# General information

1. **Feature overview** ........................................... 3
2. **Setup** ......................................................... 4
   2.1 Dependencies .................................................. 4
   2.2 Installation as part of FLINT ................................. 4
   2.3 Standalone installation ...................................... 4
   2.4 Running code .................................................. 4
3. **Potential issues** ............................................. 5
   3.1 Interface changes ............................................. 5
   3.2 Correctness ..................................................... 5
   3.3 Integer overflow .............................................. 5
   3.4 Thread safety and caches ..................................... 6
   3.5 Use of hardware floating-point arithmetic .................. 6
4. **History and changes** ......................................... 7
5. **Example programs** ........................................... 15
   5.1 pi.c ............................................................. 15
   5.2 hilbert_matrix.c ............................................. 15
   5.3 keiper_li.c .................................................... 15
   5.4 real_roots.c ................................................... 16
   5.5 poly_roots.c ................................................... 18

# Module documentation (Arb 2.x types)

1. **mag.h** – fixed-precision unsigned floating-point numbers for bounds .......................................................... 21
   1.1 Types, macros and constants .................................. 21
   1.2 Memory management ........................................... 21
   1.3 Special values .................................................. 22
   1.4 Comparisons ..................................................... 22
   1.5 Input and output ............................................... 22
   1.6 Random generation ............................................ 22
   1.7 Conversions ..................................................... 23
   1.8 Arithmetic ....................................................... 23
   1.9 Fast, unsafe arithmetic ....................................... 24
   1.10 Powers and logarithms ....................................... 24
   1.11 Special functions ............................................ 25
2. **arf.h** – arbitrary-precision floating-point numbers .......................................................... 25
   2.1 Types, macros and constants .................................. 25
   2.2 Memory management ........................................... 26
   2.3 Special values .................................................. 26
   2.4 Assignment, rounding and conversions ....................... 27
   2.5 Comparisons and bounds ..................................... 28
2.2.6 Magnitude functions ................................................. 29
2.2.7 Shallow assignment ............................................... 29
2.2.8 Random number generation .................................... 29
2.2.9 Input and output .................................................. 30
2.2.10 Addition and multiplication ................................... 30
2.2.11 Summation ....................................................... 31
2.2.12 Division .......................................................... 31
2.2.13 Square roots ..................................................... 32
2.2.14 Complex arithmetic ............................................ 32

2.3 arb.h – real numbers represented as floating-point balls
2.3.1 Types, macros and constants ..................................... 32
2.3.2 Memory management ............................................ 33
2.3.3 Assignment and rounding ....................................... 33
2.3.4 Assignment of special values ................................... 34
2.3.5 Input and output ................................................ 34
2.3.6 Random number generation .................................... 34
2.3.7 Radius and interval operations ................................. 35
2.3.8 Comparisons ..................................................... 36
2.3.9 Arithmetic ....................................................... 37
2.3.10 Powers and roots .............................................. 38
2.3.11 Exponentials and logarithms ................................... 39
2.3.12 Trigonometric functions ....................................... 40
2.3.13 Inverse trigonometric functions .............................. 41
2.3.14 Hyperbolic functions ......................................... 41
2.3.15 Constants ....................................................... 41
2.3.16 Gamma function and factorials ............................... 42
2.3.17 Zeta function .................................................. 43
2.3.18 Bernoulli numbers ............................................ 44
2.3.19 Polylogarithms .............................................. 44
2.3.20 Other special functions ....................................... 44
2.3.21 Internal helper functions ..................................... 45

2.4 arb_poly.h – polynomials over the real numbers
2.4.1 Types, macros and constants .................................... 46
2.4.2 Memory management ............................................ 46
2.4.3 Basic manipulation ............................................. 47
2.4.4 Conversions ..................................................... 47
2.4.5 Input and output ................................................ 48
2.4.6 Random generation ............................................ 48
2.4.7 Comparisons ..................................................... 48
2.4.8 Arithmetic ....................................................... 48
2.4.9 Composition ..................................................... 50
2.4.10 Evaluation ....................................................... 51
2.4.11 Product trees .................................................. 52
2.4.12 Multipoint evaluation ........................................ 52
2.4.13 Interpolation .................................................. 52
2.4.14 Differentiation ................................................ 53
2.4.15 Transforms ..................................................... 53
2.4.16 Powers and elementary functions ............................ 54
2.4.17 Gamma function and factorials ............................... 56
2.4.18 Zeta function .................................................. 57
2.4.19 Root-finding ................................................... 58

2.5 arb_mat.h – matrices over the real numbers
2.5.1 Types, macros and constants .................................... 58
2.5.2 Memory management .......................................... 59
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.3</td>
<td>Conversions</td>
<td>59</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Input and output</td>
<td>59</td>
</tr>
<tr>
<td>2.5.5</td>
<td>Comparisons</td>
<td>59</td>
</tr>
<tr>
<td>2.5.6</td>
<td>Special matrices</td>
<td>59</td>
</tr>
<tr>
<td>2.5.7</td>
<td>Norms</td>
<td>60</td>
</tr>
<tr>
<td>2.5.8</td>
<td>Arithmetic</td>
<td>60</td>
</tr>
<tr>
<td>2.5.9</td>
<td>Scalar arithmetic</td>
<td>60</td>
</tr>
<tr>
<td>2.5.10</td>
<td>Gaussian elimination and solving</td>
<td>61</td>
</tr>
<tr>
<td>2.5.11</td>
<td>Special functions</td>
<td>61</td>
</tr>
<tr>
<td>2.6</td>
<td>arb_calc.h – calculus with real-valued functions</td>
<td>61</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Types, macros and constants</td>
<td>62</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Debugging</td>
<td>62</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Subdivision-based root finding</td>
<td>62</td>
</tr>
<tr>
<td>2.6.4</td>
<td>Newton-based root finding</td>
<td>64</td>
</tr>
<tr>
<td>2.7</td>
<td>acb.h – complex numbers</td>
<td>64</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Types, macros and constants</td>
<td>65</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Memory management</td>
<td>65</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Basic manipulation</td>
<td>65</td>
</tr>
<tr>
<td>2.7.4</td>
<td>Input and output</td>
<td>66</td>
</tr>
<tr>
<td>2.7.5</td>
<td>Random number generation</td>
<td>66</td>
</tr>
<tr>
<td>2.7.6</td>
<td>Precision and comparisons</td>
<td>66</td>
</tr>
<tr>
<td>2.7.7</td>
<td>Complex parts</td>
<td>67</td>
</tr>
<tr>
<td>2.7.8</td>
<td>Arithmetic</td>
<td>67</td>
</tr>
<tr>
<td>2.7.9</td>
<td>Elementary functions</td>
<td>68</td>
</tr>
<tr>
<td>2.7.10</td>
<td>Rising factorials</td>
<td>70</td>
</tr>
<tr>
<td>2.7.11</td>
<td>Gamma function</td>
<td>70</td>
</tr>
<tr>
<td>2.7.12</td>
<td>Zeta function</td>
<td>70</td>
</tr>
<tr>
<td>2.7.13</td>
<td>Polylogarithms</td>
<td>71</td>
</tr>
<tr>
<td>2.8</td>
<td>acb_poly.h – polynomials over the complex numbers</td>
<td>71</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Types, macros and constants</td>
<td>71</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Memory management</td>
<td>71</td>
</tr>
<tr>
<td>2.8.3</td>
<td>Basic properties and manipulation</td>
<td>71</td>
</tr>
<tr>
<td>2.8.4</td>
<td>Input and output</td>
<td>72</td>
</tr>
<tr>
<td>2.8.5</td>
<td>Random generation</td>
<td>72</td>
</tr>
<tr>
<td>2.8.6</td>
<td>Comparisons</td>
<td>72</td>
</tr>
<tr>
<td>2.8.7</td>
<td>Conversions</td>
<td>73</td>
</tr>
<tr>
<td>2.8.8</td>
<td>Arithmetic</td>
<td>73</td>
</tr>
<tr>
<td>2.8.9</td>
<td>Composition</td>
<td>75</td>
</tr>
<tr>
<td>2.8.10</td>
<td>Evaluation</td>
<td>76</td>
</tr>
<tr>
<td>2.8.11</td>
<td>Product trees</td>
<td>76</td>
</tr>
<tr>
<td>2.8.12</td>
<td>Multipoint evaluation</td>
<td>76</td>
</tr>
<tr>
<td>2.8.13</td>
<td>Interpolation</td>
<td>77</td>
</tr>
<tr>
<td>2.8.14</td>
<td>Differentiation</td>
<td>77</td>
</tr>
<tr>
<td>2.8.15</td>
<td>Elementary functions</td>
<td>77</td>
</tr>
<tr>
<td>2.8.16</td>
<td>Gamma function</td>
<td>79</td>
</tr>
<tr>
<td>2.8.17</td>
<td>Power sums</td>
<td>80</td>
</tr>
<tr>
<td>2.8.18</td>
<td>Zeta function</td>
<td>80</td>
</tr>
<tr>
<td>2.8.19</td>
<td>Polylogarithms</td>
<td>81</td>
</tr>
<tr>
<td>2.8.20</td>
<td>Root-finding</td>
<td>82</td>
</tr>
<tr>
<td>2.9</td>
<td>acb_mat.h – matrices over the complex numbers</td>
<td>83</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Types, macros and constants</td>
<td>83</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Memory management</td>
<td>83</td>
</tr>
<tr>
<td>2.9.3</td>
<td>Conversions</td>
<td>83</td>
</tr>
<tr>
<td>2.9.4</td>
<td>Input and output</td>
<td>83</td>
</tr>
</tbody>
</table>
### 2.9.5 Comparisons ......................................................... 84
### 2.9.6 Special matrices .................................................... 84
### 2.9.7 Norms .............................................................. 84
### 2.9.8 Arithmetic .......................................................... 84
### 2.9.9 Scalar arithmetic ................................................... 84
### 2.9.10 Gaussian elimination and solving ............................... 85
### 2.9.11 Special functions .................................................. 86
#### 2.10 `acb_calc.h` – calculus with complex-valued functions ............. 86
   2.10.1 Types, macros and constants ................................... 86
   2.10.2 Bounds ............................................................ 86
   2.10.3 Integration ....................................................... 87
#### 2.11 `acb_hypgeom.h` – hypergeometric functions in the complex numbers ... 87
   2.11.1 Convergent series ............................................... 88
   2.11.2 Asymptotic series ................................................. 89
   2.11.3 The error function ............................................... 90
   2.11.4 Bessel functions ............................................... 90
#### 2.12 `acb_modular.h` – modular forms in the complex numbers ................ 91
   2.12.1 The modular group ............................................... 91
   2.12.2 Modular transformations ...................................... 92
   2.12.3 Jacobi theta functions ........................................ 93
   2.12.4 The Dedekind eta function ................................... 95
   2.12.5 Modular forms ................................................... 96
   2.12.6 Elliptic functions .............................................. 96
#### 2.13 `bermoulli.h` – support for Bernoulli numbers .......................... 97
   2.13.1 Generation of Bernoulli numbers ............................... 97
   2.13.2 Caching .......................................................... 97
   2.13.3 Bounding ........................................................ 97
#### 2.14 `hypgeom.h` – support for hypergeometric series ....................... 98
   2.14.1 Strategy for error bounding ................................... 98
   2.14.2 Types, macros and constants ................................ 99
   2.14.3 Memory management ........................................... 99
   2.14.4 Error bounding ................................................ 100
   2.14.5 Summation ...................................................... 100
#### 2.15 `partitions.h` – computation of the partition function ................. 100
### 3 Algorithms and proofs ............................................... 103
#### 3.1 Algorithms for mathematical constants .................................. 103
   3.1.1 Pi ................................................................. 103
   3.1.2 Logarithms of integers ......................................... 103
   3.1.3 Euler’s constant ................................................ 103
   3.1.4 Catalan’s constant ............................................. 104
   3.1.5 Khinchin’s constant .......................................... 104
   3.1.6 Glaisher’s constant .......................................... 104
   3.1.7 Apery’s constant .............................................. 104
#### 3.2 Algorithms for elementary functions ................................... 104
   3.2.1 Arctangents .................................................... 105
   3.2.2 Error propagation for arctangents ............................ 105
   3.2.3 Logarithms ...................................................... 105
   3.2.4 Error propagation for logarithms ................................ 106
#### 3.3 Algorithms for gamma functions ....................................... 106
   3.3.1 The Stirling series ............................................. 106
   3.3.2 Rational arguments ............................................ 107
#### 3.4 Algorithms for polylogarithms ......................................... 107
   3.4.1 Computation for small \( z \) .................................... 107
### 4 Module documentation (Arb 1.x types)

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td><strong>fmpr.h</strong> – arbitrary-precision floating-point numbers</td>
<td>111</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Types, macros and constants</td>
<td>111</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Memory management</td>
<td>112</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Special values</td>
<td>112</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Assignment, rounding and conversions</td>
<td>113</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Comparisons</td>
<td>115</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Random number generation</td>
<td>115</td>
</tr>
<tr>
<td>4.1.7</td>
<td>Input and output</td>
<td>116</td>
</tr>
<tr>
<td>4.1.8</td>
<td>Arithmetic</td>
<td>116</td>
</tr>
<tr>
<td>4.1.9</td>
<td>Special functions</td>
<td>118</td>
</tr>
<tr>
<td>4.2</td>
<td><strong>fmprb.h</strong> – real numbers represented as floating-point balls</td>
<td>118</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Types, macros and constants</td>
<td>119</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Memory management</td>
<td>119</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Assignment and rounding</td>
<td>119</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Assignment of special values</td>
<td>120</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Input and output</td>
<td>120</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Random number generation</td>
<td>120</td>
</tr>
<tr>
<td>4.2.7</td>
<td>Radius and interval operations</td>
<td>121</td>
</tr>
<tr>
<td>4.2.8</td>
<td>Comparisons</td>
<td>122</td>
</tr>
<tr>
<td>4.2.9</td>
<td>Arithmetic</td>
<td>123</td>
</tr>
<tr>
<td>4.2.10</td>
<td>Powers and roots</td>
<td>124</td>
</tr>
</tbody>
</table>

### 5 Credits and references

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Credits and references</td>
<td>127</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Contributors</td>
<td>127</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Software</td>
<td>127</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Citing Arb</td>
<td>128</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Bibliography</td>
<td>128</td>
</tr>
</tbody>
</table>

Bibliography | 129  

Index | 131  

---

v
Arb is a C library for arbitrary-precision floating-point ball arithmetic, developed by Fredrik Johansson (fredrik.johansson@gmail.com). It supports efficient high-precision computation with polynomials, power series, matrices and special functions over the real and complex numbers, with automatic, rigorous error control.

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

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

Arb 1.x used a different set of numerical base types (fmpr, fmprb and fmpcb). These types had a slightly simpler internal representation, but generally had worse performance. 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 fmpr and 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.
1.2 Setup

1.2.1 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

1.2.2 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:

```bash
./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.

1.2.3 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:

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

If GMP/MPFR, 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).

1.2.4 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_printd(x, 50); printf("\n");
```
printf("Computed with arb-%s\n", arb_version);
arb_clear(x);
}

Compile it with:
gcc -larb test.c

or (if Arb is built as part of FLINT):
gcc -lflint test.c

If the Arb/FLINT header and library files are not in a standard location (/usr/local on most systems), you may also have to pass options such as:
-I/path/to/arb -I/path/to/flint -L/path/to/flint -L/path/to/arb
to gcc. Finally, to run the program, make sure that the linker can find the FLINT (and Arb) libraries. If they are installed in a nonstandard location, you can for example add this path to the LD_LIBRARY_PATH environment variable.

The output of the example program should be something like the following:
3.1415926535897932384626433832795028841971693993751 +/- 4.2764e-50
Computed with arb-2.4.0

1.3 Potential issues

1.3.1 Interface changes

As this is an early version, note that any part of the interface is subject to change without warning! Most of the core interface should be stable at this point, but no guarantees are made.

1.3.2 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!

1.3.3 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 $\text{LONG\_MAX}$, 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 \texttt{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. \texttt{fmprb\_mul\_si()}). However, it still applies to conversions and operations where the result is requested exactly and sizes become an issue. For example, trying to convert the floating-point number \(2^{2^{100}}\) to an integer could result in anything from a silent wrong value to thrashing followed by a crash, and it is the user’s responsibility not to attempt such a thing.

### 1.3.4 Thread safety and caches

Arb should be fully threadsafe, provided that both MPFR and FLINT have been built in threadsafe mode. Please note that thread safety is not currently tested, and extra caution when developing multithreaded code is therefore recommended.

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

### 1.3.5 Use of hardware floating-point arithmetic

Arb uses hardware floating-point arithmetic (the \texttt{double} type in C) in two different ways.

Firstly, \texttt{double} arithmetic as well as transcendental \texttt{libm} functions (such as \texttt{exp}, \texttt{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, \texttt{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 \texttt{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.
1.4 History and changes

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

• 2014-11-15 - version 2.4.0
  – arithmetic and core functions
    • made evaluation of sin, cos and exp at medium precision faster using the sqrt trick
    • optimized arb_sinh and arb_sinh_cosh
    • optimized complex division with a small denominator
    • optimized cubing of complex numbers
    • added floor and ceil functions for the arf and arb types
    • added acb_poly powering functions
    • added acb_exp_pi_i
    • added functions for evaluation of Chebyshev polynomials
    • fixed arb_div to output nan for input containing nan
  – added a module acb_hypgeom for hypergeometric functions
    • evaluation of the generalized hypergeometric function in convergent cases
    • evaluation of confluent hypergeometric functions using asymptotic expansions
    • the Bessel function of the first kind for complex input
    • the error function for complex input
  – added a module acb_modular for modular forms and elliptic functions
    • support for working with modular transformations
    • mapping a point to the fundamental domain
    • evaluation of Jacobi theta functions and their series expansions
    • the Dedekind eta function
    • the j-invariant and the modular lambda and delta function
    • Eisenstein series
    • the Weierstrass elliptic function and its series expansion
  – miscellaneous
    • fixed mag_print printing a too large exponent
    • fixed printd methods to use a fallback instead of aborting when printing numbers too large for MPFR
    • added version number string (arb_version)
    • various additions to the documentation

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

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

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

• 2014-05-27 - version 2.0.0
– new modules mag, arf, arb, arb_poly, arb_mat, acb, acb_poly, acb_mat for higher-performance ball arith-
metric
– 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 fmpcb_calc module for calculus with complex functions
    * numerical integration using Taylor series
  - scalar functions
    * simplified fmpcb_const_euler using published error bound
    * added fmpcb_inv
    * fmpcb_trim, fmpcb_trim
    * added fmpcb_rsqrt (complex reciprocal square root)
    * fixed bug in fmpcb_sqrtpos with nonfinite input
    * slightly improved fmpcb powering code
    * added various functions for bounding fmprs by powers of two
    * added fmp_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 fmpcb_poly_pow_series, fmpcb_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 mpfr_rec_sqrt
• faster high-precision division by not computing an explicit remainder
• slightly faster computation of pi by using new reciprocal square root and division code
• added an fmpr function for approximate division to speed up certain radius operations
• added fmpr_set_d for conversion from double
• allow use of doubles to optionally compute the partition function faster but without an error bound
• bypass mpfr overflow when computing the exponential function to extremely high precision (approx-
  imately 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 comput-
  ation
• added fmprb_asin, fmprb_acos
• added various other utility functions to the fmprb module
• added a function for computing the Glaisher constant
• optimized evaluation of the Riemann zeta function at high precision
– polynomials and power series
• made squaring of polynomials faster than generic multiplication
* implemented power series reversion (various algorithms) for the fmprb_poly type
* added many fmprb_poly utility functions (shifting, truncating, setting/getting coefficients, etc.)
* improved power series division when either operand is short
* improved power series logarithm when the input is short
* improved power series exponential to use the basecase algorithm for short input regardless of the output size
* added power series square root and reciprocal square root
* added atan, tan, sin, cos, sin_cos, asin, acos fmprb_poly power series functions
* added Newton iteration macros to simplify various functions
* added gamma functions of real and complex power series ([fmprb/fmpcb]_[gamma/rgamma/lgamma]_series)
* added wrappers for computing the Hurwitz zeta function of a power series ([fmprb/fmpcb]_poly_zeta_series)
* implemented sieving and other optimizations to improve performance for evaluating the zeta function of a short power series
* improved power series composition when the inner series is linear
* added many fmpcb_poly versions of nearly all fmprb_poly functions
* improved speed and stability of series composition/reversion by balancing the power table exponents

  – other
  * added support for freeing all cached data by calling flint_cleanup()
  * introduced fmprb_ptr, fmpcb_ptr typedefs for cleaner function signatures
  * various bug fixes and general cleanup

• 2013-05-31 - version 0.6
  – made fast polynomial multiplication over the reals numerically stable by using a blockwise algorithm
  – disabled default use of the Gauss formula for multiplication of complex polynomials, to improve numerical stability
  – added division and remainder for complex polynomials
  – added fast multipoint evaluation and interpolation for complex polynomials
  – added missing fmprb_poly_sub and fmpcb_poly_sub functions
  – faster exponentials (fmprb_exp and dependent functions) at low precision, using precomputation
  – rewrote fmpr_add and fmpr_sub using mpn level code, improving efficiency at low precision
  – ported the partition function implementation from flint (using ball arithmetic in all steps of the calculation to guarantee correctness)
  – ported algorithm for computing the cosine minimal polynomial from flint (using ball arithmetic to guarantee correctness)
  – support using gmp instead of mpir
  – only use thread-local storage when enabled in flint
  – slightly faster error bounding for the zeta function

1.4. History and changes
• 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{fmprb\_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, cos\_pi, 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, evaluate2\_fmpcb} for simultaneously evaluating the derivative
    * added \texttt{fmprb\_poly} root polishing code using near-optimal Newton steps (experimental)
    * added \texttt{fmpr\_root, fmprb\_root} (currently based on MPFR)
    * added \texttt{fmpr\_min, fmpr\_max}
    * added \texttt{fmprb\_set\_interval\_fmpr, fmprb\_union}
    * added \texttt{fmpr\_bits, fmprb\_bits, fmpcb\_bits} for obtaining the mantissa width
    * added \texttt{fmprb\_hypot}
    * added complex square roots
    * improved \texttt{fmprb\_log} to slightly improve speed, and properly support huge arguments
    * fixed \texttt{exp, cosh, sinh} to work with huge arguments
    * added \texttt{fmprb\_expm1}
    * fixed \texttt{sin, cos, atan} to work with huge arguments
    * improved \texttt{fmprb\_pow} and \texttt{fmpcb\_pow}, including automatic detection of small integer and half-integer exponents
    * added many more elementary functions: \texttt{fmprb\_tan/tan/cot, fmpcb\_tan/cot}, and pi versions
    * added \texttt{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 \texttt{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 fnpr_mul, fmprb_mul and set_round, resulting in general speed improvements
  – code for computing the complex Hurwitz zeta function with derivatives
  – fixed and documented error bounds for hypergeometric series
  – better algorithm for series evaluation of the gamma function at a rational point
  – much faster generation of Bernoulli numbers
  – complex log, exp, pow, trigonometric functions (currently based on MPFR)
  – complex nth roots via Newton iteration
  – added code for arithmetic on fmpcb_polys
  – code for computing Khinchin’s constant
  – code for rising factorials of polynomials or power series
  – faster sin_cos
  – better div_2expm1
  – many other new helper functions
  – improved thread safety
  – more test code for core operations
• 2012-11-07 - version 0.3
  – converted documentation to sphinx
– new module fmpcb for ball interval arithmetic over the complex numbers
  * conversions, utility functions and arithmetic operations
– new module fmpcb_mat for matrices over the complex numbers
  * conversions, utility functions and arithmetic operations
  * multiplication, LU decomposition, solving, inverse and determinant
– new module fmpcb_poly for polynomials over the complex numbers
  * root isolation for complex polynomials
– new module fmpz_holonomic for functions/sequences defined by linear differential/difference equations with polynomial coefficients
  * functions for creating various special sequences and functions
  * some closure properties for sequences
  * Taylor series expansion for differential equations
  * computing the nth entry of a sequence using binary splitting
  * computing the nth entry mod p using fast multipoint evaluation
– generic binary splitting code with automatic error bounding is now used for evaluating hypergeometric series
– matrix powering
– various other helper functions

• 2012-09-29 - version 0.2
  – code for computing the gamma function (Karatsuba, Stirling’s series)
  – rising factorials
  – fast exp_series using Newton iteration
  – improved multiplication of small polynomials by using classical multiplication
  – implemented error propagation for square roots
  – polynomial division (Newton-based)
  – polynomial evaluation (Horner) and composition (divide-and-conquer)
  – product trees, fast multipoint evaluation and interpolation (various algorithms)
  – power series composition (Horner, Brent-Kung)
  – added the fmprb_mat module for matrices of balls of real numbers
  – matrix multiplication
  – interval-aware LU decomposition, solving, inverse and determinant
  – many helper functions and small bugfixes

• 2012-09-14 - version 0.1

• 2012-08-05 - began simplified rewrite

• 2012-04-05 - experimental ball and polynomial code
1.5 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.

1.5.1 pi.c

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

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

```
> build/examples/pi 1000000
computing pi with a precision of 3321933 bits... cpu/wall(s): 0.58 0.586
virt/peak/res/peak(MB): 28.24 36.84 8.86 15.56
3.141592654 +/- 1.335e-1000001
```

The program prints a decimal approximation of the computed ball, with the midpoint rounded to a number of decimal digits that can be passed as a second parameter to the program (default = 10). In the present implementation (see `arb_printd()`), the digits are not guaranteed to be correctly rounded.

1.5.2 hilbert_matrix.c

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

Sample output:

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

1.5.3 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:
The program prints the first and last few coefficients. It can optionally write all the computed data to a file. The working precision defaults to a value that should give all the coefficients to a few digits of accuracy, but can optionally be set higher (or lower). On a multicore system, using several threads results in faster execution.

Sample output:

```bash
> build/examples/keiper_li 1000 -threads 2
 virt/peak/res/peak(MB): 170.18 294.69 7.51 7.51
```

1.5.4 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:
```
> build/examples/real_roots 0 0.0 50.0 -maxfound 1 -refine 75
interval: 25 +/- 25
maxdepth = 30, maxeval = 100000, maxfound = 1, low_prec = 30
refined root:
14.134725141734693790457251983562470270784257115699243175685567460149963429809 +/- 8.4532e-81
---------------------------------------------------------------
Found roots: 1
Subintervals possibly containing undetected roots: 8
Function evaluations: 992
cpu/wall(s): 0.41 0.415
virt/peak/res/peak(MB): 20.76 20.76 2.23 2.23

Find roots of $\sin(x^2)$ on $(0,100)$. The algorithm cannot isolate the root at $x = 0$ (it is at the endpoint of the interval, and in any case a root of multiplicity higher than one). The failure is reported:
```
> build/examples/real_roots 2 0 100
interval: 50 +/- 50
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
---------------------------------------------------------------
Found roots: 3183
Subintervals possibly containing undetected roots: 1
Function evaluations: 34058
cpu/wall(s): 0.26 0.263
virt/peak/res/peak(MB): 20.73 20.76 1.72 1.72

This does not miss any roots:

1.5. Example programs
> build/examples/real_roots 2 1 100
interval: 50.5 +/- 49.5
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
---------------------------------------------------------------
Found roots: 3183
Subintervals possibly containing undetected roots: 0
Function evaluations: 34039
cpu/wall(s): 0.26 0.266
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:

> build/examples/real_roots 3 0.0 1.0
interval: 0.5 +/- 0.5
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
---------------------------------------------------------------
Found roots: 10198
Subintervals possibly containing undetected roots: 24695
Function evaluations: 202587
cpu/wall(s): 1.73 1.731
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.

