x}, const fmpz\_t \textit{y}, long \textit{prec})
void \texttt{acb\_div} (acb\_t \textit{z}, const acb\_t \textit{x}, const acb\_t \textit{y}, long \textit{prec})
   Sets \textit{z} to the quotient of \textit{x} and \textit{y}.
3.7.9 Elementary functions

```c
void acb_const_pi(acb_t y, long prec)
    Sets y to the constant \( \pi \).

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

void acb_log1p(acb_t z, const acb_t x, long prec)
    Sets \( z = \log(1 + x) \), computed accurately when \( x \approx 0 \).

void acb_exp(acb_t y, const acb_t z, long 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, long prec)
    Sets y to \( \exp(\pi i z) \).

void acb_sin(acb_t s, const acb_t z, long prec)

void acb_cos(acb_t c, const acb_t z, long prec)

void acb_sin_cos(arb_t s, arb_t c, const arb_t z, long prec)
    Sets \( s = \sin(z) \), \( c = \cos(z) \), evaluated as
    \( \sin(a + bi) = \sin(a) \cosh(b) + i \cos(a) \sinh(b) \),
    \( \cos(a + bi) = \cos(a) \cosh(b) - i \sin(a) \sinh(b) \).

void acb_tan(acb_t s, const acb_t z, long prec)
    Sets \( s = \tan(z) = \sin(z)/\cos(z) \), evaluated as
    \( \tan(a + bi) = \sin(2a)/(\cos(2a) + \cosh(2b)) + i \sinh(2b)/(\cos(2a) + \cosh(2b)) \).
    If \( |b| \) is small, the formula is evaluated as written; otherwise, we rewrite
    the hyperbolic functions in terms of decaying exponentials and evaluate
    the expression accurately using \( \text{arb\_expml}() \).

void acb_cot(acb_t s, const acb_t z, long prec)
    Sets \( s = \cot(z) = \cos(z)/\sin(z) \), evaluated as
    \( \cot(a + bi) = -\sin(2a)/(\cos(2a) - \cosh(2b)) + i \sinh(2b)/(\cos(2a) - \cosh(2b)) \).
    Using the same strategy as \( \text{acb\_tan}() \).
    If \( |z| \) is close to zero, however, we evaluate \( 1/\tan(z) \) to avoid
    catastrophic cancellation.

void acb_sin_pi(acb_t s, const acb_t z, long prec)

void acb_cos_pi(acb_t s, const acb_t z, long prec)

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

void acb_tan_pi(acb_t s, const acb_t z, long prec)
    Sets \( s = \tan(\pi z) \). Uses the same algorithm as \( \text{acb\_tan}() \),
    but evaluating the sine and cosine accurately via \( \text{arb\_sin\_cos\_pi}() \).

void acb_cot_pi(acb_t s, const acb_t z, long prec)
    Sets \( s = \cot(\pi z) \). Uses the same algorithm as \( \text{acb\_cot}() \),
    but evaluating the sine and cosine accurately via \( \text{arb\_sin\_cos\_pi}() \).

void acb_atan(acb_t s, const acb_t z, long prec)
    Sets \( s = \tan(\pi z) = \frac{1}{2}i(\log(1 - iz) - \log(1 + iz)) \).

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

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

void acb_pow_si(acb_t y, const acb_t b, long e, long 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 \( \text{acb\_pow}() \) may be superior).
void \texttt{acb\_pow\_arb} (\texttt{acb\_t} z, const \texttt{acb\_t} x, const \texttt{arb\_t} y, long prec)

void \texttt{acb\_pow} (\texttt{acb\_t} z, const \texttt{acb\_t} x, const \texttt{acb\_t} y, long prec)

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

void \texttt{acb\_sqrt} (\texttt{acb\_t} r, const \texttt{acb\_t} z, long 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 \texttt{acb\_rsqrt} (\texttt{acb\_t} r, const \texttt{acb\_t} z, long 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 \( 1/\sqrt{a + bi} = ((a + r) - bi)/u, r = |a + bi|, v = \sqrt{r(a + bi + r)^2}, \) requiring one real square root and one real reciprocal square root.

### 3.7.10 Rising factorials

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

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

void \texttt{acb\_rising\_ui\_rec} (\texttt{acb\_t} z, const \texttt{acb\_t} x, ulong n, long 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 is currently identical to the \texttt{rec} version. In a future version, it will use the gamma function or asymptotic series when this is more efficient.

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

void \texttt{acb\_rising2\_ui\_bs} (\texttt{acb\_t} u, \texttt{acb\_t} v, const \texttt{acb\_t} x, ulong n, long prec)

void \texttt{acb\_rising2\_ui\_rs} (\texttt{acb\_t} u, \texttt{acb\_t} v, const \texttt{acb\_t} x, ulong n, ulong step, long prec)

void \texttt{acb\_rising2\_ui} (\texttt{acb\_t} u, \texttt{acb\_t} v, const \texttt{acb\_t} x, ulong n, long prec)

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

void \texttt{acb\_rising\_ui\_get\_mag} (\texttt{mag\_t} bound, const \texttt{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 $n$.

### 3.7.11 Gamma function

void \texttt{acb\_gamma} (\texttt{acb\_t} y, const \texttt{acb\_t} x, long prec)

Computes the gamma function \( y = \Gamma(x) \).

void \texttt{acb\_rgamma} (\texttt{acb\_t} y, const \texttt{acb\_t} x, long prec)

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

void \texttt{acb\_lgamma} (\texttt{acb\_t} y, const \texttt{acb\_t} x, long 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. Warning: this function does not currently use the reflection formula, and gets very slow for \( z \) far into the left half-plane.

```c
void acb_digamma (acb_t y, const acb_t x, long prec)

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

### 3.7.12 Zeta function

```c
void acb_zeta (acb_t z, const acb_t s, long prec)

Sets \( z \) to the value of the Riemann zeta function \( \zeta(s) \). Note: for computing derivatives with respect to \( s \), use \( \text{acb_poly_zeta_series()} \) or related methods.
```

```c
void acb_hurwitz_zeta (acb_t z, const acb_t s, const acb_t a, long prec)

Sets \( z \) to the value of the Hurwitz zeta function \( \zeta(s, a) \). Note: for computing derivatives with respect to \( s \), use \( \text{acb_poly_zeta_series()} \) or related methods.
```

### 3.7.13 Polylogarithms

```c
void acb_polylog (acb_t w, const acb_t s, const acb_t z, long prec)

Sets \( w \) to the polylogarithm \( Li_s(z) \).
```

```c
void acb_polylog_si (acb_t w, long s, const acb_t z, long prec)

Sets \( w \) to the polylogarithm \( Li_s(z) \).
```

### 3.7.14 Arithmetic-geometric mean

```c
void acb_agm1 (acb_t m, const acb_t z, long 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.
```

```c
void acb_agm1_cpx (acb_ptr m, const acb_t z, long len, long 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. \( M(z + x) \in \mathbb{C}[[x]] \).
```

### 3.8 acb_poly.h – polynomials over the complex numbers

An `acb_poly_t` represents a polynomial over the complex numbers, implemented as an array of coefficients of type `acb_struct`.

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

### 3.8.1 Types, macros and constants

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

An acb_poly_t is defined as an array of length one of type acb_poly_struct, permitting an acb_poly_t to be passed by reference.

3.8.2 Memory management

void acb_poly_init (acb_poly_t poly)
Initializes the polynomial for use, setting it to the zero polynomial.

void acb_poly_clear (acb_poly_t poly)
Cleans the polynomial, deallocating all coefficients and the coefficient array.

void acb_poly_fit_length (acb_poly_t poly, long len)
Makes sure that the coefficient array of the polynomial contains at least len initialized coefficients.

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

void _acb_poly_normalise (acb_poly_t poly)
Strips any trailing coefficients which are identical to zero.

void acb_poly_swap (acb_poly_t poly1, acb_poly_t poly2)
Swaps poly1 and poly2 efficiently.

3.8.3 Basic properties and manipulation

long acb_poly_length (const acb_poly_t poly)
Returns the length of poly, i.e. zero if poly is identically zero, and otherwise one more than the index of the highest term that is not identically zero.

long 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.

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, long prec)
Sets dest to a copy of src, rounded to prec bits.

void acb_poly_set_coeff_si (acb_poly_t poly, long n, long 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, long n, const acb_t c)
Sets the coefficient with index n in poly to the value c. We require that n is nonnegative.

void acb_poly_get_coeff_acb (acb_t v, const acb_poly_t poly, long n)
Sets v to the value of the coefficient with index n in poly. We require that n is nonnegative.
acb_poly_get_coeff_ptr (poly, n)
Given \( n \geq 0 \), returns a pointer to coefficient \( n \) of \( poly \), or NULL if \( n \) exceeds the length of \( poly \).

void _acb_poly_shift_right (acb_ptr res, acb_ptr poly, long len, long n)
void acb_poly_shift_right (acb_poly_t res, const acb_poly_t poly, long 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_ptr poly, long len, long n)
void acb_poly_shift_left (acb_poly_t res, const acb_poly_t poly, long n)
Sets \( res \) to \( poly \) multiplied by \( x^n \). We require that \( n \) is nonnegative.

void acb_poly_truncate (acb_poly_t poly, long n)
Truncates \( poly \) to have length at most \( n \), i.e. degree strictly smaller than \( n \).

3.8.4 Input and output
void acb_poly_printd (const acb_poly_t poly, long digits)
Prints the polynomial as an array of coefficients, printing each coefficient using arb_printd.

3.8.5 Random generation
void acb_poly_randtest (acb_poly_t poly, flint_rand_t state, long len, long prec, long mag_bits)
Creates a random polynomial with length at most \( len \).

3.8.6 Comparisons
int acb_poly_equal (const acb_poly_t A, const acb_poly_t B)
Returns nonzero iff \( A \) and \( B \) are identical as interval polynomials.

int acb_poly_contains (const acb_poly_t poly1, const acb_poly_t poly2)
int acb_poly_contains_fmpz_poly (const acb_poly_t poly1, const fmpz_poly_t poly2)
int acb_poly_contains_fmpq_poly (const acb_poly_t poly1, const fmpq_poly_t poly2)
Returns nonzero iff \( poly2 \) is contained in \( poly1 \).

int _acb_poly_overlaps (acb_ptr poly1, long len1, acb_ptr poly2, long len2)
int acb_poly_overlaps (const acb_poly_t poly1, const acb_poly_t poly2)
Returns nonzero iff \( poly1 \) overlaps with \( poly2 \). The underscore function requires that \( len1 \) is at least as large as \( len2 \).

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

int acb_poly_is_real (const acb_poly_t poly)
Returns nonzero iff all coefficients in \( poly \) have zero imaginary part.

3.8.7 Conversions
void acb_poly_set_fmpz_poly (acb_poly_t poly, const fmpz_poly_t re, long prec)
void acb_poly_set2_fmpz_poly (acb_poly_t poly, const fmpz_poly_t re, const fmpz_poly_t im, long prec)
void `acb_poly_set_arb_poly` (acb_poly_t poly, const arb_poly_t re)

void `acb_poly_set2_arb_poly` (acb_poly_t poly, const arb_poly_t re, const arb_poly_t im)

void `acb_poly_set_fmpq_poly` (acb_poly_t poly, const fmpq_poly_t re, long prec)

void `acb_poly_set2_fmpq_poly` (acb_poly_t poly, const fmpq_poly_t re, const fmpq_poly_t im, long prec)

Sets poly to the given real part re plus the imaginary part im, both rounded to prec bits.

void `acb_poly_set_acb` (acb_poly_t poly, long src)

void `acb_poly_set_si` (acb_poly_t poly, long src)

Sets poly to src.

### 3.8.8 Arithmetic

void `_acb_poly_add` (acb_ptr C, acb_srcptr A, long lenA, acb_srcptr B, long lenB, long 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, long prec)

Sets C to the sum of A and B.

void `_acb_poly_sub` (acb_ptr C, acb_srcptr A, long lenA, acb_srcptr B, long lenB, long 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, long prec)

Sets C to the difference of A and B.

void `acb_poly_neg` (acb_poly_t C, const acb_poly_t A)

Sets C to the negation of A.

void `acb_poly_scalar_mul_2exp_si` (acb_poly_t C, const acb_poly_t A, long c)

Sets C to A multiplied by \(2^c\).

void `_acb_poly_mullow_classical` (acb_ptr C, acb_srcptr A, long lenA, acb_srcptr B, long lenB, long n, long prec)

void `_acb_poly_mullow_transpose` (acb_ptr C, acb_srcptr A, long lenA, acb_srcptr B, long lenB, long n, long prec)

void `_acb_poly_mullow_transpose_gauss` (acb_ptr C, acb_srcptr A, long lenA, acb_srcptr B, long lenB, long n, long prec)

void `_acb_poly_mullow` (acb_ptr C, acb_srcptr A, long lenA, acb_srcptr B, long lenB, long n, long 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 classical version uses a plain loop.

The transpose version evaluates the product using four real polynomial multiplications (via `_arb_poly_mullow()`).

The transpose gauss version evaluates the product using three real polynomial multiplications. This is almost always faster than transpose, but has worse numerical stability when the coefficients vary in magnitude.

The default function `_acb_poly_mullow()` automatically switches between classical and transpose multiplication.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.
void `acb_poly_mullow_classical` (acb_poly_t C, const acb_poly_t A, const acb_poly_t B, long n, long prec)

void `acb_poly_mullow_transpose` (acb_poly_t C, const acb_poly_t A, const acb_poly_t B, long n, long prec)

void `acb_poly_mullow_transpose_gauss` (acb_poly_t C, const acb_poly_t A, const acb_poly_t B, long n, long prec)

void `acb_poly_mullow` (acb_poly_t C, const acb_poly_t A, const acb_poly_t B, long n, long prec)

Sets C to the product of A and B, truncated to length n. If the same variable is passed for A and B, sets C to the square of A truncated to length n.

void `acb_poly_mul` (acb_ptr C, acb_srcptr A, long lenA, acb_srcptr B, long lenB, long 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 ≥ lenB > 0. This function is implemented as a simple wrapper for `acb_poly_mullow()`.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void `acb_poly_mul` (acb_poly_t C, const acb_poly_t A1, const acb_poly_t B2, long prec)

Sets C to the product of A and B. If the same variable is passed for A and B, sets C to the square of A.

void `acb_poly_inv_series` (acb_ptr Qinv, acb_srcptr Q, long Qlen, long len, long prec)

Sets \{Qinv, len\} to the power series inverse of \{Q, Qlen\}. Uses Newton iteration.

void `acb_poly_inv_series` (acb_poly_t Qinv, const acb_poly_t Q, long n, long prec)

Sets Qinv to the power series inverse of Q.

void `acb_poly_div_series` (acb_ptr Q, acb_srcptr A, long Alen, acb_srcptr B, long Blen, long n, long prec)

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

void `acb_poly_div_series` (acb_poly_t Q, const acb_poly_t A, const acb_poly_t B, long n, long prec)

Sets Q to the power series quotient A divided by B, truncated to length n.

void `acb_poly_div` (acb_ptr Q, acb_srcptr A, long lenA, acb_srcptr B, long lenB, long prec)

void `acb_poly_rem` (acb_ptr R, acb_srcptr A, long lenA, acb_srcptr B, long lenB, long prec)

void `acb_poly_divrem` (acb_ptr Q, acb_ptr R, acb_srcptr A, long lenA, acb_srcptr B, long lenB, long prec)

void `acb_poly_divrem` (acb_poly_t Q, acb_poly_t R, const acb_poly_t A, const acb_poly_t B, long prec)

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

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

void `acb_poly_div_root` (acb_ptr Q, acb_t R, acb_srcptr A, long len, const acb_t c, long prec)

Divides A by the polynomial \(x - c\), computing the quotient Q as well as the remainder R = \(f(c)\).

### 3.8.9 Composition

void `acb_poly_compose_horner` (acb_ptr res, acb_srcptr poly1, long len1, acb_srcptr poly2, long len2, long prec)

void `acb_poly_compose_horner` (acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, long prec)

### 3.8. acb_poly.h – polynomials over the complex numbers
Sets $h(x) = f(g(x))$ where $f$ is given by $poly1$ and $g$ is given by $poly2$, respectively using Horner’s rule, divide-and-conquer, and an automatic choice between the two algorithms. The underscore methods do not support aliasing of the output with either input polynomial.

Sets $res$ to the composition $h(x) = f(g(x))$ truncated to order $O(x^n)$ where $f$ is given by $poly1$ and $g$ is given by $poly2$, respectively using Horner’s rule, the Brent-Kung baby step-giant step algorithm, and an automatic choice between the two algorithms. We require that the constant term in $g(x)$ is exactly zero. The underscore methods do not support aliasing of the output with either input polynomial.

Sets $res$ to the power series composition $h(x) = f(g(x))$ truncated to order $O(x^n)$ where $f$ is given by $poly1$ and $g$ is given by $poly2$, respectively using Horner’s rule, the Brent-Kung baby step-giant step algorithm, and an automatic choice between the two algorithms. We require that the constant term in $g(x)$ is exactly zero. The underscore methods do not support aliasing of the output with either input polynomial.

Sets $h$ 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.

### 3.8.10 Evaluation

Sets $res$ to the power series composition $h(x) = f(g(x))$.
void _acb_poly_evaluate_rectangular (acb_t y, acb_srcptr f, long len, const acb_t x, long prec)
void acb_poly_evaluate_rectangular (acb_t y, const acb_poly_t f, const acb_t x, long prec)
void _acb_poly_evaluate (acb_t y, acb_srcptr f, long len, const acb_t x, long prec)
void acb_poly_evaluate (acb_t y, const acb_poly_t f, const acb_t x, long 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, long len, const acb_t x, long prec)
void acb_poly_evaluate2_horner (acb_t y, acb_t z, const acb_poly_t f, const acb_t x, long prec)
void _acb_poly_evaluate2_rectangular (acb_t y, acb_t z, acb_srcptr f, long len, const acb_t x, long prec)
void acb_poly_evaluate2_rectangular (acb_t y, acb_t z, const acb_poly_t f, const acb_t x, long prec)
void _acb_poly_evaluate2 (acb_t y, acb_t z, acb_srcptr f, long len, const acb_t x, long prec)
void acb_poly_evaluate2 (acb_t y, acb_t z, const acb_poly_t f, const acb_t x, long 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.

### 3.8.11 Product trees

void _acb_poly_product_roots (acb_ptr poly, acb_srcptr xs, long n, long prec)
void acb_poly_product_roots (acb_poly_t poly, acb_srcptr xs, long n, long prec)

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

acb_ptr * _acb_poly_tree_alloc (long 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, long len)

Deallocates a tree structure as allocated using _acb_poly_tree_alloc.

void _acb_poly_tree_build (acb_ptr * tree, acb_srcptr roots, long len, long 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().

### 3.8.12 Multipoint evaluation

void _acb_poly_evaluate_vec_iter (acb_ptr ys, acb_srcptr poly, long plen, acb_srcptr xs, long n, long prec)
void acb_poly_evaluate_vec_iter (acb_ptr ys, const acb_poly_t poly, acb_srcptr xs, long n, long 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, long plen, acb_ptr * tree, long len, long prec)
void _acb_poly_evaluate_vec_fast (acb_ptr ys, acb_srcptr poly, long plen, acb_srcptr xs, long n, long prec)
void **acb_poly_evaluate_vec_fast** (acb_ptr ys, const acb_poly_t poly, acb_srcptr xs, long n, long prec)
Evaluates the polynomial simultaneously at n given points, using fast multipoint evaluation.

### 3.8.13 Interpolation

void **_acb_poly_interpolate_newton** (acb_ptr poly, acb_srcptr xs, acb_srcptr ys, long n, long 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, long n, long 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.

### 3.8.14 Differentiation

void **_acb_poly_derivative** (acb_ptr res, acb_srcptr poly, long len, long 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, long prec)
Sets res to the derivative of poly.

void **_acb_poly_integral** (acb_ptr res, acb_srcptr poly, long len, long 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, long prec)
Sets res to the integral of poly.

