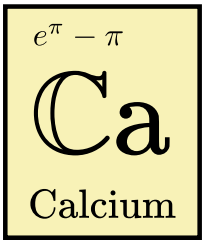

Calcium Documentation

Release 0.4.0

Fredrik Johansson

May 28, 2021

CONTENTS

Calcium (pronounced "kalkium") is a C library for exact computation with real and complex numbers. It is capable of rigorously deciding the truth of any constant relation involving algebraic numbers and many relations involving transcendental numbers, for example

$$\frac{\log(\sqrt{2}+\sqrt{3})}{\log(5+2\sqrt{6})} = \frac{1}{2}, \quad i^{\,i} = \exp\left(\frac{\pi}{\left(\left(\sqrt{-2}\right)^{\sqrt{2}}\right)^{\sqrt{2}}}\right), \quad 10^{-30} < \frac{640320^3+744}{e^{\pi\sqrt{163}}} - 1 < 10^{-29}.$$

Calcium is free software (LGPL). It depends on GMP, MPFR, Flint, Antic and Arb.

- Source code: `https://github.com/fredrik-johansson/calcium`
- Bug reports and feature requests: `https://github.com/fredrik-johansson/calcium/issues`
- Mailing list: flint-devel

This documentation is available in HTML format at `http://fredrikj.net/calcium/` and in PDF format at `http://fredrikj.net/calcium/calcium.pdf`. This edition of the documentation was updated May 28, 2021 and describes Calcium 0.4.0.

CHAPTER

ONE

TUTORIAL

For new users, this introductory Jupyter notebook is a good place to start.

CHAPTER

TWO

GENERAL INFORMATION

2.1 Introduction

2.1.1 Exact numbers in Calcium

The core idea behind Calcium is to represent real and complex numbers as elements of extension fields

$$\mathbb{Q}(a_1, \ldots, a_n)$$

of the rational numbers, where the extension numbers a_k are described by symbolic expressions (which may depend on other fields recursively). The system constructs such fields automatically as needed to represent the results of computations. Any extension field is isomorphic to a formal field

$$\mathbb{Q}(a_1, \ldots, a_n) \quad \cong \quad K_{\text{formal}} := \text{Frac}(\mathbb{Q}[X_1, \ldots, X_n]/I)$$

where I is the ideal of algebraic relations among the extension numbers. The relations may involve algebraic numbers (for example: $i^2 + 1 = 0$), transcendental numbers (for example: $e^{-\pi} \cdot e^{\pi} = 1$), or combinations thereof.

Computation in the formal field depends (in general) on multivariate polynomial arithmetic together with use of a Gröbner basis for the ideal. The map from the formal field to the true complex field is maintained using arbitrary-precision ball arithmetic where necessary.

As an important special case, Calcium can be used for arithmetic in algebraic number fields (embedded explicitly in $\mathbb{C}$)

$$\mathbb{Q}(a) \cong \mathbb{Q}[X]/\langle f(X) \rangle$$

with excellent performance thanks to internal use of the Antic library.

It will not always work perfectly: although Calcium by design should never give a mathematically erroneous answer, it may be unable to simplify a result as much as expected and it may be unable to decide a predicate (in which case it can return "Unknown"). Equality is at least decidable over the algebraic numbers $\overline{\mathbb{Q}}$ (for practical degrees and bit sizes of the numbers!), and in certain cases involving transcendentals. We hope to improve Calcium's capabilities gradually through enhancements to its built-in algorithms and through customization options.

Usage details

To understand how Calcium works more concretely, see *Example programs* and the documentation for the main Calcium number type (`ca_t`):

- *ca.h – exact real and complex numbers*

Implementation details for extension numbers and formal fields can be found in the documentation of the corresponding modules:

- *ca_ext.h – real and complex extension numbers*

- *ca_field.h – extension fields*

The following modules are used internally for arithmetic in transcendental number fields (rational function fields) $\mathbb{Q}(x_1, \ldots, x_n)$ and over the field of algebraic numbers $\overline{\mathbb{Q}}$, respectively. They may be of independent interest:

- *fmpz_mpoly_q.h – multivariate rational functions over Q*
- *qqbar.h – algebraic numbers represented by minimal polynomials*

2.1.2 FAQ

Isn't x = 0 undecidable?

In general, yes: equality over the reals is undecidable. In practice, much of calculus and elementary number theory can be done with numbers that are simple algebraic combinations of well-known elementary and special functions, and there are heuristics that work quite well for deciding predicates about such numbers. Calcium will be able to give a definitive answer at least in simple cases (for example, proving $16\operatorname{atan}(\frac{1}{5}) - 4\operatorname{atan}(\frac{1}{239}) = \pi$ or $\sqrt{5+2\sqrt{6}} = \sqrt{2}+\sqrt{3}$), and will simply answer "Unknown" when its heuristics are not powerful enough.

How does Calcium compare to ordinary numerical computing?

Calcium is far too slow to replace floating-point numbers for 99.93% of scientific computing. The target is symbolic and algebraic computation. Nevertheless, Calcium may well be useful as a tool to test and enhance the capabilities of numerical programs.

How does Calcium compare to Arb arithmetic?

The main advantage of Calcium over ball arithmetic alone is the ability to do exact comparisons. The automatic precision management in Calcium can also be convenient.

Calcium will usually be slower than Arb arithmetic. If a computation is mostly numerical, it is probably better to try using Arb first, and fall back on an exact calculation with Calcium only if that fails because an exact comparison is needed.

How does Calcium compare to symbolic computation systems (Mathematica, SymPy, etc.)?

Calculating with constant values is only a small part of what such systems have to do, but it is one of the most complex parts. Existing computer algebra systems sometimes manage this very well, and sometimes fail horribly. The most common problems are 1) getting numerical error bounds or branch cuts wrong, and 2) slowing down too much when the expressions get large. Calcium is intended to address both problems (through rigorous numerical evaluation and use of fast polynomial arithmetic).

Ultimately, Calcium will no doubt handle some problems better and others worse, and it should be considered a complement to existing computer algebra systems rather than a replacement. A symbolic expression simplifier may use Calcium evaluation as one of its tools, but this probably needs to be done selectively and in combination with many other heuristics.

Why is Calcium written in C?

The main advantage of developing Calcium as a C library is that it will not be tied to a particular programming language ecosystem: C is uniquely easy to interface from almost any other language. The second most important reason is familiarity: Calcium follows the design of Flint and Arb (coding style, naming, module layout, memory management, test code, etc.) which has proved to work quite well for libraries of this type.

There is also the performance argument. Some core functions will benefit from optimizations that are natural in C such as in-place operations and fine-grained manual memory management. However, the performance aspect should not be overemphasized: Calcium will spend most of its time in Flint and Arb kernel functions and this would probably still be true even if it were written in a slower language.

There are certainly types of mathematical functionality that will be too inconvenient to implement in C. Our intention is indeed to leave such functionality to projects written in Python, Julia, etc. which may then opt to depend on Calcium for basic operations.

What is the development status of Calcium?

Calcium is presently in early development and should be considered experimental software. The interfaces are subject to change and many important functions and optimizations have not been implemented. A more stable and functional release can be expected in late 2021.

2.2 Setup

2.2.1 Installation

Calcium has the following dependencies:

- FLINT (http://www.flintlib.org) and its dependencies (GMP/MPIR and MPFR). Calcium will require FLINT 2.7 (unreleased) or later; currently a git checkout of https://github.com/wbhart/flint2 is necessary.
- Arb (http://arblib.org) version 2.18.1 or later.
- Antic (https://github.com/wbhart/antic/) - use a git checkout.

To compile, test and install Calcium from source assuming that all dependencies have been installed, run the following commands in the source directory:

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

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

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

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

2.2.2 Running tests

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

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

```
export CALCIUM_TEST_MULTIPLIER=0.1
make check
```

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

```
make check MOD=ca
```

2.2.3 Running code

Here is an example program to get started using Calcium:

```
#include "calcium/ca.h"

int main()
{
    ca_ctx_t ctx;
    ca_t x;
```

(continues on next page)

(continued from previous page)

```
    ca_ctx_init(ctx);
    ca_init(x, ctx);

    ca_pi(x, ctx);                /* x = pi */
    ca_sub_ui(x, x, 3, ctx);      /* x = x - 3 */
    ca_pow_ui(x, x, 2, ctx);      /* x = x^2 */
    ca_print(x, ctx); printf("\n");
    printf("Computed with calcium-%s\n", calcium_version());

    ca_clear(x, ctx);
    ca_ctx_clear(ctx);
    flint_cleanup();
    return EXIT_SUCCESS;
}
```

Compile it with:

```
gcc test.c -lcalcium
```

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

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

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

Finally, to run the program, make sure that the linker can find the 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:

```
0.0200485 {a^2-6*a+9 where a = 3.14159 [Pi]}
Computed with calcium-0.0.0
```

2.3 Example programs

The *examples* directory (https://github.com/fredrik-johansson/calcium/tree/master/examples) contains complete C programs illustrating use of Calcium. Running:

```
make examples
```

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

2.3.1 elementary.c

This program evaluates several elementary expressions. For some inputs, Calcium's arithmetic should produce a simplified result automatically. Some inputs do not yet automatically simplify as much as one might hope. Calcium may still able to prove that such a number is zero or nonzero; the output of *ca_check_is_zero()* is then `T_TRUE` or `T_FALSE`.

Sample output:

```
> build/examples/elementary
>>> Exp(Pi*I) + 1
0

>>> Log(-1) / (Pi*I)
1

>>> Log(-I) / (Pi*I)
-0.500000 {-1/2}

>>> Log(1 / 10^123) / Log(100)
-61.5000 {-123/2}

>>> Log(1 + Sqrt(2)) / Log(3 + 2*Sqrt(2))
0.500000 {1/2}

>>> Sqrt(2)*Sqrt(3) - Sqrt(6)
0

>>> Exp(1+Sqrt(2)) * Exp(1-Sqrt(2)) / (Exp(1)^2)
1

>>> I^I - Exp(-Pi/2)
0

>>> Exp(Sqrt(3))^2 - Exp(Sqrt(12))
0

>>> 2*Log(Pi*I) - 4*Log(Sqrt(Pi)) - Pi*I
0

>>> -I*Pi/8*Log(2/3-2*I/3)^2 + I*Pi/8*Log(2/3+2*I/3)^2 + Pi^2/12*Log(-1-I) + Pi^2/12*Log(-1+I)␣
↪+ Pi^2/12*Log(1/3-I/3) + Pi^2/12*Log(1/3+I/3) - Pi^2/48*Log(18)
0

>>> Sqrt(5 + 2*Sqrt(6)) - Sqrt(2) - Sqrt(3)
0e-1126 {a-c-d where a = 3.14626 [Sqrt(9.89898 {2*b+5})], b = 2.44949 [b^2-6=0], c = 1.73205␣
↪[c^2-3=0], d = 1.41421 [d^2-2=0]}
>>> Is zero?
T_TRUE

>>> Sqrt(I) - (1+I)/Sqrt(2)
```

(continues on next page)

(continued from previous page)

```
0e-1126 + 0e-1126*I {(2*a-b*c-b)/2 where a = 0.707107 + 0.707107*I [Sqrt(1.00000*I {c})], b =␣
↪1.41421 [b^2-2=0], c = I [c^2+1=0]}
>>> Is zero?
T_TRUE

>>> Exp(Pi*Sqrt(163)) - (640320^3 + 744)
-7.49927e-13 {a-262537412640768744 where a = 2.62537e+17 [Exp(40.1092 {b*c})], b = 3.14159␣
↪[Pi], c = 12.7671 [c^2-163=0]}

>>> Erf(2*Log(Sqrt(1/2-Sqrt(2)/4))+Log(4)) - Erf(Log(2-Sqrt(2)))
0


cpu/wall(s): 0.022 0.022
virt/peak/res/peak(MB): 36.45 36.47 9.37 9.37
```

2.3.2 binet.c

This program computes the n-th Fibonacci number using Binet's formula $F_n = (\varphi^n - (1-\varphi)^n)/\sqrt{5}$ where $\varphi = \frac{1}{2}(1+\sqrt{5})$. The program takes n as input.

Sample output:

```
> build/examples/binet 250
7.89633e+51 {7896325826131730509282738943634332893686268675876375}

cpu/wall(s): 0.002 0.001
virt/peak/res/peak(MB): 36.14 36.14 5.81 5.81
```

This illustrates exact arithmetic in algebraic number fields. The program also illustrates another aspect of Calcium arithmetic: evaluation limits. For example, trying to compute the index $n = 10^6$ Fibonacci number hits an evaluation limit, so the value is not expanded to an explicit integer:

```
> build/examples/binet 1000000
1.95328e+208987 {(a*c-b*c)/5 where a = 4.36767e+208987 [Pow(1.61803 {(c+1)/2}, 1.00000e+6
↪{1000000})], b = 2.28955e-208988 [Pow(-0.618034 {(-c+1)/2}, 1.00000e+6 {1000000})], c = 2.
↪23607 [c^2-5=0]}

cpu/wall(s): 0.006 0.005
virt/peak/res/peak(MB): 36.14 36.14 9.05 9.05
```

Calling the program with `-limit B n` raises the bit evaluation limit to B. Setting this large enough allows F_{10^6} to expand to an integer (the following output has been truncated to avoid reproducing all 208988 digits):

```
> build/examples/binet -limit 10000000 1000000
1.95328e+208987 {1953282128...8242546875}

cpu/wall(s): 0.229 0.242
virt/peak/res/peak(MB): 36.79 37.29 7.13 7.13
```

The exact mechanisms and interfaces for evaluation limits are still a work in progress.

2.3.3 machin.c

This program checks several variations of Machin's formula

$$\frac{\pi}{4} = 4\,\mathrm{atan}\left(\frac{1}{5}\right) - \mathrm{atan}\left(\frac{1}{239}\right)$$

expressing π or logarithms of small integers in terms of arctangents or hyperbolic arctangents of rational numbers. The program actually evaluates $4\,\mathrm{atan}\left(\frac{1}{5}\right) - \mathrm{atan}\left(\frac{1}{239}\right) - \frac{\pi}{4}$ (etc.) and prints the result, which should be precisely 0, proving the identity. Inverse trigonometric functions are not yet implemented in Calcium, so the example program evaluates them using logarithms.

Sample output:

```
> build/examples/machin
[(1)*Atan(1/1) - Pi/4]   =   0
[(1)*Atan(1/2) + (1)*Atan(1/3) - Pi/4]   =   0
[(2)*Atan(1/2) + (-1)*Atan(1/7) - Pi/4]   =   0
[(2)*Atan(1/3) + (1)*Atan(1/7) - Pi/4]   =   0
[(4)*Atan(1/5) + (-1)*Atan(1/239) - Pi/4]   =   0
[(1)*Atan(1/2) + (1)*Atan(1/5) + (1)*Atan(1/8) - Pi/4]   =   0
[(1)*Atan(1/3) + (1)*Atan(1/4) + (1)*Atan(1/7) + (1)*Atan(1/13) - Pi/4]   =   0
[(12)*Atan(1/49) + (32)*Atan(1/57) + (-5)*Atan(1/239) + (12)*Atan(1/110443) - Pi/4]   =   0

[(14)*Atanh(1/31) + (10)*Atanh(1/49) + (6)*Atanh(1/161) - Log(2)]   =   0
[(22)*Atanh(1/31) + (16)*Atanh(1/49) + (10)*Atanh(1/161) - Log(3)]   =   0
[(32)*Atanh(1/31) + (24)*Atanh(1/49) + (14)*Atanh(1/161) - Log(5)]   =   0
[(144)*Atanh(1/251) + (54)*Atanh(1/449) + (-38)*Atanh(1/4801) + (62)*Atanh(1/8749) - Log(2)]␣
↪=   0
[(228)*Atanh(1/251) + (86)*Atanh(1/449) + (-60)*Atanh(1/4801) + (98)*Atanh(1/8749) - Log(3)]␣
↪=   0
[(334)*Atanh(1/251) + (126)*Atanh(1/449) + (-88)*Atanh(1/4801) + (144)*Atanh(1/8749) - Log(5)]␣
↪   =   0
[(404)*Atanh(1/251) + (152)*Atanh(1/449) + (-106)*Atanh(1/4801) + (174)*Atanh(1/8749) -␣
↪Log(7)]   =   0

cpu/wall(s): 0.016 0.016
virt/peak/res/peak(MB): 35.57 35.57 8.80 8.80
```

2.3.4 swinnerton_dyer_poly.c

This program computes the coefficients of the Swinnerton-Dyer polynomial

$$S_n = \prod (x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \ldots \pm \sqrt{p_n})$$

where p_n denotes the n-th prime number and all combinations of signs are taken. This polynomial has degree 2^n. The polynomial is expanded from its roots using naive polynomial multiplication over *ca_t* coefficients. There are far more efficient ways to construct this polynomial; this program simply illustrates that arithmetic in multivariate number fields works smoothly.

The program prints the coefficients of S_n, from the constant term to the coefficient of x^{2^n}.

Sample output:

```
> build/examples/swinnerton_dyer_poly 3
576
0
-960
0
352
0
```

(continues on next page)

(continued from previous page)

```
-40
0
1

cpu/wall(s): 0.002 0.002
virt/peak/res/peak(MB): 35.07 35.11 5.40 5.40
```

A big benchmark problem (output truncated):

```
> build/examples/swinnerton_dyer_poly 10
4.35675e+809 {43567450015...212890625}
0
...
0
1

cpu/wall(s): 9.296 9.307
virt/peak/res/peak(MB): 38.95 38.95 10.01 10.01
```

2.3.5 huge_expr.c

This program proves equality of two complicated algebraic numbers. More precisely, the program verifies that $N = -(1 - |M|^2)^2$ where N and M are given by huge symbolic expressions involving nested square roots (about 7000 operations in total).

By default, the program runs the computation using *qqbar_t* arithmetic:

```
> build/examples/huge_expr
Evaluating N...
cpu/wall(s): 7.205 7.206
Evaluating M...
cpu/wall(s): 0.933 0.934
Evaluating E = -(1-|M|^2)^2...
cpu/wall(s): 0.391 0.391
N ~ -0.161908530533112036958428699914585782034736456606 41
E ~ -0.161908530533112036958428699914585782034736456606 41
Testing E = N...
cpu/wall(s): 0.001 0

Equal = T_TRUE

Total: cpu/wall(s): 8.53 8.531
virt/peak/res/peak(MB): 54.50 64.56 24.64 34.61
```

To run the computation using *ca_t* arithmetic instead, pass the -ca flag:

```
> build/examples/huge_expr -ca
Evaluating N...
cpu/wall(s): 0.193 0.193
Evaluating M...
cpu/wall(s): 0.024 0.024
Evaluating E = -(1-|M|^2)^2...
cpu/wall(s): 0.008 0.009
N ~ -0.161908530533112036958428699914585782034736456606 41
E ~ -0.161908530533112036958428699914585782034736456606 41
Testing E = N...
cpu/wall(s): 8.017 8.019

Equal = T_TRUE
```

(continues on next page)

(continued from previous page)

```
Total: cpu/wall(s): 8.243 8.246
virt/peak/res/peak(MB): 61.67 65.29 33.97 37.54
```

This simplification problem was posted in a help request for Sage (https://ask.sagemath.org/question/52653). The C code has been generated from the symbolic expressions using a Python script.

2.3.6 hilbert_matrix.c

This program constructs the Hilbert matrix $H_n = (1/(i+j-1))_{i=1,j=1}^{n}$, computes its eigenvalues $\lambda_1, \ldots, \lambda_n$, as exact algebraic numbers, and verifies the exact trace and determinant formulas

$$\lambda_1 + \lambda_2 + \ldots + \lambda_n = \operatorname{tr}(H_n), \quad \lambda_1 \lambda_2 \cdots \lambda_n = \det(H_n).$$

Sample output:

```
> build/examples/hilbert_matrix 6
Trace:
1.87821 {6508/3465}
1.87821 {6508/3465}
Equal: T_TRUE

Det:
5.36730e-18 {1/186313420339200000}
5.36730e-18 {1/186313420339200000}
Equal: T_TRUE


cpu/wall(s): 0.07 0.069
virt/peak/res/peak(MB): 36.56 36.66 9.69 9.69
```

The program accepts the following optional arguments:

- With `-vieta`, force use of Vieta's formula internally (by default, Calcium uses Vieta's formulas when working with algebraic conjugates, but only up to some bound on the degree).
- With `-novieta`, force Calcium not to use Vieta's formulas internally.
- With `-qqbar`, do a similar computation using *qqbar_t* arithmetic.

2.3.7 dft.c

This program demonstrates the discrete Fourier transform (DFT) in exact arithmetic. For the input vector $\mathbf{x} = (x_n)_{n=0}^{N-1}$, it verifies the identity

$$\mathbf{x} - \mathrm{DFT}^{-1}(\mathrm{DFT}(\mathbf{x})) = 0$$

where

$$\mathrm{DFT}(\mathbf{x})_n = \sum_{k=0}^{N-1} \omega^{-kn} x_k, \quad \mathrm{DFT}^{-1}(\mathbf{x})_n = \frac{1}{N} \sum_{k=0}^{N-1} \omega^{kn} x_k, \quad \omega = e^{2\pi i/N}.$$

The program computes the DFT by naive $O(N^2)$ summation (not using FFT). It uses repeated multiplication of ω to precompute an array of roots of unity $1, \omega, \omega^2, \ldots, \omega^{2N-1}$ for use in both the DFT and the inverse DFT.

Usage:

```
build/examples/dft [-verbose] [-input i] [-limit B] [-timing T] N
```

The required parameter `N` selects the length of the vector.

The optional flag `-verbose` chooses whether to print the arrays.

The optional parameter `-timing T` selects a timing method (default = 0).

- 0: run the computation once and time it
- 1: run the computation repeatedly if needed to get an accurate timing, creating a new context object for each iteration so that fields are not cached
- 2: run the computation once, then run the computation at least one more time (repeatedly if needed to get an accurate timing), recycling the same context object to measure the performance with cached fields

The optional parameter `-input i` selects an input sequence (default = 0).

- 0: $x_n = n + 2$
- 1: $x_n = \sqrt{n+2}$
- 2: $x_n = \log(n+2)$
- 3: $x_n = e^{2\pi i/(n+2)}$

The optional parameter `-limit B` sets the internal degree limit for algebraic numbers.

Sample output:

```
> build/examples/dft 4 -input 1 -verbose
DFT benchmark, length N = 4

[x] =
1.41421 {a where a = 1.41421 [a^2-2=0]}
1.73205 {a where a = 1.73205 [a^2-3=0]}
2
2.23607 {a where a = 2.23607 [a^2-5=0]}

DFT([x]) =
7.38233 {a+b+c+2 where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0]}
-0.585786 + 0.504017*I {a*d-b*d+c-2 where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.
↪41421 [c^2-2=0], d = I [d^2+1=0]}
-0.553905 {-a-b+c+2 where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0]}
-0.585786 - 0.504017*I {-a*d+b*d+c-2 where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.
↪41421 [c^2-2=0], d = I [d^2+1=0]}

IDFT(DFT([x])) =
1.41421 {c where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0], d = I [d^
↪2+1=0]}
1.73205 {b where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0], d = I [d^
↪2+1=0]}
2
2.23607 {a where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0], d = I [d^
↪2+1=0]}

[x] - IDFT(DFT([x])) =
0       (= 0   T_TRUE)
0       (= 0   T_TRUE)
0       (= 0   T_TRUE)
0       (= 0   T_TRUE)

cpu/wall(s): 0.009 0.009
virt/peak/res/peak(MB): 36.28 36.28 9.14 9.14
```

CHAPTER

THREE

GENERAL MODULES

3.1 calcium.h – global definitions

3.1.1 Version

const char ***calcium_version**(void)
Returns a pointer to the version of the library as a string `X.Y.Z`.

3.1.2 Test code

double **calcium_test_multiplier**(void)
Multiplier for the number of iterations to run in each unit test. The value can be changed by setting the environment variable `CALCIUM_TEST_MULTIPLIER`. The default value is 1.0.

3.1.3 Triple-valued logic

This library uses two kinds of predicate functions:

- Predicates with signature `int foo_is_X(const foo_t x)` return the usual C boolean values `1` for true and `0` for false, unless otherwise documented. Some functions may return `0` also when truth cannot be certified (this will be documented explicitly).
- Predicates with signature `truth_t foo_check_is_X(const foo_t x)` check a mathematical property that may not be decidable (or may be too costly to decide). The return value is a *truth_t* (`T_TRUE`, `T_FALSE` or `T_UNKNOWN`).

enum truth_t
Represents one of the following truth values:

T_TRUE

T_FALSE

T_UNKNOWN

Warning: the constants `T_TRUE` and `T_FALSE` do not correspond to 1 and 0. It is erroneous to write, for example `!t` if `t` is a *truth_t*. One should instead write `t != T_TRUE`, `t == T_FALSE`, etc. depending on whether the unknown case should be included or excluded.

3.1.4 Flint, Arb and Antic types

The following types from Flint, Arb and Antic are used throughout Calcium. Although not included automatically by `calcium.h`, we document them here for convenience.

type slong
Signed full-word integer (64 bits on a 64-bit system).

type ulong
Unsigned full-word integer (64 bits on a 64-bit system).

type fmpz_t
Flint integer.

type fmpq_t
Flint rational number.

type fmpz_poly_t
Flint dense univariate polynomial over the integers.

type fmpq_poly_t
Flint dense univariate polynomial over the rational numbers.

type fmpz_mpoly_t
Flint sparse multivariate integer polynomial.

type fmpz_mpoly_ctx_t
Context for Flint sparse multivariate integer polynomial (defining the number of variables and monomial order).

type fmpz_mat_t
Flint dense matrix over the integers.

type fmpq_mat_t
Flint dense matrix over the rational numbers.

type arb_t
Arb real number.

type acb_t
Arb complex number.

type nf_t
Antic number field.

type nf_elem_t
Antic number field element.

3.1.5 Flint, Arb and Antic extras

Here we collect various utility methods for Flint, Arb and Antic types that are missing in those libraries. Some of these functions may be migrated upstream in the future.

ulong **calcium_fmpz_hash(const** *fmpz_t x***)**
Hash function for integers. The algorithm may change; presently, this simply extracts the low word (with sign).

3.1.6 Input and output

type calcium_stream_struct

type calcium_stream_t
A stream object which can hold either a file pointer or a string (with automatic resizing).

void **calcium_stream_init_file**(*calcium_stream_t out*, FILE **fp*)
Initializes the stream *out* for writing to the file *fp*. The file can be *stdout*, *stderr*, or any file opened for writing by the user.

void **calcium_stream_init_str**(*calcium_stream_t out*)
Initializes the stream *out* for writing to a string in memory. When finished, the user should free the string (the *s* member of *out* with `flint_free()`).

void **calcium_write**(*calcium_stream_t out*, **const** char **s*)
Writes the string *s* to *out*.

void **calcium_write_free**(*calcium_stream_t out*, char **s*)
Writes *s* to *out* and then frees *s* by calling `flint_free()`.

void **calcium_write_si**(*calcium_stream_t out*, *slong x*)

void **calcium_write_fmpz**(*calcium_stream_t out*, **const** *fmpz_t x*)
Writes the integer *x* to *out*.

void **calcium_write_arb**(*calcium_stream_t out*, **const** *arb_t z*, *slong digits*, *ulong flags*)

void **calcium_write_acb**(*calcium_stream_t out*, **const** *acb_t z*, *slong digits*, *ulong flags*)
Writes the Arb number *z* to *out*, showing *digits* digits and with the display style specified by *flags* (`ARB_STR_NO_RADIUS`, etc.).

CHAPTER

FOUR

NUMBERS

4.1 ca.h – exact real and complex numbers

A `ca_t` represents a real or complex number in a form suitable for exact field arithmetic or comparison. Exceptionally, a `ca_t` may represent a special nonnumerical value, such as an infinity.

4.1.1 Introduction: numbers

A *Calcium number* is a real or complex number represented as an element of a formal field $K = \mathbb{Q}(a_1, \ldots, a_n)$ where the symbols a_k denote fixed algebraic or transcendental numbers called *extension numbers*. For example, $e^{-2\pi} - 3i$ may be represented as $(1 - 3a_2^2 a_1)/a_2^2$ in the field $\mathbb{Q}(a_1, a_2)$ with $a_1 = i, a_2 = e^{\pi}$. Extension numbers and fields are documented in the following separate modules:

- *ca_ext.h – real and complex extension numbers*
- *ca_field.h – extension fields*

The user does not need to construct extension numbers or formal extension fields explicitly: each `ca_t` contains an internal pointer to its formal field, and operations on Calcium numbers generate and cache fields automatically as needed to express the results.

This representation is not canonical (in general). A given complex number can be represented in different ways depending on the choice of formal field K. Even within a fixed field K, a number can have different representations if there are algebraic relations between the extension numbers. Two numbers x and y can be tested for inequality using numerical evaluation; to test for equality, it may be necessary to eliminate dependencies between extension numbers. One of the central goals of Calcium will be to implement heuristics for such elimination.

Together with each formal field K, Calcium stores a *reduction ideal* $I = \{g_1, \ldots, g_m\}$ with $g_i \in \mathbb{Z}[a_1, \ldots, a_n]$, defining a set of algebraic relations $g_i(a_1, \ldots, a_n) = 0$. Relations can be absolute, say $g_i = a_j^2 + 1$, or relative, say $g_i = 2a_j - 4a_k - a_l a_m$. The reduction ideal effectively partitions K into equivalence classes of complex numbers (e.g. $i^2 = -1$ or $2\log(\pi i) = 4\log(\sqrt{\pi}) + \pi i$), enabling simplifications and equality proving.

Extension numbers are always sorted $a_1 \succ a_2 \succ \ldots \succ a_n$ where $\succ$ denotes a structural ordering (see `ca_cmp_repr()`). If the reduction ideal is triangular and the multivariate polynomial arithmetic uses lexicographic ordering, reduction by I eliminates numbers a_i with higher complexity in the sense of $\succ$.

The reduction ideal is an imperfect computational crutch: it is not guaranteed to capture *all* algebraic relations, and reduction is not guaranteed to produce uniquely defined representatives. However, in the specific case of an absolute number field $K = \mathbb{Q}(a)$ where a is a `qqbar_t` extension, the reduction ideal (consisting of a single minimal polynomial) is canonical and field elements of K can be chosen canonically.

4.1.2 Introduction: special values

In order to provide a closed arithmetic system and express limiting cases of operators and special functions, a *ca_t* can hold any of the following special values besides ordinary numbers:

- *Unsigned infinity*, a formal object $\tilde{\infty}$ representing the value of $1/0$. More generally, this is the value of meromorphic functions at poles.
- *Signed infinity*, a formal object $a \cdot \infty$ where the sign a is a Calcium number with $|a| = 1$. The most common values are $+\infty, -\infty, +i\infty, -i\infty$. Signed infinities are used to denote directional limits and logarithmic singularities (for example, $\log(0) = -\infty$).
- *Undefined*, a formal object representing the value of indeterminate forms such as $0/0$ and essential singularities such as $\exp(\tilde{\infty})$, where a number or infinity would not make sense as an answer.
- *Unknown*, a meta-value used to signal that the actual desired value could not be computed, either because Calcium does not (yet) have a data structure or algorithm for that case, or because doing so would be unreasonably expensive. This occurs, for example, if Calcium performs a division and is unable to decide whether the result is a number, unsigned infinity or undefined (because testing for zero fails). Wrappers may want to check output variables for *Unknown* and throw an exception (e.g. *NotImplementedError* in Python).

The distinction between *Calcium numbers* (which must represent elements of $\mathbb{C}$) and the different kinds of nonnumerical values (infinities, Undefined or Unknown) is essential. Nonnumerical values may not be used as field extension numbers a_k, and the denominator of a formal field element must always represent a nonzero complex number. Accordingly, for any given Calcium value x that is not *Unknown*, it is exactly known whether x represents A) a number, B) unsigned infinity, C) a signed infinity, or D) Undefined.

4.1.3 Number objects

For all types, a *type_t* is defined as an array of length one of type *type_struct*, permitting a *type_t* to be passed by reference.

type ca_struct

type ca_t
A *ca_t* contains an index to a field K, and data representing an element x of K. The data is either an inline rational number (*fmpq_t*), an inline Antic number field element (*nf_elem_t*) when K is an absolute algebraic number field $\mathbb{Q}(a)$, or a pointer to a heap-allocated *fmpz_mpoly_q_t* representing an element of a generic field $\mathbb{Q}(a_1, \ldots, a_n)$. Special values are encoded using magic bits in the field index.

type ca_ptr
Alias for `ca_struct *`, used for vectors of numbers.

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

4.1.4 Context objects

type ca_ctx_struct

type ca_ctx_t
A *ca_ctx_t* context object holds a cache of fields K and constituent extension numbers a_k. The field index in an individual *ca_t* instance represents a shallow reference to the object defining the field K within the context object, so creating many elements of the same field is cheap.

Since context objects are mutable (and may be mutated even when performing read-only operations on *ca_t* instances), they must not be accessed simultaneously by different threads: in multithreaded environments, the user must use a separate context object for each thread.

void **ca_ctx_init**(*ca_ctx_t ctx*)
Initializes the context object *ctx* for use. Any evaluation options stored in the context object are set to default values.

void **ca_ctx_clear**(*ca_ctx_t ctx*)
Clears the context object *ctx*, freeing any memory allocated internally. This function should only be called after all *ca_t* instances referring to this context have been cleared.

void **ca_ctx_print**(**const** *ca_ctx_t ctx*)
Prints a description of the context *ctx* to standard output. This will give a complete listing of the cached fields in *ctx*.

4.1.5 Memory management for numbers

void **ca_init**(*ca_t x*, *ca_ctx_t ctx*)
Initializes the variable x for use, associating it with the context object *ctx*. The value of x is set to the rational number 0.

void **ca_clear**(*ca_t x*, *ca_ctx_t ctx*)
Clears the variable x.

void **ca_swap**(*ca_t x*, *ca_t y*, *ca_ctx_t ctx*)
Efficiently swaps the variables x and y.

4.1.6 Symbolic expressions

void **ca_get_fexpr**(*fexpr_t res*, **const** *ca_t x*, *ulong flags*, *ca_ctx_t ctx*)
Sets *res* to a symbolic expression representing x.

int **ca_set_fexpr**(*ca_t res*, **const** *fexpr_t expr*, *ca_ctx_t ctx*)
Sets *res* to the value represented by the symbolic expression *expr*. Returns 1 on success and 0 on failure. This function essentially just traverses the expression tree using `ca` arithmetic; it does not provide advanced symbolic evaluation. It is guaranteed to at least be able to parse the output of *ca_get_fexpr()*.

4.1.7 Printing

The style of printed output is controlled by `ctx->options[CA_OPT_PRINT_FLAGS]` (see *Context options*) which can be set to any combination of the following flags:

`CA_PRINT_N`
Print a decimal approximation of the number. The approximation is guaranteed to be correctly rounded to within one unit in the last place.

If combined with `CA_PRINT_REPR`, numbers appearing within the symbolic representation will also be printed with decimal approximations.

Warning: printing a decimal approximation requires a computation, which can be expensive. It can also mutate cached data (numerical enclosures of extension numbers), affecting subsequent computations.

`CA_PRINT_DIGITS`
Multiplied by a positive integer, specifies the number of decimal digits to show with `CA_PRINT_N`. If not given, the default precision is six digits.

`CA_PRINT_REPR`
Print the symbolic representation of the number (including its recursive elements). If used together with `CA_PRINT_N`, field elements will print as `decimal {symbolic}` while extension numbers will print as `decimal [symbolic]`.

All extension numbers appearing in the field defining `x` and in the inner constructions of those extension numbers will be given local labels `a`, `b`, etc. for this printing.

`CA_PRINT_FIELD`
: For each field element, explicitly print its formal field along with its reduction ideal if present, e.g. `QQ` or `QQ(a,b,c) / <a-b, c^2+1>`.

`CA_PRINT_DEFAULT`
: The default print style. Equivalent to `CA_PRINT_N | CA_PRINT_REPR`.

`CA_PRINT_DEBUG`
: Verbose print style for debugging. Equivalent to `CA_PRINT_N | CA_PRINT_REPR | CA_PRINT_FIELD`.

As a special case, small integers are always printed as simple literals.

As illustration, here are the numbers -7, $2/3$, $(\sqrt{3}+5)/2$ and $\sqrt{2}(\log(\pi)+\pi i)$ printed in various styles:

```
# CA_PRINT_DEFAULT
-7
0.666667 {2/3}
3.36603 {(a+5)/2 where a = 1.73205 [a^2-3=0]}
1.61889 + 4.44288*I {a*c+b*c*d where a = 1.14473 [Log(3.14159 {b})], b = 3.14159 [Pi], c = 1.
↪41421 [c^2-2=0], d = I [d^2+1=0]}

# CA_PRINT_N
-7
0.666667
3.36603
1.61889 + 4.44288*I

# CA_PRINT_N | (CA_PRINT_DIGITS * 20)
-7
0.66666666666666666667
3.3660254037844386468
1.6188925298220266685 + 4.4428829381583662470*I

# CA_PRINT_REPR
-7
2/3
(a+5)/2 where a = [a^2-3=0]
a*c+b*c*d where a = Log(b), b = Pi, c = [c^2-2=0], d = [d^2+1=0]

# CA_PRINT_DEBUG
-7
0.666667 {2/3  in  QQ}
3.36603 {(a+5)/2  in  QQ(a)/<a^2-3> where a = 1.73205 [a^2-3=0]}
1.61889 + 4.44288*I {a*c+b*c*d  in  QQ(a,b,c,d)/<c^2-2, d^2+1> where a = 1.14473 [Log(3.14159
↪{b  in  QQ(b)})], b = 3.14159 [Pi], c = 1.41421 [c^2-2=0], d = I [d^2+1=0]}
```

void **ca_print**(**const** *ca_t x*, **const** *ca_ctx_t ctx*)
: Prints x to standard output.

void **ca_fprint**(FILE **fp*, **const** *ca_t x*, **const** *ca_ctx_t ctx*)
: Prints x to the file *fp*.

char ***ca_get_str**(**const** *ca_t x*, **const** *ca_ctx_t ctx*)
: Prints x to a string which is returned. The user should free this string by calling `flint_free`.

void **ca_printn**(**const** *ca_t x*, *slong n*, **const** *ca_ctx_t ctx*)
: Prints an n-digit numerical representation of x to standard output.

4.1.8 Special values

void **ca_zero**(*ca_t res*, *ca_ctx_t ctx*)

void **ca_one**(*ca_t res*, *ca_ctx_t ctx*)

void **ca_neg_one**(*ca_t res*, *ca_ctx_t ctx*)
Sets *res* to the integer 0, 1 or -1. This creates a canonical representation of this number as an element of the trivial field $\mathbb{Q}$.

void **ca_i**(*ca_t res*, *ca_ctx_t ctx*)

void **ca_neg_i**(*ca_t res*, *ca_ctx_t ctx*)
Sets *res* to the imaginary unit $i = \sqrt{-1}$, or its negation $-i$. This creates a canonical representation of i as the generator of the algebraic number field $\mathbb{Q}(i)$.

void **ca_pi**(*ca_t res*, *ca_ctx_t ctx*)
Sets *res* to the constant π. This creates an element of the transcendental number field $\mathbb{Q}(\pi)$.

void **ca_pi_i**(*ca_t res*, *ca_ctx_t ctx*)
Sets *res* to the constant πi. This creates an element of the composite field $\mathbb{Q}(i, \pi)$ rather than representing πi (or even $2\pi i$, which for some purposes would be more elegant) as an atomic quantity.

void **ca_euler**(*ca_t res*, *ca_ctx_t ctx*)
Sets *res* to Euler's constant γ. This creates an element of the (transcendental?) number field $\mathbb{Q}(\gamma)$.

void **ca_unknown**(*ca_t res*, *ca_ctx_t ctx*)
Sets *res* to the meta-value *Unknown*.

void **ca_undefined**(*ca_t res*, *ca_ctx_t ctx*)
Sets *res* to *Undefined*.

void **ca_uinf**(*ca_t res*, *ca_ctx_t ctx*)
Sets *res* to unsigned infinity $\tilde{\infty}$.

void **ca_pos_inf**(*ca_t res*, *ca_ctx_t ctx*)

void **ca_neg_inf**(*ca_t res*, *ca_ctx_t ctx*)

void **ca_pos_i_inf**(*ca_t res*, *ca_ctx_t ctx*)

void **ca_neg_i_inf**(*ca_t res*, *ca_ctx_t ctx*)
Sets *res* to the signed infinity $+\infty$, $-\infty$, $+i\infty$ or $-i\infty$.