### 1.5.5 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:

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

If \texttt{-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 \texttt{<poly>}:  

- \texttt{a <n>}: Easy polynomial \(1 + 2x + \ldots + (n+1)x^n\)
- \texttt{t <n>}: Chebyshev polynomial \(T_n\)
- \texttt{u <n>}: Chebyshev polynomial \(U_n\)
- \texttt{p <n>}: Legendre polynomial \(P_n\)
- \texttt{c <n>}: Cyclotomic polynomial \(\Phi_n\)
- \texttt{s <n>}: Swinnerton-Dyer polynomial \(S_n\)
- \texttt{b <n>}: Bernoulli polynomial \(B_n\)
- \texttt{w <n>}: Wilkinson polynomial \(W_n\)
- \texttt{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) +/- (5.77e-118, 5.77e-118j)
(3 + 2.5811555581665e-118j) +/- (5.72e-115, 5.72e-115j)
(4 + 1.0214162852427e-115j) +/- (4.38e-112, 4.38e-112j)
(5 + 1.6132683409498e-113j) +/- (2.6e-109, 2.6e-109j)
...
(95 + 4.1529419687547e-62j) +/- (6.66e-59, 6.66e-59j)
(96 + 3.5450240192667e-64j) +/- (7.37e-60, 7.37e-60j)
(97 + 1.6775595325625e-65j) +/- (6.4e-61, 6.4e-61j)
(98 + 2.0463882235299e-65j) +/- (4e-62, 4e-62j)
(99 + 2.7342546802823e-66j) +/- (7.37e-60, 7.37e-60j)
(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)
(-0.249997571907742100e-129j) +/- (3.6e-127, 3.6e-127j)
(0.2499975715212762602 + 4.219100761823281708e-129j) +/- (4.98e-127, 4.98e-127j)
(0.75000242847487273998 + 9.036064991741317014e-128j) +/- (8.88e-126, 8.88e-126j)
(1.2499975711907742101 + 7.8804123808107088267e-127j) +/- (2.66e-124, 2.66e-124j)
(1.7553405925206798575 + 5.43265269253967768e-126j) +/- (6.23e-123, 6.23e-123j)
(1.94308706466055783383 + 3.303537734250953239e-125j) +/- (7.05e-123, 7.05e-123j)
(-0.9509334829256233279 + 0.44547958157103608805j) +/- (5.5e-125, 5.5e-125j)
(-0.9509334829256233279 + -0.44547958157103608805j) +/- (5.46e-125, 5.46e-125j)
(1.9950933482925623328 + 0.44547958157103608805j) +/- (1.44e-122, 1.44e-122j)
(1.9950933482925623328 + -0.44547958157103608805j) +/- (1.43e-122, 1.43e-122j)
(-0.92177327714429290564 + -1.0954360955079385542j) +/- (9.31e-125, 9.31e-125j)
(-0.92177327714429290564 + 1.0954360955079385542j) +/- (1.02e-124, 1.02e-124j)
(1.9217732771442929056 + 1.0954360955079385542j) +/- (9.15e-123, 9.15e-123j)
(1.9217732771442929056 + -1.0954360955079385542j) +/- (8.12e-123, 8.12e-123j)
cpu/wall(s): 0.02 0.02

1.5. Example programs
2.1 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 be a few ulps larger/smaller than the optimal value). The special values zero and positive infinity are supported (but not NaN). Applications requiring more flexibility (such as correct rounding, or higher precision) should use the `arf_t` type instead.

2.1.1 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.

2.1.2 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.
void _mag_vec_clear (mag_ptr v, long n)
   Clears a vector of length n.

2.1.3 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()).

2.1.4 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.

2.1.5 Input and output

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

2.1.6 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.
2.1.7 Conversions

void mag_set_d (mag_t y, double x)
void mag_set_fmpr (mag_t y, const fmpr_t x)
void mag_set_ui (mag_t y, ulong x)
void mag_set_fmpz (mag_t y, const fmpz_t x)
    Sets y to an upper bound for |x|.
void mag_set_d_2exp_fmpz (mag_t z, double x, const fmpz_t y)
void mag_set_fmpz_2exp_fmpz (mag_t z, const fmpz_t x, const fmpz_t y)
void mag_set_ui_2exp_si (mag_t z, ulong x, long y)
    Sets z to an upper bound for |x| × 2^y.
void mag_get_fmpr (fmpr_t y, const mag_t x)
    Sets y exactly to x.
void mag_get_fmpq (fmpq_t y, const mag_t x)
    Sets y exactly to x. Assumes that no overflow occurs.
void mag_set_ui_lower (mag_t z, ulong x)
void mag_set_fmpz_lower (mag_t z, const fmpz_t x)
    Sets y to a lower bound for |x|.
void mag_set_fmpz_2exp_fmpz_lower (mag_t z, const fmpz_t x, const fmpz_t y)
    Sets z to a lower bound for |x| × 2^y.

2.1.8 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 × 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 \texttt{mag_mul_ui_lower} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x}, ulong \_\texttt{y})
void \texttt{mag_mul_fmpz_lower} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x}, const \texttt{fmpz} \_\texttt{t} \_\texttt{y})

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

void \texttt{mag_add_lower} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x}, const \texttt{mag} \_\texttt{t} \_\texttt{y})

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

void \texttt{mag_sub_lower} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x}, const \texttt{mag} \_\texttt{t} \_\texttt{y})

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

### 2.1.9 Fast, unsafe arithmetic

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

void \texttt{mag_fast_init_set} (\texttt{mag} \_\texttt{t} \_\texttt{x}, const \texttt{mag} \_\texttt{t} \_\texttt{y})

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

void \texttt{mag_fast_zero} (\texttt{mag} \_\texttt{t} \_\texttt{x})

Sets \texttt{x} to zero.

int \texttt{mag_fast_is_zero} (const \texttt{mag} \_\texttt{t} \_\texttt{x})

Returns nonzero iff \texttt{x} is zero.

void \texttt{mag_fast_mul} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x}, const \texttt{mag} \_\texttt{t} \_\texttt{y})

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

void \texttt{mag_fast_addmul} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x}, const \texttt{mag} \_\texttt{t} \_\texttt{y})

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

void \texttt{mag_fast_add_2exp_si} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x}, long \_\texttt{e})

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

### 2.1.10 Powers and logarithms

void \texttt{mag_pow_ui} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x}, ulong \_\texttt{e})
void \texttt{mag_pow_fmpz} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x}, const \texttt{fmpz} \_\texttt{t} \_\texttt{e})

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

void \texttt{mag_pow_ui_lower} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x}, ulong \_\texttt{e})

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

void \texttt{mag_sqrt} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x})

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

void \texttt{mag_rsqrt} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x})

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

void \texttt{mag_log1p} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x})

Sets \texttt{z} to an upper bound for \( \log(1 + x) \). The bound is computed accurately for small \texttt{x}.

void \texttt{mag_exp} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x})

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

void \texttt{mag_expm1} (\texttt{mag} \_\texttt{t} \_\texttt{z}, const \texttt{mag} \_\texttt{t} \_\texttt{x})

Sets \texttt{z} to an upper bound for \( \exp(x) - 1 \). The bound is computed accurately for small \texttt{x}.
void \texttt{mag\_exp\_tail} (\texttt{mag\_t} z, const \texttt{mag\_t} x, ulong N)
Sets z to an upper bound for \( \sum_{k=N}^{\infty} x^k / k! \).

2.1.11 Special functions

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

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

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

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

The \texttt{arf\_t} type is essentially identical semantically to the \texttt{fmpr\_t} type, but uses an internal representation that generally allows operation to be performed more efficiently.

The most significant differences that the user has to be aware of are:

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

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

2.2.1 Types, macros and constants

\texttt{arf\_struct}

\texttt{arf\_t}

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

\texttt{arf\_rnd\_t}

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

\texttt{ARF\_RND\_DOWN}

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

\texttt{ARF\_RND\_UP}

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

\texttt{ARF\_RND\_FLOOR}

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

\texttt{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.

### 2.2.2 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.

### 2.2.3 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 **arf_is_finite** (arf_t x)

Returns nonzero iff x is a finite floating-point value, i.e. not one of the values +∞, −∞, NaN. (Note that this is not equivalent to the negation of **arf_is_inf**().)
### 2.2.4 Assignment, rounding and conversions

- `void arf_set (arf_t y, const arf_t x)`
  - Sets `y` exactly to `x`.
- `void arf_set_mpz (arf_t y, const mpz_t x)`
  - Swaps `y` and `x` efficiently.
- `void arf_set_fmpz (arf_t y, const fmpz_t x)`
  - Initialises `y` and sets it to `x` in a single operation.
- `int arf_set_round (arf_t y, const arf_t x, long prec, arf_rnd_t rnd)`
  - Sets `y` to `x`, rounded to `prec` bits in the direction specified by `rnd`.
- `int arf_set_round_si (arf_t y, const arf_t x, long prec, arf_rnd_t rnd)`
  - Sets `y` to `x`, rounded to `prec` bits in the direction specified by `rnd`.
- `int arf_set_round_ui (arf_t y, const arf_t x, long prec, arf_rnd_t rnd)`
  - Sets `y` to `x`, rounded to `prec` bits in the direction specified by `rnd`.
- `int arf_set_round_mpz (arf_t y, const mpz_t x, long prec, arf_rnd_t rnd)`
  - Sets `y` to `x`, rounded to `prec` bits in the direction specified by `rnd`.
- `int arf_set_round_fmpz (arf_t y, const fmpz_t x, long prec, arf_rnd_t rnd)`
  - Sets `y` to `x`, rounded to `prec` bits in the direction specified by `rnd`.
- `void arf_set_si_2exp_si (arf_t y, long m, long e)`
  - Sets `y` to `m × 2^e`.
- `void arf_set_ui_2exp_si (arf_t y, ulong m, long e)`
  - Sets `y` to `m × 2^e`.
- `void arf_set_fmpz_2exp (arf_t y, const fmpz_t m, const fmpz_t e)`
  - Sets `y` to `m × 2^e`, rounded to `prec` bits in the direction specified by `rnd`.
- `void arf_set_fmpz_2exp (mpfr_t y, const fmpz_t m, const fmpz_t e)`
  - Sets `y` to `m × 2^e`, rounded to `prec` bits in the direction specified by `rnd`.
- `double arf_get_d (const arf_t x, arf_rnd_t rnd)`
  - Returns `x` rounded to a double in the direction specified by `rnd`.
- `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`. Aborts if `x` is infinite, NaN or if the exponent is unreasonably large.
- `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.
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 × 2^e ≈ 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 ⌊x⌋ and ⌈x⌉ 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.

2.2.5 Comparisons and bounds

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

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

int arf_cmp_2exp_si (const arf_t x, long e)
int arf_cmpabs_2exp_si (const arf_t x, long e)
   Compares x (respectively its absolute value) with 2^e.

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

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

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| ≤ 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 -ARFPREC_EXACT and ARFPREC_EXACT inclusive. If \( x \) is zero, -ARFPREC_EXACT is returned, and if \( x \) is infinity or NaN, ARFPREC_EXACT is returned.

### 2.2.6 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.

### 2.2.7 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 \( x \).

### 2.2.8 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)
    Identical to arf_randtest(), except that the output occasionally is set to an infinity or NaN.

2.2.9 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.

2.2.10 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)
void arf_mul_2exp_fmpz (arf_t y, const arf_t x, const fmpz_t e)
    Sets y = x2^e exactly.

int arf_mul (arf_t z, const arf_t x, const arf_t y, 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)
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 prec bits in the direction specified by rnd, returning nonzero iff the operation is inexact.

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

int arf_sub (arf_t z, const arf_t x, const arf_t y, long prec, arf_rnd_t rnd)
2.2.11 Summation

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

2.2.12 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, ulong x, const arf_t 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)
int arf_si_div (arf_t z, long x, const arf_t y, long prec, arf_rnd_t rnd)
int arf_div_fmpz (arf_t z, const arf_t x, const fmpz_t y, long prec, arf_rnd_t rnd)
int arf_fmpz_div (arf_t z, const fmpz_t x, const arf_t y, long prec, arf_rnd_t rnd)
int arf_fmpz_div_fmpz (arf_t z, const fmpz_t x, const fmpz_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.
2.2.13 Square roots

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

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

```c
int arf_rsqrt (arf_t z, const arf_t x, long prec, arf_rnd_t rnd)
```

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

2.2.14 Complex arithmetic

```c
int arf_complex_mul (arf_t e, arf_t f, const arf_t a, const arf_t b, const arf_t c, const arf_t d, 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.

```c
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.

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

The `arf_t` type is essentially identical semantically to the `fmprb_t` type, but uses an internal representation that generally allows operation to be performed more efficiently.

Whereas the midpoint and radius of an `fmprb_t` both have the same type, the `arf_t` type uses an `arf_t` for the midpoint and a `mag_t` for the radius. Code designed to manipulate the radius of an `fmprb_t` directly can be ported to the `arf_t` type by writing the radius to a temporary `arf_t` variable, manipulating that variable, and then converting back to the `mag_t` radius. Alternatively, `mag_t` methods can be used directly where available.

2.3.1 Types, macros and constants

`arb_struct`

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

`arb_ptr`

Alias for `arb_struct *`, used for vectors of numbers.
**2.3.2 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.

**2.3.3 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.
2.3.4 Assignment of special values

void \texttt{arb\_zero(}arb\_t x) \hspace{1cm} \text{Sets } x \text{ to zero.}

void \texttt{arb\_one(}arb\_t f) \hspace{1cm} \text{Sets } x \text{ to the exact integer 1.}

void \texttt{arb\_pos\_inf(}arb\_t x) \hspace{1cm} \text{Sets } x \text{ to positive infinity, with a zero radius.}

void \texttt{arb\_neg\_inf(}arb\_t x) \hspace{1cm} \text{Sets } x \text{ to negative infinity, with a zero radius.}

void \texttt{arb\_zero\_pm\_inf(}arb\_t x) \hspace{1cm} \text{Sets } x \text{ to } [0 \pm \infty], \text{ representing the whole extended real line.}

void \texttt{arb\_indeterminate(}arb\_t x) \hspace{1cm} \text{Sets } x \text{ to } [\text{NaN} \pm \infty], \text{ representing an indeterminate result.}

2.3.5 Input and output

void \texttt{arb\_print(}const arb\_t x) \hspace{1cm} \text{Prints the internal representation of } x.

void \texttt{arb\_printd(}const arb\_t x, long digits) \hspace{1cm} \text{Prints } x \text{ 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.}

2.3.6 Random number generation

void \texttt{arb\_randtest(}arb\_t x, flint\_rand\_t state, long prec, long mag\_bits) \hspace{1cm} \text{Generates a random ball. The midpoint and radius will both be finite.}

void \texttt{arb\_randtest\_exact(}arb\_t x, flint\_rand\_t state, long prec, long mag\_bits) \hspace{1cm} \text{Generates a random number with zero radius.}

void \texttt{arb\_randtest\_precise(}arb\_t x, flint\_rand\_t state, long prec, long mag\_bits) \hspace{1cm} \text{Generates a random number with radius around } 2^{-\text{prec}} \text{ the magnitude of the midpoint.}

void \texttt{arb\_randtest\_wide(}arb\_t x, flint\_rand\_t state, long prec, long mag\_bits) \hspace{1cm} \text{Generates a random number with midpoint and radius chosen independently, possibly giving a very large interval.}

void \texttt{arb\_randtest\_special(}arb\_t x, flint\_rand\_t state, long prec, long mag\_bits) \hspace{1cm} \text{Generates a random interval, possibly having NaN or an infinity as the midpoint and possibly having an infinite radius.}

void \texttt{arb\_get\_rand\_fmpq(}fmpq\_t q, flint\_rand\_t state, const arb\_t x, long bits) \hspace{1cm} \text{Sets } q \text{ to a random rational number from the interval represented by } x. \text{ A denominator is chosen by multiplying the binary denominator of } x \text{ by a random integer up to } bits \text{ bits.}

\text{The outcome is undefined if the midpoint or radius of } x \text{ 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.}
### 2.3.7 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.

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

- **void `arb_get_interval_fmpz_2exp` (fmpz_t a, fmpz_t b, fmpz_t exp, const arb_t x)**
  Computes the exact interval represented by `x`, in the form of an integer interval multiplied by a power of two, i.e. $x = [a, b] \times 2^{\text{exp}}$.

  The 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.

- **long `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)**
- **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 \texttt{arb_trim} (arb_t \texttt{y}, const arb_t \texttt{x})

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

int \texttt{arb_get_unique_fmpz} (fmpz_t \texttt{z}, const arb_t \texttt{x})

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.

void \texttt{arb_floor} (arb_t \texttt{y}, const arb_t \texttt{x}, long \texttt{prec})

void \texttt{arb_ceil} (arb_t \texttt{y}, const arb_t \texttt{x}, long \texttt{prec})

Sets \texttt{y} to a ball containing \lfloor \texttt{x} \rfloor and \lceil \texttt{x} \rceil respectively, with the midpoint of \texttt{y} rounded to at most \texttt{prec} bits.

### 2.3.8 Comparisons

int \texttt{arb_is_zero} (const arb_t \texttt{x})

Returns nonzero iff the midpoint and radius of \texttt{x} are both zero.

int \texttt{arb_is_nonzero} (const arb_t \texttt{x})

Returns nonzero iff zero is not contained in the interval represented by \texttt{x}.

int \texttt{arb_is_one} (const arb_t \texttt{f})

Returns nonzero iff \texttt{x} is exactly 1.

int \texttt{arb_is_finite} (const arb_t \texttt{x})

Returns nonzero iff the midpoint and radius of \texttt{x} are both finite floating-point numbers, i.e. not infinities or NaN.

int \texttt{arb_is_exact} (const arb_t \texttt{x})

Returns nonzero iff the radius of \texttt{x} is zero.

int \texttt{arb_is_int} (const arb_t \texttt{x})

Returns nonzero iff \texttt{x} is an exact integer.

int \texttt{arb_equal} (const arb_t \texttt{x}, const arb_t \texttt{y})

Returns nonzero iff \texttt{x} and \texttt{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 \texttt{x} and \texttt{y} certainly represent the same real 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{arb_overlaps()} or \texttt{arb_contains()}, depending on the circumstance.

int \texttt{arb_is_positive} (const arb_t \texttt{x})

int \texttt{arb_is_nonnegative} (const arb_t \texttt{x})

int \texttt{arb_is_negative} (const arb_t \texttt{x})

int \texttt{arb_is_nonpositive} (const arb_t \texttt{x})

Returns nonzero iff all points \texttt{p} in the interval represented by \texttt{x} satisfy, respectively, \texttt{p} > 0, \texttt{p} ≥ 0, \texttt{p} < 0, \texttt{p} ≤ 0. If \texttt{x} contains NaN, returns zero.

int \texttt{arb_overlaps} (const arb_t \texttt{x}, const arb_t \texttt{y})

Returns nonzero iff \texttt{x} and \texttt{y} have some point in common. If either \texttt{x} or \texttt{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 arb_t \texttt{x}, const arf_t \texttt{y})

int \texttt{arb_contains_fmpq} (const arb_t \texttt{x}, const fmpq_t \texttt{y})

int \texttt{arb_contains_fmpz} (const arb_t \texttt{x}, const fmpz_t \texttt{y})

int \texttt{arb_contains_si} (const arb_t \texttt{x}, long \texttt{y})
int arb_contains_mpfr (const arb_t x, const mpfr_t y)

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

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

int arb_contains_zero (const arb_t x)

int arb_contains_negative (const arb_t x)

int arb_contains_nonpositive (const arb_t x)

int arb_contains_positive (const arb_t x)

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

2.3.9 Arithmetic

void arb_neg (arb_t y, const arb_t x)

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

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

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

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

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

void arb_add_si (arb_t z, const arb_t x, long y, long prec)

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

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

void arb_sub (arb_t z, const arb_t x, const arb_t y, 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 prec bits. The precision can be ARF_PREC_EXACT provided that the result fits in memory.

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

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

void arb_mul_ui (arb_t z, const arb_t x, ulong 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)
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 **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 **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 **ARF_PREC_EXACT** provided that the result fits in memory.

void **arb_inv** (arb_t y, const arb_t x, long prec)

Sets \( y \) to \( 1/x \).

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

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

void **arb_ui_div** (arb_t z, ulong x, const arb_t y, long prec)

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

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

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 **arb_div_2expm1_ui** (arb_t z, const arb_t x, ulong n, long prec)

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

### 2.3.10 Powers and roots

void **arb_sqrt** (arb_t z, const arb_t x, long prec)
2.3.11 Exponentials and logarithms

void *arb_sqrtarf (arb_t z, const arf_t x, long prec)
void *arb_sqrtfmpz (arb_t z, const fmpz_t x, long prec)
void *arb_sqrt_ui (arb_t z, ulong x, long prec)
   Sets z to the square root of x, rounded to prec bits.
   If x = m ± r where m ≥ r ≥ 0, the propagated error is bounded by
   \[ \sqrt{m} - \sqrt{m-r} = \sqrt{m(1 - \sqrt{1 - r/m})} \leq \sqrt{m(r/m + (r/m)^2)/2}. \]
void *arb_sqrtpos (arb_t z, const arb_t x, 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 *arb_hypot (arb_t z, const arb_t x, const arb_t y, long prec)
   Sets z to \( \sqrt{x^2 + y^2} \).
void *arb_rsqrt (arb_t z, const arb_t x, long prec)
void *arb_rsqrtui (arb_t z, ulong x, 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 *arb_root (arb_t z, const arb_t x, ulong k, 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 k.
   For large k it is better to use arb_pow_fmpq() or arb_pow().
void *arb_pow_fmpz_binexp (arb_t y, const arb_t b, const fmpz_t e, long prec)
void *arb_pow_fmpz (arb_t y, const arb_t b, const fmpz_t e, long prec)
void *arb_pow_ui (arb_t y, const arb_t b, ulong e, long prec)
void *arb_ui_pow_ui (arb_t y, ulong b, ulong e, long prec)
void *arb_sipow_ui (arb_t y, ulong 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 ARF_PREC_EXACT.
   Note that these functions can get slow if the exponent is extremely large (in such cases arb_pow() may be
   superior).
void *arb_pow_fmpq (arb_t y, const arb_t x, const fmpz_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 arb_pow() may be
   superior).
void *arb_pow (arb_t z, const arb_t x, const arb_t y, long prec)
   Sets z = x^y, computed using binary exponentiation if y if a small exact integer, as z = (x^{1/2})^{2y} if y is a small
   exact half-integer, and generally as z = \exp(y \log x).

2.3. arb.h – real numbers represented as floating-point balls
2.3.12 Trigonometric functions

void \texttt{arb_sin} (arb_t s, const arb_t x, long \texttt{prec})

Sets \( s = \sin(x) \).

void \texttt{arb_cos} (arb_t c, const arb_t x, long \texttt{prec})

void \texttt{arb_sin_cos} (arb_t s, arb_t c, const arb_t x, long \texttt{prec})

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

void \texttt{arb_sin_pi} (arb_t s, const arb_t x, long \texttt{prec})

void \texttt{arb_cos_pi} (arb_t c, const arb_t x, long \texttt{prec})

void \texttt{arb_sin_cos_pi} (arb_t s, arb_t c, const arb_t x, long \texttt{prec})

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

void \texttt{arb_tan} (arb_t y, const arb_t x, long \texttt{prec})

Sets \( y = \tan(x) = \sin(x)/\cos(y) \).

void \texttt{arb_cot} (arb_t y, const arb_t x, long \texttt{prec})

Sets \( y = \cot(x) = \cos(x)/\sin(y) \).

void \texttt{arb_sin_cos_pi_fmpq} (arb_t s, arb_t c, const fmpq_t x, long \texttt{prec})

void \texttt{arb_sin_pi_fmpq} (arb_t s, const fmpq_t x, long \texttt{prec})

void \texttt{arb_cos_pi_fmpq} (arb_t c, const fmpq_t x, long \texttt{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 \texttt{arb_tan_pi} (arb_t y, const arb_t x, long \texttt{prec})

Sets \( y = \tan(\pi x) \).

void \texttt{arb_cot_pi} (arb_t y, const arb_t x, long \texttt{prec})

Sets \( y = \cot(\pi x) \).
2.3.13 Inverse trigonometric functions

void \texttt{arb_atan_arf} (arb_t \texttt{z}, const arf_t \texttt{x}, long \texttt{prec})

void \texttt{arb_atan} (arb_t \texttt{z}, const arb_t \texttt{x}, long \texttt{prec})

Sets \( z = \tan^{-1}(x) \).

At low to medium precision (up to about 4096 bits), \texttt{arb_atan_arf()} uses table-based argument reduction and fast Taylor series evaluation via \_arb_atan_taylor_rs(). At high precision, it falls back to MPFR. The function \texttt{arb_atan()} simply calls \texttt{arb_atan_arf()} with the midpoint as input, and separately adds the propagated error. See \textit{Algorithms for elementary functions} for further remarks.

void \texttt{arb_atan2} (arb_t \texttt{z}, const arb_t \texttt{b}, const arb_t \texttt{a}, long \texttt{prec})

Sets \( r \) to an the argument (phase) of the complex number \( a + bi \), with the branch cut discontinuity on \((-\infty, 0] \).

We define \( \tan^{-1}(0, 0) = 0 \), and for \( a < 0 \), \( \tan^{-1}(0, a) = \pi \).

2.3.14 Hyperbolic functions

void \texttt{arb_sinh} (arb_t \texttt{s}, const arb_t \texttt{x}, long \texttt{prec})

void \texttt{arb_cosh} (arb_t \texttt{c}, const arb_t \texttt{x}, long \texttt{prec})

void \texttt{arb_sinh_cosh} (arb_t \texttt{s}, arb_t \texttt{c}, const arb_t \texttt{x}, long \texttt{prec})

Sets \( s = \sinh(x) \), \( c = \cosh(x) \). If the midpoint of \( x \) is close to zero and the hyperbolic sine is to be computed, evaluates \((e^{2x} \pm 1)/(2e^x)\) via \_arb_expm1() to avoid loss of accuracy. Otherwise evaluates \((e^x \pm e^{-x})/2\).

void \texttt{arb_tanh} (arb_t \texttt{y}, const arb_t \texttt{x}, long \texttt{prec})

void \texttt{arb_coth} (arb_t \texttt{y}, const arb_t \texttt{x}, long \texttt{prec})

Sets \( y = \tanh(x) = \sinh(x)/\cosh(x) \), evaluated via \_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.