### 3.8.15 Elementary functions

void **_acb_poly_pow_ui_trunc_binexp** (acb_ptr res, acb_srcptr f, long flen, ulong exp, long len, long 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, long len, long 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, long flen, ulong exp, long prec)
Sets res to \([f, \text{flen}]\) raised to the power \(\text{exp}\). Does not support aliasing of the input and output, and requires that \(\text{flen}\) is positive.

void acb_poly_pow_ui (acb_poly_t res, const acb_poly_t poly, ulong exp, long prec)
Sets res to \(\text{poly}\) raised to the power \(\text{exp}\).

void _acb_poly_pow_series (acb_ptr h, acb_srcptr f, long flen, acb_srcptr g, long glen, long len, long prec)
Sets \([h, \text{len}]\) to the power series \(f(x)^{g(x)} = \exp(g(x) \log f(x))\) truncated to length \(\text{len}\). This function detects special cases such as \(g\) being an exact small integer or \(\pm 1/2\), and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that \(\text{flen}\) and \(\text{glen}\) do not exceed \(\text{len}\).

void acb_poly_pow_series (acb_poly_t h, const acb_poly_t f, const acb_poly_t g, long len, long 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_ptr f, long flen, acb_ptr g, long glen, long len, long prec)
Sets \([h, \text{len}]\) to the power series \(f(x)^{g(x)} = \exp(g \log f(x))\) truncated to length \(\text{len}\). This function detects special cases such as \(g\) being an exact small integer or \(\pm 1/2\), and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that \(\text{flen}\) does not exceed \(\text{len}\).

void acb_poly_pow_acb_series (acb_poly_t h, const acb_poly_t f, const acb_t g, long len, long prec)
Sets \(h\) to the power series \(f(x)^{g(x)} = \exp(g \log f(x))\) truncated to length \(\text{len}\). This function detects special cases such as \(g\) being an exact small integer or \(\pm 1/2\), and computes such powers more efficiently.

void _acb_poly_sqrt_series (acb_ptr g, acb_srcptr h, long hlen, long n, long 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 g, acb_srcptr h, long hlen, long n, long 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, long flen, long n, long 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, long flen, long n, long prec)
Sets \(res\) the power series inverse tangent of \(f\), truncated to length \(n\).

Uses the formula
\[
\tan^{-1}(f(x)) = \int f'(x)/(1 + f(x)^2)dx, 
\]
adding the function of the constant term in $f$ as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that $flen$ and $n$ are greater than zero.

void _acb_poly_exp_series_basecase (acb_ptr $f$, acb_srcptr $h$, long $hlen$, long $n$, long $prec$)
void _acb_poly_exp_series_basecase (acb_poly_t $f$, const acb_poly_t $h$, long $n$, long $prec$)
void _acb_poly_exp_series (acb_ptr $f$, acb_srcptr $h$, long $hlen$, long $n$, long $prec$)
void acb_poly_exp_series (acb_poly_t $f$, const acb_poly_t $h$, long $n$, long $prec$)

Sets $f$ to the power series exponential of $h$, truncated to length $n$.

The basecase version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where $m$ is the length of $h$.

The main implementation uses Newton iteration, starting from a small number of terms given by the basecase algorithm. The complexity is $O(M(n))$. Redundant operations in the Newton iteration are avoided by using the scheme described in [HZ2004].

The underscore methods support aliasing and allow the input to be shorter than the output, but require the lengths to be nonzero.

void _acb_poly_sin_cos_series_basecase (acb_ptr $s$, acb_ptr $c$, acb_srcptr $h$, long $hlen$, long $n$, long $prec$)
void acb_poly_sin_cos_series_basecase (acb_poly_t $s$, acb_poly_t $c$, const acb_poly_t $h$, long $n$, long $prec$)
void _acb_poly_sin_cos_series_tangent (acb_ptr $s$, acb_ptr $c$, acb_srcptr $h$, long $hlen$, long $n$, long $prec$)
void acb_poly_sin_cos_series_tangent (acb_poly_t $s$, acb_poly_t $c$, const acb_poly_t $h$, long $n$, long $prec$)
void _acb_poly_sin_cos_series (acb_ptr $s$, acb_ptr $c$, acb_srcptr $h$, long $hlen$, long $n$, long $prec$)
void acb_poly_sin_cos_series (acb_poly_t $s$, acb_poly_t $c$, const acb_poly_t $h$, long $n$, long $prec$)

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

void _acb_poly_sin_series (acb_ptr $s$, acb_srcptr $h$, long $hlen$, long $n$, long $prec$)
void acb_poly_sin_series (acb_poly_t $s$, const acb_poly_t $h$, long $n$, long $prec$)
void _acb_poly_cos_series (acb_ptr $c$, acb_srcptr $h$, long $hlen$, long $n$, long $prec$)
void acb_poly_cos_series (acb_poly_t $c$, const acb_poly_t $h$, long $n$, long $prec$)

Respectively evaluates the power series sine or cosine. These functions simply wrap _acb_poly_sin_cos_series(). The underscore methods support aliasing and require the lengths to be nonzero.

void _acb_poly_tan_series (acb_ptr $g$, acb_srcptr $h$, long $hlen$, long $len$, long $prec$)
void \texttt{acb\_poly\_tan\_series}(\texttt{acb\_poly\_t} \texttt{g}, \texttt{const acb\_poly\_t} \texttt{h}, \texttt{long} \texttt{n}, \texttt{long} \texttt{prec})

Sets \texttt{g} to the power series tangent of \texttt{h}.

For small \texttt{n} takes the quotient of the sine and cosine as computed using the basecase algorithm. For large \texttt{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.

### 3.8.16 Gamma function

void \texttt{acb\_poly\_gamma\_series}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{acb\_poly\_t} \texttt{h}, \texttt{long} \texttt{len}, \texttt{long} \texttt{prec})

void \texttt{acb\_poly\_rgamma\_series}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{acb\_poly\_t} \texttt{h}, \texttt{long} \texttt{len}, \texttt{long} \texttt{prec})

void \texttt{acb\_poly\_lgamma\_series}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{acb\_poly\_t} \texttt{h}, \texttt{long} \texttt{len}, \texttt{long} \texttt{prec})

These functions first generate the Taylor series at the constant term of \texttt{h}, and then call \texttt{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 \texttt{hlen} and \texttt{n} are greater than zero.

void \texttt{acb\_poly\_rising\_ui\_series}(\texttt{acb\_ptr} \texttt{res}, \texttt{acb\_srcptr} \texttt{f}, \texttt{long} \texttt{flen}, \texttt{ulong} \texttt{r}, \texttt{long} \texttt{trunc}, \texttt{long} \texttt{prec})

void \texttt{acb\_poly\_rising\_ui\_series}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{acb\_poly\_t} \texttt{f}, \texttt{ulong} \texttt{r}, \texttt{long} \texttt{trunc}, \texttt{long} \texttt{prec})

Sets \texttt{res} to the rising factorial \((f)(f + 1)(f + 2)\cdots(f + r - 1)\), truncated to length \texttt{trunc}. The underscore method assumes that \texttt{flen}, \texttt{r} and \texttt{trunc} are at least 1, and does not support aliasing. Uses binary splitting.

### 3.8.17 Power sums

void \texttt{acb\_poly\_powsum\_series\_naive}(\texttt{acb\_ptr} \texttt{z}, \texttt{acb\_t} \texttt{s}, \texttt{acb\_t} \texttt{a}, \texttt{acb\_t} \texttt{q}, \texttt{long} \texttt{n}, \texttt{long} \texttt{len}, \texttt{long} \texttt{prec})

void \texttt{acb\_poly\_powsum\_series\_naive\_threaded}(\texttt{acb\_ptr} \texttt{z}, \texttt{acb\_t} \texttt{s}, \texttt{acb\_t} \texttt{a}, \texttt{acb\_t} \texttt{q}, \texttt{long} \texttt{n}, \texttt{long} \texttt{len}, \texttt{long} \texttt{prec})

Computes

\[
z = S(s, a, n) = \sum_{k=0}^{n-1} \frac{q^k}{(k + a)^{s+t}}
\]

as a power series in \(t\) truncated to length \texttt{len}. This function evaluates the sum naively term by term. The \texttt{threaded} version splits the computation over the number of threads returned by \texttt{flint\_get\_num\_threads()}.

void \texttt{acb\_poly\_powsum\_one\_series\_sieved}(\texttt{acb\_ptr} \texttt{z}, \texttt{acb\_t} \texttt{s}, \texttt{long} \texttt{n}, \texttt{long} \texttt{len}, \texttt{long} \texttt{prec})

Computes

\[
z = S(s, 1, n) \sum_{k=1}^{n} \frac{1}{k^{s+t}}
\]

as a power series in \(t\) truncated to length \texttt{len}. This function stores a table of powers that have already been calculated, computing \((ij)^r\) as \(i^r j^r\) whenever \(k = ij\) is composite. As a further optimization, it groups all even \(k\) and evaluates the sum as a polynomial in \(2^{-(s+t)}\). This scheme requires about \(n/\log n\) powers, \(n/2\)
multiplications, and temporary storage of \( n/6 \) power series. Due to the extra power series multiplications, it is only faster than the naive algorithm when \( \text{len} \) is small.

### 3.8.18 Zeta function

```c
void _acb_poly_zeta_em_choose_param (arf_t bound, ulong * N, ulong * M, const acb_t s, const acb_t a, long d, long target, long prec)

Chooses \( N \) and \( M \) for Euler-Maclaurin summation of the Hurwitz zeta function, using a default algorithm.
```

```c
void _acb_poly_zeta_em_bound1 (arf_t bound, const acb_t s, const acb_t a, long N, long M, long d, long wp)
```

```c
void _acb_poly_zeta_em_bound (arb_ptr vec, const acb_t s, const acb_t a, ulong N, ulong M, long d, long wp)
```

Compute bounds for Euler-Maclaurin evaluation of the Hurwitz zeta function or its power series, using the formulas in [Joh2013].

```c
void _acb_poly_zeta_em_tail_naive (acb_ptr z, const acb_t s, const acb_t Na, acb_srcptr Nasx, long M, long len, long prec)
```

```c
void _acb_poly_zeta_em_tail_bsplit (acb_ptr z, const acb_t s, const acb_t Na, acb_srcptr Nasx, long M, long len, long prec)
```

Evaluates the tail in the Euler-Maclaurin sum for the Hurwitz zeta function, respectively using the naive recurrence and binary splitting.

```c
void _acb_poly_zeta_em_sum (acb_ptr z, const acb_t s, const acb_t a, int deflate, ulong N, ulong M, long d, long 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 \( s = 1 \)).

```c
void _acb_poly_zeta_cpx_series (acb_ptr z, const acb_t s, const acb_t a, int deflate, long d, long 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 \( 2^{-\text{prec}} \).

```c
void _acb_poly_zeta_series (acb_ptr res, acb_srcptr h, long len, const acb_t a, int deflate, long len, long prec)
```

```c
void acb_poly_zeta_series (acb_poly_t res, const acb_poly_t f, const acb_t a, int deflate, long n, long 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.

### 3.8.19 Other special functions

```c
void _acb_poly_polylog_cpx_small (acb_ptr w, const acb_t s, const acb_t z, long len, long prec)
```

```c
void _acb_poly_polylog_cpx_zeta (acb_ptr w, const acb_t s, const acb_t z, long len, long prec)
```
void \texttt{acb\_poly\_polylog\_cpx}(acb\_ptr w, const acb\_t s, const acb\_t z, long len, long 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 small version uses the standard power series expansion with respect to \( z \), convergent when \(|z| < 1\). The zeta version evaluates the polylogarithm as a sum of two Hurwitz zeta functions. The default version automatically delegates to the small version when \( z \) is close to zero, and the zeta version otherwise. For further details, see Algorithms for polylogarithms.

void \texttt{acb\_poly\_polylog\_series}(acb\_ptr w, const acb\_poly\_t poly, acb\_poly\_t polyder, len, long prec)

Sets \( w \) to the polylogarithm \( \text{Li}_s(z) \) where \( s \) is a given power series, truncating the output to length \( \text{len} \). The underscore method requires all lengths to be positive and supports aliasing between all inputs and outputs.

3.8.20 Root-finding

void \texttt{acb\_poly\_root\_inclusion}(acb\_ptr r, const acb\_t m, acb\_srcptr poly, acb\_srcptr polyder, long len, long 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

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

which is a contradiction (see [Kob2010]).
long _acb_poly_validate_roots (acb_ptr roots, acb_srcptr poly, long len, long prec)
Given a list of approximate roots of the input polynomial, this function sets a rigorous bounding interval for each root, and determines which roots are isolated from all the other roots. It then rearranges the list of roots so that the isolated roots are at the front of the list, and returns the count of isolated roots.

If the return value equals the degree of the polynomial, then all roots have been found. If the return value is smaller, all the remaining output intervals are guaranteed to contain roots, but it is possible that not all of the polynomial’s roots are contained among them.

void _acb_poly_refine_roots_durand_kerner (acb_ptr roots, acb_srcptr poly, long len, long 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).

long _acb_poly_find_roots (acb_ptr roots, acb_srcptr poly, acb_srcptr initial, long len, long maxiter, long prec)

long acb_poly_find_roots (acb_ptr roots, const acb_poly_t poly, acb_srcptr initial, long maxiter, long 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.

3.9 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.

3.9.1 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.

3.9.2 Memory management

void acb_mat_init (acb_mat_t mat, long r, long 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.

3.9.3 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_fmpq_mat (acb_mat_t dest, const fmpq_mat_t src, long prec)
    Sets dest to src. The operands must have identical dimensions.

3.9.4 Random generation

void acb_mat_randtest (acb_mat_t mat, flint_rand_t state, long prec, long mag_bits)
    Sets mat to a random matrix with up to prec bits of precision and with exponents of width up to mag_bits.

3.9.5 Input and output

void acb_mat_printd (const acb_mat_t mat, long digits)
    Prints each entry in the matrix with the specified number of decimal digits.

3.9.6 Comparisons

int acb_mat_equal (const acb_mat_t mat1, const acb_mat_t mat2)
    Returns nonzero iff the matrices have the same dimensions and identical entries.

int acb_mat_overlaps (const acb_mat_t mat1, const acb_mat_t mat2)
    Returns nonzero iff the matrices have the same dimensions and each entry in mat1 overlaps with the corresponding entry in mat2.

int acb_mat_contains (const acb_mat_t mat1, const acb_mat_t mat2)
int acb_mat_contains_fmpz_mat (const acb_mat_t mat1, const fmpz_mat_t mat2)
int acb_mat_contains_fmpq_mat (const acb_mat_t mat1, const fmpq_mat_t mat2)
    Returns nonzero iff the matrices have the same dimensions and each entry in mat2 is contained in the corresponding entry in mat1.

int acb_mat_is_real (const acb_mat_t mat)
    Returns nonzero iff all entries in mat have zero imaginary part.

3.9. acb_mat.h – matrices over the complex numbers
3.9.7 Special matrices

void \texttt{acb\_mat\_zero} (acb\_mat\_t mat)
Sets all entries in mat to zero.

void \texttt{acb\_mat\_one} (acb\_mat\_t mat)
Sets the entries on the main diagonal to ones, and all other entries to zero.

3.9.8 Norms

void \texttt{acb\_mat\_bound\_inf\_norm} (mag\_t b, const acb\_mat\_t A)
Sets b to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of A.

3.9.9 Arithmetic

void \texttt{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 \texttt{acb\_mat\_add} (acb\_mat\_t res, const acb\_mat\_t mat1, const acb\_mat\_t mat2, long prec)
Sets res to the sum of mat1 and mat2. The operands must have the same dimensions.

void \texttt{acb\_mat\_sub} (acb\_mat\_t res, const acb\_mat\_t mat1, const acb\_mat\_t mat2, long prec)
Sets res to the difference of mat1 and mat2. The operands must have the same dimensions.

void \texttt{acb\_mat\_mul} (acb\_mat\_t res, const acb\_mat\_t mat1, const acb\_mat\_t mat2, long prec)
Sets res to the matrix product of mat1 and mat2. The operands must have compatible dimensions for matrix multiplication.

void \texttt{acb\_mat\_pow\_ui} (acb\_mat\_t res, const acb\_mat\_t mat, ulong exp, long prec)
Sets res to mat raised to the power exp. Requires that mat is a square matrix.

3.9.10 Scalar arithmetic

void \texttt{acb\_mat\_scalar\_mul\_2exp\_si} (acb\_mat\_t B, const acb\_mat\_t A, long c)
Sets B to A multiplied by $2^c$.

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

void \texttt{acb\_mat\_scalar\_addmul\_fmpz} (acb\_mat\_t B, const acb\_mat\_t A, const fmpz\_t c, long prec)

void \texttt{acb\_mat\_scalar\_addmul\_arb} (acb\_mat\_t B, const acb\_mat\_t A, const arb\_t c, long prec)

void \texttt{acb\_mat\_scalar\_addmul\_acb} (acb\_mat\_t B, const acb\_mat\_t A, const acb\_t c, long prec)
Sets B to $B + A \times c$.

void \texttt{acb\_mat\_scalar\_mul\_si} (acb\_mat\_t B, const acb\_mat\_t A, long c, long prec)

void \texttt{acb\_mat\_scalar\_mul\_fmpz} (acb\_mat\_t B, const acb\_mat\_t A, const fmpz\_t c, long prec)

void \texttt{acb\_mat\_scalar\_mul\_arb} (acb\_mat\_t B, const acb\_mat\_t A, const arb\_t c, long prec)

void \texttt{acb\_mat\_scalar\_mul\_acb} (acb\_mat\_t B, const acb\_mat\_t A, const acb\_t c, long prec)
Sets B to $A \times c$.

void \texttt{acb\_mat\_scalar\_div\_si} (acb\_mat\_t B, const acb\_mat\_t A, long c, long prec)

void \texttt{acb\_mat\_scalar\_div\_fmpz} (acb\_mat\_t B, const acb\_mat\_t A, const fmpz\_t c, long prec)

void \texttt{acb\_mat\_scalar\_div\_arb} (acb\_mat\_t B, const acb\_mat\_t A, const arb\_t c, long prec)
void **acb_mat_scalar_div_acb**(acb_mat_t B, const acb_mat_t A, const acb_t c, long prec)
Sets B to A/c.

### 3.9.11 Gaussian elimination and solving

**int acb_mat_lu**(long * perm, acb_mat_t LU, const acb_mat_t A, long prec)

Given an \( n \times n \) matrix A, computes an LU decomposition \( PLU = A \) using Gaussian elimination with partial pivoting. The input and output matrices can be the same, performing the decomposition in-place.

Entry \( i \) in the permutation vector perm is set to the row index in the input matrix corresponding to row \( i \) in the output matrix.

The algorithm succeeds and returns nonzero if it can find \( n \) invertible (i.e. not containing zero) pivot entries. This guarantees that the matrix is invertible.

The algorithm fails and returns zero, leaving the entries in \( P \) and \( LU \) undefined, if it cannot find \( n \) invertible pivot elements. In this case, either the matrix is singular, the input matrix was computed to insufficient precision, or the LU decomposition was attempted at insufficient precision.

void **acb_mat_solve_lu_precomp**(acb_mat_t X, const long * perm, const acb_mat_t LU, const acb_mat_t B, long prec)

Solves \( AX = B \) given the precomputed nonsingular LU decomposition \( A = PLU \). The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with LU.

**int acb_mat_solve**(acb_mat_t X, const acb_mat_t A, const acb_mat_t B, long prec)

Solves \( AX = B \) where \( A \) is a nonsingular \( n \times n \) matrix and \( X \) and \( B \) are \( n \times m \) matrices, using LU decomposition.

If \( m > 0 \) and \( A \) cannot be inverted numerically (indicating either that \( A \) is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that \( A \) is invertible and that the exact solution matrix is contained in the output.

**int acb_mat_inv**(acb_mat_t X, const acb_mat_t A, long prec)

Sets \( X = A^{-1} \) where \( A \) is a square matrix, computed by solving the system \( AX = I \).

If \( A \) cannot be inverted numerically (indicating either that \( A \) is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the matrix is invertible and that the exact inverse is contained in the output.

**void acb_mat_det**(acb_t det, const acb_mat_t A, long 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.

### 3.9.12 Characteristic polynomial

**void _acb_mat_charpoly**(acb_ptr cp, const acb_mat_t mat, long prec)

**void acb_mat_charpoly**(acb_poly_t cp, const acb_mat_t mat, long 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.

---

3.9. acb_mat.h – matrices over the complex numbers
# 3.10 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).

## 3.10.1 Types, macros and constants

**acb_calc_func_t**

Typedef for a pointer to a function with signature:

```c
int func(acb_ptr out, const acb_t inp, void * param, long order, long prec)
```

implementing a univariate complex function \( f(x) \). When called, `func` should write to `out` the first `order` coefficients in the Taylor series expansion of \( f(x) \) at the point `inp`, evaluated at a precision of `prec` bits. The `param` argument may be used to pass through additional parameters to the function. The return value is reserved for future use as an error code. It can be assumed that `out` and `inp` are not aliased and that `order` is positive.

## 3.10.2 Bounds

**acb_calc_cauchy_bound** (arb_t bound, acb_calc_func_t func, void * param, const acb_t x, const arb_t radius, long maxdepth, long prec)

Sets `bound` to a ball containing the value of the integral

\[
C(x, r) = \frac{1}{2\pi r} \int_{|z-x|=r} |f(z)|dz = \int_0^1 |f(x + re^{2\pi it})|dt
\]

where \( f \) is specified by `(func, param)` and \( r \) is given by `radius`. The integral is computed using a simple step sum. The integration range is subdivided until the order of magnitude of \( b \) can be determined (i.e. its error bound is smaller than its midpoint), or until the step length has been cut in half `maxdepth` times. This function is currently implemented completely naïvely, and repeatedly subdivides the whole integration range instead of performing adaptive subdivisions.

## 3.10.3 Integration

**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, long accuracy_goal, long prec)

Computes the integral

\[
I = \int_a^b f(t)dt
\]
where \( f \) is specified by \((\text{func}, \text{param})\), following a straight-line path between the complex numbers \( a \) and \( b \) which both must be finite.

The integral is approximated by piecewise centered Taylor polynomials. Rigorous truncation error bounds are calculated using the Cauchy integral formula. More precisely, if the Taylor series of \( f \) centered at the point \( m \) is 

\[
\sum_{n=0}^{\infty} a_n x^n,
\]

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

\[
\sum_{n=N}^{\infty} \frac{C(m, R) |x|^{n+1}}{R^n (N+1)} = \frac{C(m, R) R x}{(R-x)(N+1)} \left( \frac{x}{R} \right)^{N}.
\]

It is required that any singularities of \( f \) are isolated from the path of integration by a distance strictly greater than the positive value \( \text{outer\_radius} \) (which is the integration radius used for the Cauchy bound). Taylor series step lengths are chosen so as not to exceed \( \text{inner\_radius} \), which must be strictly smaller than \( \text{outer\_radius} \) for convergence. A smaller \( \text{inner\_radius} \) 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 \( \text{inner\_radius} \) to half the value of \( \text{outer\_radius} \), 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 \( \text{accuracy\_goal} \) (in the future, this might change to a relative accuracy). Arithmetic operations and function evaluations are performed at a precision of \( \text{prec} \) bits. Note that due to accumulation of numerical errors, both values may have to be set higher (and the endpoints may have to be computed more accurately) to achieve a desired accuracy.

This function chooses the evaluation points uniformly rather than implementing adaptive subdivision.

### 3.11 acb_hypgeom.h – hypergeometric functions in the complex numbers

The generalized hypergeometric function is formally defined by

\[
_{p}F_{q}(a_1, \ldots, a_p; b_1, \ldots, b_q; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \cdots (a_p)_k}{(b_1)_k \cdots (b_q)_k} \frac{z^k}{k!}.
\]

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.

#### 3.11.1 Convergent series

In this section, we define

\[
T(k) = \frac{\prod_{i=0}^{p-1} (a_i)_k z^k}{\prod_{i=0}^{q-1} (b_i)_k}
\]

and

\[
_{p}H_{q}(a_0, \ldots, a_{p-1}; b_0 \ldots b_{q-1}; z) = _{p+1}F_{q}(a_0, \ldots, a_{p-1}, 1; b_0 \ldots b_{q-1}; z) = \sum_{k=0}^{\infty} T(k)
\]

For the conventional generalized hypergeometric function \(_{p}F_{q}\), compute \(_{p}H_{q+1}\) with the explicit parameter \( b_q = 1 \), or remove a 1 from the \( a_i \) parameters if there is one.
void \texttt{acb_hypgeom_pfq\_bound\_factor}(\texttt{mag\_t }C, \texttt{acb\_srcptr }a, \texttt{long }p, \texttt{acb\_srcptr }b, \texttt{long }q, \texttt{const }\texttt{acb\_t }z, \texttt{ulong }n)

Computes 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

$$
\left| \frac{a + k}{b + k} \right| = \left| 1 + \frac{a - b}{b + k} \right| \leq 1 + \frac{|a - b|}{|b + n|}
$$

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 $C = \sum_{k=0}^{\infty} D^k = (1 - D)^{-1}$.

As currently implemented, the bound becomes infinite when $n$ is too small, even if the series converges.

long \texttt{acb_hypgeom_pfq\_choose\_n}(\texttt{acb\_srcptr }a, \texttt{long }p, \texttt{acb\_srcptr }b, \texttt{long }q, \texttt{const }\texttt{acb\_t }z, \texttt{long 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 \texttt{acb_hypgeom_pfq\_sum\_forward}(\texttt{acb\_t }s, \texttt{acb\_t }t, \texttt{acb\_srcptr }a, \texttt{long }p, \texttt{acb\_srcptr }b, \texttt{long }q, \texttt{const }\texttt{acb\_t }z, \texttt{long prec})

void \texttt{acb_hypgeom_pfq\_sum\_rs}(\texttt{acb\_t }s, \texttt{acb\_t }t, \texttt{acb\_srcptr }a, \texttt{long }p, \texttt{acb\_srcptr }b, \texttt{long }q, \texttt{const }\texttt{acb\_t }z, \texttt{long }n, \texttt{long prec})

void \texttt{acb_hypgeom_pfq\_sum}(\texttt{acb\_t }s, \texttt{acb\_t }t, \texttt{acb\_srcptr }a, \texttt{long }p, \texttt{acb\_srcptr }b, \texttt{long }q, \texttt{const }\texttt{acb\_t }z, \texttt{long }n, \texttt{long 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 \texttt{forward} version computes the sum using forward recurrence.

The \texttt{rs} version computes the sum in reverse order using rectangular splitting. It only computes a magnitude bound for the value of $t$.

The default version automatically chooses an algorithm depending on the inputs.

void \texttt{acb_hypgeom_pfq\_direct}(\texttt{acb\_t }res, \texttt{acb\_srcptr }a, \texttt{long }p, \texttt{acb\_srcptr }b, \texttt{long }q, \texttt{const }\texttt{acb\_t }z, \texttt{long }n, \texttt{long prec})

Computes

$$
p H_q(z) = \sum_{k=0}^{\infty} T(k) = \sum_{k=0}^{n-1} T(k) + \varepsilon
$$

directly from the defining series, including a rigorous bound for the truncation error $\varepsilon$ in the output.

If $n < 0$, this function chooses a number of terms automatically using \texttt{acb_hypgeom_pfq\_choose\_n}().
3.11.2 Asymptotic series

Let $U(a, b, z)$ denote the confluent hypergeometric function of the second kind with the principal branch cut, and let $U^* = z^a U(a, b, z)$. For all $z \neq 0$ and $b \notin \mathbb{Z}$ (but valid for all $b$ as a limit), we have (DLMF 13.2.42)

$$U(a, b, z) = \frac{\Gamma(1-b)}{\Gamma(a-b+1)} M(a, b, z) + \frac{\Gamma(b-1)}{\Gamma(a)} z^{1-b} M(a-b+1, 2-b, z).$$

Moreover, for all $z \neq 0$ we have

$$\frac{1}{1} F_{1}(a, b, z) = \frac{(-z)^{-a}}{\Gamma(b-a)} U^*(a, b, z) + \frac{z^{a-b}e^{z}}{\Gamma(a)} U^*(b-a, b, -z)$$

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 $\text{Re}(z) \geq r$, set $R = 1$. Otherwise, if $\text{Im}(z) \geq r$ or $\text{Re}(z) \geq 0 \wedge |z| \geq r$, set $R = 2$. Otherwise, if $|z| \geq 2r$, set $R = 3$. Otherwise, the bound is infinite. If the bound is finite, we have

$$|\varepsilon_n(z)| \leq 2\alpha C_n \left| \frac{(a)_{n}(a-b+1)_{n}}{n! z^{n}} \right| \exp(2\alpha \nu \sigma_{1}/|z|)$$

in terms of the following auxiliary quantities

$$\sigma = |(b-2a)/z|$$

$$C_n = \begin{cases} 1 & \text{if } R = 1 \\ \chi(n) & \text{if } R = 2 \\ (\chi(n) + \nu^{2} n^{2})^{n} & \text{if } R = 3 \end{cases}$$

$$\nu = \left(\frac{1}{4} + \frac{1}{2} \sqrt{1-4\sigma^{2}}\right)^{-1/2} \leq 1 + 2\sigma^{2}$$

$$\chi(n) = \sqrt{\pi} \Gamma(\frac{1}{2} n + 1) / \Gamma(\frac{1}{2} n + \frac{1}{2})$$

$$\sigma' = \begin{cases} \sigma & \text{if } R \neq 3 \\ \nu \sigma & \text{if } R = 3 \end{cases}$$

$$\alpha = (1 - \sigma')^{-1}$$

$$\nu = \frac{1}{4}[2a^{2} - 2ab + b] + \sigma'(1 + \frac{1}{4} \sigma')(1 - \sigma')^{-2}$$

void \textbf{acb_hypgeom_u_asymp} (acb_t res, const acb_t a, const acb_t b, const acb_t z, long n, long prec)

Sets \textit{res} to $U^*(a, b, z)$ computed using $n$ terms of the asymptotic series, with a rigorous bound for the error included in the output. We require $n \geq 0$.  

3.11. acb_hypgeom.h – hypergeometric functions in the complex numbers
3.11.3 The error function

```c
void acb_hypgeom_erf_1f1a (acb_t res, const acb_t z, long prec)
void acb_hypgeom_erf_1f1b (acb_t res, const acb_t z, long prec)
void acb_hypgeom_erf_asym (acb_t res, const acb_t z, long prec, long prec2)
void acb_hypgeom_erf (acb_t res, const acb_t z, long prec)
```

Computes the error function respectively using

\[ \text{erf}(z) = \frac{2z}{\sqrt{\pi}} F_1 \left( \frac{1}{2}, \frac{3}{2}, -z^2 \right) \]
\[ \text{erf}(z) = \frac{2ze^{-z^2}}{\sqrt{\pi}} F_1 \left( 1, \frac{3}{2}, z^2 \right) \]
\[ \text{erf}(z) = \frac{z}{\sqrt{2z^2}} \left( 1 - \frac{e^{-z^2}}{\sqrt{\pi}} U \left( \frac{1}{2}, \frac{1}{2}, z^2 \right) \right) \]

and an automatic algorithm choice. The \texttt{asymp} version takes a second precision to use for the \texttt{U} term.

```c
void acb_hypgeom_erfc (acb_t res, const acb_t s, const acb_t z, long prec)
```

Computes the complementary error function \( \text{erfc}(z) = 1 - \text{erf}(z) \). This function avoids catastrophic cancellation for large positive \( z \).

```c
void acb_hypgeom_erfi (acb_t res, const acb_t s, const acb_t z, long prec)
```

Computes the imaginary error function \( \text{erfi}(z) = -i \text{erf}(iz) \). This is a trivial wrapper of \texttt{acb_hypgeom_erf()}.

3.11.4 Bessel functions

```c
void acb_hypgeom_bessel_j_asymp (acb_t res, const acb_t nu, const acb_t z, long prec)
```

Computes the Bessel function of the first kind via \texttt{acb_hypgeom_u_asymp()}.

\[ J_\nu(z) = \frac{z^\nu}{\Gamma(\nu + 1)} F_1 \left( \nu + \frac{1}{2}, 2\nu + 1, 2iz \right) = A_+ B_+ + A_- B_- \]

where

\[ A_\pm = z^\nu (z^2)^{-1/2} - \nu (\mp iz) z^{-1/2} = (\pm iz)^{-1/2} \pm 2iz \] \[ B_\pm = e^{\pm iz} U^* \left( \nu + \frac{1}{2}, 2\nu + 1, \mp 2iz \right) \]

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 \( A_\pm = \exp(\mp i[(2\nu + 1)/4]i)(2\pi z)^{-1/2} \).

```c
void acb_hypgeom_bessel_j_0f1 (acb_t res, const acb_t nu, const acb_t z, long prec)
```

Computes the Bessel function of the first kind from

\[ J_\nu(z) = \frac{1}{\Gamma(\nu + 1)} \left( \frac{z}{2} \right)^\nu \eta F_1 \left( \nu + 1, -\frac{z^2}{4} \right) \]

```c
void acb_hypgeom_bessel_j (acb_t res, const acb_t nu, const acb_t z, long prec)
```

Computes the Bessel function of the first kind \( J_\nu(z) \) using an automatic algorithm choice.
3.11.5 Incomplete gamma functions

void \texttt{acb\_hypgeom\_gamma\_upper\_asym} (acb\_t res, const acb\_t s, const acb\_t z, int modified, long prec)

void \texttt{acb\_hypgeom\_gamma\_upper\_1f1a} (acb\_t res, const acb\_t s, const acb\_t z, int modified, long prec)

void \texttt{acb\_hypgeom\_gamma\_upper\_1f1b} (acb\_t res, const acb\_t s, const acb\_t z, int modified, long prec)

void \texttt{acb\_hypgeom\_gamma\_upper\_singular} (acb\_t res, long s, const acb\_t z, int modified, long prec)

void \texttt{acb\_hypgeom\_gamma\_upper} (acb\_t res, const acb\_t s, const acb\_t z, int modified, long prec)

Computes the upper incomplete gamma function respectively using

\[ \Gamma(s, z) = e^{-z} U(1 - s, 1 - s, z) \]

\[ \Gamma(s, z) = \Gamma(s) - \frac{z^s}{s} F_1(s, s + 1, -z) \]

\[ \Gamma(s, z) = \Gamma(s) - \frac{z^s e^{-z}}{s} F_1(1, s + 1, z) \]

\[ \Gamma(s, z) = \left(\frac{-1}{n!}\right)^n (\psi(n + 1) - \log(z)) + \left(\frac{-1}{n + 1}\right)^n \frac{z}{2} F_2(1, 1, 2, 2 + n, -z) - z^{-n} \sum_{k=0}^{n-1} \frac{(-z)^k}{(k - n)k!}, \quad n = -s \in \mathbb{Z}_{\geq 0} \]

and an automatic algorithm choice. The automatic version also handles other special input such as \( z = 0 \) and \( s = 1, 2, 3 \). The \texttt{singular} version evaluates the finite sum directly and therefore assumes that \( s \) is not too large. If \texttt{modified} is set, computes the exponential integral \( z^{-s} \Gamma(s, z) = E_{1-s}(z) \) instead.

void \texttt{acb\_hypgeom\_expint} (acb\_t res, const acb\_t s, const acb\_t z, long prec)

Computes the exponential integral \( E_s(z) \). This is a trivial wrapper of \texttt{acb\_hypgeom\_gamma\_upper}().

3.12 acb\_modular.h – modular forms in the complex numbers

This module provides methods for numerical evaluation of modular forms, Jacobi theta functions, and elliptic functions.

In the context of this module, \texttt{tau} or \texttt{\tau} always denotes an element of the complex upper half-plane \( \mathbb{H} = \{ z \in \mathbb{C} : \Im(z) > 0 \} \). We also often use the variable \( q \), variously defined as \( q = e^{2\pi i \tau} \) (usually in relation to modular forms) or \( q = e^{\pi i \tau} \) (usually in relation to theta functions) and satisfying \( |q| < 1 \). We will clarify the local meaning of \( q \) every time such a quantity appears as a function of \( \tau \).

As usual, the numerical functions in this module compute strict error bounds: if \texttt{tau} is represented by an \texttt{acb\_t} whose content overlaps with the real line (or lies in the lower half-plane), and \texttt{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 \( |q| < 1 \).

3.12.1 The modular group

\texttt{psl2z\_struct}

\texttt{psl2z\_t}

Represents an element of the modular group 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 \texttt{fmpz}.
void \texttt{psl2z\_init} \hphantom{.}\texttt{(psl2z\_t g)}
  \begin{quote}
  Initializes \( g \) and set it to the identity element.
  \end{quote}

void \texttt{psl2z\_clear} \hphantom{.}\texttt{(psl2z\_t g)}
  \begin{quote}
  Clears \( g \).
  \end{quote}

void \texttt{psl2z\_swap} \hphantom{.}\texttt{(psl2z\_t f, psl2z\_t g)}
  \begin{quote}
  Swaps \( f \) and \( g \) efficiently.
  \end{quote}

void \texttt{psl2z\_set} \hphantom{.}\texttt{(psl2z\_t f, const psl2z\_t g)}
  \begin{quote}
  Sets \( f \) to a copy of \( g \).
  \end{quote}

void \texttt{psl2z\_one} \hphantom{.}\texttt{(psl2z\_t g)}
  \begin{quote}
  Sets \( g \) to the identity element.
  \end{quote}

int \texttt{psl2z\_is\_one} \hphantom{.}\texttt{(const psl2z\_t g)}
  \begin{quote}
  Returns nonzero iff \( g \) is the identity element.
  \end{quote}

void \texttt{psl2z\_print} \hphantom{.}\texttt{(const psl2z\_t g)}
  \begin{quote}
  Prints \( g \) to standard output.
  \end{quote}

int \texttt{psl2z\_equal} \hphantom{.}\texttt{(const psl2z\_t f, const psl2z\_t g)}
  \begin{quote}
  Returns nonzero iff \( f \) and \( g \) are equal.
  \end{quote}

void \texttt{psl2z\_mul} \hphantom{.}\texttt{(psl2z\_t h, const psl2z\_t f, const psl2z\_t g)}
  \begin{quote}
  Sets \( h \) to the product of \( f \) and \( g \), namely the matrix product with the signs canonicalized.
  \end{quote}

void \texttt{psl2z\_inv} \hphantom{.}\texttt{(psl2z\_t h, const psl2z\_t g)}
  \begin{quote}
  Sets \( h \) to the inverse of \( g \).
  \end{quote}

int \texttt{psl2z\_is\_correct} \hphantom{.}\texttt{(const psl2z\_t g)}
  \begin{quote}
  Returns nonzero iff \( g \) contains correct data, i.e. satisfying \( ad - bc = 1 \), \( c \geq 0 \), and \( d > 0 \) if \( c = 0 \).
  \end{quote}

void \texttt{psl2z\_randtest} \hphantom{.}\texttt{(psl2z\_t g, flint\_rand\_t state, long bits)}
  \begin{quote}
  Sets \( g \) to a random element of \( \text{PSL}(2, \mathbb{Z}) \) with entries of bit length at most \( \text{bits} \) (or 1, if \( \text{bits} \) is not positive). We first generate \( a \) and \( d \), compute their Bezout coefficients, divide by the GCD, and then correct the signs.
  \end{quote}

\subsection{3.12.2 Modular transformations}

void \texttt{acb\_modular\_transform} \hphantom{.}\texttt{(acb\_t w, const psl2z\_t g, const acb\_t z, long prec)}
  \begin{quote}
  Applies the modular transformation \( g \) to the complex number \( z \), evaluating
  \[ w = gz = \frac{az + b}{cz + d}. \]
  \end{quote}

void \texttt{acb\_modular\_fundamental\_domain\_approx\_d} \hphantom{.}\texttt{(psl2z\_t g, double x, double y, double one\_minus\_eps)}

void \texttt{acb\_modular\_fundamental\_domain\_approx\_arf} \hphantom{.}\texttt{(psl2z\_t g, const arf\_t x, const arf\_t y, const arf\_t one\_minus\_eps, long prec)}
  \begin{quote}
  Attempts to determine a modular transformation \( g \) that maps the complex number \( x + yi \) to the fundamental domain or just slightly outside the fundamental domain, where the target tolerance (not a strict bound) is specified by \( \text{one\_minus\_eps} \).
  \end{quote}

The inputs are assumed to be finite numbers, with \( y \) positive.

Uses floating-point iteration, repeatedly applying either the transformation \( z \leftarrow z + b \) or \( z \leftarrow -1/z \). The iteration is terminated if \( |x| \leq 1/2 \) and \( x^2 + y^2 \geq 1 - \varepsilon \) where \( 1 - \varepsilon \) is passed as \( \text{one\_minus\_eps} \). It is also terminated if too many steps have been taken without convergence, or if the numbers end up too large or too small for the working precision.
The algorithm can fail to produce a satisfactory transformation. The output $g$ is always set to some correct modular transformation, but it is up to the user to verify a posteriori that $g$ maps $x + yi$ close enough to the fundamental domain.