4.1.9 Assignment and conversion

void **ca_set**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to a copy of *x*.

void **ca_set_si**(*ca_t res*, *slong v*, *ca_ctx_t ctx*)

void **ca_set_ui**(*ca_t res*, *ulong v*, *ca_ctx_t ctx*)

void **ca_set_fmpz**(*ca_t res*, **const** *fmpz_t v*, *ca_ctx_t ctx*)

void **ca_set_fmpq**(*ca_t res*, **const** *fmpq_t v*, *ca_ctx_t ctx*)
Sets *res* to the integer or rational number *v*. This creates a canonical representation of this number as an element of the trivial field $\mathbb{Q}$.

void **ca_set_d**(*ca_t res*, double *x*, *ca_ctx_t ctx*)

void **ca_set_d_d**(*ca_t res*, double *x*, double *y*, *ca_ctx_t ctx*)
Sets *res* to the value of *x*, or the complex value $x + yi$. NaN is interpreted as *Unknown* (not *Undefined*).

void **ca_transfer**(*ca_t res*, *ca_ctx_t res_ctx*, **const** *ca_t src*, *ca_ctx_t src_ctx*)
Sets *res* to *src* where the corresponding context objects *res_ctx* and *src_ctx* may be different.

This operation preserves the mathematical value represented by *src*, but may result in a different internal representation depending on the settings of the context objects.

4.1.10 Conversion of algebraic numbers

void **ca_set_qqbar**(*ca_t res*, **const** *qqbar_t x*, *ca_ctx_t ctx*)
Sets *res* to the algebraic number x.

If x is rational, *res* is set to the canonical representation as an element in the trivial field $\mathbb{Q}$.

If x is irrational, this function always sets *res* to an element of a univariate number field $\mathbb{Q}(a)$. It will not, for example, identify $\sqrt{2}+\sqrt{3}$ as an element of $\mathbb{Q}(\sqrt{2},\sqrt{3})$. However, it may attempt to find a simpler number field than that generated by x itself. For example:

- If x is quadratic, it will be expressed as an element of $\mathbb{Q}(\sqrt{N})$ where N has no small repeated factors (obtained by performing a smooth factorization of the discriminant).
- TODO: if possible, coerce x to a low-degree cyclotomic field.

int **ca_get_fmpz**(*fmpz_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

int **ca_get_fmpq**(*fmpz_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

int **ca_get_qqbar**(*qqbar_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Attempts to evaluate x to an explicit integer, rational or algebraic number. If successful, sets *res* to this number and returns 1. If unsuccessful, returns 0.

The conversion certainly fails if x does not represent an integer, rational or algebraic number (respectively), but can also fail if x is too expensive to compute under the current evaluation limits. In particular, the evaluation will be aborted if an intermediate algebraic number (or more precisely, the resultant polynomial prior to factorization) exceeds `CA_OPT_QQBAR_DEG_LIMIT` or the coefficients exceed some multiple of `CA_OPT_PREC_LIMIT`. Note that evaluation may hit those limits even if the minimal polynomial for x itself is small. The conversion can also fail if no algorithm has been implemented for the functions appearing in the construction of x.

int **ca_can_evaluate_qqbar**(**const** *ca_t x*, *ca_ctx_t ctx*)
Checks if *ca_get_qqbar()* has a chance to succeed. In effect, this checks if all extension numbers are manifestly algebraic numbers (without doing any evaluation).

4.1.11 Random generation

void **ca_randtest_rational**(*ca_t res*, flint_rand_t *state*, *slong bits*, *ca_ctx_t ctx*)
Sets *res* to a random rational number with numerator and denominator up to *bits* bits in size.

void **ca_randtest**(*ca_t res*, flint_rand_t *state*, *slong depth*, *slong bits*, *ca_ctx_t ctx*)
Sets *res* to a random number generated by evaluating a random expression. The algorithm randomly selects between generating a "simple" number (a random rational number or quadratic field element with coefficients up to *bits* in size, or a random builtin constant), or if *depth* is nonzero, applying a random arithmetic operation or function to operands produced through recursive calls with *depth* - 1. The output is guaranteed to be a number, not a special value.

void **ca_randtest_special**(*ca_t res*, flint_rand_t *state*, *slong depth*, *slong bits*, *ca_ctx_t ctx*)
Randomly generates either a special value or a number.

void **ca_randtest_same_nf**(*ca_t res*, flint_rand_t *state*, **const** *ca_t x*, *slong bits*, *slong den_bits*, *ca_ctx_t ctx*)
Sets *res* to a random element in the same number field as x, with numerator coefficients up to *bits* in size and denominator up to *den_bits* in size. This function requires that x is an element of an absolute number field.

4.1.12 Representation properties

The following functions deal with the representation of a *ca_t* and hence can always be decided quickly and unambiguously. The return value for predicates is 0 for false and 1 for true.

int **ca_equal_repr**(**const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Returns whether x and y have identical representation. For field elements, this checks if x and y belong to the same formal field (with generators having identical representation) and are represented by the same rational function within that field.

For special values, this tests equality of the special values, with *Unknown* handled as if it were a value rather than a meta-value: that is, *Unknown* = *Unknown* gives 1, and *Unknown* = y gives 0 for any other kind of value y. If neither x nor y is *Unknown*, then representation equality implies that x and y describe to the same mathematical value, but if either operand is *Unknown*, the result is meaningless for mathematical comparison.

int **ca_cmp_repr**(**const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Compares the representations of x and y in a canonical sort order, returning -1, 0 or 1. This only performs a lexicographic comparison of the representations of x and y; the return value does not say anything meaningful about the numbers represented by x and y.

ulong **ca_hash_repr**(**const** *ca_t x*, *ca_ctx_t ctx*)
Hashes the representation of x.

int **ca_is_unknown**(**const** *ca_t x*, *ca_ctx_t ctx*)
Returns whether x is Unknown.

int **ca_is_special**(**const** *ca_t x*, *ca_ctx_t ctx*)
Returns whether x is a special value or metavalue (not a field element).

int **ca_is_qq_elem**(**const** *ca_t x*, *ca_ctx_t ctx*)
Returns whether x is represented as an element of the rational field $\mathbb{Q}$.

int **ca_is_qq_elem_zero**(**const** *ca_t x*, *ca_ctx_t ctx*)

int **ca_is_qq_elem_one**(**const** *ca_t x*, *ca_ctx_t ctx*)

int **ca_is_qq_elem_integer**(**const** *ca_t x*, *ca_ctx_t ctx*)
Returns whether x is represented as the element 0, 1 or any integer in the rational field $\mathbb{Q}$.

int **ca_is_nf_elem**(**const** *ca_t x*, *ca_ctx_t ctx*)
Returns whether x is represented as an element of a univariate algebraic number field $\mathbb{Q}(a)$.

int **ca_is_cyclotomic_nf_elem**(*slong* *p, *ulong* *q, **const** *ca_t x*, *ca_ctx_t ctx*)
Returns whether x is represented as an element of a univariate cyclotomic field, i.e. $\mathbb{Q}(a)$ where a is a root of unity. If p and q are not *NULL* and x is represented as an element of a cyclotomic field, this also sets p and q to the minimal integers with $0 \le p < q$ such that the generating root of unity is $a = e^{2\pi i p/q}$. Note that the answer 0 does not prove that x is not a cyclotomic number, and the order q is also not necessarily the generator of the *smallest* cyclotomic field containing x. For the purposes of this function, only nontrivial cyclotomic fields count; the return value is 0 if x is represented as a rational number.

int **ca_is_generic_elem**(**const** *ca_t x*, *ca_ctx_t ctx*)
Returns whether x is represented as a generic field element; i.e. it is not a special value, not represented as an element of the rational field, and not represented as an element of a univariate algebraic number field.

4.1.13 Value predicates

The following predicates check a mathematical property which might not be effectively decidable. The result is a *truth_t* to allow representing an unknown outcome.

truth_t **ca_check_is_number**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is a number. The result is `T_TRUE` is x is a field element (and hence a complex number), `T_FALSE` if x is an infinity or *Undefined*, and `T_UNKNOWN` if x is *Unknown*.

truth_t **ca_check_is_zero**(**const** *ca_t x*, *ca_ctx_t ctx*)

truth_t **ca_check_is_one**(**const** *ca_t x*, *ca_ctx_t ctx*)

truth_t **ca_check_is_neg_one**(**const** *ca_t x*, *ca_ctx_t ctx*)

truth_t **ca_check_is_i**(**const** *ca_t x*, *ca_ctx_t ctx*)

truth_t **ca_check_is_neg_i**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is equal to the number 0, 1, -1, i, or $-i$.

truth_t **ca_check_is_algebraic**(**const** *ca_t x*, *ca_ctx_t ctx*)

truth_t **ca_check_is_rational**(**const** *ca_t x*, *ca_ctx_t ctx*)

truth_t **ca_check_is_integer**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is respectively an algebraic number, a rational number, or an integer.

truth_t **ca_check_is_real**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is a real number. Warning: this returns `T_FALSE` if x is an infinity with real sign.

truth_t **ca_check_is_negative_real**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is a negative real number. Warning: this returns `T_FALSE` if x is negative infinity.

truth_t **ca_check_is_imaginary**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is an imaginary number. Warning: this returns `T_FALSE` if x is an infinity with imaginary sign.

truth_t **ca_check_is_undefined**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is the special value *Undefined*.

truth_t **ca_check_is_infinity**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is any infinity (unsigned or signed).

truth_t **ca_check_is_uinf**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is unsigned infinity $\tilde{\infty}$.

truth_t **ca_check_is_signed_inf**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is any signed infinity.

truth_t **ca_check_is_pos_inf**(**const** *ca_t x*, *ca_ctx_t ctx*)

truth_t **ca_check_is_neg_inf**(**const** *ca_t x*, *ca_ctx_t ctx*)

truth_t **ca_check_is_pos_i_inf**(**const** *ca_t x*, *ca_ctx_t ctx*)

truth_t **ca_check_is_neg_i_inf**(**const** *ca_t x*, *ca_ctx_t ctx*)
Tests if x is equal to the signed infinity $+\infty$, $-\infty$, $+i\infty$, $-i\infty$, respectively.

4.1.14 Comparisons

truth_t **ca_check_equal**(**const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Tests $x = y$ as a mathematical equality. The result is `T_UNKNOWN` if either operand is *Unknown*. The result may also be `T_UNKNOWN` if x and y are numerically indistinguishable and cannot be proved equal or unequal by an exact computation.

truth_t **ca_check_lt**(**const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)

truth_t **ca_check_le**(**const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)

truth_t **ca_check_gt**(**const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)

truth_t **ca_check_ge**(**const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Compares x and y, implementing the respective operations $x < y$, $x \le y$, $x > y$, $x \ge y$. Only real numbers and $-\infty$ and $+\infty$ are considered comparable. The result is `T_FALSE` (not `T_UNKNOWN`) if either operand is not comparable (being a nonreal complex number, unsigned infinity, or undefined).

4.1.15 Field structure operations

void **ca_merge_fields**(*ca_t resx*, *ca_t resy*, **const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Sets *resx* and *resy* to copies of x and y coerced to a common field. Both x and y must be field elements (not special values).

In the present implementation, this simply merges the lists of generators, avoiding duplication. In the future, it will be able to eliminate generators satisfying algebraic relations.

void **ca_condense_field**(*ca_t res*, *ca_ctx_t ctx*)
Attempts to demote the value of *res* to a trivial subfield of its current field by removing unused generators. In particular, this demotes any obviously rational value to the trivial field $\mathbb{Q}$.

This function is applied automatically in most operations (arithmetic operations, etc.).

ca_ext_ptr **ca_is_gen_as_ext**(**const** *ca_t x*, *ca_ctx_t ctx*)
If x is a generator of its formal field, $x = a_k \in \mathbb{Q}(a_1, \ldots, a_n)$, returns a pointer to the extension number defining a_k. If x is not a generator, returns *NULL*.

4.1.16 Arithmetic

void **ca_neg**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the negation of x. For numbers, this operation amounts to a direct negation within the formal field. For a signed infinity $c\infty$, negation gives $(-c)\infty$; all other special values are unchanged.

void **ca_add_fmpq**(*ca_t res*, **const** *ca_t x*, **const** *fmpq_t y*, *ca_ctx_t ctx*)

void **ca_add_fmpz**(*ca_t res*, **const** *ca_t x*, **const** *fmpz_t y*, *ca_ctx_t ctx*)

void **ca_add_ui**(*ca_t res*, **const** *ca_t x*, *ulong y*, *ca_ctx_t ctx*)

void **ca_add_si**(*ca_t res*, **const** *ca_t x*, *slong y*, *ca_ctx_t ctx*)

void **ca_add**(*ca_t res*, **const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Sets *res* to the sum of x and y. For special values, the following rules apply ($c\infty$ denotes a signed infinity, $|c| = 1$):

- $c\infty + d\infty = c\infty$ if $c = d$
- $c\infty + d\infty =$ Undefined if $c \ne d$
- $\tilde{\infty} + c\infty = \tilde{\infty} + \tilde{\infty} =$ Undefined
- $c\infty + z = c\infty$ if $z \in \mathbb{C}$
- $\tilde{\infty} + z = \tilde{\infty}$ if $z \in \mathbb{C}$
- $z +$ Undefined $=$ Undefined for any value z (including *Unknown*)

In any other case involving special values, or if the specific case cannot be distinguished, the result is *Unknown*.

void **ca_sub_fmpq**(*ca_t res*, **const** *ca_t x*, **const** *fmpq_t y*, *ca_ctx_t ctx*)

void **ca_sub_fmpz**(*ca_t res*, **const** *ca_t x*, **const** *fmpz_t y*, *ca_ctx_t ctx*)

void **ca_sub_ui**(*ca_t res*, **const** *ca_t x*, *ulong y*, *ca_ctx_t ctx*)

void **ca_sub_si**(*ca_t res*, **const** *ca_t x*, *slong y*, *ca_ctx_t ctx*)

void **ca_fmpq_sub**(*ca_t res*, **const** *fmpq_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)

void **ca_fmpz_sub**(*ca_t res*, **const** *fmpz_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)

void **ca_ui_sub**(*ca_t res*, *ulong x*, **const** *ca_t y*, *ca_ctx_t ctx*)

void **ca_si_sub**(*ca_t res*, *slong x*, **const** *ca_t y*, *ca_ctx_t ctx*)

void **ca_sub**(*ca_t res*, **const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Sets *res* to the difference of x and y. This is equivalent to computing $x + (-y)$.

void **ca_mul_fmpq**(*ca_t res*, **const** *ca_t x*, **const** *fmpq_t y*, *ca_ctx_t ctx*)

void **ca_mul_fmpz**(*ca_t res*, **const** *ca_t x*, **const** *fmpz_t y*, *ca_ctx_t ctx*)

void **ca_mul_ui**(*ca_t res*, **const** *ca_t x*, *ulong y*, *ca_ctx_t ctx*)

void **ca_mul_si**(*ca_t res*, **const** *ca_t x*, *slong y*, *ca_ctx_t ctx*)

void **ca_mul**(*ca_t res*, **const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Sets *res* to the product of x and y. For special values, the following rules apply ($c\infty$ denotes a signed infinity, $|c| = 1$):

- $c\infty \cdot d\infty = cd\infty$
- $c\infty \cdot \tilde{\infty} = \tilde{\infty}$
- $\tilde{\infty} \cdot \tilde{\infty} = \tilde{\infty}$
- $c\infty \cdot z = \operatorname{sgn}(z)c\infty$ if $z \in \mathbb{C} \setminus \{0\}$
- $c\infty \cdot 0 =$ Undefined
- $\tilde{\infty} \cdot 0 =$ Undefined
- $z \cdot$ Undefined = Undefined for any value z (including *Unknown*)

In any other case involving special values, or if the specific case cannot be distinguished, the result is *Unknown*.

void **ca_inv**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the multiplicative inverse of x. In a univariate algebraic number field, this always produces a rational denominator, but the denominator might not be rationalized in a multivariate field. For special values and zero, the following rules apply:

- $1/(c\infty) = 1/\tilde{\infty} = 0$
- $1/0 = \tilde{\infty}$
- 1/Undefined = Undefined
- 1/Unknown = Unknown

If it cannot be determined whether x is zero or nonzero, the result is *Unknown*.

void **ca_fmpq_div**(*ca_t res*, **const** *fmpq_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)

void **ca_fmpz_div**(*ca_t res*, **const** *fmpz_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)

void **ca_ui_div**(*ca_t res*, *ulong x*, **const** *ca_t y*, *ca_ctx_t ctx*)

void **ca_si_div**(*ca_t res*, *slong x*, **const** *ca_t y*, *ca_ctx_t ctx*)

void **ca_div_fmpq**(*ca_t res*, **const** *ca_t x*, **const** *fmpq_t y*, *ca_ctx_t ctx*)

void **ca_div_fmpz**(*ca_t res*, **const** *ca_t x*, **const** *fmpz_t y*, *ca_ctx_t ctx*)

void **ca_div_ui**(*ca_t res*, **const** *ca_t x*, *ulong y*, *ca_ctx_t ctx*)

void **ca_div_si**(*ca_t res*, **const** *ca_t x*, *slong y*, *ca_ctx_t ctx*)

void **ca_div**(*ca_t res*, **const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Sets *res* to the quotient of x and y. This is equivalent to computing $x \cdot (1/y)$. For special values and division by zero, this implies the following rules ($c\infty$ denotes a signed infinity, $|c| = 1$):

- $(c\infty)/(d\infty) = (c\infty)/\tilde{\infty} = \tilde{\infty}/(c\infty) = \tilde{\infty}/\tilde{\infty} =$ Undefined
- $c\infty/z = (c/\operatorname{sgn}(z))\infty$ if $z \in \mathbb{C} \setminus \{0\}$
- $c\infty/0 = \tilde{\infty}/0 = \tilde{\infty}$
- $z/(c\infty) = z/\tilde{\infty} = 0$ if $z \in \mathbb{C}$
- $z/0 = \tilde{\infty}$ if $z \in \mathbb{C} \setminus \{0\}$

- $0/0$ = Undefined
- z/Undefined = Undefined for any value z (including *Unknown*)
- Undefined/z = Undefined for any value z (including *Unknown*)

In any other case involving special values, or if the specific case cannot be distinguished, the result is *Unknown*.

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

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

void **ca_fmpz_poly_evaluate**(*ca_t res*, **const** *fmpz_poly_t poly*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_fmpq_poly_evaluate**(*ca_t res*, **const** *fmpq_poly_t poly*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the polynomial *poly* evaluated at x.

void **ca_fmpz_mpoly_evaluate_horner**(*ca_t res*, **const** *fmpz_mpoly_t f*, *ca_srcptr x*, **const** *fmpz_mpoly_ctx_t mctx*, *ca_ctx_t ctx*)

void **ca_fmpz_mpoly_evaluate_iter**(*ca_t res*, **const** *fmpz_mpoly_t f*, *ca_srcptr x*, **const** *fmpz_mpoly_ctx_t mctx*, *ca_ctx_t ctx*)

void **ca_fmpz_mpoly_evaluate**(*ca_t res*, **const** *fmpz_mpoly_t f*, *ca_srcptr x*, **const** *fmpz_mpoly_ctx_t mctx*, *ca_ctx_t ctx*)
Sets *res* to the multivariate polynomial f evaluated at the vector of arguments x.

void **ca_fmpz_mpoly_q_evaluate**(*ca_t res*, **const** *fmpz_mpoly_q_t f*, *ca_srcptr x*, **const** *fmpz_mpoly_ctx_t mctx*, *ca_ctx_t ctx*)
Sets *res* to the multivariate rational function f evaluated at the vector of arguments x.

void **ca_fmpz_mpoly_q_evaluate_no_division_by_zero**(*ca_t res*, **const** *fmpz_mpoly_q_t f*, *ca_srcptr x*, **const** *fmpz_mpoly_ctx_t mctx*, *ca_ctx_t ctx*)

void **ca_inv_no_division_by_zero**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
These functions behave like the normal arithmetic functions, but assume (and do not check) that division by zero cannot occur. Division by zero will result in undefined behavior.

4.1.17 Powers and roots

void **ca_sqr**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the square of x.

void **ca_pow_fmpq**(*ca_t res*, **const** *ca_t x*, **const** *fmpq_t y*, *ca_ctx_t ctx*)

void **ca_pow_fmpz**(*ca_t res*, **const** *ca_t x*, **const** *fmpz_t y*, *ca_ctx_t ctx*)

void **ca_pow_ui**(*ca_t res*, **const** *ca_t x*, *ulong y*, *ca_ctx_t ctx*)

void **ca_pow_si**(*ca_t res*, **const** *ca_t x*, *slong y*, *ca_ctx_t ctx*)

void **ca_pow**(*ca_t res*, **const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Sets *res* to x raised to the power y. Handling of special values is not yet implemented.

void **ca_pow_si_arithmetic**(*ca_t res*, **const** *ca_t x*, *slong n*, *ca_ctx_t ctx*)
Sets *res* to x raised to the power n. Whereas *ca_pow()*, *ca_pow_si()* etc. may create x^n as an extension number if n is large, this function always perform the exponentiation using field arithmetic.

void **ca_sqrt_inert**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_sqrt_nofactor**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_sqrt_factor**(*ca_t res*, **const** *ca_t x*, *ulong flags*, *ca_ctx_t ctx*)

void **ca_sqrt**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the principal square root of *x*.

For special values, the following definitions apply:

- $\sqrt{c\infty} = \sqrt{c}\infty$
- $\sqrt{\tilde{\infty}} = \tilde{\infty}$.
- Both *Undefined* and *Unknown* map to themselves.

The *inert* version outputs the generator in the formal field $\mathbb{Q}(\sqrt{x})$ without simplifying.

The *factor* version writes $x = A^2 B$ in K where K is the field of x, and outputs $A\sqrt{B}$ or $-A\sqrt{B}$ (whichever gives the correct sign) as an element of $K(\sqrt{B})$ or some subfield thereof. This factorization is only a heuristic and is not guaranteed to make B minimal. Factorization options can be passed through to *flags*: see *ca_factor()* for details.

The *nofactor* version will not perform a general factorization, but may still perform other simplifications. It may in particular attempt to simplify $\sqrt{x}$ to a single element in $\overline{\mathbb{Q}}$.

void **ca_sqrt_ui**(*ca_t res*, *ulong n*, *ca_ctx_t ctx*)
Sets *res* to the principal square root of *n*.

4.1.18 Complex parts

void **ca_abs**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the absolute value of *x*.

For special values, the following definitions apply:

- $|c\infty| = |\tilde{\infty}| = +\infty$.
- Both *Undefined* and *Unknown* map to themselves.

This function will attempt to simplify its argument through an exact computation. It may in particular attempt to simplify $|x|$ to a single element in $\overline{\mathbb{Q}}$.

In the generic case, this function outputs an element of the formal field $\mathbb{Q}(|x|)$.

void **ca_sgn**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the sign of *x*, defined by

$$\operatorname{sgn}(x) = \begin{cases} 0 & x = 0 \\ \frac{x}{|x|} & x \neq 0 \end{cases}$$

for numbers. For special values, the following definitions apply:

- $\operatorname{sgn}(c\infty) = c$.
- $\operatorname{sgn}(\tilde{\infty}) =$ Undefined.
- Both *Undefined* and *Unknown* map to themselves.

This function will attempt to simplify its argument through an exact computation. It may in particular attempt to simplify $\operatorname{sgn}(x)$ to a single element in $\overline{\mathbb{Q}}$.

In the generic case, this function outputs an element of the formal field $\mathbb{Q}(\operatorname{sgn}(x))$.

void **ca_csgn**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the extension of the real sign function taking the value 1 for z strictly in the right half plane, -1 for z strictly in the left half plane, and the sign of the imaginary part when z is on the imaginary axis. Equivalently, $\operatorname{csgn}(z) = z/\sqrt{z^2}$ except that the value is 0 when z is exactly zero. This function gives *Undefined* for unsigned infinity and $\operatorname{csgn}(\operatorname{sgn}(c\infty)) = \operatorname{csgn}(c)$ for signed infinities.

void **ca_arg**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the complex argument (phase) of x, normalized to the range $(-\pi, +\pi]$. The argument of 0 is defined as 0. For special values, the following definitions apply:

- $\arg(c\infty) = \arg(c)$.
- $\arg(\tilde{\infty}) =$ Undefined.
- Both *Undefined* and *Unknown* map to themselves.

void **ca_re**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the real part of x. The result is *Undefined* if x is any infinity (including a real infinity).

void **ca_im**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the imaginary part of x. The result is *Undefined* if x is any infinity (including an imaginary infinity).

void **ca_conj_deep**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_conj_shallow**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_conj**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the complex conjugate of x. The *shallow* version creates a new extension element $\overline{x}$ unless x can be trivially conjugated in-place in the existing field. The *deep* version recursively conjugates the extension numbers in the field of x.

void **ca_floor**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the floor function of x. The result is *Undefined* if x is any infinity (including a real infinity). For complex numbers, this is presently defined to take the floor of the real part.

void **ca_ceil**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the ceiling function of x. The result is *Undefined* if x is any infinity (including a real infinity). For complex numbers, this is presently defined to take the ceiling of the real part.

4.1.19 Exponentials and logarithms

void **ca_exp**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the exponential function of x.

For special values, the following definitions apply:

- $e^{+\infty} = +\infty$
- $e^{c\infty} = \tilde{\infty}$ if $0 < \operatorname{Re}(c) < 1$.
- $e^{c\infty} = 0$ if $\operatorname{Re}(c) < 0$.
- $e^{c\infty} =$ Undefined if $\operatorname{Re}(c) = 0$.
- $e^{\tilde{\infty}} =$ Undefined.
- Both *Undefined* and *Unknown* map to themselves.

The following symbolic simplifications are performed automatically:

- $e^0 = 1$
- $e^{\log(z)} = z$
- $e^{(p/q)\log(z)} = z^{p/q}$ (for rational p/q)
- $e^{(p/q)\pi i} =$ algebraic root of unity (for small rational p/q)

In the generic case, this function outputs an element of the formal field $\mathbb{Q}(e^x)$.

void **ca_log**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the natural logarithm of x.

For special values and at the origin, the following definitions apply:

- For any infinity, $\log(c\infty) = \log(\tilde{\infty}) = +\infty$.

- $\log(0) = -\infty$. The result is *Unknown* if deciding $x = 0$ fails.
- Both *Undefined* and *Unknown* map to themselves.

The following symbolic simplifications are performed automatically:

- $\log(1) = 0$
- $\log(e^z) = z + 2\pi i k$
- $\log(\sqrt{z}) = \frac{1}{2}\log(z) + 2\pi i k$
- $\log(z^a) = a\log(z) + 2\pi i k$
- $\log(x) = \log(-x) + \pi i$ for negative real x

In the generic case, this function outputs an element of the formal field $\mathbb{Q}(\log(x))$.

4.1.20 Trigonometric functions

void **ca_sin_cos_exponential**(*ca_t res1*, *ca_t res2*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_sin_cos_direct**(*ca_t res1*, *ca_t res2*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_sin_cos_tangent**(*ca_t res1*, *ca_t res2*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_sin_cos**(*ca_t res1*, *ca_t res2*, **const** *ca_t x*, *ca_ctx_t ctx*)

Sets *res1* to the sine of x and *res2* to the cosine of x. Either *res1* or *res2* can be *NULL* to compute only the other function. Various representations are implemented:

- The *exponential* version expresses the sine and cosine in terms of complex exponentials. Simple algebraic values will simplify to rational numbers or elements of cyclotomic fields.
- The *direct* method expresses the sine and cosine in terms of the original functions (perhaps after applying some symmetry transformations, which may interchange sin and cos). Extremely simple algebraic values will automatically simplify to elements of real algebraic number fields.
- The *tangent* version expresses the sine and cosine in terms of $\tan(x/2)$, perhaps after applying some symmetry transformations. Extremely simple algebraic values will automatically simplify to elements of real algebraic number fields.

By default, the standard function uses the *exponential* representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the *CA_OPT_TRIG_FORM* context setting.

For special values, the following definitions apply:

- $\sin(\pm i\infty) = \pm i\infty$
- $\cos(\pm i\infty) = +\infty$
- All other infinities give Undefined

void **ca_sin**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_cos**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

Sets *res* to the sine or cosine of x. These functions are shortcuts for *ca_sin_cos()*.

void **ca_tan_sine_cosine**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_tan_exponential**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_tan_direct**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_tan**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

Sets *res* to the tangent of x. The *sine_cosine* version evaluates the tangent as a quotient of a sine and cosine, the *direct* version evaluates it directly as a tangent (possibly after transforming the variable), and the *exponential* version evaluates it in terms of complex exponentials. Simple algebraic values will automatically simplify to elements of trigonometric or cyclotomic number fields.

By default, the standard function uses the *exponential* representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the *CA_OPT_TRIG_FORM* context setting.

For special values, the following definitions apply:

- At poles, $\tan((n + \frac{1}{2})\pi) = \tilde{\infty}$
- $\tan(e^{i\theta}\infty) = +i, \quad 0 < \theta < \pi$
- $\tan(e^{i\theta}\infty) = -i, \quad -\pi < \theta < 0$
- $\tan(\pm\infty) = \tan(\tilde{\infty}) =$ Undefined

void **ca_cot**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the cotangent *x*. This is equivalent to computing the reciprocal of the tangent.

void **ca_atan_logarithm**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_atan_direct**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_atan**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the inverse tangent of *x*.

The *direct* version expresses the result as an inverse tangent (possibly after transforming the variable). The *logarithm* version expresses it in terms of complex logarithms. Simple algebraic inputs will automatically simplify to rational multiples of π.

By default, the standard function uses the *logarithm* representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the *CA_OPT_TRIG_FORM* context setting (exponential mode results in logarithmic forms).

For special values, the following definitions apply:

- $\operatorname{atan}(\pm i) = \pm i\infty$
- $\operatorname{atan}(c\infty) = \operatorname{csgn}(c)\pi/2$
- $\operatorname{atan}(\tilde{\infty}) =$ Undefined

void **ca_asin_logarithm**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_acos_logarithm**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_asin_direct**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_acos_direct**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_asin**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)

void **ca_acos**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the inverse sine (respectively, cosine) of *x*.

The *direct* version expresses the result as an inverse sine or cosine (possibly after transforming the variable). The *logarithm* version expresses it in terms of complex logarithms. Simple algebraic inputs will automatically simplify to rational multiples of π.

By default, the standard function uses the *logarithm* representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the *CA_OPT_TRIG_FORM* context setting (exponential mode results in logarithmic forms).

The inverse cosine is presently implemented as $\operatorname{acos}(x) = \pi/2 - \operatorname{asin}(x)$.

4.1.21 Special functions

void **ca_gamma**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the gamma function of *x*.

void **ca_erf**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the error function of *x*.

void **ca_erfc**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the complementary error function of *x*.

void **ca_erfi**(*ca_t res*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets *res* to the imaginary error function of *x*.

4.1.22 Numerical evaluation

void **ca_get_acb_raw**(*acb_t res*, **const** *ca_t x*, *slong prec*, *ca_ctx_t ctx*)
Sets *res* to an enclosure of the numerical value of *x*. A working precision of *prec* bits is used internally for the evaluation, without adaptive refinement. If *x* is any special value, *res* is set to *acb_indeterminate*.

void **ca_get_acb**(*acb_t res*, **const** *ca_t x*, *slong prec*, *ca_ctx_t ctx*)

void **ca_get_acb_accurate_parts**(*acb_t res*, **const** *ca_t x*, *slong prec*, *ca_ctx_t ctx*)
Sets *res* to an enclosure of the numerical value of *x*. The working precision is increased adaptively to try to ensure *prec* accurate bits in the output. The *accurate_parts* version tries to ensure *prec* accurate bits for both the real and imaginary part separately.

The refinement is stopped if the working precision exceeds `CA_OPT_PREC_LIMIT` (or twice the initial precision, if this is larger). The user may call *acb_rel_accuracy_bits* to check is the calculation was successful.

The output is not rounded down to *prec* bits (to avoid unnecessary double rounding); the user may call *acb_set_round* when rounding is desired.

char ***ca_get_decimal_str**(**const** *ca_t x*, *slong digits*, *ulong flags*, *ca_ctx_t ctx*)
Returns a decimal approximation of *x* with precision up to *digits*. The output is guaranteed to be correct within 1 ulp in the returned digits, but the number of returned digits may be smaller than *digits* if the numerical evaluation does not succeed.

If *flags* is set to 1, attempts to achieve full accuracy for both the real and imaginary parts separately.

If *x* is not finite or a finite enclosure cannot be produced, returns the string "?".

The user should free the returned string with `flint_free`.

4.1.23 Rewriting and simplification

void **ca_rewrite_complex_normal_form**(*ca_t res*, **const** *ca_t x*, int *deep*, *ca_ctx_t ctx*)
Sets *res* to *x* rewritten using standardizing transformations over the complex numbers:

- Elementary functions are rewritten in terms of (complex) exponentials, roots and logarithms
- Complex parts are rewritten using logarithms, square roots, and (deep) complex conjugates
- Algebraic numbers are rewritten in terms of cyclotomic fields where applicable

If *deep* is set, the rewriting is applied recursively to the tower of extension numbers; otherwise, the rewriting is only applied to the top-level extension numbers.

The result is not a normal form in the strong sense (the same number can have many possible representations even after applying this transformation), but in practice this is a powerful heuristic for simplification.

4.1.24 Factorization

type ca_factor_struct

type ca_factor_t
Represents a real or complex number in factored form $b_1^{e_1} b_2^{e_2} \cdots b_n^{e_n}$ where b_i and e_i are *ca_t* numbers (the exponents need not be integers).

void **ca_factor_init**(*ca_factor_t fac*, *ca_ctx_t ctx*)
Initializes *fac* and sets it to the empty factorization (equivalent to the number 1).

void **ca_factor_clear**(*ca_factor_t fac*, *ca_ctx_t ctx*)
Clears the factorization structure *fac*.

void **ca_factor_one**(*ca_factor_t fac*, *ca_ctx_t ctx*)
Sets *fac* to the empty factorization (equivalent to the number 1).

void **ca_factor_print**(**const** *ca_factor_t fac*, *ca_ctx_t ctx*)
Prints a description of *fac* to standard output.

void **ca_factor_insert**(*ca_factor_t fac*, **const** *ca_t base*, **const** *ca_t exp*, *ca_ctx_t ctx*)
Inserts b^e into *fac* where b is given by *base* and e is given by *exp*. If a base element structurally identical to *base* already exists in *fac*, the corresponding exponent is incremented by *exp*; otherwise, this factor is appended.

void **ca_factor_get_ca**(*ca_t res*, **const** *ca_factor_t fac*, *ca_ctx_t ctx*)
Expands *fac* back to a single *ca_t* by evaluating the powers and multiplying out the result.

void **ca_factor**(*ca_factor_t res*, **const** *ca_t x*, *ulong flags*, *ca_ctx_t ctx*)
Sets *res* to a factorization of x of the form $x = b_1^{e_1} b_2^{e_2} \cdots b_n^{e_n}$. Requires that x is not a special value. The type of factorization is controlled by *flags*, which can be set to a combination of constants in the following section.

Factorization options

The following flags select the structural polynomial factorization to perform over formal fields $\mathbb{Q}(a_1, \ldots, a_n)$. Each flag in the list strictly encompasses the factorization power of the preceding flag, so it is unnecessary to pass more than one flag.

CA_FACTOR_POLY_NONE
No polynomial factorization at all.

CA_FACTOR_POLY_CONTENT
Only extract the rational content.

CA_FACTOR_POLY_SQF
Perform a squarefree factorization in addition to extracting the rational content.

CA_FACTOR_POLY_FULL
Perform a full multivariate polynomial factorization.

The following flags select the factorization to perform over $\mathbb{Z}$. Integer factorization is applied if x is an element of $\mathbb{Q}$, and to the extracted rational content of polynomials. Each flag in the list strictly encompasses the factorization power of the preceding flag, so it is unnecessary to pass more than one flag.

CA_FACTOR_ZZ_NONE
No integer factorization at all.

CA_FACTOR_ZZ_SMOOTH
Perform a smooth factorization to extract small prime factors (heuristically up to CA_OPT_SMOOTH_LIMIT bits) in addition to identifying perfect powers.

`CA_FACTOR_ZZ_FULL`
: Perform a complete integer factorization into prime numbers. This is prohibitively slow for general integers exceeding 70-80 digits.

4.1.25 Context options

The *options* member of a *ca_ctx_t* object is an array of *slong* values controlling simplification behavior and various other settings. The values of the array at the following indices can be changed by the user (example: `ctx->options[CA_OPT_PREC_LIMIT] = 65536`).

It is recommended to set options controlling evaluation only at the time when a context object is created. Changing such options later should normally be harmless, but since the update will not apply retroactively to objects that have already been computed and cached, one might not see the expected behavior. Superficial options (printing) can be changed at any time.

`CA_OPT_VERBOSE`
: Whether to print debug information. Default value: 0.

`CA_OPT_PRINT_FLAGS`
: Printing style. See *Printing* for details. Default value: `CA_PRINT_DEFAULT`.

`CA_OPT_MPOLY_ORD`
: Monomial ordering to use for multivariate polynomials. Possible values are `ORD_LEX`, `ORD_DEGLEX` and `ORD_DEGREVLEX`. Default value: `ORD_LEX`. This option must be set before doing any computations.

`CA_OPT_PREC_LIMIT`
: Maximum precision to use internally for numerical evaluation with Arb, and in some cases for the magntiude of exact coefficients. This parameter affects the possibility to prove inequalities and find simplifications between related extension numbers. This is not a strict limit; some calculations may use higher precision when there is a good reason to do so. Default value: 4096.

`CA_OPT_QQBAR_DEG_LIMIT`
: Maximum degree of *qqbar_t* elements allowed internally during simplification of algebraic numbers. This limit may be exceeded when the user provides explicit *qqbar_t* input of higher degree. Default value: 120.

`CA_OPT_LOW_PREC`
: Numerical precision to use for fast checks (typically, before attempting more expensive operations). Default value: 64.

`CA_OPT_SMOOTH_LIMIT`
: Size in bits for factors in smooth integer factorization. Default value: 32.

`CA_OPT_LLL_PREC`
: Precision to use to find integer relations using LLL. Default value: 128.

`CA_OPT_POW_LIMIT`
: Largest exponent to expand powers automatically. This only applies in multivariate and transcendental fields: in number fields, `CA_OPT_PREC_LIMIT` applies instead. Default value: 20.

`CA_OPT_USE_GROEBNER`
: Boolean flag for whether to use Gröbner basis computation. This flag and the following limits affect the ability to prove multivariate identities. Default value: 1.

`CA_OPT_GROEBNER_LENGTH_LIMIT`
: Maximum length of ideal basis allowed in Buchberger's algorithm. Default value: 100.

`CA_OPT_GROEBNER_POLY_LENGTH_LIMIT`
: Maximum length of polynomials allowed in Buchberger's algorithm. Default value: 1000.

`CA_OPT_GROEBNER_POLY_BITS_LIMIT`
: Maximum coefficient size in bits of polynomials allowed in Buchberger's algorithm. Default value: 10000.

CA_OPT_VIETA_LIMIT
Maximum degree n of algebraic numbers for which to add Vieta's formulas to the reduction ideal. This must be set relatively low since the number of terms in Vieta's formulas is $O(2^n)$ and the resulting Gröbner basis computations can be expensive. Default value: 6.

CA_OPT_TRIG_FORM
Default representation of trigonometric functions. The following values are possible:

CA_TRIG_DIRECT
Use the direct functions (with some exceptions).

CA_TRIG_EXPONENTIAL
Use complex exponentials.

CA_TRIG_SINE_COSINE
Use sines and cosines.

CA_TRIG_TANGENT
Use tangents.

Default value: `CA_TRIG_EXPONENTIAL`.

The *exponential* representation is currently used by default as typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. This may change in the future.

