User’s Guide

to

the PARI library

(version 2.7.7)

The PARI Group

Institut de Mathématiques de Bordeaux, UMR 5251 du CNRS.
Université Bordeaux 1, 351 Cours de la Libération
F-33405 TALENCE Cedex, FRANCE
e-mail: pari@math.u-bordeaux.fr

Home Page:
http://pari.math.u-bordeaux.fr/
Chapter 4: Programming PARI in Library Mode

4.1 Introduction: initializations, universal objects

4.2 Important technical notes
  4.2.1 Backward compatibility
  4.2.2 Types
  4.2.3 Type recursivity
  4.2.4 Variations on basic functions
  4.2.5 Portability: 32-bit / 64-bit architectures
  4.2.6 Using malloc / free

4.3 Garbage collection
  4.3.1 Why and how
  4.3.2 Variants
  4.3.3 Examples
  4.3.4 Comments

4.4 Creation of PARI objects, assignments, conversions
  4.4.1 Creation of PARI objects
  4.4.2 Sizes
  4.4.3 Assignments
  4.4.4 Copy
  4.4.5 Clones
  4.4.6 Conversions

4.5 Implementation of the PARI types
  4.5.1 Type \texttt{t\_INT} (integer)
  4.5.2 Type \texttt{t\_REAL} (real number)
  4.5.3 Type \texttt{t\_INTMOD}
  4.5.4 Type \texttt{t\_FRAC} (rational number)
  4.5.5 Type \texttt{t\_FFELT} (finite field element)
  4.5.6 Type \texttt{t\_COMPLEX} (complex number)
  4.5.7 Type \texttt{t\_PADIC} (\(p\)-adic numbers)
  4.5.8 Type \texttt{t\_QUAD} (quadratic number)
  4.5.9 Type \texttt{t\_POLMOD} (polmod)
  4.5.10 Type \texttt{t\_POL} (polynomial)
  4.5.11 Type \texttt{t\_SER} (power series)
  4.5.12 Type \texttt{t\_RFRAC} (rational function)
  4.5.13 Type \texttt{t\_QFR} (indefinite binary quadratic form)
  4.5.14 Type \texttt{t\_QFI} (definite binary quadratic form)
  4.5.15 Type \texttt{t\_VEC} and \texttt{t\_COL} (vector)
  4.5.16 Type \texttt{t\_MAT} (matrix)
  4.5.17 Type \texttt{t\_VECSMALL} (vector of small integers)
  4.5.18 Type \texttt{t\_STR} (character string)
  4.5.19 Type \texttt{t\_ERROR} (error context)
  4.5.20 Type \texttt{t\_CLOSURE} (closure)
  4.5.21 Type \texttt{t\_LIST} (list)

4.6 PARI variables
  4.6.1 Multivariate objects
  4.6.2 Creating variables
4.7 Input and output
4.7.1 Input
4.7.2 Output to screen or file, output to string
4.7.3 Errors
4.7.4 Warnings
4.7.5 Debugging output
4.7.6 Timers and timing output
4.8 Iterators, Numerical integration, Sums, Products
4.8.1 Iterators
4.8.2 Iterating over primes
4.8.3 Numerical analysis
4.9 Catching exceptions
4.9.1 Basic use
4.9.2 Advanced use
4.10 A complete program