```c
void acb_modular_fundamental_domain_approx(acb_t w, psl2z_t g, const acb_t z, const arf_t one_minus_eps, long prec)
```

Attempts to determine a modular transformation $g$ that maps the complex number $z$ to the fundamental domain or just slightly outside the fundamental domain, where the target tolerance (not a strict bound) is specified by `one_minus_eps`. It also computes the transformed value $w = g z$.

This function first tries to use `acb_modular_fundamental_domain_approx_d()` and checks if the result is acceptable. If this fails, it calls `acb_modular_fundamental_domain_approx_arf()` with higher precision. Finally, $w = g z$ 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.

```c
int acb_modular_is_in_fundamental_domain(const acb_t z, const arf_t tol, long 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 `tol`. Returns zero if this is false or cannot be determined.

### 3.12.3 Jacobi theta functions

Unfortunately, there are many inconsistent notational variations for Jacobi theta functions in the literature. Unless otherwise noted, we use the functions

\[
\theta_1(z, \tau) = -i \sum_{n=-\infty}^{\infty} (-1)^n \exp(i[(n + 1/2)^2 \tau + (2n + 1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} (-1)^n q^{n(n+1)} \sin((2n + 1)\pi z)
\]

\[
\theta_2(z, \tau) = \sum_{n=-\infty}^{\infty} \exp(i[(n + 1/2)^2 \tau + (2n + 1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos((2n + 1)\pi z)
\]

\[
\theta_3(z, \tau) = \sum_{n=-\infty}^{\infty} \exp(i[n^2 \tau + 2nz]) = 1 + 2 \sum_{n=1}^{\infty} q^n \cos(2n\pi z)
\]

\[
\theta_4(z, \tau) = \sum_{n=-\infty}^{\infty} (-1)^n \exp(i[n^2 \tau + 2nz]) = 1 + 2 \sum_{n=1}^{\infty} (-1)^n q^n \cos(2n\pi z)
\]

where $q = \exp(\pi i \tau)$ 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.

```c
void acb_modular_theta_transform(int * R, int * S, int * C, const psl2z_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

\[
\theta_{1+i}(z, \tau) = \exp(\pi i R_i/4) \cdot A \cdot B \cdot \theta_{1+i}(z', \tau')
\]

where $z' = z$, $A = B = 1$ if $C = 0$, and

\[
z' = \frac{-z}{ct + d}, \quad A = \sqrt{\frac{i}{ct + d}}, \quad B = \exp\left(-\pi ic\frac{z^2}{ct + d}\right)
\]

if $C = 1$. Note that $A$ is well-defined with the principal branch of the square root since $A^2 = i/(ct + d)$ lies in the right half-plane.
Firstly, if \( c = 0 \), we have \( \theta_i(z, \tau) = \exp(-\pi i b/4)\theta_i(z, \tau + b) \) for \( i = 1, 2 \), whereas \( \theta_3 \) 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])

\[
\begin{align*}
\theta_{0,0}(z, \tau) &= \theta_3(z, \tau), \\
\theta_{0,1}(z, \tau) &= \theta_4(z, \tau) \\
\theta_{1,0}(z, \tau) &= \theta_2(z, \tau), \\
\theta_{1,1}(z, \tau) &= i\theta_1(z, \tau) \\
\theta_{m+2,n}(z, \tau) &= (-1)^n\theta_{m,n}(z, \tau) \\
\theta_{m,n+2}(z, \tau) &= \theta_{m,n}(z, \tau).
\end{align*}
\]

Then we may write

\[
\begin{align*}
\theta_1(z, \tau) &= \varepsilon_1 A \theta_1(z', \tau') \\
\theta_2(z, \tau) &= \varepsilon_2 A \theta_1(-c,1+a)(z', \tau') \\
\theta_3(z, \tau) &= \varepsilon_3 A \theta_1(1-d-c,1-b+a)(z', \tau') \\
\theta_4(z, \tau) &= \varepsilon_4 A \theta_1(1+a,1-b)(z', \tau')
\end{align*}
\]

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 `acb_modular_epsilon_arg`), then:

\[
\begin{align*}
\varepsilon_1(a,b,c,d) &= \exp(\pi i[R(-d,b,c,-a) + 1]/4) \\
\varepsilon_2(a,b,c,d) &= \exp(\pi i[-R(a,b,c,d) + (5 + (2 - c)a)]/4) \\
\varepsilon_3(a,b,c,d) &= \exp(\pi i[-R(a,b,c,d) + (4 + (c - d - 2)(b - a))]}/4) \\
\varepsilon_4(a,b,c,d) &= \exp(\pi i[-R(a,b,c,d) + (3 - (2 + d)b)/4)
\end{align*}
\]

These formulas are easily derived from the formulas in [Rad1973] (Rademacher has the transformed/untransformed variables exchanged, and his “\( \varepsilon \)” differs from ours by a constant offset in the phase).

**void acb_modular_addseq_theta** (long * exponents, long * aindex, long * bindex, long num)

Constructs an addition sequence for the first num squares and triangular numbers interleaved (excluding zero), i.e. 1, 2, 4, 6, 9, 12, 16, 20, 25, 30 etc.

**void acb_modular_theta_sum** (acb_ptr theta1, acb_ptr theta2, acb_ptr theta3, acb_ptr theta4, const acb_t w, int w_is_unit, const acb_t q, long len, long prec)

Simultaneously computes the first len coefficients of each of the formal power series

\[
\begin{align*}
\theta_1(z+x, \tau)/q_{1/4} \in \mathbb{C}[[x]] \\
\theta_2(z+x, \tau)/q_{1/4} \in \mathbb{C}[[x]] \\
\theta_3(z+x, \tau) \in \mathbb{C}[[x]] \\
\theta_4(z+x, \tau) \in \mathbb{C}[[x]]
\end{align*}
\]

given \( w = \exp(\pi i z) \) and \( q = \exp(\pi i \tau) \), by summing a finite truncation of the respective theta function series. In particular, with len equal to 1, computes the respective value of the theta function at the point \( z \). We require len to be positive. If w_is_unit is nonzero, w is assumed to lie on the unit circle, i.e. z is assumed to be real.

Note that the factor \( q_{1/4} \) is removed from \( \theta_1 \) and \( \theta_2 \). To get the true theta function values, the user has to multiply this factor back. This convention avoids unnecessary computations, since the user can compute \( q_{1/4} = \exp(\pi i \tau/4) \) followed by \( q = (q_{1/4})^4 \), and in many cases when computing products or quotients of theta functions, the factor \( q_{1/4} \) can be eliminated entirely.

This function is intended for \( |q| \ll 1 \). It can be called with any \( q \), but will return useless intervals if convergence is not rapid. For general evaluation of theta functions, the user should only call this function after applying a suitable modular transformation.
We consider the sums together, alternatingly updating \((\theta_1, \theta_2)\) or \((\theta_3, \theta_4)\). For \(k = 0, 1, 2, \ldots\), the powers of \(q\) are \([(k + 2)^2/4] = 1, 2, 4, 6, 9\) etc. and the powers of \(w\) are ±\((k + 2) = ±2, ±3, ±4, \ldots\) etc. The scheme is illustrated by the following table:

\[
\begin{array}{c|cccc}
\theta_1, \theta_2 & q^0 & (w^1 ± w^{-1}) \\
\hline
k = 0 & \theta_3, \theta_4 & q^1 & (w^2 ± w^{-2}) \\
& 1 & \theta_1, \theta_2 & q^2 & (w^3 ± w^{-3}) \\
& 2 & \theta_3, \theta_4 & q^4 & (w^4 ± w^{-4}) \\
& 3 & \theta_1, \theta_2 & q^6 & (w^5 ± w^{-5}) \\
& 4 & \theta_3, \theta_4 & q^8 & (w^6 ± w^{-6}) \\
& 5 & & & q^{12} & (w^7 ± w^{-7}) \\
\end{array}
\]

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

\[
2Q^W W^{N+2} \left[ 1 + Q^W W + Q^F W^2 + \ldots \right] = \frac{2Q^W W^{N+2}}{1 - Q^W W}
\]

provided that the denominator is positive (otherwise we set the error bound to infinity). When \(len\) is greater than 1, consider the derivative of order \(r\). The term of index \(k\) and order \(r\) picks up a factor of magnitude \((k + 2)^r\) from differentiation of \(w^{k+2}\) (it also picks up a factor \(\pi^r\), but we omit this until we rescale the coefficients at the end of the computation). Thus we have the error bound

\[
2Q^E W^{N+2} (N + 2)^r \left[ 1 + Q^E W \frac{(N + 3)^r}{(N + 2)^r} + Q^F W^2 \frac{(N + 4)^r}{(N + 2)^r} + \ldots \right]
\]

which by the inequality \((1 + m/(N + 2))^r \leq \exp(mr/(N + 2))\) can be bounded as

\[
\frac{2Q^E W^{N+2} (N + 2)^r}{1 - Q^F W \exp(r/(N + 2))},
\]

again valid when the denominator is positive.

To actually evaluate the series, we write the even cosine terms as \(w^{2n} + w^{-2n}\), the odd cosine terms as \(w(w^{2n} + w^{-2n-2})\), and the sine terms as \(w(w^{2n} - w^{-2n-2})\). This way we only need even powers of \(w\) and \(w^{-1}\). The implementation is not yet optimized for real \(z\), in which case further work can be saved.

This function does not permit aliasing between input and output arguments.

void **acb_modular_theta_notransform** (acb_t theta1, acb_t theta2, acb_t theta3, acb_t theta4, const acb_t z, const acb_t tau, long * prec)

Evaluates the Jacobi theta functions \(\theta_i(z, \tau), i = 1, 2, 3, 4\) simultaneously. This function does not move \(\tau\) to the fundamental domain. This is generally worse than **acb_modular_theta()**, but can be slightly better for moderate input.

void **acb_modular_theta** (acb_t theta1, acb_t theta2, acb_t theta3, acb_t theta4, const acb_t z, const acb_t tau, long 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 **acb_modular_theta_sum()**.

### 3.12.4 The Dedekind \(\eta\) function

void **acb_modular_addseq_eta** (long * exponents, long * aindex, long * bindex, long num)

Constructs an addition sequence for the first \(num\) generalized pentagonal numbers (excluding zero), i.e. 1, 2, 5, 7, 12, 15, 22, 26, 35, 40 etc.
void \texttt{acb_modular_eta_sum} (acb\_t eta, const acb\_t q, long prec)

Evaluates the Dedekind eta function without the leading 24th root, i.e.

\[ \exp(-\pi i \tau / 12) \eta(\tau) = \sum_{n=-\infty}^{\infty} (-1)^n q^{(3n^2-n)/2} \]

given \( q = \exp(2\pi i \tau) \), by summing the defining series.

This function is intended for \(|q| \ll 1\). It can be called with any \( q \), but will return useless intervals if convergence is not rapid. For general evaluation of the eta function, the user should only call this function after applying a suitable modular transformation.

int \texttt{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,

\[ \eta(a \tau + b, c \tau + d) = \varepsilon(a, b, c, d) \sqrt{c \tau + d} \eta(\tau). \]

void \texttt{acb_modular_eta} (acb\_t r, const acb\_t tau, long 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}().

### 3.12.5 Modular forms

void \texttt{acb_modular_j} (acb\_t r, const acb\_t tau, long 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 \( j(\tau) = 32(\theta_2^8 + \theta_3^8 + \theta_4^8)^3 / (\theta_2 \theta_3 \theta_4)^6 \) where \( \theta_k = \theta_k(0, \tau) \).

void \texttt{acb_modular_lambda} (acb\_t r, const acb\_t tau, long 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} (acb\_t r, const acb\_t tau, long prec)

Computes the modular discriminant \( \Delta(\tau) = \eta(\tau)^{24} \), which transforms as

\[ \Delta(a \tau + b, c \tau + d) = (c \tau + d)^{12} \Delta(\tau). \]

The modular discriminant is sometimes defined with an extra factor \((2\pi)^{12}\), which we omit in this implementation.

void \texttt{acb_modular_eisenstein} (acb\_ptr r, const acb\_t tau, long len, long prec)

Computes simultaneously the first \( len \) entries in the sequence of Eisenstein series \( G_4(\tau), G_6(\tau), G_8(\tau), \ldots \), defined by

\[ G_{2k}(\tau) = \sum_{m^2 + n^2 \neq 0} \frac{1}{(m + n\tau)^{2k}} \]

and satisfying

\[ G_{2k}(a \tau + b, c \tau + d) = (c \tau + d)^{2k} G_{2k}(\tau). \]

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.
### 3.12.6 Elliptic functions

void `acb_modular_elliptic_p` (acb_t wp, const acb_t z, const acb_t tau, long prec)

Computes Weierstrass’s elliptic function

\[ \wp(z, \tau) = \frac{1}{z^2} + \sum_{n^2 + m^2 \neq 0} \left( \frac{1}{(z + m + n\tau)^2} - \frac{1}{(m + n\tau)^2} \right) \]

which satisfies \( \wp(z, \tau) = \wp(z + 1, \tau) = \wp(z + \tau, \tau) \). To evaluate the function efficiently, we use the formula

\[ \wp(z, \tau) = \pi^2 \theta^2_2(0, \tau) \theta^2_3(0, \tau) \theta^2_4(z, \tau) - \frac{\pi^2}{3} \left[ \theta^4_3(0, \tau) + \theta^4_4(0, \tau) \right] . \]

void `acb_modular_elliptic_p_zpx` (acb_ptr wp, const acb_t z, const acb_t tau, long len, long prec)

Computes the formal power series \( \wp(z + x, \tau) \in \mathbb{C}[[x]] \), truncated to length \( len \). In particular, with \( len = 2 \), simultaneously computes \( \wp(z, \tau) \), \( \wp'(z, \tau) \) which together generate the field of elliptic functions with periods 1 and \( \tau \).

### 3.12.7 Elliptic integrals

void `acb_modular_elliptic_k` (acb_t w, const acb_t m, long prec)

Computes the complete elliptic integral of the first kind \( K(m) \), using the arithmetic-geometric mean: \( K(m) = \pi/(2M(\sqrt{1-m})) \).

void `acb_modular_elliptic_k_cpx` (acb_ptr w, const acb_t m, long len, long prec)

Sets the coefficients in the array \( w \) to the power series expansion of the complete elliptic integral of the first kind at the point \( m \) truncated to length \( len \), i.e. \( K(m + x) \in \mathbb{C}[[x]] \).

### 3.13 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).

#### 3.13.1 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 Bloemen et al. http://remcobloemen.nl/2009/11/even-faster-zeta-calculation.html). 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.

```c
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.
```

3.13.2 Caching

```c
long bernoulli_cache_num

fmpz * bernoulli_cache
    Cache of Bernoulli numbers. Uses thread-local storage if enabled in FLINT.

void bernoulli_cache_compute (long n)
    Makes sure that the Bernoulli numbers up to at least \( B_{n-1} \) are cached. Calling flint_cleanup() frees the cache.
```

3.13.3 Bounding

```c
long 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_fmpz_ui_zeta (fmpz_t num, fmpz_t den, ulong 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_fmpz_ui (fmpz_t num, fmpz_t den, ulong n)