2.3.15 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} (arb_t \texttt{z}, long \texttt{prec})

Computes \( \pi \).

void \texttt{arb_const_sqrt_pi} (arb_t \texttt{z}, long \texttt{prec})

Computes \( \sqrt{\pi} \).

void \texttt{arb_const_log_sqrt2pi} (arb_t \texttt{z}, long \texttt{prec})

Computes \( \log(\sqrt{2\pi}) \).

void \texttt{arb_const_log2} (arb_t \texttt{z}, long \texttt{prec})

Computes \( \log(2) \).
void \texttt{arb\_const\_log10}(\texttt{arb\_t} z, long prec)
    Computes \( \log(10) \).

void \texttt{arb\_const\_euler}(\texttt{arb\_t} z, 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, 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, long prec)
    Computes \( e = \exp(1) \).

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

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

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

2.3.16 Gamma function and factorials

void \texttt{arb\_rising\_ui\_bs}(\texttt{arb\_t} z, const \texttt{arb\_t} x, ulong n, long prec)
    Computes the rising factorial \( z = x(x+1)(x+2)\cdots(x+n-1) \).
    The \textit{bs} version uses binary splitting. The \textit{rs} version uses rectangular splitting. The \textit{rec} version uses either \textit{bs} or \textit{rs} depending on the input. The default version is currently identical to the \textit{rec} version. In a future version, it will use the gamma function or asymptotic series when this is more efficient.

void \texttt{arb\_rising\_ui\_rs}(\texttt{arb\_t} z, const \texttt{arb\_t} x, ulong n, ulong \textit{step}, long prec)
    Computes the rising factorial \( z = x(x+1)(x+2)\cdots(x+n-1) \) using binary splitting. If the denominator or numerator of \( x \) is large compared to \textit{prec}, it is more efficient to convert \( x \) to an approximation and use \texttt{arb\_rising\_ui\_rec}.

void \texttt{arb\_rising\_ui\_rec}(\texttt{arb\_t} z, const \texttt{arb\_t} x, ulong n, long prec)
    Computes the rising factorial \( z = x(x+1)(x+2)\cdots(x+n-1) \) using binary splitting. If the denominator or numerator of \( x \) is large compared to \textit{prec}, it is more efficient to convert \( x \) to an approximation and use \texttt{arb\_rising\_ui\_rec}.

void \texttt{arb\_rising\_fmpq\_ui}(\texttt{arb\_t} z, const \texttt{fmpq\_t} x, ulong n, long prec)
    Computes the rising factorial \( z = x(x+1)(x+2)\cdots(x+n-1) \) using binary splitting. If the denominator or numerator of \( x \) is large compared to \textit{prec}, it is more efficient to convert \( x \) to an approximation and use \texttt{arb\_rising\_ui\_rec}.

void \texttt{arb\_rising\_2\_ui\_bs}(\texttt{arb\_t} u, \texttt{arb\_t} v, const \texttt{arb\_t} x, ulong n, long prec)
void \texttt{arb\_rising\_2\_ui\_rs}(\texttt{arb\_t} u, \texttt{arb\_t} v, const \texttt{arb\_t} x, ulong n, ulong \textit{step}, 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 \textit{step} to override the default choice), and an automatic algorithm choice.

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

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

void \texttt{arb\_gamma}(\texttt{arb\_t} z, const \texttt{arb\_t} x, long prec)
void \texttt{arb\_gamma\_fmpq}(\texttt{arb\_t} z, const \texttt{fmpq\_t} x, long prec)
void arb_gamma_fmpz (arb_ptr x, ulong start, long num, long prec)
  Computes the gamma function $z = \Gamma(x)$.

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

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

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

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

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

void arb_zeta_ui_asym (arb_ptr x, ulong s, long prec)
  Assuming $s \geq 2$, approximates $\zeta(s)$ by $1 + 2^{-s} \zeta(s)$ along with a correct error bound. We use the following bounds: for $s > b$, $\zeta(s) - 1 < 2^{-b}$, and generally, $\zeta(s) - (1 + 2^{-b}) < 2^{2^{-\text{prec}}}$.

void arb_zeta_ui_euler_product (arb_ptr x, ulong s, long prec)
  Computes $\zeta(s)$ using the Euler product. This is fast only if $s$ is large compared to the precision.

Writing $P(a, b) = \prod_{s \leq r \leq k} (1 - p^{-s})$, 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 arb_zeta_ui_borwein_bsplit (arb_ptr x, ulong s, long prec)
  Computes $\zeta(s)$ for arbitrary $s \geq 2$ using a binary splitting implementation of Borwein’s algorithm. This has quasilinear complexity with respect to the precision (assuming that $s$ is fixed).

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

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

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

Note: the Hurwitz zeta function is also available, but takes complex arguments (see \texttt{acb_hurwitz_zeta()}). For computing derivatives with respect to $s$, use \texttt{arb_poly_zeta_series()}. 

2.3.18 Bernoulli numbers

void **arb_bernoulli_ui** (arb_t b, ulong n, long 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 **arb_bernoulli_ui_zeta** (arb_t b, ulong n, long 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).

2.3.19 Polylogarithms

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

2.3.20 Other special functions

void **arb_fib_fmpz** (arb_t z, const fmpz_t n, long prec)
void **arb_fib_ui** (arb_t z, ulong n, long 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 **arb_agm** (arb_t z, const arb_t x, const arb_t y, long prec)
Sets $z$ to the arithmetic-geometric mean of $x$ and $y$.

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

void **arb_chebyshev_t2_ui** (arb_t a, arb_t b, ulong n, const arb_t x, long prec)
2.3. arb.h – real numbers represented as floating-point balls

2.3.21 Internal helper functions

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

void _arb_exp_arf_bb (arb_t z, const arf_t x, long prec, int m1)
    Computes the exponential function using the bit-burst algorithm. If $m1$ is nonzero, the exponential function minus one is computed accurately.

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

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

    Computing $\log(2)$ costs roughly 100-200 multiplications, so it 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.

## 2.4 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.

### 2.4.1 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.

### 2.4.2 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 sure 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 should not exceed the allocation length.

void **_arb_poly_normalise** (arb_poly_t poly)

Strips any trailing coefficients which are identical to zero.

### 2.4.3 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_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_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 poly, so the return value of this function is effectively an upper bound.

### 2.4.4 Conversions

void **arb_poly_set_fmpz_poly** (arb_poly_t poly, const fmpz_poly_t src, long prec)

void **arb_poly_set_fmpq_poly** (arb_poly_t poly, const fmpq_poly_t src, long prec)

void **arb_poly_set_si** (arb_poly_t poly, long src)

Sets poly to src, rounding the coefficients to prec bits.
2.4.5 Input and output

void arb_poly_printd (const arb_poly_t poly, long digits)
    Prints the polynomial as an array of coefficients, printing each coefficient using arb_printd.

2.4.6 Random generation

void arb_poly_randtest (arb_poly_t poly, flint_rand_t state, long len, long prec, long mag_bits)
    Creates a random polynomial with length at most len.

2.4.7 Comparisons

int arb_poly_contains (const arb_poly_t poly1, const arb_poly_t poly2)
    Returns nonzero iff poly1 contains poly2.

int arb_poly_contains_fmpz_poly (const arb_poly_t poly1, const fmpz_poly_t poly2)
    Returns nonzero iff poly1 contains poly2.

int arb_poly_contains_fmpq_poly (const arb_poly_t poly1, const fmpq_poly_t poly2)
    Returns nonzero iff poly1 contains poly2.

int arb_poly_equal (const arb_poly_t A, const arb_poly_t B)
    Returns nonzero iff A and B are equal as polynomial balls, i.e. all coefficients have equal midpoint and radius.

int _arb_poly_overlaps (arb_srcptr poly1, long len1, arb_srcptr poly2, long len2)
    Returns nonzero iff poly1 overlaps with poly2. The underscore function requires that len1 is at least as large as len2.

2.4.8 Arithmetic

void _arb_poly_add (arb_ptr C, arb_srcptr A, long lenA, arb_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 arb_poly_add (arb_poly_t C, const arb_poly_t A, const arb_poly_t B, long prec)
    Sets C to the sum of A and B.

void _arb_poly_sub (arb_ptr C, arb_srcptr A, long lenA, arb_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 arb_poly_sub (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_poly_t 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, lenA\} and \{B, lenB\}, truncated to length \(n\). The output is not allowed to be aliased with either of the inputs. We require \(\text{lenA} \geq \text{lenB} > 0, n > 0, \text{lenA} + \text{lenB} - 1 \geq n\).

The \textit{classical} version uses a plain loop. This has good numerical stability but gets slow for large \(n\).

The \textit{block} version decomposes the product into several subproducts which are computed exactly over the integers.

It first attempts to find an integer \(c\) such that \(A(2^c x)\) and \(B(2^c x)\) have slowly varying coefficients, to reduce the number of blocks.

The scaling factor \(c\) is chosen in a quick, heuristic way by picking the first and last nonzero terms in each polynomial. If the indices in \(A\) are \(a_2, a_1\) and the log-2 magnitudes are \(e_2, e_1\), and the indices in \(B\) are \(b_2, b_1\) with corresponding magnitudes \(f_2, f_1\), then we compute \(c\) as the weighted arithmetic mean of the slopes, rounded to the nearest integer:

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

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

The default algorithm chooses the \textit{classical} algorithm for short polynomials and the \textit{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 \textbf{arb\_poly\_mullow\_classical} (arb\_poly\_t C, const arb\_poly\_t A, const arb\_poly\_t B, long n, long prec)
void \textbf{arb\_poly\_mullow\_ztrunc} (arb\_poly\_t C, const arb\_poly\_t A, const arb\_poly\_t B, long n, long prec)
void \textbf{arb\_poly\_mullow\_block} (arb\_poly\_t C, const arb\_poly\_t A, const arb\_poly\_t B, long n, long prec)
void \textbf{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_mul (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 \textbf{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 \(\{Q, \text{len}\}\) to the power series inverse of \(\{A, \text{Alen}\}\). Uses Newton iteration.

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

void \textbf{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 \(A\) divided by \(B\), truncated to length \(n\).
2.4.9 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 prec)
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)
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_compose_series_brent_kung (arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, long n, long prec)
void _arb_poly_compose_series (arb_ptr res, arb_srcptr poly1, long len1, arb_srcptr poly2, long len2, 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_newton (arb_ptr h, arb_srcptr f, long flen, long n, long prec)
void _arb_poly_revert_series_newton (arb_poly_t h, const arb_poly_t f, 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\_lagrange\_fast (arb\_poly\_t h, const arb\_poly\_t f, long n, long prec)

void \_arb\_poly\_revert\_series (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)

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.

### 2.4.10 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)

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

void \_arb\_poly\_evaluate\_acb\_horner (acb\_t y, arb\_srcptr f, long len, const acb\_t x, long prec)

void \_arb\_poly\_evaluate\_acb\_horner (acb\_t y, const arb\_poly\_t f, const acb\_t x, long prec)

void \_arb\_poly\_evaluate\_acb\_rectangular (acb\_t y, arb\_srcptr f, long len, const acb\_t x, long prec)

void \_arb\_poly\_evaluate\_acb\_rectangular (acb\_t y, const arb\_poly\_t f, const acb\_t x, long prec)

void \_arb\_poly\_evaluate\_acb (acb\_t y, arb\_srcptr f, long len, const acb\_t x, long prec)

void \_arb\_poly\_evaluate\_acb (acb\_t y, const arb\_poly\_t f, const acb\_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\_horner (arb\_t y, arb\_t z, arb\_srcptr f, long len, const arb\_t x, long prec)

void \_arb\_poly\_evaluate2\_horner (arb\_t y, arb\_t z, const arb\_poly\_t f, const arb\_t x, long prec)

void \_arb\_poly\_evaluate2\_rectangular (arb\_t y, arb\_t z, arb\_srcptr f, long len, const arb\_t x, long prec)

void \_arb\_poly\_evaluate2\_rectangular (arb\_t y, arb\_t z, const arb\_poly\_t f, const arb\_t x, long prec)

void \_arb\_poly\_evaluate2 (arb\_t y, arb\_t z, arb\_srcptr f, long len, const arb\_t x, long prec)

void \_arb\_poly\_evaluate2 (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.
Sets \( y = f(x) \), \( z = f'(x) \), evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

### 2.4.11 Product trees

- **void _arb_poly_product_roots** (arb_ptr poly, arb_srcptr xs, long n, long prec)
- **void arb_poly_product_roots** (arb_poly_t poly, arb_srcptr xs, long n, long prec)

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

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

### 2.4.12 Multipoint evaluation

- **void _arb_poly_evaluate_vec_iter** (arb_ptr ys, arb_srcptr poly, long plen, arb_srcptr xs, long n, long prec)
- **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.

### 2.4.13 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)

Recover the unique polynomial of length at most n that interpolates the given x and y values. This implementation uses the barycentric form of Lagrange interpolation.

void _arb_poly_interpolation_weights (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.

2.4.14 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.

2.4.15 Transforms

void _arb_poly_borel_transform (arb_ptr res, arb_srcptr poly, long len, long prec)

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)

2.4. arb_poly.h – polynomials over the real numbers
2.4.16 Powers and elementary functions

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 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.

void **_arb_poly_pow_ui_trunc_binexp** (arb_ptr res, arbc_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 **arb_poly_pow_ui_trunc_binexp** (arb_poly_t res, const arb_poly_t poly, long exp, long len, long prec)

Sets res to poly raised to the power exp, truncated to length len. Uses binary exponentiation.

void **_arb_poly_pow_ui** (arb_ptr res, arbc_srcptr f, long flen, ulong exp, long prec)

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

void **arb_poly_pow_ui** (arb_poly_t res, const arb_poly_t poly, ulong exp, long prec)

Sets res to poly raised to the power exp.

void **_arb_poly_pow_series** (arb_ptr h, arbc_srcptr f, long flen, arbc_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 $\pm1/2$, 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 $\pm1/2$, and computes such powers more efficiently.

void **_arb_poly_pow_arb_series** (arb_ptr h, arbc_srcptr f, long flen, 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. This function detects special cases such as g being an exact small integer or $\pm1/2$, 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_sqrt_series** (arb_ptr g, arbc_srcptr h, long hlen, long n, long prec)
void \texttt{arb\_poly\_sqrt\_series} (\texttt{arb\_poly\_t} g, \texttt{const arb\_poly\_t} h, long n, long \texttt{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 \texttt{\_arb\_poly\_rsqrt\_series} (\texttt{arb\_ptr} g, \texttt{arb\_srcptr} h, long \texttt{hlen}, long \texttt{n}, long \texttt{prec})

void \texttt{arb\_poly\_rsqrt\_series} (\texttt{arb\_poly\_t} g, \texttt{const arb\_poly\_t} h, long \texttt{n}, long \texttt{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 \texttt{\_arb\_poly\_log\_series} (\texttt{arb\_ptr} res, \texttt{arb\_srcptr} f, long \texttt{flen}, long \texttt{n}, long \texttt{prec})

void \texttt{arb\_poly\_log\_series} (\texttt{arb\_poly\_t} res, \texttt{const arb\_poly\_t} f, long \texttt{n}, long \texttt{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 \texttt{\_arb\_poly\_atan\_series} (\texttt{arb\_ptr} res, \texttt{arb\_srcptr} f, long \texttt{flen}, long \texttt{n}, long \texttt{prec})

void \texttt{arb\_poly\_atan\_series} (\texttt{arb\_poly\_t} res, \texttt{const arb\_poly\_t} f, long \texttt{n}, long \texttt{prec})

void \texttt{\_arb\_poly\_asin\_series} (\texttt{arb\_ptr} res, \texttt{arb\_srcptr} f, long \texttt{flen}, long \texttt{n}, long \texttt{prec})

void \texttt{arb\_poly\_asin\_series} (\texttt{arb\_poly\_t} res, \texttt{const arb\_poly\_t} f, long \texttt{n}, long \texttt{prec})

void \texttt{\_arb\_poly\_acos\_series} (\texttt{arb\_ptr} res, \texttt{arb\_srcptr} f, long \texttt{flen}, long \texttt{n}, long \texttt{prec})

void \texttt{arb\_poly\_acos\_series} (\texttt{arb\_poly\_t} res, \texttt{const arb\_poly\_t} f, long \texttt{n}, long \texttt{prec})

Sets \( res \) respectively to the power series inverse tangent, inverse sine and inverse cosine of \( f \), truncated to length \( n \).

Uses the formulas

\[
\tan^{-1}(f(x)) = \int f'(x)/(1 + f(x)^2) \, dx,
\]

\[
\sin^{-1}(f(x)) = \int f'(x)/(1 - f(x)^2)^{1/2} \, dx,
\]

\[
\cos^{-1}(f(x)) = -\int f'(x)/(1 - f(x)^2)^{1/2} \, dx,
\]

adding the inverse function of the constant term in \( f \) as the constant of integration.

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

void \texttt{\_arb\_poly\_exp\_series\_basecase} (\texttt{arb\_ptr} f, \texttt{arb\_srcptr} h, long \texttt{hlen}, long \texttt{n}, long \texttt{prec})

void \texttt{arb\_poly\_exp\_series\_basecase} (\texttt{arb\_poly\_t} f, \texttt{const arb\_poly\_t} h, long \texttt{n}, long \texttt{prec})

void \texttt{\_arb\_poly\_exp\_series} (\texttt{arb\_ptr} f, \texttt{arb\_srcptr} h, long \texttt{hlen}, long \texttt{n}, long \texttt{prec})

void \texttt{arb\_poly\_exp\_series} (\texttt{arb\_poly\_t} f, \texttt{const arb\_poly\_t} h, long \texttt{n}, long \texttt{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)
void arb_poly_sin_cos_series_basecase (arb_poly_t s, arb_poly_t c, const arb_poly_t h, long n, long prec)
void _arb_poly_sin_cos_series_tangent (arb_ptr s, arb_ptr c, arb_srcptr h, long hlen, long n, long prec)
void arb_poly_sin_cos_series_tangent (arb_poly_t s, arb_poly_t c, const arb_poly_t h, long n, long prec)
void _arb_poly_sin_cos_series (arb_ptr s, arb_ptr c, arb_srcptr h, long hlen, long n, long prec)
void arb_poly_sin_cos_series (arb_poly_t s, arb_poly_t c, const arb_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 _arb_poly_tan_series(). This requires $O(M(n))$ operations. When $h = h_0 + h_1$ where the constant term $h_0$ is nonzero, the evaluation is done as $\sin(h_0 + h_1) = \cos(h_0) \sin(h_1) + \sin(h_0) \cos(h_1)$, $\cos(h_0 + h_1) = \cos(h_0) \cos(h_1) - \sin(h_0) \sin(h_1)$, to improve accuracy and avoid dividing by zero at the poles of the tangent function.

The default version automatically selects between the basecase and tangent algorithms depending on the input.

The 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_cos_series (arb_ptr c, arb_srcptr h, long hlen, long n, long prec)
void arb_poly_cos_series (arb_poly_t c, const arb_poly_t h, long n, long prec)
void _arb_poly_tan_series (arb_ptr g, arb_srcptr h, long hlen, long len, long prec)
void arb_poly_tan_series (arb_poly_t g, const arb_poly_t h, long n, long prec)
```

Respectively evaluates the power series sine or cosine. These functions simply wrap _arb_poly_sin_cos_series(). The underscore methods support aliasing and require the lengths to be nonzero.

Sets $g$ to the power series tangent of $h$.

For small $n$ takes the quotient of the sine and cosine as computed using the basecase algorithm. For large $n$, uses Newton iteration to invert the inverse tangent series. The complexity is $O(M(n))$.

The underscore version does not support aliasing, and requires the lengths to be nonzero.

### 2.4.17 Gamma function and factorials

```c
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_riemann_siegel_theta_series (arb_ptr res, arb_srcptr h, long hlen, long n, long prec)
void _arb_poly_lgamma_series (arb_ptr 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)

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.

## 2.4.18 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
\[
\sum_{k=0}^{n-1} (-1)^k \frac{\gamma_k(a)x^k}{k!}.
\]

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_theta_series (arb_poly_t res, const arb_poly_t h, long n, long prec)
Sets res to the series expansion of the Riemann-Siegel theta function
\[
\theta(h) = \arg \left( \Gamma \left( \frac{2th + 1}{4} \right) \right) - \frac{\log \pi}{2} h
\]
where the argument of the gamma function is chosen continuously as the imaginary part of the log gamma function.

The underscore method does not support aliasing of the input and output arrays, and requires that the lengths are greater than zero.

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

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

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

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

void _arb_poly_newton_refine_root (arb_t r, arb_srcptr poly, long len, const arb_t start, const arb_t convergence_interval, const arf_t convergence_factor, 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: prec is the target accuracy and eval_extra_prec is the estimated number of guard bits that need to be added to evaluate the polynomial accurately close to the root (typically, if the polynomial has large coefficients of alternating signs, this needs to be approximately the bit size of the coefficients).

2.5 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.

2.5.1 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.

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.5.5 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.

2.5.6 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.
2.5.7 Norms

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

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

2.5.8 Arithmetic

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

Sets $dest$ to the exact negation of $src$. The operands must have the same dimensions.

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

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

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

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

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

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

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

Sets $res$ to the matrix product of $mat1$ and $mat2$. The operands must have compatible dimensions for matrix multiplication.

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

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

Sets $res$ to $mat$ raised to the power $exp$. Requires that $mat$ is a square matrix.

2.5.9 Scalar arithmetic

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

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

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

void \texttt{arb\_mat\_scalar\_addmul\_fmpz}(\texttt{arb\_mat\_t} B, \texttt{const arb\_mat\_t} A, \texttt{const fmpz\_t} c, \texttt{long} prec)

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

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

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

void \texttt{arb\_mat\_scalar\_mul\_fmpz}(\texttt{arb\_mat\_t} B, \texttt{const arb\_mat\_t} A, \texttt{const fmpz\_t} c, \texttt{long} prec)

void \texttt{arb\_mat\_scalar\_mul\_arb}(\texttt{arb\_mat\_t} B, \texttt{const arb\_mat\_t} A, \texttt{const arb\_t} c, \texttt{long} prec)

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

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

void \texttt{arb\_mat\_scalar\_div\_fmpz}(\texttt{arb\_mat\_t} B, \texttt{const arb\_mat\_t} A, \texttt{const fmpz\_t} c, \texttt{long} prec)

void \texttt{arb\_mat\_scalar\_div\_arb}(\texttt{arb\_mat\_t} B, \texttt{const arb\_mat\_t} A, \texttt{const arb\_t} c, \texttt{long} prec)

Sets $B$ to $A/c$. 

60 Chapter 2. Module documentation (Arb 2.x types)
2.5.10 Gaussian elimination and solving

```c
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 \( \text{perm} \) is set to the row index in the input matrix corresponding to row \( i \) in the output matrix.

The algorithm succeeds and returns nonzero if it can find \( n \) invertible (i.e. not containing zero) pivot entries. This guarantees that the matrix is invertible.

The algorithm fails and returns zero, leaving the entries in \( P \) and \( LU \) undefined, if it cannot find \( n \) invertible pivot elements. In this case, either the matrix is singular, the input matrix was computed to insufficient precision, or the LU decomposition was attempted at insufficient precision.

```c
void arb_mat_solve_lu_precomp (arb_mat_t X, const long * perm, const arb_mat_t LU, const arb_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 \( LU \).

```c
int arb_mat_solve (arb_mat_t X, const arb_mat_t A, const arb_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.

```c
int arb_mat_inv (arb_mat_t X, const arb_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.

```c
void arb_mat_det (arb_t det, const arb_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.

2.5.11 Special functions

```c
void arb_mat_exp (arb_mat_t B, const arb_mat_t A, long 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 \( 2c^{N}/N! \).

2.6 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.

### 2.6.1 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.

### 2.6.2 Debugging

**int arb_calc_verbose**

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

### 2.6.3 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_srcptr**

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

```c
void arf_interval_init(arf_interval_t v)
void arf_interval_clear(arf_interval_t v)
```
long \texttt{arf_calc_isolate_roots} (\texttt{arf_interval_ptr} * \texttt{found}, \texttt{int} ** \texttt{flags}, \texttt{arb_calc_func_t} \texttt{func}, \texttt{void} * \texttt{param}, \texttt{const arf_interval_t} \texttt{interval}, \texttt{long maxdepth}, \texttt{long maxeval}, \texttt{long maxfound}, \texttt{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 \( 1 \pm 2^{-1000} \) as input.

int \texttt{arb_calc_refine_root_bisect} (\texttt{arf_interval_t} \texttt{r}, \texttt{arb_calc_func_t} \texttt{func}, \texttt{void} * \texttt{param}, \texttt{const arf_interval_t} \texttt{start}, \texttt{long iter}, \texttt{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.
2.6.4 Newton-based root finding

void \texttt{arb\_calc\_newton\_conv\_factor} (\texttt{arf\_t conv\_factor}, \texttt{arb\_calc\_func\_t func}, \texttt{void * param}, \texttt{const arb\_t conv\_region}, \texttt{long prec})

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

int \texttt{arb\_calc\_newton\_step} (\texttt{arb\_t xnew}, \texttt{arb\_calc\_func\_t func}, \texttt{void * param}, \texttt{const arb\_t x}, \texttt{const arb\_t conv\_region}, \texttt{const arb\_t conv\_factor}, \texttt{long prec})

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

The Newton update consists of setting \( x' = [m' - r', m' + r'] \) where \( m' = m - f(m)/f'(m) \) and \( r' = Cr^2 \).

The expression \( m - f(m)/f'(m) \) is evaluated using ball arithmetic at a working precision of \( \texttt{prec} \) bits, and the rounding error during this evaluation is accounted for in the output. We now check that \( x' \in I \) and \( r' < r \). If both conditions are satisfied, we set \( \texttt{xnew} \) to \( x' \) and return \texttt{ARB\_CALC\_SUCCESS}. If either condition fails, we set \( \texttt{xnew} \) to \( x \) and return \texttt{ARB\_CALC\_NO\_CONVERGENCE}, indicating that no progress is made.

int \texttt{arb\_calc\_refine\_root\_newton} (\texttt{arb\_t r}, \texttt{arb\_calc\_func\_t func}, \texttt{void * param}, \texttt{const arb\_t start}, \texttt{const arb\_t conv\_region}, \texttt{const arb\_t conv\_factor}, \texttt{long eval\_extra\_prec}, \texttt{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 \texttt{arb\_calc\_newton\_step}, except for the precision parameters: \( \texttt{prec} \) is the target accuracy and \( \texttt{eval\_extra\_prec} \) is the estimated number of guard bits that need to be added to evaluate the function accurately close to the root (for example, if the function is a polynomial with large coefficients of alternating signs and Horner’s rule is used to evaluate it, the extra precision should typically be approximately the bit size of the coefficients).

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

On failure, \texttt{ARB\_CALC\_ IMPRECISE\_INPUT} or \texttt{ARB\_CALC\_NO\_ CONVERGENCE} may be returned. In this case, \( \texttt{r} \) 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).