Chapter 5: Technical Reference Guide: the basics
5.1 Initializing the library
5.1.1 General purpose
5.1.2 Technical functions
5.1.3 Notions specific to the GP interpreter
5.1.4 Public callbacks
5.1.5 Configuration variables
5.1.6 Saving and restoring the GP context
5.1.7 GP history
5.2 Handling GENs
5.2.1 Allocation
5.2.2 Length conversions
5.2.3 Read type-dependent information
5.2.4 Eval type-dependent information
5.2.5 Set type-dependent information
5.2.6 Type groups
5.2.7 Accessors and components
5.3 Global numerical constants
5.3.1 Constants related to word size
5.3.2 Masks used to implement the GEN type
5.3.3 \log_2 \pi
5.4 Iterating over small primes, low-level interface
5.5 Handling the PARI stack
5.5.1 Allocating memory on the stack
5.5.2 Stack-independent binary objects
5.5.3 Garbage collection
5.5.4 Garbage collection: advanced use
5.5.5 Debugging the PARI stack
5.5.6 Copies
5.5.7 Simplify
5.6 The PARI heap
5.6.1 Introduction
5.6.2 Public interface
5.6.3 Implementation note
7.2.7 FpXQ, Fq ................................. 102
7.2.8 FpXQ .................................. 104
7.2.9 Fq .................................. 104
7.2.10 FpX, FpXY .......................... 106
7.2.11 FpXq, FqX .......................... 106
7.2.12 FLx .................................. 110
7.2.13 FLxV .................................. 113
7.2.14 FLxT .................................. 113
7.2.15 FLxq .................................. 113
7.2.16 FLxX .................................. 114
7.2.17 FLxqX .................................. 115
7.2.18 FLxqXQ .................................. 116
7.2.19 F2x .................................. 116
7.2.20 F2xq .................................. 118
7.2.21 F2xqV, F2xqM .......................... 118
7.2.22 Functions returning objects with t_INTMOD coefficients .......... 119
7.2.23 Chinese remainder theorem over \( \mathbb{Z} \) .................. 120
7.2.24 Rational reconstruction .................. 121
7.2.25 Hensel lifts .......................... 121
7.2.26 Other \( p \)-adic functions ................. 123
7.2.27 Conversions involving single precision objects .............. 124
7.3 Higher arithmetic over \( \mathbb{Z} \): primes, factorization ............ 127
7.3.1 Pure powers .................................. 127
7.3.2 Factorization .................................. 128
7.3.3 Checks associated to arithmetic functions .................. 129
7.3.4 Incremental integer factorization .................. 130
7.3.5 Integer core, squarefree factorization .................. 131
7.3.6 Primes, primality and compositeness tests ................. 131
7.3.7 Iterators over primes .......................... 132
7.4 Integral, rational and generic linear algebra .................. 133
7.4.1 ZC / ZV, ZM .................................. 133
7.4.2 ZV / zm .................................. 135
7.4.3 ZMV / zmV (vectors of ZM/zm) ................. 136
7.4.4 RgC / RgV, RgM .......................... 136
7.4.5 Blackbox linear algebra .................. 140
7.4.6 Obsolete functions .......................... 141
7.5 Integral, rational and generic polynomial arithmetic ......... 141
7.5.1 ZX .................................. 141
7.5.2 ZXQ .................................. 143
7.5.3 ZXV .................................. 144
7.5.4 ZXT .................................. 144
7.5.5 ZXX .................................. 144
7.5.6 QX .................................. 144
7.5.7 QXQ .................................. 145
7.5.8 ZX .................................. 146
7.5.9 RgX .................................. 146
Chapter 8: Operations on general PARI objects .................. 153
8.1 Assignment .................................. 153
8.2 Conversions .................................. 153
10.2.3 PARI colors ........................................... 187
10.2.4 Obsolete output functions ............................. 188

10.3 Files ...................................................... 188
10.3.1 pariFILE ............................................. 189
10.3.2 Temporary files ...................................... 189

10.4 Errors ...................................................... 189
10.4.1 Internal errors, "system" errors ......................... 190
10.4.2 Syntax errors, type errors .............................. 190
10.4.3 Overflows ............................................. 192
10.4.4 Errors triggered intentionally ......................... 193
10.4.5 Mathematical errors .................................. 193
10.4.6 Miscellaneous functions ............................... 194

10.5 Hashtables .................................................. 194

10.6 Dynamic arrays ........................................... 196
10.6.1 Initialization .......................................... 196
10.6.2 Adding elements ...................................... 196
10.6.3 Accessing elements ................................... 196
10.6.4 Stack of stacks ....................................... 196
10.6.5 Public interface ....................................... 197