4.1.26 Internal representation

CA_FMPQ(x)

CA_FMPQ_NUMREF(x)

CA_FMPQ_DENREF(x)
Assuming that x holds an element of the trivial field $\mathbb{Q}$, this macro returns a pointer which can be used as an *fmpq_t*, or respectively to the numerator or denominator as an *fmpz_t*.

CA_MPOLY_Q(x)
Assuming that x holds a generic field element as data, this macro returns a pointer which can be used as an *fmpz_mpoly_q_t*.

CA_NF_ELEM(x)
Assuming that x holds an Antic number field element as data, this macro returns a pointer which can be used as an *nf_elem_t*.

void **_ca_make_field_element**(*ca_t x*, *slong new_index*, *ca_ctx_t ctx*)
Changes the internal representation of x to that of an element of the field with index *new_index* in the context object *ctx*. This may destroy the value of x.

void **_ca_make_fmpq**(*ca_t x*, *ca_ctx_t ctx*)
Changes the internal representation of x to that of an element of the trivial field $\mathbb{Q}$. This may destroy the value of x.

4.2 ca_vec.h – vectors of real and complex numbers

A *ca_vec_t* represents a vector of real or complex numbers, implemented as an array of coefficients of type *ca_struct*.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients (taking *ca_ptr* and *ca_srcptr* arguments), and a non-underscore method which takes *ca_vec_t* input and performs automatic memory management.

Unlike *ca_poly_t*, a *ca_vec_t* is not normalised by removing zero coefficients; it retains the exact length assigned by the user.

4.2.1 Types, macros and constants

type ca_vec_struct

type ca_vec_t
Contains a pointer to an array of entries (*coeffs*), the used length (*length*), and the allocated size of the array (*alloc*).

A *ca_vec_t* is defined as an array of length one of type *ca_vec_struct*, permitting an *ca_vec_t* to be passed by reference.

ca_vec_entry(*vec*, *i*)
Macro returning a pointer to entry *i* in the vector *vec*. The index must be in bounds.

4.2.2 Memory management

ca_ptr **_ca_vec_init**(*slong len*, *ca_ctx_t ctx*)
Returns a pointer to an array of *len* coefficients initialized to zero.

void **ca_vec_init**(*ca_vec_t vec*, *slong len*, *ca_ctx_t ctx*)
Initializes *vec* to a length *len* vector. All entries are set to zero.

void **_ca_vec_clear**(*ca_ptr vec*, *slong len*, *ca_ctx_t ctx*)
Clears all *len* entries in *vec* and frees the pointer *vec* itself.

void **ca_vec_clear**(*ca_vec_t vec*, *ca_ctx_t ctx*)
Clears the vector *vec*.

void **_ca_vec_swap**(*ca_ptr vec1*, *ca_srcptr vec2*, *slong len*, *ca_ctx_t ctx*)
Swaps the entries in *vec1* and *vec2* efficiently.

void **ca_vec_swap**(*ca_vec_t vec1*, *ca_vec_t vec2*, *ca_ctx_t ctx*)
Swaps the vectors *vec1* and *vec2* efficiently.

4.2.3 Length

slong **ca_vec_length**(**const** *ca_vec_t vec*, *ca_ctx_t ctx*)
Returns the length of *vec*.

void **_ca_vec_fit_length**(*ca_vec_t vec*, *slong len*, *ca_ctx_t ctx*)
Allocates space in *vec* for *len* elements.

void **ca_vec_set_length**(*ca_vec_t vec*, *slong len*, *ca_ctx_t ctx*)
Sets the length of *vec* to *len*. If *vec* is shorter on input, it will be zero-extended. If *vec* is longer on input, it will be truncated.

4.2.4 Assignment

void **_ca_vec_set**(*ca_ptr res*, *ca_srcptr src*, *slong len*, *ca_ctx_t ctx*)
Sets *res* to a copy of *src* of length *len*.

void **ca_vec_set**(*ca_vec_t res*, **const** *ca_vec_t src*, *ca_ctx_t ctx*)
Sets *res* to a copy of *src*.

4.2.5 Special vectors

void **_ca_vec_zero**(*ca_ptr res*, *slong len*, *ca_ctx_t ctx*)
Sets the *len* entries in *res* to zeros.

void **ca_vec_zero**(*ca_vec_t res*, *slong len*, *ca_ctx_t ctx*)
Sets *res* to the length *len* zero vector.

4.2.6 Input and output

void **ca_vec_print**(**const** *ca_vec_t vec*, *ca_ctx_t ctx*)
Prints *vec* to standard output. The coefficients are printed on separate lines.

void **ca_vec_printn**(**const** *ca_vec_t poly*, *slong digits*, *ca_ctx_t ctx*)
Prints a decimal representation of *vec* with precision specified by *digits*. The coefficients are comma-separated and the whole list is enclosed in square brackets.

4.2.7 List operations

void **ca_vec_append**(*ca_vec_t vec*, **const** *ca_t f*, *ca_ctx_t ctx*)
Appends *f* to the end of *vec*.

4.2.8 Arithmetic

void **_ca_vec_neg**(*ca_ptr res*, *ca_srcptr src*, *slong len*, *ca_ctx_t ctx*)

void **ca_vec_neg**(*ca_vec_t res*, **const** *ca_vec_t src*, *ca_ctx_t ctx*)
Sets *res* to the negation of *src*.

void **_ca_vec_add**(*ca_ptr res*, *ca_srcptr vec1*, *ca_srcptr vec2*, *slong len*, *ca_ctx_t ctx*)

void **_ca_vec_sub**(*ca_ptr res*, *ca_srcptr vec1*, *ca_srcptr vec2*, *slong len*, *ca_ctx_t ctx*)
Sets *res* to the sum or difference of *vec1* and *vec2*, all vectors having length *len*.

void **_ca_vec_scalar_mul_ca**(*ca_ptr res*, *ca_srcptr src*, *slong len*, **const** *ca_t c*, *ca_ctx_t ctx*)
Sets *res* to *src* multiplied by *c*, all vectors having length *len*.

void **_ca_vec_scalar_div_ca**(*ca_ptr res*, *ca_srcptr src*, *slong len*, **const** *ca_t c*, *ca_ctx_t ctx*)
Sets *res* to *src* divided by *c*, all vectors having length *len*.

void **_ca_vec_scalar_addmul_ca**(*ca_ptr res*, *ca_srcptr src*, *slong len*, **const** *ca_t c*, *ca_ctx_t ctx*)
Adds *src* multiplied by *c* to the vector *res*, all vectors having length *len*.

void **_ca_vec_scalar_submul_ca**(*ca_ptr res*, *ca_srcptr src*, *slong len*, **const** *ca_t c*, *ca_ctx_t ctx*)
Subtracts *src* multiplied by *c* from the vector *res*, all vectors having length *len*.

4.2.9 Comparisons and properties

truth_t **_ca_vec_check_is_zero**(*ca_srcptr vec*, *slong len*, *ca_ctx_t ctx*)
Returns whether *vec* is the zero vector.

4.2.10 Internal representation

int **_ca_vec_is_fmpq_vec**(*ca_srcptr vec*, *slong len*, *ca_ctx_t ctx*)
Checks if all elements of *vec* are structurally rational numbers.

int **_ca_vec_fmpq_vec_is_fmpz_vec**(*ca_srcptr vec*, *slong len*, *ca_ctx_t ctx*)
Assuming that all elements of *vec* are structurally rational numbers, checks if all elements are integers.

void **_ca_vec_fmpq_vec_get_fmpz_vec_den**(fmpz *c, *fmpz_t den*, *ca_srcptr vec*, *slong len*, *ca_ctx_t ctx*)
Assuming that all elements of *vec* are structurally rational numbers, converts them to a vector of integers *c* on a common denominator *den*.

void **_ca_vec_set_fmpz_vec_div_fmpz**(*ca_ptr res*, **const** fmpz *v, **const** *fmpz_t den*, *slong len*, *ca_ctx_t ctx*)
Sets *res* to the rational vector given by numerators *v* and the common denominator *den*.

CHAPTER

FIVE

MATRICES AND POLYNOMIALS

5.1 ca_poly.h – dense univariate polynomials over the real and complex numbers

A *ca_poly_t* represents a univariate polynomial over the real or complex numbers (an element of $\mathbb{R}[X]$ or $\mathbb{C}[X]$), implemented as an array of coefficients of type *ca_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.

Warnings:

- A polynomial is always normalised by removing zero coefficients at the top. Coefficients will not be removed when Calcium is unable to prove that they are zero. The represented degree can therefore be larger than the degree of the mathematical polynomial. When the correct degree is needed, it is important to verify the leading coefficient. (Of course, this will never be an issue with polynomials that are explicitly monic, for example.)
- The special values *Undefined*, unsigned infinity and signed infinity supported by the scalar *ca_t* type are not really meaningful as coefficients of polynomials. We normally assume that the user does not assign those values to coefficients of polynomials, and the functions in this module will likewise normally not generate such coefficients. *Unknown* can still appear as a coefficient representing a number that is inaccessible for computation.

A polynomial with numerical coefficients and with a nonzero leading coefficient is called *proper*. The function *ca_poly_is_proper()* can be used to check for violations.

5.1.1 Types, macros and constants

type ca_poly_struct

type ca_poly_t

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

A *ca_poly_t* is defined as an array of length one of type *ca_poly_struct*, permitting an *ca_poly_t* to be passed by reference.

5.1.2 Memory management

void **ca_poly_init**(*ca_poly_t poly*, *ca_ctx_t ctx*)
Initializes the polynomial for use, setting it to the zero polynomial.

void **ca_poly_clear**(*ca_poly_t poly*, *ca_ctx_t ctx*)
Clears the polynomial, deallocating all coefficients and the coefficient array.

void **ca_poly_fit_length**(*ca_poly_t poly*, *slong len*, *ca_ctx_t ctx*)
Makes sure that the coefficient array of the polynomial contains at least *len* initialized coefficients.

void **_ca_poly_set_length**(*ca_poly_t poly*, *slong len*, *ca_ctx_t ctx*)
Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void **_ca_poly_normalise**(*ca_poly_t poly*, *ca_ctx_t ctx*)
Strips any top coefficients which can be proved identical to zero.

5.1.3 Assignment and simple values

void **ca_poly_zero**(*ca_poly_t poly*, *ca_ctx_t ctx*)
Sets *poly* to the zero polynomial.

void **ca_poly_one**(*ca_poly_t poly*, *ca_ctx_t ctx*)
Sets *poly* to the constant polynomial 1.

void **ca_poly_x**(*ca_poly_t poly*, *ca_ctx_t ctx*)
Sets *poly* to the monomial x.

void **ca_poly_set_ca**(*ca_poly_t poly*, **const** *ca_t c*, *ca_ctx_t ctx*)

void **ca_poly_set_si**(*ca_poly_t poly*, *slong c*, *ca_ctx_t ctx*)
Sets *poly* to the constant polynomial c.

void **ca_poly_set**(*ca_poly_t res*, **const** *ca_poly_t src*, *ca_ctx_t ctx*)

void **ca_poly_set_fmpz_poly**(*ca_poly_t res*, **const** *fmpz_poly_t src*, *ca_ctx_t ctx*)

void **ca_poly_set_fmpq_poly**(*ca_poly_t res*, **const** *fmpq_poly_t src*, *ca_ctx_t ctx*)
Sets *poly* the polynomial *src*.

void **ca_poly_set_coeff_ca**(*ca_poly_t poly*, *slong n*, **const** *ca_t x*, *ca_ctx_t ctx*)
Sets the coefficient at position n in *poly* to x.

void **ca_poly_transfer**(*ca_poly_t res*, *ca_ctx_t res_ctx*, **const** *ca_poly_t src*, *ca_ctx_t src_ctx*)
Sets *res* to *src* where the corresponding context objects *res_ctx* and *src_ctx* may be different.

This operation preserves the mathematical value represented by *src*, but may result in a different internal representation depending on the settings of the context objects.

5.1.4 Random generation

void **ca_poly_randtest**(*ca_poly_t poly*, flint_rand_t *state*, *slong len*, *slong depth*, *slong bits*, *ca_ctx_t ctx*)
Sets *poly* to a random polynomial of length up to *len* and with entries having complexity up to *depth* and *bits* (see *ca_randtest()*).

void **ca_poly_randtest_rational**(*ca_poly_t poly*, flint_rand_t *state*, *slong len*, *slong bits*, *ca_ctx_t ctx*)
Sets *poly* to a random rational polynomial of length up to *len* and with entries up to *bits* bits in size.

5.1.5 Input and output

void **ca_poly_print**(**const** *ca_poly_t poly*, *ca_ctx_t ctx*)
Prints *poly* to standard output. The coefficients are printed on separate lines.

void **ca_poly_printn**(**const** *ca_poly_t poly*, *slong digits*, *ca_ctx_t ctx*)
Prints a decimal representation of *poly* with precision specified by *digits*. The coefficients are comma-separated and the whole list is enclosed in square brackets.

5.1.6 Degree and leading coefficient

int **ca_poly_is_proper**(**const** *ca_poly_t poly*, *ca_ctx_t ctx*)
Checks that *poly* represents an element of $\mathbb{C}[X]$ with well-defined degree. This returns 1 if the leading coefficient of *poly* is nonzero and all coefficients of *poly* are numbers (not special values). It returns 0 otherwise. It returns 1 when *poly* is precisely the zero polynomial (which does not have a leading coefficient).

int **ca_poly_make_monic**(*ca_poly_t res*, **const** *ca_poly_t poly*, *ca_ctx_t ctx*)
Makes *poly* monic by dividing by the leading coefficient if possible and returns 1. Returns 0 if the leading coefficient cannot be certified to be nonzero, or if *poly* is the zero polynomial.

void **_ca_poly_reverse**(*ca_ptr res*, *ca_srcptr poly*, *slong len*, *slong n*, *ca_ctx_t ctx*)

void **ca_poly_reverse**(*ca_poly_t res*, **const** *ca_poly_t poly*, *slong n*, *ca_ctx_t ctx*)
Sets *res* to the reversal of *poly* considered as a polynomial of length *n*, zero-padding if needed. The underscore method assumes that *len* is positive and less than or equal to *n*.

5.1.7 Comparisons

truth_t **_ca_poly_check_equal**(*ca_srcptr poly1*, *slong len1*, *ca_srcptr poly2*, *slong len2*, *ca_ctx_t ctx*)
truth_t **ca_poly_check_equal**(**const** *ca_poly_t poly1*, **const** *ca_poly_t poly2*, *ca_ctx_t ctx*)
Checks if *poly1* and *poly2* represent the same polynomial. The underscore method assumes that *len1* is at least as large as *len2*.

truth_t **ca_poly_check_is_zero**(**const** *ca_poly_t poly*, *ca_ctx_t ctx*)
Checks if *poly* is the zero polynomial.

truth_t **ca_poly_check_is_one**(**const** *ca_poly_t poly*, *ca_ctx_t ctx*)
Checks if *poly* is the constant polynomial 1.

5.1.8 Arithmetic

void **_ca_poly_shift_left**(*ca_ptr res*, *ca_srcptr poly*, *slong len*, *slong n*, *ca_ctx_t ctx*)
void **ca_poly_shift_left**(*ca_poly_t res*, **const** *ca_poly_t poly*, *slong n*, *ca_ctx_t ctx*)
Sets *res* to *poly* shifted *n* coefficients to the left; that is, multiplied by x^n.

void **_ca_poly_shift_right**(*ca_ptr res*, *ca_srcptr poly*, *slong len*, *slong n*, *ca_ctx_t ctx*)
void **ca_poly_shift_right**(*ca_poly_t res*, **const** *ca_poly_t poly*, *slong n*, *ca_ctx_t ctx*)
Sets *res* to *poly* shifted *n* coefficients to the right; that is, divided by x^n.

void **ca_poly_neg**(*ca_poly_t res*, **const** *ca_poly_t src*, *ca_ctx_t ctx*)
Sets *res* to the negation of *src*.

void **_ca_poly_add**(*ca_ptr res*, *ca_srcptr poly1*, *slong len1*, *ca_srcptr poly2*, *slong len2*, *ca_ctx_t ctx*)
void **ca_poly_add**(*ca_poly_t res*, **const** *ca_poly_t poly1*, **const** *ca_poly_t poly2*, *ca_ctx_t ctx*)
Sets *res* to the sum of *poly1* and *poly2*.

void **_ca_poly_sub**(*ca_ptr res*, *ca_srcptr poly1*, *slong len1*, *ca_srcptr poly2*, *slong len2*, *ca_ctx_t ctx*)

void **ca_poly_sub**(*ca_poly_t res*, **const** *ca_poly_t poly1*, **const** *ca_poly_t poly2*, *ca_ctx_t ctx*)
Sets *res* to the difference of *poly1* and *poly2*.

void **_ca_poly_mul**(*ca_ptr res*, *ca_srcptr poly1*, *slong len1*, *ca_srcptr poly2*, *slong len2*, *ca_ctx_t ctx*)
void **ca_poly_mul**(*ca_poly_t res*, **const** *ca_poly_t poly1*, **const** *ca_poly_t poly2*, *ca_ctx_t ctx*)
Sets *res* to the product of *poly1* and *poly2*.

void **_ca_poly_mullow**(*ca_ptr C*, *ca_srcptr poly1*, *slong len1*, *ca_srcptr poly2*, *slong len2*, *slong n*, *ca_ctx_t ctx*)
void **ca_poly_mullow**(*ca_poly_t res*, **const** *ca_poly_t poly1*, **const** *ca_poly_t poly2*, *slong n*, *ca_ctx_t ctx*)
Sets *res* to the product of *poly1* and *poly2* truncated to length *n*.

void **ca_poly_mul_ca**(*ca_poly_t res*, **const** *ca_poly_t poly*, **const** *ca_t c*, *ca_ctx_t ctx*)
Sets *res* to *poly* multiplied by the scalar *c*.

void **ca_poly_div_ca**(*ca_poly_t res*, **const** *ca_poly_t poly*, **const** *ca_t c*, *ca_ctx_t ctx*)
Sets *res* to *poly* divided by the scalar *c*.

void **_ca_poly_divrem_basecase**(*ca_ptr Q*, *ca_ptr R*, *ca_srcptr A*, *slong lenA*, *ca_srcptr B*, *slong lenB*, **const** *ca_t invB*, *ca_ctx_t ctx*)
int **ca_poly_divrem_basecase**(*ca_poly_t Q*, *ca_poly_t R*, **const** *ca_poly_t A*, **const** *ca_poly_t B*, *ca_ctx_t ctx*)
void **_ca_poly_divrem**(*ca_ptr Q*, *ca_ptr R*, *ca_srcptr A*, *slong lenA*, *ca_srcptr B*, *slong lenB*, **const** *ca_t invB*, *ca_ctx_t ctx*)
int **ca_poly_divrem**(*ca_poly_t Q*, *ca_poly_t R*, **const** *ca_poly_t A*, **const** *ca_poly_t B*, *ca_ctx_t ctx*)
int **ca_poly_div**(*ca_poly_t Q*, **const** *ca_poly_t A*, **const** *ca_poly_t B*, *ca_ctx_t ctx*)
int **ca_poly_rem**(*ca_poly_t R*, **const** *ca_poly_t A*, **const** *ca_poly_t B*, *ca_ctx_t ctx*)
If the leading coefficient of B can be proved invertible, sets Q and R to the quotient and remainder of polynomial division of A by B and returns 1. If the leading coefficient cannot be proved invertible, returns 0. The underscore method takes a precomputed inverse of the leading coefficient of B.

void **_ca_poly_pow_ui_trunc**(*ca_ptr res*, *ca_srcptr f*, *slong flen*, *ulong exp*, *slong len*, *ca_ctx_t ctx*)
void **ca_poly_pow_ui_trunc**(*ca_poly_t res*, **const** *ca_poly_t poly*, *ulong exp*, *slong len*, *ca_ctx_t ctx*)
Sets *res* to *poly* raised to the power *exp*, truncated to length *len*.

void **_ca_poly_pow_ui**(*ca_ptr res*, *ca_srcptr f*, *slong flen*, *ulong exp*, *ca_ctx_t ctx*)
void **ca_poly_pow_ui**(*ca_poly_t res*, **const** *ca_poly_t poly*, *ulong exp*, *ca_ctx_t ctx*)
Sets *res* to *poly* raised to the power *exp*.

5.1.9 Evaluation and composition

void **_ca_poly_evaluate_horner**(*ca_t res*, *ca_srcptr f*, *slong len*, **const** *ca_t x*, *ca_ctx_t ctx*)
void **ca_poly_evaluate_horner**(*ca_t res*, **const** *ca_poly_t f*, **const** *ca_t a*, *ca_ctx_t ctx*)
void **_ca_poly_evaluate**(*ca_t res*, *ca_srcptr f*, *slong len*, **const** *ca_t x*, *ca_ctx_t ctx*)
void **ca_poly_evaluate**(*ca_t res*, **const** *ca_poly_t f*, **const** *ca_t a*, *ca_ctx_t ctx*)
Sets *res* to f evaluated at the point a.

void **_ca_poly_compose_horner**(*ca_ptr res*, *ca_srcptr poly1*, *slong len1*, *ca_srcptr poly2*, *slong len2*, *ca_ctx_t ctx*)
void **ca_poly_compose_horner**(*ca_poly_t res*, **const** *ca_poly_t poly1*, **const** *ca_poly_t poly2*, *ca_ctx_t ctx*)
void **_ca_poly_compose_divconquer**(*ca_ptr res*, *ca_srcptr poly1*, *slong len1*, *ca_srcptr poly2*, *slong len2*, *ca_ctx_t ctx*)
void **ca_poly_compose_divconquer**(*ca_poly_t res*, **const** *ca_poly_t poly1*, **const** *ca_poly_t poly2*, *ca_ctx_t ctx*)

void **_ca_poly_compose**(*ca_ptr res*, *ca_srcptr poly1*, *slong len1*, *ca_srcptr poly2*, *slong len2*, *ca_ctx_t ctx*)

void **ca_poly_compose**(*ca_poly_t res*, **const** *ca_poly_t poly1*, **const** *ca_poly_t poly2*, *ca_ctx_t ctx*)

Sets *res* to the composition of *poly1* with *poly2*.

5.1.10 Derivative and integral

void **_ca_poly_derivative**(*ca_ptr res*, *ca_srcptr poly*, *slong len*, *ca_ctx_t ctx*)

void **ca_poly_derivative**(*ca_poly_t res*, **const** *ca_poly_t poly*, *ca_ctx_t ctx*)

Sets *res* to the derivative of *poly*. The underscore method needs one less coefficient than *len* for the output array.

void **_ca_poly_integral**(*ca_ptr res*, *ca_srcptr poly*, *slong len*, *ca_ctx_t ctx*)

void **ca_poly_integral**(*ca_poly_t res*, **const** *ca_poly_t poly*, *ca_ctx_t ctx*)

Sets *res* to the integral of *poly*. The underscore method needs one more coefficient than *len* for the output array.

5.1.11 Power series division

void **_ca_poly_inv_series**(*ca_ptr res*, *ca_srcptr f*, *slong flen*, *slong len*, *ca_ctx_t ctx*)

void **ca_poly_inv_series**(*ca_poly_t res*, **const** *ca_poly_t f*, *slong len*, *ca_ctx_t ctx*)

Sets *res* to the power series inverse of f truncated to length *len*.

void **_ca_poly_div_series**(*ca_ptr res*, *ca_srcptr f*, *slong flen*, *ca_srcptr g*, *slong glen*, *slong len*, *ca_ctx_t ctx*)

void **ca_poly_div_series**(*ca_poly_t res*, **const** *ca_poly_t f*, **const** *ca_poly_t g*, *slong len*, *ca_ctx_t ctx*)

Sets *res* to the power series quotient of f and g truncated to length *len*. This function divides by zero if g has constant term zero; the user should manually remove initial zeros when an exact cancellation is required.

5.1.12 Elementary functions

void **_ca_poly_exp_series**(*ca_ptr res*, *ca_srcptr f*, *slong flen*, *slong len*, *ca_ctx_t ctx*)

void **ca_poly_exp_series**(*ca_poly_t res*, **const** *ca_poly_t f*, *slong len*, *ca_ctx_t ctx*)

Sets *res* to the power series exponential of f truncated to length *len*.

void **_ca_poly_log_series**(*ca_ptr res*, *ca_srcptr f*, *slong flen*, *slong len*, *ca_ctx_t ctx*)

void **ca_poly_log_series**(*ca_poly_t res*, **const** *ca_poly_t f*, *slong len*, *ca_ctx_t ctx*)

Sets *res* to the power series logarithm of f truncated to length *len*.

void **_ca_poly_atan_series**(*ca_ptr res*, *ca_srcptr f*, *slong flen*, *slong len*, *ca_ctx_t ctx*)

void **ca_poly_atan_series**(*ca_poly_t res*, **const** *ca_poly_t f*, *slong len*, *ca_ctx_t ctx*)

Sets *res* to the power series inverse tangent of f truncated to length *len*.

5.1.13 Greatest common divisor

slong **_ca_poly_gcd_euclidean**(*ca_ptr res*, *ca_srcptr A*, *slong lenA*, *ca_srcptr B*, *slong lenB*, *ca_ctx_t ctx*)

int **ca_poly_gcd_euclidean**(*ca_poly_t res*, **const** *ca_poly_t A*, **const** *ca_poly_t B*, *ca_ctx_t ctx*)

slong **_ca_poly_gcd**(*ca_ptr res*, *ca_srcptr A*, *slong lenA*, *ca_srcptr B*, *slong lenB*, *ca_ctx_t ctx*)

int **ca_poly_gcd**(*ca_poly_t res*, **const** *ca_poly_t A*, **const** *ca_poly_t g*, *ca_ctx_t ctx*)

Sets *res* to the GCD of A and B and returns 1 on success. On failure, returns 0 leaving the value of *res* arbitrary. The computation can fail if testing a leading coefficient for zero fails in the execution of the GCD algorithm. The output is normalized to be monic if it is not the zero polynomial.

The underscore methods assume lenA $\geq$ lenB ≥ 1, and that both A and B have nonzero leading coefficient. They return the length of the GCD, or 0 if the computation fails.

The *euclidean* version implements the standard Euclidean algorithm. The default version first checks for rational polynomials or attempts to certify numerically that the polynomials are coprime and otherwise falls back to an automatic choice of algorithm (currently only the Euclidean algorithm).

5.1.14 Roots and factorization

int **ca_poly_factor_squarefree**(*ca_t c*, *ca_poly_vec_t fac*, *ulong* **exp*, **const** *ca_poly_t F*, *ca_ctx_t ctx*)

Computes the squarefree factorization of F, giving a product $F = cf_1 f_2^2 \dots f_n^n$ where all f_i with $f_i \neq 1$ are squarefree and pairwise coprime. The nontrivial factors f_i are written to *fac* and the corresponding exponents are written to *exp*. This algorithm can fail if GCD computation fails internally. Returns 1 on success and 0 on failure.

int **ca_poly_squarefree_part**(*ca_poly_t res*, **const** *ca_poly_t poly*, *ca_ctx_t ctx*)

Sets *res* to the squarefree part of *poly*, normalized to be monic. This algorithm can fail if GCD computation fails internally. Returns 1 on success and 0 on failure.

void **_ca_poly_set_roots**(*ca_ptr poly*, *ca_srcptr roots*, **const** *ulong* **exp*, *slong n*, *ca_ctx_t ctx*)

void **ca_poly_set_roots**(*ca_poly_t poly*, *ca_vec_t roots*, **const** *ulong* **exp*, *ca_ctx_t ctx*)

Sets *poly* to the monic polynomial with the n roots given in the vector *roots*, with multiplicities given in the vector *exp*. In other words, this constructs the polynomial $(x - r_0)^{e_0}(x - r_1)^{e_1} \cdots (x - r_{n-1})^{e_{n-1}}$. Uses binary splitting.

int **_ca_poly_roots**(*ca_ptr roots*, *ca_srcptr poly*, *slong len*, *ca_ctx_t ctx*)

int **ca_poly_roots**(*ca_vec_t roots*, *ulong* **exp*, **const** *ca_poly_t poly*, *ca_ctx_t ctx*)

Attempts to compute all complex roots of the given polynomial *poly*. On success, returns 1 and sets *roots* to a vector containing all the distinct roots with corresponding multiplicities in *exp*. On failure, returns 0 and leaves the values in *roots* arbitrary. The roots are returned in arbitrary order.

Failure will occur if the leading coefficient of *poly* cannot be proved to be nonzero, if determining the correct multiplicities fails, or if the builtin algorithms do not have a means to represent the roots symbolically.

The underscore method assumes that the polynomial is squarefree. The non-underscore method performs a squarefree factorization.

5.1.15 Vectors of polynomials

type ca_poly_vec_struct

type ca_poly_vec_t
Represents a vector of polynomials.

ca_poly_struct ***_ca_poly_vec_init**(*slong len*, *ca_ctx_t ctx*)

void **ca_poly_vec_init**(*ca_poly_vec_t res*, *slong len*, *ca_ctx_t ctx*)
Initializes a vector with *len* polynomials.

void **_ca_poly_vec_fit_length**(*ca_poly_vec_t vec*, *slong len*, *ca_ctx_t ctx*)
Allocates space for *len* polynomials in *vec*.

void **ca_poly_vec_set_length**(*ca_poly_vec_t vec*, *slong len*, *ca_ctx_t ctx*)
Resizes *vec* to length *len*, zero-extending if needed.

void **_ca_poly_vec_clear**(*ca_poly_struct* **vec*, *slong len*, *ca_ctx_t ctx*)

void **ca_poly_vec_clear**(*ca_poly_vec_t vec*, *ca_ctx_t ctx*)
Clears the vector *vec*.

void **ca_poly_vec_append**(*ca_poly_vec_t vec*, **const** *ca_poly_t poly*, *ca_ctx_t ctx*)
Appends *poly* to the end of the vector *vec*.

5.2 ca_mat.h – matrices over the real and complex numbers

A *ca_mat_t* represents a dense matrix over the real or complex numbers, implemented as an array of entries of type *ca_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.

5.2.1 Types, macros and constants

type ca_mat_struct

type ca_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).

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

ca_mat_entry(*mat*, *i*, *j*)
Macro giving a pointer to the entry at row i and column j.

ca_mat_nrows(*mat*)
Returns the number of rows of the matrix.

ca_mat_ncols(*mat*)
Returns the number of columns of the matrix.

ca_ptr **ca_mat_entry_ptr**(*ca_mat_t mat*, *slong i*, *slong j*)
Returns a pointer to the entry at row i and column j. Equivalent to *ca_mat_entry* but implemented as a function.

5.2.2 Memory management

void **ca_mat_init**(*ca_mat_t mat*, *slong r*, *slong c*, *ca_ctx_t ctx*)
Initializes the matrix, setting it to the zero matrix with r rows and c columns.

void **ca_mat_clear**(*ca_mat_t mat*, *ca_ctx_t ctx*)
Clears the matrix, deallocating all entries.

void **ca_mat_swap**(*ca_mat_t mat1*, *ca_mat_t mat2*, *ca_ctx_t ctx*)
Efficiently swaps *mat1* and *mat2*.

void **ca_mat_window_init**(*ca_mat_t window*, **const** *ca_mat_t mat*, *slong r1*, *slong c1*, *slong r2*, *slong c2*, *ca_ctx_t ctx*)
Initializes *window* to a window matrix into the submatrix of *mat* starting at the corner at row *r1* and column *c1* (inclusive) and ending at row *r2* and column *c2* (exclusive).

void **ca_mat_window_clear**(*ca_mat_t window*, *ca_ctx_t ctx*)
Frees the window matrix.

5.2.3 Assignment and conversions

void **ca_mat_set**(*ca_mat_t dest*, **const** *ca_mat_t src*, *ca_ctx_t ctx*)

void **ca_mat_set_fmpz_mat**(*ca_mat_t dest*, **const** *fmpz_mat_t src*, *ca_ctx_t ctx*)

void **ca_mat_set_fmpq_mat**(*ca_mat_t dest*, **const** *fmpq_mat_t src*, *ca_ctx_t ctx*)
Sets *dest* to *src*. The operands must have identical dimensions.

void **ca_mat_set_ca**(*ca_mat_t mat*, **const** *ca_t c*, *ca_ctx_t ctx*)
Sets *mat* to the matrix with the scalar *c* on the main diagonal and zeros elsewhere.

void **ca_mat_transfer**(*ca_mat_t res*, *ca_ctx_t res_ctx*, **const** *ca_mat_t src*, *ca_ctx_t src_ctx*)
Sets *res* to *src* where the corresponding context objects *res_ctx* and *src_ctx* may be different.

This operation preserves the mathematical value represented by *src*, but may result in a different internal representation depending on the settings of the context objects.

5.2.4 Random generation

void **ca_mat_randtest**(*ca_mat_t mat*, flint_rand_t *state*, *slong depth*, *slong bits*, *ca_ctx_t ctx*)
Sets *mat* to a random matrix with entries having complexity up to *depth* and *bits* (see *ca_randtest()*).

void **ca_mat_randtest_rational**(*ca_mat_t mat*, flint_rand_t *state*, *slong bits*, *ca_ctx_t ctx*)
Sets *mat* to a random rational matrix with entries up to *bits* bits in size.

void **ca_mat_randops**(*ca_mat_t mat*, flint_rand_t *state*, *slong count*, *ca_ctx_t ctx*)
Randomizes *mat* in-place by performing elementary row or column operations. More precisely, at most count random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, the determinant) unchanged.

5.2.5 Input and output

void **ca_mat_print**(**const** *ca_mat_t mat*, *ca_ctx_t ctx*)
Prints *mat* to standard output. The entries are printed on separate lines.

void **ca_mat_printn**(**const** *ca_mat_t mat*, *slong digits*, *ca_ctx_t ctx*)
Prints a decimal representation of *mat* with precision specified by *digits*. The entries are comma-separated with square brackets and comma separation for the rows.

5.2.6 Special matrices

void **ca_mat_zero**(*ca_mat_t mat*, *ca_ctx_t ctx*)
Sets all entries in *mat* to zero.

void **ca_mat_one**(*ca_mat_t mat*, *ca_ctx_t ctx*)
Sets the entries on the main diagonal of *mat* to one, and all other entries to zero.

void **ca_mat_ones**(*ca_mat_t mat*, *ca_ctx_t ctx*)
Sets all entries in *mat* to one.

void **ca_mat_pascal**(*ca_mat_t mat*, int *triangular*, *ca_ctx_t ctx*)
Sets *mat* to a Pascal matrix, whose entries are binomial coefficients. If *triangular* is 0, constructs a full symmetric matrix with the rows of Pascal's triangle as successive antidiagonals. If *triangular* is 1, constructs the upper triangular matrix with the rows of Pascal's triangle as columns, and if *triangular* is -1, constructs the lower triangular matrix with the rows of Pascal's triangle as rows.

void **ca_mat_stirling**(*ca_mat_t mat*, int *kind*, *ca_ctx_t ctx*)
Sets *mat* to a Stirling matrix, whose entries are Stirling numbers. If *kind* is 0, the entries are set to the unsigned Stirling numbers of the first kind. If *kind* is 1, the entries are set to the signed

Stirling numbers of the first kind. If *kind* is 2, the entries are set to the Stirling numbers of the second kind.

void **ca_mat_hilbert**(*ca_mat_t mat*, *ca_ctx_t ctx*)
Sets *mat* to the Hilbert matrix, which has entries $A_{i,j} = 1/(i+j+1)$.

void **ca_mat_dft**(*ca_mat_t mat*, int *type*, *ca_ctx_t ctx*)
Sets *mat* to the DFT (discrete Fourier transform) matrix of order n where n is the smallest dimension of *mat* (if *mat* is not square, the matrix is extended periodically along the larger dimension). The *type* parameter selects between four different versions of the DFT matrix (in which $\omega = e^{2\pi i/n}$):

- Type 0 – entries $A_{j,k} = \omega^{-jk}$
- Type 1 – entries $A_{j,k} = \omega^{jk}/n$
- Type 2 – entries $A_{j,k} = \omega^{-jk}/\sqrt{n}$
- Type 3 – entries $A_{j,k} = \omega^{jk}/\sqrt{n}$

The type 0 and 1 matrices are inverse pairs, and similarly for the type 2 and 3 matrices.

5.2.7 Comparisons and properties

truth_t **ca_mat_check_equal**(**const** *ca_mat_t A*, **const** *ca_mat_t B*, *ca_ctx_t ctx*)
Compares A and B for equality.

truth_t **ca_mat_check_is_zero**(**const** *ca_mat_t A*, *ca_ctx_t ctx*)
Tests if A is the zero matrix.

truth_t **ca_mat_check_is_one**(**const** *ca_mat_t A*, *ca_ctx_t ctx*)
Tests if A has ones on the main diagonal and zeros elsewhere.

5.2.8 Conjugate and transpose

void **ca_mat_transpose**(*ca_mat_t res*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Sets *res* to the transpose of A.

void **ca_mat_conj**(*ca_mat_t res*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Sets *res* to the entrywise complex conjugate of A.

void **ca_mat_conj_transpose**(*ca_mat_t res*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Sets *res* to the conjugate transpose (Hermitian transpose) of A.

5.2.9 Arithmetic

void **ca_mat_neg**(*ca_mat_t res*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Sets *res* to the negation of A.

void **ca_mat_add**(*ca_mat_t res*, **const** *ca_mat_t A*, **const** *ca_mat_t B*, *ca_ctx_t ctx*)
Sets *res* to the sum of A and B.

void **ca_mat_sub**(*ca_mat_t res*, **const** *ca_mat_t A*, **const** *ca_mat_t B*, *ca_ctx_t ctx*)
Sets *res* to the difference of A and B.

void **ca_mat_mul_classical**(*ca_mat_t res*, **const** *ca_mat_t A*, **const** *ca_mat_t B*, *ca_ctx_t ctx*)
void **ca_mat_mul_same_nf**(*ca_mat_t res*, **const** *ca_mat_t A*, **const** *ca_mat_t B*, *ca_field_t K*, *ca_ctx_t ctx*)
void **ca_mat_mul**(*ca_mat_t res*, **const** *ca_mat_t A*, **const** *ca_mat_t B*, *ca_ctx_t ctx*)
Sets *res* to the matrix product of A and B. The *classical* version uses classical multiplication. The *same_nf* version assumes (not checked) that both A and B have coefficients in the same simple algebraic number field K or in $\mathbb{Q}$. The default version chooses an algorithm automatically.

void **ca_mat_mul_si**(*ca_mat_t B*, **const** *ca_mat_t A*, *slong c*, *ca_ctx_t ctx*)

void **ca_mat_mul_fmpz**(*ca_mat_t B*, **const** *ca_mat_t A*, **const** *fmpz_t c*, *ca_ctx_t ctx*)

void **ca_mat_mul_fmpq**(*ca_mat_t B*, **const** *ca_mat_t A*, **const** *fmpq_t c*, *ca_ctx_t ctx*)

void **ca_mat_mul_ca**(*ca_mat_t B*, **const** *ca_mat_t A*, **const** *ca_t c*, *ca_ctx_t ctx*)
Sets B to A multiplied by the scalar c.

void **ca_mat_div_si**(*ca_mat_t B*, **const** *ca_mat_t A*, *slong c*, *ca_ctx_t ctx*)

void **ca_mat_div_fmpz**(*ca_mat_t B*, **const** *ca_mat_t A*, **const** *fmpz_t c*, *ca_ctx_t ctx*)

void **ca_mat_div_fmpq**(*ca_mat_t B*, **const** *ca_mat_t A*, **const** *fmpq_t c*, *ca_ctx_t ctx*)

void **ca_mat_div_ca**(*ca_mat_t B*, **const** *ca_mat_t A*, **const** *ca_t c*, *ca_ctx_t ctx*)
Sets B to A divided by the scalar c.

void **ca_mat_add_ca**(*ca_mat_t B*, **const** *ca_mat_t A*, **const** *ca_t c*, *ca_ctx_t ctx*)

void **ca_mat_sub_ca**(*ca_mat_t B*, **const** *ca_mat_t A*, **const** *ca_t c*, *ca_ctx_t ctx*)
Sets B to A plus or minus the scalar c (interpreted as a diagonal matrix).

void **ca_mat_addmul_ca**(*ca_mat_t B*, **const** *ca_mat_t A*, **const** *ca_t c*, *ca_ctx_t ctx*)

void **ca_mat_submul_ca**(*ca_mat_t B*, **const** *ca_mat_t A*, **const** *ca_t c*, *ca_ctx_t ctx*)
Sets the matrix B to B plus (or minus) the matrix A multiplied by the scalar c.