2.7 \texttt{acb\_h} – complex numbers

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

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

The interface for the \texttt{acb\_t} type is slightly less developed than that for the \texttt{arb\_t} type. In many cases, the user can easily perform missing operations by directly manipulating the real and imaginary parts.
2.7.1 Types, macros and constants

acb_struct

acb_t
Anacb_structconsists of a pair ofarb_structs. Anacb_tis defined as an array of length one of typeacb_struct, permitting anacb_tto be passed by reference.

acb_ptr
Alias foracb_struct*, used for vectors of numbers.

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

acb_realref(x)
Macro returning a pointer to the real part ofxasaarb_t.

acb_imagref(x)
Macro returning a pointer to the imaginary part ofxasaarb_t.

2.7.2 Memory management

voidacb_init(arbt x)
Initializes the variablexfor use, and sets its value to zero.

voidacb_clear(arbt x)
Clears the variablex, freeing or recycling its allocated memory.

acb_ptracb_vec_init(longn)
Returns a pointer to an array ofninitializedacb_structs.

voidacb_vec_clear(acb_ptrv, longn)
Clears an array ofninitializedacb_structs.

2.7.3 Basic manipulation

intacb_is_zero(constacb_tz)
Returns nonzero iffzis zero.

intacb_is_one(constacb_tz)
Returns nonzero iffzis exactly 1.

intacb_is_exact(constacb_tz)
Returns nonzero iffzis exact.

voidacb_zero(acb_tz)
voidacb_one(acb_tz)
voidacb_onei(acb_tz)
Setszrespectively to0, 1, i = \sqrt{-1}.

voidacb_set(acb_tz, constacb_tx)
voidacb_set_ui(acb_tz, longx)
voidacb_set_si(acb_tz, longx)
voidacb_set_fmpz(acb_tz, constfmpz_tx)
void \texttt{acb\_set\_arb} (acb\_t \texttt{z}, \texttt{const arb\_t c})
  Sets \texttt{z} to the value of \texttt{x}.

void \texttt{acb\_set\_fmpq} (acb\_t \texttt{z}, \texttt{const fmpq\_t x}, long \texttt{prec})

void \texttt{acb\_set\_round} (acb\_t \texttt{z}, \texttt{const acb\_t x}, long \texttt{prec})

void \texttt{acb\_set\_round\_fmpz} (acb\_t \texttt{z}, \texttt{const fmpz\_t x}, long \texttt{prec})
  Sets \texttt{z} to \texttt{x}, rounded to \texttt{prec} bits.

void \texttt{acb\_set\_round\_arb} (acb\_t \texttt{z}, \texttt{const arb\_t x}, long \texttt{prec})

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

2.7.4 Input and output

void \texttt{acb\_print} (\texttt{const acb\_t x})
  Prints the internal representation of \texttt{x}.

void \texttt{acb\_printd} (\texttt{const acb\_t z}, long \texttt{digits})
  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.

2.7.5 Random number generation

void \texttt{acb\_randtest} (acb\_t \texttt{z}, flint\_rand\_t \texttt{state}, long \texttt{prec}, long \texttt{mag\_bits})
  Generates a random complex number by generating separate random real and imaginary parts.

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

2.7.6 Precision and comparisons

int \texttt{acb\_equal} (\texttt{const acb\_t x}, \texttt{const acb\_t y})
  Returns nonzero iff \texttt{x} and \texttt{y} are identical as sets, i.e. if the real and imaginary parts are equal as balls.

Note that this is not the same thing as testing whether both \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 x}, \texttt{const acb\_t y})
  Returns nonzero iff \texttt{x} and \texttt{y} have some point in common.

void \texttt{acb\_get\_abs\_ubound\_arf} (arf\_t \texttt{u}, \texttt{const acb\_t z}, long \texttt{prec})
  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} (arf\_t \texttt{u}, \texttt{const acb\_t z}, long \texttt{prec})
  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} (arf\_t \texttt{u}, \texttt{const acb\_t z}, long \texttt{prec})
  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 x})
  Sets \texttt{u} to an upper bound for the absolute value of \texttt{x}. 

66 Chapter 2. Module documentation (Arb 2.x types)
void **acb_get_mag_lower** (mag_t u, const acb_t x)
Sets u to a lower bound for the absolute value of x.

int **acb_contains_fmpq** (const acb_t x, const fmpq_t y)
int **acb_contains_fmpz** (const acb_t x, const fmpz_t y)

int **acb_contains** (const acb_t x, const acb_t y)
Returns nonzero iff y is contained in x.

int **acb_contains_zero** (const acb_t x)
Returns nonzero iff zero is contained in x.

long **acb_bits** (const acb_t x)
Returns the maximum of arb_bits applied to the real and imaginary parts of x, i.e. the minimum precision sufficient to represent x exactly.

void **acb_trim** (acb_t y, const acb_t x)
Sets y to a a copy of x with both the real and imaginary parts trimmed (see arb_trim()).

int **acb_is_real** (const acb_t x)
Returns nonzero iff the imaginary part of x is zero. It does not test whether the real part of x also is finite.

### 2.7.7 Complex parts

void **acb_arg** (arb_t r, const acb_t z, long prec)
Sets r to a real interval containing the complex argument (phase) of z. We define the complex argument have a discontinuity on \((-\infty, 0]\), with the special value \(\arg(0) = 0\), and \(\arg(a + 0i) = \pi\) for \(a < 0\). Equivalently, if \(z = a + bi\), the argument is given by \(\text{atan2}(b, a)\) (see arb_atan2()).

void **acb_abs** (arb_t r, const acb_t z, long prec)
Sets r to the absolute value of z.

### 2.7.8 Arithmetic

void **acb_neg** (acb_t z, const acb_t x)
Sets z to the negation of x.

void **acb_conj** (acb_t z, const acb_t x)
Sets z to the complex conjugate of x.

void **acb_add_ui** (acb_t z, const acb_t x, ulong y, long prec)
void **acb_add_fmpz** (acb_t z, const acb_t x, const fmpz_t y, long prec)
void **acb_add_arb** (acb_t z, const acb_t x, const arb_t y, long prec)
void **acb_add** (acb_t z, const acb_t x, const acb_t y, long prec)
Sets z to the sum of x and y.

void **acb_sub_ui** (acb_t z, const acb_t x, ulong y, long prec)
void **acb_sub_fmpz** (acb_t z, const acb_t x, const fmpz_t y, long prec)
void **acb_sub_arb** (acb_t z, const acb_t x, const arb_t y, long prec)
void **acb_sub** (acb_t z, const acb_t x, const acb_t y, long prec)
Sets z to the difference of x and y.

void **acb_mul_onei** (acb_t z, const acb_t x)
Sets z to x multiplied by the imaginary unit.
void \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} (acb\_t \textit{z}, const acb\_t \textit{x}, const acb\_t \textit{y}, long \textit{prec})

Sets \textit{z} to the product of \textit{x} and \textit{y}. If at least one part of \textit{x} or \textit{y} is zero, the operations is reduced to two real multiplications. If \textit{x} and \textit{y} are the same pointers, they are assumed to represent the same mathematical quantity and the squaring formula is used.

void \texttt{acb\_mul\_2exp\_si} (acb\_t \textit{z}, const acb\_t \textit{x}, long \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} (acb\_t \textit{z}, const acb\_t \textit{x}, const acb\_t \textit{y}, long \textit{prec})

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 the product of \textit{x} and \textit{y}.

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

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} plus 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 \textit{z} minus the product of \textit{x} and \textit{y}.

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}.

### 2.7.9 Elementary functions

void \texttt{acb\_const\_pi} (acb\_t \textit{y}, long \textit{prec})

Sets \textit{y} to the constant \(\pi\).

void \texttt{acb\_log} (acb\_t \textit{y}, const acb\_t \textit{z}, long \textit{prec})

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

void \texttt{acb\_exp} (acb\_t \textit{y}, const acb\_t \textit{z}, long \textit{prec})

Sets \textit{y} to the exponential function of \textit{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 arb_expm1() and arb_sin_cos_pi().

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 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, arb_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 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 acb_tan(), but evaluating the sine and cosine accurately via 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 acb_cot(), but evaluating the sine and cosine accurately via arb_sin_cos_pi().

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 acb_pow() may be superior).

void acb_pow_arb (acb_t z, const acb_t x, const arb_t y, long prec)

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

void acb_sqrt (acb_t r, const 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 acb_rsqrt (acb_t r, const 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 $1/\sqrt{a + bi} = ((a + r) - bi)/v, r = |a + bi|, v = \sqrt{r^2a + bi + r^2}$, requiring one real square root and one real reciprocal square root.
2.7.10 Rising factorials

void *acb_rising_ui_bs (acb_t z, const acb_t x, ulong n, long prec)
void *acb_rising_ui_rs (acb_t z, const acb_t x, ulong n, ulong step, long prec)
void *acb_rising_ui_rec (acb_t z, const acb_t x, ulong n, long prec)
void *acb_rising_ui (acb_t z, const acb_t x, ulong n, long prec)

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

The bs version uses binary splitting. The rs version uses rectangular splitting. The rec version uses either bs or rs depending on the input. The default version is currently identical to the rec version. In a future version, it will use the gamma function or asymptotic series when this is more efficient.

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

void *acb_rising2_ui_bs (acb_t u, acb_t v, const acb_t x, ulong n, long prec)
void *acb_rising2_ui_rs (acb_t u, acb_t v, const acb_t x, ulong n, ulong step, long prec)
void *acb_rising2_ui (acb_t u, acb_t v, const 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 step to override the default choice), and an automatic algorithm choice.

2.7.11 Gamma function

void *acb_gamma (acb_t y, const acb_t x, long prec)
void *acb_ggamma (acb_t y, const acb_t x, long prec)
void *acb_lgamma (acb_t y, const acb_t x, long prec)

Computes the gamma function \( y = \Gamma(x) \).

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

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.

void *acb_digamma (acb_t y, const acb_t x, long prec)

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

2.7.12 Zeta function

void *acb_zeta (acb_t z, const acb_t s, long prec)
void *acb_hurwitz_zeta (acb_t z, const acb_t s, const acb_t a, long prec)

Sets \( z \) to the value of the Riemann zeta function \( \zeta(s) \). Note: for computing derivatives with respect to \( s \), use acb_poly_zeta_series() or related methods.

Sets \( z \) to the value of the Hurwitz zeta function \( \zeta(s, a) \). Note: for computing derivatives with respect to \( s \), use acb_poly_zeta_series() or related methods.
2.7.13 Polylogarithms

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

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

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

2.8 \texttt{acb\_poly.h} – polynomials over the complex numbers

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

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

2.8.1 Types, macros and constants

\texttt{acb\_poly\_struct}

\texttt{acb\_poly\_t}

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

An \texttt{acb\_poly\_t} is defined as an array of length one of type \texttt{acb\_poly\_struct}, permitting an \texttt{acb\_poly\_t} to be passed by reference.

2.8.2 Memory management

void \texttt{acb\_poly\_init}(\texttt{acb\_poly\_t poly})

Initializes the polynomial for use, setting it to the zero polynomial.

void \texttt{acb\_poly\_clear}(\texttt{acb\_poly\_t poly})

Clears the polynomial, deallocating all coefficients and the coefficient array.

void \texttt{acb\_poly\_fit\_length}(\texttt{acb\_poly\_t poly}, \texttt{long len})

Makes sure that the coefficient array of the polynomial contains at least $len$ initialized coefficients.

void \texttt{acb\_poly\_set\_length}(\texttt{acb\_poly\_t poly}, \texttt{long len})

Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void \texttt{acb\_poly\_normalise}(\texttt{acb\_poly\_t poly})

Strips any trailing coefficients which are identical to zero.

void \texttt{acb\_poly\_swap}(\texttt{acb\_poly\_t poly1}, \texttt{acb\_poly\_t poly2})

Swaps $poly1$ and $poly2$ efficiently.

2.8.3 Basic properties and manipulation

long \texttt{acb\_poly\_length}(\texttt{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 \texttt{acb\_poly\_degree}(\texttt{const acb\_poly\_t poly})
\begin{itemize}
\item Returns the degree of \texttt{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.
\end{itemize}

void \texttt{acb\_poly\_zero}(\texttt{acb\_poly\_t poly})
\begin{itemize}
\item Sets \texttt{poly} to the zero polynomial.
\end{itemize}

void \texttt{acb\_poly\_one}(\texttt{acb\_poly\_t poly})
\begin{itemize}
\item Sets \texttt{poly} to the constant polynomial 1.
\end{itemize}

void \texttt{acb\_poly\_set}(\texttt{acb\_poly\_t dest, const acb\_poly\_t src})
\begin{itemize}
\item Sets \texttt{dest} to a copy of \texttt{src}.
\end{itemize}

void \texttt{acb\_poly\_set\_coeff\_si}(\texttt{acb\_poly\_t poly, long n, long c})
\begin{itemize}
\item Sets the coefficient with index \texttt{n} in \texttt{poly} to the value \texttt{c}. We require that \texttt{n} is nonnegative.
\end{itemize}

void \texttt{acb\_poly\_set\_coeff\_acb}(\texttt{acb\_poly\_t poly, long n, const acb\_t c})
\begin{itemize}
\item Sets \texttt{v} to the value of the coefficient with index \texttt{n} in \texttt{poly}. We require that \texttt{n} is nonnegative.
\end{itemize}

\texttt{acb\_poly\_get\_coeff\_ptr}(\texttt{poly, n})
\begin{itemize}
\item Given \texttt{n} \geq 0, returns a pointer to coefficient \texttt{n} of \texttt{poly}, or NULL if \texttt{n} exceeds the length of \texttt{poly}.
\end{itemize}

void \texttt{acb\_poly\_shift\_right}(\texttt{acb\_ptr res, acb\_srcptr poly, long len, long n})
\begin{itemize}
\item Sets \texttt{res} to \texttt{poly} divided by \texttt{x}^{\texttt{n}}, throwing away the lower coefficients. We require that \texttt{n} is nonnegative.
\end{itemize}

void \texttt{acb\_poly\_shift\_left}(\texttt{acb\_ptr res, acb\_srcptr poly, long len, long n})
\begin{itemize}
\item Sets \texttt{res} to \texttt{poly} multiplied by \texttt{x}^{\texttt{n}}. We require that \texttt{n} is nonnegative.
\end{itemize}

void \texttt{acb\_poly\_truncate}(\texttt{acb\_poly\_t poly, long n})
\begin{itemize}
\item Truncates \texttt{poly} to have length at most \texttt{n}, i.e. degree strictly smaller than \texttt{n}.
\end{itemize}

2.8.4 Input and output

void \texttt{acb\_poly\_printd}(\texttt{const acb\_poly\_t poly, long digits})
\begin{itemize}
\item Prints the polynomial as an array of coefficients, printing each coefficient using \texttt{arb\_printd}.
\end{itemize}

2.8.5 Random generation

void \texttt{acb\_poly\_randtest}(\texttt{acb\_poly\_t poly, flint\_rand\_t state, long len, long prec, long mag\_bits})
\begin{itemize}
\item Creates a random polynomial with length at most \texttt{len}.
\end{itemize}

2.8.6 Comparisons

int \texttt{acb\_poly\_equal}(\texttt{const acb\_poly\_t A, const acb\_poly\_t B})
\begin{itemize}
\item Returns nonzero if \texttt{A} and \texttt{B} are identical as interval polynomials.
\end{itemize}

int \texttt{acb\_poly\_contains}(\texttt{const acb\_poly\_t poly1, const acb\_poly\_t poly2})
\begin{itemize}
\item Returns nonzero if \texttt{poly2} is contained in \texttt{poly1}.
\end{itemize}

int \texttt{acb\_poly\_contains\_fmpz\_poly}(\texttt{const acb\_poly\_t poly1, const fmpz\_poly\_t poly2})
\begin{itemize}
\item Returns nonzero if \texttt{poly2} is contained in \texttt{poly1}.
\end{itemize}

int \texttt{acb\_poly\_contains\_fmpq\_poly}(\texttt{const acb\_poly\_t poly1, const fmpq\_poly\_t poly2})
\begin{itemize}
\item Returns nonzero if \texttt{poly2} is contained in \texttt{poly1}.
int _acb_poly_overlaps (acb_srcptr poly1, long len1, acb_srcptr 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.

### 2.8.7 Conversions

void acb_poly_set_fmpz_poly (acb_poly_t poly, const fmpz_poly_t re, 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.

### 2.8.8 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.

---

2.8. acb_poly.h – polynomials over the complex numbers
The \emph{transpose} version evaluates the product using four real polynomial multiplications (via \_acb_poly_mullow()).

The \emph{transpose_gauss} version evaluates the product using three real polynomial multiplications. This is almost always faster than \emph{transpose}, but has worse numerical stability when the coefficients vary in magnitude.

The default function \_acb_poly_mullow() automatically switches between \emph{classical} and \emph{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.

\begin{verbatim}
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)

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 \geq 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_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 (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 leading coefficient of \(B\) must not contain zero. The implementation reverses the inputs and performs power series division.

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 \(R = f(c)\).
\end{verbatim}
2.8.9 Composition

void _acb_poly-compose_hornern (acb_ptr res, acb_srcptr poly1, long len1, acb_srcptr poly2, long len2, long prec)
void acb_poly-compose_hornern (acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, long prec)
void _acb_poly-compose_divconquer (acb_ptr res, acb_srcptr poly1, long len1, acb_srcptr poly2, long len2, long prec)
void acb_poly-compose_divconquer (acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, long prec)
void _acb_poly-compose (acb_ptr res, acb_srcptr poly1, long len1, acb_srcptr poly2, long len2, long prec)
void acb_poly-compose (acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, 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 _acb_poly-compose-series_horner (acb_ptr res, acb_srcptr poly1, long len1, acb_srcptr poly2, long n, long prec)
void acb_poly-compose-series_horner (acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, long n, long prec)
void _acb_poly-compose-series_brent_kung (acb_ptr res, acb_srcptr poly1, long len1, acb_srcptr poly2, long len2, long n, long prec)
void acb_poly-compose-series_brent_kung (acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, long n, long prec)
void _acb_poly-compose-series (acb_ptr res, acb_srcptr poly1, long len1, acb_srcptr poly2, long len2, long n, long prec)
void acb_poly-compose-series (acb_poly_t res, const acb_poly_t poly1, const acb_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 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.

void _acb_poly-revert-series-lagrange (acb_ptr h, acb_srcptr f, long flen, long n, long prec)
void acb_poly-revert-series-lagrange (acb_poly_t h, const acb_poly_t f, long n, long prec)
void _acb_poly-revert-series-newton (acb_ptr h, acb_srcptr f, long flen, long n, long prec)
void acb_poly-revert-series-newton (acb_poly_t h, const acb_poly_t f, long n, long prec)
void _acb_poly-revert-series-lagrange-fast (acb_ptr h, acb_srcptr f, long flen, long n, long prec)
void acb_poly-revert-series-lagrange-fast (acb_poly_t h, const acb_poly_t f, long n, long prec)
void _acb_poly-revert-series (acb_ptr h, acb_srcptr f, long flen, long n, long prec)
void acb_poly-revert-series (acb_poly_t h, const acb_poly_t f, long n, long prec)

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

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

void _acb_poly_evaluate_horner (acb_t y, acb_srcptr f, long len, const acb_t x, long prec)
void acb_poly_evaluate_horner (acb_t y, const acb_poly_t f, const acb_t x, long prec)
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.

2.8.11 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 structured 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_treeAlloc().

2.8.12 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.

### 2.8.13 Interpolation

void _acb_poly_interpolate_newton (acb_ptr poly, acb_srcptr xs, acb_srcptr ys, long n, long prec)

void acb_poly_interpolate_newton (acb_poly_t 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)

void acb_poly_interpolate_barycentric (acb_poly_t 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.

void _acb_poly_interpolation_weights (acb_ptr w, acb_ptr * tree, long len, long prec)

void _acb_poly_interpolate_fast_precomp (acb_ptr poly, acb_ptr ys, acb_ptr * tree, acb_srcptr weights, long len, long prec)

void _acb_poly_interpolate_fast (acb_ptr poly, acb_srcptr xs, acb_srcptr ys, long len, long prec)

void acb_poly_interpolate_fast (acb_poly_t 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, 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.

### 2.8.14 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 \( poly \).

### 2.8.15 Elementary functions

void _acb_poly_pow_ui_trunc_binexp (acb_ptr res, acb_srcptr f, longflen, 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 \texttt{acb\_poly\_pow\_ui\_trunc\_binexp}(\texttt{acb\_poly\_t res, const acb\_poly\_t poly, ulong exp, long len, long prec})

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

void \texttt{acb\_poly\_pow\_ui}(\texttt{acb\_ptr res, acb\_srcptr f, long flen, ulong exp, long prec})

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

void \texttt{acb\_poly\_pow\_ui}(\texttt{acb\_poly\_t res, const acb\_poly\_t poly, ulong exp, long prec})

Sets \texttt{res} to \texttt{poly} raised to the power \texttt{exp}.

void \texttt{acb\_poly\_sqrt\_series}(\texttt{acb\_ptr g, acb\_srcptr h, long hlen, long n, long prec})

Sets \texttt{g} to the power series square root of \texttt{h}, truncated to length \texttt{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 \texttt{hlen} and \texttt{n} are greater than zero.

void \texttt{acb\_poly\_rsqrt\_series}(\texttt{acb\_ptr g, acb\_srcptr h, long hlen, long n, long prec})

Sets \texttt{g} to the reciprocal power series square root of \texttt{h}, truncated to length \texttt{n}. Uses division-free Newton iteration.

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

void \texttt{acb\_poly\_log\_series}(\texttt{acb\_ptr res, acb\_srcptr f, long flen, long n, long prec})

void \texttt{acb\_poly\_log\_series}(\texttt{acb\_poly\_t res, const acb\_poly\_t f, long n, long prec})

Sets \texttt{res} to the power series logarithm of \texttt{f}, truncated to length \texttt{n}. Uses the formula \( \log(f(x)) = \int f'(x)/f(x)dx \), adding the logarithm of the constant term in \texttt{f} as the constant of integration.

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

void \texttt{acb\_poly\_atan\_series}(\texttt{acb\_ptr res, acb\_srcptr f, long flen, long n, long prec})

void \texttt{acb\_poly\_atan\_series}(\texttt{acb\_poly\_t res, const acb\_poly\_t f, long n, long prec})

Sets \texttt{res} the power series inverse tangent of \texttt{f}, truncated to length \texttt{n}.

Uses the formula

\[
\tan^{-1}(f(x)) = \int f'(x)/(1 + f(x)^2)dx,
\]

adding the function of the constant term in \texttt{f} as the constant of integration.

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

void \texttt{acb\_poly\_exp\_series\_basecase}(\texttt{acb\_ptr f, acb\_srcptr h, long hlen, long n, long prec})

void \texttt{acb\_poly\_exp\_series\_basecase}(\texttt{acb\_poly\_t f, const acb\_poly\_t h, long n, long prec})

void \texttt{acb\_poly\_exp\_series}(\texttt{acb\_ptr f, acb\_srcptr h, long hlen, long n, long prec})

void \texttt{acb\_poly\_exp\_series}(\texttt{acb\_poly\_t f, const acb\_poly\_t h, long n, long prec})

Sets \texttt{f} to the power series exponential of \texttt{h}, truncated to length \texttt{n}.

The basecase version uses a simple recurrence for the coefficients, requiring \( O(nm) \) operations where \( m \) is the length of \texttt{h}.
The main implementation uses Newton iteration, starting from a small number of terms given by the basecase algorithm. The complexity is $O(M(n))$. Redundant operations in the Newton iteration are avoided by using the scheme described in [HZ2004].