10.7 Vectors and Matrices ..................................... 197
10.7.1 Access and extract .................................... 197
10.7.2 Componentwise operations ............................. 199
10.7.3 Low-level vectors and columns functions ............. 199

10.8 Vectors of small integers .................................. 200
10.8.1 t_VECSMALL .......................................... 200
10.8.2 Vectors of t_VECSMALL ................................ 201

Chapter 11: Functions related to the GP interpreter ................. 203
11.1 Handling closures ........................................ 203
11.1.1 Functions to evaluate t_CLOSURE ..................... 203
11.1.2 Functions to handle control flow changes ............. 204
11.1.3 Functions to deal with lexical local variables ........ 204
11.1.4 Functions returning new closures ..................... 204
11.1.5 Functions used by the gp debugger (break loop) ....... 205
11.1.6 Standard wrappers for iterators ....................... 205

11.2 Defaults .................................................. 205

11.3 Records and Lazy vectors .................................. 208

12.1 General Number Fields ................................... 209
12.1.1 Number field types ................................... 209
12.1.2 Extracting info from a nf structure ................. 210
12.1.3 Extracting info from a bnf structure ................ 211
12.1.4 Extracting info from a bnf structure ................ 212
12.1.5 Extracting info from an rnf structure ............... 212
12.1.6 Extracting info from a bid structure ................. 213
12.1.7 Increasing accuracy .................................. 213
12.1.8 Number field arithmetic .............................. 214
12.1.9 Elements in factored form ............................ 216
12.1.10 Ideal arithmetic ..................................... 217
12.1.11 Maximal ideals ...................................... 219
Chapter 4:  
Programming PARI in Library Mode

The User’s Guide to Pari/GP gives in three chapters a general presentation of the system, of the gp calculator, and detailed explanation of high level PARI routines available through the calculator. The present manual assumes general familiarity with the contents of these chapters and the basics of ANSI C programming, and focuses on the usage of the PARI library. In this chapter, we introduce the general concepts of PARI programming and describe useful general purpose functions; the following chapters describes all public low or high-level functions, underlying or extending the GP functions seen in Chapter 3 of the User’s guide.

4.1 Introduction: initializations, universal objects.

To use PARI in library mode, you must write a C program and link it to the PARI library. See the installation guide or the Appendix to the User’s Guide to Pari/GP on how to create and install the library and include files. A sample Makefile is presented in Appendix A, and a more elaborate one in examples/Makefile. The best way to understand how programming is done is to work through a complete example. We will write such a program in Section 4.10. Before doing this, a few explanations are in order.

First, one must explain to the outside world what kind of objects and routines we are going to use. This is done* with the directive

```
#include <pari/pari.h>
```

In particular, this defines the fundamental type for all PARI objects: the type GEN, which is simply a pointer to long.

Before any PARI routine is called, one must initialize the system, and in particular the PARI stack which is both a scratchboard and a repository for computed objects. This is done with a call to the function

```
void pari_init(size_t size, ulong maxprime)
```

The first argument is the number of bytes given to PARI to work with, and the second is the upper limit on a precomputed prime number table; size should not reasonably be taken below 500000 but you may set maxprime = 0, although the system still needs to precompute all primes up to about $2^{16}$. For lower-level variants allowing finer control, e.g. preventing PARI from installing its own error or signal handlers, see Section 5.1.2.

We have now at our disposal:

- a PARI stack containing nothing. This is a big connected chunk of size bytes of memory, where all computations take place. In large computations, intermediate results quickly clutter up memory so some kind of garbage collecting is needed. Most systems do garbage collecting when the memory is getting scarce, and this slows down the performance. PARI takes a different approach,

* This assumes that PARI headers are installed in a directory which belongs to your compiler’s search path for header files. You might need to add flags like -I/usr/local/include or modify C_INCLUDE_PATH.
admittedly more demanding on the programmer: you must do your own cleaning up when the intermediate results are not needed anymore. We will see later how (and when) this is done.