void bernoulli_fmpz_ui (fmpz_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.
```
3.14 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 \textit{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.

3.14.1 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

\[
\begin{align*}
P(k) &= a_p k^p + a_{p-1} k^{p-1} + \ldots + a_0 \\
Q(k) &= b_q k^q + b_{q-1} k^{q-1} + \ldots + b_0.
\end{align*}
\]

For convergence, we require \( p < q \), or \( p = q \) with \(|z||a_p| < |b_q|\). We also assume that \( P(k) \) and \( Q(k) \) have no roots among the positive integers (if there are positive integer roots, the sum is either finite or undefined). With these conditions satisfied, our goal is to find a parameter \( n \geq 0 \) such that

\[
\left| \sum_{k=n}^{\infty} T(k)z^k \right| \leq 2^{-d}.
\]

We can rewrite the hypergeometric term ratio as

\[ zR(k) = z \frac{P(k)}{Q(k)} = z \left( \frac{a_p}{b_q} \right) \frac{1}{k^{q-p}} F(k) \]

where

\[ F(k) = \frac{1 + \tilde{a}_1/k + \tilde{a}_2/k^2 + \ldots + \tilde{a}_q/k^p}{1 + b_1/k + b_2/k^2 + \ldots + b_q/k^q} = 1 + O(1/k) \]

and where \( \tilde{a}_i = a_{p-i}/a_p, \tilde{b}_i = b_{q-i}/b_q \). Next, we define

\[
C = \max_{1 \leq i \leq p} |\tilde{a}_i|^{1/i}, \quad D = \max_{1 \leq i \leq q} |\tilde{b}_i|^{1/i}.
\]

Now, if \( k > C \), the magnitude of the numerator of \( F(k) \) is bounded from above by

\[
1 + \sum_{i=1}^{p} \left( \frac{C}{k} \right)^i \leq 1 + \frac{C}{k-C}
\]

and if \( k > 2D \), the magnitude of the denominator of \( F(k) \) is bounded from below by

\[
1 - \sum_{i=1}^{q} \left( \frac{D}{k} \right)^i \geq 1 + \frac{D}{D-k}.
\]
Putting the inequalities together gives the following bound, valid for $k > K = \max(C, 2D)$:

$$|F(k)| \leq \frac{k(k - D)}{(k - C)(k - 2D)} = \left(1 + \frac{C}{k - C}\right)\left(1 + \frac{D}{k - 2D}\right).$$

Let $r = q - p$ and $\tilde{z} = |za_p/b_q|$. Assuming $k > \max(C, 2D, \tilde{z}^{1/r})$, we have

$$|zR(k)| \leq G(k) = \frac{\tilde{z}F(k)}{k^r}$$

where $G(k)$ is monotonically decreasing. Now we just need to find an $n$ such that $G(n) < 1$ and for which $|T(n)|/(1 - G(n)) \leq 2^{-d}$. This can be done by computing a floating-point guess for $n$ then trying successively larger values.

This strategy leaves room for some improvement. For example, if $\tilde{b}_1$ is positive and large, the bound $B$ becomes very pessimistic (a larger positive $\tilde{b}_1$ causes faster convergence, not slower convergence).

### 3.14.2 Types, macros and constants

**hypgeom_struct**

**hypgeom_t**

Stores polynomials $A$, $B$, $P$, $Q$ and precomputed bounds, representing a fixed hypergeometric series.

### 3.14.3 Memory management

void **hypgeom_init** (hypgeom_t hyp)

void **hypgeom_clear** (hypgeom_t hyp)

### 3.14.4 Error bounding

**long hypgeom_estimate_terms**(const mag_t z, int r, long 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.

**long hypgeom_bound**(mag_t error, int r, long C, long D, long K, const mag_t TK, const mag_t z, long 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

$$\frac{\tilde{z}}{k^r} \frac{k(k - D)}{(k - C)(k - 2D)}.$$

Given this information, we compute a $\varepsilon$ and an integer $n$ such that $|\sum_{k=n}^{\infty} T(k)| \leq \varepsilon \leq 2^{-\text{prec}}$. The output variable *error* is set to the value of $\varepsilon$, and $n$ is returned.

**void hypgeom_precompute**(hypgeom_t hyp)

Precomputes the bounds data $C$, $D$, $K$ and an upper bound for $T(K)$.
3.14.5 Summation

void \texttt{fmprb\_hypgeom\_sum} (\texttt{fmprb\_t }P, \texttt{fmprb\_t }Q, \texttt{const hypgeom\_t }hyp, \texttt{const long }n, \texttt{long }prec)
Computes $P/Q = \sum_{k=0}^{n-1} T(k)$ where $T(k)$ is defined by hyp, using binary splitting and a working precision of prec bits.

void \texttt{fmprb\_hypgeom\_infsum} (\texttt{fmprb\_t }P, \texttt{fmprb\_t }Q, \texttt{hypgeom\_t }hyp, \texttt{long tol}, \texttt{long prec})
Computes $P/Q = \sum_{k=0}^{\infty} T(k)$ where $T(k)$ is defined by hyp, using binary splitting and working precision of prec bits. The number of terms is chosen automatically to bound the truncation error by at most $2^{-\text{tol}}$. The bound for the truncation error is included in the output as part of $P$.

void \texttt{arb\_hypgeom\_sum} (\texttt{arb\_t }P, \texttt{arb\_t }Q, \texttt{const hypgeom\_t }hyp, \texttt{const long }n, \texttt{long }prec)
Computes $P/Q = \sum_{k=0}^{n-1} T(k)$ where $T(k)$ is defined by hyp, using binary splitting and a working precision of prec bits.

void \texttt{arb\_hypgeom\_infsum} (\texttt{arb\_t }P, \texttt{arb\_t }Q, \texttt{hypgeom\_t }hyp, \texttt{long tol}, \texttt{long prec})
Computes $P/Q = \sum_{k=0}^{\infty} T(k)$ where $T(k)$ is defined by hyp, using binary splitting and working precision of prec bits. The number of terms is chosen automatically to bound the truncation error by at most $2^{-\text{tol}}$. The bound for the truncation error is included in the output as part of $P$.

3.15 partitions.h – computation of the partition function

This module implements the asymptotically fast algorithm for evaluating the integer partition function $p(n)$ described in [Joh2012]. The idea is to evaluate a truncation of the Hardy-Ramanujan-Rademacher series using tight precision estimates, and symbolically factoring the occurring exponential sums.

An implementation based on floating-point arithmetic can also be found in FLINT. That version relies on some numerical subroutines that have not been proved correct.

The implementation provided here uses ball arithmetic throughout to guarantee a correct error bound for the numerical approximation of $p(n)$. Optionally, hardware double arithmetic can be used for low-precision terms. This gives a significant speedup for small (e.g. $n < 10^6$).

void \texttt{partitions\_rademacher\_bound} (\texttt{arf\_t }b, \texttt{const fmpz\_t }n, \texttt{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.

\texttt{partitions\_hrr\_sum\_arb} (\texttt{arb\_t }x, \texttt{const fmpz\_t }n, \texttt{long }N0, \texttt{long }N, \texttt{int use\_doubles})
Evaluates the partial sum $\sum_{k=N0}^{N-1} t(n, k)$ of the Hardy-Ramanujan-Rademacher series.
If \texttt{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 \texttt{use\_doubles} to zero gives a fully guaranteed bound.

void \texttt{partitions\_fmpz\_fmpz} (\texttt{fmpz\_t }p, \texttt{const fmpz\_t }n, \texttt{int use\_doubles})
Computes the partition function $p(n)$ using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing $p(n)$ and verifies that the ball contains a unique integer.
If $n$ is sufficiently large and a number of threads greater than 1 has been selected with \texttt{flint\_set\_num\_threads()}, the computation time will be reduced by using two threads.
See `partitions_hrr_sum_arb()` for an explanation of the `use_doubles` option.

void `partitions_fmpz_ui` (fmpz_t p, ulong n)

Computes the partition function $p(n)$ using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing $p(n)$ and verifies that the ball contains a unique integer.

void `partitions_fmpz_ui_using_doubles` (fmpz_t p, ulong n)

Computes the partition function $p(n)$, enabling the use of doubles internally. This significantly speeds up evaluation for small $n$ (e.g. $n < 10^6$), but the error bounds are not certified (see remarks for `partitions_hrr_sum_arb()`).
4.1 Algorithms for mathematical constants

Most mathematical constants are evaluated using the generic hypergeometric summation code.

4.1.1 Pi

\( \pi \) is computed using the Chudnovsky series

\[
\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k(6k)!}{k!(3k)!} \left( \frac{1}{13591409} + \frac{1}{545140134k} \right)
\]

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 \( \frac{1}{\sqrt{640320}} \) instead of \( \sqrt{640320} \) at the end.

4.1.2 Logarithms of integers

We use the formulas

\[
\log(2) = \frac{3}{4} \sum_{k=0}^{\infty} \frac{(-1)^k (k!)^2}{2^k (2k+1)!}
\]

\[
\log(10) = 46 \tanh(1/31) + 34 \tanh(1/49) + 20 \tanh(1/161)
\]

4.1.3 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

\[
S_0(x) = \sum_{k=0}^{\infty} \frac{H_k}{(k!)^2} \left( \frac{x}{2} \right)^{2k}, \quad I_0(x) = \sum_{k=0}^{\infty} \frac{1}{(k!)^2} \left( \frac{x}{2} \right)^{2k}
\]

\[
2x I_0(x) K_0(x) \sim \sum_{k=0}^{\infty} \frac{[(2k)!]^3}{(k!)^4 8^{2k} x^{2k}}.
\]
All series are evaluated using binary splitting. The first two series are evaluated simultaneously, with the summation taken up to \( k = N - 1 \) inclusive where \( N \geq \alpha n + 1 \) and \( \alpha \approx 4.9706257595442318644 \) satisfies \( \alpha (\log \alpha - 1) = 3 \). The third series is taken up to \( k = 2n - 1 \) inclusive. With these parameters, it is shown in [BJ2013] that the error is bounded by \( 24 e^{-8n} \).

### 4.1.4 Catalan’s constant

Catalan’s constant is computed using the hypergeometric series

\[
C = \sum_{k=0}^{\infty} \frac{(-1)^k 4^{k+1} (40k^2 + 56k + 19) \ [k+1]!^2 [2k+2]!^3}{(k+1)^3 (2k+1) [4k+4]!^2}
\]

### 4.1.5 Khinchin’s constant

Khinchin’s constant \( K_0 \) is computed using the formula

\[
\log K_0 = \frac{1}{\log 2} \left[ \sum_{k=2}^{N-1} \log \left( \frac{k-1}{k} \right) \log \left( \frac{k+1}{k} \right) + \sum_{n=1}^{\infty} \frac{\zeta(2n, N)}{n} \sum_{k=1}^{2n-1} \frac{(-1)^k}{k} \right]
\]

where \( N \geq 2 \) is a free parameter that can be used for tuning [BBC1997]. If the infinite series is truncated after \( n = M \), the remainder is smaller in absolute value than

\[
\sum_{n=M+1}^{\infty} \zeta(2n, N) = \sum_{n=M+1}^{\infty} \sum_{k=0}^{\infty} (k+N)^{-2n} \leq \sum_{n=M+1}^{\infty} \left( N^{-2n} + \int_{0}^{\infty} (t + N)^{-2n} dt \right)
\]

\[
= \sum_{n=M+1}^{\infty} \frac{1}{N^{2n}} \left( 1 + \frac{N}{2n-1} \right) \leq \sum_{n=M+1}^{\infty} \frac{N+1}{N^{2n}} = \frac{1}{N^{2M} (N-1)} \leq \frac{1}{N^{2M}}.
\]

Thus, for an error of at most \( 2^{-p} \) in the series, it is sufficient to choose \( M \geq p/(2 \log_2 N) \).

### 4.1.6 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 \( s = 2 \).

### 4.1.7 Apery’s constant

Apery’s constant \( \zeta(3) \) is computed using the hypergeometric series

\[
\zeta(3) = \frac{1}{64} \sum_{k=0}^{\infty} \frac{(-1)^k (205k^2 + 250k + 77)}{[(2k+1)k]^5} (k!)^{10}.
\]

### 4.2 Algorithms for gamma functions

#### 4.2.1 The Stirling series

In general, the gamma function is computed via the Stirling series

\[
\log \Gamma(z) = \left( z - \frac{1}{2} \right) \log z - z + \frac{\ln 2\pi}{2} + \sum_{k=1}^{n-1} \frac{B_{2k}}{2k(2k-1)z^{2k-1}} + R(n, z)
\]
where ([Olv1997] pp. 293-295) the remainder term is exactly

\[ R_n(z) = \int_0^\infty \frac{B_{2n}(x)}{2n(x+z)^{2n}} \, dx. \]

To evaluate the gamma function of a power series argument, we substitute \( z \to z + t \in \mathbb{C}[[t]] \).

Using the bound for \( |x + z| \) given by [Olv1997] and the fact that the numerator of the integrand is bounded in absolute value by \( 2|B_{2n}| \), the remainder can be shown to satisfy the bound

\[ |[k]R_n(z + t)| \leq 2|B_{2n}| \frac{\Gamma(2n + k - 1)}{\Gamma(k + 1)\Gamma(2n + 1)} |z| \frac{(b/|z|)^{2n+k}}{b/|z|}. \]

where \( b = \frac{1}{\cos(\arg(z)/2)} \). Note that by trigonometric identities, assuming that \( z = x + yi \), we have \( b = \sqrt{1+t^2} \)

To use the Stirling series at \( p \)-bit precision, we select parameters \( r, n \) such that the remainder \( R(n, z) \) approximately is bounded by \( 2^{-p} \). If \( |z| \) is too small for the Stirling series to give sufficient accuracy directly, we first translate to \( z + r \) using the formula \( \Gamma(z) = \Gamma(z + r)/((z + 1)(z + 2) \cdots (z + r - 1)). \)

To obtain a remainder smaller than \( 2^{-p} \), we must choose an \( r \) such that, in the real case, \( z + r > \beta p \), where \( \beta > \log(2)/(2\pi) \approx 0.11 \). 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.

### 4.2.2 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

\[
\Gamma(1/3) = \left( \frac{12\pi^4}{\sqrt{10}} \sum_{k=0}^{\infty} \frac{(6k)!(1)^k}{(k!)^3(3k)!3^k160^{16k}} \right)^{1/6}, \quad \Gamma(1/4) = \sqrt{\frac{(2\pi)^{3/2}}{\text{agm}(1, \sqrt{2})}}.
\]

An alternative formula which could be used for \( \Gamma(1/3) \) is

\[
\Gamma(1/3) = \frac{2^{4/9}\pi^{2/3}}{3^{1/12} \left( \frac{1}{2} \sqrt{2 + \sqrt{3}} \right)^{1/3}},
\]

but this appears to be slightly slower in practice.
4.3 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 \( z = 1 \).

4.3.1 Computation for small \( z \)

The power sum converges rapidly when \( |z| \ll 1 \). To compute the series expansion with respect to \( s \), we substitute \( s \to s + x \in \mathbb{C}[\![x]\!] \) and obtain

\[
\text{Li}_{s+x}(z) = \sum_{d=0}^{\infty} x^d \frac{(-1)^d}{d!} \sum_{k=1}^{\infty} T(k)
\]

where

\[
T(k) = \frac{z^k \log^d(k)}{k^s}.
\]

The remainder term \(|\sum_{k=N}^{\infty} T(k)|\) is bounded via \( \text{mag\_polylog\_tail()} \).

4.3.2 Expansion for general \( z \)

For general complex \( s, z \), we write the polylogarithm as a sum of two Hurwitz zeta functions

\[
\text{Li}_s(z) = \frac{\Gamma(v)}{(2\pi)^v} \left[ i^v \zeta \left( v, \frac{1}{2} + \frac{\log(-z)}{2\pi i} \right) + i^{-v} \zeta \left( v, \frac{1}{2} - \frac{\log(-z)}{2\pi i} \right) \right]
\]

in which \( s = 1 - v \). With the principal branch of \( \log(-z) \), we obtain the conventional analytic continuation of the polylogarithm with a branch cut on \( z \in (1, +\infty) \).

To compute the series expansion with respect to \( v \), we substitute \( v \to v + x \in \mathbb{C}[\![x]\!] \) in this formula (at the end of the computation, we map \( x \to -x \) to obtain the power series for \( \text{Li}_{s+x}(z) \)). The right hand side becomes

\[
\Gamma(v+x)[E_1 Z_1 + E_2 Z_2]
\]

where \( E_1 = (i/(2\pi))^v Z_1 = \zeta(v+x, \ldots) \), \( E_2 = (1/(2\pi i))^v Z_2 = \zeta(v+x, \ldots) \).

When \( v = 1 \), the \( Z_1 \) and \( Z_2 \) terms become Laurent series with a leading \( 1/x \) term. In this case, we compute the deflated series \( \tilde{Z}_1, \tilde{Z}_2 = \zeta(x, \ldots) - 1/x \). Then

\[
E_1 Z_1 + E_2 Z_2 = (E_1 + E_2)/x + E_1 \tilde{Z}_1 + E_2 \tilde{Z}_2.
\]

Note that \((E_1 + E_2)/x\) is a power series, since the constant term in \( E_1 + E_2 \) is zero when \( v = 1 \). So we simply compute one extra derivative of both \( E_1 \) and \( E_2 \), and shift them one step. When \( v = 0, -1, -2, \ldots \), the \( \Gamma(v+x) \) prefactor has a pole. In this case, we proceed analogously and formally multiply \( x \Gamma(v+x) \) with \( |E_1 Z_1 + E_2 Z_2|/x \).

Note that the formal cancellation only works when the order \( s \) (or \( v \)) is an exact integer: it is not currently possible to use this method when \( s \) is a small ball containing any of 0, 1, 2, \ldots (then the result becomes indeterminate).

The Hurwitz zeta method becomes inefficient when \(|z| \to 0\) (it gives an indeterminate result when \( z = 0 \)). This is not a problem since we just use the defining series for the polylogarithm in that region. It also becomes inefficient when \(|z| \to \infty\), for which an asymptotic expansion would better.
5.1 fmpr.h – arbitrary-precision floating-point numbers

This type is now obsolete: use arf_t instead.

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

The component $x$ is called the mantissa, and $y$ is called the exponent. Note that this is just one among many possible conventions: the mantissa (alternatively significand) is sometimes viewed as a fraction in the interval $[1/2, 1)$, with the exponent pointing to the position above the top bit rather than the position of the bottom bit, and with a separate sign.

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

An fmpr number is exact and has no inherent “accuracy”. We use the term precision to denote either the target precision of an operation, or the bit size of a mantissa (which in general is unrelated to the “accuracy” of the number: for example, the floating-point value 1 has a precision of 1 bit in this sense and is simultaneously an infinitely accurate approximation of the integer 1 and a 2-bit accurate approximation of $\sqrt{2} = 1.011010100\ldots2$).

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.

5.1.1 Types, macros and constants

fmpr_struct

An fmpr_struct holds a mantissa and an exponent. If the mantissa and exponent are sufficiently small, their values are stored as immediate values in the fmpr_struct; large values are represented by pointers to heap-allocated arbitrary-precision integers. Currently, both the mantissa and exponent are implemented using the FLINT fmpz type. Special values are currently encoded by the mantissa being set to zero.

fmpr_t

An fmpr_t is defined as an array of length one of type fmpr_struct, permitting an fmpr_t to be passed by reference.

fmpr_rnd_t

Specifies the rounding mode for the result of an approximate operation.
**FMPR_RND_DOWN**
Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards zero.

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

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

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

**FMPR_RND_NEAR**
Specifies that the result of an operation should be rounded to the nearest representable number, rounding to an odd mantissa if there is a tie between two values. **Warning:** this rounding mode is currently not implemented (except for a few conversions functions where this stated explicitly).

**FMPR_PREC_EXACT**
If passed as the precision parameter to a function, indicates that no rounding is to be performed. This must only be used when it is known that the result of the operation can be represented exactly and fits in memory (the typical use case is working small integer values). Note that, for example, adding two numbers whose exponents are far apart can easily produce an exact result that is far too large to store in memory.

### 5.1.2 Memory management

```c
void fmpfr_init (fmpr_t x)
    Initializes the variable x for use. Its value is set to zero.
```

```c
void fmpfr_clear (fmpr_t x)
    Clears the variable x, freeing or recycling its allocated memory.
```

### 5.1.3 Special values