5.2.10 Powers

void **ca_mat_sqr**(*ca_mat_t B*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Sets B to the square of A.

void **ca_mat_pow_ui_binexp**(*ca_mat_t B*, **const** *ca_mat_t A*, *ulong exp*, *ca_ctx_t ctx*)
Sets B to A raised to the power *exp*, evaluated using binary exponentiation.

5.2.11 Polynomial evaluation

void **_ca_mat_ca_poly_evaluate**(*ca_mat_t res*, *ca_srcptr poly*, *slong len*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)

void **ca_mat_ca_poly_evaluate**(*ca_mat_t res*, **const** *ca_poly_t poly*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Sets *res* to $f(A)$ where f is the polynomial given by *poly* and A is a square matrix. Uses the Paterson-Stockmeyer algorithm.

5.2.12 Gaussian elimination and LU decomposition

truth_t **ca_mat_find_pivot**(*slong *pivot_row*, *ca_mat_t mat*, *slong start_row*, *slong end_row*, *slong column*, *ca_ctx_t ctx*)
Attempts to find a nonzero entry in *mat* with column index *column* and row index between *start_row* (inclusive) and *end_row* (exclusive).

If the return value is `T_TRUE`, such an element exists, and *pivot_row* is set to the row index. If the return value is `T_FALSE`, no such element exists (all entries in this part of the column are zero). If the return value is `T_UNKNOWN`, it is unknown whether such an element exists (zero certification failed).

This function is destructive: any elements that are nontrivially zero but can be certified zero will be overwritten by exact zeros.

int **ca_mat_lu_classical**(*slong *rank*, *slong *P*, *ca_mat_t LU*, **const** *ca_mat_t A*, int *rank_check*, *ca_ctx_t ctx*)

int **ca_mat_lu_recursive**(*slong *rank*, *slong *P*, *ca_mat_t LU*, **const** *ca_mat_t A*, int *rank_check*, *ca_ctx_t ctx*)

int **ca_mat_lu**(*slong* *rank, *slong* *P, *ca_mat_t* LU, **const** *ca_mat_t* A, int *rank_check*, *ca_ctx_t* ctx)

Computes a generalized LU decomposition $A = PLU$ of a given matrix A, writing the rank of A to *rank*.

If A is a nonsingular square matrix, LU will be set to a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r, U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and set the rank to 0 if A is detected to be rank-deficient.

The algorithm can fail if it fails to certify that a pivot element is zero or nonzero, in which case the correct rank cannot be determined. The return value is 1 on success and 0 on failure. On failure, the data in the output variables `rank`, `P` and `LU` will be meaningless.

The *classical* version uses iterative Gaussian elimination. The *recursive* version uses a block recursive algorithm to take advantage of fast matrix multiplication.

int **ca_mat_fflu**(*slong* *rank, *slong* *P, *ca_mat_t* LU, *ca_t* den, **const** *ca_mat_t* A, int *rank_check*, *ca_ctx_t* ctx)

Similar to *ca_mat_lu()*, but computes a fraction-free LU decomposition using the Bareiss algorithm. The denominator is written to *den*. Note that despite being "fraction-free", this algorithm may introduce fractions due to incomplete symbolic simplifications.

truth_t **ca_mat_nonsingular_lu**(*slong* *P, *ca_mat_t* LU, **const** *ca_mat_t* A, *ca_ctx_t* ctx)

Wrapper for *ca_mat_lu()*. If A can be proved to be invertible/nonsingular, returns `T_TRUE` and sets P and LU to a LU decomposition $A = PLU$. If A can be proved to be singular, returns `T_FALSE`. If A cannot be proved to be either singular or nonsingular, returns `T_UNKNOWN`. When the return value is `T_FALSE` or `T_UNKNOWN`, the LU factorization is not completed and the values of P and LU are arbitrary.

truth_t **ca_mat_nonsingular_fflu**(*slong* *P, *ca_mat_t* LU, *ca_t* den, **const** *ca_mat_t* A, *ca_ctx_t* ctx)

Wrapper for *ca_mat_fflu()*. Similar to *ca_mat_nonsingular_lu()*, but computes a fraction-free LU decomposition using the Bareiss algorithm. The denominator is written to *den*. Note that despite being "fraction-free", this algorithm may introduce fractions due to incomplete symbolic simplifications.

5.2.13 Solving and inverse

truth_t **ca_mat_inv**(*ca_mat_t* X, **const** *ca_mat_t* A, *ca_ctx_t* ctx)

Determines if the square matrix A is nonsingular, and if successful, sets $X = A^{-1}$ and returns `T_TRUE`. Returns `T_FALSE` if A is singular, and `T_UNKNOWN` if the rank of A cannot be determined.

truth_t **ca_mat_nonsingular_solve_adjugate**(*ca_mat_t* X, **const** *ca_mat_t* A, **const** *ca_mat_t* B, *ca_ctx_t* ctx)

truth_t **ca_mat_nonsingular_solve_fflu**(*ca_mat_t* X, **const** *ca_mat_t* A, **const** *ca_mat_t* B, *ca_ctx_t* ctx)

truth_t **ca_mat_nonsingular_solve_lu**(*ca_mat_t* X, **const** *ca_mat_t* A, **const** *ca_mat_t* B, *ca_ctx_t* ctx)

truth_t **ca_mat_nonsingular_solve**(*ca_mat_t* X, **const** *ca_mat_t* A, **const** *ca_mat_t* B, *ca_ctx_t* ctx)

Determines if the square matrix A is nonsingular, and if successful, solves $AX = B$ and returns `T_TRUE`. Returns `T_FALSE` if A is singular, and `T_UNKNOWN` if the rank of A cannot be determined.

void **ca_mat_solve_tril_classical**(*ca_mat_t* X, **const** *ca_mat_t* L, **const** *ca_mat_t* B, int *unit*, *ca_ctx_t* ctx)

void **ca_mat_solve_tril_recursive**(*ca_mat_t* X, **const** *ca_mat_t* L, **const** *ca_mat_t* B, int *unit*, *ca_ctx_t* ctx)

void **ca_mat_solve_tril**(*ca_mat_t X*, **const** *ca_mat_t L*, **const** *ca_mat_t B*, int *unit*, *ca_ctx_t ctx*)
void **ca_mat_solve_triu_classical**(*ca_mat_t X*, **const** *ca_mat_t U*, **const** *ca_mat_t B*, int *unit*, *ca_ctx_t ctx*)
void **ca_mat_solve_triu_recursive**(*ca_mat_t X*, **const** *ca_mat_t U*, **const** *ca_mat_t B*, int *unit*, *ca_ctx_t ctx*)
void **ca_mat_solve_triu**(*ca_mat_t X*, **const** *ca_mat_t U*, **const** *ca_mat_t B*, int *unit*, *ca_ctx_t ctx*)

Solves the lower triangular system $LX = B$ or the upper triangular system $UX = B$, respectively. It is assumed (not checked) that the diagonal entries are nonzero. If *unit* is set, the main diagonal of L or U is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The *classical* versions perform the computations iteratively while the *recursive* versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

void **ca_mat_solve_fflu_precomp**(*ca_mat_t X*, **const** *slong* *perm, **const** *ca_mat_t A*, **const** *ca_t den*, **const** *ca_mat_t B*, *ca_ctx_t ctx*)
void **ca_mat_solve_lu_precomp**(*ca_mat_t X*, **const** *slong* *P, **const** *ca_mat_t LU*, **const** *ca_mat_t B*, *ca_ctx_t ctx*)

Solves $AX = B$ given the precomputed nonsingular LU decomposition $A = PLU$ or fraction-free LU decomposition with denominator *den*. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with LU.

5.2.14 Rank and echelon form

int **ca_mat_rank**(*slong* *rank, **const** *ca_mat_t A*, *ca_ctx_t ctx*)

Computes the rank of the matrix A. If successful, returns 1 and writes the rank to `rank`. If unsuccessful, returns 0.

int **ca_mat_rref_fflu**(*slong* *rank, *ca_mat_t R*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
int **ca_mat_rref_lu**(*slong* *rank, *ca_mat_t R*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
int **ca_mat_rref**(*slong* *rank, *ca_mat_t R*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)

Computes the reduced row echelon form (rref) of a given matrix. On success, sets R to the rref of A, writes the rank to *rank*, and returns 1. On failure to certify the correct rank, returns 0, leaving the data in *rank* and R meaningless.

The *fflu* version computes a fraction-free LU decomposition and then converts the output ro rref form. The *lu* version computes a regular LU decomposition and then converts the output to rref form. The default version uses an automatic algorithm choice and may implement additional methods for special cases.

int **ca_mat_right_kernel**(*ca_mat_t X*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)

Sets X to a basis of the right kernel (nullspace) of A. The output matrix X will be resized in-place to have a number of columns equal to the nullity of A. Returns 1 on success. On failure, returns 0 and leaves the data in X meaningless.

5.2.15 Determinant and trace

void **ca_mat_trace**(*ca_t trace*, **const** *ca_mat_t mat*, *ca_ctx_t ctx*)

Sets *trace* to the sum of the entries on the main diagonal of *mat*.

void **ca_mat_det_berkowitz**(*ca_t det*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
int **ca_mat_det_lu**(*ca_t det*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
int **ca_mat_det_bareiss**(*ca_t det*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
void **ca_mat_det_cofactor**(*ca_t det*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)

void **ca_mat_det**(*ca_t det*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Sets *det* to the determinant of the square matrix A. Various algorithms are available:

- The *berkowitz* version uses the division-free Berkowitz algorithm performing $O(n^4)$ operations. Since no zero tests are required, it is guaranteed to succeed.
- The *cofactor* version performs cofactor expansion. This is currently only supported for matrices up to size 4.
- The *lu* and *bareiss* versions use rational LU decomposition and fraction-free LU decomposition (Bareiss algorithm) respectively, requiring $O(n^3)$ operations. These algorithms can fail if zero certification fails (see *ca_mat_nonsingular_lu()*); they return 1 for success and 0 for failure. Note that the Bareiss algorithm, despite being "fraction-free", may introduce fractions due to incomplete symbolic simplifications.

The default function chooses an algorithm automatically. It will, in addition, recognize trivially rational and integer matrices and evaluate those determinants using *fmpq_mat_t* or *fmpz_mat_t*.

The various algorithms can produce different symbolic forms of the same determinant. Which algorithm performs better depends strongly and sometimes unpredictably on the structure of the matrix.

void **ca_mat_adjugate_cofactor**(*ca_mat_t adj*, *ca_t det*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)

void **ca_mat_adjugate_charpoly**(*ca_mat_t adj*, *ca_t det*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)

void **ca_mat_adjugate**(*ca_mat_t adj*, *ca_t det*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Sets *adj* to the adjuate matrix of A and *det* to the determinant of A, both computed simultaneously. The *cofactor* version uses cofactor expansion. The *charpoly* version computes and evaluates the characteristic polynomial. The default version uses an automatic algorithm choice.

5.2.16 Characteristic polynomial

void **_ca_mat_charpoly_berkowitz**(*ca_ptr cp*, **const** *ca_mat_t mat*, *ca_ctx_t ctx*)

void **ca_mat_charpoly_berkowitz**(*ca_poly_t cp*, **const** *ca_mat_t mat*, *ca_ctx_t ctx*)

int **_ca_mat_charpoly_danilevsky**(*ca_ptr cp*, **const** *ca_mat_t mat*, *ca_ctx_t ctx*)

int **ca_mat_charpoly_danilevsky**(*ca_poly_t cp*, **const** *ca_mat_t mat*, *ca_ctx_t ctx*)

void **_ca_mat_charpoly**(*ca_ptr cp*, **const** *ca_mat_t mat*, *ca_ctx_t ctx*)

void **ca_mat_charpoly**(*ca_poly_t cp*, **const** *ca_mat_t mat*, *ca_ctx_t ctx*)
Sets *poly* to the characteristic polynomial of *mat* which must be a square matrix. If the matrix has n rows, the underscore method requires space for $n + 1$ output coefficients.

The *berkowitz* version uses a division-free algorithm requiring $O(n^4)$ operations. The *danilevsky* version only performs $O(n^3)$ operations, but performs divisions and needs to check for zero which can fail. This version returns 1 on success and 0 on failure. The default version chooses an algorithm automatically.

int **ca_mat_companion**(*ca_mat_t mat*, **const** *ca_poly_t poly*, *ca_ctx_t ctx*)
Sets *mat* to the companion matrix of *poly*. This function verifies that the leading coefficient of *poly* is provably nonzero and that the output matrix has the right size, returning 1 on success. It returns 0 if the leading coefficient of *poly* cannot be proved nonzero or if the size of the output matrix does not match.

5.2.17 Eigenvalues and eigenvectors

int **ca_mat_eigenvalues**(*ca_vec_t lambda*, *ulong* **exp*, **const** *ca_mat_t mat*, *ca_ctx_t ctx*)
Attempts to compute all complex eigenvalues of the given matrix *mat*. On success, returns 1 and sets *lambda* to the distinct eigenvalues with corresponding multiplicities in *exp*. The eigenvalues are returned in arbitrary order. On failure, returns 0 and leaves the values in *lambda* and *exp* arbitrary.

This function effectively computes the characteristic polynomial and then calls *ca_poly_roots*.

truth_t **ca_mat_diagonalization**(*ca_mat_t D*, *ca_mat_t P*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Matrix diagonalization: attempts to compute a diagonal matrix D and an invertible matrix P such that $A = PDP^{-1}$. Returns `T_TRUE` if A is diagonalizable and the computation succeeds, `T_FALSE` if A is provably not diagonalizable, and `T_UNKNOWN` if it is unknown whether A is diagonalizable. If the return value is not `T_TRUE`, the values in D and P are arbitrary.

5.2.18 Jordan canonical form

int **ca_mat_jordan_blocks**(*ca_vec_t lambda*, *slong* **num_blocks*, *slong* **block_lambda*, *slong* **block_size*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Computes the blocks of the Jordan canonical form of A. On success, returns 1 and sets *lambda* to the unique eigenvalues of A, sets *num_blocks* to the number of Jordan blocks, entry i of *block_lambda* to the index of the eigenvalue in Jordan block i, and entry i of *block_size* to the size of Jordan block i. On failure, returns 0, leaving arbitrary values in the output variables. The user should allocate space in *block_lambda* and *block_size* for up to n entries where n is the size of the matrix.

The Jordan form is unique up to the ordering of blocks, which is arbitrary.

void **ca_mat_set_jordan_blocks**(*ca_mat_t mat*, **const** *ca_vec_t lambda*, *slong num_blocks*, *slong* **block_lambda*, *slong* **block_size*, *ca_ctx_t ctx*)
Sets *mat* to the concatenation of the Jordan blocks given in *lambda*, *num_blocks*, *block_lambda* and *block_size*. See *ca_mat_jordan_blocks()* for an explanation of these variables.

int **ca_mat_jordan_transformation**(*ca_mat_t mat*, **const** *ca_vec_t lambda*, *slong num_blocks*, *slong* **block_lambda*, *slong* **block_size*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Given the precomputed Jordan block decomposition (*lambda*, *num_blocks*, *block_lambda*, *block_size*) of the square matrix A, computes the corresponding transformation matrix P such that $A = PJP^{-1}$. On success, writes P to *mat* and returns 1. On failure, returns 0, leaving the value of *mat* arbitrary.

int **ca_mat_jordan_form**(*ca_mat_t J*, *ca_mat_t P*, **const** *ca_mat_t A*, *ca_ctx_t ctx*)
Computes the Jordan decomposition $A = PJP^{-1}$ of the given square matrix A. The user can pass *NULL* for the output variable P, in which case only J is computed. On success, returns 1. On failure, returns 0, leaving the values of J and P arbitrary.

This function is a convenience wrapper around *ca_mat_jordan_blocks()*, *ca_mat_set_jordan_blocks()* and *ca_mat_jordan_transformation()*. For computations with the Jordan decomposition, it is often better to use those methods directly since they give direct access to the spectrum and block structure.

5.2.19 Matrix functions

int **ca_mat_exp**(*ca_mat_t* *res*, **const** *ca_mat_t* *A*, *ca_ctx_t* *ctx*)

Matrix exponential: given a square matrix A, sets *res* to e^A and returns 1 on success. If unsuccessful, returns 0, leaving the values in *res* arbitrary.

This function uses Jordan decomposition. The matrix exponential always exists, but computation can fail if computing the Jordan decomposition fails.

truth_t **ca_mat_log**(*ca_mat_t* *res*, **const** *ca_mat_t* *A*, *ca_ctx_t* *ctx*)

Matrix logarithm: given a square matrix A, sets *res* to a logarithm $\log(A)$ and returns `T_TRUE` on success. If A can be proved to have no logarithm, returns `T_FALSE`. If the existence of a logarithm cannot be proved, returns `T_UNKNOWN`.

This function uses the Jordan decomposition, and the branch of the matrix logarithm is defined by taking the principal values of the logarithms of all eigenvalues.

CHAPTER

SIX

FIELD AND EXTENSION NUMBER CONSTRUCTIONS

These modules are used internally by the *ca_t* type to construct towers of algebraic and transcendental number fields. The user does not normally need to use these modules directly outside of advanced applications requiring inspection of the symbolic representations of numbers.

6.1 ca_ext.h – real and complex extension numbers

A *ca_ext_t* represents a fixed real or complex number a. The content of a *ca_ext_t* can be one of the following:

- An algebraic constant represented in canonical form by a *qqbar_t* instance (example: i, represented as the root of $x^2 + 1$ with positive imaginary part).
- A constant of the form $f(x_1, \ldots, x_n)$ where f is a builtin symbolic function and $x_1, \ldots, x_n$ are given *ca_t* instances.
- A builtin symbolic constant such as π. (This is just a special case of the above with a zero-length argument list.)
- (Not implemented): a user-defined constant or function defined by suppling a function pointer for Arb numerical evaluation to specified precision.

The *ca_ext_t* structure is heavy-weight object, not just meant to act as a node in a symbolic expression. It will cache various data to support repeated computation with this particular number, including its numerical enclosure and number field data in the case of algebraic numbers.

Extension numbers are used internally by the *ca_t* type to define the embeddings $\mathbb{Q}(a) \to \mathbb{C}$ of formal fields. The user does not normally need to create *ca_ext_t* instances directly; the intended way for the user to work with the extension number a is to create a *ca_t* representing the field element $1 \cdot a$. The underlying *ca_ext_t* may be accessed to determine symbolic and numerical properties of this number.

Since extension numbers may depend recursively on nontrivial fields for function arguments, *ca_ext_t* operations require a *ca_ctx_t* context object.

6.1.1 Type and macros

For all types, a *type_t* is defined as an array of length one of type *type_struct*, permitting a *type_t* to be passed by reference.

type ca_ext_struct

type ca_ext_t
An extension number object contains a header, a hash value, data (a *qqbar_t* instance and an Antic *nf_t* in the case of algebraic numbers, and a pointer to arguments in the case of a symbolic function), and a cached *acb_t* enclosure (in the case of a *qqbar_t*, the enclosure internal to that structure is used).

type ca_ext_ptr
Alias for `ca_ext_struct *`.

type ca_ext_srcptr
Alias for `const ca_ext_struct *`.

CA_EXT_HEAD(x)
Accesses the head (a `calcium_func_code`) of x. This is *CA_QQBar* if x represents an algebraic constant in canonical form, and *CA_Exp*, *CA_Pi*, etc. for symbolic functions and constants.

CA_EXT_HASH(x)
Accesses the hash value of x.

CA_EXT_QQBAR(x)
Assuming that x represents an algebraic constant in canonical form, accesses this *qqbar_t* object.

CA_EXT_QQBAR_NF(x)
Assuming that x represents an algebraic constant in canonical form, accesses the corresponding Antic number field *nf_t* object.

CA_EXT_FUNC_ARGS(x)
Assuming that x represents a symbolic constant or function, accesses the argument list (as a *ca_ptr*).

CA_EXT_FUNC_NARGS(x)
Assuming that x represents a symbolic constant or function, accesses the number of function arguments.

CA_EXT_FUNC_ENCLOSURE(x)
Assuming that x represents a symbolic constant or function, accesses the cached *acb_t* numerical enclosure.

CA_EXT_FUNC_PREC(x)
Assuming that x represents a symbolic constant or function, accesses the working precision of the cached numerical enclosure.

6.1.2 Memory management

void **ca_ext_init_qqbar**(*ca_ext_t res*, **const** *qqbar_t x*, *ca_ctx_t ctx*)
Initializes *res* and sets it to the algebraic constant x.

void **ca_ext_init_const**(*ca_ext_t res*, calcium_func_code *func*, *ca_ctx_t ctx*)
Initializes *res* and sets it to the constant defined by *func* (example: *func* = *CA_Pi* for $x = \pi$).

void **ca_ext_init_fx**(*ca_ext_t res*, calcium_func_code *func*, **const** *ca_t x*, *ca_ctx_t ctx*)
Initializes *res* and sets it to the univariate function value $f(x)$ where f is defined by *func* (example: *func* = *CA_Exp* for e^x).

void **ca_ext_init_fxy**(*ca_ext_t res*, calcium_func_code *func*, **const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Initializes *res* and sets it to the bivariate function value $f(x, y)$ where f is defined by *func* (example: *func* = *CA_Pow* for x^y).

void **ca_ext_init_fxn**(*ca_ext_t res*, calcium_func_code *func*, *ca_srcptr x*, *slong nargs*, *ca_ctx_t ctx*)
Initializes *res* and sets it to the multivariate function value $f(x_1, \ldots, x_n)$ where f is defined by *func* and n is given by *nargs*.

void **ca_ext_init_set**(*ca_ext_t res*, **const** *ca_ext_t x*, *ca_ctx_t ctx*)
Initializes *res* and sets it to a copy of x.

void **ca_ext_clear**(*ca_ext_t res*, *ca_ctx_t ctx*)
Clears *res*.

6.1.3 Structure

slong **ca_ext_nargs(const** *ca_ext_t x*, *ca_ctx_t ctx***)**
Returns the number of function arguments of x. The return value is 0 for any algebraic constant and for any built-in symbolic constant such as π.

void **ca_ext_get_arg(***ca_t res*, **const** *ca_ext_t x*, *slong i*, *ca_ctx_t ctx***)**
Sets *res* to argument i (indexed from zero) of x. This calls *flint_abort* if i is out of range.

ulong **ca_ext_hash(const** *ca_ext_t x*, *ca_ctx_t ctx***)**
Returns a hash of the structural representation of x.

int **ca_ext_equal_repr(const** *ca_ext_t x*, **const** *ca_ext_t y*, *ca_ctx_t ctx***)**
Tests x and y for structural equality, returning 0 (false) or 1 (true).

int **ca_ext_cmp_repr(const** *ca_ext_t x*, **const** *ca_ext_t y*, *ca_ctx_t ctx***)**
Compares the representations of x and y in a canonical sort order, returning -1, 0 or 1. This only performs a structural comparison of the symbolic representations; the return value does not say anything meaningful about the numbers represented by x and y.

6.1.4 Input and output

void **ca_ext_print(const** *ca_ext_t x*, **const** *ca_ctx_t ctx***)**
Prints a description of x to standard output.

6.1.5 Numerical evaluation

void **ca_ext_get_acb_raw(***acb_t res*, *ca_ext_t x*, *slong prec*, *ca_ctx_t ctx***)**
Sets *res* to an enclosure of the numerical value of x. A working precision of *prec* bits is used for the evaluation, without adaptive refinement.

6.1.6 Cache

type ca_ext_cache_struct

type ca_ext_cache_t
Represents a set of structurally distinct *ca_ext_t* instances. This object contains an array of pointers to individual heap-allocated *ca_ext_struct* objects as well as a hash table for quick lookup.

void **ca_ext_cache_init(***ca_ext_cache_t cache*, *ca_ctx_t ctx***)**
Initializes *cache* for use.

void **ca_ext_cache_clear(***ca_ext_cache_t cache*, *ca_ctx_t ctx***)**
Clears *cache*, freeing the memory allocated internally.

ca_ext_ptr **ca_ext_cache_insert(***ca_ext_cache_t cache*, **const** *ca_ext_t x*, *ca_ctx_t ctx***)**
Adds x to *cache* without duplication. If a structurally identical instance already exists in *cache*, a pointer to that instance is returned. Otherwise, a copy of x is inserted into *cache* and a pointer to that new instance is returned.

6.2 ca_field.h – extension fields

A *ca_field_t* represents the parent field $K = \mathbb{Q}(a_1, \ldots, a_n)$ of a *ca_t* element. A *ca_field_t* contains a list of pointers to *ca_ext_t* objects as well as a reduction ideal.

The user does not normally need to create *ca_field_t* objects manually: a *ca_ctx_t* context object manages a cache of fields automatically.

Internally, three types of field representation are used:

- The trivial field $\mathbb{Q}$.
- An Antic number field $\mathbb{Q}(a)$ where a is defined by a *qqbar_t*
- A generic field $\mathbb{Q}(a_1, \ldots, a_n)$ where $n \geq 1$, and a_1 is not defined by a *qqbar_t* if $n = 1$.

The field type mainly affects the internal storage of the field elements; the distinction is mostly transparent to the external interface.

6.2.1 Type and macros

For all types, a *type_t* is defined as an array of length one of type *type_struct*, permitting a *type_t* to be passed by reference.

type ca_field_struct

type ca_field_t
Represents a formal field.

type ca_field_ptr
Alias for `ca_field_struct *`.

type ca_field_srcptr
Alias for `const ca_field_struct *`.

CA_FIELD_LENGTH(K)
Accesses the number n of extension numbers of K. This is 0 if $K = \mathbb{Q}$.

CA_FIELD_EXT(K)
Accesses the array of extension numbers as a *ca_ext_ptr*.

CA_FIELD_EXT_ELEM(K, i)
Accesses the extension number at position i (indexed from zero) as a *ca_ext_t*.

CA_FIELD_HASH(K)
Accesses the hash value of K.

CA_FIELD_IS_QQ(K)
Returns whether K is the trivial field $\mathbb{Q}$.

CA_FIELD_IS_NF(K)
Returns whether K represents an Antic number field $K = \mathbb{Q}(a)$ where a is represented by a *qqbar_t*.

CA_FIELD_IS_GENERIC(K)
Returns whether K represents a generic field.

CA_FIELD_NF(K)
Assuming that K represents an Antic number field $K = \mathbb{Q}(a)$, accesses the *nf_t* object representing this field.

CA_FIELD_NF_QQBAR(K)
Assuming that K represents an Antic number field $K = \mathbb{Q}(a)$, accesses the *qqbar_t* object representing a.

CA_FIELD_IDEAL(*K*)
Assuming that K represents a multivariate field, accesses the reduction ideal as a *fmpz_mpoly_t* array.

CA_FIELD_IDEAL_ELEM(*K*, *i*)
Assuming that K represents a multivariate field, accesses element i (indexed from zero) of the reduction ideal as a *fmpz_mpoly_t*.

CA_FIELD_IDEAL_LENGTH(*K*)
Assuming that K represents a multivariate field, accesses the number of polynomials in the reduction ideal.

CA_FIELD_MCTX(*K*, *ctx*)
Assuming that K represents a multivariate field, accesses the *fmpz_mpoly_ctx_t* context object for multivariate polynomial arithmetic on the internal representation of elements in this field.

6.2.2 Memory management

void **ca_field_init_qq**(*ca_field_t K*, *ca_ctx_t ctx*)
Initializes K to represent the trivial field $\mathbb{Q}$.

void **ca_field_init_nf**(*ca_field_t K*, **const** *qqbar_t x*, *ca_ctx_t ctx*)
Initializes K to represent the algebraic number field $\mathbb{Q}(x)$.

void **ca_field_init_const**(*ca_field_t K*, *ulong func*, *ca_ctx_t ctx*)
Initializes K to represent the field $\mathbb{Q}(x)$ where x is a builtin constant defined by *func* (example: *func* = *CA_Pi* for $x = \pi$).

void **ca_field_init_fx**(*ca_field_t K*, *ulong func*, **const** *ca_t x*, *ca_ctx_t ctx*)
Initializes K to represent the field $\mathbb{Q}(a)$ where $a = f(x)$, given a number x and a builtin univariate function *func* (example: *func* = *CA_Exp* for e^x).

void **ca_field_init_fxy**(*ca_field_t K*, calcium_func_code *func*, **const** *ca_t x*, **const** *ca_t y*, *ca_ctx_t ctx*)
Initializes K to represent the field $\mathbb{Q}(a, b)$ where $a = f(x, y)$.

void **ca_field_init_multi**(*ca_field_t K*, *slong len*, *ca_ctx_t ctx*)
Initializes K to represent a multivariate field $\mathbb{Q}(a_1, \ldots, a_n)$ in n extension numbers. The extension numbers must subsequently be assigned one by one using *ca_field_set_ext()*.

void **ca_field_set_ext**(*ca_field_t K*, *slong i*, *slong x_index*, *ca_ctx_t ctx*)
Sets the extension number at position i (here indexed from 0) of K to the generator of the field with index *x_index* in *ctx*. (It is assumed that the generating field is a univariate field.)

This only inserts a shallow reference: the field at index *x_index* must be kept alive until K has been cleared.

void **ca_field_clear**(*ca_field_t K*, *ca_ctx_t ctx*)
Clears the field K. This does not clear the individual extension numbers, which are only held as references.

6.2.3 Input and output

void **ca_field_print**(**const** *ca_field_t K*, **const** *ca_ctx_t ctx*)
Prints a description of the field K to standard output.

6.2.4 Ideal

void **ca_field_build_ideal**(*ca_field_t K*, *ca_ctx_t ctx*)
Given K with assigned extension numbers, builds the reduction ideal in-place.

void **ca_field_build_ideal_erf**(*ca_field_t K*, *ca_ctx_t ctx*)
Builds relations for error functions present among the extension numbers in K. This heuristic adds relations that are consequences of the functional equations $\operatorname{erf}(x) = -\operatorname{erf}(-x)$, $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$, $\operatorname{erfi}(x) = -i \operatorname{erf}(ix)$.

6.2.5 Structure operations

int **ca_field_cmp**(**const** *ca_field_t K1*, **const** *ca_field_t K2*, *ca_ctx_t ctx*)
Compares the field objects *K1* and *K2* in a canonical sort order, returning -1, 0 or 1. This only performs a lexicographic comparison of the representations of *K1* and *K2*; the return value does not say anything meaningful about the relative structures of *K1* and *K2* as mathematical fields.

6.2.6 Cache

type ca_field_cache_struct

type ca_field_cache_t
Represents a set of distinct *ca_field_t* instances. This object contains an array of pointers to individual heap-allocated *ca_field_struct* objects as well as a hash table for quick lookup.

void **ca_field_cache_init**(*ca_field_cache_t cache*, *ca_ctx_t ctx*)
Initializes *cache* for use.

void **ca_field_cache_clear**(*ca_field_cache_t cache*, *ca_ctx_t ctx*)
Clears *cache*, freeing the memory allocated internally. This does not clear the individual extension numbers, which are only held as references.

ca_field_ptr **ca_field_cache_insert_ext**(*ca_field_cache_t cache*, *ca_ext_struct* ***x*, *slong len*, *ca_ctx_t ctx*)
Adds the field defined by the length-*len* list of extension numbers x to *cache* without duplication. If such a field already exists in *cache*, a pointer to that instance is returned. Otherwise, a field with extension numbers x is inserted into *cache* and a pointer to that new instance is returned. Upon insertion of a new field, the reduction ideal is constructed via *ca_field_build_ideal()*.

CHAPTER

SEVEN

SYMBOLIC EXPRESSIONS

7.1 fexpr.h – flat-packed symbolic expressions

This module supports working with symbolic expressions.

7.1.1 Introduction

Formally, a symbolic expression is either:

- An atom, being one of the following:
  - An integer, for example 0 or -34.
  - A symbol, for example `x`, `Mul`, `SomeUserNamedSymbol`. Symbols should be valid C identifiers (containing only the characters `A-Z`, `a-z`, `0-9`, `_`, and not starting with a digit).
  - A string, for example `"Hello, world!"`. For the moment, we only consider ASCII strings, but there is no obstacle in principle to supporting UTF-8.
- A non-atomic expression, representing a function call $e_0(e_1, \ldots, e_n)$ where $e_0, \ldots, e_n$ are symbolic expressions.

The meaning of an expression depends on the interpretation of symbols in a given context. For example, with a standard intepretation (used within Calcium) of the symbols `Mul`, `Add` and `Neg`, the expression `Mul(3, Add(Neg(x), y))` encodes the formula $3 \cdot ((-x) + y)$ where `x` and `y` are symbolic variables. See *fexpr_builtin.h – builtin symbols* for documentation of builtin symbols.

Computing and embedding data

Symbolic expressions are usually not the best data structure to use directly for heavy-duty computations. Functions acting on symbolic expressions will typically convert to a dedicated data structure (e.g. polynomials) internally and (optionally) convert the final result back to a symbolic expression.

Symbolic expressions do not allow embedding arbitrary binary objects such as Flint/Arb/Antic/Calcium types as atoms. This is done on purpose to make symbolic expressions easy to use as a data exchange format. To embed an object in an expression, one has the following options:

- Represent the object structurally using atoms supported natively by symbolic expressions (for example, an integer polynomial can be represented as a list of coefficients or as an arithmetic expression tree).
- Introduce a dummy symbol to represent the object, maintaining an external translation table mapping this symbol to the intended value.
- Encode the object using a string or symbol name. This is generally not recommended, as it requires parsing; properly used, symbolic expressions have the benefit of being able to represent the parsed structure.

Flat-packed representation

Symbolic expressions are often implemented using trees of pointers (often together with hash tables for uniqueness), requiring some form of memory management. The *fexpr_t* type, by contrast, stores a symbolic expression using a "flat-packed" representation without internal pointers. The expression data is just an array of words (`ulong`). The first word is a header encoding type information (whether the expression is a function call or an atom, and the type of the atom) and the total number of words in the expression. For atoms, the data is stored either in the header word itself (small integers and short symbols/strings) or in the following words. For function calls, the header is followed by the expressions $e_0, \ldots, e_n$ packed contiguously in memory.

Pros:

- Memory management is trivial.
- Copying an expression is just copying an array of words.
- Comparing expressions for equality is just comparing arrays of words.
- Merging expressions is basically just concatenating arrays of words.
- Expression data can be shared freely in binary form between threads and even between machines (as long as all machines have the same word size and endianness).

Cons:

- Repeated instances of the same subexpression cannot share memory (a workaround is to introduce local dummy symbols for repeated subexpressions).
- Extracting a subexpression for modification generally requires making a complete copy of that subxepression (however, for read-only access to subexpressions, one can use "view" expressions which have zero overhead).
- Manipulating a part of an expression generally requires rebuilding the whole expression.
- Building an expression incrementally is typically $O(n^2)$. As a workaround, it is a good idea to work with balanced (low-depth) expressions and try to construct an expression in one go (for example, to create a sum, create a single `Add` expression with many arguments instead of chaining binary `Add` operations).

7.1.2 Types and macros

type fexpr_struct

type fexpr_t
An *fexpr_struct* consists of a pointer to an array of words along with a record of the number of allocated words.

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

type fexpr_ptr
Alias for `fexpr_struct *`, used for arrays of expressions.

type fexpr_srcptr
Alias for `const fexpr_struct *`, used for arrays of expressions when passed as constant input to functions.

type fexpr_vec_struct

type fexpr_vec_t
A type representing a vector of expressions with managed length. The structure contains an *fexpr_ptr entries* for the entries, an integer *length* (the size of the vector), and an integer *alloc* (the number of allocated entries).

fexpr_vec_entry(*vec*, *i*)
Returns a pointer to entry *i* in the vector *vec*.

7.1.3 Memory management

void **fexpr_init**(*fexpr_t expr*)
Initializes *expr* for use. Its value is set to the atomic integer 0.

void **fexpr_clear**(*fexpr_t expr*)
Clears *expr*, freeing its allocated memory.

fexpr_ptr **_fexpr_vec_init**(*slong len*)
Returns a heap-allocated vector of *len* initialized expressions.

void **_fexpr_vec_clear**(*fexpr_ptr vec*, *slong len*)
Clears the *len* expressions in *vec* and frees *vec* itself.

void **fexpr_fit_size**(*fexpr_t expr*, *slong size*)
Ensures that *expr* has room for *size* words.

void **fexpr_set**(*fexpr_t res*, **const** *fexpr_t expr*)
Sets *res* to the a copy of *expr*.

void **fexpr_swap**(*fexpr_t a*, *fexpr_t b*)
Swaps *a* and *b* efficiently.

7.1.4 Size information

slong **fexpr_depth**(**const** *fexpr_t expr*)
Returns the depth of *expr* as a symbolic expression tree.

slong **fexpr_num_leaves**(**const** *fexpr_t expr*)
Returns the number of leaves (atoms, counted with repetition) in the expression *expr*.

slong **fexpr_size**(**const** *fexpr_t expr*)
Returns the number of words in the internal representation of *expr*.

slong **fexpr_size_bytes**(**const** *fexpr_t expr*)
Returns the number of bytes in the internal representation of *expr*. The count excludes the size of the structure itself. Add `sizeof(fexpr_struct)` to get the size of the object as a whole.

slong **fexpr_allocated_bytes**(**const** *fexpr_t expr*)
Returns the number of allocated bytes in the internal representation of *expr*. The count excludes the size of the structure itself. Add `sizeof(fexpr_struct)` to get the size of the object as a whole.

7.1.5 Comparisons

int **fexpr_equal**(**const** *fexpr_t a*, **const** *fexpr_t b*)
Checks if *a* and *b* are exactly equal as expressions.

int **fexpr_equal_si**(**const** *fexpr_t expr*, *slong c*)

int **fexpr_equal_ui**(**const** *fexpr_t expr*, *ulong c*)
Checks if *expr* is an atomic integer exactly equal to *c*.

ulong **fexpr_hash**(**const** *fexpr_t expr*)
Returns a hash of the expression *expr*.

int **fexpr_cmp_fast**(**const** *fexpr_t a*, **const** *fexpr_t b*)
Compares *a* and *b* using an ordering based on the internal representation, returning -1, 0 or 1. This can be used, for instance, to maintain sorted arrays of expressions for binary search; the sort order has no mathematical significance.

7.1.6 Atoms

int **fexpr_is_integer**(**const** *fexpr_t expr*)
Returns whether *expr* is an atomic integer

int **fexpr_is_symbol**(**const** *fexpr_t expr*)
Returns whether *expr* is an atomic symbol.

int **fexpr_is_string**(**const** *fexpr_t expr*)
Returns whether *expr* is an atomic string.

int **fexpr_is_atom**(**const** *fexpr_t expr*)
Returns whether *expr* is any atom.

void **fexpr_zero**(*fexpr_t res*)
Sets *res* to the atomic integer 0.

int **fexpr_is_zero**(**const** *fexpr_t expr*)
Returns whether *expr* is the atomic integer 0.

int **fexpr_is_neg_integer**(**const** *fexpr_t expr*)
Returns whether *expr* is any negative atomic integer.

void **fexpr_set_si**(*fexpr_t res*, *slong c*)

void **fexpr_set_ui**(*fexpr_t res*, *ulong c*)

void **fexpr_set_fmpz**(*fexpr_t res*, **const** *fmpz_t c*)
Sets *res* to the atomic integer *c*.

void **fexpr_get_fmpz**(*fmpz_t res*, **const** *fexpr_t expr*)
Sets *res* to the atomic integer in *expr*. This aborts if *expr* is not an atomic integer.

void **fexpr_set_symbol_builtin**(*fexpr_t res*, *slong id*)
Sets *res* to the builtin symbol with internal index *id* (see *fexpr_builtin.h – builtin symbols*).

int **fexpr_is_builtin_symbol**(**const** *fexpr_t expr*, *slong id*)
Returns whether *expr* is the builtin symbol with index *id* (see *fexpr_builtin.h – builtin symbols*).

int **fexpr_is_any_builtin_symbol**(**const** *fexpr_t expr*)
Returns whether *expr* is any builtin symbol (see *fexpr_builtin.h – builtin symbols*).

void **fexpr_set_symbol_str**(*fexpr_t res*, **const** char **s*)
Sets *res* to the symbol given by *s*.

char ***fexpr_get_symbol_str**(**const** *fexpr_t expr*)
Returns the symbol in *expr* as a string. The string must be freed with `flint_free()`. This aborts if *expr* is not an atomic symbol.

void **fexpr_set_string**(*fexpr_t res*, **const** char **s*)
Sets *res* to the atomic string *s*.

char ***fexpr_get_string**(**const** *fexpr_t expr*)
Assuming that *expr* is an atomic string, returns a copy of this string. The string must be freed with `flint_free()`.