- the following universal objects (by definition, objects which do not belong to the stack): the integers 0, 1, −1, 2 and −2 (respectively called gen_0, gen_1, gen_m1, gen_2 and gen_m2), the fraction $\frac{1}{2}$ (ghalf). All of these are of type GEN.
- a heap which is just a linked list of permanent universal objects. For now, it contains exactly the ones listed above. You will probably very rarely use the heap yourself; and if so, only as a collection of copies of objects taken from the stack (called clones in the sequel). Thus you need not bother with its internal structure, which may change as PARI evolves. Some complex PARI functions create clones for special garbage collecting purposes, usually destroying them when returning.
- a table of primes (in fact of differences between consecutive primes), called diffptr, of type byteptr (pointer to unsigned char). Its use is described in Section 5.4 later. Using it directly is deprecated, high-level iterators provide a cleaner and more flexible interface, see Section 4.8.2 (such iterators use the private prime table, but extend it dynamically).
- access to all the built-in functions of the PARI library. These are declared to the outside world when you include pari.h, but need the above things to function properly. So if you forget the call to pari_init, you will get a fatal error when running your program.

### 4.2 Important technical notes

#### 4.2.1 Backward compatibility

The PARI function names evolved over time, and deprecated functions are eventually deleted. The file pariold.h contains macros implementing a weak form of backward compatibility. In particular, whenever the name of a documented function changes, a define is added to this file so that the old name expands to the new one (provided the prototype didn’t change also).

This file is included by pari.h, but a large section is commented out by default. Define PAR Old NAMES before including pari.h to pollute your namespace with lots of obsolete names like un*: that might enable you to compile old programs without having to modify them. The preferred way to do that is to add -DPAR Old NAMES to your compiler CFLAGS, so that you don’t need to modify the program files themselves.

Of course, it’s better to fix the program if you can!

#### 4.2.2 Types

Although PARI objects all have the C type GEN, we will freely use the word type to refer to PARI dynamic subtypes: t_INT, t_REAL, etc. The declaration

```c
GEN x;
```

declares a C variable of type GEN, but its “value” will be said to have type t_INT, t_REAL, etc. The meaning should always be clear from the context.

* For (long)gen1. Since 2004 and version 2.2.9, typecasts are completely unnecessary in PARI programs.
4.2.3 Type recursivity.

Conceptually, most PARI types are recursive. But the GEN type is a pointer to long, not to GEN. So special macros must be used to access GEN’s components. The simplest one is gel(V, i), where el stands for element, to access component number i of the GEN V. This is a valid lvalue (may be put on the left side of an assignment), and the following two constructions are exceedingly frequent

\[
gel(V, i) = x; \\
x = gel(V, i);
\]

where x and V are GENs. This macro accesses and modifies directly the components of V and do not create a copy of the coefficient, contrary to all the library functions.

More generally, to retrieve the values of elements of lists of ... of lists of vectors we have the gmael macros (for multidimensional array element). The syntax is gmaeln(V, a1, ..., an), where V is a GEN, the ai are indexes, and n is an integer between 1 and 5. This stands for \(x[a_1][a_2]...[a_n]\), and returns a GEN. The macros gel (resp. gmael) are synonyms for gmael1 (resp. gmael2).

Finally, the macro gcoeff(M, i, j) has exactly the meaning of \(M[i,j]\) in GP when M is a matrix. Note that due to the implementation of t_MATS as horizontal lists of vertical vectors, gcoeff(x, y) is actually equivalent to gmael(y, x). One should use gcoeff in matrix context, and gmael otherwise.