```c
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, +∞, −∞, NaN.
```

```c
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, +∞, −∞, NaN.
```
int fmpr_is_inf (const fmpr_t *x)
Returns nonzero iff \( x \) equals either \(+\infty\) or \(-\infty\).

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

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

int fmpr_is_finite (fmpr_t *x)
Returns nonzero iff \( x \) is a finite floating-point value, i.e. not one of the values \(+\infty\), \(-\infty\), NaN. (Note that this is not equivalent to the negation of \( \text{fmpr_is_inf()} \).)

### 5.1.4 Assignment, rounding and conversions

void fmpr_swap (fmpr_t *x, fmpr_t *y)
Swaps \( x \) and \( y \) efficiently.

long _fmpr_set_round_mpn (long *shift, fmpz_t man, mp_srcptr x, mp_size_t xn, int negative, long 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.

long fmpr_set_round_ui_2exp_fmpz (fmpr_t *z, mp_limb_t lo, const fmpz_t exp, int negative, long 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 \( \text{rnd} \).

void fmpr_set_error_result (fmpr_t err, const fmpr_t result, long ret)  
Given the return value \( \text{ret} \) 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, long ret, long prec, fmpr_rnd_t rnd)  
Like \( \text{fmpr_set_error_result} \), but adds \( \text{err_in} \) to the error.

void fmprulp (fmpr_t u, const fmpr_t *x, long 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, long r, long prec)
    Assume that r is the return code and x is the floating-point result from a single floating-point rounding. Then this function returns nonzero iff x and r define an error of exactly 0 or 1 ulp. In other words, this function checks that fmpr_set_error_result () gives exactly 0 or 1 ulp as expected.

int fmpr_get_mpfr (mpfr_t x, const mpfr_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 the the exact value of the argument v of type double.

void fmpr_set_ui (fmpr_t x, ulong c)

void fmpr_set_si (fmpr_t x, long c)

void fmpr_set_fmpz (fmpr_t x, const fmpz_t c)
    Sets x exactly to the integer c.

void fmpr_get_fmpz (fmpz_t z, const fmpr_t x, mpfr_rnd_t rnd)
    Sets z to x rounded to the nearest integer in the direction specified by rnd. If rnd is FMPR_RND_NEAR, rounds to the nearest even integer in case of a tie. Aborts if x is infinite, NaN or if the exponent is unreasonably large.

long fmpr_get_si (const fmpr_t x, mpfr_rnd_t rnd)
    Returns x rounded to the nearest integer in the direction specified by rnd. If rnd is FMPR_RND_NEAR, rounds to the nearest even integer in case of a tie. Aborts if x is infinite, NaN, or the value is too large to fit in a long.

void fmpr_get_fmpq (fmpq_t y, const fmpr_t x)
    Sets y to the exact value of x. The result is undefined if x is not a finite fraction.

long fmpr_get_fmpq (fmpr_t x, const fmpq_t y, long prec, mpfr_rnd_t rnd)
    Sets x to the value of y, rounded according to prec and rnd.

void fmpr_set_fmpz_2exp (fmpr_t x, const fmpz_t man, const fmpz_t exp)

void fmpr_set_si_2exp_si (fmpr_t x, long man, long exp)

void fmpr_set_ui_2exp_si (fmpr_t x, ulong man, long exp)
    Sets x to man \times 2^{exp}.

long fmpr_set_round_fmpz_2exp (fmpr_t x, const fmpz_t man, const fmpz_t exp, long prec, mpfr_rnd_t rnd)
    Sets x to man \times 2^{exp}, rounded according to prec and rnd.

void fmpr_get_fmpq_2exp (fmpz_t man, fmpq_t exp, const fmpr_t x)
    Sets man and exp to the unique integers such that x = man \times 2^{exp} and man is odd, provided that x is a nonzero finite fraction. If x is zero, both man and exp are set to zero. If x is infinite or NaN, the result is undefined.

int fmpr_get_fmpz_fixed_mpfr (fmpz_t y, const fmpr_t x, const fmpz_t e)

int fmpr_get_fmpz_fixed_si (fmpz_t y, const fmpr_t x, long e)
    Converts x to a mantissa with predetermined exponent, i.e. computes an integer y such that \( y \times 2^e \approx x \), truncating if necessary. Returns 0 if exact and 1 if truncation occurred.
5.1.5 Comparisons

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

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

int fmpr_cmpabs (const fmpr_t x, const fmpr_t y)
    Compares the absolute values of x and y.

int fmpr_cmpabs_ui (const fmpr_t x, ulong y)
    Compares x (respectively its absolute value) with 2^e.

int fmpr_cmp_2exp_si (const fmpr_t x, long e)
    Compares x (respectively its absolute value) with 2^e.

int fmpr_cmpabs_2exp_si (const fmpr_t x, long 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)
    Sets z respectively to the minimum and the maximum of a and b.

long 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, long e)
    Returns nonzero iff x equals n2^e for some integer n.

void fmpr_abs_bound_le_2exp_fmpz (fmpz_t b, const fmpr_t x)
    Sets b to the smallest integer such that |x| ≤ 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.

long 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 FMURP_PREC_EXACT and FMURP_PREC_EXACT inclusive. If x is zero, FMURP_PREC_EXACT is returned, and if x is infinity or NaN, FMURP_PREC_EXACT is returned.

5.1.6 Random number generation

void fmpr_randtest (fmpr_t x, flint_rand_t state, long bits, long 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, long bits, long 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, long bits, long mag_bits)
    Identical to fmpr_randtest, except that the output occasionally is set to an infinity or NaN.
5.1.7 Input and output

void **fmp**_**r_print** (const **fmp**_**r_t** *x)
Prints the mantissa and exponent of *x* as integers, precisely showing the internal representation.

void **fmp**_**r_printd** (const **fmp**_**r_t** *x, long **d**igits)
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.

5.1.8 Arithmetic

void **fmp**_**r_neg** (**fmp**_**r_t** *y, const **fmp**_**r_t** *x)
Sets *y* to the negation of *x*.

long **fmp**_**r_neg_round** (const **fmp**_**r_t** *x, long **p**rec, **fmp**_**r_d**rnd_t *r**nd)
Sets *y* to the negation of *x*, rounding the result.

void **fmp**_**r_abs** (const **fmp**_**r_t** *y, const **fmp**_**r_t** *x)
Sets *y* to the absolute value of *x*.

long **fmp**_**r_add** (**fmp**_**r_t** *z, const **fmp**_**r_t** *x, const **fmp**_**r_t** *y, long **p**rec, **fmp**_**r_d**rnd_t *r**nd)
Sets *z* = *x* + *y*, rounded according to **prec** and **rnd**. The precision can be **FMPR_PREC_EXACT** to perform an exact addition, provided that the result fits in memory.

long **fmp**_**r_add_round** (**fmp**_**r_t** *z, const **fmp**_**r_t** *x, const **fmp**_**r_t** *y, long **p**rec, **fmp**_**r_d**rnd_t *r**nd)
Sets *z* = *x* + *y*, rounded according to **prec** and **rnd**. The precision can be **FMPR_PREC_EXACT** to perform an exact addition, provided that the result fits in memory.

long **fmp**_**r_add_ui** (**fmp**_**r_t** *z, const **fmp**_**r_t** *x, ulong *y, long **p**rec, **fmp**_**r_d**rnd_t *r**nd)
Sets *z* = *x* + *y*, rounded according to **prec** and **rnd**. The precision can be **FMPR_PREC_EXACT** to perform an exact addition, provided that the result fits in memory.

long **_fmp**_**r_add_eps** (**fmp**_**r_t** *z, const **fmp**_**r_t** *x, int **s**ign, long **p**rec, **fmp**_**r_d**rnd_t *r**nd)
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.

long **fmp**_**r_sub** (**fmp**_**r_t** *z, const **fmp**_**r_t** *x, const **fmp**_**r_t** *y, long **p**rec, **fmp**_**r_d**rnd_t *r**nd)
Sets *z* = *x* − *y*, rounded according to **prec** and **rnd**. The precision can be **FMPR_PREC_EXACT** to perform an exact addition, provided that the result fits in memory.

long **fmp**_**r_sub_ui** (**fmp**_**r_t** *z, const **fmp**_**r_t** *x, ulong *y, long **p**rec, **fmp**_**r_d**rnd_t *r**nd)
Sets *z* = *x* − *y*, rounded according to **prec** and **rnd**. The precision can be **FMPR_PREC_EXACT** to perform an exact addition, provided that the result fits in memory.

long **fmp**_**r_subui** (**fmp**_**r_t** *z, const **fmp**_**r_t** *x, long *y, long **p**rec, **fmp**_**r_d**rnd_t *r**nd)
Sets *z* = *x* − *y*, rounded according to **prec** and **rnd**. The precision can be **FMPR_PREC_EXACT** to perform an exact addition, provided that the result fits in memory.

long **fmp**_**r_sum** (const **fmp**_**r_t** *x, const **fmp**_**r**tstruct * **t**erms, long **l**en, long **p**rec, **fmp**_**r_d**rnd_t *r**nd)
Sets *s* to the sum of the array **terms** of length **len**, rounded to **prec** bits in the direction **rnd**. The sum is computed as if done without any intermediate rounding error, with only a single rounding applied to the final result. Unlike repeated calls to **fmp**_**r_add**, this function does not overflow if the magnitudes of the terms are far apart. Warning: this function is implemented naively, and the running time is quadratic with respect to **len** in the worst case.

long **fmp**_**r_mul** (**fmp**_**r_t** *z, const **fmp**_**r_t** *x, const **fmp**_**r_t** *y, long **p**rec, **fmp**_**r_d**rnd_t *r**nd)
Sets *z* = *x* × *y*, rounded according to **prec** and **rnd**. The precision can be **FMPR_PREC_EXACT** to perform an exact multiplication, provided that the result fits in memory.

void **fmp**_**r_mul_2exp_si** (**fmp**_**r_t** *y, const **fmp**_**r_t** *x, long **e**)

Sets *y* = *x* × 2^**e**.
void \texttt{fmpr\_mul\_2exp\_fmpz} (\texttt{fmpr\_t y}, \texttt{const fmpr\_t x}, \texttt{const fmpz\_t e})

Sets \( y \) to \( x \) multiplied by \( 2^e \) without rounding.

long \texttt{fmpr\_div} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{const fmpr\_t y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_div\_ui} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{ulong y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_ui\_div} (\texttt{fmpr\_t z}, \texttt{ulong x}, \texttt{const fmpr\_t y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_div\_si} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{long y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_si\_div} (\texttt{fmpr\_t z}, \texttt{long x}, \texttt{const fmpr\_t y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_div\_fmpz} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{const fmpr\_t y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_fmpz\_div\_fmpz} (\texttt{fmpr\_t z}, \texttt{const fmpz\_t x}, \texttt{const fmpz\_t y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

Sets \( z = x/y \), rounded according to \( prec \) and \( rnd \). If \( y \) is zero, \( z \) is set to NaN.

void \texttt{fmpr\_divappr\_abs\_ubound} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{const fmpr\_t y}, \texttt{long 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.

long \texttt{fmpr\_addmul} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{const fmpr\_t y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_addmul\_ui} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{ulong y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_addmul\_si} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{long y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_addmul\_fmpz} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{const fmpr\_t y}, \texttt{long prec}, \texttt{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 \texttt{FMPR\_PREC\_EXACT} to perform an exact addition, provided that the result fits in memory.

long \texttt{fmpr\_submul} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{const fmpr\_t y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_submul\_ui} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{ulong y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_submul\_si} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{long y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_submul\_fmpz} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{const fmpr\_t y}, \texttt{long prec}, \texttt{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 \texttt{FMPR\_PREC\_EXACT} to perform an exact subtraction, provided that the result fits in memory.

long \texttt{fmpr\_sqrt} (\texttt{fmpr\_t y}, \texttt{const fmpr\_t x}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

long \texttt{fmpr\_sqrt\_fmpz} (\texttt{fmpr\_t z}, \texttt{const fmpz\_t x}, \texttt{long prec}, \texttt{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.

long \texttt{fmpr\_rsqrt} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

Sets \( z \) to the reciprocal square root of \( x \), rounded according to \( prec \) and \( rnd \). The result is NaN if \( x \) is negative. At high precision, this is faster than computing a square root.

long \texttt{fmpr\_root} (\texttt{fmpr\_t z}, \texttt{const fmpr\_t x}, \texttt{ulong k}, \texttt{long prec}, \texttt{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 \texttt{fmpr\_pow\_sloppy\_fmpz} (\texttt{fmpr\_t y}, \texttt{const fmpr\_t x}, \texttt{const fmpz\_t e}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})

void \texttt{fmpr\_pow\_sloppy\_ui} (\texttt{fmpr\_t y}, \texttt{const fmpr\_t x}, \texttt{ulong e}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})
void `fmpr_pow_sloppy_si` (fmpr_t y, const fmpr_t b, long e, long 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 $b \geq 0$.

5.1.9 Special functions

long `fmpr_log` (fmpr_t y, const fmpr_t x, long 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.

long `fmpr_log1p` (fmpr_t y, const fmpr_t x, long prec, fmpr_rnd_t rnd)
Sets $y$ to $\log(1 + x)$, rounded according to `prec` and `rnd`. This function computes an accurate value when $x$ is
small. The result is NaN if $1 + x$ is negative. This function is currently implemented using MPFR and does not
support large exponents.

long `fmpr_exp` (fmpr_t y, const fmpr_t x, long prec, fmpr_rnd_t rnd)
Sets $y$ to $\exp(x)$, rounded according to `prec` and `rnd`. This function is currently implemented using MPFR and
does not support large exponents.

long `fmpr_expm1` (fmpr_t y, const fmpr_t x, long prec, fmpr_rnd_t rnd)
Sets $y$ to $\exp(x) - 1$, rounded according to `prec` and `rnd`. This function computes an accurate value when $x$ is
small. This function is currently implemented using MPFR and does not support large exponents.

5.2 `fmprb.h` – real numbers represented as floating-point balls

This type is now obsolete: use `arb_t` instead.
An `fmprb_t` represents a ball over the real numbers.

5.2.1 Types, macros and constants

`fmprb_struct`

`fmprb_t`
An `fmprb_struct` consists of a pair of `fmpr_struct`s. An `fmprb_t` is defined as an array of length one of type
`fmprb_struct`, permitting an `fmprb_t` to be passed by reference.

`fmprb_ptr`
Alias for `fmprb_struct *`, used for vectors of numbers.

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

`FMPRB_RAD_PREC`
The precision used for operations on the radius. This is small enough to fit in a single word, currently 30 bits.

`fmprb_midref` ($x$)
Macro returning a pointer to the midpoint of $x$ as an `fmpr_t`.

`fmprb_radref` ($x$)
Macro returning a pointer to the radius of $x$ as an `fmpr_t`. 
5.2.2 Memory management

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

void fmprb_clear (fmprb_t x)
Clears the variable x, freeing or recycling its allocated memory.

fmprb_ptr _fmprb_vec_init (long n)
Returns a pointer to an array of n initialized fmprb_struct:s.

void _fmprb_vec_clear (fmprb_ptr v, long n)
Clears an array of n initialized fmprb_struct:s.

5.2.3 Assignment and rounding

void fmprb_set (fmprb_t y, const fmprb_t x)
Sets y to a copy of x.

void fmprb_set_round (fmprb_t y, const fmprb_t x, long prec)
Sets y to a copy of x, rounded to prec bits.

void fmprb_set_fmpr (fmprb_t y, const fmpr_t x)
void fmprb_set_si (fmprb_t y, long x)
void fmprb_set_ui (fmprb_t y, ulong x)
void fmprb_set_fmpz (fmprb_t y, const fmpz_t x)
Sets y exactly to x.

void fmprb_set_fmpq (fmprb_t y, const fmpq_t x, long prec)
Sets y to the rational number x, rounded to prec bits.

void fmprb_set_fmpz_2exp (fmprb_t x, const fmpz_t y, const fmpz_t exp)
Sets x to y multiplied by 2 raised to the power exp.

void fmprb_set_round_fmpz_2exp (fmprb_t y, const fmpz_t x, const fmpz_t exp, long prec)
Sets x to y multiplied by 2 raised to the power exp, rounding the result to prec bits.

5.2.4 Assignment of special values

void fmprb_zero (fmprb_t x)
Sets x to zero.

void fmprb_one (fmprb_t x)
Sets x to the exact integer 1.

void fmprb_pos_inf (fmprb_t x)
Sets x to positive infinity, with a zero radius.

void fmprb_neg_inf (fmprb_t x)
Sets x to negative infinity, with a zero radius.

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

void fmprb_indeterminate (fmprb_t x)
Sets x to \([\text{NaN} \pm \infty]\), representing an indeterminate result.
5.2.5 Input and output

void \texttt{fmprb\_print}(\texttt{const fmprb\_t x})
Prints the internal representation of x.

void \texttt{fmprb\_printd}(\texttt{const fmprb\_t x, long 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.

5.2.6 Random number generation

void \texttt{fmprb\_randtest}(\texttt{fmprb\_t x, flint\_rand\_t state, long prec, long mag\_bits})
Generates a random ball. The midpoint and radius will both be finite.

void \texttt{fmprb\_randtest\_exact}(\texttt{fmprb\_t x, flint\_rand\_t state, long prec, long mag\_bits})
Generates a random number with zero radius.

void \texttt{fmprb\_randtest\_precise}(\texttt{fmprb\_t x, flint\_rand\_t state, long prec, long mag\_bits})
Generates a random number with radius at most \(2^{-\text{prec}}\) the magnitude of the midpoint.

void \texttt{fmprb\_randtest\_wide}(\texttt{fmprb\_t x, flint\_rand\_t state, long prec, long mag\_bits})
Generates a random number with midpoint and radius chosen independently, possibly giving a very large interval.

void \texttt{fmprb\_randtest\_special}(\texttt{fmprb\_t x, flint\_rand\_t state, long prec, long mag\_bits})
Generates a random interval, possibly having NaN or an infinity as the midpoint and possibly having an infinite radius.

void \texttt{fmprb\_get\_rand\_fmpq}(\texttt{fmpq\_t q, flint\_rand\_t state, const fmprb\_t x, long 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 \(\text{bits}\) bits.

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

5.2.7 Radius and interval operations

void \texttt{fmprb\_add\_error\_fmp}\_r(\texttt{fmprb\_t x, const fmp\_t err})
Adds \textit{err}, which is assumed to be nonnegative, to the radius of \textit{x}.

void \texttt{fmprb\_add\_error\_2exp\_si}(\texttt{fmprb\_t x, long e})
Adds \(2^e\) to the radius of \textit{x}.

void \texttt{fmprb\_add\_error\_2exp\_fmpz}(\texttt{fmprb\_t x, const fmpz\_t e})
Adds \(2^e\) to the radius of \textit{x}.

void \texttt{fmprb\_add\_error}(\texttt{fmprb\_t x, const fmprb\_t err})
Adds the supremum of \textit{err}, which is assumed to be nonnegative, to the radius of \textit{x}.

void \texttt{fmprb\_union}(\texttt{fmprb\_t z, const fmprb\_t x, const fmprb\_t y, long prec})
Sets \textit{z} to a ball containing both \textit{x} and \textit{y}.

void \texttt{fmprb\_get\_abs\_ubound\_fmp}\_r(\texttt{fmp\_t u, const fmprb\_t x, long prec})
Sets \textit{u} to the upper bound of the absolute value of \textit{x}, rounded up to \textit{prec} bits. If \textit{x} contains NaN, the result is NaN.

void \texttt{fmprb\_get\_abs\_lbound\_fmp}\_r(\texttt{fmp\_t u, const fmprb\_t x, long prec})
Sets \textit{u} to the lower bound of the absolute value of \textit{x}, rounded down to \textit{prec} bits. If \textit{x} contains NaN, the result is NaN.
void \texttt{fmprb\_get\_interval\_fmpz\_2exp}\ (\texttt{fmpz\_t a, fmpz\_t b, fmpz\_t exp, const fmprb\_t x})
\begin{itemize}
  \item Computes the exact interval represented by \(x\), in the form of an integer interval multiplied by a power of two, i.e. \(x = [a, b] \times 2^{\text{exp}}\).