7.1.7 Input and output

void **fexpr_write**(*calcium_stream_t stream*, **const** *fexpr_t expr*)
Writes *expr* to *stream*.

void **fexpr_print**(**const** *fexpr_t expr*)
Prints *expr* to standard output.

char ***fexpr_get_str**(**const** *fexpr_t expr*)
Returns a string representation of *expr*. The string must be freed with `flint_free()`.

Warning: string literals appearing in expressions are currently not escaped.

7.1.8 LaTeX output

void **fexpr_write_latex**(*calcium_stream_t stream*, **const** *fexpr_t expr*, *ulong flags*)
Writes the LaTeX representation of *expr* to *stream*.

void **fexpr_print_latex**(**const** *fexpr_t expr*, *ulong flags*)
Prints the LaTeX representation of *expr* to standard output.

char ***fexpr_get_str_latex**(**const** *fexpr_t expr*, *ulong flags*)
Returns a string of the LaTeX representation of *expr*. The string must be freed with `flint_free()`.

Warning: string literals appearing in expressions are currently not escaped.

The *flags* parameter allows specifying options for LaTeX output. The following flags are supported:

`FEXPR_LATEX_SMALL`
Generate more compact formulas, most importantly by printing fractions inline as p/q instead of as $\frac{p}{q}$. This flag is automatically activated within subscripts and superscripts and in certain other parts of formulas.

`FEXPR_LATEX_LOGIC`
Use symbols for logical operators such as Not, And, Or, which by default are rendered as words for legibility.

7.1.9 Function call structure

slong **fexpr_nargs**(**const** *fexpr_t expr*)
Returns the number of arguments n in the function call $f(e_1, \ldots, e_n)$ represented by *expr*. If *expr* is an atom, returns -1.

void **fexpr_func**(*fexpr_t res*, **const** *fexpr_t expr*)
Assuming that *expr* represents a function call $f(e_1, \ldots, e_n)$, sets *res* to the function expression f.

void **fexpr_view_func**(*fexpr_t view*, **const** *fexpr_t expr*)
As *fexpr_func()*, but sets *view* to a shallow view instead of copying the expression. The variable *view* must not be initialized before use or cleared after use, and *expr* must not be modified or cleared as long as *view* is in use.

void **fexpr_arg**(*fexpr_t res*, **const** *fexpr_t expr*, *slong i*)
Assuming that *expr* represents a function call $f(e_1, \ldots, e_n)$, sets *res* to the argument e_{i+1}. Note that indexing starts from 0. The index must be in bounds, with $0 \le i < n$.

void **fexpr_view_arg**(*fexpr_t view*, **const** *fexpr_t expr*, *slong i*)
As *fexpr_arg()*, but sets *view* to a shallow view instead of copying the expression. The variable *view* must not be initialized before use or cleared after use, and *expr* must not be modified or cleared as long as *view* is in use.

void **fexpr_view_next**(*fexpr_t view*)
Assuming that *view* is a shallow view of a function argument e_i in a function call $f(e_1, \ldots, e_n)$, sets *view* to a view of the next argument e_{i+1}. This function can be called when *view* refers to the

last argument e_n, provided that *view* is not used afterwards. This function can also be called when *view* refers to the function *f*, in which case it will make *view* point to e_1.

int **fexpr_is_builtin_call**(**const** *fexpr_t expr*, *slong id*)
Returns whether *expr* has the form $f(\ldots)$ where f is a builtin function defined by *id* (see *fexpr_builtin.h – builtin symbols*).

int **fexpr_is_any_builtin_call**(**const** *fexpr_t expr*)
Returns whether *expr* has the form $f(\ldots)$ where f is any builtin function (see *fexpr_builtin.h – builtin symbols*).

7.1.10 Composition

void **fexpr_call0**(*fexpr_t res*, **const** *fexpr_t f*)

void **fexpr_call1**(*fexpr_t res*, **const** *fexpr_t f*, **const** *fexpr_t x1*)

void **fexpr_call2**(*fexpr_t res*, **const** *fexpr_t f*, **const** *fexpr_t x1*, **const** *fexpr_t x2*)

void **fexpr_call3**(*fexpr_t res*, **const** *fexpr_t f*, **const** *fexpr_t x1*, **const** *fexpr_t x2*, **const** *fexpr_t x3*)

void **fexpr_call4**(*fexpr_t res*, **const** *fexpr_t f*, **const** *fexpr_t x1*, **const** *fexpr_t x2*, **const** *fexpr_t x3*, **const** *fexpr_t x4*)

void **fexpr_call_vec**(*fexpr_t res*, **const** *fexpr_t f*, *fexpr_srcptr args*, *slong len*)
Creates the function call $f(x_1, \ldots, x_n)$. The *vec* version takes the arguments as an array *args* and n is given by *len*. Warning: aliasing between inputs and outputs is not implemented.

void **fexpr_call_builtin1**(*fexpr_t res*, *slong f*, **const** *fexpr_t x1*)

void **fexpr_call_builtin2**(*fexpr_t res*, *slong f*, **const** *fexpr_t x1*, **const** *fexpr_t x2*)
Creates the function call $f(x_1, \ldots, x_n)$, where f defines a builtin symbol.

7.1.11 Subexpressions and replacement

int **fexpr_contains**(**const** *fexpr_t expr*, **const** *fexpr_t x*)
Returns whether *expr* contains the expression x as a subexpression (this includes the case where *expr* and x are equal).

int **fexpr_replace**(*fexpr_t res*, **const** *fexpr_t expr*, **const** *fexpr_t x*, **const** *fexpr_t y*)
Sets *res* to the expression *expr* with all occurrences of the subexpression x replaced by the expression y. Returns a boolean value indicating whether any replacements have been performed. Aliasing is allowed between *res* and *expr* but not between *res* and x or y.

int **fexpr_replace2**(*fexpr_t res*, **const** *fexpr_t expr*, **const** *fexpr_t x1*, **const** *fexpr_t y1*, **const** *fexpr_t x2*, **const** *fexpr_t y2*)
Like *fexpr_replace()*, but simultaneously replaces *x1* by *y1* and *x2* by *y2*.

int **fexpr_replace_vec**(*fexpr_t res*, **const** *fexpr_t expr*, **const** *fexpr_vec_t xs*, **const** *fexpr_vec_t ys*)
Sets *res* to the expression *expr* with all occurrences of the subexpressions given by entries in *xs* replaced by the corresponding expressions in *ys*. It is required that *xs* and *ys* have the same length. Returns a boolean value indicating whether any replacements have been performed. Aliasing is allowed between *res* and *expr* but not between *res* and the entries of *xs* or *ys*.

7.1.12 Arithmetic expressions

void **fexpr_set_fmpq**(*fexpr_t res*, **const** *fmpq_t x*)
Sets *res* to the rational number x. This creates an atomic integer if the denominator of x is one, and otherwise creates a division expression.

void **fexpr_set_arf**(*fexpr_t res*, **const** arf_t *x*)

void **fexpr_set_d**(*fexpr_t res*, double *x*)
Sets *res* to an expression for the value of the floating-point number x. NaN is represented as `Undefined`. For a regular value, this creates an atomic integer or a rational fraction if the exponent is small, and otherwise creates an expression of the form `Mul(m, Pow(2, e))`.

void **fexpr_set_re_im_d**(*fexpr_t res*, double *x*, double *y*)
Sets *res* to an expression for the complex number with real part x and imaginary part y.

void **fexpr_neg**(*fexpr_t res*, **const** *fexpr_t a*)

void **fexpr_add**(*fexpr_t res*, **const** *fexpr_t a*, **const** *fexpr_t b*)

void **fexpr_sub**(*fexpr_t res*, **const** *fexpr_t a*, **const** *fexpr_t b*)

void **fexpr_mul**(*fexpr_t res*, **const** *fexpr_t a*, **const** *fexpr_t b*)

void **fexpr_div**(*fexpr_t res*, **const** *fexpr_t a*, **const** *fexpr_t b*)

void **fexpr_pow**(*fexpr_t res*, **const** *fexpr_t a*, **const** *fexpr_t b*)
Constructs an arithmetic expression with given arguments. No simplifications whatsoever are performed.

int **fexpr_is_arithmetic_operation**(**const** *fexpr_t expr*)
Returns whether *expr* is of the form $f(e_1, \ldots, e_n)$ where f is one of the arithmetic operators `Pos`, `Neg`, `Add`, `Sub`, `Mul`, `Div`.

void **fexpr_arithmetic_nodes**(*fexpr_vec_t nodes*, **const** *fexpr_t expr*)
Sets *nodes* to a vector of subexpressions of *expr* such that *expr* is an arithmetic expression with *nodes* as leaves. More precisely, *expr* will be constructed out of nested application the arithmetic operators `Pos`, `Neg`, `Add`, `Sub`, `Mul`, `Div` with integers and expressions in *nodes* as leaves. Powers `Pow` with an atomic integer exponent are also allowed. The nodes are output without repetition but are not automatically sorted in a canonical order.

int **fexpr_get_fmpz_mpoly_q**(*fmpz_mpoly_q_t res*, **const** *fexpr_t expr*, **const** *fexpr_vec_t vars*, **const** *fmpz_mpoly_ctx_t ctx*)
Sets *res* to the expression *expr* as a formal rational function of the subexpressions in *vars*. The vector *vars* must have the same length as the number of variables specified in *ctx*. To build *vars* automatically for a given expression, *fexpr_arithmetic_nodes()* may be used.

Returns 1 on success and 0 on failure. Failure can occur for the following reasons:

- A subexpression is encountered that cannot be interpreted as an arithmetic operation and does not appear (exactly) in *vars*.
- Overflow (too many terms or too large exponent).
- Division by zero (a zero denominator is encountered).

It is important to note that this function views *expr* as a formal rational function with *vars* as formal indeterminates. It does thus not check for algebraic relations between *vars* and can implicitly divide by zero if *vars* are not algebraically independent.

void **fexpr_set_fmpz_mpoly**(*fexpr_t res*, **const** *fmpz_mpoly_t poly*, **const** *fexpr_vec_t vars*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fexpr_set_fmpz_mpoly_q**(*fexpr_t res*, **const** *fmpz_mpoly_q_t frac*, **const** *fexpr_vec_t vars*, **const** *fmpz_mpoly_ctx_t ctx*)
Sets *res* to an expression for the multivariate polynomial *poly* (or rational function *frac*), using the expressions in *vars* as the variables. The length of *vars* must agree with the number of variables in *ctx*. If *NULL* is passed for *vars*, a default choice of symbols is used.

int **fexpr_expanded_normal_form**(*fexpr_t res*, **const** *fexpr_t expr*, *ulong flags*)
Sets *res* to *expr* converted to expanded normal form viewed as a formal rational function with its non-arithmetic subexpressions as terminal nodes. This function first computes nodes with *fexpr_arithmetic_nodes()*, sorts the nodes, evaluates to a rational function with *fexpr_get_fmpz_mpoly_q()*, and then converts back to an expression with *fexpr_set_fmpz_mpoly_q()*. Optional *flags* are reserved for future use.

7.1.13 Vectors

void **fexpr_vec_init**(*fexpr_vec_t vec*, *slong len*)
Initializes *vec* to a vector of length *len*. All entries are set to the atomic integer 0.

void **fexpr_vec_clear**(*fexpr_vec_t vec*)
Clears the vector *vec*.

void **fexpr_vec_print**(**const** *fexpr_vec_t vec*)
Prints *vec* to standard output.

void **fexpr_vec_swap**(*fexpr_vec_t x*, *fexpr_vec_t y*)
Swaps x and y efficiently.

void **fexpr_vec_fit_length**(*fexpr_vec_t vec*, *slong len*)
Ensures that *vec* has space for *len* entries.

void **fexpr_vec_set**(*fexpr_vec_t dest*, **const** *fexpr_vec_t src*)
Sets *dest* to a copy of *src*.

void **fexpr_vec_append**(*fexpr_vec_t vec*, **const** *fexpr_t expr*)
Appends *expr* to the end of the vector *vec*.

slong **fexpr_vec_insert_unique**(*fexpr_vec_t vec*, **const** *fexpr_t expr*)
Inserts *expr* without duplication into vec, returning its position. If this expression already exists, *vec* is unchanged. If this expression does not exist in *vec*, it is appended.

void **fexpr_vec_set_length**(*fexpr_vec_t vec*, *slong len*)
Sets the length of *vec* to *len*, truncating or zero-extending as needed.

void **_fexpr_vec_sort_fast**(*fexpr_ptr vec*, *slong len*)
Sorts the *len* entries in *vec* using the comparison function *fexpr_cmp_fast()*.

7.2 fexpr_builtin.h – builtin symbols

This module defines symbol names with a predefined meaning for use in symbolic expressions. These symbols will eventually all support LaTeX rendering as well as symbolic and numerical evaluation (where applicable).

By convention, all builtin symbol names are at least two characters long and start with an uppercase letter. Single-letter symbol names and symbol names beginning with a lowercase letter are reserved for variables.

For any builtin symbol name `Symbol`, the header file `fexpr_builtin.h` defines a C constant `FEXPR_Symbol` as an index to a builtin symbol table. The symbol will be documented as `Symbol` below.

7.2.1 C helper functions

slong **fexpr_builtin_lookup**(**const** char *s)
Returns the internal index used to encode the builtin symbol with name s in expressions. If s is not the name of a builtin symbol, returns -1.

const char ***fexpr_builtin_name**(*slong* n)
Returns a read-only pointer for a string giving the name of the builtin symbol with index n.

slong **fexpr_builtin_length**(void)
Returns the number of builtin symbols.

7.2.2 Variables and iteration

Expressions involving the following symbols have a special role in binding variables.

For
Generator expression. This is a syntactical construct which does not represent a mathematical object on its own. In general, `For(x, ...)` defines the symbol `x` as a locally bound variable in the scope of the parent expression. The following arguments `...` specify an evaluation range, set or point. Their interpretation depends on the parent operator. The following cases are possible.

Case 1: `For(x, S)` specifies iteration or comprehension for `x` ranging over the values of the set `S`. This interpretation is used in operators that aggregate values over a set. The `For` expression may be followed by a filter predicate `P(x)` restricting the range to a subset of `S`. Examples:

`Set(f(x), For(x, S))` denotes $\{f(x) : x \in S\}$.

`Set(f(x), For(x, S), P(x))` denotes $\{f(x) : x \in S \text{ and } P(x)\}$.

`Sum(f(x), For(x, S))` denotes $\sum_{x \in S} f(x)$.

`Sum(f(x), For(x, S), P(x))` denotes $\sum_{x \in S,\, P(x)} f(x)$.

Case 2: `For(x, a, b)` specifies that `x` ranges between the endpoints `a` and `b` in the context of `Sum`, `Product`, `Integral`, and similar operators. Examples:

`Sum(f(n), For(n, a, b))` denotes $\sum_{n=a}^{b} f(n)$. The iteration is empty if $b < a$.

`Integral(f(x), For(x, a, b))` denotes $\int_a^b f(x)dx$, where the integral follows a straight-line path from a to b. Swapping a and b negates the value.

Case 3: `For(x, a)` specifies that `x` approaches the point `a` in the context of `Limit`-type operator, or differentiation with respect to `x` at the point `a` in the context of a `Derivative`-type operator. Examples:

`Derivative(f(x), For(x, a))` denotes $f'(a)$.

`Limit(f(x), For(x, a))` denotes $\lim_{x \to a} f(x)$.

Case 4: `For(x, a, n)` specifies differentiation with respect to `x` at the point `a` to order `n` in the context of a `Derivative`-type operator. Examples:

`Derivative(f(x), For(x, a, n))` denotes $f^{(n)}(a)$.

`Where`

`Where(f(x), Def(x, a))` defines the symbol `x` as an alias for the expression `a` and evaluates the expression `f(x)` with this bound value of `x`. This is equivalent to `f(a)`. This may be rendered as $f(x)$ where $x = a$.

`Where(f(x), Def(f(t), a))` defines the symbol `f` as a function mapping the dummy variable `t` to `a`.

`Where(Add(a, b), Def(Tuple(a, b), T))` is a destructuring assignment.

`Def`

Definition expression. This is a syntactical construct which does not represent a mathematical object on its own. The `Def` expression is used only within a `Where`-expression; see that documentation of that symbol for more examples.

`Def(x, a)` defines the symbol `x` as an alias for the expression `a`.

`Def(f(x, y, z), a)` defines the symbol `f` as a function of three variables. The dummy variables `x`, `y` and `z` may appear within the expression `a`.

`Fun`

`Fun(x, expr)` defines an anonymous univariate function mapping the symbol `x` to the expression `expr`. The symbol `x` becomes locally bound within this `Fun` expression.

`Step`

`Repeat`

7.2.3 Booleans and logic

`Equal`

`Equal(a, b)`, signifying $a = b$, is `True` if `a` and `b` represent the same object, and `False` otherwise. This operator can be called with any number of arguments, in which case it evaluates whether all arguments are equal.

`NotEqual`

`NotEqual(a, b)`, signifying $a \neq b$, is equivalent to `Not(Equal(a, b))`.

`Same`

`Same(a, b)` gives `a` (or equivalently `b`) if `a` and `b` represent the same object, and `Undefined` otherwise. This can be used to assert or emphasize that two expressions represent the same value within a formula. This operator can be called with any number of arguments, in which case it asserts that all arguments are equal.

`True`

`True` is a logical constant.

`False`

`False` is a logical constant.

`Not`

`Not(x)` is the logical negation of `x`.

`And`

`And(x, y)` is the logical AND of `x` and `y`. This function can be called with any number of arguments.

`Or`

`Or(x, y)` is the logical OR of `x` and `y`. This function can be called with any number of arguments.

`Equivalent`
: `Equivalent(x, y)` denotes the logical equivalence $x \Leftrightarrow y$. Semantically, this is the same as `Equal` called with logical arguments.

`Implies`
: `Implies(x, y)` denotes the logical implication $x \implies y$.

`Exists`
: Existence quantifier.

  `Exists(f(x), For(x, S))` denotes $f(x)$ for some $x \in S$.

  `Exists(f(x), For(x, S), P(x))` denotes $f(x)$ for some $x \in S$ with $P(x)$.

`All`
: Universal quantifier.

  `All(f(x), For(x, S))` denotes $f(x)$ for all $x \in S$.

  `All(f(x), For(x, S), P(x))` denotes $f(x)$ for all $x \in S$ with $P(x)$.

`Cases`
: `Cases(Case(f(x), P(x)), Case(g(x), Otherwise))` denotes:

$$\begin{cases} f(x), & P(x) \\ g(x), & \text{otherwise} \end{cases}$$

  `Cases(Case(f(x), P(x)), Case(g(x), Q(x)), Case(h(x), Otherwise))` denotes:

$$\begin{cases} f(x), & P(x) \\ g(x), & Q(x) \\ h(x), & \text{otherwise} \end{cases}$$

  If both $P(x)$ and $Q(x)$ are true simultaneously, no ordering is implied; it is assumed that $f(x)$ and $g(x)$ give the same value for any such x. More generally, this operator can be called with any number of case distinctions.

  If the *Otherwise* case is omitted, the result is undefined if neither predicate is true.

`Case`
: See `Cases`.

`Otherwise`
: See `Cases`.

7.2.4 Tuples, lists and sets

`Tuple`

`List`

`Set`

`Item`

`Element`

`NotElement`

`EqualAndElement`

`Length`

`Cardinality`

`Concatenation`

`Union`

`Intersection`

`SetMinus`

`Subset`

`SubsetEqual`

`CartesianProduct`

`CartesianPower`

`Subsets`
`Subsets(S)` is the power set $\mathscr{P}(S)$ comprising all subsets of the set `S`.

`Sets`
`Sets` is the class Sets of all sets.

`Tuples`
`Tuples` is the class of all tuples.

`Tuples(S)` is the set of all tuples with elements in the set `S`.

`Tuples(S, n)` is the set of all length-`n` tuples with elements in the set `S`.

7.2.5 Numbers and arithmetic

Undefined

`Undefined`
`Undefined` is the special value `u` (undefined).

Particular numbers

`Pi`
`Pi` is the constant π.

`NumberI`
`NumberI` is the imaginary unit i. The verbose name leaves `i` and `I` to be used as a variable names.

`NumberE`
`NumberE` is the base of the natural logarithm e. The verbose name leaves `e` and `E` to be used as a variable names.

`GoldenRatio`
`GoldenRatio` is the golden ratio φ.

`Euler`
`Euler` is Euler's constant γ.

`CatalanConstant`
`CatalanConstant` is Catalan's constant G.

`KhinchinConstant`
`KhinchinConstant` is Khinchin's constant K.

`GlaisherConstant`
`GlaisherConstant` is Glaisher's constant A.

`RootOfUnity`
`RootOfUnity(n)` is the principal complex n-th root of unity $\zeta_n = e^{2\pi i/n}$.

`RootOfUnity(n, k)` is the complex n-th root of unity ζ_n^k.

Number constructors

Remark: the rational number with numerator p and denominator q can be constructed as `Div(p, q)`.

`Decimal`
: `Decimal(str)` gives the rational number specified by the string *str* in ordinary decimal floating-point notation (for example `-3.25e-725`).

`AlgebraicNumberSerialized`

`PolynomialRootIndexed`

`PolynomialRootNearest`

`Enclosure`

`Approximation`

`Guess`

`Unknown`

Arithmetic operations

`Pos`

`Neg`

`Add`

`Sub`

`Mul`

`Div`

`Pow`

`Sqrt`

`Root`

Inequalities

`Less`

`LessEqual`

`Greater`

`GreaterEqual`

`EqualNearestDecimal`

Sets of numbers

`NN`
: `NN` is the set of natural numbers (including 0), $\mathbb{N}$.

`ZZ`
: `ZZ` is the set of integers, $\mathbb{Z}$.

`QQ`
: `QQ` is the set of rational numbers, $\mathbb{Q}$.

`RR`
: `RR` is the set of real numbers, $\mathbb{R}$.

`CC`
: `CC` is the set of complex numbers, $\mathbb{C}$.

`Primes`
: `Primes` is the set of positive prime numbers, $\mathbb{P}$

`IntegersGreaterEqual`
: `IntegersGreaterEqual(x)`, given an extended real number x, gives the set $\mathbb{Z}_{\geq x}$ of integers greater than or equal to x.

`IntegersLessEqual`
: `IntegersLessEqual(x)`, given an extended real number x, gives the set $\mathbb{Z}_{\leq x}$ of integers less than or equal to x.

`Range`
: `Range(a, b)`, given integers a and b, gives the set $\{a, a+1, \ldots, b\}$ of integers between a and b. This is the empty set if a is greater than b.

`AlgebraicNumbers`
: The set of complex algebraic numbers $\overline{\mathbb{Q}}$.

`RealAlgebraicNumbers`
: The set of real algebraic numbers $\overline{\mathbb{Q}}_{\mathbb{R}}$.

`Interval`
: `Interval(a, b)`, given extended real numbers a and b, gives the closed interval $[a, b]$.

`OpenInterval`
: `OpenInterval(a, b)`, given extended real numbers a and b, gives the open interval (a, b).

`ClosedOpenInterval`
: `ClosedOpenInterval(a, b)`, given extended real numbers a and b, gives the closed-open interval $[a, b)$.

`OpenClosedInterval`
: `OpenClosedInterval(a, b)`, given extended real numbers a and b, gives the closed-open interval $(a, b]$.

`RealBall`
: `RealBall(m, r)`, given a real number m and an extended real number r, gives the the closed real ball $[m \pm r]$ with center m and radius r.

`OpenRealBall`
: `OpenRealBall(m, r)`, given a real number m and an extended real number r, gives the the open real ball $(m \pm r)$ with center m and radius r.

`OpenComplexDisk`
: `OpenComplexDisk(m, r)`, given a complex number m and an extended real number r, gives the open complex disk $D(m, r)$ with center m and radius r.

`ClosedComplexDisk`
: `ClosedComplexDisk(m, r)`, given a complex number m and a real number r, gives the closed complex disk $\overline{D}(m, r)$ with center m and radius r.

`UpperHalfPlane`
: `UpperHalfPlane` is the set $\mathbb{H}$ of complex numbers with positive imaginary part.

`UnitCircle`

`BernsteinEllipse`

`Lattice`

Infinities and extended numbers

`Infinity`
: `Infinity` is the positive signed infinity ∞.

`UnsignedInfinity`
: `UnsignedInfinity` is the unsigned infinity $\tilde{\infty}$.

`RealSignedInfinities`
: `RealSignedInfinities` is the set of real signed infinities $\{+\infty, -\infty\}$.

`ComplexSignedInfinities`
: `ComplexSignedInfinities` is the set of complex signed infinities $\{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\}$.

`RealInfinities`
: `RealInfinities` is the set of real infinities (signed and unsigned) $\{+\infty, -\infty\} \cup \{\tilde{\infty}\}$.

`ComplexInfinities`
: `ComplexInfinities` is the set of complex infinities (signed and unsigned) $\{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\} \cup \{\tilde{\infty}\}$.

`ExtendedRealNumbers`
: `ExtendedRealNumbers` is the set of extended real numbers $\mathbb{R} \cup \{+\infty, -\infty\}$.

`ProjectiveRealNumbers`
: `ProjectiveRealNumbers` is the set of projectively extended real numbers $\mathbb{R} \cup \{\tilde{\infty}\}$.

`SignExtendedComplexNumbers`
: `SignExtendedComplexNumbers` is the set of complex numbers extended with signed infinities $\mathbb{C} \cup \{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\}$.

`ProjectiveComplexNumbers`
: `ProjectiveComplexNumbers` is the set of projectively extended complex numbers (also known as the Riemann sphere) $\mathbb{C} \cup \{\tilde{\infty}\}$.

`RealSingularityClosure`
: `RealSingularityClosure` is the Calcium singularity closure for real functions, encompassing real numbers, signed infinities, unsigned infinity, and *undefined* (u). This set is defined as $\mathbb{R}_{\text{Sing}} = \mathbb{R} \cup \{+\infty, -\infty\} \cup \{\tilde{\infty}\} \cup \{\mathrm{u}\}$.

`ComplexSingularityClosure`
: `ComplexSingularityClosure` is the Calcium singularity closure for complex functions, encompassing complex numbers, signed infinities, unsigned infinity, and *undefined* (u). This set is defined as $\mathbb{C}_{\text{Sing}} = \mathbb{C} \cup \{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\} \cup \{\tilde{\infty}\} \cup \{\mathrm{u}\}$.

7.2.6 Operators and calculus

Sums and products

`Sum`

`Product`

`PrimeSum`

`PrimeProduct`

`DivisorSum`

`DivisorProduct`

Solutions and zeros

`Zeros`

`UniqueZero`

`Solutions`

`UniqueSolution`

Extreme values

`Supremum`

`Infimum`

`Minimum`

`Maximum`

`ArgMin`

`ArgMax`

`ArgMinUnique`

`ArgMaxUnique`

Limits

`Limit`

`SequenceLimit`

`RealLimit`

`LeftLimit`

`RightLimit`

`ComplexLimit`

`MeromorphicLimit`

`SequenceLimitInferior`

`SequenceLimitSuperior`

`AsymptoticTo`

Derivatives

`Derivative`

`RealDerivative`

`ComplexDerivative`

`ComplexBranchDerivative`

`MeromorphicDerivative`

Integrals

`Integral`

Complex analysis

`Path`

`CurvePath`

`Poles`

`IsHolomorphicOn`

`IsMeromorphicOn`

`Residue`

`ComplexZeroMultiplicity`

`AnalyticContinuation`

7.2.7 Matrices and linear algebra

`Matrix`

`Row`

`Column`

`RowMatrix`

`ColumnMatrix`

`DiagonalMatrix`

`Matrix2x2`

`ZeroMatrix`

`IdentityMatrix`

`Det`

`Spectrum`

`SingularValues`

`Matrices`

`SL2Z`

`PSL2Z`

`SpecialLinearGroup`

`GeneralLinearGroup`

`HilbertMatrix`

7.2.8 Polynomials, series and rings

`Pol`

`Ser`

`Polynomial`

`Coefficient`

`PolynomialDegree`

`Polynomials`

`PolynomialFractions`

`FormalPowerSeries`

`FormalLaurentSeries`

`FormalPuiseuxSeries`

`Zero`

`One`

`Characteristic`

`Rings`

`CommutativeRings`

`Fields`

`QuotientRing`

`FiniteField`

`EqualQSeriesEllipsis`

`IndefiniteIntegralEqual`

`QSeriesCoefficient`

`Call`

`CallIndeterminate`

7.2.9 Special functions

Number parts and step functions

`Abs`

`Sign`

`Re`

`Im`

`Arg`

`Conjugate`

`Csgn`

`RealAbs`

`Max`

`Min`

`Floor`

Ceil

KroneckerDelta

Primes and divisibility

IsOdd

IsEven

CongruentMod

Divides

Mod

GCD

LCM

XGCD

IsPrime

Prime

PrimePi

DivisorSigma

MoebiusMu

EulerPhi

DiscreteLog

LegendreSymbol

JacobiSymbol

KroneckerSymbol

SquaresR

LiouvilleLambda

Elementary functions

Exp

Log

Sin

Cos

Tan

Cot

Sec

Csc

Sinh

Cosh

Tanh

Coth

Sech

Csch

Asin

Acos

Atan

Acot

Asec

Acsc

Asinh

Acosh

Atanh

Acoth

Asech

Acsch

Atan2

Sinc

LambertW

Combinatorial functions

SloaneA

SymmetricPolynomial

Cyclotomic

Fibonacci

BernoulliB

BernoulliPolynomial

StirlingCycle

StirlingS1

StirlingS2

EulerE

EulerPolynomial

BellNumber

PartitionsP

LandauG

Gamma function and factorials

`Factorial`

`Binomial`

`Gamma`

`LogGamma`

`DoubleFactorial`

`RisingFactorial`

`FallingFactorial`

`HarmonicNumber`

`DigammaFunction`

`DigammaFunctionZero`

`BetaFunction`

`BarnesG`

`LogBarnesG`

`StirlingSeriesRemainder`

`LogBarnesGRemainder`

Orthogonal polynomials

`ChebyshevT`

`ChebyshevU`

`LegendreP`

`JacobiP`

`HermiteH`

`LaguerreL`

`GegenbauerC`

`SphericalHarmonicY`

`LegendrePolynomialZero`

`GaussLegendreWeight`

Exponential integrals

`Erf`

`Erfc`

`Erfi`

`UpperGamma`

`LowerGamma`

`IncompleteBeta`

`IncompleteBetaRegularized`

`LogIntegral`

ExpIntegralE

ExpIntegralEi

SinIntegral

SinhIntegral

CosIntegral

CoshIntegral

FresnelC

FresnelS

Bessel and Airy functions

AiryAi

AiryBi

AiryAiZero

AiryBiZero

BesselJ

BesselI

BesselY

BesselK

HankelH1

HankelH2

BesselJZero

BesselYZero

CoulombF

CoulombG

CoulombH

CoulombC

CoulombSigma

Hypergeometric functions

Hypergeometric0F1

Hypergeometric1F1

Hypergeometric1F2

Hypergeometric2F1

Hypergeometric2F2

Hypergeometric2F0

Hypergeometric3F2

HypergeometricU

HypergeometricUStar

HypergeometricUStarRemainder

Hypergeometric0F1Regularized

Hypergeometric1F1Regularized

Hypergeometric1F2Regularized

Hypergeometric2F1Regularized

Hypergeometric2F2Regularized

Hypergeometric3F2Regularized

Zeta and L-functions

RiemannZeta

RiemannZetaZero

RiemannHypothesis

RiemannXi

HurwitzZeta

LerchPhi

PolyLog

MultiZetaValue

DirichletL

DirichletLZero

DirichletLambda

DirichletCharacter

DirichletGroup

PrimitiveDirichletCharacters

GeneralizedRiemannHypothesis

ConreyGenerator

GeneralizedBernoulliB

StieltjesGamma

KeiperLiLambda

GaussSum

Elliptic integrals

AGM

AGMSequence

EllipticK

EllipticE

EllipticPi

IncompleteEllipticF

IncompleteEllipticE

IncompleteEllipticPi

CarlsonRF

`CarlsonRG`

`CarlsonRJ`

`CarlsonRD`

`CarlsonRC`

`CarlsonHypergeometricR`

`CarlsonHypergeometricT`

Elliptic, theta and modular functions

`JacobiTheta`

`JacobiThetaQ`

`DedekindEta`

`ModularJ`

`ModularLambda`

`EisensteinG`

`EisensteinE`

`DedekindSum`

`WeierstrassP`

`WeierstrassZeta`

`WeierstrassSigma`

`EllipticRootE`

`HilbertClassPolynomial`

`EulerQSeries`

`DedekindEtaEpsilon`

`ModularGroupAction`

`ModularGroupFundamentalDomain`

`ModularLambdaFundamentalDomain`

`PrimitiveReducedPositiveIntegralBinaryQuadraticForms`

`JacobiThetaEpsilon`

`JacobiThetaPermutation`

Nonsemantic markup

`Ellipsis`
: `Ellipsis` renders as ... in LaTeX. It can be used to indicate missing function arguments for display purposes, but it has no predefined builtin semantics.

`Parentheses`
: `Parentheses(x)` semantically represents `x`, but renders with parentheses ((x)) when converted to LaTeX.

`Brackets`
: `Brackets(x)` semantically represents `x`, but renders with brackets ($[x]$) when converted to LaTeX.

`Braces`
: `Braces(x)` semantically represents `x`, but renders with braces ($\{x\}$) when converted to LaTeX.

`AngleBrackets`
: `AngleBrackets(x)` semantically represents `x`, but renders with angle brackets ($\langle x \rangle$) when converted to LaTeX.

`Logic`
: `Logic(x)` semantically represents `x`, but forces logical expressions within x to be rendered using symbols instead of text.

`ShowExpandedNormalForm`
: `ShowExpandedNormalForm(x)` semantically represents `x`, but displays the expanded normal form of the expression instead of rendering the expression verbatim. Warning: this triggers a nontrivial (potentially very expensive) computation.

`Subscript`

CHAPTER

EIGHT

BASIC ALGEBRAIC STRUCTURES

The following modules implement useful exact structures independently of the *ca_t* type. They are used internally in Calcium, but the interfaces are stable and use in appropriate external applications is encouraged.

8.1 fmpz_mpoly_q.h – multivariate rational functions over Q

An *fmpz_mpoly_q_t* represents an element of $\mathbb{Q}(x_1, \ldots, x_n)$ for fixed n as a pair of Flint multivariate polynomials (*fmpz_mpoly_t*). Instances are always kept in canonical form by ensuring that the GCD of numerator and denominator is 1 and that the coefficient of the leading term of the denominator is positive.

The user must create a multivariate polynomial context (*fmpz_mpoly_ctx_t*) specifying the number of variables n and the monomial ordering.

8.1.1 Types and macros

type fmpz_mpoly_q_struct

type fmpz_mpoly_q_t
An *fmpz_mpoly_q_struct* consists of a pair of *fmpz_mpoly_struct*:s. An *fmpz_mpoly_q_t* is defined as an array of length one of type *fmpz_mpoly_q_struct*, permitting an *fmpz_mpoly_q_t* to be passed by reference.

fmpz_mpoly_q_numref(*x*)
Macro returning a pointer to the numerator of x which can be used as an *fmpz_mpoly_t*.

fmpz_mpoly_q_denref(*x*)
Macro returning a pointer to the denominator of x which can be used as an *fmpz_mpoly_t*.

8.1.2 Memory management

void **fmpz_mpoly_q_init**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_ctx_t ctx*)
Initializes *res* for use, and sets its value to zero.

void **fmpz_mpoly_q_clear**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_ctx_t ctx*)
Clears *res*, freeing or recycling its allocated memory.

8.1.3 Assignment

void **fmpz_mpoly_q_swap**(*fmpz_mpoly_q_t* *x*, *fmpz_mpoly_q_t* *y*, **const** *fmpz_mpoly_ctx_t* *ctx*)
Swaps the values of x and y efficiently.

void **fmpz_mpoly_q_set**(*fmpz_mpoly_q_t* *res*, **const** *fmpz_mpoly_q_t* *x*, **const** *fmpz_mpoly_ctx_t* *ctx*)
void **fmpz_mpoly_q_set_fmpq**(*fmpz_mpoly_q_t* *res*, **const** *fmpq_t* *x*, **const** *fmpz_mpoly_ctx_t* *ctx*)
void **fmpz_mpoly_q_set_fmpz**(*fmpz_mpoly_q_t* *res*, **const** *fmpz_t* *x*, **const** *fmpz_mpoly_ctx_t* *ctx*)
void **fmpz_mpoly_q_set_si**(*fmpz_mpoly_q_t* *res*, *slong* *x*, **const** *fmpz_mpoly_ctx_t* *ctx*)
Sets *res* to the value x.

8.1.4 Canonicalisation

void **fmpz_mpoly_q_canonicalise**(*fmpz_mpoly_q_t* *x*, **const** *fmpz_mpoly_ctx_t* *ctx*)
Puts the numerator and denominator of x in canonical form by removing common content and making the leading term of the denominator positive.

int **fmpz_mpoly_q_is_canonical**(**const** *fmpz_mpoly_q_t* *x*, **const** *fmpz_mpoly_ctx_t* *ctx*)
Returns whether x is in canonical form.

In addition to verifying that the numerator and denominator have no common content and that the leading term of the denominator is positive, this function checks that the denominator is nonzero and that the numerator and denominator have correctly sorted terms (these properties should normally hold; verifying them provides an extra consistency check for test code).

8.1.5 Properties

int **fmpz_mpoly_q_is_zero**(**const** *fmpz_mpoly_q_t* *x*, **const** *fmpz_mpoly_ctx_t* *ctx*)
Returns whether x is the constant 0.

int **fmpz_mpoly_q_is_one**(**const** *fmpz_mpoly_q_t* *x*, **const** *fmpz_mpoly_ctx_t* *ctx*)
Returns whether x is the constant 1.

void **fmpz_mpoly_q_used_vars**(int **used*, **const** *fmpz_mpoly_q_t* *f*, **const** *fmpz_mpoly_ctx_t* *ctx*)
void **fmpz_mpoly_q_used_vars_num**(int **used*, **const** *fmpz_mpoly_q_t* *f*, **const** *fmpz_mpoly_ctx_t* *ctx*)
void **fmpz_mpoly_q_used_vars_den**(int **used*, **const** *fmpz_mpoly_q_t* *f*, **const** *fmpz_mpoly_ctx_t* *ctx*)
For each variable, sets the corresponding entry in *used* to the boolean flag indicating whether that variable appears in the rational function (respectively its numerator or denominator).

8.1.6 Special values

void **fmpz_mpoly_q_zero**(*fmpz_mpoly_q_t* *res*, **const** *fmpz_mpoly_ctx_t* *ctx*)
Sets *res* to the constant 0.

void **fmpz_mpoly_q_one**(*fmpz_mpoly_q_t* *res*, **const** *fmpz_mpoly_ctx_t* *ctx*)
Sets *res* to the constant 1.

void **fmpz_mpoly_q_gen**(*fmpz_mpoly_q_t* *res*, *slong* *i*, **const** *fmpz_mpoly_ctx_t* *ctx*)
Sets *res* to the generator x_{i+1}. Requires $0 \le i < n$ where n is the number of variables of *ctx*.

8.1.7 Input and output

void **fmpz_mpoly_q_print_pretty**(**const** *fmpz_mpoly_q_t f*, **const** char **x, *fmpz_mpoly_ctx_t ctx*)

Prints *res* to standard output. If x is not *NULL*, the strings in x are used as the symbols for the variables.

8.1.8 Random generation

void **fmpz_mpoly_q_randtest**(*fmpz_mpoly_q_t res*, flint_rand_t *state*, *slong length*, mp_limb_t *coeff_bits*, *slong exp_bound*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to a random rational function where both numerator and denominator have up to *length* terms, coefficients up to size *coeff_bits*, and exponents strictly smaller than *exp_bound*.