The underscore methods support aliasing and allow the input to be shorter than the output, but require the lengths to be nonzero.

```c
void _acb_poly_sin_cos_series_basecase (acb_ptr s, acb_ptr c, acb_srcptr h, 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)
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)
void _acb_poly_tan_series (acb_ptr g, acb_srcptr h, long hlen, long len, long prec)
void acb_poly_tan_series (acb_poly_t g, 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 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 _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)
void _acb_poly_tan_series (acb_ptr g, acb_srcptr h, long hlen, long len, long prec)
void acb_poly_tan_series (acb_poly_t g, 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.

The underscore version does not support aliasing, and requires the lengths to be nonzero.

2.8.16 Gamma function

```c
void _acb_poly_gamma_series (acb_ptr res, acb_srcptr h, long hlen, long n, long prec)
void acb_poly_gamma_series (acb_poly_t res, const acb_poly_t h, long n, long prec)
```
void _acb_poly_rgamma_series (acb_ptr res, acb_srcptr h, long hlen, long n, long prec)
void acb_poly_rgamma_series (acb_poly_t res, const acb_poly_t h, long n, long prec)
void _acb_poly_lgamma_series (acb_ptr res, acb_srcptr h, long hlen, long n, long prec)
void acb_poly_lgamma_series (acb_poly_t res, const acb_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 _acb_poly_compose_series(). The Taylor coefficients are generated using Stirling’s series.

The underscore methods support aliasing of the input and output arrays, and require that hlen and n are greater than zero.

void _acb_poly_rising_ui_series (acb_ptr res, acb_srcptr f, long flen, ulong r, long trunc, long prec)
void acb_poly_rising_ui_series (acb_poly_t res, const acb_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.

2.8.17 Power sums

void _acb_poly_powsum_series_naive (acb_ptr z, const acb_t s, const acb_t a, const acb_t q, long n, long len, long prec)
void _acb_poly_powsum_series_naive_threaded (acb_ptr z, const acb_t s, const acb_t a, const acb_t q, long n, long len, long prec)

Computes

$$z = S(s, a, n) = \sum_{k=0}^{n-1} q^k \frac{1}{(k + a)^{s+t}}$$

as a power series in t truncated to length len. This function evaluates the sum naively term by term. The threaded version splits the computation over the number of threads returned by flint_get_num_threads().

void _acb_poly_powsum_one_series_sieved (acb_ptr z, const acb_t s, long n, long len, long prec)

Computes

$$z = S(s, 1, n) \sum_{k=1}^{n} \frac{1}{k^{s+t}}$$

as a power series in t truncated to length len. This function stores a table of powers that have already been calculated, computing $(ij)^r$ as $i^r j^r$ whenever $k = ij$ is composite. As a further optimization, it groups all even k and evaluates the sum as a polynomial in $2^{-(s+t)}$. This scheme requires about $n/\log n$ powers, $n/2$ multiplications, and temporary storage of $n/6$ power series. Due to the extra power series multiplications, it is only faster than the naive algorithm when len is small.

2.8.18 Zeta function

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.

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)
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].

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)
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.

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 wp)
Evaluates the truncated Euler-Maclaurin sum of order \( N, M \) for the length-\( d \) truncated Taylor series of the Hurwitz zeta function \( \zeta(s, a) \) at \( s \), using a working precision of \( \text{prec} \) bits. With \( a = 1 \), this gives the usual Riemann zeta function.

If \( \text{deflate} \) is nonzero, \( \zeta(s, a) - 1/(s-1) \) is evaluated (which permits series expansion at \( s = 1 \)).

void _acb_poly_zeta_cpx_series (acb_ptr z, const acb_t s, const acb_t a, int deflate, 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 \( 2^{-\text{prec}} \).

void _acb_poly_zeta_series (acb_ptr res, acb_srcptr h, long hlen, const acb_t a, int deflate, long len, long prec)
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.

### 2.8.19 Polylogarithms

void _acb_poly_polylog_cpx_small (acb_ptr w, const acb_t s, const acb_t z, long len, long prec)
void _acb_poly_polylog_cpx_zeta (acb_ptr w, const acb_t s, const acb_t z, long len, long prec)
void _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 _acb_poly_polylog_series (acb_ptr w, acb_srcptr s, long slen, const acb_t z, long len, long prec)
Arb Documentation, Release 2.4.0

void \texttt{acb\_poly\_polylog\_series} (\texttt{acb\_poly} \ w, \texttt{const acb\_poly} \ s, \texttt{const acb} \ z, \texttt{long} \ len, \texttt{long} \ prec)

Sets \( w \) to the polylogarithm \( \text{Li}_s(z) \) where \( s \) is a given power series, truncating the output to length \( len \). The underscore method requires all lengths to be positive and supports aliasing between all inputs and outputs.

2.8.20 Root-finding

void \_\texttt{acb\_poly\_root\_inclusion} (\texttt{acb} \ r, \texttt{const acb} \ m, \texttt{acb\_srcptr} \ poly, \texttt{acb\_srcptr} \ polyder, \texttt{long} \ len, \texttt{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

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

which is a contradiction (see [Kob2010]).

long \_\texttt{acb\_poly\_validate\_roots} (\texttt{acb\_ptr} \ roots, \texttt{acb\_srcptr} \ poly, \texttt{long} \ len, \texttt{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 \_\texttt{acb\_poly\_refine\_roots\_durand\_kerner} (\texttt{acb\_ptr} \ roots, \texttt{acb\_srcptr} \ poly, \texttt{long} \ len, \texttt{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 \_\texttt{acb\_poly\_find\_roots} (\texttt{acb\_ptr} \ roots, \texttt{acb\_srcptr} \ poly, \texttt{acb\_srcptr} \ initial, \texttt{long} \ len, \texttt{long} \ maxiter, \texttt{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 \texttt{initial}. If \texttt{initial} is set to \texttt{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.
2.9 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.

2.9.1 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.

2.9.2 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.

2.9.3 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.

2.9.4 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.
2.9.5 Comparisons

```c
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.
```

```c
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.
```

```c
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.
```

2.9.6 Special matrices

```c
void acb_mat_zero(acb_mat_t mat)
    Sets all entries in mat to zero.
```

```c
void acb_mat_one(acb_mat_t mat)
    Sets the entries on the main diagonal to ones, and all other entries to zero.
```

2.9.7 Norms

```c
void acb_mat_bound_inf_norm(mag_t b, const acb_mat_t A)
    Sets b to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of A.
```

2.9.8 Arithmetic

```c
void acb_mat_neg(acb_mat_t dest, const acb_mat_t src)
    Sets dest to the exact negation of src. The operands must have the same dimensions.
```

```c
void 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.
```

```c
void 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.
```

```c
void 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.
```

```c
void 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.
```

2.9.9 Scalar arithmetic

```c
void acb_mat_scalar_mul_2exp_si(acb_mat_t B, const acb_mat_t A, long c)
    Sets B to A multiplied by $2^c$.
```

```c
void acb_mat_scalar_addmul_si(acb_mat_t B, const acb_mat_t A, long c, long prec)
```

```c
void acb_mat_scalar_addmul_fmpz(acb_mat_t B, const acb_mat_t A, const fmpz_t c, long prec)
```
**2.9.10 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.

2.9.11 Special functions

```c
void acb_mat_exp(acb_mat_t B, const acb_mat_t A, long prec)
```
Sets $B$ to the exponential of the matrix $A$, defined by the Taylor series

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

The function is evaluated as $\exp(A/2^r)^2$, where $r$ is chosen to give rapid convergence of the Taylor series. 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 $2c^N/N!$.

2.10 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).

2.10.1 Types, macros and constants

```c
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.

2.10.2 Bounds

```c
void 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

$$C(x, r) = \frac{1}{2\pi r} \oint_{|z-x|=r} |f(z)|dz = \int_0^1 |f(x + re^{2\pi it})|dt$$

where $f$ is specified by `(func, param)` and $r$ is given by `radius`. The integral is computed using a simple step sum. The integration range is subdivided until the order of magnitude of $b$ can be determined (i.e. its error bound is smaller than its midpoint), or until the step length has been cut in half `maxdepth` times. This function is currently implemented completely naively, and repeatedly subdivides the whole integration range instead of performing adaptive subdivisions.
2.10.3 Integration

```c
int acb_calc_integrate_taylor(acb_t res, acb_calc_func_t func, void * param, const acb_t a, const acb_t b, const arf_t inner_radius, const arf_t outer_radius, 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

\[ f(m + x) = \sum_{n=0}^{\infty} a_n x^n, \]

then

\[ \int f(m + x) = \left(\sum_{n=0}^{N-1} a_n \frac{x^{n+1}}{n+1}\right) + \left(\sum_{n=N}^{\infty} a_n \frac{x^{n+1}}{n+1}\right). \]

For sufficiently small \( x \), the second series converges and its absolute value is bounded by

\[ \sum_{n=N}^{\infty} \frac{C(m, R)}{R^n} \frac{|x|^{n+1}}{N + 1} = \frac{C(m, R)Rx}{(R-x)(N + 1)} \left(\frac{x}{R}\right)^N. \]

It is required that any singularities of \( f \) are isolated from the path of integration by a distance strictly greater than the positive value \( \text{outer\_radius} \) (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.

2.11 acb_hypgeom.h – hypergeometric functions in the complex numbers

The generalized hypergeometric function is formally defined by

\[ _pF_q(a_1, \ldots, a_p; b_1, \ldots, b_q; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \cdots (a_p)_k}{(b_1)_k \cdots (b_q)_k} \frac{z^k}{k!}. \]

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.
2.11.1 Convergent series

In this section, we define

\[
T(k) = \frac{\prod_{i=0}^{p-1}(a_i)_k}{\prod_{i=0}^{q-1}(b_i)_k} z^k
\]

and

\[
\pFq{(a_0, \ldots, a_{p-1}; b_0 \ldots b_{q-1}; z)}{p+1}{(a_0, \ldots, a_{p-1}; 1; b_0 \ldots b_{q-1}; z)} = \sum_{k=0}^{\infty} T(k)
\]

For the conventional generalized hypergeometric function \(p \cdot F_q\), compute \(p \cdot H_{q+1}\) 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} (mag_t \(C\), acb_srcptr \(a\), long \(p\), acb_srcptr \(b\), long \(q\), const acb_t \(z\), 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

\[
\left| \frac{a + k}{b + k} \right| = \left| 1 + \frac{a - b}{b + k} \right| \leq 1 + \left| \frac{a - b}{b + n} \right|
\]

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} (acb_srcptr \(a\), long \(p\), acb_srcptr \(b\), long \(q\), const acb_t \(z\), long \(prec\))

Heuristically attempts to choose a number of terms \(n\) to sum of a hypergeometric series at a working precision of \(prec\) bits.

Uses double precision arithmetic internally. As currently implemented, it can fail to produce a good result if the parameters are extremely large or extremely close to nonpositive integers.

Numerical cancellation is assumed to be significant, so truncation is done when the current term is \(prec\) bits smaller than the largest encountered term.

This function will also attempt to pick a reasonable truncation point for divergent series.

void \texttt{acb\hypgeom\_pfq\_sum\_forward} (acb_t \(s\), acb_t \(t\), acb_srcptr \(a\), long \(p\), acb_srcptr \(b\), long \(q\), const acb_t \(z\), long \(n\), long \(prec\))

void \texttt{acb\hypgeom\_pfq\_sum\_rs} (acb_t \(s\), acb_t \(t\), acb_srcptr \(a\), long \(p\), acb_srcptr \(b\), long \(q\), const acb_t \(z\), long \(n\), long \(prec\))

void \texttt{acb\hypgeom\_pfq\_sum} (acb_t \(s\), acb_t \(t\), acb_srcptr \(a\), long \(p\), acb_srcptr \(b\), long \(q\), const acb_t \(z\), long \(n\), 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 forward version computes the sum using forward recurrence.

The 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.

```c
void acb_hypgeom_pfq_direct(acb_t res, acb_srcptr a, long p, acb_srcptr b, long q, const acb_t z, long n, 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 `acb_hypgeom_pfq_choose_n()`.

### 2.11.2 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

\[
\frac{1}{\Gamma(b)} \binom{-a}{b} = \frac{(-z)^{-a}}{\Gamma(b-a)} U^*(a, b, z) + \frac{z^{a-b} e^z}{\Gamma(a)} U^*(b - a, b, -z)
\]

which is equivalent to DLMF 13.2.41 (but simpler in form).

We have the asymptotic expansion

\[
U^*(a, b, z) \sim {}_2F_0(a, a - b + 1, -1/z)
\]

where \( {}_2F_0(a, b, z) \) denotes a formal hypergeometric series, i.e.

\[
U^*(a, b, z) = \sum_{k=0}^{n-1} \frac{(a)_k (a - b + 1)_k}{k! (-z)^k} + \varepsilon_n(z).
\]

The error term \( \varepsilon_n(z) \) is bounded according to DLMF 13.7. A case distinction is made depending on whether \( z \) lies in one of three regions which we index by \( R \). Our formula for the error bound increases with the value of \( R \), so we can always choose the larger out of two indices if \( z \) lies in the union of two regions.

Let \( r = |b - 2a| \). If \( \text{Re}(z) \geq r \), set \( R = 1 \). Otherwise, if \( \text{Im}(z) \geq r \) or \( \text{Re}(z) \geq 0 \) and \( |z| \geq r \), set \( R = 2 \). Otherwise, if \( |z| \geq 2r \), set \( R = 3 \). Otherwise, the bound is infinite. If the bound is finite, we have

\[
|\varepsilon_n(z)| \leq 2\alpha C_n \left| \frac{(a)_n (a - b + 1)_n}{n! z^n} \right| \exp(2\alpha \rho C_1 / |z|)
\]
in terms of the following auxiliary quantities

\[
\sigma = |(b - 2a)/z|
\]

\[C_n = \begin{cases} 1 & \text{if } R = 1 \\ \chi(n) & \text{if } R = 2 \\ (\chi(n) + \rho \nu^2 n) \nu^n & \text{if } R = 3 \end{cases}
\]

\[
\nu = \left(\frac{1}{2} + \frac{1}{2} \sqrt{1 - 4\sigma^2}\right)^{-1/2} \leq 1 + 2\sigma^2
\]

\[
\chi(n) = \sqrt{\pi} \Gamma\left(\frac{1}{2} n + 1\right) / \Gamma\left(\frac{1}{2} n + \frac{1}{2}\right)
\]

\[
\sigma' = \begin{cases} \sigma & \text{if } R \neq 3 \\ \nu \sigma & \text{if } R = 3 \end{cases}
\]

\[
\alpha = (1 - \sigma')^{-1}
\]

\[\rho = \frac{1}{2} \left[ 2a^2 - 2ab + b \right] + \sigma'(1 + \frac{1}{2} \sigma')(1 - \sigma')^{-2}
\]

void \texttt{acb\_hypgeom\_u\_asymp}(acb\_t res, const acb\_t a, const acb\_t b, const acb\_t z, long n, long prec)

Sets \texttt{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 \(n \geq 0\).

### 2.11.3 The error function

void \texttt{acb\_hypgeom\_erf\_1f1a}(acb\_t res, const acb\_t z, long prec)

void \texttt{acb\_hypgeom\_erf\_1f1b}(acb\_t res, const acb\_t z, long prec)

void \texttt{acb\_hypgeom\_erf\_asymp}(acb\_t res, const acb\_t z, long prec, long prec2)

void \texttt{acb\_hypgeom\_erf}(acb\_t res, const acb\_t z, long prec)

Computes the error function respectively using

\[
\text{erf}(z) = \frac{2z}{\sqrt{\pi}} \, _1F_1\left(\frac{1}{2}, \frac{3}{2}, -z^2\right)
\]

\[
\text{erf}(z) = \frac{2ze^{-z^2}}{\sqrt{\pi}} \, _1F_1\left(1, \frac{3}{2}, z^2\right)
\]

\[
\text{erf}(z) = \frac{z}{\sqrt{\pi} z^2} \left(1 - \frac{e^{-z^2}}{\sqrt{\pi}} U\left(\frac{1}{2}, \frac{1}{2}, z^2\right)\right).
\]

and an automatic algorithm choice. The \texttt{asymp} version takes a second precision to use for the \(U\) term.

### 2.11.4 Bessel functions

void \texttt{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()}. For all complex \(\nu, z\), we have

\[
J_\nu(z) = \frac{z^\nu}{2^\nu \pi^\nu \Gamma(\nu + 1)} \, _1F_1\left(\nu + \frac{1}{2}, 2\nu + 1, 2iz\right) = A_+ B_+ + A_- B_-
\]

where

\[
A_\pm = z^\nu (z^2 - 1)^{-\nu/2} (\mp iz)^{1/2} \nu(2\pi)^{-1/2} = (\pm iz)^{-1/2 - \nu} z^\nu (2\pi)^{-1/2}
\]

\[
B_\pm = e^{\pm iz} U^*\left(\nu + \frac{1}{2}, 2\nu + 1, \mp 2iz\right).
\]
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 $A_{\pm} = \exp(\pm i(2\nu + 1)/4\pi)(2\pi z)^{-1/2}$.

void \textbf{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

$$J_{\nu}(z) = \frac{1}{\Gamma(\nu + 1)} \left(\frac{z}{2}\right)^{\nu} \binom{\nu}{0} F_1 \left(\nu + 1, -\frac{z^2}{4}\right).$$

void \textbf{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.

2.12 \texttt{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, $\text{tau}$ or $\tau$ always denotes an element of the complex upper half-plane $\mathbb{H} = \{ z \in \mathbb{C} : \text{Im}(z) > 0 \}$. We also often use the variable $q$, variously defined as $q = e^{2\pi i \tau}$ (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 $\text{tau}$ is represented by an \texttt{acb\_t} whose content overlaps with the real line (or lies in the lower half-plane), and $\text{tau}$ 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 $|q| < 1$.

2.12.1 The modular group

\texttt{psl2z\_struct}

\texttt{psl2z\_t}
Represents an element of the modular group $\text{PSL}(2, \mathbb{Z})$, namely an integer matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

with $ad - bc = 1$, and with signs canonicalized such that $c \geq 0$, and $d > 0$ if $c = 0$. The struct members $a$, $b$, $c$, $d$ are of type \texttt{fmpz}.

void \textbf{psl2z\_init} (psl2z\_t g)
Initializes $g$ and set it to the identity element.

void \textbf{psl2z\_clear} (psl2z\_t g)
Clears $g$.

void \textbf{psl2z\_swap} (psl2z\_t f, psl2z\_t g)
Swaps $f$ and $g$ efficiently.

void \textbf{psl2z\_set} (psl2z\_t f, const psl2z\_t g)
Sets $f$ to a copy of $g$.

void \textbf{psl2z\_one} (psl2z\_t g)
Sets $g$ to the identity element.

int \textbf{psl2z\_is\_one} (const psl2z\_t g)
Returns nonzero iff $g$ is the identity element.
void **psl2z_print** (const psl2z_t g)

Prints \( g \) to standard output.

int **psl2z_equal** (const psl2z_t f, const psl2z_t g)

Returns nonzero iff \( f \) and \( g \) are equal.

void **psl2z_mul** (psl2z_t h, const psl2z_t f, const psl2z_t g)

Sets \( h \) to the product of \( f \) and \( g \), namely the matrix product with the signs canonicalized.

void **psl2z_inv** (psl2z_t h, const psl2z_t g)

Sets \( h \) to the inverse of \( g \).

int **psl2z_is_correct** (const psl2z_t g)

Returns nonzero iff \( g \) contains correct data, i.e. satisfying \( ad - bc = 1 \), \( c \geq 0 \), and \( d > 0 \) if \( c = 0 \).

void **psl2z_randtest** (psl2z_t g, flint_rand_t state, long bits)

Sets \( g \) to a random element of \( \text{PSL}(2, \mathbb{Z}) \) with entries of bit length at most \( \text{bits} \) (or 1, if \( \text{bits} \) is not positive). We first generate \( a \) and \( d \), compute their Bezout coefficients, divide by the GCD, and then correct the signs.

### 2.12.2 Modular transformations

void **acb_modular_transform** (acb_t w, const psl2z_t g, const acb_t z, long prec)

Applies the modular transformation \( g \) to the complex number \( z \), evaluating

\[
 w = gz = \frac{az + b}{cz + d}. 
\]

void **acb_modular_fundamental_domain_approx_d** (psl2z_t g, double x, double y, double one_minus_eps)

Attempts to determine a modular transformation \( g \) that maps the complex number \( x + yi \) to the fundamental domain or just slightly outside the fundamental domain, where the target tolerance (not a strict bound) is specified by \( \text{one\_minus\_eps} \).

The inputs are assumed to be finite numbers, with \( y \) positive.

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.

void **acb_modular_fundamental_domain_approx_arf** (psl2z_t g, const arf_t x, const arf_t y, 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 \( \text{one\_minus\_eps} \). It also computes the transformed value \( w = gz \).

This function first tries to use **acb_modular_fundamental_domain_approx_d**() and checks if the result is acceptable. If this fails, it calls **acb_modular_fundamental_domain_approx_arf**() with higher precision. Finally, \( w = gz \) is evaluated by a single application of \( g \).

The algorithm can fail to produce a satisfactory transformation. The output \( g \) is always set to some correct modular transformation, but it is up to the user to verify a posteriori that \( w \) is close enough to the fundamental domain.
integer \texttt{acb_modular_is_in_fundamental_domain} (const \texttt{acb_t} \( z \), const \texttt{arf_t} \( tol \), long \texttt{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.

### 2.12.3 Jacobi theta functions

Unfortunately, there are many inconsistent notational variations for Jacobi theta functions in the literature. Unless otherwise noted, we use the functions

\[
\begin{align*}
\theta_1(z, \tau) &= -i \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i(n+1/2)^2 \tau + (2n+1)z) = 2q^{1/4} \sum_{n=0}^{\infty} (-1)^n q^{n(n+1)} \sin((2n+1)\pi z) \\
\theta_2(z, \tau) &= \sum_{n=-\infty}^{\infty} \exp(\pi i(n+1/2)^2 \tau + (2n+1)z) = 2q^{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos((2n+1)\pi z) \\
\theta_3(z, \tau) &= \sum_{n=-\infty}^{\infty} \exp(\pi i n^2 \tau + 2nz) = 1 + 2 \sum_{n=1}^{\infty} q^n \cos(2\pi nz) \\
\theta_4(z, \tau) &= \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i n^2 \tau + 2nz) = 1 + 2 \sum_{n=1}^{\infty} (-1)^n q^{2n} \cos(2\pi nz)
\end{align*}
\]

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.

\[\text{void \texttt{acb_modular_theta_transform} (int } \ast \text{ R, int } \ast \text{ S, int } \ast \text{ C, const psl2z_t } \ast \text{ g})\]

We wish to write a theta function with quasiperiod \( \tau \) in terms of a theta function with quasiperiod \( \tau' = g \tau \), given some \( g = (a, b; c, d) \in \text{PSL}(2, \mathbb{Z}) \). For \( i = 0, 1, 2, 3 \), this function computes integers \( R_i \) and \( S_i \), \( R \) and \( S \) should be arrays of length 4 and \( C \in \{0, 1\} \) such that

\[
\theta_{1+i}(z, \tau) = \exp(\pi i R_i/4) \cdot A \cdot B \cdot \theta_{1+S_i}(z', \tau')
\]

where \( z' = z \), \( A = B = 1 \) if \( C = 0 \), and

\[
z' = \frac{-z}{c \tau + d}, \quad A = \sqrt{\frac{i}{c \tau + d}}, \quad B = \exp\left(-\pi i c \frac{z^2}{c \tau + d}\right)
\]

if \( C = 1 \). Note that \( A \) is well-defined with the principal branch of the square root since \( A^2 = i/(c \tau + d) \) lies in the right half-plane.

Firstly, if \( c = 0 \), we have \( \theta_i(z, \tau) = \exp(-\pi i b/4) \theta_i(z, \tau + b) \) for \( i = 1, 2, \) whereas \( \theta_3 \) 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 \( q \) 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

\[
\begin{align*}
\theta_1(z, \tau) &= \varepsilon_1 AB \theta_1(z', \tau') \\
\theta_2(z, \tau) &= \varepsilon_2 AB \theta_{1-c,1+a}(z', \tau') \\
\theta_3(z, \tau) &= \varepsilon_3 AB \theta_{1+d-c,1-b+a}(z', \tau') \\
\theta_4(z, \tau) &= \varepsilon_4 AB \theta_{1+d,1-b}(z', \tau')
\end{align*}
\]
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 \texttt{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)(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 \texttt{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 \texttt{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 $w = \exp(\pi iz)$ 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 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 $[(k + 2)/4] = 1, 2, 4, 6, 9$ etc. and the powers of $w$ are $\pm (k + 2) = \pm 2, \pm 3, \pm 4, \ldots$ etc. The scheme is illustrated by the following table:

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\theta_1, \theta_2$</th>
<th>$q^k$</th>
<th>$(w^k \pm w^{-k})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\theta_3, \theta_4$</td>
<td>$q^0$</td>
<td>$(w^0 \pm w^{-0})$</td>
</tr>
<tr>
<td>1</td>
<td>$\theta_1, \theta_2$</td>
<td>$q^1$</td>
<td>$(w^1 \pm w^{-1})$</td>
</tr>
<tr>
<td>2</td>
<td>$\theta_3, \theta_4$</td>
<td>$q^2$</td>
<td>$(w^2 \pm w^{-2})$</td>
</tr>
<tr>
<td>3</td>
<td>$\theta_1, \theta_2$</td>
<td>$q^3$</td>
<td>$(w^3 \pm w^{-3})$</td>
</tr>
<tr>
<td>4</td>
<td>$\theta_3, \theta_4$</td>
<td>$q^4$</td>
<td>$(w^4 \pm w^{-4})$</td>
</tr>
<tr>
<td>5</td>
<td>$\theta_1, \theta_2$</td>
<td>$q^5$</td>
<td>$(w^5 \pm w^{-5})$</td>
</tr>
<tr>
<td>6</td>
<td>$\theta_3, \theta_4$</td>
<td>$q^6$</td>
<td>$(w^6 \pm w^{-6})$</td>
</tr>
<tr>
<td>7</td>
<td>$\theta_1, \theta_2$</td>
<td>$q^7$</td>
<td>$(w^7 \pm w^{-7})$</td>
</tr>
</tbody>
</table>

For some integer $N \geq 1$, the summation is stopped just before term $k = N$. Let $Q = |q|$, $W = \max(|w|, |w^{-1}|)$, $E = [(N + 2)/4]$ and $F = [(N + 1)/2] + 1$. The error of the zeroth derivative can be bounded as

$$2Q^E W^{N+2} \left[ 1 + Q^F W + Q^{2F} W^2 + \ldots \right] = \frac{2Q^E W^{N+2}}{1 - Q^F W}$$
provided that the denominator is positive (otherwise we set the error bound to infinity). When \( \text{len} \) is greater than 1, consider the derivative of order \( r \). The term of index \( k \) and order \( r \) picks up a factor of magnitude \( (k+2)^r \) from differentiation of \( w^{k+2} \) (it also picks up a factor \( \pi^r \), but we omit this until we rescale the coefficients at the end of the computation). Thus we have the error bound

\[
2Q^nW^N + 2(N + 2)^r \left[ 1 + Q^nW \frac{(N + 3)^r}{(N + 2)^r} + Q^nW^2 \frac{(N + 4)^r}{(N + 2)^r} + \ldots \right]
\]

which by the inequality \((1 + m/(N + 2))^r \leq \exp(mr/(N + 2))\) can be bounded as

\[
\frac{2Q^nW^N + 2(N + 2)^r}{1 - Q^nW \exp(r/(N + 2))} \cdot \exp(2w^r - 1)
\]

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().

### 2.12.4 The Dedekind eta function

**void acb_modular_addseq_eta** (long * exponents, long * aindex, long * bindex, long num)

Constructs an addition sequence for the first \( \text{num} \) generalized pentagonal numbers (excluding zero), i.e. 1, 2, 5, 7, 12, 15, 22, 26, 35, 40 etc.

**void acb_modular_eta_sum** (acb_t eta, const acb_t q, long prec)

Evaluates the Dedekind eta function without the leading 24th root, i.e.

\[
\exp(-\pi^r/12)\eta(\tau) = \sum_{n=-\infty}^{\infty} (-1)^n q^{(3n^2 - n)/2}
\]

given \( q = \exp(2\pi^r) \), by summing the defining series.

This function is intended for \(|q| \ll 1\). It can be called with any \( q \), but will return useless intervals if convergence is not rapid. For general evaluation of the eta function, the user should only call this function after applying a suitable modular transformation.

**int acb_modular_epsilon_arg** (const psl2z_t g)

Given \( g = (a, b; c, d) \), computes an integer \( R \) such that \( \varepsilon(a, b, c, d) = \exp(\pi i R/12) \) is the 24th root of unity in the transformation formula for the Dedekind eta function,

\[
\eta\left(\frac{a \tau + b}{c \tau + d}\right) = \varepsilon(a, b, c, d) \sqrt{c \tau + d} \eta(\tau).
\]

**void acb_modular_eta** (acb_t r, const acb_t tau, 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 acb_modular_eta_sum().
2.12.5 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 \( j(\tau) = 32(\theta_2^8 + \theta_3^8 + \theta_4^8)^3 / (\theta_2 \theta_3 \theta_4)^8 \) where \( \theta_i = \theta_i(0, \tau) \).

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 formal power series
\[
\Delta(\tau) = \eta(\tau)^{24},
\]
where \( \Delta(\tau) \) transforms as
\[
\Delta\left(\frac{a\tau + b}{c\tau + d}\right) = (c\tau + d)^{12}\Delta(\tau).
\]
The modular discriminant is sometimes defined with an extra factor \((2\pi)^{12}\), which we omit in this implementation.

void \texttt{acb_modular\_eisenstein (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
\[
G_{2k}\left(\frac{a\tau + b}{c\tau + d}\right) = (c\tau + d)^{2k}G_{2k}(\tau).
\]
We first evaluate \( G_4(\tau) \) and \( G_6(\tau) \) on the fundamental domain using theta functions, and then compute the Eisenstein series of higher index using a recurrence relation.

2.12.6 Elliptic functions

void \texttt{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
\[
\wp(z, \tau) = \pi^2 \theta_2^2(0, \tau) \theta_3^2(0, \tau) \frac{\theta_4^2(z, \tau)}{\theta_1^2(z, \tau)} - \frac{\pi^2}{3} \left[ \theta_4^4(0, \tau) + \theta_3^4(0, \tau) \right].
\]

void \texttt{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 \( \tau \).
2.13 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).

2.13.1 Generation of Bernoulli numbers

\texttt{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.

\texttt{void bernoulli\_rev\_init (bernoulli\_rev\_t iter, ulong n)}

Initializes the iterator \textit{iter}. The first Bernoulli number to be generated by calling \texttt{bernoulli\_rev\_next()} is \( B_n \). It is assumed that \( n \) is even.

\texttt{void bernoulli\_rev\_next (fmpz\_t numer, fmpz\_t denom, bernoulli\_rev\_t iter)}

Sets \textit{numer} and \textit{denom} to the exact, reduced numerator and denominator of the Bernoulli number \( B_k \) and advances the state of \textit{iter} so that the next invocation generates \( B_{k-2} \).

\texttt{void bernoulli\_rev\_clear (bernoulli\_rev\_t iter)}

Frees all memory allocated internally by \textit{iter}.

2.13.2 Caching

\texttt{long bernoulli\_cache\_num}

\texttt{fmpq * bernoulli\_cache}

Cache of Bernoulli numbers. Uses thread-local storage if enabled in FLINT.

\texttt{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.

2.13.3 Bounding

\texttt{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 \texttt{LONG\_MIN}. Otherwise, the returned exponent \( b \) is never more than one percent larger than the true magnitude.

This function is intended for use when \( n \) small enough that one might comfortably compute \( B_n \) exactly. It aborts if \( n \) is so large that internal overflow occurs.