  \item The outcome is undefined if the midpoint or radius of \(x\) is non-finite, or if the difference in magnitude between the midpoint and radius is so large that representing the endpoints exactly would cause overflows.
\end{itemize}

void \texttt{fmprb\_set\_interval\_fmpr}\ (\texttt{fmprb\_t x, const fmpr\_t a, const fmpr\_t b, long prec})
\begin{itemize}
  \item Sets \(x\) to a ball containing the interval \([a, b]\). We require that \(a \leq b\).
\end{itemize}

long \texttt{fmprb\_rel\_error\_bits}\ (\texttt{const fmprb\_t x})
\begin{itemize}
  \item 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 FMPR\_PREC\_EXACT.
\end{itemize}

long \texttt{fmprb\_rel\_accuracy\_bits}\ (\texttt{const fmprb\_t x})
\begin{itemize}
  \item Returns the effective relative accuracy of \(x\) measured in bits, equal to the negative of the return value from \texttt{fmprb\_rel\_error\_bits}.
\end{itemize}

long \texttt{fmprb\_bits}\ (\texttt{const fmprb\_t x})
\begin{itemize}
  \item 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.
\end{itemize}

void \texttt{fmprb\_trim}\ (\texttt{fmprb\_t y, const fmprb\_t x})
\begin{itemize}
  \item 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.
\end{itemize}

int \texttt{fmprb\_get\_unique\_fmpz}\ (\texttt{fmpz\_t z, const fmprb\_t x})
\begin{itemize}
  \item 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.
\end{itemize}

5.2.8 Comparisons

int \texttt{fmprb\_is\_zero}\ (\texttt{const fmprb\_t x})
\begin{itemize}
  \item Returns nonzero iff the midpoint and radius of \(x\) are both zero.
\end{itemize}

int \texttt{fmprb\_is\_nonzero}\ (\texttt{const fmprb\_t x})
\begin{itemize}
  \item Returns nonzero iff zero is not contained in the interval represented by \(x\).
\end{itemize}

int \texttt{fmprb\_is\_one}\ (\texttt{const fmprb\_t x})
\begin{itemize}
  \item Returns nonzero iff \(x\) is exactly 1.
\end{itemize}

int \texttt{fmprb\_is\_finite}\ (\texttt{fmprb\_t x})
\begin{itemize}
  \item Returns nonzero iff the midpoint and radius of \(x\) are both finite floating-point numbers, i.e. not infinities or NaN.
\end{itemize}

int \texttt{fmprb\_is\_exact}\ (\texttt{const fmprb\_t x})
\begin{itemize}
  \item Returns nonzero iff the radius of \(x\) is zero.
\end{itemize}

int \texttt{fmprb\_is\_int}\ (\texttt{const fmprb\_t x})
\begin{itemize}
  \item Returns nonzero iff \(x\) is an exact integer.
\end{itemize}

int \texttt{fmprb\_equal}\ (\texttt{const fmprb\_t x, const fmprb\_t y})
\begin{itemize}
  \item Returns nonzero iff \(x\) and \(y\) are equal as balls, i.e. have both the same midpoint and radius.
  \item Note that this is not the same thing as testing whether both \(x\) and \(y\) certainly represent the same real number, unless either \(x\) or \(y\) is exact (and neither contains NaN). To test whether both operands might represent the same mathematical quantity, use \texttt{fmprb\_overlaps()} or \texttt{fmprb\_contains()}, depending on the circumstance.
\end{itemize}

int \texttt{fmprb\_is\_positive}\ (\texttt{const fmprb\_t x})
\begin{itemize}
\end{itemize}
int fmprb_is_nonnegative (const fmprb_t x)
int fmprb_is_negative (const fmprb_t x)
int fmprb_is_nonpositive (const fmprb_t x)

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

int fmprb_overlaps (const fmprb_t x, const fmprb_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 fmprb_contains_fmp (const fmprb_t x, const fmpr_t y)
int fmprb_contains_fmpq (const fmprb_t x, const fmpq_t y)
int fmprb_contains_fmpz (const fmprb_t x, const fmpz_t y)
int fmprb_contains_si (const fmprb_t x, long y)
int fmprb_contains_mpfr (const fmprb_t x, const mpfr_t y)
int fmprb_contains_zero (const fmprb_t x)

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 fmprb_contains_negative (const fmprb_t x)
int fmprb_contains_nonpositive (const fmprb_t x)
int fmprb_contains_positive (const fmprb_t x)
int fmprb_contains_nonnegative (const fmprb_t x)

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

5.2.9 Arithmetic

void fmprb_neg (fmprb_t y, const fmprb_t x)
Sets $y$ to the negation of $x$.

void fmprb_abs (fmprb_t y, const fmprb_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 fmprb_add (fmprb_t z, const fmprb_t x, const fmprb_t y, long prec)
void fmprb_add_ui (fmprb_t z, const fmprb_t x, ulong y, long prec)
void fmprb_add_si (fmprb_t z, const fmprb_t x, long y, long prec)
void fmprb_add_fmpz (fmprb_t z, const fmprb_t x, const fmpz_t y, long prec)
void fmprb_add_mpfr (fmprb_t z, const fmprb_t x, const mpfr_t y, long prec)

Sets $z = x + y$, rounded to $prec$ bits. The precision can be FMPR_PREC_EXACT provided that the result fits in memory.

void fmprb_sub (fmprb_t z, const fmprb_t x, const fmprb_t y, long prec)
void fmprb_sub_ui (fmprb_t z, const fmprb_t x, ulong y, long prec)
void \texttt{fmprb\_sub\_si} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, long \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_sub\_fmpz} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmpz\_t} \texttt{y}, long \texttt{prec})

Sets \( z = x - y \), rounded to \texttt{prec} bits. The precision can be \texttt{FMPR\_PREC\_EXACT} provided that the result fits in memory.

void \texttt{fmprb\_mul} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmprb\_t} \texttt{y}, long \texttt{prec})

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

void \texttt{fmprb\_mul\_si} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmpz\_t} \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_mul\_fmpz} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmpz\_t} \texttt{y}, long \texttt{prec})

Sets \( z = x \times y \), rounded to \texttt{prec} bits. The precision can be \texttt{FMPR\_PREC\_EXACT} provided that the result fits in memory.

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

void \texttt{fmprb\_mul\_2exp\_fmpz} (\texttt{fmprb\_t} \texttt{y}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmpz\_t} \texttt{e})

Sets \( y \) to \( x \) multiplied by \( 2^e \).

void \texttt{fmprb\_inv} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, long \texttt{prec})

Sets \( z \) to the multiplicative inverse of \( x \).

void \texttt{fmprb\_div} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmprb\_t} \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_div\_ui} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, ulong \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_div\_si} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmpz\_t} \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_div\_fmpz} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmpz\_t} \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_div\_fmpfr} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmpfr\_t} \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_fmpfr\_div\_fmpz} (\texttt{fmprb\_t} \texttt{y}, const \texttt{fmpfr\_t} \texttt{num}, const \texttt{fmpz\_t} \texttt{den}, long \texttt{prec})

void \texttt{fmprb\_ui\_div} (\texttt{fmprb\_t} \texttt{z}, ulong \texttt{x}, const \texttt{fmprb\_t} \texttt{y}, long \texttt{prec})

Sets \( z = x/y \), rounded to \texttt{prec} bits. If \( y \) contains zero, \( z \) is set to \( 0 \pm \infty \). Otherwise, error propagation uses the rule

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

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 \texttt{fmprb\_div\_2exp\_mul\_ui} (\texttt{fmprb\_t} \texttt{y}, const \texttt{fmprb\_t} \texttt{x}, ulong \texttt{n}, long \texttt{prec})

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

void \texttt{fmprb\_add\_mul} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmprb\_t} \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_add\_mul\_ui} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, ulong \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_add\_mul\_si} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmpz\_t} \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_add\_mul\_fmpz} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmpz\_t} \texttt{y}, long \texttt{prec})

Sets \( z = z + x \times y \), rounded to \texttt{prec} bits. The precision can be \texttt{FMPR\_PREC\_EXACT} provided that the result fits in memory.

void \texttt{fmprb\_sub\_mul} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmprb\_t} \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_sub\_mul\_ui} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, ulong \texttt{y}, long \texttt{prec})

void \texttt{fmprb\_sub\_mul\_si} (\texttt{fmprb\_t} \texttt{z}, const \texttt{fmprb\_t} \texttt{x}, const \texttt{fmpz\_t} \texttt{y}, long \texttt{prec})
void \texttt{fmprb\_submul\_fmpz} (\texttt{fmprb\_t} \(z\), \texttt{const fmprb\_t} \(x\), \texttt{const fmpz\_t} \(y\), \texttt{long} \(prec\))

Sets \(z = z - x \times y\), rounded to \(prec\) bits. The precision can be \texttt{FMPR\_PREC\_EXACT} provided that the result fits in memory.

### 5.2.10 Powers and roots

void \texttt{fmprb\_sqrt} (\texttt{fmprb\_t} \(z\), \texttt{const fmprb\_t} \(x\), \texttt{long} \(prec\))

void \texttt{fmprb\_sqrt\_ui} (\texttt{fmprb\_t} \(z\), \texttt{ulong} \(x\), \texttt{long} \(prec\))

void \texttt{fmprb\_sqrt\_fmpz} (\texttt{fmprb\_t} \(z\), \texttt{const fmpz\_t} \(x\), \texttt{long} \(prec\))

Sets \(z\) to the square root of \(x\), rounded to \(prec\) bits. Error propagation is done using the following rule: assuming \(m > r \geq 0\), the error is largest at \(m - r\), and we have \(\sqrt{m} - \sqrt{m - r} \leq r/(2\sqrt{m - r})\).

void \texttt{fmprb\_sqrt\_pos} (\texttt{fmprb\_t} \(z\), \texttt{const fmprb\_t} \(x\), \texttt{long} \(prec\))

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), and producing an output interval not containing any negative numbers (unless the radius is infinite).

void \texttt{fmprb\_hypot} (\texttt{fmprb\_t} \(z\), \texttt{const fmprb\_t} \(x\), \texttt{const fmprb\_t} \(y\), \texttt{long} \(prec\))

Sets \(z\) to \(\sqrt{x^2 + y^2}\).

void \texttt{fmprb\_rsqrt} (\texttt{fmprb\_t} \(z\), \texttt{const fmprb\_t} \(x\), \texttt{long} \(prec\))

void \texttt{fmprb\_rsqrt\_ui} (\texttt{fmprb\_t} \(z\), \texttt{ulong} \(x\), \texttt{long} \(prec\))

Sets \(z\) to the reciprocal square root of \(x\), rounded to \(prec\) bits. At high precision, this is faster than computing a square root.

void \texttt{fmprb\_root} (\texttt{fmprb\_t} \(z\), \texttt{const fmprb\_t} \(x\), \texttt{ulong} \(k\), \texttt{long} \(prec\))

Sets \(z\) to the \(k\)-th root of \(x\), rounded to \(prec\) bits. As currently implemented, this function is only fast for small fixed \(k\). For large \(k\) it is better to use \texttt{fmprb\_pow\_fmpq()} or \texttt{fmprb\_pow()}.

void \texttt{fmprb\_aqm} (\texttt{fmprb\_t} \(z\), \texttt{const fmprb\_t} \(x\), \texttt{const fmprb\_t} \(y\), \texttt{long} \(prec\))

Sets \(z\) to the arithmetic-geometric mean of \(x\) and \(y\).
6.1 Credits and references

Arb is licensed GNU General Public License version 2, or any later version.

Arb includes code by Bill Hart and Sebastian Pancratz taken from FLINT (also licensed GPL 2.0+).

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 the present, 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 During that period, he was a postdoc at INRIA-Bordeaux and IMB, supervised by Andreas Enge.

6.1.1 Contributors

The following people (among others) have contributed patches or bug reports.

• Jonathan Bober
• Yuri Matiyasevich
• Abhinav Baid
• Ondřej Čertík
• Andrew Booker
• Francesco Biscani
• Clemens Heuberger
• Pascal Molin
• Ricky Farr

6.1.2 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

6.1.3 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:

@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}
}

6.1.4 Bibliography


Symbols

acb_mat_charpoly (C function), 93
acb_poly_add (C function), 80
acb_poly_agm1_series (C function), 89
acb_poly_atan_series (C function), 85
acb_poly_compose (C function), 82
acb_poly_compose_divconquer (C function), 81
acb_poly_compose_horner (C function), 81
acb_poly_compose_series (C function), 82
acb_poly_compose_series_brent_kung (C function), 82
acb_poly_compose_series_horner (C function), 82
acb_poly_cos_series (C function), 86
acb_poly_derivative (C function), 84
acb_poly_div (C function), 81
acb_poly_div_root (C function), 81
acb_poly_div_series (C function), 81
acb_poly_divrem (C function), 81
acb_poly_elliptic_k_series (C function), 89
acb_poly_elliptic_p_series (C function), 89
acb_poly_erf_series (C function), 89
acb_poly_evaluate (C function), 83
acb_poly_evaluate2 (C function), 83
acb_poly_evaluate2_horner (C function), 83
acb_poly_evaluate2_rectangular (C function), 83
acb_poly_evaluate_horner (C function), 82
acb_poly_evaluate_rectangular (C function), 82
acb_poly_evaluate_vec_fast (C function), 83
acb_poly_evaluate_vec_fast_precmp (C function), 83
acb_poly_evaluate_vec_iter (C function), 83
acb_poly_exp_series (C function), 86
acb_poly_exp_series_basecase (C function), 86
acb_poly_find_roots (C function), 90
acb_poly_gamma_series (C function), 87
acb_poly_integral (C function), 84
acb_poly_interpolate_barycentric (C function), 84
acb_poly_interpolate_fast (C function), 84
acb_poly_interpolate_fast_precmp (C function), 84
acb_poly_interpolate_newton (C function), 84
acb_poly_interpolation_weights (C function), 84
acb_poly_inv_series (C function), 81
acb_poly_lgamma_series (C function), 89
acb_poly_log_series (C function), 85
acb_poly_mul (C function), 81
acb_poly_mullow (C function), 80
acb_poly_mullow_classical (C function), 80
acb_poly_mullow_transpose (C function), 80
acb_poly_mullow_transpose_gauss (C function), 80
acb_poly_normalise (C function), 78
acb_poly_overlaps (C function), 79
acb_poly_polylog_cpx (C function), 89
acb_poly_polylog_cpx_small (C function), 88
acb_poly_polylog_cpx_zeta (C function), 88
acb_poly_polylog_series (C function), 89
acb_poly_pow_acb_series (C function), 85
acb_poly_pow_series (C function), 85
acb_poly_pow_ui (C function), 84
acb_poly_pow_ui_trunc_binexp (C function), 84
acb_poly_powsum_one_series_sieved (C function), 87
acb_poly_powsum_series_naive (C function), 87
acb_poly_powsum_series_naive_threaded (C function), 87
acb_poly_product_roots (C function), 83
acb_poly_refine_roots_durand_kerner (C function), 90
acb_poly_rem (C function), 81
acb_poly_revert_series (C function), 82
acb_poly_revert_series_lagrange (C function), 82
acb_poly_revert_series_lagrange_fast (C function), 82
acb_poly_revert_series_newton (C function), 82
acb_poly_rgamma_series (C function), 89
acb_poly_rising_ui_series (C function), 87
acb_poly_root_inclusion (C function), 89
acb_poly_rqrrt_series (C function), 85
acb_poly_set_length (C function), 78
acb_poly_shift_left (C function), 79
acb_poly_shift_right (C function), 79
acb_poly_sin_cos_series (C function), 86
acb_poly_sin_cos_series_basecase (C function), 86
acb_poly_sin_cos_series_tangent (C function), 86
acb_poly_sin_series (C function), 86
acb_poly_sqrt_series (C function), 85
acb_poly_sub (C function), 80
acb_poly_tan_series (C function), 86
acb_poly_tree_alloc (C function), 83
acb_poly_tree_build (C function), 83
Index

arb_mat_entry (C macro), 90
arb_mat_equal (C function), 91
arb_mat_exp (C function), 94
arb_mat_init (C function), 91
arb_mat_inv (C function), 93
arb_mat_is_real (C function), 91
arb_mat_lu (C function), 93
arb_mat_mul (C function), 92
arb_mat_ncols (C macro), 91
arb_mat_neg (C function), 92
arb_mat_nrows (C macro), 90
arb_mat_one (C function), 92
arb_mat_overlaps (C function), 91
arb_mat_pow_ui (C function), 92
arb_mat_printd (C function), 91
arb_mat_randtest (C function), 91
arb_mat_scalar_addmul_acb (C function), 92
arb_mat_scalar_addmul_arb (C function), 92
arb_mat_scalar_addmul_fmpz (C function), 92
arb_mat_scalar_addmul_si (C function), 92
arb_mat_scalar_div_acb (C function), 92
arb_mat_scalar_div_arb (C function), 92
arb_mat_scalar_div_fmpz (C function), 92
arb_mat_scalar_div_si (C function), 92
arb_mat_scalar_mul_2exp_si (C function), 92
arb_mat_scalar_mul_acb (C function), 92
arb_mat_scalar_mul_arb (C function), 92
arb_mat_scalar_mul_fmpz (C function), 92
arb_mat_scalar_mul_si (C function), 92
arb_mat_set (C function), 91
arb_mat_set_fmpq_mat (C function), 91
arb_mat_set_fmpz_mat (C function), 91
arb_mat_set_lu_precomp (C function), 93
arb_mat_solve (C function), 93
arb_mat_sub (C function), 92
arb_mat_t (C type), 90
arb_mat_zero (C function), 92
arb_modular_addseq_eta (C function), 103
arb_modular_addseq_theta (C function), 102
arb_modular_delta (C function), 104
arb_modular_eisenstein (C function), 104
arb_modular_elliptic_k (C function), 105
arb_modular_elliptic_k_cpx (C function), 105
arb_modular_elliptic_p (C function), 105
arb_modular_elliptic_p_cpx (C function), 105
arb_modular_epsilon_arg (C function), 103
arb_modular_eta (C function), 104
arb_modular_eta_sum (C function), 103
arb_modular_transform (C function), 100
arb_mul (C function), 74
arb_mul_2exp_fmpz (C function), 74
arb_mul_2exp_si (C function), 74
arb_mul_arb (C function), 74
arb_mul_fmpz (C function), 74
arb_mul_onei (C function), 74
arb_one (C function), 71
arb_onei (C function), 71
arb_overlaps (C function), 72
arb_poly_add (C function), 80
arb_poly_agm1_series (C function), 89
arb_poly_atan_series (C function), 85
arb_poly_clear (C function), 78
arb_poly_compose (C function), 82
arb_poly_compose_divconquer (C function), 82
arb_poly_compose_horner (C function), 81
arb_poly_compose_series (C function), 82
arb_poly_compose_series_brent_kung (C function), 82
arb_poly_compose_series_horner (C function), 82
arb_poly_contains (C function), 79
arb_poly_contains_fmpq_poly (C function), 79
arb_poly_contains_fmpz_poly (C function), 79
arb_poly_cos_series (C function), 86
arb_poly_degree (C function), 78
arb_poly_derivative (C function), 84
arb_poly_div_series (C function), 81
arb_poly_divrem (C function), 81
arb_poly_elliptic_k_series (C function), 89
arb_poly_elliptic_p_series (C function), 89
arb_poly_equal (C function), 79
arb_poly_erf_series (C function), 89
arb_poly_evaluate (C function), 83
arb_poly_evaluate2 (C function), 83
arb_poly_evaluate2_horner (C function), 83
arb_poly_evaluate2_rectangular (C function), 83
arb_poly_evaluate_horner (C function), 82
arb_poly_evaluate_rectangular (C function), 83
arb_poly_evaluate_vec_fast (C function), 83
arb_poly_evaluate_vec_iter (C function), 83
arb_poly_exp_series (C function), 86
arb_fmpz_div_fmpz (C function), 43
arb_gamma (C function), 47
arb_gamma_fmpz (C function), 47
arb_get_abs_lbound_arf (C function), 39
arb_get_abs_ubound_arf (C function), 39
arb_get_fmprb (C function), 37
arb_get_fmpz_mid_rad_10exp (C function), 40
arb_get_interval_arf (C function), 39
arb_get_interval_fmpz_2exp (C function), 39
arb_get_interval_mpf (C function), 40
arb_get_mag (C function), 39
arb_get_mag_lower (C function), 39
arb_get_mag_lower_nonnegative (C function), 39
arb_get_rand_fmpq (C function), 39
arb_get_str (C function), 37
arb_get_unique_fmpz (C function), 40
arb_hurwitz_zeta (C function), 48
arb_imagref (C macro), 71
arb_indeterminate (C function), 38
arb_init (C function), 37
arb_inv (C function), 42
arb_is_exact (C function), 40
arb_is_finite (C function), 40
arb_is_negative (C function), 41
arb_is_nonnegative (C function), 41
arb_is_nonpositive (C function), 41
arb_is_nonzero (C function), 40
arb_is_one (C function), 40
arb_is_positive (C function), 41
arb_is_zero (C function), 40
arb_lgamma (C function), 47
arb_log (C function), 44
arb_log1p (C function), 44
arb_log_arf (C function), 44
arb_log_fmpz (C function), 44
arb_log_ui (C function), 44
arb_log_ui_from_prev (C function), 44