8.1.9 Comparisons

int **fmpz_mpoly_q_equal**(**const** *fmpz_mpoly_q_t x*, **const** *fmpz_mpoly_q_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

Returns whether x and y are equal.

8.1.10 Arithmetic

void **fmpz_mpoly_q_neg**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to the negation of x.

void **fmpz_mpoly_q_add**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_mpoly_q_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_add_fmpq**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpq_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_add_fmpz**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_add_si**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, *slong y*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to the sum of x and y.

void **fmpz_mpoly_q_sub**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_mpoly_q_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_sub_fmpq**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpq_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_sub_fmpz**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_sub_si**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, *slong y*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to the difference of x and y.

void **fmpz_mpoly_q_mul**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_mpoly_q_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_mul_fmpq**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpq_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_mul_fmpz**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_mul_si**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, *slong y*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to the product of x and y.

void **fmpz_mpoly_q_div**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_mpoly_q_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_div_fmpq**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpq_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_div_fmpz**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_t y*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_div_si**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, *slong y*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to the quotient of x and y. Division by zero calls *flint_abort*.

void **fmpz_mpoly_q_inv**(*fmpz_mpoly_q_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to the inverse of x. Division by zero calls *flint_abort*.

8.1.11 Content

void **_fmpz_mpoly_q_content**(*fmpz_t num*, *fmpz_t den*, **const** *fmpz_mpoly_t xnum*, **const** *fmpz_mpoly_t xden*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_q_content**(*fmpq_t res*, **const** *fmpz_mpoly_q_t x*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to the content of the coefficients of x.

8.2 qqbar.h – algebraic numbers represented by minimal polynomials

A *qqbar_t* represents a real or complex algebraic number (an element of $\overline{\mathbb{Q}}$) by its unique reduced minimal polynomial in $\mathbb{Z}[x]$ and an isolating complex interval. The precision of isolating intervals is maintained automatically to ensure that all operations on *qqbar_t* instances are exact.

This representation is useful for working with individual algebraic numbers of moderate degree (up to 100, say). Arithmetic in this representation is expensive: an arithmetic operation on numbers of degrees m and n involves computing and then factoring an annihilating polynomial of degree mn and potentially also performing numerical root-finding. For doing repeated arithmetic, it is generally more efficient to work with the *ca_t* type in a fixed number field. The *qqbar_t* type is used internally by the *ca_t* type to represent the embedding of number fields in $\mathbb{R}$ or $\mathbb{C}$ and to decide predicates for algebraic numbers.

8.2.1 Types and macros

type qqbar_struct

type qqbar_t
A *qqbar_struct* consists of an *fmpz_poly_struct* and an *acb_struct*. A *qqbar_t* is defined as an array of length one of type *qqbar_struct*, permitting a *qqbar_t* to be passed by reference.

type qqbar_ptr
Alias for `qqbar_struct *`, used for *qqbar* vectors.

type qqbar_srcptr
Alias for `const qqbar_struct *`, used for *qqbar* vectors when passed as readonly input to functions.

QQBAR_POLY(x)
Macro returning a pointer to the minimal polynomial of x which can be used as an *fmpz_poly_t*.

QQBAR_COEFFS(x)
Macro returning a pointer to the array of *fmpz* coefficients of the minimal polynomial of x.

QQBAR_ENCLOSURE(x)
Macro returning a pointer to the enclosure of x which can be used as an *acb_t*.

8.2.2 Memory management

void **qqbar_init**(*qqbar_t res*)
Initializes the variable *res* for use, and sets its value to zero.

void **qqbar_clear**(*qqbar_t res*)
Clears the variable *res*, freeing or recycling its allocated memory.

qqbar_ptr **_qqbar_vec_init**(*slong len*)
Returns a pointer to an array of *len* initialized *qqbar_struct*:s.

void **_qqbar_vec_clear**(*qqbar_ptr vec*, *slong len*)
Clears all *len* entries in the vector *vec* and frees the vector itself.

8.2.3 Assignment

void **qqbar_swap**(*qqbar_t x*, *qqbar_t y*)
Swaps the values of x and y efficiently.

void **qqbar_set**(*qqbar_t res*, **const** *qqbar_t x*)

void **qqbar_set_si**(*qqbar_t res*, *slong x*)

void **qqbar_set_ui**(*qqbar_t res*, *ulong x*)

void **qqbar_set_fmpz**(*qqbar_t res*, **const** *fmpz_t x*)

void **qqbar_set_fmpq**(*qqbar_t res*, **const** *fmpq_t x*)
Sets *res* to the value x.

void **qqbar_set_re_im**(*qqbar_t res*, **const** *qqbar_t x*, **const** *qqbar_t y*)
Sets *res* to the value $x + yi$.

int **qqbar_set_d**(*qqbar_t res*, double *x*)

int **qqbar_set_re_im_d**(*qqbar_t res*, double *x*, double *y*)
Sets *res* to the value x or $x + yi$ respectively. These functions performs error handling: if x and y are finite, the conversion succeeds and the return flag is 1. If x or y is non-finite (infinity or NaN), the conversion fails and the return flag is 0.

8.2.4 Properties

slong **qqbar_degree**(**const** *qqbar_t x*)
Returns the degree of x, i.e. the degree of the minimal polynomial.

int **qqbar_is_rational**(**const** *qqbar_t x*)
Returns whether x is a rational number.

int **qqbar_is_integer**(**const** *qqbar_t x*)
Returns whether x is an integer (an element of $\mathbb{Z}$).

int **qqbar_is_algebraic_integer**(**const** *qqbar_t x*)
Returns whether x is an algebraic integer, i.e. whether its minimal polynomial has leading coefficient 1.

int **qqbar_is_zero**(**const** *qqbar_t x*)

int **qqbar_is_one**(**const** *qqbar_t x*)

int **qqbar_is_neg_one**(**const** *qqbar_t x*)
Returns whether x is the number 0, 1, -1.

int **qqbar_is_i**(**const** *qqbar_t x*)

int **qqbar_is_neg_i**(**const** *qqbar_t x*)
Returns whether x is the imaginary unit i (respectively $-i$).

int **qqbar_is_real**(**const** *qqbar_t x*)
Returns whether x is a real number.

void **qqbar_height**(*fmpz_t res*, **const** *qqbar_t x*)
Sets *res* to the height of x (the largest absolute value of the coefficients of the minimal polynomial of x).

slong **qqbar_height_bits**(**const** *qqbar_t x*)
Returns the height of x (the largest absolute value of the coefficients of the minimal polynomial of x) measured in bits.

int **qqbar_within_limits**(**const** *qqbar_t x*, *slong deg_limit*, *slong bits_limit*)
Checks if x has degree bounded by *deg_limit* and height bounded by *bits_limit* bits, returning 0 (false) or 1 (true). If *deg_limit* is set to 0, the degree check is skipped, and similarly for *bits_limit*.

int **qqbar_binop_within_limits**(**const** *qqbar_t x*, **const** *qqbar_t y*, *slong deg_limit*, *slong bits_limit*)
Checks if $x + y$, $x - y$, $x \cdot y$ and x/y certainly have degree bounded by *deg_limit* (by multiplying the degrees for x and y to obtain a trivial bound). For *bits_limits*, the sum of the bit heights of x and y is checked against the bound (this is only a heuristic). If *deg_limit* is set to 0, the degree check is skipped, and similarly for *bits_limit*.

8.2.5 Conversions

void **_qqbar_get_fmpq**(*fmpz_t num*, *fmpz_t den*, **const** *qqbar_t x*)
Sets *num* and *den* to the numerator and denominator of x. Aborts if x is not a rational number.

void **qqbar_get_fmpq**(*fmpq_t res*, **const** *qqbar_t x*)
Sets *res* to x. Aborts if x is not a rational number.

void **qqbar_get_fmpz**(*fmpz_t res*, **const** *qqbar_t x*)
Sets *res* to x. Aborts if x is not an integer.

8.2.6 Special values

void **qqbar_zero**(*qqbar_t res*)
Sets *res* to the number 0.

void **qqbar_one**(*qqbar_t res*)
Sets *res* to the number 1.

void **qqbar_i**(*qqbar_t res*)
Sets *res* to the imaginary unit i.

void **qqbar_phi**(*qqbar_t res*)
Sets *res* to the golden ratio $\varphi = \frac{1}{2}(\sqrt{5} + 1)$.

8.2.7 Input and output

void **qqbar_print**(**const** *qqbar_t x*)
Prints *res* to standard output. The output shows the degree and the list of coefficients of the minimal polynomial followed by a decimal representation of the enclosing interval. This function is mainly intended for debugging.

void **qqbar_printn**(**const** *qqbar_t x*, *slong n*)
Prints *res* to standard output. The output shows a decimal approximation to n digits.

void **qqbar_printnd**(**const** *qqbar_t x*, *slong n*)
Prints *res* to standard output. The output shows a decimal approximation to n digits, followed by the degree of the number.

For example, *print*, *printn* and *printnd* with $n = 6$ give the following output for the numbers 0, 1, i, φ, $\sqrt{2} - \sqrt{3}i$:

```
deg 1 [0, 1] 0
deg 1 [-1, 1] 1.00000
deg 2 [1, 0, 1] 1.00000*I
deg 2 [-1, -1, 1] [1.6180339887498948482045868343656381177 2 +/- 6.00e-39]
deg 4 [25, 0, 2, 0, 1] [1.4142135623730950488016887242096980786 +/- 8.67e-38] + [-1.
↪7320508075688772935274463415058723 67 +/- 1.10e-37]*I

0
1.00000
1.00000*I
1.61803
```

(continues on next page)

(continued from previous page)

```
1.41421 - 1.73205*I

0 (deg 1)
1.00000 (deg 1)
1.00000*I (deg 2)
1.61803 (deg 2)
1.41421 - 1.73205*I (deg 4)
```

8.2.8 Random generation

void **qqbar_randtest**(*qqbar_t res*, flint_rand_t *state*, *slong deg*, *slong bits*)
Sets *res* to a random algebraic number with degree up to *deg* and with height (measured in bits) up to *bits*.

void **qqbar_randtest_real**(*qqbar_t res*, flint_rand_t *state*, *slong deg*, *slong bits*)
Sets *res* to a random real algebraic number with degree up to *deg* and with height (measured in bits) up to *bits*.

void **qqbar_randtest_nonreal**(*qqbar_t res*, flint_rand_t *state*, *slong deg*, *slong bits*)
Sets *res* to a random nonreal algebraic number with degree up to *deg* and with height (measured in bits) up to *bits*. Since all algebraic numbers of degree 1 are real, *deg* must be at least 2.

8.2.9 Comparisons

int **qqbar_equal**(**const** *qqbar_t x*, **const** *qqbar_t y*)
Returns whether x and y are equal.

int **qqbar_equal_fmpq_poly_val**(**const** *qqbar_t x*, **const** *fmpq_poly_t f*, **const** *qqbar_t y*)
Returns whether x is equal to $f(y)$. This function is more efficient than evaluating $f(y)$ and comparing the results.

int **qqbar_cmp_re**(**const** *qqbar_t x*, **const** *qqbar_t y*)
Compares the real parts of x and y, returning -1, 0 or +1.

int **qqbar_cmp_im**(**const** *qqbar_t x*, **const** *qqbar_t y*)
Compares the imaginary parts of x and y, returning -1, 0 or +1.

int **qqbar_cmpabs_re**(**const** *qqbar_t x*, **const** *qqbar_t y*)
Compares the absolute values of the real parts of x and y, returning -1, 0 or +1.

int **qqbar_cmpabs_im**(**const** *qqbar_t x*, **const** *qqbar_t y*)
Compares the absolute values of the imaginary parts of x and y, returning -1, 0 or +1.

int **qqbar_cmpabs**(**const** *qqbar_t x*, **const** *qqbar_t y*)
Compares the absolute values of x and y, returning -1, 0 or +1.

int **qqbar_cmp_root_order**(**const** *qqbar_t x*, **const** *qqbar_t y*)
Compares x and y using an arbitrary but convenient ordering defined on the complex numbers. This is useful for sorting the roots of a polynomial in a canonical order.

We define the root order as follows: real roots come first, in descending order. Nonreal roots are subsequently ordered first by real part in descending order, then in ascending order by the absolute value of the imaginary part, and then in descending order of the sign. This implies that complex conjugate roots are adjacent, with the root in the upper half plane first.

ulong **qqbar_hash**(**const** *qqbar_t x*)
Returns a hash of x. As currently implemented, this function only hashes the minimal polynomial of x. The user should mix in some bits based on the numerical value if it is critical to distinguish between conjugates of the same minimal polynomial. This function is also likely to produce serial runs of values for lexicographically close minimal polynomials. This is not necessarily a problem

for use in hash tables, but if it is important that all bits in the output are random, the user should apply an integer hash function to the output.

8.2.10 Complex parts

void **qqbar_conj**(*qqbar_t res*, **const** *qqbar_t x*)
Sets *res* to the complex conjugate of *x*.

void **qqbar_re**(*qqbar_t res*, **const** *qqbar_t x*)
Sets *res* to the real part of *x*.

void **qqbar_im**(*qqbar_t res*, **const** *qqbar_t x*)
Sets *res* to the imaginary part of *x*.

void **qqbar_re_im**(*qqbar_t res1*, *qqbar_t res2*, **const** *qqbar_t x*)
Sets *res1* to the real part of *x* and *res2* to the imaginary part of *x*.

void **qqbar_abs**(*qqbar_t res*, **const** *qqbar_t x*)
Sets *res* to the absolute value of *x*:

void **qqbar_abs2**(*qqbar_t res*, **const** *qqbar_t x*)
Sets *res* to the square of the absolute value of *x*.

void **qqbar_sgn**(*qqbar_t res*, **const** *qqbar_t x*)
Sets *res* to the complex sign of *x*, defined as 0 if *x* is zero and as $x/|x|$ otherwise.

int **qqbar_sgn_re**(**const** *qqbar_t x*)
Returns the sign of the real part of *x* (-1, 0 or +1).

int **qqbar_sgn_im**(**const** *qqbar_t x*)
Returns the sign of the imaginary part of *x* (-1, 0 or +1).

int **qqbar_csgn**(**const** *qqbar_t x*)
Returns the extension of the real sign function taking the value 1 for *x* strictly in the right half plane, -1 for *x* strictly in the left half plane, and the sign of the imaginary part when *x* is on the imaginary axis. Equivalently, $\operatorname{csgn}(x) = x/\sqrt{x^2}$ except that the value is 0 when *x* is zero.

8.2.11 Integer parts

void **qqbar_floor**(*fmpz_t res*, **const** *qqbar_t x*)
Sets *res* to the floor function of *x*. If *x* is not real, the value is defined as the floor function of the real part of *x*.

void **qqbar_ceil**(*fmpz_t res*, **const** *qqbar_t x*)
Sets *res* to the ceiling function of *x*. If *x* is not real, the value is defined as the ceiling function of the real part of *x*.

8.2.12 Arithmetic

void **qqbar_neg**(*qqbar_t res*, **const** *qqbar_t x*)
Sets *res* to the negation of *x*.

void **qqbar_add**(*qqbar_t res*, **const** *qqbar_t x*, **const** *qqbar_t y*)

void **qqbar_add_fmpq**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpq_t y*)

void **qqbar_add_fmpz**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpz_t y*)

void **qqbar_add_ui**(*qqbar_t res*, **const** *qqbar_t x*, *ulong y*)

void **qqbar_add_si**(*qqbar_t res*, **const** *qqbar_t x*, *slong y*)
Sets *res* to the sum of *x* and *y*.

void **qqbar_sub**(*qqbar_t res*, **const** *qqbar_t x*, **const** *qqbar_t y*)

void **qqbar_sub_fmpq**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpq_t y*)

void **qqbar_sub_fmpz**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpz_t y*)

void **qqbar_sub_ui**(*qqbar_t res*, **const** *qqbar_t x*, *ulong y*)

void **qqbar_sub_si**(*qqbar_t res*, **const** *qqbar_t x*, *slong y*)

void **qqbar_fmpq_sub**(*qqbar_t res*, **const** *fmpq_t x*, **const** *qqbar_t y*)

void **qqbar_fmpz_sub**(*qqbar_t res*, **const** *fmpz_t x*, **const** *qqbar_t y*)

void **qqbar_ui_sub**(*qqbar_t res*, *ulong x*, **const** *qqbar_t y*)

void **qqbar_si_sub**(*qqbar_t res*, *slong x*, **const** *qqbar_t y*)

Sets *res* to the difference of x and y.

void **qqbar_mul**(*qqbar_t res*, **const** *qqbar_t x*, **const** *qqbar_t y*)

void **qqbar_mul_fmpq**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpq_t y*)

void **qqbar_mul_fmpz**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpz_t y*)

void **qqbar_mul_ui**(*qqbar_t res*, **const** *qqbar_t x*, *ulong y*)

void **qqbar_mul_si**(*qqbar_t res*, **const** *qqbar_t x*, *slong y*)

Sets *res* to the product of x and y.

void **qqbar_mul_2exp_si**(*qqbar_t res*, **const** *qqbar_t x*, *slong e*)

Sets *res* to x multiplied by 2^e.

void **qqbar_sqr**(*qqbar_t res*, **const** *qqbar_t x*)

Sets *res* to the square of x.

void **qqbar_inv**(*qqbar_t res*, **const** *qqbar_t x*, **const** *qqbar_t y*)

Sets *res* to the multiplicative inverse of y. Division by zero calls *flint_abort*.

void **qqbar_div**(*qqbar_t res*, **const** *qqbar_t x*, **const** *qqbar_t y*)

void **qqbar_div_fmpq**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpq_t y*)

void **qqbar_div_fmpz**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpz_t y*)

void **qqbar_div_ui**(*qqbar_t res*, **const** *qqbar_t x*, *ulong y*)

void **qqbar_div_si**(*qqbar_t res*, **const** *qqbar_t x*, *slong y*)

void **qqbar_fmpq_div**(*qqbar_t res*, **const** *fmpq_t x*, **const** *qqbar_t y*)

void **qqbar_fmpz_div**(*qqbar_t res*, **const** *fmpz_t x*, **const** *qqbar_t y*)

void **qqbar_ui_div**(*qqbar_t res*, *ulong x*, **const** *qqbar_t y*)

void **qqbar_si_div**(*qqbar_t res*, *slong x*, **const** *qqbar_t y*)

Sets *res* to the quotient of x and y. Division by zero calls *flint_abort*.

void **qqbar_scalar_op**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpz_t a*, **const** *fmpz_t b*, **const** *fmpz_t c*)

Sets *res* to the rational affine transformation $(ax + b)/c$, performed as a single operation. There are no restrictions on a, b and c except that c must be nonzero. Division by zero calls *flint_abort*.

8.2.13 Powers and roots

void **qqbar_sqrt**(*qqbar_t res*, **const** *qqbar_t x*)

void **qqbar_sqrt_ui**(*qqbar_t res*, *ulong x*)

Sets *res* to the principal square root of x.

void **qqbar_rsqrt**(*qqbar_t res*, **const** *qqbar_t x*)

Sets *res* to the reciprocal of the principal square root of x. Division by zero calls *flint_abort*.

void **qqbar_pow_ui**(*qqbar_t res*, **const** *qqbar_t x*, *ulong n*)

void **qqbar_pow_si**(*qqbar_t res*, **const** *qqbar_t x*, *ulong n*)

void **qqbar_pow_fmpz**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpz_t n*)

void **qqbar_pow_fmpq**(*qqbar_t res*, **const** *qqbar_t x*, **const** *fmpq_t n*)

Sets *res* to x raised to the n-th power. Raising zero to a negative power aborts.

void **qqbar_root_ui**(*qqbar_t* *res*, **const** *qqbar_t* *x*, *ulong* *n*)

void **qqbar_fmpq_root_ui**(*qqbar_t* *res*, **const** *fmpq_t* *x*, *ulong* *n*)
Sets *res* to the principal n-th root of x. The order n must be positive.

void **qqbar_fmpq_pow_si_ui**(*qqbar_t* *res*, **const** *fmpq_t* *x*, *slong* *m*, *ulong* *n*)
Sets *res* to the principal branch of $x^{m/n}$. The order n must be positive. Division by zero calls *flint_abort*.

int **qqbar_pow**(*qqbar_t* *res*, **const** *qqbar_t* *x*, **const** *qqbar_t* *y*)
General exponentiation: if x^y is an algebraic number, sets *res* to this value and returns 1. If x^y is transcendental or undefined, returns 0. Note that this function returns 0 instead of aborting on division zero.

8.2.14 Numerical enclosures

The following functions guarantee a polished output in which both the real and imaginary parts are accurate to *prec* bits and exact when exactly representable (that is, when a real or imaginary part is a sufficiently small dyadic number). In some cases, the computations needed to polish the output may be expensive. When polish is unnecessary, *qqbar_enclosure_raw()* may be used instead. Alternatively, *qqbar_cache_enclosure()* can be used to avoid recomputations.

void **qqbar_get_acb**(*acb_t* *res*, **const** *qqbar_t* *x*, *slong* *prec*)
Sets *res* to an enclosure of x rounded to *prec* bits.

void **qqbar_get_arb**(*arb_t* *res*, **const** *qqbar_t* *x*, *slong* *prec*)
Sets *res* to an enclosure of x rounded to *prec* bits, assuming that x is a real number. If x is not real, *res* is set to $[\mathrm{NaN} \pm\infty]$.

void **qqbar_get_arb_re**(*arb_t* *res*, **const** *qqbar_t* *x*, *slong* *prec*)
Sets *res* to an enclosure of the real part of x rounded to *prec* bits.

void **qqbar_get_arb_im**(*arb_t* *res*, **const** *qqbar_t* *x*, *slong* *prec*)
Sets *res* to an enclosure of the imaginary part of x rounded to *prec* bits.

void **qqbar_cache_enclosure**(*qqbar_t* *res*, *slong* *prec*)
Polishes the internal enclosure of *res* to at least *prec* bits of precision in-place. Normally, *qqbar* operations that need high-precision enclosures compute them on the fly without caching the results; if *res* will be used as an invariant operand for many operations, calling this function as a precomputation step can improve performance.

8.2.15 Numerator and denominator

void **qqbar_denominator**(*fmpz_t* *res*, **const** *qqbar_t* *y*)
Sets *res* to the denominator of y, i.e. the leading coefficient of the minimal polynomial of y.

void **qqbar_numerator**(*qqbar_t* *res*, **const** *qqbar_t* *y*)
Sets *res* to the numerator of y, i.e. y multiplied by its denominator.

8.2.16 Conjugates

void **qqbar_conjugates**(*qqbar_ptr* *res*, **const** *qqbar_t* *x*)
Sets the entries of the vector *res* to the d algebraic conjugates of x, including x itself, where d is the degree of x. The output is sorted in a canonical order (as defined by *qqbar_cmp_root_order()*).

8.2.17 Polynomial evaluation

void **_qqbar_evaluate_fmpq_poly**(*qqbar_t res*, **const** fmpz *poly, **const** *fmpz_t den*, *slong len*, **const** *qqbar_t x*)

void **qqbar_evaluate_fmpq_poly**(*qqbar_t res*, **const** *fmpq_poly_t poly*, **const** *qqbar_t x*)

void **_qqbar_evaluate_fmpz_poly**(*qqbar_t res*, **const** fmpz *poly, *slong len*, **const** *qqbar_t x*)

void **qqbar_evaluate_fmpz_poly**(*qqbar_t res*, **const** *fmpz_poly_t poly*, **const** *qqbar_t x*)

Sets *res* to the value of the given polynomial *poly* evaluated at the algebraic number *x*. These methods detect simple special cases and automatically reduce *poly* if its degree is greater or equal to that of the minimal polynomial of *x*. In the generic case, evaluation is done by computing minimal polynomials of representation matrices.

int **qqbar_evaluate_fmpz_mpoly_iter**(*qqbar_t res*, **const** *fmpz_mpoly_t poly*, *qqbar_srcptr x*, *slong deg_limit*, *slong bits_limit*, **const** *fmpz_mpoly_ctx_t ctx*)

int **qqbar_evaluate_fmpz_mpoly_horner**(*qqbar_t res*, **const** *fmpz_mpoly_t poly*, *qqbar_srcptr x*, *slong deg_limit*, *slong bits_limit*, **const** *fmpz_mpoly_ctx_t ctx*)

int **qqbar_evaluate_fmpz_mpoly**(*qqbar_t res*, **const** *fmpz_mpoly_t poly*, *qqbar_srcptr x*, *slong deg_limit*, *slong bits_limit*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to the value of *poly* evaluated at the algebraic numbers given in the vector *x*. The number of variables is defined by the context object *ctx*.

The parameters *deg_limit* and *bits_limit* define evaluation limits: if any temporary result exceeds these limits (not necessarily the final value, in case of cancellation), the evaluation is aborted and 0 (failure) is returned. If evaluation succeeds, 1 is returned.

The *iter* version iterates over all terms in succession and computes the powers that appear. The *horner* version uses a multivariate implementation of the Horner scheme. The default algorithm currently uses the Horner scheme.

8.2.18 Polynomial roots

void **qqbar_roots_fmpz_poly**(*qqbar_ptr res*, **const** *fmpz_poly_t poly*, int *flags*)

void **qqbar_roots_fmpq_poly**(*qqbar_ptr res*, **const** *fmpq_poly_t poly*, int *flags*)

Sets the entries of the vector *res* to the *d* roots of the polynomial *poly*. Roots with multiplicity appear with repetition in the output array. By default, the roots will be sorted in a convenient canonical order (as defined by *qqbar_cmp_root_order()*). Instances of a repeated root always appear consecutively.

The following *flags* are supported:

- QQBAR_ROOTS_IRREDUCIBLE - if set, *poly* is assumed to be irreducible (it may still have constant content), and no polynomial factorization is performed internally.
- QQBAR_ROOTS_UNSORTED - if set, the roots will not be guaranteed to be sorted (except for repeated roots being listed consecutively).

void **qqbar_eigenvalues_fmpz_mat**(*qqbar_ptr res*, **const** *fmpz_mat_t mat*, int *flags*)

void **qqbar_eigenvalues_fmpq_mat**(*qqbar_ptr res*, **const** *fmpz_mat_t mat*, int *flags*)

Sets the entries of the vector *res* to the eigenvalues of the square matrix *mat*. These functions compute the characteristic polynomial of *mat* and then call *qqbar_roots_fmpz_poly()* with the same flags.

8.2.19 Roots of unity and trigonometric functions

The following functions use word-size integers p and q instead of *fmpq_t* instances to express rational numbers. This is to emphasize that the computations are feasible only with small q in this representation of algebraic numbers since the associated minimal polynomials have degree $O(q)$. The input p and q do not need to be reduced *a priori*, but should not be close to the word boundaries (they may be added and subtracted internally).

void **qqbar_root_of_unity**(*qqbar_t res*, *slong p*, *ulong q*)
Sets *res* to the root of unity $e^{2\pi ip/q}$.

int **qqbar_is_root_of_unity**(*slong* *p, *ulong* *q, **const** *qqbar_t x*)
If x is not a root of unity, returns 0. If x is a root of unity, returns 1. If p and q are not *NULL* and x is a root of unity, this also sets p and q to the minimal integers with $0 \le p < q$ such that $x = e^{2\pi ip/q}$.

void **qqbar_exp_pi_i**(*qqbar_t res*, *slong p*, *ulong q*)
Sets *res* to the root of unity $e^{\pi ip/q}$.

void **qqbar_cos_pi**(*qqbar_t res*, *slong p*, *ulong q*)

void **qqbar_sin_pi**(*qqbar_t res*, *slong p*, *ulong q*)

int **qqbar_tan_pi**(*qqbar_t res*, *slong p*, *ulong q*)

int **qqbar_cot_pi**(*qqbar_t res*, *slong p*, *ulong q*)

int **qqbar_sec_pi**(*qqbar_t res*, *slong p*, *ulong q*)

int **qqbar_csc_pi**(*qqbar_t res*, *slong p*, *ulong q*)
Sets *res* to the trigonometric function $\cos(\pi x)$, $\sin(\pi x)$, etc., with $x = \frac{p}{q}$. The functions tan, cot, sec and csc return the flag 1 if the value exists, and return 0 if the evaluation point is a pole of the function.

int **qqbar_log_pi_i**(*slong* *p, *ulong* *q, **const** *qqbar_t x*)
If $y = \log(x)/(\pi i)$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $-1 < y \le 1$ and returns 1. If y is not algebraic, returns 0.

int **qqbar_atan_pi**(*slong* *p, *ulong* *q, **const** *qqbar_t x*)
If $y = \operatorname{atan}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $|y| < \frac{1}{2}$ and returns 1. If y is not algebraic, returns 0.

int **qqbar_asin_pi**(*slong* *p, *ulong* *q, **const** *qqbar_t x*)
If $y = \operatorname{asin}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $|y| \le \frac{1}{2}$ and returns 1. If y is not algebraic, returns 0.

int **qqbar_acos_pi**(*slong* *p, *ulong* *q, **const** *qqbar_t x*)
If $y = \operatorname{acos}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $0 \le y \le 1$ and returns 1. If y is not algebraic, returns 0.

int **qqbar_acot_pi**(*slong* *p, *ulong* *q, **const** *qqbar_t x*)
If $y = \operatorname{acot}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $-\frac{1}{2} < y \le \frac{1}{2}$ and returns 1. If y is not algebraic, returns 0.

int **qqbar_asec_pi**(*slong* *p, *ulong* *q, **const** *qqbar_t x*)
If $y = \operatorname{asec}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $0 \le y \le 1$ and returns 1. If y is not algebraic, returns 0.

int **qqbar_acsc_pi**(*slong* *p, *ulong* *q, **const** *qqbar_t x*)
If $y = \operatorname{acsc}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $-\frac{1}{2} \le y \le \frac{1}{2}$ and returns 1. If y is not algebraic, returns 0.

8.2.20 Guessing and simplification

int **qqbar_guess**(*qqbar_t res*, **const** *acb_t z*, *slong max_deg*, *slong max_bits*, int *flags*, *slong prec*)

Attempts to find an algebraic number *res* of degree at most *max_deg* and height at most *max_bits* bits matching the numerical enclosure *z*. The return flag indicates success. This is only a heuristic method, and the return flag neither implies a rigorous proof that *res* is the correct result, nor a rigorous proof that no suitable algebraic number with the given *max_deg* and *max_bits* exists. (Proof of nonexistence could in principle be computed, but this is not yet implemented.)

The working precision *prec* should normally be the same as the precision used to compute *z*. It does not make much sense to run this algorithm with precision smaller than $O(max\_deg \cdot max\_bits)$.

This function does a single iteration at the target *max_deg*, *max_bits*, and *prec*. For best performance, one should invoke this function repeatedly with successively larger parameters when the size of the intended solution is unknown or may be much smaller than a worst-case bound.

int **qqbar_express_in_field**(*fmpq_poly_t res*, **const** *qqbar_t alpha*, **const** *qqbar_t x*, *slong max_bits*, int *flags*, *slong prec*)

Attempts to express *x* in the number field generated by *alpha*, returning success (0 or 1). On success, *res* is set to a polynomial f of degree less than the degree of *alpha* and with height (counting both the numerator and the denominator, when the coefficients of g are put on a common denominator) bounded by *max_bits* bits, such that $f(\alpha) = x$.

(Exception: the *max_bits* parameter is currently ignored if *x* is rational, in which case *res* is just set to the value of *x*.)

This function looks for a linear relation heuristically using a working precision of *prec* bits. If *x* is expressible in terms of *alpha*, then this function is guaranteed to succeed when *prec* is taken large enough. The identity $f(\alpha) = x$ is checked rigorously, i.e. a return value of 1 implies a proof of correctness. In principle, choosing a sufficiently large *prec* can be used to prove that *x* does not lie in the field generated by *alpha*, but the present implementation does not support doing so automatically.

This function does a single iteration at the target *max_bits* and and *prec*. For best performance, one should invoke this function repeatedly with successively larger parameters when the size of the intended solution is unknown or may be much smaller than a worst-case bound.

8.2.21 Symbolic expressions and conversion to radicals

void **qqbar_get_quadratic**(*fmpz_t a*, *fmpz_t b*, *fmpz_t c*, **const** *fmpz_t q*, **const** *qqbar_t x*, int *factoring*)

Assuming that *x* has degree 1 or 2, computes integers *a*, *b*, *c* and *q* such that

$$x = \frac{a + b\sqrt{c}}{q}$$

and such that *c* is not a perfect square, *q* is positive, and *q* has no content in common with both *a* and *b*. In other words, this determines a quadratic field $\mathbb{Q}(\sqrt{c})$ containing *x*, and then finds the canonical reduced coefficients *a*, *b* and *q* expressing *x* in this field. For convenience, this function supports rational *x*, for which *b* and *c* will both be set to zero. The following remarks apply to irrationals.

The radicand *c* will not be a perfect square, but will not automatically be squarefree since this would require factoring the discriminant. As a special case, *c* will be set to -1 if *x* is a Gaussian rational number. Otherwise, behavior is controlled by the *factoring* parameter.

- If *factoring* is 0, no factorization is performed apart from removing powers of two.
- If *factoring* is 1, a complete factorization is performed (*c* will be minimal). This can be very expensive if the discriminant is large.

- If *factoring* is 2, a smooth factorization is performed to remove small factors from c. This is a tradeoff that provides pretty output in most cases while avoiding extreme worst-case slowdown. The smooth factorization guarantees finding all small factors (up to some trial division limit determined internally by Flint), but large factors are only found heuristically.

int **qqbar_set_fexpr**(*qqbar_t res*, **const** *fexpr_t expr*)

Sets *res* to the algebraic number represented by the symbolic expression *expr*, returning 1 on success and 0 on failure.

This function performs a "static" evaluation using *qqbar* arithmetic, supporting only closed-form expressions with explicitly algebraic subexpressions. It can be used to recover values generated by `qqbar_get_expr_formula()` and variants. For evaluating more complex expressions involving other types of values or requiring symbolic simplifications, the user should preprocess *expr* so that it is in a form which can be parsed by *qqbar_set_fexpr()*.

The following expressions are supported:

- Integer constants
- Arithmetic operations with algebraic operands
- Square roots of algebraic numbers
- Powers with algebraic base and exponent an explicit rational number
- NumberI, GoldenRatio, RootOfUnity
- Floor, Ceil, Abs, Sign, Csgn, Conjugate, Re, Im, Max, Min
- Trigonometric functions with argument an explicit rational number times Pi
- Exponentials with argument an explicit rational number times Pi * NumberI
- The Decimal() constructor
- AlgebraicNumberSerialized() (assuming valid data, which is not checked)
- PolynomialRootIndexed()
- PolynomialRootNearest()

Examples of formulas that are not supported, despite the value being an algebraic number:

- `Pi - Pi` (general transcendental simplifications are not performed)
- `1 / Infinity` (only numbers are handled)
- `Sum(n, For(n, 1, 10))` (only static evaluation is performed)

void **qqbar_get_fexpr_repr**(*fexpr_t res*, **const** *qqbar_t x*)

Sets *res* to a symbolic expression reflecting the exact internal representation of x. The output will have the form `AlgebraicNumberSerialized(List(coeffs), enclosure)`. The output can be converted back to a `qqbar_t` value using *qqbar_set_fexpr()*. This is the recommended format for serializing algebraic numbers as it requires minimal computation, but it has the disadvantage of not being human-readable.

void **qqbar_get_fexpr_root_nearest**(*fexpr_t res*, **const** *qqbar_t x*)

Sets *res* to a symbolic expression unambiguously describing x in the form `PolynomialRootNearest(List(coeffs), point)` where *point* is an approximation of x guaranteed to be closer to x than any conjugate root. The output can be converted back to a `qqbar_t` value using *qqbar_set_fexpr()*. This is a useful format for human-readable presentation, but serialization and deserialization can be expensive.

void **qqbar_get_fexpr_root_indexed**(*fexpr_t res*, **const** *qqbar_t x*)

Sets *res* to a symbolic expression unambiguously describing x in the form `PolynomialRootIndexed(List(coeffs), index)` where *index* is the index of x among its conjugate roots in the builtin root sort order. The output can be converted back to a `qqbar_t` value using *qqbar_set_fexpr()*. This is a useful format for human-readable presentation when the numerical value is important, but serialization and deserialization can be expensive.

int **qqbar_get_fexpr_formula**(*fexpr_t res*, **const** *qqbar_t x*, *ulong flags*)

Attempts to express the algebraic number x as a closed-form expression using arithmetic operations, radicals, and possibly exponentials or trigonometric functions, but without using `PolynomialRootNearest` or `PolynomialRootIndexed`. Returns 0 on failure and 1 on success.

The *flags* parameter toggles different methods for generating formulas. It can be set to any combination of the following. If *flags* is 0, only rational numbers will be handled.

QQBAR_FORMULA_ALL
Toggles all methods (potentially expensive).

QQBAR_FORMULA_GAUSSIANS
Detect Gaussian rational numbers $a + bi$.

QQBAR_FORMULA_QUADRATICS
Solve quadratics in the form $a + b\sqrt{d}$.

QQBAR_FORMULA_CYCLOTOMICS
Detect elements of cyclotomic fields. This works by trying plausible cyclotomic fields (based on the degree of the input), using LLL to find candidate number field elements, and certifying candidates through an exact computation. Detection is heuristic and is not guaranteed to find all cyclotomic numbers.

QQBAR_FORMULA_CUBICS

QQBAR_FORMULA_QUARTICS

QQBAR_FORMULA_QUINTICS
Solve polynomials of degree 3, 4 and (where applicable) 5 using cubic, quartic and quintic formulas (not yet implemented).

QQBAR_FORMULA_DEPRESSION
Use depression to try to generate simpler numbers.

QQBAR_FORMULA_DEFLATION
Use deflation to try to generate simpler numbers. This allows handling number of the form $a^{1/n}$ where a can be represented in closed form.

QQBAR_FORMULA_SEPARATION
Try separating real and imaginary parts or sign and magnitude of complex numbers. This allows handling numbers of the form $a+bi$ or $m \cdot s$ (with $m > 0, |s| = 1$) where a and b or m and s can be represented in closed form. This is only attempted as a fallback after other methods fail: if an explicit Cartesian or magnitude-sign represented is desired, the user should manually separate the number into complex parts before calling *qqbar_get_fexpr_formula()*.

QQBAR_FORMULA_EXP_FORM

QQBAR_FORMULA_TRIG_FORM

QQBAR_FORMULA_RADICAL_FORM

QQBAR_FORMULA_AUTO_FORM
Select output form for cyclotomic numbers. The *auto* form (equivalent to no flags being set) results in radicals for numbers of low degree, trigonometric functions for real numbers, and complex exponentials for nonreal numbers. The other flags (not fully implemented) can be used to force exponential form, trigonometric form, or radical form.

8.2.22 Internal functions

void **qqbar_fmpz_poly_composed_op**(*fmpz_poly_t res*, **const** *fmpz_poly_t A*, **const** *fmpz_poly_t B*, int *op*)

Given nonconstant polynomials A and B, sets *res* to a polynomial whose roots are $a+b$, $a-b$, ab or a/b for all roots a of A and all roots b of B. The parameter *op* selects the arithmetic operation: 0 for addition, 1 for subtraction, 2 for multiplication and 3 for division. If *op* is 3, B must not have zero as a root.

void **qqbar_binary_op**(*qqbar_t res*, **const** *qqbar_t x*, **const** *qqbar_t y*, int *op*)

Performs a binary operation using a generic algorithm. This does not check for special cases.

int **_qqbar_validate_uniqueness**(*acb_t res*, **const** *fmpz_poly_t poly*, **const** *acb_t z*, *slong max_prec*)

Given z known to be an enclosure of at least one root of *poly*, certifies that the enclosure contains a unique root, and in that case sets *res* to a new (possibly improved) enclosure for the same root, returning 1. Returns 0 if uniqueness cannot be certified.

The enclosure is validated by performing a single step with the interval Newton method. The working precision is determined from the accuracy of z, but limited by *max_prec* bits.

This method slightly inflates the enclosure z to improve the chances that the interval Newton step will succeed. Uniqueness on this larger interval implies uniqueness of the original interval, but not existence; when existence has not been ensured a priori, *_qqbar_validate_existence_uniqueness()* should be used instead.

int **_qqbar_validate_existence_uniqueness**(*acb_t res*, **const** *fmpz_poly_t poly*, **const** *acb_t z*, *slong max_prec*)

Given any complex interval z, certifies that the enclosure contains a unique root of *poly*, and in that case sets *res* to a new (possibly improved) enclosure for the same root, returning 1. Returns 0 if existence and uniqueness cannot be certified.