```c
void _bernoulli_fmpq_ui_zeta (fmpz_t num, fmpz_t den, ulong n)
```

Sets \( \text{num} \) and \( \text{den} \) to the reduced numerator and denominator of the Bernoulli number \( B_n \).

This function computes the denominator \( d \) using von Staudt-Clausen theorem, numerically approximates \( B_n \) using \texttt{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).

```c
void _bernoulli_fmpq_ui (fmpz_t num, fmpz_t den, ulong n)
```

```c
void bernoulli_fmpq_ui (fmpq_t b, ulong n)
```

Computes the Bernoulli number \( B_n \) as an exact fraction, for an isolated integer \( n \). This function reads \( B_n \) from the global cache if the number is already cached, but does not automatically extend the cache by itself.

### 2.14 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 \texttt{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.

#### 2.14.1 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

\[
P(k) = a_p k^p + a_{p-1} k^{p-1} + \ldots + a_0 \\
Q(k) = b_q k^q + b_{q-1} k^{q-1} + \ldots + b_0.
\]

For convergence, we require \( p < q \), or \( p = q \) with \( |z||a_p| < |b_q| \). We also assume that \( P(k) \) and \( Q(k) \) have no roots among the positive integers (if there are positive integer roots, the sum is either finite or undefined). With these conditions satisfied, our goal is to find a parameter \( n \geq 0 \) such that

\[
\sum_{k=n}^{\infty} T(k)z^k \leq 2^{-d}.
\]
We can rewrite the hypergeometric term ratio as

\[ zR(k) = z \frac{P(k)}{Q(k)} = z \left( \frac{a_p}{b_q} \right) \frac{1}{k^{s-p}} F(k) \]

where

\[ F(k) = \frac{1}{k} \frac{\alpha_1/k + \alpha_2/k^2 + \ldots + \alpha_q/k^p}{\beta_1/k + \beta_2/k^2 + \ldots + \beta_q/k^q} = 1 + O(1/k) \]

and where \( \alpha_i = a_{p-i}/a_p, \beta_i = b_{q-i}/b_q \). Next, we define

\[ C = \max_{1 \leq i \leq p} |\alpha_i|^{1/i}, \quad D = \max_{1 \leq i \leq q} |\beta_i|^{1/i}. \]

Now, if \( k > C \), the magnitude of the numerator of \( F(k) \) is bounded from above by

\[ 1 + \sum_{i=1}^{p} \left( \frac{C}{k} \right)^i \leq 1 + \frac{C}{k-C} \]

and if \( k > 2D \), the magnitude of the denominator of \( F(k) \) is bounded from below by

\[ 1 - \sum_{i=1}^{q} \left( \frac{D}{k} \right)^i \geq 1 + \frac{D}{D-k}. \]

Putting the inequalities together gives the following bound, valid for \( k > K = \max(C, 2D) \):

\[ |F(k)| \leq \frac{k(k-D)}{(k-C)(k-2D)} = \left( 1 + \frac{C}{k-C} \right) \left( 1 + \frac{D}{k-2D} \right). \]

Let \( r = q-p \) and \( \tilde{z} = |z a_p/b_q| \). Assuming \( k > \max(C, 2D, \tilde{z}^{1/r}) \), we have

\[ |zR(k)| \leq G(k) = \frac{\tilde{z} F(k)}{k^r} \]

where \( G(k) \) is monotonically decreasing. Now we just need to find an \( n \) such that \( G(n) < 1 \) and for which \( |T(n)|/(1-G(n)) \leq 2^{-d} \). This can be done by computing a floating-point guess for \( n \) then trying successively larger values.

This strategy leaves room for some improvement. For example, if \( \beta_1 \) is positive and large, the bound \( B \) becomes very pessimistic (a larger positive \( \beta_1 \) causes faster convergence, not slower convergence).

### 2.14.2 Types, macros and constants

**hypgeom_struct**

**hypgeom_t**

Stores polynomials \( A, B, P, Q \) and precomputed bounds, representing a fixed hypergeometric series.

### 2.14.3 Memory management

void **hypgeom_init**(hypgeom_t hyp)

void **hypgeom_clear**(hypgeom_t hyp)
2.14.4 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 \( \text{LONG\_MAX}/2 \).

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

\[
\frac{\tilde{z}}{k!} \approx \frac{k(k - D)}{(k - C)(k - 2D)}.
\]

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 \( \text{error} \) 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 \( T(K) \).

2.14.5 Summation

void fmprb_hypgeom_sum (fmprb_t P, fmprb_t Q, const hypgeom_t hyp, const long n, long prec)
Computes \( P, Q \) such that \( P/Q = \sum_{k=0}^{n-1} T(k) \) where \( T(k) \) is defined by \( \text{hyp} \), using binary splitting and a working precision of \( \text{prec} \) bits.

void fmprb_hypgeom_infsum (fmprb_t P, fmprb_t Q, hypgeom_t hyp, long tol, long prec)
Computes \( P/Q = \sum_{k=0}^{\infty} T(k) \) where \( T(k) \) is defined by \( \text{hyp} \), using binary splitting and working precision of \( \text{prec} \) bits. The number of terms is chosen automatically to bound the truncation error by at most \( 2^{-\text{tol}} \). The bound for the truncation error is included in the output as part of \( P \).

void arb_hypgeom_sum (arb_t P, arb_t Q, const hypgeom_t hyp, const long n, long prec)
Computes \( P, Q \) such that \( P/Q = \sum_{k=0}^{n-1} T(k) \) where \( T(k) \) is defined by \( \text{hyp} \), using binary splitting and a working precision of \( \text{prec} \) bits.

void arb_hypgeom_infsum (arb_t P, arb_t Q, hypgeom_t hyp, long tol, long prec)
Computes \( P/Q = \sum_{k=0}^{\infty} T(k) \) where \( T(k) \) is defined by \( \text{hyp} \), using binary splitting and working precision of \( \text{prec} \) bits. The number of terms is chosen automatically to bound the truncation error by at most \( 2^{-\text{tol}} \). The bound for the truncation error is included in the output as part of \( P \).

2.15 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. $n < 10^6$).

```c
void partitions_rademacher_bound (arf_t b, const fmpz_t n, ulong N)
    Sets $b$ to an upper bound for

    \[
    M(n, N) = \frac{44\pi^2}{225\sqrt{3}} N^{-1/2} + \frac{\pi\sqrt{2}}{75} \left( \frac{N}{n-1} \right)^{1/2} \sinh \left( \frac{\pi}{N} \sqrt{\frac{2n}{3}} \right).
    \]

    This formula gives an upper bound for the truncation error in the Hardy-Ramanujan-Rademacher formula when the series is taken up to the term $t(n, N)$ inclusive.
```

```c
partitions_hrr_sum_arb (arb_t x, const fmpz_t n, long N0, long N, int use_doubles)
    Evaluates the partial sum $\sum_{k=N_0}^{N} t(n, k)$ of the Hardy-Ramanujan-Rademacher series.

    If $use_doubles$ is nonzero, doubles and the system’s standard library math functions are used to evaluate the smallest terms. This significantly speeds up evaluation for small $n$ (e.g. $n < 10^6$), and gives a small speed improvement for larger $n$, but the result is not guaranteed to be correct. In practice, the error is estimated very conservatively, and unless the system’s standard library is broken, use of doubles can be considered safe. Setting $use_doubles$ to zero gives a fully guaranteed bound.
```

```c
void partitions_fmpz_fmpz (fmpz_t p, const fmpz_t n, int use_doubles)
    Computes the partition function $p(n)$ using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing $p(n)$ and verifies that the ball contains a unique integer.

    If $n$ is sufficiently large and a number of threads greater than 1 has been selected with `flint_set_num_threads()`, the computation time will be reduced by using two threads.

    See `partitions_hrr_sum_arb()` for an explanation of the $use_doubles$ option.
```

```c
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.
```

```c
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()`).
```
3.1 Algorithms for mathematical constants

Most mathematical constants are evaluated using the generic hypergeometric summation code.

3.1.1 Pi

\[ \pi = \frac{1}{12} \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!((13591409 + 545140134k)}{(3k)! (k!)^3 640320^{3k+3/2}} \]

which is hypergeometric and adds roughly 14 digits per term. Methods based on the arithmetic-geometric mean seem to be slower by a factor three in practice.

A small trick is to compute \( \frac{1}{\sqrt{640320}} \) instead of \( \sqrt{640320} \) at the end.

3.1.2 Logarithms of integers

We use the formulas

\[ \log(2) = 3 \sum_{k=0}^{\infty} \frac{(-1)^k (k!)^2}{2^k (2k + 1)!} \]

\[ \log(10) = 46 \text{ atanh}(1/31) + 34 \text{ atanh}(1/49) + 20 \text{ atanh}(1/161) \]

3.1.3 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 \( 24 \varepsilon^{-8n} \).

### 3.1.4 Catalan’s constant

Catalan’s constant is computed using the hypergeometric series

\[
C = \sum_{k=0}^{\infty} \frac{(-1)^k 4^{4k+1} (40k^2 + 56k + 19) [(k+1)!]^2 [(2k+2)!]^3}{(k+1)(2k+1)[(4k+4)!]^2}
\]

### 3.1.5 Khinchin’s constant

Khinchin’s constant \( K_0 \) is computed using the formula

\[
\log K_0 = \frac{1}{\log 2} \left[ \sum_{k=2}^{N-1} \log \left( \frac{k-1}{k} \right) \log \left( \frac{k+1}{k} \right) + \sum_{n=1}^{\infty} \frac{\zeta(2n, N) 2^{n-1} (-1)^{k+1}}{n \sum_{k=1}^{2n-1} \left( \frac{1}{k} \right) \right] \]

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

\[
\sum_{n=M+1}^{\infty} \zeta(2n, N) = \sum_{n=M+1}^{\infty} \sum_{k=0}^{\infty} (k+N)^{-2n} \leq \sum_{n=M+1}^{\infty} \left( N^{-2n} + \int_{0}^{\infty} (t+N)^{-2n} dt \right) = \sum_{n=M+1}^{\infty} \frac{1}{N^{2n}} \left( \frac{N+1}{2n-1} \right) \leq \frac{1}{N^{2M}}.
\]

Thus, for an error of at most \( 2^{-p} \) in the series, it is sufficient to choose \( M \geq p/(2 \log_2 N) \).

### 3.1.6 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 \( s = 2 \).

### 3.1.7 Apery’s constant

Apery’s constant \( \zeta(3) \) is computed using the hypergeometric series

\[
\zeta(3) = \frac{1}{64} \sum_{k=0}^{\infty} (-1)^k (205k^2 + 250k + 77) \frac{(k!)^3}{[(2k+1)!]^3}.
\]

### 3.2 Algorithms for elementary functions

(This section is incomplete.)

We typically compute elementary functions using the following steps: reduction to a combination of standard function (exp, log, atan, sin, cos of a real argument), reduction to a standard domain, convergence-accelerating argument reduction (using functional equations and possibly precomputed lookup tables), followed by Taylor series evaluation.
3.2.1 Arctangents

It is sufficient to consider $x \in [0, 1)$, since $\tan(x) = \tan(-x)$ for $x < 0$, $\tan(1) = \pi/4$, and $\tan(x) = \pi/2 - \tan(1/x)$ for $x > 1$.

For sufficiently small $x$, we use the Taylor series

$$\tan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \ldots$$

Applying the argument-halving formula

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

$r$ times gives $x \leq 2^{-r}$.

Applying the formula

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

gives $0 \leq w < 1/q$. At low precision, picking a moderately large $q$ (say $q = 2^8$), and using a lookup table for $\tan(p/q)$, $p = 0, 1, \ldots, q - 1$, is much better than repeated argument-halving. This transformation can be applied repeatedly with a sequence of increasing values of $q$. For example, $(q_1 = 2^4, q_2 = 2^8)$ requires only $2^5$ precomputed table entries, but the evaluation costs one extra division.

At high precision, the $\tan(p/q)$ values can be evaluated using binary splitting. Choosing $q = 2, 4, 8, \ldots$ results in a version of the bit-burst a algorithm.

3.2.2 Error propagation for arctangents

A generic derivative-based error bound is

$$\sup_{\xi \in [-1,1]} |\tan(m) - \tan(m + \xi r)| \leq \frac{r}{1 + \max(0, |m| - r)^2} \leq r.$$ 

An exact representation for the propagated error is given by

$$\sup_{\xi \in [-1,1]} |\tan(m) - \tan(m + \xi r)| = \begin{cases} 
\tan \left( \frac{r}{1 + |m|} \right) & \text{if } r \leq |m| \\
\frac{\pi}{2} - \tan \left( \frac{1 + |m|(|m|-r)}{r} \right) & \text{if } r > |m|.
\end{cases}$$

3.2.3 Logarithms

It is sufficient to consider $\log(1 + x)$ where $x \in [0, 1)$, since $\log(t^{2^n}) = \log(t) + n \log(2)$.

We only use the Taylor series

$$\log(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$$

directly when $x$ is so small that the linear or quadratic term gives full accuracy. In general we use the more efficient series

$$\text{atanh}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \ldots$$

together with the identity $\log(1 + x) = 2 \text{atanh}(x/(2 + x))$. 

3.2. Algorithms for elementary functions 105
Applying the argument-halving formula
\[ \log(1 + x) = 2 \log \left(\sqrt{1 + x}\right) \]
r times gives \( x \leq 2^{-r} \).

Applying the formula
\[ \log(1 + x) = \log(p/q) + \log(1 + w), \quad w = \frac{qx - p}{p + q}, \quad p = \lfloor qx \rfloor \]
gives \( 0 \leq w < 1/q \) (see analogous remarks for the arctangent).

### 3.2.4 Error propagation for logarithms

A generic derivative-based error bound is
\[ \sup_{\xi \in [-1, 1]} |\log(m) - \log(m + \xi r)| \leq \frac{r}{m - r}. \]

An exact representation for the propagated error is given by
\[ \sup_{\xi \in [-1, 1]} |\log(m) - \log(m + \xi r)| = \log(1 + r/(m - r)). \]

Of course, these formulas require \( m > r \geq 0 \) (otherwise, the real logarithm is undefined).

### 3.3 Algorithms for gamma functions

#### 3.3.1 The Stirling series

In general, the gamma function is computed via the Stirling series
\[ \log \Gamma(z) = \left( z - \frac{1}{2} \right) \log z - z + \frac{\ln 2\pi}{2} + \sum_{k=1}^{n-1} \frac{B_{2k}}{2k(2k-1)} z^{2k-1} + R_n(z) \]
where ([Olv1997] pp. 293-295) the remainder term is exactly
\[ R_n(z) = \int_0^\infty \frac{B_{2n} - \tilde{B}_{2n}(x)}{2n(x + z)^{2n}} dx. \]

To evaluate the gamma function of a power series argument, we substitute \( z \to z + t \in \mathbb{C}[[t]] \).

Using the bound for \(|x + z|\) given by [Olv1997] and the fact that the numerator of the integrand is bounded in absolute value by \( 2|B_{2n}| \), the remainder can be shown to satisfy the bound
\[ |[i^k] R_n(z + t)| \leq 2|B_{2n}| \frac{\Gamma(2n + k - 1)}{\Gamma(k + 1)\Gamma(2n + 1)} |z|^k (b/|z|)^{2n+k} \]
where \( b = 1/\cos(\arg(z)/2) \). Note that by trigonometric identities, assuming that \( z = x + yi \), we have \( b = \sqrt{1 + t^2} \)
where
\[ t = \frac{y}{\sqrt{x^2 + y^2}} = \frac{\sqrt{x^2 + y^2} - x}{y}. \]
To use the Stirling series at $p$-bit precision, we select parameters $r, n$ such that the remainder $R(n, z)$ approximately is bounded by $2^{-p}$. If $|z|$ is too small for the Stirling series to give sufficient accuracy directly, we first translate to $z + r$ using the formula $\Gamma(z) = \Gamma(z + r)/(z(z + 1)(z + 2) \cdots (z + r - 1))$.

To obtain a remainder smaller than $2^{-p}$, we must choose an $r$ such that, in the real case, $z + r > \beta p$, where $\beta \approx \log(2)/(2\pi) \approx 0.11$. In practice, a slightly larger factor $\beta \approx 0.2$ more closely balances $n$ and $r$. A much larger $\beta$ (e.g. $\beta = 1$) could be used to reduce the number of Bernoulli numbers that have to be precomputed, at the expense of slower repeated evaluation.

### 3.3.2 Rational arguments

We use efficient methods to compute $y = \Gamma(p/q)$ where $q$ is one of $1, 2, 3, 4, 6$ and $p$ is a small integer. The cases $\Gamma(1) = 1$ and $\Gamma(1/2) = \sqrt{\pi}$ are trivial. We reduce all remaining cases to $\Gamma(1/3)$ or $\Gamma(1/4)$ using the following relations:

\[
\begin{align*}
\Gamma(2/3) &= \frac{2\pi}{3^{3/2}\Gamma(1/3)}, \\
\Gamma(1/6) &= \frac{\Gamma(1/3)^2}{(\pi/3)^{1/2} 2^{1/3}}, \\
\Gamma(5/6) &= \frac{2\pi(\pi/3)^{1/2} 2^{1/3}}{\Gamma(1/3)^2}.
\end{align*}
\]

We compute $\Gamma(1/3)$ and $\Gamma(1/4)$ rapidly to high precision using

\[
\Gamma(1/3) = \left(\frac{12\pi^4}{\sqrt{10}} \sum_{k=0}^{\infty} \frac{(6k)!(-1)^k}{(3k)!3^k 160^k} \right)^{1/6}, \\
\Gamma(1/4) = \left(\frac{(2\pi)^{3/2}}{\operatorname{agm}(1, \sqrt{2})} \right).
\]

An alternative formula which could be used for $\Gamma(1/3)$ is

\[
\Gamma(1/3) = \frac{2^{4/9}\pi^{2/3}}{3^{1/4} \left(\operatorname{agm}\left(1, \frac{1}{2}\sqrt{2 + \sqrt{3}}\right)\right)^{1/3}},
\]

but this appears to be slightly slower in practice.

### 3.4 Algorithms for polylogarithms

The polylogarithm is defined for $s, z \in \mathbb{C}$ with $|z| < 1$ by

\[
\operatorname{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}
\]

and for $|z| \geq 1$ by analytic continuation, except for the singular point $z = 1$.

#### 3.4.1 Computation for small $z$

The power sum converges rapidly when $|z| \ll 1$. To compute the series expansion with respect to $s$, we substitute $s \to s + x \in \mathbb{C}[[x]]$ and obtain

\[
\operatorname{Li}_{s+x}(z) = \sum_{d=0}^{\infty} x^d \frac{(-1)^d}{d!} \sum_{k=1}^{\infty} T(k)
\]
where

\[ T(k) = \frac{z^k \log^d(k)}{k^s}. \]

Let \( U(k) \) be a nonincreasing bound for the magnitude of the term ratio

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

Then the remainder after the \( k = N - 1 \) term is bounded by

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

whenever \( U(N) < 1 \).

If \( s = \sigma + i\tau \) where \( \sigma, \tau \in \mathbb{R} \), we can take

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

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

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

and

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

We can replace \( \sigma \) with any lower bound, conveniently the nearest integer in the direction of \(-\infty\), and \( |z| \) with any upper bound.

To bound \( B(m, n) \) when \( m \) is large, it is useful to note that \( B(m, n) = \exp(n \log(1 + 1/m)) \leq \exp(n/m) \).

### 3.4.2 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 \( E_1 = (i/(2\pi))^{v+x}, Z_1 = \zeta(v + x, \ldots), E_2 = (1/(2\pi i))^{v+x}, Z_2 = \zeta(v + x, \ldots). \)

When \( v = 1 \), the \( Z_1 \) and \( Z_2 \) terms become Laurent series with a leading \( 1/x \) term. In this case, we compute the deflated series \( \tilde{Z}_1, \tilde{Z}_2 = \zeta(x, \ldots) - 1/x \). Then

\[ E_1 \tilde{Z}_1 + E_2 \tilde{Z}_2 = (E_1 + E_2)/x + E_1 \tilde{Z}_1 + E_2 \tilde{Z}_2. \]
Note that \((E_1 + E_2)/x\) is a power series, since the constant term in \(E_1 + E_2\) is zero when \(v = 1\). So we simply compute one extra derivative of both \(E_1\) and \(E_2\), and shift them one step. When \(v = 0, -1, -2, \ldots\), the \(\Gamma(v + x)\) prefactor has a pole. In this case, we proceed analogously and formally multiply \(x \Gamma(v + x)\) with \([E_1 Z_1 + E_2 Z_2]/x\).

Note that the formal cancellation only works when the order \(s\) (or \(v\)) is an exact integer: it is not currently possible to use this method when \(s\) is a small ball containing any of \(0, 1, 2, \ldots\) (then the result becomes indeterminate).

The Hurwitz zeta method becomes inefficient when \(|z| \to 0\) (it gives an indeterminate result when \(z = 0\)). This is not a problem since we just use the defining series for the polylogarithm in that region. It also becomes inefficient when \(|z| \to \infty\), for which an asymptotic expansion would better.
4.1 fmpr.h – arbitrary-precision floating-point numbers

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

The component $x$ is called the mantissa, and $y$ is called the exponent. Note that this is just one among many possible conventions: the mantissa (alternatively significand) is sometimes viewed as a fraction in the interval $[1/2, 1)$, with the exponent pointing to the position above the top bit rather than the position of the bottom bit, and with a separate sign.

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

An `fmpr` number is exact and has no inherent “accuracy”. We use the term precision to denote either the target precision of an operation, or the bit size of a mantissa (which in general is unrelated to the “accuracy” of the number: for example, the floating-point value 1 has a precision of 1 bit in this sense and is simultaneously an infinitely accurate approximation of the integer 1 and a 2-bit accurate approximation of $\sqrt{2} = 1.011010100\ldots_2$).

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

### 4.1.1 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.

### 4.1.2 Memory management

`void fmpr_init (fmpr_t x)`

Initializes the variable `x` for use. Its value is set to zero.

`void fmpr_clear (fmpr_t x)`

Clears the variable `x`, freeing or recycling its allocated memory.

### 4.1.3 Special values

`void fmpr_zero (fmpr_t x)`

`void fmpr_one (fmpr_t x)`

`void fmpr_pos_inf (fmpr_t x)`

`void fmpr_neg_inf (fmpr_t x)`

`void fmpr_nan (fmpr_t x)`

Sets `x` respectively to 0, 1, +∞, −∞, NaN.

`int fmpr_is_zero (const fmpr_t x)`

`int fmpr_is_one (const fmpr_t x)`

`int fmpr_is_pos_inf (const fmpr_t x)`

`int fmpr_is_neg_inf (const fmpr_t x)`

`int fmpr_is_nan (const fmpr_t x)`

Returns nonzero iff `x` respectively equals 0, 1, +∞, −∞, NaN.

`int fmpr_is_inf (const fmpr_t x)`

Returns nonzero iff `x` equals either +∞ or −∞.
int \texttt{fmpfr\_is\_normal}(\texttt{const \ fmpfr\_t x})
\hspace{1em} \text{Returns nonzero iff } x \text{ is a finite, nonzero floating-point value, i.e. not one of the special values 0, } +\infty, -\infty, \text{NaN.}

int \texttt{fmpfr\_is\_special}(\texttt{const \ fmpfr\_t x})
\hspace{1em} \text{Returns nonzero iff } x \text{ is one of the special values 0, } +\infty, -\infty, \text{NaN, i.e. not a finite, nonzero floating-point value.}

int \texttt{fmpfr\_is\_finite}(\texttt{fmpfr\_t x})
\hspace{1em} \text{Returns nonzero iff } x \text{ is a finite floating-point value, i.e. not one of the values } +\infty, -\infty, \text{NaN. (Note that this is not equivalent to the negation of } \texttt{fmpfr\_is\_inf}().}

4.1.4 Assignment, rounding and conversions

long \_\texttt{fmpfr\_normalise}(\texttt{fmpz\_t man, fmpz\_t exp, long prec, fmpfr\_rnd\_t rnd})
\hspace{1em} \text{Rounds the mantissa and exponent in-place.}

void \texttt{fmpfr\_set}(\texttt{fmpfr\_t y, const \ fmpfr\_t x})
\hspace{1em} \text{Sets } y \text{ to a copy of } x.

void \texttt{fmpfr\_swap}(\texttt{fmpfr\_t x, fmpfr\_t y})
\hspace{1em} \text{Swaps } x \text{ and } y \text{ efficiently.}

long \texttt{fmpfr\_set\_round}(\texttt{fmpfr\_t y, const \ fmpfr\_t x, long prec, fmpfr\_rnd\_t rnd})
\hspace{1em} \text{Sets } y \text{ to a copy of } x \text{ rounded in the direction specified by } \texttt{rnd} \text{ to the number of bits specified by } \texttt{prec}. 

long \_\texttt{fmpfr\_set\_round\_mpn}(\texttt{long \* shift, fmpz\_t man, mp\_srrptr x, mp\_size\_t xn, int negative, long prec, fmpfr\_rnd\_t rnd})
\hspace{1em} \text{Given an integer represented by a pointer } x \text{ to a raw array of } \texttt{xn} \text{ limbs (negated if } \texttt{negative} \text{ is nonzero), sets } \texttt{man} \text{ to the corresponding floating-point mantissa rounded to } \texttt{prec} \text{ bits in direction } \texttt{rnd}, \text{sets } \texttt{shift} \text{ to the exponent, and returns the error bound. We require that } \texttt{xn} \text{ is positive and that the leading limb of } x \text{ is nonzero.}

long \texttt{fmpfr\_set\_round\_ui\_2exp\_fmpz}(\texttt{fmpfr\_t z, mp\_limb\_t lo, const \ fmpz\_t exp, int negative, long prec, fmpfr\_rnd\_t rnd})
\hspace{1em} \text{Sets } z \text{ to the unsigned integer } \texttt{lo} \text{ times two to the power } \texttt{exp}, \text{negating the value if } \texttt{negative} \text{ is nonzero, and rounding the result to } \texttt{prec} \text{ bits in direction } \texttt{rnd}.

long \texttt{fmpfr\_set\_round\_uiui\_2exp\_fmpz}(\texttt{fmpfr\_t z, mp\_limb\_t hi, mp\_limb\_t lo, const \ fmpz\_t exp, int negative, long prec, fmpfr\_rnd\_t rnd})
\hspace{1em} \text{Sets } z \text{ to the unsigned two-limb integer } \{\texttt{hi}, \texttt{lo}\} \text{ times two to the power } \texttt{exp}, \text{negating the value if } \texttt{negative} \text{ is nonzero, and rounding the result to } \texttt{prec} \text{ bits in direction } \texttt{rnd}.

void \texttt{fmpfr\_set\_error\_result}(\texttt{fmpfr\_t err, const \ fmpfr\_t result, long rret})
\hspace{1em} Given the return value } rret \text{ and output variable } result \text{ from a function performing a rounding (e.g. } \texttt{fmpfr\_set\_round} \text{ or } \texttt{fmpfr\_add}, \text{sets } err \text{ to a bound for the absolute error.}

void \texttt{fmpfr\_add\_error\_result}(\texttt{fmpfr\_t err, const \ fmpfr\_t err\_in, const \ fmpfr\_t result, long rret, long prec, fmpfr\_rnd\_t rnd})
\hspace{1em} Like } \texttt{fmpfr\_set\_error\_result}, \text{ but adds } err\_in \text{ to the error.}

void \texttt{fmpfr\_ulp}(\texttt{fmpfr\_t u, const \ fmpfr\_t x, long prec})
\hspace{1em} \text{Sets } u \text{ to the floating-point unit in the last place (ulp) of } x. \text{ The ulp is defined as in the MPFR documentation and satisfies } 2^{-n}|x| < u \leq 2^{-n+1}|x| \text{ for any finite nonzero } x. \text{ If } x \text{ is a special value, } u \text{ is set to the absolute value of } x.

int \texttt{fmpfr\_check\_ulp}(\texttt{const \ fmpfr\_t x, long r, long prec})
\hspace{1em} Assume that } r \text{ is the return code and } x \text{ is the floating-point result from a single floating-point rounding. Then}
this function returns nonzero iff \( x \) and \( r \) define an error of exactly 0 or 1 ulp. In other words, this function checks that \( \text{fmpr_set_error_result()} \) gives exactly 0 or 1 ulp as expected.