arb_mat_add (C function), 65
arb_mat_bound_inf_norm (C function), 65
arb_mat_charpoly (C function), 67
arb_mat_clear (C function), 64
arb_mat_det (C function), 67
arb_mat_entry (C macro), 64
arb_mat_equal (C function), 65
arb_mat_exp (C function), 67
arb_mat_init (C function), 64
arb_mat_inv (C function), 67
arb_mat_mul (C function), 66
arb_mat_mul_classical (C function), 66
arb_mat_mul_threaded (C function), 66
arb_mat_ncols (C macro), 64
arb_mat_neg (C function), 65
arb_mat_nrows (C macro), 64
arb_mat_one (C function), 65
arb_mat_overlaps (C function), 65
arb_mat_pow_ui (C function), 66
arb_mat_printd (C function), 65
arb_mat_randtest (C function), 65
arb_mat_scalar_addmul_arb (C function), 66
arb_mat_scalar_addmul_fmpz (C function), 66
arb_mat_scalar_addmul_si (C function), 66
arb_mat_scalar_div_arb (C function), 66
arb_mat_scalar_div_fmpz (C function), 66
arb_mat_scalar_div_si (C function), 66
arb_mat_scalar_mul_2exp_si (C function), 66
arb_mat_scalar_mul_arb (C function), 66
arb_mat_scalar_mul_fmpz (C function), 66
arb_mat_scalar_mul_fmpz (C function), 66
arb_mat_scalar_mul_si (C function), 66
arb_mat_scalar_mul_si (C function), 66
arb_mat_scalar.mul (C function), 66
arb_mat_set (C function), 64
arb_mat_set_fmpq_mat (C function), 64
arb_mat_set_fmpz_mat (C function), 64
arb_mat_solve (C function), 67
arb_mat_solve_lu_precomp (C function), 66
arb_mat_struct (C type), 64
arb_mat_sub (C function), 65
arb_mat_t (C type), 64
arb_mat_zero (C function), 65
arb_neg (C function), 41
arb_neg_round (C function), 41
arb_one (C function), 38
arb_overlaps (C function), 41
arb_poly_acos_series (C function), 60
arb_poly_add (C function), 53
arb_poly_asin_series (C function), 60
arb_poly_atan_series (C function), 59
arb_poly_binomial_transform (C function), 59
arb_poly_binomial_transform_basecase (C function), 59
arb_poly_binomial_transform_convolution (C function), 59
arb_poly_borel_transform (C function), 58
arb_poly_log_series (C function), 60
arb_poly_clear (C function), 52
arb_poly_compose (C function), 55
arb_poly_compose_divconquer (C function), 55
arb_poly_compose_horner (C function), 55
arb_poly_compose_series (C function), 56
arb_poly_compose_series_brent_kung (C function), 56
arb_poly_compose_series_horner (C function), 55
arb_poly_contains (C function), 53
arb_poly_contains_fmpz_poly (C function), 53
arb_poly_contains_fmpq_poly (C function), 53
arb_poly_cos_series (C function), 62
arb_poly_degree (C function), 52
arb_poly_derivative (C function), 58
arb_poly_divrem (C function), 55
arb_poly_equal (C function), 53
arb_poly_evaluate (C function), 56
arb_poly_evaluate2 (C function), 57
arb_poly_evaluate2_acb (C function), 57
arb_poly_evaluate2_acb_horner (C function), 57
arb_poly_evaluate2_acb_rectangular (C function), 57
arb_poly_evaluate2_horner (C function), 57
arb_poly_evaluate2_rectangular (C function), 57
arb_poly_evaluate_acb (C function), 56
arb_poly_evaluate_acb_horner (C function), 56
arb_poly_evaluate_acb_rectangular (C function), 56
arb_poly_evaluate_horner (C function), 56
arb_poly_evaluate_rectangular (C function), 56
arb_poly_evaluate_vec_fast (C function), 58
arb_poly_evaluate_vec_iter (C function), 57
arb_poly_exp_series (C function), 61
arb_poly_exp_series_basecase (C function), 61
arb_poly_exp_series_iter (C function), 61
arb_poly_fit_length (C function), 52
arb_poly_gamma_series (C function), 62
arb_poly_get_coeff_arb (C function), 52
arb_poly_get_coeff_ptr (C macro), 52
arb_poly_get_unique_fmpz_poly (C function), 53
arb_poly_init (C function), 52
arb_poly_integral (C function), 58
arb_poly_interpolate_barycentric (C function), 58
arb_poly_interpolate_fast (C function), 58
arb_poly_interpolate_newton (C function), 58
arb_poly_inv_borel_transform (C function), 59
arb_poly_inv_series (C function), 55
arb_poly_length (C function), 52
arb_poly_lgamma_series (C function), 62
arb_poly_log_series (C function), 60
arb_poly_mullow (C function), 54
arb_poly_mullow_block (C function), 54
arb_poly_mullow_classical (C function), 54
arb_poly_mullow_ztrunc (C function), 54
arb_poly_neg (C function), 54
arb_poly_one (C function), 52
arb_poly_overlaps (C function), 53
arb_poly_pow_arb_series (C function), 60
arb_poly_pow_series (C function), 60
arb_poly_pow_ui (C function), 59
arb_poly_pow_ui_trunc_binexp (C function), 59
arb_poly_printd (C function), 53
arb_poly_product_roots (C function), 57
arb_poly_randtest (C function), 53
arb_poly_revert_series (C function), 56
arb_poly_revert_series_lagrange (C function), 56
arb_poly_revert_series_lagrange_fast (C function), 56
arb_poly_revert_series_newton (C function), 56
arb_poly_rgamma_series (C function), 62
arb_poly_riemann_siegel_theta_series (C function), 63
arb_poly_riemann_siegel_z_series (C function), 63
arb_poly_rising_ui_series (C function), 62
arb_poly_rsqrt_series (C function), 60
arb_poly_scalar_mul_2exp_si (C function), 54
arb_poly_set (C function), 52
arb_poly_set_coeff_arb (C function), 52
arb_poly_set_coeff_si (C function), 52
arb_poly_set_fmpz_poly (C function), 53
arb_poly_set_fmpq_poly (C function), 53
arb_poly_set_round (C function), 52
arb_poly_set_si (C function), 53
arb_poly_shift_left (C function), 52
arb_poly_shift_right (C function), 52
arb_poly_sin_cos_series (C function), 62
arb_poly_sin_cos_series_basecase (C function), 61
arb_poly_sin_cos_series_tangent (C function), 61
arb_poly_sin_series (C function), 61
arb_poly_sqrt_series (C function), 60
arb_poly_struct (C type), 51
arb_poly_sub (C function), 54
arb_poly_swinnerton_dyer_ui (C function), 64
arb_poly_t (C type), 51
arb_poly_tan_series (C function), 62
arb_poly_truncate (C function), 52
arb_poly_zero (C function), 52
arb_poly_zeta_series (C function), 62
arb_polylog (C function), 49
arb_polylog_si (C function), 49
arb_pos_inf (C function), 38
arb_pow (C function), 44
arb_pow_fmpz (C function), 43
arb_pow_fmpz_binexp (C function), 43
arb_pow_fmpq (C function), 44
arb_pow_fmpz (C function), 43
arb_ptr (C type), 36
arb_radref (C macro), 36
arb_randtest (C function), 38
arb_print (C function), 38
arb_printd (C function), 38
arb_printn (C function), 38
arb_ptr (C type), 36
arb_radref (C macro), 36
arb_randtest (C function), 38

140 Index
arb_randtest_exact (C function), 38
arb_randtest_precise (C function), 38
arb_randtest_special (C function), 38
arb_randtest_wide (C function), 38
arb_rel_accuracy_bits (C function), 40
arb_rel_error_bits (C function), 40
arb_rgamma (C function), 47
arb_rising2_ui (C function), 47
arb_rising2_ui_bs (C function), 47
arb_rising2_ui_rs (C function), 47
arb_rising_fmpq_ui (C function), 47
arb_rising_ui (C function), 47
arb_rising_ui_bs (C function), 47
arb_rising_ui_rec (C function), 47
arb_rising_ui_rs (C function), 47
arb_root (C function), 43
arb_rsqrt (C function), 43
arb_rsqrt_ui (C function), 43
arb_set (C function), 37
arb_set_arf (C function), 37
arb_set_fmpq (C function), 37
arb_set_fmprb (C function), 37
arb_set_fmpz (C function), 37
arb_set_fmpz_2exp (C function), 37
arb_set_interval_arf (C function), 39
arb_set_interval_mpfr (C function), 39
arb_set_round (C function), 37
arb_set_round_fmpz (C function), 37
arb_set_round_fmpz_2exp (C function), 37
arb_set_si (C function), 37
arb_set_str (C function), 37
arb_set_ui (C function), 37
arb_si_pow_ui (C function), 43
arb_sin (C function), 44
arb_sin_cos (C function), 44
arb_sin_cos_pi (C function), 45
arb_sin_pi (C function), 45
arb_sin_pi_fmpq (C function), 45
arb_sinh (C function), 45
arb_sinh_cosh (C function), 45
arb_sqrt (C function), 43
arb_sqrt1pm1 (C function), 43
arb_sqrt_arf (C function), 43
arb_sqrt_fmpz (C function), 43
arb_sqrt_ui (C function), 43
arb_sqrtpos (C function), 43
arb_submul (C function), 42
arb_submul_arf (C function), 42
arb_submul_fmpz (C function), 42
arb_submul_si (C function), 42
arb_submul_ui (C function), 42
arb_swap (C function), 37
arb_1 (C type), 36
arb_tan (C function), 44
arb_tan_pi (C function), 45
arb_tanh (C function), 46
arb_trim (C function), 40
arb_ui_div (C function), 43
arb_ui_pow_ui (C function), 43
arb_union (C function), 39
arb_zero (C function), 38
arf_abs (C function), 33
arf_abs_bound_le_2exp_fmpz (C function), 32
arf_abs_bound_lt_2exp_fmpz (C function), 32
arf_abs_bound_lt_2exp_si (C function), 32
arf_abs (C function), 33
arf_abs_bound_le_2exp_fmpz (C function), 32
arf_abs_bound_lt_2exp_fmpz (C function), 32
arf_abs_bound_lt_2exp_si (C function), 32
arf_add (C function), 33
arf_add_fmpz (C function), 34
arf_add_fmpz_2exp (C function), 34
arf_add_si (C function), 33
arf_add_ui (C function), 33
arf_addmul (C function), 34
arf_addmul_fmpz (C function), 34
arf_addmul_mpz (C function), 34
arf_addmul_si (C function), 34
arf_addmul_ui (C function), 34
arf_bits (C function), 31
arf ceil (C function), 31
arf clear (C function), 29
arf cmp (C function), 31
arf cmp_2exp_si (C function), 31
arf cmpabs (C function), 31
arf cmpabs_2exp_si (C function), 31
arf cmpabs_mag (C function), 31
arf cmpabs_ui (C function), 31
arf complex_mul (C function), 35
arf complex_mul fallback (C function), 35
arf complex sqr (C function), 35
arf debug (C function), 33
arf div (C function), 34

Index 141
arf_div_fmpz (C function), 35
arf_div_si (C function), 34
arf_div_ui (C function), 34
arf_equal (C function), 31
arf_equal_si (C function), 31
arf_floor (C function), 31
arf_fmpz_div (C function), 35
arf_fmpz_div_fmpz (C function), 35
arf_get_d (C function), 30
arf_get_fmpr (C function), 30
arf_get_fmpz (C function), 31
arf_get_fmpz_2exp (C function), 30
arf_get_fmpz_fixed_fmpz (C function), 31
arf_get_fmpz_fixed_si (C function), 31
arf_get_mag (C function), 32
arf_get_mag_lower (C function), 32
arf_get_mpfr (C function), 30
arf_get_si (C function), 31
arf_init (C function), 29
arf_init_neg_mag_shallow (C function), 32
arf_init_neg_shallow (C function), 32
arf_init_set_mag_shallow (C function), 32
arf_init_set_mag_shallow (C function), 32
arf_init_set_shallow (C function), 32
arf_init_set_t (C function), 32
arf_init_set_t (C function), 32
arf_interval_clear (C function), 68
arf_interval_get_arb (C function), 69
arf_interval_init (C function), 68
arf_interval_interval_t (C type), 68
arf_interval_printd (C function), 69
arf_interval_ptr (C type), 68
arf_interval_set (C function), 68
arf_interval_srcptr (C type), 68
arf_interval_struct (C type), 68
arf_interval_swap (C function), 68
arf_is_finite (C function), 29
arf_is_inf (C function), 29
arf_is_int (C function), 31
arf_is_int_2exp_si (C function), 32
arf_is_nan (C function), 29
arf_is_neg_inf (C function), 29
arf_is_normal (C function), 29
arf_is_one (C function), 29
arf_is_pos_inf (C function), 29
arf_is_special (C function), 29
arf_is_zero (C function), 29
arf_mag_add_ulp (C function), 32
arf_mag_fast_add_ulp (C function), 32
arf_mag_set_ulp (C function), 32
arf_max (C function), 31
arf_min (C function), 31
arf_mul (C function), 33
arf_mul_2exp_fmpz (C function), 33
arf_mul_2exp_si (C function), 33
arf_mul_fmpz (C function), 33
arf_mul_mpz (C function), 33
arf_mul_si (C function), 33
arf_mul_ui (C function), 33
arf_neg (C function), 29
arf_neg_inf (C function), 29
arf_neg_round (C function), 33
arf_one (C function), 29
arf_pos_inf (C function), 29
arf_rnd_t (C type), 28
arf_rndtest (C function), 33
arf_rndtest_not_zero (C function), 33
arf_rndtest_special (C function), 33
ARF_RND_CEIL (C macro), 28
ARF_RND_DOWN (C macro), 28
ARF_RND_FLOOR (C macro), 28
ARF_RND_NEAR (C macro), 29
arf_rsqrt (C function), 35
arf_set (C function), 30
arf_set_d (C function), 30
arf_set_fmpr (C function), 30
arf_set_fmpz (C function), 30
arf_set_fmpz_2exp (C function), 30
arf_set_mag (C function), 32
arf_set_mpfr (C function), 30
arf_set_mpz (C function), 30
arf_set_round (C function), 30
arf_set_round_fmpz (C function), 30
arf_set_round_fmpz_2exp (C function), 30
arf_set_round_mpz (C function), 30
arf_set_round_si (C function), 30
arf_set_round_ui (C function), 30
arf_set_t (C function), 30
arf_set_si_2exp_si (C function), 30
arf_set_ui_2exp_si (C function), 30
arf_sgn (C function), 31
arf_sqrt (C function), 35
arf_sqrt_fmpz (C function), 35
arf_sqrt_ui (C function), 35
arf_sub (C function), 34
arf_sub_fmpz (C function), 34
arf_sub_mpz (C function), 34
arf_submul (C function), 34
arf_submul_fmpz (C function), 34
arf_submul_mpz (C function), 34
arf_submul_si (C function), 34
arf_submul_ui (C function), 34
arf_sum (C function), 34
arf_swap (C function), 30
arf_t (C type), 28
arf_ui_div (C function), 34
arf_zero (C function), 29
bernoulli_bound_2exp_si (C function), 106
bernoulli_cache (C variable), 106
bernoulli_cache_compute (C function), 106
bernoulli_cache_num (C variable), 106
bernoulli_fmpq_ui (C function), 106
bernoulli_rev_clear (C function), 106
bernoulli_rev_init (C function), 106
bernoulli_rev_next (C function), 106
bernoulli_rev_t (C type), 105
fmpr_abs (C function), 120
fmpr_abs_bound_le_2exp_fmpz (C function), 119
fmpr_abs_bound_lt_2exp_fmpz (C function), 119
fmpr_abs_bound_lt_2exp_si (C function), 119
fmpr_add (C function), 120
fmpr_add_error_result (C function), 117
fmpr_add_fmpz (C function), 120
fmpr_add_si (C function), 120
fmpr_add_ui (C function), 120
fmpr_addmul (C function), 121
fmpr_addmul_fmpz (C function), 121
fmpr_addmul_si (C function), 121
fmpr_addmul_ui (C function), 121
fmpr_bits (C function), 119
fmpr_check_ulp (C function), 117
fmpr_clear (C function), 116
fmpr_cmp (C function), 119
fmpr_cmp_2exp_fmpz (C function), 119
fmpr_cmpabs (C function), 119
fmpr_cmpabs_2exp_si (C function), 119
fmpr_cmpabs_ui (C function), 119
fmpr_div (C function), 121
fmpr_div_fmpz (C function), 121
fmpr_div_fmpz_2exp (C function), 118
fmpr_div_fmpz_fixed_fmpz (C function), 118
fmpr_div_fmpz_fixed_si (C function), 118
fmpr_div.ui (C function), 121
fmpr_divappr_abs_ubound (C function), 121
fmpr_equal (C function), 119
fmpr_exp (C function), 122
fmpr_exp1 (C function), 122
fmpr_fmpz_div (C function), 121
fmpr_fmpz_div_fmpz (C function), 121
fmpr_get_d (C function), 118
fmpr_get_fmpz (C function), 118
fmpr_get_fmpz_2exp (C function), 118
fmpr_get_fmpz_fixed_fmpz (C function), 118
fmpr_get_fmpz_fixed_si (C function), 118
fmpr_init (C function), 116
fmpr_is_finite (C function), 117
fmpr_is_inf (C function), 116
fmpr_is_Int (C function), 119
fmpr_is_int_2exp_si (C function), 119
fmpr_is_nan (C function), 116
fmpr_is_neg_inf (C function), 116
fmpr_is_normal (C function), 116
fmpr_is_one (C function), 116
fmpr_is_pos_inf (C function), 116
fmpr_is_special (C function), 117
fmpr_is_zero (C function), 116
fmpr_log (C function), 122
fmpr_log1p (C function), 122
fmpr_max (C function), 119
fmpr_min (C function), 119
fmpr_mul (C function), 120
fmpr_mul_2exp_fmpz (C function), 120
fmpr_mul_2exp_si (C function), 120
fmpr_mul_fmpz (C function), 120
fmpr_mul_si (C function), 120
fmpr_mul_ui (C function), 120
fmpr_nan (C function), 116
fmpr_neg (C function), 120
fmpr_neg_error_result (C function), 117
fmpr_neg_fmpz (C function), 120
fmpr_neg_fmpz_2exp (C function), 118
fmpr_neg_fmpz_fixed_fmpz (C function), 118
fmpr_neg_fmpz_fixed_si (C function), 118
fmpr_neg_int (C function), 118
Arb Documentation, Release 2.5.0

fmpr_set_fmpz (C function), 118
fmpr_set_fmpz_2exp (C function), 118
fmpr_set_mpfr (C function), 118
fmpr_set_round (C function), 117
fmpr_set_round_fmpz (C function), 117
fmpr_set_round_fmpz_2exp (C function), 118
fmpr_set_round_ui_2exp_fmpz (C function), 117
fmpr_set_round_uiui_2exp_fmpz (C function), 117
fmpr_set_si (C function), 118
fmpr_set_si_2exp_si (C function), 118
fmpr_set_ui (C function), 118
fmpr_set_ui_2exp_si (C function), 118
fmpr_sgn (C function), 119
fmpr_sqrt (C function), 121
fmpr_sqrt_fmpz (C function), 121
fmpr_sqrt_ui (C function), 121
fmpr_struct (C type), 115
fmpr_sub (C function), 120
fmpr_sub_fmpz (C function), 120
fmpr_sub_si (C function), 120
fmpr_sub_ui (C function), 120
fmpr_submul (C function), 121
fmpr_submul_fmpz (C function), 121
fmpr_submul_si (C function), 121
fmpr_submul_ui (C function), 121
fmpr_sum (C function), 120
fmpr_swap (C function), 117
fmpr_t (C type), 115
fmpr_ui_div (C function), 121
fmpr_ulp (C function), 117
fmpr_zero (C function), 116
fmprb_abs (C function), 126
fmprb_add (C function), 126
fmprb_add_error (C function), 124
fmprb_add_error_2exp_fmpz (C function), 124
fmprb_add_error_2exp_si (C function), 124
fmprb_add_error_fmpz (C function), 124
fmprb_add_fmpz (C function), 124
fmprb_add_fmpr (C function), 124
fmprb_add_fmpq (C function), 124
fmprb_addmul (C function), 126
fmprb_addmul_fmpz (C function), 126
fmprb_addmul_si (C function), 126
fmprb_addmul_ui (C function), 126
fmprb_agm (C function), 128
fmprb_clear (C function), 125
fmprb_contains (C function), 126
fmprb_contains_fmpq (C function), 126
fmprb_contains_fmpr (C function), 126
fmprb_contains_fmpz (C function), 126
fmprb_contains_mpfr (C function), 126
fmprb_contains_negative (C function), 126
fmprb_contains_nonnegative (C function), 126
fmprb_contains_nonpositive (C function), 126
fmprb_contains_positive (C function), 126
fmprb_contains_si (C function), 126
fmprb_contains_zero (C function), 126
fmprb_div (C function), 127
fmprb_div_2expm1_ui (C function), 127
fmprb_div_mpfr (C function), 127
fmprb_div_fmpz (C function), 127
fmprb_div_si (C function), 127
fmprb_div_ui (C function), 127
fmprb_equal (C function), 125
fmprb_fmpz_div_fmpz (C function), 127
fmprb_fmpz_div_fmpq (C function), 127
fmprb_get_abs_lbound_fmpr (C function), 124
fmprb_get_abs_ubound_fmpr (C function), 124
fmprb_get_interval_fmpz_2exp (C function), 124
fmprb_get_rand_fmpq (C function), 124
fmprb_get_unique_fmpz (C function), 125
fmprb_hypgeom_infsym (C function), 109
fmprb_hypgeom_sum (C function), 109
fmprb_hypot (C function), 128
fmprb_indeterminate (C function), 123
fmprb_init (C function), 123
fmprb_inv (C function), 127
fmprb_is_exact (C function), 125
fmprb_is_finite (C function), 125
fmprb_is_int (C function), 125
fmprb_is_negative (C function), 126
fmprb_is_nonnegative (C function), 126
fmprb_is_nonpositive (C function), 126
fmprb_is_nonzero (C function), 125
fmprb_is_one (C function), 125
fmprb_is_positive (C function), 125
fmprb_is_zero (C function), 125
fmprb_midref (C macro), 122
fmprb_mul (C function), 127
fmprb_mul_2exp_fmpz (C function), 127
fmprb_mul_2exp_si (C function), 127
fmprb_mul_fmpz (C function), 127
fmprb_mul_si (C function), 127
fmprb_mul_ui (C function), 127
fmprb_neg (C function), 126
fmprb_neg_inf (C function), 123
fmprb_one (C function), 123
fmprb_overlaps (C function), 126
fmprb_pos_inf (C function), 123
fmprb_rand (C function), 125
fmprb_randtest (C function), 124
fmprb_randtest_exact (C function), 124
mag_set_d (C function), 25
mag_set_d_2exp_fmpz (C function), 25
mag_set_fmpr (C function), 25
mag_set_fmpz (C function), 25
mag_set_fmpz_2exp_fmpz (C function), 25
mag_set_fmpz_2exp_fmpz_lower (C function), 25
mag_set_fmpz_lower (C function), 25
mag_set_ui (C function), 25
mag_set_ui_2exp_si (C function), 25
mag_set_ui_lower (C function), 25
mag_sqrt (C function), 26
mag_struct (C type), 23
mag_sub_lower (C function), 26
mag_swap (C function), 23
mag_t (C type), 23
mag_zero (C function), 24

partitions_fmpz_fmpz (C function), 109
partitions_fmpz_ui (C function), 110
partitions_fmpz_ui_using_doubles (C function), 110
partitions_hrr_sum_arb (C function), 109
partitions_rademacher_bound (C function), 109
psl2z_clear (C function), 100
psl2z_equal (C function), 100
psl2z_init (C function), 99
psl2z_inv (C function), 100
psl2z_is_correct (C function), 100
psl2z_is_one (C function), 100
psl2z_mul (C function), 100
psl2z_one (C function), 100
psl2z_print (C function), 100
psl2z_randtest (C function), 100
psl2z_set (C function), 100
psl2z_struct (C type), 99
psl2z_swap (C function), 100
psl2z_t (C type), 99