The enclosure is validated by performing a single step with the interval Newton method. The working precision is determined from the accuracy of z, but limited by *max_prec* bits.

void **_qqbar_enclosure_raw**(*acb_t res*, **const** *fmpz_poly_t poly*, **const** *acb_t z*, *slong prec*)

void **qqbar_enclosure_raw**(*acb_t res*, **const** *qqbar_t x*, *slong prec*)

Sets *res* to an enclosure of x accurate to about *prec* bits (the actual accuracy can be slightly lower, or higher).

This function uses repeated interval Newton steps to polish the initial enclosure z, doubling the working precision each time. If any step fails to improve the accuracy significantly, the root is recomputed from scratch to higher precision.

If the initial enclosure is accurate enough, *res* is set to this value without rounding and without further computation.

int **_qqbar_acb_lindep**(fmpz **rel*, acb_srcptr *vec*, *slong len*, int *check*, *slong prec*)

Attempts to find an integer vector *rel* giving a linear relation between the elements of the real or complex vector *vec*, using the LLL algorithm.

The working precision is set to the minimum of *prec* and the relative accuracy of *vec* (that is, the difference between the largest magnitude and the largest error magnitude within *vec*). 95% of the bits within the working precision are used for the LLL matrix, and the remaining 5% bits are used to validate the linear relation by evaluating the linear combination and checking that the resulting interval contains zero. This validation does not prove the existence or nonexistence of a linear relation, but it provides a quick heuristic way to eliminate spurious relations.

If *check* is set, the return value indicates whether the validation was successful; otherwise, the return value simply indicates whether the algorithm was executed normally (failure may occur, for example, if the input vector is non-finite).

In principle, this method can be used to produce a proof that no linear relation exists with coefficients up to a specified bit size, but this has not yet been implemented.

8.3 utils_flint.h – extra methods for Flint types

8.3.1 General methods for multivariate polynomials

void **fmpz_mpoly_primitive_part**(*fmpz_mpoly_t res*, **const** *fmpz_mpoly_t f*, **const** *fmpz_mpoly_ctx_t ctx*)
Sets *res* to the primitive part of *f*, obtained by dividing out the content of all coefficients and normalizing the leading coefficient to be positive. The zero polynomial is unchanged.

void **fmpz_mpoly_symmetric_gens**(*fmpz_mpoly_t res*, *ulong k*, *slong* *vars, *slong n*, **const** *fmpz_mpoly_ctx_t ctx*)

void **fmpz_mpoly_symmetric**(*fmpz_mpoly_t res*, *ulong k*, **const** *fmpz_mpoly_ctx_t ctx*)
Sets *res* to the elementary symmetric polynomial $e_k(X_1, \ldots, X_n)$.

The *gens* version takes $X_1, \ldots, X_n$ to be the subset of generators given by *vars* and *n*. The indices in *vars* start from zero. Currently, the indices in *vars* must be distinct.

type fmpz_mpoly_vec_struct

type fmpz_mpoly_vec_t
A type holding a vector of *fmpz_mpoly_t*.

fmpz_mpoly_vec_entry(*vec*, *i*)
Macro for accessing the entry at position *i* in *vec*.

void **fmpz_mpoly_vec_init**(*fmpz_mpoly_vec_t vec*, *slong len*, **const** *fmpz_mpoly_ctx_t ctx*)
Initializes *vec* to a vector of length *len*, setting all entries to the zero polynomial.

void **fmpz_mpoly_vec_print**(**const** *fmpz_mpoly_vec_t vec*, **const** *fmpz_mpoly_ctx_t ctx*)
Prints *vec* to standard output.

void **fmpz_mpoly_vec_swap**(*fmpz_mpoly_vec_t x*, *fmpz_mpoly_vec_t y*, **const** *fmpz_mpoly_ctx_t ctx*)
Swaps *x* and *y* efficiently.

void **fmpz_mpoly_vec_fit_length**(*fmpz_mpoly_vec_t vec*, *slong len*, **const** *fmpz_mpoly_ctx_t ctx*)
Allocates room for *len* entries in *vec*.

void **fmpz_mpoly_vec_set**(*fmpz_mpoly_vec_t dest*, **const** *fmpz_mpoly_vec_t src*, **const** *fmpz_mpoly_ctx_t ctx*)
Sets *dest* to a copy of *src*.

void **fmpz_mpoly_vec_append**(*fmpz_mpoly_vec_t vec*, **const** *fmpz_mpoly_t f*, **const** *fmpz_mpoly_ctx_t ctx*)
Appends *f* to the end of *vec*.

slong **fmpz_mpoly_vec_insert_unique**(*fmpz_mpoly_vec_t vec*, **const** *fmpz_mpoly_t f*, **const** *fmpz_mpoly_ctx_t ctx*)
Inserts *f* without duplication into *vec* and returns its index. If this polynomial already exists, *vec* is unchanged. If this polynomial does not exist in *vec*, it is appended.

void **fmpz_mpoly_vec_set_length**(*fmpz_mpoly_vec_t vec*, *slong len*, **const** *fmpz_mpoly_ctx_t ctx*)
Sets the length of *vec* to *len*, truncating or zero-extending as needed.

void **fmpz_mpoly_vec_randtest_not_zero**(*fmpz_mpoly_vec_t vec*, flint_rand_t *state*, *slong len*, *slong poly_len*, *slong bits*, *ulong exp_bound*, *fmpz_mpoly_ctx_t ctx*)
Sets *vec* to a random vector with exactly *len* entries, all nonzero, with random parameters defined by *poly_len*, *bits* and *exp_bound*.

void **fmpz_mpoly_vec_set_primitive_unique**(*fmpz_mpoly_vec_t res*, **const** *fmpz_mpoly_vec_t src*, **const** *fmpz_mpoly_ctx_t ctx*)
Sets *res* to a vector containing all polynomials in *src* reduced to their primitive parts, without

duplication. The zero polynomial is skipped if present. The output order is arbitrary.

8.3.2 Ideals and Gröbner bases

The following methods deal with ideals in $\mathbb{Q}[X_1, \ldots, X_n]$. We use primitive integer polynomials as normalised generators in place of monic rational polynomials.

void **fmpz_mpoly_spoly**(*fmpz_mpoly_t res*, **const** *fmpz_mpoly_t f*, **const** *fmpz_mpoly_t g*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to the S-polynomial of f and g, scaled to an integer polynomial by computing the LCM of the leading coefficients.

void **fmpz_mpoly_reduction_primitive_part**(*fmpz_mpoly_t res*, **const** *fmpz_mpoly_t f*, **const** *fmpz_mpoly_vec_t vec*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets *res* to the primitive part of the reduction (remainder of multivariate quasidivision with remainder) with respect to the polynomials *vec*.

int **fmpz_mpoly_vec_is_groebner**(**const** *fmpz_mpoly_vec_t G*, **const** *fmpz_mpoly_vec_t F*, **const** *fmpz_mpoly_ctx_t ctx*)

If F is *NULL*, checks if G is a Gröbner basis. If F is not *NULL*, checks if G is a Gröbner basis for F.

int **fmpz_mpoly_vec_is_autoreduced**(**const** *fmpz_mpoly_vec_t F*, **const** *fmpz_mpoly_ctx_t ctx*)

Checks whether the vector F is autoreduced (or inter-reduced).

void **fmpz_mpoly_vec_autoreduction**(*fmpz_mpoly_vec_t H*, **const** *fmpz_mpoly_vec_t F*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets H to the autoreduction (inter-reduction) of F.

void **fmpz_mpoly_vec_autoreduction_groebner**(*fmpz_mpoly_vec_t H*, **const** *fmpz_mpoly_vec_t G*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets H to the autoreduction (inter-reduction) of G. Assumes that G is a Gröbner basis. This produces a reduced Gröbner basis, which is unique (up to the sort order of the entries in the vector).

pair_t **fmpz_mpoly_select_pop_pair**(*pairs_t pairs*, **const** *fmpz_mpoly_vec_t G*, **const** *fmpz_mpoly_ctx_t ctx*)

Given a vector *pairs* of indices (i, j) into G, selects one pair for elimination in Buchberger's algorithm. The pair is removed from *pairs* and returned.

void **fmpz_mpoly_buchberger_naive**(*fmpz_mpoly_vec_t G*, **const** *fmpz_mpoly_vec_t F*, **const** *fmpz_mpoly_ctx_t ctx*)

Sets G to a Gröbner basis for F, computed using a naive implementation of Buchberger's algorithm.

int **fmpz_mpoly_buchberger_naive_with_limits**(*fmpz_mpoly_vec_t G*, **const** *fmpz_mpoly_vec_t F*, *slong ideal_len_limit*, *slong poly_len_limit*, *slong poly_bits_limit*, **const** *fmpz_mpoly_ctx_t ctx*)

As *fmpz_mpoly_buchberger_naive()*, but halts if during the execution of Buchberger's algorithm the length of the ideal basis set exceeds *ideal_len_limit*, the length of any polynomial exceeds *poly_len_limit*, or the size of the coefficients of any polynomial exceeds *poly_bits_limit*. Returns 1 for success and 0 for failure. On failure, G is a valid basis for F but it might not be a Gröbner basis.

8.3.3 Index pairs

type pair_t
A pair of *slong* indices a and b.

type pairs_struct

type pairs_t
A type holding a vector of *pair_t*.

void **pairs_init**(*pairs_t vec*)
Initializes *vec* for use, setting it to the empty vector of pairs.

void **pairs_fit_length**(*pairs_t vec*, *slong len*)
Allocates space for *len* elements in *vec*.

void **pairs_clear**(*pairs_t vec*)
Frees *vec*.

void **pairs_append**(*pairs_t vec*, *slong i*, *slong j*)
Appends the pair (i, j) to the end of *vec*.

void **pairs_insert_unique**(*pairs_t vec*, *slong i*, *slong j*)
Inserts (i, j) without duplication into *vec*. If this pair already exists, *vec* is unchanged. If this pair does not exist in *vec*, it is appended.

CHAPTER

NINE

PYTHON INTERFACE

9.1 Python interface (pycalcium / pyca)

Calcium includes a simple Python interface (`pycalcium`, or `pyca` for short) implemented using `ctypes`.

9.1.1 Introduction

Setup and usage

Make sure the Calcium library and its dependencies are built and in the path of the system's dynamic library loader. Then make sure that `<calcium_source_dir>/pycalcium` is in the Python path, for example by adding it to `PYTHONPATH`, adding it to `sys.path`, or simply starting Python inside the `pycalcium` directory.

Import the module and run a calculation:

```
>>> import pyca
>>> pyca.ca(1) / 3
0.333333 {1/3}
>>> pyca.exp(pyca.pi * pyca.i / 2)
1.00000*I {a where a = I [a^2+1=0]}
```

If you don't mind polluting the global namespace, import everything:

```
>>> from pyca import *
>>> exp(pi*i/2) + ca(1)/3
0.333333 + 1.00000*I {(3*a+1)/3 where a = I [a^2+1=0]}
```

Current limitations

- Leaks memory (for example, when printing).
- Because `ctypes` is used, this is not as efficient as a Cython wrapper. This interface should be used for testing and not for absolute performance.
- Does not wrap various functions.

9.1.2 API documentation

Functions are available both as regular functions and as methods on the `ca` class.

pyca.**fmpz_to_python_int**(*xref*)

pyca.**fmpq_set_python**(*cref*, *x*)

class pyca.**fexpr_struct**
Low-level wrapper for qqbar_struct, for internal use by ctypes.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

alloc
Structure/Union member

data
Structure/Union member

class pyca.**qqbar_struct**
Low-level wrapper for qqbar_struct, for internal use by ctypes.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

enclosure
Structure/Union member

poly
Structure/Union member

class pyca.**ca_struct**
Low-level wrapper for ca_struct, for internal use by ctypes.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

data
Structure/Union member

class pyca.**ca_ctx_struct**
Low-level wrapper for ca_ctx_struct, for internal use by ctypes.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

content
Structure/Union member

class pyca.**ca_mat_struct**
Low-level wrapper for ca_mat_struct, for internal use by ctypes.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

c
Structure/Union member

entries
Structure/Union member

r
Structure/Union member

rows
Structure/Union member

class pyca.**ca_vec_struct**
Low-level wrapper for ca_vec_struct, for internal use by ctypes.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

alloc
Structure/Union member

entries
Structure/Union member

length
Structure/Union member

class pyca.**ca_poly_struct**
Low-level wrapper for ca_poly_struct, for internal use by ctypes.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

alloc
Structure/Union member

coeffs
Structure/Union member

length
Structure/Union member

class pyca.**ca_poly_vec_struct**
Low-level wrapper for ca_poly_vec_struct, for internal use by ctypes.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

alloc
Structure/Union member

entries
Structure/Union member

length
Structure/Union member

class pyca.**fexpr**(*val=0*)

static inject(*vars=False*)
Inject all builtin symbol names into the calling namespace. For interactive use only!

```
>>> fexpr.inject()
>>> n = fexpr("n")
>>> Sum(Sin(Pi*n/3)/Factorial(n), For(n,0,Infinity))
Sum(Div(Sin(Div(Mul(Pi, n), 3)), Factorial(n)), For(n, 0, Infinity))
```

builtins()

__init__(*val=0*)
Initialize self. See help(type(self)) for accurate signature.

static from_param(*arg*)

latex()

nwords()

size_bytes()

allocated_bytes()

num_leaves()

depth()

is_atom()

is_atom_integer()

is_symbol()

head()

nargs()

args()

contains(*x*)
Check if *x* appears exactly as a subexpression in *self*.

```
>>> f = fexpr("f"); x = fexpr("x"); y = fexpr("y")
>>> (f(x+1).contains(f), f(x+1).contains(x), f(x+1).contains(y))
(True, True, False)
>>> (f(x+1).contains(1), f(x+1).contains(2))
(True, False)
>>> (f(x+1).contains(x+1), f(x+1).contains(f(x+1)))
(True, True)
```

replace(*old*, *new=None*)
Replace subexpression.

```
>>> f = fexpr("f"); x = fexpr("x"); y = fexpr("y")
>>> f(x+1, x-1).replace(x, y)
f(Add(y, 1), Sub(y, 1))
>>> f(x+1, x-1).replace(x+1, y-1)
f(Sub(y, 1), Sub(x, 1))
>>> f(x+1, x-1).replace(f, f+1)
Add(f, 1)(Add(x, 1), Sub(x, 1))
>>> f(x+1, x-1).replace(x+2, y)
f(Add(x, 1), Sub(x, 1))
```

__bool__()

expanded_normal_form()
Converts this expression to expanded normal form as a formal rational function of its non-arithmetic subexpressions.

```
>>> x = fexpr("x"); y = fexpr("y")
>>> (x / x**2).expanded_normal_form()
Div(1, x)
>>> (((x ** 0) + 3) ** 5).expanded_normal_form()
1024
>>> ((x+y+1)**3 - (y+1)**3 - (x+y)**3 - (x+1)**3).expanded_normal_form()
Add(Mul(-1, Pow(x, 3)), Mul(6, x, y), Mul(-1, Pow(y, 3)), -1)
>>> (1/((1/y + 1/x))).expanded_normal_form()
Div(Mul(x, y), Add(x, y))
>>> (((x+y)**5 * (x-y)) / (x**2 - y**2)).expanded_normal_form()
Add(Pow(x, 4), Mul(4, Pow(x, 3), y), Mul(6, Pow(x, 2), Pow(y, 2)), Mul(4, x, Pow(y,␣
↪3)), Pow(y, 4))
>>> (1 / (x - x)).expanded_normal_form()
Traceback (most recent call last):
  ...
ValueError: expanded_normal_form: overflow, formal division by zero or unsupported␣
↪expression
```

nstr(*n=16*)
Evaluates this expression numerically using Arb, returning a decimal string correct within 1 ulp in the last output digit. Attempts to obtain *n* digits (but the actual output accuracy may be lower).

```
>>> Exp = fexpr("Exp"); Exp(1).nstr()
'2.718281828459045'
>>> Pi = fexpr("Pi"); Pi.nstr(30)
'3.14159265358979323846264338328'
>>> Log = fexpr("Log"); Log(-2).nstr()
'0.6931471805599453 + 3.141592653589793*I'
>>> Im = fexpr("Im")
>>> Im(Log(2)).nstr()    # exact zero
'0'
```

Here the imaginary part is zero, but Arb is not able to compute so exactly. The output `0e-N` indicates only that the absolute value is bounded by `1e-N`:

```
>>> Exp(Log(-2)).nstr()
'-2.000000000000000 + 0e-22*I'
>>> Im(Exp(Log(-2))).nstr()
'0e-731'
```

The algorithm fails if the expression or any subexpression is not a finite complex number:

```
>>> Log(0).nstr()
Traceback (most recent call last):
  ...
ValueError: nstr: unable to evaluate to a number
```

Expressions must be constant:

```
>>> fexpr("x").nstr()
Traceback (most recent call last):
  ...
ValueError: nstr: unable to evaluate to a number
```

class pyca.**qqbar**(*val=0*)

Wrapper around the qqbar type, representing an algebraic number.

```
>>> (qqbar(2).sqrt() / qqbar(-2).sqrt()) ** 2
-1.00000 (deg 1)
>>> qqbar(0.5) == qqbar(1) / 2
True
>>> qqbar(0.1) == qqbar(1) / 10
False
>>> qqbar(3+4j)
3.00000 + 4.00000*I (deg 2)
>>> qqbar(3+4j).root(5)
1.35607 + 0.254419*I (deg 10)
>>> qqbar(3+4j).root(5) ** 5
3.00000 + 4.00000*I (deg 2)
```

The constructor can evaluate fexpr symbolic expressions provided that the expressions are constant and composed strictly of algebraic-valued basic operations applied to algebraic numbers.

```
>>> fexpr.inject()
>>> qqbar(Pow(0, 0))
1.00000 (deg 1)
>>> qqbar(Sqrt(2) * Abs(1+1j) + (+Re(3-4j)) + (-Im(5+6j)))
-1.00000 (deg 1)
>>> qqbar((Floor(Sqrt(1000)) + Ceil(Sqrt(1000)) + Sign(1+1j) / Sign(1-1j) + Csgn(1j) +␣
↪Conjugate(1j)) ** Div(-1, 3))
0.250000 (deg 1)
>>> [qqbar(RootOfUnity(3)), qqbar(RootOfUnity(3,2))]
[-0.500000 + 0.866025*I (deg 2), -0.500000 - 0.866025*I (deg 2)]
```

(continues on next page)

(continued from previous page)

```
>>> qqbar(Decimal("0.125")) == qqbar(125)/1000
True
>>> qqbar(Decimal("-2.7e5")) == -270000
True
```

__init__(*val=0*)
Initialize self. See help(type(self)) for accurate signature.

static from_param(*arg*)

__bool__()

re()

im()

conj()

conjugate()

floor()

ceil()

sgn()
The sign of this algebraic number.

```
>>> qqbar(-3).sgn()
-1.00000 (deg 1)
>>> qqbar(2+3j).sgn()
0.554700 + 0.832050*I (deg 4)
>>> qqbar(0).sgn()
0 (deg 1)
```

sign()
The sign of this algebraic number.

```
>>> qqbar(-3).sgn()
-1.00000 (deg 1)
>>> qqbar(2+3j).sgn()
0.554700 + 0.832050*I (deg 4)
>>> qqbar(0).sgn()
0 (deg 1)
```

sqrt()
Principal square root of this algebraic number.

```
>>> qqbar(-1).sqrt()
1.00000*I (deg 2)
>>> qqbar(-1).sqrt().sqrt()
0.707107 + 0.707107*I (deg 4)
```

root(*n*)
Principal nth root of this algebraic number.

```
>>> qqbar(3).root(1)
3.00000 (deg 1)
>>> qqbar(3).root(2)
1.73205 (deg 2)
>>> qqbar(3).root(3)
1.44225 (deg 3)
```

static polynomial_roots(*coeffs*)

Returns the roots of the polynomial defined by coeffs as a list. Roots are sorted in a fixed, canonical order.

At present, the implementation only allows integers (not algebraic numbers) as coefficients.

```
>>> qqbar.polynomial_roots([])
[]
>>> qqbar.polynomial_roots([0])
[]
>>> qqbar.polynomial_roots([1,2])
[-0.500000 (deg 1)]
>>> qqbar.polynomial_roots([3,2,1])
[-1.00000 + 1.41421*I (deg 2), -1.00000 - 1.41421*I (deg 2)]
>>> qqbar.polynomial_roots([1,2,1])
[-1.00000 (deg 1), -1.00000 (deg 1)]
```

conjugates()

Returns the algebraic conjugates of this algebraic number. The output is a list, with elements sorted in a fixed, canonical order.

```
>>> qqbar(0).conjugates()
[0 (deg 1)]
>>> ((qqbar(5).sqrt()+1)/2).conjugates()
[1.61803 (deg 2), -0.618034 (deg 2)]
>>> qqbar(2+3j).conjugates()
[2.00000 + 3.00000*I (deg 2), 2.00000 - 3.00000*I (deg 2)]
>>> qqbar.polynomial_roots([4,3,2,1])[0].conjugates()
[-1.65063 (deg 3), -0.174685 + 1.54687*I (deg 3), -0.174685 - 1.54687*I (deg 3)]
>>> qqbar.polynomial_roots([-35,0,315,0,-693,0,429])[0].conjugates()
[0.949108 (deg 6), 0.741531 (deg 6), 0.405845 (deg 6), -0.405845 (deg 6), -0.741531␣
↪(deg 6), -0.949108 (deg 6)]
```

minpoly()

Returns the minimal polynomial of self over the integers as a list of Python integers specifying the coefficients.

```
>>> qqbar(0).minpoly()
[0, 1]
>>> (qqbar(2) / 3).minpoly()
[-2, 3]
>>> qqbar(2).sqrt().minpoly()
[-2, 0, 1]
>>> ((qqbar(2).sqrt() + 1).root(3) + 1).minpoly()
[2, -12, 21, -22, 15, -6, 1]
>>> qqbar(0.5).minpoly()
[-1, 2]
>>> qqbar(0.1).minpoly()
[-3602879701896397, 36028797018963968]
>>> (qqbar(1) / 10).minpoly()
[-1, 10]
```

is_real()

Check if this algebraic number is a real number.

```
>>> qqbar(2).sqrt().is_real()
True
>>> qqbar(-2).sqrt().is_real()
False
```

is_rational()

Check if this algebraic number is a rational number.

```
>>> (qqbar(-5) / 7).is_rational()
True
>>> (qqbar(-5) / 7).sqrt().is_rational()
False
```

is_integer()
Check if this algebraic number is an integer.

```
>>> qqbar(3).is_integer()
True
>>> (qqbar(3) / 5).is_integer()
False
```

degree()
The degree of this algebraic number (the degree of the minimal polynomial).

```
>>> qqbar(5).degree()
1
>>> qqbar(5).sqrt().degree()
2
```

p()
Assuming that self is a rational number, returns the numerator as a Python integer.

```
>>> (qqbar(-2)/3).p()
-2
>>> qqbar(-1).sqrt().p()
Traceback (most recent call last):
  ...
ValueError: self must be a rational number
```

q()
Assuming that self is a rational number, returns the denominator as a Python integer.

```
>>> (qqbar(-2)/3).q()
3
>>> qqbar(-1).sqrt().q()
Traceback (most recent call last):
  ...
ValueError: self must be a rational number
```

fexpr(*formula=True*, *root_index=False*, *serialized=False*, *gaussians=True*, *quadratics=True*, *cyclotomics=True*, *cubics=True*, *quartics=True*, *quintics=True*, *depression=True*, *deflation=True*, *separation=True*)

fexpr_repr()

class pyca.ca_ctx(***kwargs*)
Python class wrapping the ca_ctx_t context object. Currently only supports a global instance.

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

static from_param(*arg*)

class pyca.ca(*val=0*, *context=None*)
Python class wrapping the ca_t type for numbers.

Examples:

```
>>> ca(1)
1
>>> ca()
```

(continues on next page)

(continued from previous page)

```
0
>>> ca(0)
0
>>> ca(-5)
-5
>>> ca(2.25)
2.25000 {9/4}
>>> ca(1) / 3
0.333333 {1/3}
>>> (-1) ** ca(0.5)
1.00000*I {a where a = I [a^2+1=0]}
>>> ca(1-2j)
1.00000 - 2.00000*I {-2*a+1 where a = I [a^2+1=0]}
>>> ca(0.1)           # be careful with float input!
0.100000 {3602879701896397/36028797018963968}
>>> ca(float("inf"))
+Infinity
>>> ca(float("nan"))
Unknown
>>> 3 < pi < ca(22)/7
True
>>> ca(qqbar(200).sqrt())
14.1421 {10*a where a = 1.41421 [a^2-2=0]}
```

__init__(*val=0*, *context=None*)
Initialize self. See help(type(self)) for accurate signature.

static from_param(*arg*)

static inf()
The special value positive infinity.

Examples:

```
>>> inf == ca.inf()    # alias for the method
True
>>> inf
+Infinity
>>> -inf
-Infinity
>>> abs(-inf)
+Infinity
>>> inf + inf
+Infinity
>>> (-inf) + (-inf)
-Infinity
>>> -inf * inf
-Infinity
>>> inf / inf
Undefined
>>> 1 / inf
0
>>> re(inf)
Undefined
>>> im(inf)
Undefined
>>> sign(inf)
1
>>> sign((1+i)*inf) == (1+i)/sqrt(2)
True
```

static uinf()
The special value unsigned infinity.

Examples:

```
>>> uinf == ca.uinf()    # alias for the method
True
>>> uinf
UnsignedInfinity
>>> abs(uinf)
+Infinity
>>> -3 * uinf
UnsignedInfinity
>>> 1/uinf
0
>>> sign(uinf)
Undefined
>>> uinf * uinf
UnsignedInfinity
>>> uinf / uinf
Undefined
>>> uinf + uinf
Undefined
>>> re(uinf)
Undefined
>>> im(uinf)
Undefined
```

static undefined()

The special value undefined.

Examples:

```
>>> undefined == ca.undefined()    # alias for the method
True
>>> undefined
Undefined
>>> undefined == undefined
True
>>> undefined * 0
Undefined
```

static unknown()

The special meta-value unknown. This meta-value is not comparable.

Examples:

```
>>> unknown == unknown
Traceback (most recent call last):
  ...
NotImplementedError: unable to decide predicate: equal
>>> unknown == 0
Traceback (most recent call last):
  ...
NotImplementedError: unable to decide predicate: equal
>>> unknown == undefined
Traceback (most recent call last):
  ...
NotImplementedError: unable to decide predicate: equal
>>> unknown + unknown
Unknown
>>> unknown + 3
Unknown
>>> unknown + undefined
Undefined
```

static pi()

The constant pi.

Examples:

```
>>> pi == ca.pi()    # alias for the method
True
>>> pi
3.14159 {a where a = 3.14159 [Pi]}
>>> sin(pi/6)
0.500000 {1/2}
>>> (pi - 3) ** 3
0.00283872 {a^3-9*a^2+27*a-27 where a = 3.14159 [Pi]}
```

static euler()

Euler's constant (gamma).

Examples:

```
>>> euler == ca.euler()    # alias for the method
True
>>> euler
0.577216 {a where a = 0.577216 [Euler]}
>>> exp(euler)
1.78107 {a where a = 1.78107 [Exp(0.577216 {b})], b = 0.577216 [Euler]}
```

static i()

The imaginary unit.

Examples:

```
>>> i == ca.i()    # alias for the method
True
>>> i == I == j    # extra aliases for convenience
True
>>> i
1.00000*I {a where a = I [a^2+1=0]}
>>> i**2
-1
>>> i**3
-1.00000*I {-a where a = I [a^2+1=0]}
>>> abs(i)
1
>>> sign(i)
1.00000*I {a where a = I [a^2+1=0]}
>>> abs(sqrt(1+i) / sqrt(1-i))
1
>>> re(i), im(i)
(0, 1)
```

__bool__()

static operands_with_same_context(*a*, *b*)

pow_arithmetic(*n*)

nstr(*n=16*, *parts=False*)

Evaluates this expression numerically using Arb, returning a decimal string correct within 1 ulp in the last output digit. Attempts to obtain n digits (but the actual output accuracy may be lower).

Examples:

```
>>> ca(0).nstr()
'0'
```

(continues on next page)

(continued from previous page)

```
>>> pi.nstr(30)
'3.14159265358979323846264338328'
```

By default, the result is considered accurate when the larger of the real and imaginary parts is known to n digits. If *parts* is set, the algorithm will attempt to compute both real and imaginary parts accurately. It will, in particular, attempt to recognize when a real or imaginary part is exactly zero:

```
>>> (log(1+i) + log(1-i)).nstr()
'0.6931471805599453 + 0e-25*I'
>>> (log(1+i) + log(1-i)).nstr(parts=True)
'0.6931471805599453'
```

An exception is raised if the numerical evaluation code fails to produce a finite enclosure:

```
>>> (1 / ca(0)).nstr()
Traceback (most recent call last):
  ...
ValueError: nstr: unable to evaluate to a number
```

rewrite_cnf(*deep=True*)

re()

Real part.

Examples:

```
>>> re(2+3j) == ca(2+3j).re()
True
>>> re(2+3*i)
2
```

im()

Imaginary part.

Examples:

```
>>> im(2+3j) == ca(2+3j).im()    # alias for the method
True
>>> im(2+3*i)
3
```

conj()

Complex conjugate.

Examples:

```
>>> conj(1j) == conjugate(1j) == ca(1j).conj() == ca(1j).conjugate()    # alias for␣
↪the method
True
>>> conj(2+i)
2.00000 - 1.00000*I {-a+2 where a = I [a^2+1=0]}
>>> conj(pi)
3.14159 {a where a = 3.14159 [Pi]}
```

conjugate()

Complex conjugate.

Examples:

```
>>> conj(1j) == conjugate(1j) == ca(1j).conj() == ca(1j).conjugate()    # alias for␣
↪the method
True
>>> conj(2+i)
2.00000 - 1.00000*I {-a+2 where a = I [a^2+1=0]}
>>> conj(pi)
3.14159 {a where a = 3.14159 [Pi]}
```

floor()

Floor function.

Examples:

```
>>> floor(3) == ca(3).floor()    # alias for the method
True
>>> floor(pi)
3
>>> floor(-pi)
-4
```

ceil()

Ceiling function.

Examples:

```
>>> ceil(3) == ca(3).ceil()    # alias for the method
True
>>> ceil(pi)
4
>>> ceil(-pi)
-3
```

sgn()

Sign function.

Examples:

```
>>> sgn(2) == sign(2) == ca(2).sgn()    # aliases for the method
True
>>> sign(0)
0
>>> sign(sqrt(2))
1
>>> sign(-sqrt(2))
-1
>>> sign(-sqrt(-2))
-1.00000*I {-a where a = I [a^2+1=0]}
```

sign()

Sign function.

Examples:

```
>>> sgn(2) == sign(2) == ca(2).sgn()    # aliases for the method
True
>>> sign(0)
0
>>> sign(sqrt(2))
1
>>> sign(-sqrt(2))
-1
>>> sign(-sqrt(-2))
-1.00000*I {-a where a = I [a^2+1=0]}
```

csgn()

Real-valued complex extension of real sign function.

Examples:

```
>>> csgn(3)
1
>>> csgn(3*i)
1
>>> csgn(-3*i)
-1
>>> csgn(-3)
-1
>>> csgn(0)
0
>>> csgn(1+i)
1
>>> csgn((1+i)*inf)
1
>>> csgn(-i*inf)
-1
>>> csgn(uinf)
Undefined
```

arg()

Complex argument (phase).

Examples:

```
>>> arg(2) == arg(0) == ca(0).arg() == 0
True
>>> arg(-10)
-3.14159 {-a where a = 3.14159 [Pi]}
>>> arg(sqrt(-3))
1.57080 {(a)/2 where a = 3.14159 [Pi]}
>>> arg(-sqrt(sqrt(-3)))
-2.35619 {(-3*a)/4 where a = 3.14159 [Pi]}
```

sqrt()

Principal square root.

Examples:

```
>>> sqrt(2) == ca(2).sqrt()    # alias for the method
True
>>> sqrt(0)
0
>>> sqrt(1)
1
>>> sqrt(2)
1.41421 {a where a = 1.41421 [a^2-2=0]}
>>> sqrt(-1)
1.00000*I {a where a = I [a^2+1=0]}
>>> sqrt(inf)
+Infinity
>>> sqrt(-inf)
+I * Infinity
>>> sqrt(uinf)
UnsignedInfinity
>>> sqrt(undefined)
Undefined
>>> sqrt(unknown)
Unknown
```

log()
Natural logarithm.

Examples:

```
>>> log(2) == ca(2).log()    # alias for the method
True
>>> log(1)
0
>>> log(exp(2))
2
>>> log(2)
0.693147 {a where a = 0.693147 [Log(2)]}
>>> log(4) == 2*log(2)
True
>>> log(1+sqrt(2)) / log(3+2*sqrt(2))
0.500000 {1/2}
>>> log(ca(10)**(10**30)) / log(10)
1.00000e+30 {1000000000000000000000000000000}
>>> log(-1)
3.14159*I {a*b where a = 3.14159 [Pi], b = I [b^2+1=0]}
>>> log(i)
1.57080*I {(a*b)/2 where a = 3.14159 [Pi], b = I [b^2+1=0]}
>>> log(0)
-Infinity
>>> log(inf)
+Infinity
>>> log(-inf)
+Infinity
>>> log(uinf)
+Infinity
>>> log(undefined)
Undefined
>>> log(unknown)
Unknown
```

exp()
Exponential function.

Examples:

```
>>> exp(0)
1
>>> exp(1)
2.71828 {a where a = 2.71828 [Exp(1)]}
>>> exp(-1)
0.367879 {a where a = 0.367879 [Exp(-1)]}
>>> exp(-1) * exp(1)
1
>>> exp(7*pi*i/2)
-1.00000*I {-a where a = I [a^2+1=0]}
>>> exp(inf)
+Infinity
>>> exp(-inf)
0
>>> exp(uinf)
Undefined
>>> exp(undefined)
Undefined
>>> exp(unknown)
Unknown
```

sin(*form=None*)
Sine function.

Examples:

```
>>> sin(0)
0
>>> sin(pi)
0
>>> sin(pi/2)
1
>>> sin(pi/6)
0.500000 {1/2}
>>> sin(sqrt(2))**2 + cos(sqrt(2))**2
1
>>> sin(3 + pi) + sin(3)
0
>>> sin(1, form="exponential")
0.841471 - 0e-24*I {(-a^2*b+b)/(2*a) where a = 0.540302 + 0.841471*I [Exp(1.00000*I
↪{b})], b = I [b^2+1=0]}
>>> sin(1, form="direct")
0.841471 {a where a = 0.841471 [Sin(1)]}
>>> sin(1, form="tangent")
0.841471 {(2*a)/(a^2+1) where a = 0.546302 [Tan(0.500000 {1/2})]}
>>> sin(inf)
Undefined
>>> sin(i * inf)
+I * Infinity
>>> sin(-i * inf)
-I * Infinity
```

cos(*form=None*)

Cosine function.

Examples:

```
>>> cos(0)
1
>>> cos(pi)
-1
>>> cos(pi/2)
0
>>> cos(pi/3)
0.500000 {1/2}
>>> cos(pi/6)**2
0.750000 {3/4}
>>> cos(1)**2 + sin(1)**2
1
>>> cos(1, form="exponential")
0.540302 - 0e-24*I {(a^2+1)/(2*a) where a = 0.540302 + 0.841471*I [Exp(1.00000*I {b}
↪)], b = I [b^2+1=0]}
>>> cos(1, form="direct")
0.540302 {a where a = 0.540302 [Cos(1)]}
>>> cos(1, form="tangent")
0.540302 {(-a^2+1)/(a^2+1) where a = 0.546302 [Tan(0.500000 {1/2})]}
>>> cos(i * inf)
+Infinity
>>> cos(-i * inf)
+Infinity
>>> cos(inf)
Undefined
```

tan(*form=None*)

Tangent function.

Examples:

```
>>> tan(0)
0
>>> tan(pi)
0
>>> tan(pi/2)
UnsignedInfinity
>>> tan(pi/4)
1
>>> tan(3*pi/4)
-1
>>> tan(pi/3)**2
3
>>> tan(1 + pi) - tan(1)
0
>>> tan(1, form="direct")
1.55741 {a where a = 1.55741 [Tan(1)]}
>>> tan(1, form="sine_cosine")
1.55741 {(b)/(a) where a = 0.540302 [Cos(1)], b = 0.841471 [Sin(1)]}
>>> tan(1, form="exponential")
1.55741 + 0e-23*I {(-a^2*b+b)/(a^2+1) where a = 0.540302 + 0.841471*I [Exp(1.00000*I
↪{b})], b = I [b^2+1=0]}
>>> tan(inf)
Undefined
>>> tan(i * inf)
1.00000*I {a where a = I [a^2+1=0]}
>>> tan(-i * inf)
-1.00000*I {-a where a = I [a^2+1=0]}
```

atan(*form=None*)

Inverse tangent function.

Examples:

```
>>> atan(0)
0
>>> atan(1)
0.785398 {(a)/4 where a = 3.14159 [Pi]}
>>> atan(1-sqrt(2))
-0.392699 {(-a)/8 where a = 3.14159 [Pi]}
>>> atan(inf)
1.57080 {(a)/2 where a = 3.14159 [Pi]}
>>> atan(-inf)
-1.57080 {(-a)/2 where a = 3.14159 [Pi]}
>>> atan(i*inf)
1.57080 {(a)/2 where a = 3.14159 [Pi]}
>>> atan(-i*inf)
-1.57080 {(-a)/2 where a = 3.14159 [Pi]}
>>> atan((1+i)*inf)
1.57080 {(a)/2 where a = 3.14159 [Pi]}
>>> atan(tan(23*pi/27)) == -4*pi/27
True
>>> atan(2, form="direct")
1.10715 {a where a = 1.10715 [Atan(2)]}
>>> atan(2, form="logarithm")
1.10715 - 0e-24*I {(a*b)/2 where a = 0e-24 - 2.21430*I [Log(-0.600000 - 0.800000*I
↪{(-4*b-3)/5})], b = I [b^2+1=0]}
```

asin(*form=None*)

Inverse sine function.

Examples:

```
>>> asin(0)
0
>>> asin(1)
1.57080 {(a)/2 where a = 3.14159 [Pi]}
>>> asin(-1)
-1.57080 {(-a)/2 where a = 3.14159 [Pi]}
>>> asin(sqrt(2)/2)
0.785398 {(a)/4 where a = 3.14159 [Pi]}
>>> asin(i)
0.881374*I {-a*c where a = -0.881374 [Log(0.414214 {b-1})], b = 1.41421 [b^2-2=0], c␣
↪= I [c^2+1=0]}
>>> sin(asin(sqrt(2)-1)) == sqrt(2)-1
True
>>> asin(sin(sqrt(2)-1)) == sqrt(2)-1
True
>>> asin(sqrt(2)-1, form="logarithm")
0.427079 + 0e-24*I {-a*d where a = 0e-24 + 0.427079*I [Log(0.910180 + 0.414214*I
↪{b+c*d-d})], b = 0.910180 [Sqrt(0.828427 {2*c-2})], c = 1.41421 [c^2-2=0], d = I␣
↪[d^2+1=0]}
>>> asin(sqrt(2)-1, form="direct")
0.427079 {a where a = 0.427079 [Asin(0.414214 {b-1})], b = 1.41421 [b^2-2=0]}
>>> asin(inf)
-I * Infinity
>>> asin(-inf)
+I * Infinity
>>> asin(inf*i)
+I * Infinity
>>> asin(-inf*i)
-I * Infinity
>>> asin(uinf)
UnsignedInfinity
>>> asin(undefined)
Undefined
```

acos(*form=None*)

Inverse cosine function.