```c
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.

```c
void fmpr_set_mpfr (fmpr_t x, const mpfr_t y)
```
Sets \( x \) to the exact value of the MPFR variable \( y \).

```c
double fmpr_get_d (const fmpr_t x, mpfr_rnd_t rnd)
```
Returns \( x \) rounded to a double in the direction specified by \( \text{rnd} \).

```c
void fmpr_set_d (fmpr_t x, double v)
```
Sets \( x \) the the exact value of the argument \( v \) of type double.

```c
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 \( c \).

```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 \( \text{rnd} \). If \( \text{rnd} \) is \text{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.

```c
long fmpr_get_si (const fmpr_t x, mpfr_rnd_t rnd)
```
Returns \( x \) rounded to the nearest integer in the direction specified by \( \text{rnd} \). If \( \text{rnd} \) is \text{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.

```c
void fmpr_get_fmpq (fmpq_t y, const fmpr_t x)
void fmpr_set_fmpq (fmpr_t x, const fmpq_t y, long prec, mpfr_rnd_t 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)
void fmpr_set_round_fmpz_2exp (fmpr_t x, const fmpz_t man, const fmpz_t exp, long prec, mpfr_rnd_t rnd)
void fmpr_get_fmpz_2exp (const fmpz_t man, fmpz_t exp, const fmpr_t x)
```
Sets \( x \) to \( \text{man} \times 2^{\text{exp}} \).

```c
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 \( \text{man} \times 2^{\text{exp}} \), rounded according to \( \text{prec} \) and \( \text{rnd} \).

```c
int fmpr_get_fmpz_fixed_fmpz (fmpz_t y, const fmpr_t x, const fmpz_t e)
int fmpr_get_fmpz_fixed_si (fmpz_t y, const fmpr_t x, 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.
4.1.5 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)
int fmpr_cmpabs_ui (const fmpr_t x, ulong y)
   Comparing the absolute values of x and y.

int fmpr_cmp_2exp_si (const fmpr_t x, long e)
int fmpr_cmpabs_2exp_si (const fmpr_t x, long e)
   Comparing x (respectively its absolute value) with \(2^e\).

int fmpr_sgn (const fmpr_t x)
   Returns \(-1\), 0 or +1 according to the sign of x. The sign of NaN is undefined.

void fmpr_min (fmpr_t z, const fmpr_t a, const fmpr_t b)
void fmpr_max (fmpr_t z, const fmpr_t a, const fmpr_t b)
   Sets z respectively to the minimum and the maximum of a and b.

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 \(\pm n2^e\) 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 \(|x| \leq 2^b\). If x is zero, infinity or NaN, the result is undefined.

void fmpr_abs_bound_lt_2exp_fmpz (fmpz_t b, const fmpr_t x)
   Sets b to the smallest integer such that \(|x| < 2^b\). If x is zero, infinity or NaN, the result is undefined.

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 -FMPR_PREC_EXACT and FMPR_PREC_EXACT inclusive. If x is zero, -FMPR_PREC_EXACT is returned, and if x is infinity or NaN, FMPR_PREC_EXACT is returned.

4.1.6 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.
4.1.7 Input and output

void **fmpr_print** (const **fmpr_t** x)
Prints the mantissa and exponent of x as integers, precisely showing the internal representation.

void **fmpr_printd** (const **fmpr_t** x, long digits)
Prints x as a decimal floating-point number, rounding to the specified number of digits. This function is currently implemented using MPFR, and does not support large exponents.

4.1.8 Arithmetic

void **fmpr_neg** (fmpr_t y, const fmpr_t x)
Sets y to the negation of x.

long **fmpr_neg_round** (fmpr_t y, const fmpr_t x, long prec, fmpr_rnd_t rnd)
Sets y to the negation of x, rounding the result.

void **fmpr_abs** (fmpr_t y, const fmpr_t x)
Sets y to the absolute value of x.

long **fmpr_add** (fmpr_t z, const fmpr_t x, const fmpr_t y, long prec, fmpr_rnd_t rnd)
Sets z = x + y, rounded according to prec and rnd. The precision can be FMPR_PREC_EXACT to perform an exact addition, provided that the result fits in memory.

long **fmpr_add_eps** (fmpr_t z, const fmpr_t x, int sign, long prec, fmpr_rnd_t rnd)
Sets z to the value that results by adding an infinitesimal quantity of the given sign to x, and rounding. The result is undefined if x is zero.

long **fmpr_sub** (fmpr_t z, const fmpr_t x, const fmpr_t y, long prec, fmpr_rnd_t rnd)
Sets z = x − y, rounded according to prec and rnd. The precision can be FMPR_PREC_EXACT to perform an exact addition, provided that the result fits in memory.

long **fmpr_sum** (fmpr_t s, const fmpr_struct * terms, long len, long prec, fmpr_rnd_t rnd)
Sets s to the sum of the array terms of length len, rounded to prec bits in the direction rnd. The sum is computed as if done without any intermediate rounding error, with only a single rounding applied to the final result. Unlike repeated calls to fmpr_add, this function does not overflow if the magnitudes of the terms are far apart. Warning: this function is implemented naively, and the running time is quadratic with respect to len in the worst case.

long **fmpr_mul** (fmpr_t z, const fmpr_t x, const fmpr_t y, long prec, fmpr_rnd_t rnd)
Sets z = x ∗ y, rounded according to prec and rnd. The precision can be FMPR_PREC_EXACT to perform an exact multiplication, provided that the result fits in memory.

void **fmpr_mul_2exp_si** (fmpr_t y, const fmpr_t x, long e)
void \texttt{fmpr\_mul\_2exp\_fmpz} (\texttt{fmpr\_t y}, \texttt{const fmpr\_t x}, \texttt{const fmpz\_t e})
\hspace{1em} 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 fmpr\_t x}, \texttt{const fmpr\_t y}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})
\hspace{1em} Sets \(z = x/y\), rounded according to \texttt{prec} and \texttt{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})
\hspace{1em} Sets \(z\) to an upper bound for \(|x|/|y|\), computed to a precision of approximately \texttt{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})
\hspace{1em} Sets \(z = z + x \times y\), rounded according to \texttt{prec} and \texttt{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})
\hspace{1em} Sets \(z = z - x \times y\), rounded according to \texttt{prec} and \texttt{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 fmpr\_t x}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})
\hspace{1em} Sets \(z\) to the square root of \(x\), rounded according to \texttt{prec} and \texttt{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})
\hspace{1em} Sets \(z\) to the reciprocal square root of \(x\), rounded according to \texttt{prec} and \texttt{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})
\hspace{1em} Sets \(z\) to the \(k\)-th root of \(x\), rounded to \texttt{prec} bits in the direction \texttt{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 \texttt{fmpr\_pow\_sloppy\_si} (\texttt{fmpr\_t y}, \texttt{const fmpr\_t b}, \texttt{long e}, \texttt{long prec}, \texttt{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 $b \geq 0$.

4.1.9 Special functions

\begin{itemize}
\item \texttt{long fmpr\_log} (\texttt{fmpr\_t y}, \texttt{const fmpr\_t x}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})
\hspace{1em} Sets $y$ to $\log(x)$, rounded according to \texttt{prec} and \texttt{rnd}. The result is NaN if $x$ is negative. This function is currently implemented using MPFR and does not support large exponents.
\item \texttt{long fmpr\_log1p} (\texttt{fmpr\_t y}, \texttt{const fmpr\_t x}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})
\hspace{1em} Sets $y$ to $\log(1 + x)$, rounded according to \texttt{prec} and \texttt{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.
\item \texttt{long fmpr\_exp} (\texttt{fmpr\_t y}, \texttt{const fmpr\_t x}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})
\hspace{1em} Sets $y$ to $\exp(x)$, rounded according to \texttt{prec} and \texttt{rnd}. This function is currently implemented using MPFR and does not support large exponents.
\item \texttt{long fmpr\_expml} (\texttt{fmpr\_t y}, \texttt{const fmpr\_t x}, \texttt{long prec}, \texttt{fmpr\_rnd\_t rnd})
\hspace{1em} Sets $y$ to $\exp(x) - 1$, rounded according to \texttt{prec} and \texttt{rnd}. This function computes an accurate value when $x$ is small. This function is currently implemented using MPFR and does not support large exponents.
\end{itemize}

4.2 \texttt{fmprb\_h} – real numbers represented as floating-point balls

An \texttt{fmprb\_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 \texttt{fmprb\_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 \texttt{fmpr} 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). There are some caveats: in general, ball arithmetic only makes sense for (Lipschitz) continuous functions, and trying to approximate functions close to singularities might result in slow convergence, or failure to converge.

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 radius of a ball is not allowed to be negative or NaN.

### 4.2.1 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*.

### 4.2.2 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.

### 4.2.3 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 \texttt{fmp}r\texttt{b} \_\texttt{s}et \_\texttt{fmp} \texttt{r} (fmp\texttt{b}\_\texttt{t} y, const fmp\texttt{r}\_\texttt{t} x)

void \texttt{fmp}r\texttt{b} \_\texttt{s}et \_\texttt{s}i (fmp\texttt{b}\_\texttt{t} y, long x)

void \texttt{fmp}r\texttt{b} \_\texttt{s}et \_\texttt{ui} (fmp\texttt{b}\_\texttt{t} y, ulong x)

void \texttt{fmp}r\texttt{b} \_\texttt{s}et \_\texttt{fmpz} (fmp\texttt{b}\_\texttt{t} y, const fmpz\_\texttt{t} x)

Sets y exactly to x.

void \texttt{fmp}r\texttt{b} \_\texttt{s}et \_\texttt{fmpq} (fmp\texttt{b}\_\texttt{t} y, const fmpq\_\texttt{t} x, long prec)

Sets y to the rational number x, rounded to prec bits.

void \texttt{fmp}r\texttt{b} \_\texttt{s}et \_\texttt{fmpz} \_\texttt{2}exp (fmp\texttt{r}\_\texttt{t} x, const fmpz\_\texttt{t} y, const fmpz\_\texttt{t} exp)

Sets x to y multiplied by 2 raised to the power exp.

void \texttt{fmp}r\texttt{b} \_\texttt{s}et \_\texttt{round} \_\texttt{fmpz} \_\texttt{2}exp (fmp\texttt{r}\_\texttt{t} y, const fmpz\_\texttt{t} x, const fmpz\_\texttt{t} exp, long prec)

Sets x to y multiplied by 2 raised to the power exp, rounding the result to prec bits.

\subsection*{4.2.4 Assignment of special values}

void \texttt{fmp}r\texttt{b} \_\texttt{z}ero (fmp\texttt{b}\_\texttt{t} x)

Sets x to zero.

void \texttt{fmp}r\texttt{b} \_\texttt{one} (fmp\texttt{b}\_\texttt{t} x)

Sets x to the exact integer 1.

void \texttt{fmp}r\texttt{b} \_\texttt{pos} \_\texttt{inf} (fmp\texttt{b}\_\texttt{t} x)

Sets x to positive infinity, with a zero radius.

void \texttt{fmp}r\texttt{b} \_\texttt{neg} \_\texttt{inf} (fmp\texttt{b}\_\texttt{t} x)

Sets x to negative infinity, with a zero radius.

void \texttt{fmp}r\texttt{b} \_\texttt{zero} \_\texttt{pm} \_\texttt{inf} (fmp\texttt{b}\_\texttt{t} x)

Sets x to $[0 \pm \infty]$, representing the whole extended real line.

void \texttt{fmp}r\texttt{b} \_\texttt{indeterminate} (fmp\texttt{b}\_\texttt{t} x)

Sets x to $[\text{NaN} \pm \infty]$, representing an indeterminate result.

\subsection*{4.2.5 Input and output}

void \texttt{fmp}r\texttt{b} \_\texttt{pr}in\texttt{t} \ (const fmp\texttt{b}\_\texttt{t} x)

Prints the internal representation of x.

void \texttt{fmp}r\texttt{b} \_\texttt{prin}td \ (const fmp\texttt{b}\_\texttt{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.

\subsection*{4.2.6 Random number generation}

void \texttt{fmp}r\texttt{b} \_\texttt{rand}t\texttt{est} \ (fmp\texttt{r}\_\texttt{t} x, flint\_\texttt{rand} \_\texttt{t} state, long prec, long mag\_\texttt{bits})

Generates a random ball. The midpoint and radius will both be finite.

void \texttt{fmp}r\texttt{b} \_\texttt{rand}t\texttt{est} \_\texttt{exact} \ (fmp\texttt{r}\_\texttt{t} x, flint\_\texttt{rand} \_\texttt{t} state, long prec, long mag\_\texttt{bits})

Generates a random number with zero radius.

void \texttt{fmp}r\texttt{b} \_\texttt{rand}t\texttt{est} \_\texttt{prec}ise \ (fmp\texttt{r}\_\texttt{t} x, flint\_\texttt{rand} \_\texttt{t} state, long prec, long mag\_\texttt{bits})

Generates a random number with radius at most $2^{-\text{prec}}$ the magnitude of the midpoint.
void \texttt{fmprb\_randtest\_wide}(\texttt{fmprb} \, \texttt{t} \, \texttt{x}, \texttt{flint\_rand\_t} \, \texttt{state}, \texttt{long} \, \texttt{prec}, \texttt{long} \, \texttt{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} \, \texttt{t} \, \texttt{x}, \texttt{flint\_rand\_t} \, \texttt{state}, \texttt{long} \, \texttt{prec}, \texttt{long} \, \texttt{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} \, \texttt{t} \, \texttt{q}, \texttt{flint\_rand\_t} \, \texttt{state}, \texttt{const} \texttt{fmprb} \, \texttt{t} \, \texttt{x}, \texttt{long} \, \texttt{bits})

Sets \texttt{q} to a random rational number from the interval represented by \texttt{x}. A denominator is chosen by multiplying the binary denominator of \texttt{x} by a random integer up to \texttt{bits} bits.

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

### 4.2.7 Radius and interval operations

void \texttt{fmprb\_add\_error\_fmpq}(\texttt{fmprb} \, \texttt{t} \, \texttt{x}, \texttt{const} \texttt{fmpr} \, \texttt{t} \, \texttt{err})

Adds \texttt{err}, which is assumed to be nonnegative, to the radius of \texttt{x}.

void \texttt{fmprb\_add\_error\_2exp\_si}(\texttt{fmprb} \, \texttt{t} \, \texttt{x}, \texttt{long} \, \texttt{e})

void \texttt{fmprb\_add\_error\_2exp\_fmpz}(\texttt{fmprb} \, \texttt{t} \, \texttt{x}, \texttt{const} \texttt{fmpz} \, \texttt{t} \, \texttt{e})

Adds \(2^e\) to the radius of \texttt{x}.

void \texttt{fmprb\_add\_error}(\texttt{fmprb} \, \texttt{t} \, \texttt{x}, \texttt{const} \texttt{fmprb} \, \texttt{t} \, \texttt{err})

Adds the supremum of \texttt{err}, which is assumed to be nonnegative, to the radius of \texttt{x}.

void \texttt{fmprb\_union}(\texttt{fmprb} \, \texttt{t} \, \texttt{z}, \texttt{const} \texttt{fmprb} \, \texttt{t} \, \texttt{x}, \texttt{const} \texttt{fmprb} \, \texttt{t} \, \texttt{y}, \texttt{long} \, \texttt{prec})

Sets \texttt{z} to a ball containing both \texttt{x} and \texttt{y}.

void \texttt{fmprb\_get\_abs\_ubound\_fmpq}(\texttt{fmpq} \, \texttt{t} \, \texttt{u}, \texttt{const} \texttt{fmprb} \, \texttt{t} \, \texttt{x}, \texttt{long} \, \texttt{prec})

Sets \texttt{u} to the upper bound of the absolute value of \texttt{x}, rounded up to \texttt{prec} bits. If \texttt{x} contains NaN, the result is NaN.

void \texttt{fmprb\_get\_abs\_lbound\_fmpq}(\texttt{fmpq} \, \texttt{t} \, \texttt{u}, \texttt{const} \texttt{fmprb} \, \texttt{t} \, \texttt{x}, \texttt{long} \, \texttt{prec})

Sets \texttt{u} to the lower bound of the absolute value of \texttt{x}, rounded down to \texttt{prec} bits. If \texttt{x} contains NaN, the result is NaN.

void \texttt{fmprb\_get\_interval\_fmpz\_2exp}(\texttt{fmpz} \, \texttt{t} \, \texttt{a}, \texttt{fmpz} \, \texttt{t} \, \texttt{b}, \texttt{fmpz} \, \texttt{t} \, \texttt{exp}, \texttt{const} \texttt{fmprb} \, \texttt{t} \, \texttt{x})

Computes the exact interval represented by \texttt{x}, in the form of an integer interval multiplied by a power of two, i.e. \(x = [a, b] \times 2^{\text{exp}}\).

The outcome is undefined if the midpoint or radius of \texttt{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 \texttt{fmprb\_set\_interval\_fmpq}(\texttt{fmprb} \, \texttt{t} \, \texttt{x}, \texttt{const} \texttt{fmpq} \, \texttt{t} \, \texttt{a}, \texttt{const} \texttt{fmpq} \, \texttt{t} \, \texttt{b}, \texttt{long} \, \texttt{prec})

Sets \texttt{x} to a ball containing the interval \([a, b]\). We require that \(a \leq b\).

long \texttt{fmprb\_rel\_error\_bits}(\texttt{const} \texttt{fmprb} \, \texttt{t} \, \texttt{x})

Returns the effective relative error of \texttt{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 \texttt{FMPR\_PREC\_EXACT}.

long \texttt{fmprb\_rel\_accuracy\_bits}(\texttt{const} \texttt{fmprb} \, \texttt{t} \, \texttt{x})

Returns the effective relative accuracy of \texttt{x} measured in bits, equal to the negative of the return value from \texttt{fmprb\_rel\_error\_bits}.

long \texttt{fmprb\_bits}(\texttt{const} \texttt{fmprb} \, \texttt{t} \, \texttt{x})

Returns the number of bits needed to represent the absolute value of the mantissa of the midpoint of \texttt{x}, i.e. the minimum precision sufficient to represent \texttt{x} exactly. Returns 0 if the midpoint of \texttt{x} is a special value.
void \texttt{fmprb\_trim}(fmprb\_t y, const fmprb\_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.

\begin{verbatim}
int fmprb\_get\_unique\_fmpz(fmpz\_t z, const fmprb\_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.
\end{verbatim}

### 4.2.8 Comparisons

\begin{verbatim}
int fmprb\_is\_zero(const fmprb\_t x)  
Returns nonzero iff the midpoint and radius of \( x \) are both zero.

int fmprb\_is\_nonzero(const fmprb\_t x)  
Returns nonzero iff zero is not contained in the interval represented by \( x \).

int fmprb\_is\_one(const fmprb\_t x)  
Returns nonzero iff \( x \) is exactly 1.

int fmprb\_is\_finite(fmprb\_t x)  
Returns nonzero iff the midpoint and radius of \( x \) are both finite floating-point numbers, i.e. not infinities or NaN.

int fmprb\_is\_exact(const fmprb\_t x)  
Returns nonzero iff the radius of \( x \) is zero.

int fmprb\_is\_int(const fmprb\_t x)  
Returns nonzero iff \( x \) is an exact integer.

int fmprb\_equal(const fmprb\_t x, const fmprb\_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 \texttt{fmprb\_overlaps()} or \texttt{fmprb\_contains()}, depending on the circumstance.

int fmprb\_is\_positive(const fmprb\_t x)  
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 fmp\_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)  
\end{verbatim}
int fmprb_contains (const fmprb_t x, const fmprb_t y)
Returns nonzero iff the given number (or ball) y is contained in the interval represented by x.
If x is contains NaN, this function always returns nonzero (as it could represent anything, and in particular could represent all the points included in y). If y contains NaN and x does not, it always returns zero.

int 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 ≤ 0, p > 0, p ≥ 0. If x contains NaN, returns nonzero.

4.2.9 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_fmpr (fmprb_t z, const fmprb_t x, const fmpr_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 fmprb_sub_si (fmprb_t z, const fmprb_t x, long y, long prec)
void fmprb_sub_fmpz (fmprb_t z, const fmprb_t x, const fmpz_t y, long prec)
void fmprb_sub_fmpr (fmprb_t z, const fmprb_t x, const fmpr_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_mul (fmprb_t z, const fmprb_t x, const fmprb_t y, long prec)
void fmprb_mul_ui (fmprb_t z, const fmprb_t x, ulong y, long prec)
void fmprb_mul_si (fmprb_t z, const fmprb_t x, long y, long prec)
void fmprb_mul_fmpz (fmprb_t z, const fmprb_t x, const fmpz_t y, long prec)
void fmprb_mul_fmpr (fmprb_t z, const fmprb_t x, const fmpr_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_mul_2exp_si (fmprb_t y, const fmprb_t x, long e)
void fmprb_mul_2exp_fmpz (fmprb_t y, const fmprb_t x, const fmpz_t e)
Sets y to x multiplied by 2^e.

void fmprb_inv (fmprb_t z, const fmprb_t x, long prec)
Sets z to the multiplicative inverse of x.
void fmprb_div (fmprb_t z, const fmprb_t x, const fmprb_t y, long prec)

4.2. fmprb.h – real numbers represented as floating-point balls
4.2.10 Powers and roots

void \texttt{fmprb_sqrt} (fmprb_t z, const fmprb_t x, long prec)

void \texttt{fmprb_sqrt_ui} (fmprb_t z, const fmprb_t x, long prec)

void \texttt{fmprb_sqrt_fmpz} (fmprb_t z, const fmprb_t x, const fmpz_t y, long prec)

void \texttt{fmprb_sqrt_fmpfr} (fmprb_t z, const fmprb_t x, const fmpfr_t y, long prec)

void \texttt{fmpfr_sqrt} (fmpfr_t z, const fmpfr_t x, long prec)

void \texttt{fmpfr_sqrt_ui} (fmpfr_t z, const fmpfr_t x, ulong n, long prec)

void \texttt{fmpfr_sqrt_fmpz} (fmpfr_t z, const fmpfr_t x, const fmpz_t y, long prec)

void \texttt{fmpfr_sqrt_fmpfr} (fmpfr_t z, const fmpfr_t x, const fmpfr_t y, long prec)

void \texttt{fmpfr_hypot} (fmpfr_t z, const fmpfr_t x, const fmpfr_t y, long prec)

Sets \( z = \sqrt{x^2 + y^2} \).
void \texttt{fmpfr\_rsqrt\_ui} (fmpfr\_t \texttt{z}, ulong \texttt{x}, long \texttt{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{fmpfr\_root} (fmpfr\_t \texttt{z}, const fmpfr\_t \texttt{x}, ulong \texttt{k}, long \texttt{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{fmpfr\_pow\_fmpq()} or \texttt{fmpfr\_pow()}. 

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

Sets \( z \) to the arithmetic-geometric mean of \( x \) and \( y \).
5.1 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.

5.1.1 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

5.1.2 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
5.1.3 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:

```latex
@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}
}
```

5.1.4 Bibliography
BIBLIOGRAPHY


Symbols

acb_poly_add (C function), 73
acb_poly_atan_series (C function), 78
acb_poly_compose (C function), 75
acb_poly_compose_divconquer (C function), 75
acb_poly_compose_horner (C function), 75
acb_poly_compose_series (C function), 75
acb_poly_compose_series_brent_kung (C function), 75
acb_poly_compose_series_horner (C function), 75
acb_poly_cos_series (C function), 79
acb_poly_derivative (C function), 77
acb_poly_div (C function), 74
acb_poly_div_root (C function), 74
acb_poly_div_series (C function), 74
acb_poly_divrem (C function), 74
acb_poly_evaluate (C function), 76
acb_poly_evaluate2 (C function), 76
acb_poly_evaluate2_horner (C function), 76
acb_poly_evaluate2_rectangular (C function), 76
acb_poly_evaluate_horner (C function), 76
acb_poly_evaluate_rectangular (C function), 76
acb_poly_evaluate_vec_fast (C function), 77
acb_poly_evaluate_vec_fast_precomp (C function), 77
acb_poly_evaluate_vec_iter (C function), 77
acb_poly_exp_series (C function), 78
acb_poly_exp_series_basecase (C function), 78
acb_poly_find_roots (C function), 82
acb_poly_gamma_series (C function), 79
acb_poly_integral (C function), 77
acb_poly_interpolate_barycentric (C function), 77
acb_poly_interpolate_fast (C function), 77
acb_poly_interpolate_precomp (C function), 77
acb_poly_interpolate_newton (C function), 77
acb_poly_interpolation_weights (C function), 77
acb_poly_inv_series (C function), 78
acb_poly_log_series (C function), 81
acb_poly_mul (C function), 74
acb_poly_mul_mullow (C function), 73
acb_poly_mul_mullow_classical (C function), 73
acb_poly_mul_mullow_transpose (C function), 73
acb_poly_mul_mullow_transpose_gauss (C function), 73
acb_poly_normalise (C function), 71
acb_poly_overlaps (C function), 71
acb_poly_polylog_cpx (C function), 81
acb_poly_polylog_cpx_small (C function), 81
acb_poly_polylog_cpx_zeta (C function), 81
acb_poly_polylog_series (C function), 81
acb_poly_pow_ui (C function), 78
acb_poly_pow_ui_trunc_binexp (C function), 77
acb_poly_powsum_one_series_sieved (C function), 80
acb_poly_powsum_series_naive (C function), 80
acb_poly_powsum_series_naive_threaded (C function), 80
acb_poly_product_roots (C function), 76
acb_poly_refine_roots_durand_kerner (C function), 82
acb_poly_rem (C function), 74
acb_poly_revert_series (C function), 75
acb_poly_revert_series_lagrange (C function), 75
acb_poly_revert_series_lagrange_fast (C function), 75
acb_poly_revert_series_newton (C function), 75
acb_poly_rgamma_series (C function), 79
acb_poly_rising_ui_series (C function), 80
acb_poly_root_inclusion (C function), 82
acb_poly_sqrt_series (C function), 78
acb_poly_sub (C function), 73
acb_poly_tree_alloc (C function), 76
acb_poly_tree_build (C function), 76
acb_poly_tree_free (C function), 76
acb_poly_validate_roots (C function), 82
acb_poly_zeta_cpx_series (C function), 81
acb_poly_zeta_em_bound (C function), 80
acb_poly_zeta_em_bound1 (C function), 80
acb_poly_zeta_em_choose_param (C function), 80
acb_poly_zeta_em_sum (C function), 81
Arb Documentation, Release 2.4.0

Index 133

_acb_abs (C function), 67
_acb_add (C function), 67
_acb_add_arb (C function), 67
_acb_add_fmpz (C function), 67
_acb_add_ui (C function), 67
_acb_addmul (C function), 68
_acb_addmul_arb (C function), 68
_acb_addmul_fmpz (C function), 68
_acb_addmul_si (C function), 68
_acb_addmul_ui (C function), 68
_acb_arg (C function), 67
_acb_bits (C function), 67
_acb_cale_caucauy_bound (C function), 86
_acb_calc_func_t (C type), 86
_acb_calc_integrate_taylor (C function), 87
_acb_clear (C function), 65
_acb_conj (C function), 67
_acb_const_pi (C function), 68
_acb_contin (C function), 67
_acbContains_fmpq (C function), 67
_acbContains_fmpz (C function), 67
_acbContains_zero (C function), 67
_acb cos (C function), 69
_acb cos _pi (C function), 69
_acb cot (C function), 69
_acb cot _pi (C function), 69
_acb_cube (C function), 68
_acb digamma (C function), 70
_acb div (C function), 68
_acb div_fmpz (C function), 68
_acb div _si (C function), 68
_acb div _ui (C function), 68
_acb equal (C function), 66
_acb exp (C function), 68
_acb exp _pi _j (C function), 69
_acb gamma (C function), 70
_acb get abs _lbound _arf (C function), 66
_acb get abs _ubound _arf (C function), 66
_acb get mag (C function), 66
_acb get mag _lower (C function), 66
_acb get_rad _ubound _arf (C function), 66
_acb hurwitz _zeta (C function), 70
_acb hypgeome_bessel _i (C function), 91
_acb hypgeome_bessel _i _0f1 (C function), 91
_acb hypgeome_bessel _j (C function), 90
_acb hypgeome_bessel _j _asymp (C function), 90
_acb hypgeome_ erf (C function), 90
_acb hypgeome_ erf _1f1a (C function), 90
_acb hypgeome_ erf _1f1b (C function), 90
_acb hypgeome_ erf _asymp (C function), 90
_acb hypgeome_pfq_bound_factor (C function), 88
_acb hypgeome_pfq_choose _n (C function), 88
_acb hypgeome_pfq_direct (C function), 89
_acb hypgeome_pfq_sum (C function), 88
_acb hypgeome_pfq_sum_forward (C function), 88
_acb hypgeome_pfq_sum_rs (C function), 88
_acb hypgeome_u_asym (C function), 90
_acb init (C function), 65
_acb inv (C function), 68
_acb is_exact (C function), 65
_acb is_one (C function), 65
_acb is_real (C function), 67
_acb is_zero (C function), 65
_acb lgamma (C function), 70
_acb log (C function), 68
_acb mat add (C function), 84
_acb mat bound _inf _norm (C function), 84
_acb mat clear (C function), 83
_acb mat contains (C function), 84
_acb mat contains _fmpq_mat (C function), 84
_acb mat contains _fmpz_mat (C function), 84
_acb mat det (C function), 85
_acb mat entry (C macro), 83
_acb mat equal (C function), 84
_acb mat exp (C function), 86
_acb mat init (C function), 83
_acb mat inv (C function), 85
_acb mat lu (C function), 85
_acb mat mul (C function), 84
_acb mat ncols (C macro), 83
_acb mat neg (C function), 84
_acb mat nrows (C macro), 83
_acb mat one (C function), 84
_acb mat overlaps (C function), 84
_acb mat pow _ui (C function), 84
_acb mat printd (C function), 83
_acb mat scalar _addmul _acb (C function), 85
_acb mat scalar _addmul _arb (C function), 84
_acb mat scalar _addmul _fmpz (C function), 84
_acb mat scalar _addmul _si (C function), 84
_acb mat scalar _div _acb (C function), 85
_acb mat scalar _div _arb (C function), 85
_acb mat scalar _div _fmpz (C function), 85
_acb mat scalar _div _si (C function), 85
_acb mat scalar _mul _2exp _si (C function), 84
_acb mat scalar _mul _acb (C function), 85
_acb mat scalar _mul _arb (C function), 85
_acb mat scalar _mul _fmpz (C function), 85
_acb mat scalar _mul _si (C function), 85
_acb mat scalar _mul _ui (C function), 85
_acb mat scalar _mul _ui (C function), 85
arb_calc_isolate_roots (C function), 63
arb_calc_newton_conv_factor (C function), 64
arb_calc_newton_step (C function), 64
ARB_CALC_NO_CONVERGENCE (C macro), 62
arb_calc_refine_root_bisect (C function), 63
arb_calc_refine_root_newton (C function), 64
ARB_CALC_SUCCESS (C macro), 64
arb_calc_verbose (C variable), 62
arb_ceil (C function), 36
arb_chebyshev_t2_ui (C function), 44
arb_chebyshev_t_ui (C function), 44
arb_chebyshev_u2_ui (C function), 44
arb_chebyshev_u_ui (C function), 44
arb_clear (C function), 33
arb_const_apery (C function), 42
arb_const_catalan (C function), 42
arb_const_e (C function), 42
arb_const_euler (C function), 42
arb_const_glaiasher (C function), 42
arb_const_khinchin (C function), 42
arb_const_log10 (C function), 41
arb_const_log2 (C function), 41
arb_const_log_sqrt2pi (C function), 41
arb_const_pi (C function), 41
arb_const_sqrt_pi (C function), 41
arb_contains (C function), 37
arb_contains_arf (C function), 36
arb_contains_fmpq (C function), 36
arb_contains_fmpz (C function), 36
arb_contains_mpfr (C function), 36
arb_contains_negative (C function), 37
arb_contains_nonnegative (C function), 37
arb_contains_nonpositive (C function), 37
arb_contains_positive (C function), 37
arb_contains_si (C function), 36
arb_contains_zero (C function), 37
arb_cos (C function), 40
arb_cos_pi (C function), 40
arb_cos_pi_fmpq (C function), 40
arb_cosh (C function), 41
arb_cot (C function), 40
arb_cot_pi (C function), 40
arb_coth (C function), 41
arb_digamma (C function), 43
arb_div (C function), 38
arb_div_2expm1_ui (C function), 38
arb_div_arf (C function), 38
arb_div_fmpz (C function), 38
arb_div_ui (C function), 38
arb_equal (C function), 36
arb_expm1 (C function), 40
arb_exp (C function), 40
arb_exp_arf_bb (C function), 46
arb_expf (C function), 40
arb_fac_ui (C function), 42
arb_fib_fmpz (C function), 44
arb_fib_ui (C function), 44
arb_floor (C function), 36
arb_fmpz_div_fmpq (C function), 38
arb_gamma (C function), 42
arb_gamma_fmpq (C function), 42
arb_gamma_fmpz (C function), 42
arb_get_abs_lbound_arf (C function), 35
arb_get_abs_ubound_arf (C function), 35
arb_get_fmprb (C function), 33
arb_get_interval_arf (C function), 35
arb_get_interval_fmpz_2exp (C function), 35
arb_get_interval_mpfr (C function), 35
arb_get_mag (C function), 35
arb_get_mag_lower (C function), 35
arb_get_mag_lower_nonnegative (C function), 35
arb_get_rand_fmpq (C function), 34
arb_get_unique_fmpz (C function), 36
arb_hypgeom_infsum (C function), 100
arb_hypgeom_sum (C function), 100
arb_hypot (C function), 39
arb_imagref (C macro), 65
arb_indeterminate (C function), 34
arb_init (C function), 33
arb_inv (C function), 38
arb_is_exact (C function), 36
arb_is_finite (C function), 36
arb_is_int (C function), 36
arb_is_negative (C function), 36
arb_is_nonnegative (C function), 36
arb_is_nonpositive (C function), 36
arb_is_nonzero (C function), 36
arb_is_one (C function), 36
arb_is_positive (C function), 36
arb_is_zero (C function), 36
arb_lgamma (C function), 43
arb_log (C function), 39
arb_log_arf (C function), 39
arb_log_fmpz (C function), 39
arb_log_fmpq (C function), 39
arb_log_ui (C function), 39
arb_log ui_from_prev (C function), 40
arb_mat_add (C function), 60
arb_mat_bound_inf_norm (C function), 60
arb_mat_clear (C function), 59
arb_mat_equal (C function), 59
arb_mat_exp (C function), 59
arb_mat_det (C function), 59
arb_mat_entry (C macro), 58
arb_mat_equal (C function), 59
arb_mat_exp (C function), 61
arb_mat_init (C function), 59
arb_mat_inv (C function), 61
Index
arb_poly_pow_ui (C function), 54
arb_poly_pow_ui_trunc_bineqxp (C function), 54
arb_poly_print (C function), 48
arb_poly_product_roots (C function), 52
arb_poly_randtest (C function), 48
arb_poly_revert_series (C function), 51
arb_poly_revert_series_lagrange (C function), 50
arb_poly_revert_series_lagrange_fast (C function), 51
arb_poly_revert_series_newton (C function), 50
arb_poly_rgamma_series (C function), 57
arb_poly_riemann_siegel_theta_series (C function), 57
arb_poly_riemann_siegel_z_series (C function), 57
arb_poly_rising_ui_series (C function), 57
arb_poly_rising_ui_series_basecase (C function), 57
arb_poly_scalar_mul_2exp_si (C function), 48
arb_poly_set_coeff_arb (C function), 47
arb_poly_set_coeff_si (C function), 47
arb_poly_set_fmpq_poly (C function), 47
arb_poly_set_fmpz_poly (C function), 47
arb_poly_set_si (C function), 47
arb_poly_shift_left (C function), 47
arb_poly_shift_right (C function), 47
arb_poly_sin_cos_series (C function), 56
arb_poly_sin_cos_series_basecase (C function), 56
arb_poly_sin_cos_series_tangent (C function), 56
arb_poly_sin_series (C function), 56
arb_poly_sqrt_series (C function), 54
arb_poly_sqrt_arf (C function), 38
arb_poly_sqrt_fmpz (C function), 39
arb_poly_sqrt_ui (C function), 39
arb_poly_sub (C function), 37
arb_poly_sub_arf (C function), 37
arb_poly_sub_fmpz (C function), 37
arb_poly_sub_si (C function), 37
arb_poly_sub_ui (C function), 37
arb_poly_submul (C function), 38
arb_poly_submul_arf (C function), 38
arb_poly_submul_fmpq (C function), 38
arb_poly_submul_fmpz (C function), 38
arb_poly_submul_si (C function), 38
arb_poly_submul_ui (C function), 38
arb_poly_swap (C function), 32
arb_poly_tan (C function), 40
arb_poly_tan_pi (C function), 40
arb_poly_rising2_ui (C function), 42
arb_poly_rising2_ui_bs (C function), 42
arb_poly_rising2_ui_rs (C function), 42
arb_poly_rising_fmpq_ui (C function), 42
arb_poly_rising_ui (C function), 42
arb_poly_rising_ui_bs (C function), 42
arb_poly_rising_ui_rec (C function), 42
arb_poly_rising_ui_rs (C function), 42
arb_root (C function), 39
arb_rsqrt (C function), 39
arb_rsqrt_ui (C function), 39
arb_set (C function), 33
arb_set_arf (C function), 33
arb_set_fmpq (C function), 33
arb_set_fmpz (C function), 33
arb_set_fmpz_arb (C function), 33
arb_set_fmpz_fmpq (C function), 33
arb_set_fmpz_fmpz (C function), 33
arb_set_fmpz_si (C function), 33
arb_set_fmpz_2exp (C function), 33
arb_set_interval_arf (C function), 35
arb_set_interval_mpfr (C function), 35
arb_set_round (C function), 33
arb_set_round_fmpz (C function), 33
arb_set_round_fmpz_2exp (C function), 33
arb_set_si (C function), 33
arb_set_ui (C function), 33
arb_set_ui_pow_ui (C function), 39
arb_sin (C function), 40
arb_sin_cos (C function), 40
arb_sin_pi (C function), 40
arb_sin_pi_fmpq (C function), 40
arb_sinh (C function), 41
arb_sinh_cosh (C function), 41
arb_sqrt (C function), 38
arb_sqrt_arf (C function), 38
arb_sqrt_fmpz (C function), 39
arb_sqrt_ui (C function), 39
arb_sqrtipo (C function), 39
arb_srcptr (C type), 32
arb_struct (C type), 32
arb_swap (C function), 32
arb_sub (C function), 37
arb_sub_arf (C function), 37
arb_sub_fmpz (C function), 37
arb_sub_si (C function), 37
arb_sub_ui (C function), 37
arb_submul (C function), 38
arb_submul_arf (C function), 38
arb_submul_fmpq (C function), 38
arb_submul_fmpz (C function), 38
arb_submul_si (C function), 38
arb_submul_ui (C function), 38
arb_swap (C function), 33
arb_t (C type), 32
arb_tan (C function), 40
arb_tan_pi (C function), 40
Index
bernoulli_bound_2exp_si (C function), 97
bernoulli_cache (C variable), 97
bernoulli_cache_compute (C function), 97
bernoulli_cache_num (C variable), 97
bernoulli_fmpq_ui (C function), 98
bernoulli_rev_clear (C function), 97
bernoulli_rev_init (C function), 97
bernoulli_rev_next (C function), 97
bernoulli_rev_t (C type), 97

F

fmpr_abs (C function), 116
fmpr_abs_bound_le_2exp_fmpz (C function), 115
fmpr_abs_bound_lt_2exp_fmpz (C function), 115
fmpr_abs_bound_lt_2exp_si (C function), 115
fmpr_add (C function), 116
fmpr_add_error_result (C function), 113
fmpr_add_fmpz (C function), 116
fmpr_add_si (C function), 116
fmpr_add_ui (C function), 116
fmpr_addmul (C function), 117
fmpr_addmul_fmpz (C function), 117
fmpr_addmul_si (C function), 117
fmpr_addmul_ui (C function), 117
fmpr_bits (C function), 115
fmpr_check_ulp (C function), 113
fmpr_clear (C function), 112
fmpr_cmp (C function), 115
fmpr_cmp_2exp_si (C function), 115
fmpr_cmpabs (C function), 115
fmpr_cmpabs_2exp_si (C function), 115
fmpr_cmpabs_ui (C function), 115
fmpr_div (C function), 117
fmpr_div_fmpz (C function), 117
fmpr_div_si (C function), 117
fmpr_div_ui (C function), 117
fmpr_divappr_abs_ubound (C function), 117
fmpr_equal (C function), 115
fmpr_exp (C function), 118
fmpr_expm1 (C function), 118
fmpr_fmpz_div (C function), 117
fmpr_fmpz_div_fmpz (C function), 117
fmpr_get_d (C function), 114
fmpr_get_fmpq (C function), 114
fmpr_get_fmpz (C function), 114
fmpr_get_fmpz_2exp (C function), 114
fmpr_get_fmpz_fixed_fmpz (C function), 114
fmpr_get_fmpz_fixed_si (C function), 114
fmpr_get_mpfr (C function), 114
fmpr_get_si (C function), 114
fmpr_init (C function), 112
fmpr_is_finite (C function), 113
fmpr_is_inf (C function), 112
fmpr_is_int (C function), 115
fmpr_is_int_2exp_si (C function), 115

fmpr_is_nan (C function), 112
fmpr_is_neg_inf (C function), 112
fmpr_is_normal (C function), 112
fmpr_is_one (C function), 112
fmpr_is_pos_inf (C function), 112
fmpr_is_special (C function), 113
fmpr_is_zero (C function), 112
fmpr_log (C function), 118
fmpr_log1p (C function), 118
fmpr_max (C function), 115
fmpr_min (C function), 115
fmpr_mul (C function), 116
fmpr_mul_2exp_fmpz (C function), 116
fmpr_mul_2exp_si (C function), 116
fmpr_mul_fmpz (C function), 116
fmpr_mul_si (C function), 116
fmpr_mul_ui (C function), 116
fmpr_nan (C function), 112
fmpr_neg (C function), 116
fmpr_neg_inf (C function), 112
fmpr_neg_round (C function), 116
fmpr_one (C function), 112
fmpr_pos_inf (C function), 112
fmpr_pow_sloppy_fmpz (C function), 117
fmpr_pow_sloppy_si (C function), 117
fmpr_pow_sloppy_ui (C function), 117
FMPR_PREC_EXACT (C macro), 112
fmpr_print (C function), 116
fmpr_printd (C function), 116
fmpr_randtest (C function), 115
fmpr_randtest_not_zero (C function), 115
FMPR_RND_CEIL (C macro), 112
FMPR_RND_DOWN (C macro), 111
FMPR_RND_FLOOR (C macro), 112
FMPR_RND_NEAR (C macro), 112
fmpr_round (C function), 116
fmpr_round_t (C type), 111
FMPR_RND_UP (C macro), 111
fmpr_swap (C function), 113
fmpr_zero (C function), 112
fmprb_abs (C function), 123
fmprb_add (C function), 123
fmprb_add_error (C function), 121
fmprb_add_error_2exp_fmpz (C function), 121
fmprb_add_error_2exp_si (C function), 121
fmprb_add_fmpr (C function), 121
fmprb_add_fmpz (C function), 121
fmprb_add_fmpq (C function), 121
fmprb_add_ui (C function), 123
fmprb_addmul (C function), 124
fmprb_addmul_fmpz (C function), 124
fmprb_addmul_fmpq (C function), 124
fmprb_addmul_ui (C function), 124
fmprb_agm (C function), 125
fmprb_bits (C function), 121
fmprb_clear (C function), 119
fmprb_contains (C function), 122
fmprb_contains_fmpq (C function), 122
fmprb_contains_fmpr (C function), 122
fmprb_contains_mpfr (C function), 122
fmprb_contains_negative (C function), 123
fmprb_contains_nonnegative (C function), 123
fmprb_contains_nonpositive (C function), 123
fmprb_contains_positive (C function), 123
fmprb_contains_si (C function), 122
fmprb_mpmul (C function), 122
fmprb_neg (C function), 121
fmprb_neg_round (C function), 121
fmprb_inclusion_test (C function), 126
fmprb_normal_round (C function), 126
fmprb_options (C function), 126
fmprb_print (C function), 126
fmprb_round (C function), 126
fmprb_round_t (C type), 126
fmprb_sub (C function), 126
fmprb_submul (C function), 126
fmprb_submul_fmpz (C function), 126
fmprb_submul_fmpq (C function), 126
fmprb_submul_ui (C function), 126
fmprb_ulp (C function), 126
fmprb_zero (C function), 126

Index
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmprb_div_si</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_div_ui</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_equal</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_fmpz_div_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_get_abs_lbound_fmpr</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_get_abs_ubound_fmpr</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_get_interval_fmpz_2exp</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_get_rand_fmpq</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_get_unique_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_hypgeom_infsum</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_hypgeom_sum</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_hypot</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_indeterminate</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_init</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_inv</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_is_exact</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_is_finite</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_is_int</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_is_negative</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_is_nonnegative</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_is_nonpositive</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_is_nonzero</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_is_one</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_is_positive</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_is_zero</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_midref</td>
<td>C macro</td>
</tr>
<tr>
<td>fmprb_mul</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_mul_2exp_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_mul_2exp_si</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_mul_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_mul_si</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_mul_ui</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_neg</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_neg_inf</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_one</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_overlaps</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_pos_inf</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_print</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_printd</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_ptr</td>
<td>C type</td>
</tr>
<tr>
<td>FMPRB_RAD_PREC</td>
<td>C macro</td>
</tr>
<tr>
<td>fmprb_radref</td>
<td>C macro</td>
</tr>
<tr>
<td>fmprb_randtest</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_randtest_exact</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_randtest_precise</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_randtest_special</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_randtest_wide</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_rel_accuracy_bits</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_rel_error_bits</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_root</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_rsqrt</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_rsqrt_ui</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_set</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_set_fmpq</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_set_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_set_fmpz_2exp</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_set_interval_fmpr</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_set_round</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_set_round_fmpz_2exp</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_set_si</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_set_ui</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_sqrt</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_sqrt_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_sqrt_ui</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_sqrtpos</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_srcptr</td>
<td>C type</td>
</tr>
<tr>
<td>fmprb_struct</td>
<td>C type</td>
</tr>
<tr>
<td>fmprb_sub</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_sub_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_sub_fmpr</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_sub_si</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_sub_ui</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_submul</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_submul_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_submul_si</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_submul_ui</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_t</td>
<td>C type</td>
</tr>
<tr>
<td>fmprb_trim</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_ui_div</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_union</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_zero</td>
<td>C function</td>
</tr>
<tr>
<td>fmprb_zero_pm_inf</td>
<td>C function</td>
</tr>
</tbody>
</table>

**H**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>hypgeom_bound</td>
<td>C function</td>
</tr>
<tr>
<td>hypgeom_clear</td>
<td>C function</td>
</tr>
<tr>
<td>hypgeom_estimate_terms</td>
<td>C function</td>
</tr>
<tr>
<td>hypgeom_init</td>
<td>C function</td>
</tr>
<tr>
<td>hypgeom_precompute</td>
<td>C function</td>
</tr>
<tr>
<td>hypgeom_struct</td>
<td>C type</td>
</tr>
</tbody>
</table>

**M**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>mag_add</td>
<td>C function</td>
</tr>
<tr>
<td>mag_add_2exp_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>mag_add_lower</td>
<td>C function</td>
</tr>
<tr>
<td>mag_addmul</td>
<td>C function</td>
</tr>
<tr>
<td>mag_bernoulli_div_fac_ui</td>
<td>C function</td>
</tr>
<tr>
<td>mag_clear</td>
<td>C function</td>
</tr>
<tr>
<td>mag_cmp</td>
<td>C function</td>
</tr>
<tr>
<td>mag_cmp_2exp_si</td>
<td>C function</td>
</tr>
<tr>
<td>mag_div</td>
<td>C function</td>
</tr>
<tr>
<td>mag_div_fmpz</td>
<td>C function</td>
</tr>
<tr>
<td>mag_div_ui</td>
<td>C function</td>
</tr>
<tr>
<td>mag_equal</td>
<td>C function</td>
</tr>
<tr>
<td>mag_exp</td>
<td>C function</td>
</tr>
<tr>
<td>mag_exp_tail</td>
<td>C function</td>
</tr>
<tr>
<td>mag_expm1</td>
<td>C function</td>
</tr>
</tbody>
</table>
mag_fac_ui (C function), 25
mag_fast_add_2exp_si (C function), 24
mag_fast_addmul (C function), 24
mag_fast_init_set (C function), 24
mag_fast_init_set_arf (C function), 29
mag_fast_is_zero (C function), 24
mag_fast_mul (C function), 24
mag_fast_zero (C function), 24
mag_get_fmpz (C function), 23
mag_get_fmpr (C function), 23
mag_inf (C function), 22
mag_init (C function), 21
mag_init_set (C function), 21
mag_init_set_arf (C function), 29
mag_is_finite (C function), 22
mag_is_inf (C function), 22
mag_is_special (C function), 22
mag_is_zero (C function), 22
mag_log1p (C function), 24
mag_max (C function), 22
mag_min (C function), 22
mag_mul (C function), 23
mag_mul_2exp_fmpz (C function), 23
mag_mul_2exp_si (C function), 23
mag_mul_fmpz (C function), 23
mag_mul_fmpz_lower (C function), 24
mag_mul_lower (C function), 23
mag_mul_ui (C function), 23
mag_mul_ui_lower (C function), 23
mag_one (C function), 22
mag_pow_fmpz (C function), 24
mag_pow_ui (C function), 24
mag_pow_ui_lower (C function), 24
mag_print (C function), 22
mag_randtest (C function), 22
mag_randtest_special (C function), 22
mag_rfac_ui (C function), 25
mag_rsqrt (C function), 24
mag_set (C function), 21
mag_set_d (C function), 23
mag_set_d_2exp_fmpz (C function), 23
mag_set_fmpr (C function), 23
mag_set_fmpz (C function), 23
mag_set_fmpz_2exp_fmpz (C function), 23
mag_set_fmpz_2exp_fmpz_lower (C function), 23
mag_set_fmpz_lower (C function), 23
mag_set_ui (C function), 23
mag_set_ui_2exp_si (C function), 23
mag_set_ui_lower (C function), 23
mag_sqrt (C function), 24
mag_struct (C type), 21
mag_sub_lower (C function), 24
mag_swap (C function), 21
mag_t (C type), 21
mag_zero (C function), 22

P

partitions_fmpz_fmpz (C function), 101
partitions_fmpz_ui (C function), 101
partitions_fmpz_ui_using_doubles (C function), 101
partitions_hrr_sum_arb (C function), 101
partitions_rademacher_bound (C function), 101
psl2z_clear (C function), 91
psl2z_equal (C function), 92
psl2z_init (C function), 91
psl2z_inv (C function), 92
psl2z_is_correct (C function), 92
psl2z_is_one (C function), 91
psl2z_mul (C function), 92
psl2z_one (C function), 91
psl2z_print (C function), 91
psl2z_randtest (C function), 92
psl2z_set (C function), 91
psl2z_struct (C type), 91
psl2z_swap (C function), 91
psl2z_t (C type), 91