Examples:

```
>>> acos(0)
1.57080 {(a)/2 where a = 3.14159 [Pi]}
>>> acos(1)
0
>>> acos(-1)
3.14159 {a where a = 3.14159 [Pi]}
>>> acos(sqrt(2)/2)
0.785398 {(a)/4 where a = 3.14159 [Pi]}
>>> acos(i)
1.57080 - 0.881374*I {(2*a*d+b)/2 where a = -0.881374 [Log(0.414214 {c-1})], b = 3.
↪14159 [Pi], c = 1.41421 [c^2-2=0], d = I [d^2+1=0]}
>>> cos(acos(sqrt(2) - 1)) == sqrt(2) - 1
True
>>> acos(inf)
+I * Infinity
>>> acos(-inf)
-I * Infinity
>>> acos(inf*i)
-I * Infinity
>>> acos(-inf*i)
+I * Infinity
>>> acos(uinf)
UnsignedInfinity
```

(continues on next page)

(continued from previous page)

```
>>> acos(undefined)
Undefined
```

erf()

Error function.

Examples:

```
>>> erf(0)
0
>>> erf(1)
0.842701 {a where a = 0.842701 [Erf(1)]}
>>> erf(inf)
1
>>> erf(-inf)
-1
>>> erf(i*inf)
+I * Infinity
>>> erf(-i*inf)
-I * Infinity
>>> erf(uinf)
Undefined
```

erfc()

Complementary error function.

Examples:

```
>>> erfc(inf)
0
>>> erfc(-inf)
2
>>> erfc(1000)
1.86004e-434298 {a where a = 1.86004e-434298 [Erfc(1000)]}
>>> erfc(i*inf)
-I * Infinity
>>> erfc(-i*inf)
+I * Infinity
>>> erfc(sqrt(2)) + erf(sqrt(2))
1
>>> erfc(uinf)
Undefined
```

erfi()

Imaginary error function.

Examples:

```
>>> erfi(0)
0
>>> erfi(1)
1.65043 {a where a = 1.65043 [Erfi(1)]}
>>> erfi(inf)
+Infinity
>>> erfi(-inf)
-Infinity
>>> erfi(i*inf)
1.00000*I {a where a = I [a^2+1=0]}
>>> erfi(-i*inf)
-1.00000*I {-a where a = I [a^2+1=0]}
>>> erf(2)**2 + erfi(i*2)**2
0
```

gamma()
Gamma function.

Examples:

```
>>> [gamma(n) for n in range(1,11)]
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
>>> gamma(0)
UnsignedInfinity
>>> 1 / gamma(0)
0
>>> gamma(0.5)
1.77245 {a where a = 1.77245 [Sqrt(3.14159 {b})], b = 3.14159 [Pi]}
>>> gamma(0.5)**2 == pi
True
>>> pi * gamma(pi) / gamma(pi+1)
1
```

class pyca.**ca_mat**(**args*, *context=None*)
Python class wrapping the ca_mat_t type for matrices.

Examples:

```
>>> ca_mat(2,3)
ca_mat of size 2 x 3
[0, 0, 0]
[0, 0, 0]
>>> ca_mat([[1,2],[3,4],[5,6]])
ca_mat of size 3 x 2
[1, 2]
[3, 4]
[5, 6]
>>> ca_mat(2, 5, range(10))
ca_mat of size 2 x 5
[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9]
>>> ca_mat([[1,-2],[2,1]]) * ca_mat([[1,pi],[1,2]])
ca_mat of size 2 x 2
[-1, -0.858407 {a-4 where a = 3.14159 [Pi]}]
[ 3, 8.28319 {2*a+2 where a = 3.14159 [Pi]}]
```

A nontrivial calculation:

```
>>> H = ca_mat([[ca(1)/(i+j+1) for i in range(5)] for j in range(5)])
>>> H
ca_mat of size 5 x 5
[            1, 0.500000 {1/2}, 0.333333 {1/3}, 0.250000 {1/4}, 0.200000 {1/5}]
[0.500000 {1/2}, 0.333333 {1/3}, 0.250000 {1/4}, 0.200000 {1/5}, 0.166667 {1/6}]
[0.333333 {1/3}, 0.250000 {1/4}, 0.200000 {1/5}, 0.166667 {1/6}, 0.142857 {1/7}]
[0.250000 {1/4}, 0.200000 {1/5}, 0.166667 {1/6}, 0.142857 {1/7}, 0.125000 {1/8}]
[0.200000 {1/5}, 0.166667 {1/6}, 0.142857 {1/7}, 0.125000 {1/8}, 0.111111 {1/9}]
>>> H.trace()
1.78730 {563/315}
>>> sum(c*multiplicity for (c, multiplicity) in H.eigenvalues())
1.78730 {563/315}
>>> H.det()
3.74930e-12 {1/266716800000}
>>> prod(c**multiplicity for (c, multiplicity) in H.eigenvalues())
3.74930e-12 {1/266716800000}
```

__init__(**args*, *context=None*)
Initialize self. See help(type(self)) for accurate signature.

static from_param(*arg*)

__bool__()

static operands_with_same_context(*a*, *b*)

nrows()

ncols()

entries()

table()

tolist()

trace()
The trace of this matrix.

Examples:

```
>>> ca_mat([[1,2],[3,pi]]).trace()
4.14159 {a+1 where a = 3.14159 [Pi]}
```

det(*algorithm=None*)
The determinant of this matrix.

Examples:

```
>>> ca_mat([[1,1-i*pi],[1+i*pi,1]]).det()
-9.86960 {-a^2 where a = 3.14159 [Pi], b = I [b^2+1=0]}
```

conjugate()
Entrywise complex conjugate.

```
>>> ca_mat([[5,1-i]]).conjugate()
ca_mat of size 1 x 2
[5, 1.00000 + 1.00000*I {a+1 where a = I [a^2+1=0]}]
```

conj()
Entrywise complex conjugate.

```
>>> ca_mat([[5,1-i]]).conjugate()
ca_mat of size 1 x 2
[5, 1.00000 + 1.00000*I {a+1 where a = I [a^2+1=0]}]
```

transpose()
Matrix transpose.

```
>>> ca_mat([[5,1-i]]).transpose()
ca_mat of size 2 x 1
[                                                5]
[1.00000 - 1.00000*I {-a+1 where a = I [a^2+1=0]}]
```

conjugate_transpose()
Conjugate matrix transpose.

```
>>> ca_mat([[5,1-i]]).conjugate_transpose()
ca_mat of size 2 x 1
[                                               5]
[1.00000 + 1.00000*I {a+1 where a = I [a^2+1=0]}]
```

conj_transpose()
Conjugate matrix transpose.

```
>>> ca_mat([[5,1-i]]).conjugate_transpose()
ca_mat of size 2 x 1
[                                                5]
[1.00000 + 1.00000*I {a+1 where a = I [a^2+1=0]}]
```

charpoly()
Characteristic polynomial of this matrix.

```
>>> ca_mat([[5,pi],[1,-1]]).charpoly()
ca_poly of length 3
[-8.14159 {-a-5 where a = 3.14159 [Pi]}, -4, 1]
```

eigenvalues()
Eigenvalues of this matrix. Returns a list of (value, multiplicity) pairs.

```
>>> ca_mat(4, 4, range(16)).eigenvalues()
[(32.4642 {a+15 where a = 17.4642 [a^2-305=0]}, 1), (-2.46425 {-a+15 where a = 17.
↪4642 [a^2-305=0]}, 1), (0, 2)]
>>> ca_mat([[1,pi],[-pi,1]]).eigenvalues()[0]
(1.00000 + 3.14159*I {a*b+1 where a = 3.14159 [Pi], b = I [b^2+1=0]}, 1)
```

rref()
Reduced row echelon form.

```
>>> ca_mat([[1,2,3],[4,5,6],[7,8,9]]).rref()
ca_mat of size 3 x 3
[1, 0, -1]
[0, 1,  2]
[0, 0,  0]
>>> ca_mat([[1,pi,2,pi],[1/pi,3,1/(pi+1),4],[1,1,1,1]]).rref()
ca_mat of size 3 x 4
[1, 0, 0, 0.401081 {(a^3-a^2-2*a)/(3*a^2+3*a-2) where a = 3.14159 [Pi]}]
[0, 1, 0,  1.35134 {(4*a^2+4*a-2)/(3*a^2+3*a-2) where a = 3.14159 [Pi]}]
[0, 0, 1,     -0.752416 {(-a^3+a)/(3*a^2+3*a-2) where a = 3.14159 [Pi]}]
>>> ca_mat([[1, 0, 0], [0, 1-exp(ca(2)**-10000), 0]]).rref()
Traceback (most recent call last):
  ...
NotImplementedError: failed to compute rref
```

rank()
Rank of this matrix.

```
>>> ca_mat([[1,2,3],[4,5,6],[7,8,9]]).rank()
2
>>> ca_mat([[1, 0, 0], [0, 1-exp(ca(2)**-10000), 0]]).rank()
Traceback (most recent call last):
  ...
NotImplementedError: failed to compute rank
```

inv()
Matrix inverse.

```
>>> ca_mat([[1,1],[0,1/pi]]).inv()
ca_mat of size 2 x 2
[1, -3.14159 {-a where a = 3.14159 [Pi]}]
[0,   3.14159 {a where a = 3.14159 [Pi]}]
>>> ca_mat([[1, 1], [2, 2]]).inv()
Traceback (most recent call last):
  ...
ZeroDivisionError: singular matrix
>>> ca_mat([[1, 0], [0, 1-exp(ca(2)**-10000)]]).inv()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  ...
NotImplementedError: failed to prove matrix singular or nonsingular
```

solve(*other*, *algorithm=None*)

Solve linear system (with a nonsingular matrix).

```
>>> ca_mat([[1,2],[3,4]]).solve(ca_mat([[5],[6]]))
ca_mat of size 2 x 1
[           -4]
[4.50000 {9/2}]
>>> ca_mat([[1,1],[2,2]]).solve(ca_mat([[5],[6]]))
Traceback (most recent call last):
  ...
ZeroDivisionError: singular matrix
>>> ca_mat([[1, 0], [0, 1-exp(ca(2)**-10000)]]).solve(ca_mat([[5],[6]]))
Traceback (most recent call last):
  ...
NotImplementedError: failed to prove matrix singular or nonsingular
```

right_kernel()

Returns a basis of the right kernel (nullspace) of *self*.

```
>>> A = ca_mat([[3,4,6],[5,6,7]])
>>> X = A.right_kernel()
>>> X
ca_mat of size 3 x 1
[              4]
[-4.50000 {-9/2}]
[              1]
>>> A * X
ca_mat of size 2 x 1
[0]
[0]
```

diagonalization()

Matrix diagonalization: given a square matrix *self*, returns a diagonal matrix D and an invertible matrix P such that *self* equals PDP^{-1}. Raises *ValueError* if *self* is not diagonalizable.

```
>>> A = ca_mat([[1,2],[3,4]])
>>> D, P = A.diagonalization()
>>> D
ca_mat of size 2 x 2
[5.37228 {(a+5)/2 where a = 5.74456 [a^2-33=0]},                                 ␣
↪            0]
[                                                0, -0.372281 {(-a+5)/2 where a = 5.
↪74456 [a^2-33=0]}]
>>> P * D * P.inv()
ca_mat of size 2 x 2
[1, 2]
[3, 4]
```

A diagonalizable matrix without distinct eigenvalues:

```
>>> A = ca_mat([[-1,3,-1],[-3,5,-1],[-3,3,1]])
>>> D, P = A.diagonalization()
>>> D
ca_mat of size 3 x 3
[1, 0, 0]
[0, 2, 0]
[0, 0, 2]
```

(continues on next page)

(continued from previous page)

```
>>> P
ca_mat of size 3 x 3
[1, 1, -0.333333 {-1/3}]
[1, 1,                0]
[1, 0,                1]
>>> P * D * P.inv() == A
True
```

`log()`

Matrix logarithm.

```
>>> ca_mat([[4,2],[2,4]]).log().det() == log(2)*(log(2)+log(3))
True
>>> ca_mat([[1,1],[0,1]]).log()
ca_mat of size 2 x 2
[0, 1]
[0, 0]
>>> ca_mat([[0,1],[0,0]]).log()
Traceback (most recent call last):
  ...
ZeroDivisionError: singular matrix
>>> ca_mat([[0,0,1],[0,1,0],[1,0,0]]).log() / (pi*I)
ca_mat of size 3 x 3
[  0.500000 {1/2}, 0, -0.500000 {-1/2}]
[               0, 0,                0]
[-0.500000 {-1/2}, 0,  0.500000 {1/2}]
>>> ca_mat([[0,0,1],[0,1,0],[1,0,0]]).log().exp()
ca_mat of size 3 x 3
[0, 0, 1]
[0, 1, 0]
[1, 0, 0]
```

`exp()`

Matrix exponential.

```
>>> ca_mat([[1,2],[0,3]]).exp()
ca_mat of size 2 x 2
[2.71828 {a where a = 2.71828 [Exp(1)]}, 17.3673 {b^3-b where a = 20.0855 [Exp(3)],␣
↪b = 2.71828 [Exp(1)]}]
[                                    0,                              20.0855 {a where␣
↪a = 20.0855 [Exp(3)]}]
>>> ca_mat([[1,2],[3,4]]).exp()[0,0]
51.9690 {(-a*c+11*a+b*c+11*b)/22 where a = 215.354 [Exp(5.37228 {(c+5)/2})], b = 0.
↪689160 [Exp(-0.372281 {(-c+5)/2})], c = 5.74456 [c^2-33=0]}
>>> ca_mat([[0,0,1],[1,0,0],[0,1,0]]).exp().det()
1
>>> ca_mat([[0,1,0,0,0],[0,0,2,0,0],[0,0,0,3,0],[0,0,0,0,4],[0,0,0,0,0]]).exp()
ca_mat of size 5 x 5
[1, 1, 1, 1, 1]
[0, 1, 2, 3, 4]
[0, 0, 1, 3, 6]
[0, 0, 0, 1, 4]
[0, 0, 0, 0, 1]
```

This example currently fails (due to failure to compute the exact Jordan decomposition internally):

```
>>> ca_mat([[0,0,1],[0,2,0],[-1,0,0]]).log().exp()
Traceback (most recent call last):
  ...
NotImplementedError: unable to compute matrix exponential
```

jordan_form(*transform=False*)

Jordan decomposiion: given a square matrix *self*, returns a block diagonal matrix J composed of Jordan blocks and optionally an invertible matrix P such that *self* equals PJP^{-1}.

```
>>> A = ca_mat([[20,77,59,40], [0,-2,-3,-2], [-10,-35,-23,-15], [2,7,3,1]])
>>> J, P = A.jordan_form(transform=True)
>>> P * J * P.inv()
ca_mat of size 4 x 4
[ 20,  77,  59,  40]
[  0,  -2,  -3,  -2]
[-10, -35, -23, -15]
[  2,   7,   3,   1]
>>> A = ca_mat([[log(2), log(3)], [log(4), log(5)]])
>>> J, P = A.jordan_form(transform=True)
>>> J[0,0]
2.46769 {(a+b+e)/2 where a = 2.63279 [Sqrt(6.93159 {b^2-2*b*e+8*d*e+e^2})], b = 1.
↪60944 [Log(5)], c = 1.38629 [Log(4)], d = 1.09861 [Log(3)], e = 0.693147 [Log(2)]}
>>> P * J * P.inv() == A
True
```

class pyca.**ca_vec**(*n=0*, *context=None*)

Python class wrapping the ca_vec_t type for vectors.

__init__(*n=0*, *context=None*)

Initialize self. See help(type(self)) for accurate signature.

static from_param(*arg*)

class pyca.**ca_poly_vec**(*n=0*, *context=None*)

__init__(*n=0*, *context=None*)

Initialize self. See help(type(self)) for accurate signature.

static from_param(*arg*)

class pyca.**ca_poly**(*val=0*, *context=None*)

Python class wrapping the ca_poly_t type for polynomials.

__init__(*val=0*, *context=None*)

Initialize self. See help(type(self)) for accurate signature.

static from_param(*arg*)

__bool__()

static operands_with_same_context(*a*, *b*)

mul_series(*other*, *n*)

div_series(*other*, *n*)

inv_series(*n*)

Power series inverse truncated to length n.

Examples:

```
>>> ca_poly([1,2,3]).inv_series(10)
ca_poly of length 10
[1, -2, 1, 4, -11, 10, 13, -56, 73, 22]
>>> ca_poly([sqrt(2), 1, 2]).inv_series(5).inv_series(5)
ca_poly of length 3
[1.41421 {a where a = 1.41421 [a^2-2=0]}, 1, 2]
>>> ca_poly([]).inv_series(3)
ca_poly of length 3
[UnsignedInfinity, Undefined, Undefined]
>>> ca_poly([0,1,2]).inv_series(3)
```

(continues on next page)

(continued from previous page)

```
ca_poly of length 3
[UnsignedInfinity, Undefined, Undefined]
>>> ca_poly([inf,1,2]).inv_series(3)
ca_poly of length 3
[Undefined, Undefined, Undefined]
```

log_series(*n*)

Power series logarithm truncated to length n.

Examples:

```
>>> ca_poly([1,1]).log_series(4)
ca_poly of length 4
[0, 1, -0.500000 {-1/2}, 0.333333 {1/3}]
>>> ca_poly([2,1]).log_series(4)
ca_poly of length 4
[0.693147 {a where a = 0.693147 [Log(2)]}, 0.500000 {1/2}, -0.125000 {-1/8}, 0.
↪0416667 {1/24}]
>>> ca_poly([-2,2,3]).log_series(5).exp_series(5)
ca_poly of length 3
[-2, 2, 3]
>>> ca_poly([0,1]).log_series(3)
ca_poly of length 3
[-Infinity, Undefined, Undefined]
>>> ca_poly([inf,1]).log_series(3)
ca_poly of length 3
[Undefined, Undefined, Undefined]
```

exp_series(*n*)

Power series exponential truncated to length n.

Examples:

```
>>> ca_poly([0,1]).exp_series(4)
ca_poly of length 4
[1, 1, 0.500000 {1/2}, 0.166667 {1/6}]
>>> ca_poly([1,-0.5]).exp_series(2)
ca_poly of length 2
[2.71828 {a where a = 2.71828 [Exp(1)]}, -1.35914 {(-a)/2 where a = 2.71828 [Exp(1)]}
↪]
>>> ca_poly([inf]).exp_series(3)
ca_poly of length 3
[Undefined, Undefined, Undefined]
>>> ca_poly([unknown]).exp_series(3)
ca_poly of length 3
[Unknown, Unknown, Unknown]
```

atan_series(*n*)

Power series inverse tangent truncated to length n.

```
>>> ca_poly([0,1]).atan_series(6)
ca_poly of length 6
[0, 1, 0, -0.333333 {-1/3}, 0, 0.200000 {1/5}]
>>> ca_poly([1,1]).atan_series(4)
ca_poly of length 4
[0.785398 {(a)/4 where a = 3.14159 [Pi]}, 0.500000 {1/2}, -0.250000 {-1/4}, 0.
↪0833333 {1/12}]
>>> f = ca_poly([2,3,4])
>>> 2*f.atan_series(5) - ((2*f).div_series(1-f**2, 5)).atan_series(5) == pi
True
>>> ca_poly([0]).atan_series(3)
```

(continues on next page)

(continued from previous page)

```
ca_poly of length 0
[]
>>> ca_poly([i,1]).atan_series(3)
ca_poly of length 3
[+I * Infinity, Undefined, Undefined]
>>> ca_poly([-i,1]).atan_series(3)
ca_poly of length 3
[-I * Infinity, Undefined, Undefined]
>>> ca_poly([unknown,1]).atan_series(3)
ca_poly of length 3
[Unknown, Unknown, Unknown]
```

gcd(*other*)

Polynomial GCD.

Examples:

```
>>> x = ca_poly([0,1]); (x+1).gcd(x-1)
ca_poly of length 1
[1]
>>> x = ca_poly([0,1]); (x**2 + pi**2).gcd(x+i*pi)
ca_poly of length 2
[3.14159*I {a*b where a = 3.14159 [Pi], b = I [b^2+1=0]}, 1]
```

roots()

Roots of this polynomial. Returns a list of (root, multiplicity) pairs.

```
>>> ca_poly([2,11,20,12]).roots()
[(-0.666667 {-2/3}, 1), (-0.500000 {-1/2}, 2)]
```

factor_squarefree()

Squarefree factorization of this polynomial Returns (lc, L) where L is a list of (factor, multiplicity) pairs.

```
>>> ca_poly([9,6,7,-28,12]).factor_squarefree()
(12, [(ca_poly of length 3
[0.333333 {1/3}, 0.666667 {2/3}, 1], 1), (ca_poly of length 2
[-1.50000 {-3/2}, 1], 2)])
```

squarefree_part()

Squarefree part of this polynomial.

```
>>> ca_poly([9,6,7,-28,12]).squarefree_part()
ca_poly of length 4
[-0.500000 {-1/2}, -0.666667 {-2/3}, -0.833333 {-5/6}, 1]
```

integral()

Integral of this polynomial.

```
>>> ca_poly([1,1,1,1]).integral()
ca_poly of length 5
[0, 1, 0.500000 {1/2}, 0.333333 {1/3}, 0.250000 {1/4}]
```

derivative()

Derivative of this polynomial.

```
>>> ca_poly([1,1,1,1]).derivative()
ca_poly of length 3
[1, 2, 3]
```

monic()

Make this polynomial monic.

```
>>> ca_poly([1,2,3]).monic()
ca_poly of length 3
[0.333333 {1/3}, 0.666667 {2/3}, 1]
>>> ca_poly().monic()
Traceback (most recent call last):
  ...
ValueError: failed to make monic
```

is_proper()

Checks if this polynomial definitely has finite coefficients and that the leading coefficient is provably nonzero.

```
>>> ca_poly([]).is_proper()
True
>>> ca_poly([1,2,3]).is_proper()
True
>>> ca_poly([1,2,1-exp(ca(2)**-10000)]).is_proper()
False
>>> ca_poly([inf]).is_proper()
False
```

degree()

Degree of this polynomial.

```
>>> ca_poly([1,2,3]).degree()
2
>>> ca_poly().degree()
-1
>>> ca_poly([1,2,1-exp(ca(2)**-10000)]).degree()
Traceback (most recent call last):
  ...
NotImplementedError: unable to determine degree
```

pyca.**re**(x)

pyca.**im**(x)

pyca.**sgn**(x)

pyca.**sign**(x)

pyca.**csgn**(x)

pyca.**arg**(x)

pyca.**floor**(x)

pyca.**ceil**(x)

pyca.**conj**(x)

pyca.**conjugate**(x)

pyca.**sqrt**(x)

pyca.**log**(x)

pyca.**exp**(x)

pyca.**erf**(x)

pyca.**erfc**(x)

pyca.**erfi**(x)

pyca.**gamma**(x)

pyca.**fac**(*x*)
Alias for gamma(x+1).

Examples:

```
>>> fac(10)
3.62880e+6 {3628800}
```

pyca.**cos**(*x*, *form=None*)

pyca.**sin**(*x*, *form=None*)

pyca.**tan**(*x*, *form=None*)

pyca.**atan**(*x*, *form=None*)

pyca.**asin**(*x*, *form=None*)

pyca.**acos**(*x*, *form=None*)

pyca.**cosh**(*x*)
The hyperbolic cosine function is not yet implemented in Calcium. This placeholder function evaluates the hyperbolic cosine function using exponentials.

Examples:

```
>>> cosh(1)
1.54308 {(a^2+1)/(2*a) where a = 2.71828 [Exp(1)]}
```

pyca.**sinh**(*x*)
The hyperbolic sine function is not yet implemented in Calcium. This placeholder function evaluates the hyperbolic sine function using exponentials.

Examples:

```
>>> sinh(1)
1.17520 {(a^2-1)/(2*a) where a = 2.71828 [Exp(1)]}
```

pyca.**tanh**(*x*)
The hyperbolic tangent function is not yet implemented in Calcium. This placeholder function evaluates the hyperbolic tangent function using exponentials.

Examples:

```
>>> tanh(1)
0.761594 {(a^2-1)/(a^2+1) where a = 2.71828 [Exp(1)]}
```

pyca.**prod**(*s*)

pyca.**gd**(*x*)

pyca.**test_floor_ceil**()

pyca.**test_power_identities**()

pyca.**test_log**()

pyca.**test_exp**()

pyca.**test_erf**()

pyca.**test_gudermannian**()

pyca.**test_gamma**()

pyca.**test_conversions**()

pyca.**test_notebook_examples**()

pyca.**test_qqbar_misc**()

pyca.**test_context_switch**()

pyca.**test_improved_zero_recognition**()

pyca.**test_trigonometric**()

pyca.**test_comparisons**()

pyca.**test_xfail**()

pyca.**test_latex**()

pyca.**latex_test_report**()

CHAPTER

TEN

CREDITS AND REFERENCES

10.1 Bibliography

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

All referenced works: [BBK2014], [BF2020], [BFSS2006], [Boe2020], [Car2004], [Cho1999], [Coh1996], [Coh2000], [Fie2007], [GCL1992], [Har2010], [Har2015], [Har2018], [Joh2017], [JR1999], [Mos1971], [MP2006], [RF1994], [Ric1992], [Ric1995], [Ric1997], [Ric2007], [Ric2009], [Ste2002], [Ste2010], [Str1997], [Str2012], [vdH1995], [vdH2006], [vHP2012], [Zip1985].

CHAPTER

ELEVEN

VERSION HISTORY

11.1 History and changes

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

Old releases of the code can be accessed from https://github.com/fredrik-johansson/calcium/releases

11.1.1 2021-05-28 - version 0.4

- Algebraic numbers
  - Fixed bug in special-casing of roots of unity in qqbar_root_ui.
  - Fixed qqbar_randtest with bits == 1.
  - Faster qqbar_cmp_re for nearby reals.
  - Faster qqbar polynomial evaluation and powering using linear algebra.
  - Improved qqbar_abs, qqbar_abs2 to produce cleaner enclosures.
  - Use a slightly better method to detect real numbers in qqbar_sgn_im.
  - Added qqbar_hash.
  - Added qqbar_get_fmpq, qqbar_get_fmpz.
  - Added qqbar_pow_fmpq, qqbar_pow_fmpz, qqbar_pow_si.
  - Added qqbar_numerator, qqbar_denominator.
- Basic arithmetic and elementary functions
  - Improved ca_condense_field: automatically demote to a simple number field when the only used extension number is algebraic.
  - Improved multivariate field arithmetic to automatically remove algebraic or redundant monomial factors from denominators.
  - Added ca_pow_si_arithmetic (guaranteed in-field exponentiation).
  - Added polynomial evaluation functions (ca_fmpz_poly_evaluate, ca_fmpq_poly_evaluate, ca_fmpz_poly_evaluate, ca_fmpz_mpoly_q_evaluate).
  - Added several helper functions (ca_is_special, ca_is_qq_elem, ca_is_qq_elem_zero, ca_is_qq_elem_one, ca_is_qq_elem_integer, ca_is_nf_elem, ca_is_cyclotomic_nf_elem, ca_is_generic_elem).
  - Added ca_rewrite_complex_normal_form.
  - Perform direct complex conjugation in cyclotomic fields.

  - Use ca_get_acb_raw instead of ca_get_acb when printing to avoid expensive recomputations.
  - Added alternative algorithms for various basic functions.
  - Deep complex conjugation.
  - Use complex conjugation in is_real, is_imaginary, is_negative_real.
  - Added functions for unsafe inversion for internal use.
  - Significantly stronger zero testing in mixed algebraic-transcendental fields.
  - Added ca_arg.
  - Added special case for testing equality between number field elements and rationals.
  - Added ca_sin_cos, ca_sin, ca_cos, ca_tan and variants.
  - Added ca_atan, ca_asin, ca_acos and variants.
  - Added ca_csgn.
  - Improved ca_get_acb and ca_get_acb_accurate_parts to fall back on exact zero tests when direct numerical evaluation does not give precise enclosures.
  - Added ca_get_decimal_str.
  - More automatic simplifications of logarithms (simplify logarithms of exponentials, square roots and powers raised to integer powers).
  - More automatic simplifications of square roots (simplify square roots of exponentials, square roots and powers raised to integer powers).
  - Improved order comparisons (ca_check_ge etc.) to handle special values and to fall back on strong equality tests.
  - Fixed a test failure in the ca_mat module.

- Polynomials
  - Added ca_poly_inv_series, ca_poly_div_series (power series division).
  - Added ca_poly_exp_series (power series exponential).
  - Added ca_poly_log_series (power series logarithm).
  - Added ca_poly_atan_series (power series arctangent).

- Other
  - Added fmpz_mpoly_q_used_vars.
  - Remove useless rpath line from configure (reported by Julien Puydt).
  - Added missing declaration of fexpr_hash.
  - Fixed crashes on OS X in Python interface (contributed by deinst).
  - Fixed memory leaks in Python string conversions (contributed by deinst).
  - Reserve I, E for symbolic expressions in Python interface.

11.1.2 2021-04-23 - version 0.3

- Symbolic expressions
  - Added the fexpr module for flat-packed unevaluated symbolic expressions.
  - LaTeX output.
  - Basic manipulation (construction, replacement, accessing subexpressions).
  - Numerical evaluation with Arb.
  - Expanded normal form.
  - Conversion methods for other types.
  - Enable LaTeX rendering of objects in Jupyter notebooks.
- Algebraic numbers
  - Fix a major performance issue (slow root refinement) that made Calcium as a whole far slower than necessary.
  - Added qqbar_cmp_root_order; sort polynomial roots consistently by default.
  - Added qqbar_get_quadratic.
  - Added qqbar_equal_fmpq_poly_val and use it to speed up checking guessed values.
  - Conversion of qqbar_t to and from symbolic expression (qqbar_set_fexpr, qqbar_get_fexpr_repr, qqbar_get_fexpr_root_nearest, qqbar_get_fexpr_root_indexed, qqbar_get_fexpr_formula).
  - Fixed bugs in qqbar_cmpabs_re, cmpabs_im.
  - Optimized qqbar_cmp_im and qqbar_cmpabs_im for conjugates with mirror symmetry.
  - Added qqbar_pow (taking a qqbar exponent).
  - Special-case roots of unity in qqbar_pow_ui, qqbar_root_ui, qqbar_abs and qqbar_abs2.
  - Wrapped qqbar in Python.
- Polynomials
  - Added several utility functions.
  - Optimized polynomial multiplication with rational entries.
  - Fast polynomial multiplication over number fields.
- Matrices
  - Fast matrix multiplication over number fields.
  - Right kernel (ca_mat_right_kernel).
  - Matrix diagonalization (ca_mat_diagonalization).
  - Jordan normal form (ca_mat_jordan_form, ca_mat_jordan_transformation, ca_mat_jordan_blocks).
  - Matrix exponential (ca_mat_exp).
  - Matrix logarithm (ca_mat_log).
  - Polynomial evaluation (ca_mat_ca_poly_evaluate).
  - Cofactor expansion algorithm for determinant and adjugate (ca_mat_adjugate_cofactor).
  - Added several utility functions.
  - Improved algorithm selection in ca_mat_inv.
  - Solving using the adjugate matrix.

  - Danilevsky characteristic polynomial algorithm (ca_mat_charpoly_danilevsky).
- Field elements
  - Use factoring in ca_sqrt to enable more simplifications.
  - Simplify square roots and logarithms of negative real numbers.
  - Optimized ca_sub.
  - Conversion of ca_t to and from symbolic expressions (ca_set_fexpr, ca_get_fexpr).
  - Added function for assigning elements between context objects (ca_transfer).
  - Fixed a possible memory corruption bug when Vieta's formulas are used.
  - Optimized constructing square roots of rational numbers.
- Other
  - Added demonstration notebook to documentation.
  - Fixed OSX compatibility in Python wrapper (contributed by deinst).
  - Fixed bug in calcium_write_acb.
  - Fixed bug in fmpz_mpoly_vec_set_primitive_unique (contributed by gbunting).

11.1.3 2020-10-16 - version 0.2

- Basic arithmetic and expression simplification
  - Use Gröbner basis for reduction ideals, making simplification much more robust.
  - Compute all linear relations with LLL simultaneously instead of piecemeal.
  - Make monomial ordering configurable (default is lex as before).
  - Use Vieta's formulas to simplify expressions involving conjugate algebraic numbers.
  - Denest exponentials of symbolic logarithms.
  - Denest logarithms of symbolic powers and square roots.
  - Denest powers of symbolic powers.
  - Simplify exponentials that evaluate to roots of unity.
  - Simplify logarithms of roots of unity.
  - Improve ideal reduction to avoid some unnecessary GCD computations.
- Python wrapper
  - Calcium now includes a minimal ctypes-based Python wrapper for testing.
- New ca_mat module for matrices
  - Mostly using naive basecase algorithms.
  - Matrix arithmetic, basic manipulation.
  - Construction of special matrices (Hilbert, Pascal, Stirling, DFT).
  - LU factorization.
  - Fraction-free LU decomposition.
  - Nonsingular solving and inverse.
  - Reduced row echelon form.
  - Rank.
  - Trace and determinant.

  - Characteristic polynomial.
  - Computation of eigenvalues with multiplicities.
- New ca_poly module for polynomials
  - Mostly using naive basecase algorithms.
  - Polynomial arithmetic, basic manipulation.
  - Polynomial division.
  - Evaluation and composition.
  - Derivative and integral.
  - GCD (Euclidean algorithm).
  - Squarefree factorization.
  - Computation of roots with multiplicities.
  - Construction from given roots.
- New ca_vec module for vectors.
  - Memory management and basic scalar operations.
- Bug fixes
  - Fix bug in powering number field elements.
  - Fix bug in qqbar_log_pi_i.
  - Fix aliasing bug in ca_pow.
- New basic functions
  - Conversion from double: ca_set_d, ca_set_d_d.
  - Special functions: ca_erf, ca_erfi, ca_erfc, with algebraic relations.
  - Special functions: ca_gamma (incomplete simplification algorithms).
- New utils_flint module for Flint utilities
  - Vectors of multivariate polynomials.
  - Construction of elementary symmetric polynomials.
  - Gröbner basis computation (naive Buchberger algorithm).
- Documentation and presentation
  - Various improvements to the documentation.
  - DFT example program.

11.1.4 2020-09-08 - version 0.1

- Initial test release.
- ca module (exact real and complex numbers).
- fmpz_mpoly_q module (multivariate rational functions over Q).
- qqbar module (algebraic numbers represented by minimal polynomials).
- Example programs.

BIBLIOGRAPHY

[BBK2014] D. H. Bailey, J. M. Borwein and A. D. Kaiser. "Automated simplification of large symbolic expressions". Journal of Symbolic Computation Volume 60, January 2014, Pages 120-136. https://doi.org/10.1016/j.jsc.2013.09.001

[BF2020] F. Beukers and J. Forsgård. "Gamma-evaluations of hypergeometric series". Preprint, 2020. https://arxiv.org/abs/2004.08117

[BFSS2006] A. Bostan, P. Flajolet, B. Salvy and É. Schost. "Fast computation of special resultants". Journal of Symbolic Computation, 41(1):1–29, January 2006. https://doi.org/10.1016/j.jsc.2005.07.001

[Boe2020] H. Boehm. "Towards an API for the real numbers". PLDI 2020: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2020, Pages 562-576. https://doi.org/10.1145/3385412.3386037

[Car2004] J. Carette. "Understanding expression simplification." ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation, pp. 72-79. 2004. https://doi.org/10.1145/1005285.1005298

[Cho1999] T. Chow. "What is a closed-form number?". The American Mathematical Monthly Volume 106, 1999 - Issue 5. https://doi.org/10.1080/00029890.1999.12005066

[Coh1996] H. Cohen. *A course in computational algebraic number theory*. Third edition, Springer, 1996. https://doi.org/10.1007/978-3-662-02945-9

[Coh2000] H. Cohen. *Advanced topics in computational number theory*. Springer, 2000. https://doi.org/10.1007/978-1-4419-8489-0

[Fie2007] C. Fieker, "Sparse representation for cyclotomic fields". Experiment. Math. Volume 16, Issue 4 (2007), 493-500. https://doi.org/10.1080/10586458.2007.10129012

[GCL1992] K. O. Geddes, S. R. Czapor and G. Labahn. *Algorithms for computer algebra*. Springer, 1992. https://doi.org/10.1007/b102438

[Har2010] W. B. Hart. "Fast library for number theory: an introduction." International Congress on Mathematical Software. Springer, Berlin, Heidelberg, 2010. https://doi.org/10.1007/978-3-642-15582-6_18

[Har2015] W. B. Hart. "ANTIC: Algebraic number theory in C". Computeralgebra-Rundbrief: Vol. 56, 2015

[Har2018] W. B. Hart. "Algebraic number theory". Unpublished manuscript, 2018.

[Joh2017] F. Johansson. "Arb: efficient arbitrary-precision midpoint-radius interval arithmetic". IEEE Transactions on Computers, vol 66, issue 8, 2017, pp. 1281-1292. https://doi.org/10.1109/TC.2017.2690633

[JR1999] D. Jeffrey and A. D. Rich. "Simplifying square roots of square roots by denesting". Computer Algebra Systems: A Practical Guide, M.J. Wester, Ed., Wiley 1999.

[MP2006] M. Monagan and R. Pearce. “Rational simplification modulo a polynomial ideal”. Proceedings of the 2006 international symposium on Symbolic and algebraic computation - ISSAC ‘06. https://doi.org/10.1145/1145768.1145809

[Mos1971] J. Moses. “Algebraic simplification - a guide for the perplexed”. Proceedings of the second ACM symposium on Symbolic and algebraic manipulation (1971), 282-304. https://doi.org/10.1145/362637.362648

[RF1994] D. Richardson and J. Fitch. “The identity problem for elementary functions and constants”. ISSAC ‘94: Proceedings of the international symposium on Symbolic and algebraic computation, August 1994, 285-290. https://doi.org/10.1145/190347.190429

[Ric1992] D. Richardson. “The elementary constant problem”. ISSAC ‘92: Papers from the international symposium on Symbolic and algebraic computation, August 1992, 108-116. https://doi.org/10.1145/143242.143284

[Ric1995] D. Richardson. “A simplified method of recognizing zero among elementary constants”. ISSAC ‘95: Proceedings of the 1995 international symposium on Symbolic and algebraic computation, April 1995, 104-109. https://doi.org/10.1145/220346.220360

[Ric1997] D. Richardson. “How to recognize zero”. Journal of Symbolic Computation 24.6 (1997): 627-645. https://doi.org/10.1006/jsco.1997.0157

[Ric2007] D. Richardson. “Zero tests for constants in simple scientific computation”. Mathematics in Computer Science volume 1, pages 21-37 (2007). https://doi.org/10.1007/s11786-007-0002-x

[Ric2009] D. Richardson. “Recognising zero among implicitly defined elementary numbers”. Preprint, 2009.

[Ste2002] A. Steel. “A new scheme for computing with algebraically closed fields”. In: Fieker C., Kohel D.R. (eds) Algorithmic Number Theory. ANTS 2002. Lecture Notes in Computer Science, vol 2369. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45455-1_38

[Ste2010] A. Steel. “Computing with algebraically closed fields”. Journal of Symbolic Computation 45 (2010) 342-372. https://doi.org/10.1016/j.jsc.2009.09.005

[Str1997] A. Strzebonski. “Computing in the field of complex algebraic numbers”. Journal of Symbolic Computation (1997) 24, 647-656. https://doi.org/10.1006/jsco.1997.0158

[Str2012] A. Strzebonski. “Real root isolation for exp-log-arctan functions”. Journal of Symbolic Computation 47 (2012) 282–314. https://doi.org/10.1016/j.jsc.2011.11.004

[vHP2012] M. van Hoeij and V. Pal. “Isomorphisms of algebraic number fields”. Journal de Théorie des Nombres de Bordeaux, Vol. 24, No. 2 (2012), pp. 293-305. https://doi.org/10.2307/43973105

[vdH1995] J. van der Hoeven, “Automatic numerical expansions”. Proc. of the conference Real numbers and computers (1995), 261-274. https://www.texmacs.org/joris/ane/ane-abs.html

[vdH2006] J. van der Hoeven, “Computations with effective real numbers”. Theoretical Computer Science, Volume 351, Issue 1, 14 February 2006, Pages 52-60. https://doi.org/10.1016/j.tcs.2005.09.060

[Zip1985] R. Zippel. “Simplification of expressions involving radicals”. Journal of Symbolic Computation (1985) 1, 189-210. https://doi.org/10.1016/S0747-7171(85)80014-6

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

Q

R

S

T

U

W

X

Z