4.2.4 Variations on basic functions. In the library syntax descriptions in Chapter 3, we have only given the basic names of the functions. For example gadd(x, y) assumes that x and y are GENs, and creates the result \(x + y\) on the PARI stack. For most of the basic operators and functions, many other variants are available. We give some examples for gadd, but the same is true for all the basic operators, as well as for some simple common functions (a complete list is given in Chapter 6):

GEN gaddgs(GEN x, long y)

GEN gaddsg(long x, GEN y)

In the following one, z is a preexisting GEN and the result of the corresponding operation is put into z. The size of the PARI stack does not change:

void gaddz(GEN x, GEN y, GEN z)

(This last form is inefficient in general and deprecated outside of PARI kernel programming.) Low level kernel functions implement these operators for specialized arguments and are also available: Level 0 deals with operations at the word level (longs and ulong, Level 1 with t_INT and t_REAL and Level 2 with the rest (modular arithmetic, polynomial arithmetic and linear algebra). Here are some examples of Level 1 functions:

GEN addii(GEN x, GEN y): here x and y are GENs of type t_INT (this is not checked).

GEN addrr(GEN x, GEN y): here x and y are GENs of type t_REAL (this is not checked).

There also exist functions addir, addri, mpadd (whose two arguments can be of type t_INT or t_REAL), addis (to add a t_INT and a long) and so on.

The Level 1 names are self-explanatory once you know that i stands for a t_INT, r for a t_REAL, mp for i or r, s for a signed C long integer, u for an unsigned C long integer; finally the suffix z means that the result is not created on the PARI stack but assigned to a preexisting GEN object passed as an extra argument. Chapter 6 gives a description of these low-level functions.
Level 2 names are more complicated, see Section 7.1 for all the gory details, and we content ourselves with a simple example used to implement \( t \_\text{INTMOD} \) arithmetic:

\[
\text{GEN } \text{Fp\_add}(\text{GEN } x, \text{GEN } y, \text{GEN } m) \text{ returns the sum of } x \text{ and } y \text{ modulo } m. \text{ Here } x, y, m \text{ are } t \_\text{INTs} \text{ (this is not checked). The operation is more efficient if the inputs } x, y \text{ are reduced modulo } m \text{, but this is not a necessary condition.}
\]

**Important Note.** These specialized functions are of course more efficient than the generic ones, but note the hidden danger here: the types of the objects involved (which is not checked) must be severely controlled, e.g. using \text{addii} on a \( t \_\text{FRAC} \) argument will cause disasters. Type mismatches may corrupt the PARI stack, though in most cases they will just immediately overflow the stack. Because of this, the PARI philosophy of giving a result which is as exact as possible, enforced for generic functions like \text{gadd} or \text{gmul}, is dropped in kernel routines of Level 1, where it is replaced by the much simpler rule: the result is a \( t \_\text{INT} \) if and only if all arguments are integer types (\( t \_\text{INT} \) but also \text{C long} and \text{ulong}) and a \( t \_\text{REAL} \) otherwise. For instance, multiplying a \( t \_\text{REAL} \) by a \( t \_\text{INT} \) always yields a \( t \_\text{REAL} \) if you use \text{mulir}, where \text{gmul} returns the \( t \_\text{INT gen} \) 0 if the integer is 0.

### 4.2.5 Portability: 32-bit / 64-bit architectures.

PARI supports both 32-bit and 64-bit based machines, but not simultaneously! The library is compiled assuming a given architecture, and some of the header files you include (through \text{pari.h}) will have been modified to match the library.

Portable macros are defined to bypass most machine dependencies. If you want your programs to run identically on 32-bit and 64-bit machines, you have to use these, and not the corresponding numeric values, whenever the precise size of your \text{long} integers might matter. Here are the most important ones:

<table>
<thead>
<tr>
<th>64-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITS_IN_LONG</td>
<td>64</td>
</tr>
<tr>
<td>LONG_IS_64BIT</td>
<td>defined</td>
</tr>
<tr>
<td>DEFAULTPREC</td>
<td>3</td>
</tr>
<tr>
<td>MEDDEFAULTPREC</td>
<td>4</td>
</tr>
<tr>
<td>BIGDEFAULTPREC</td>
<td>5</td>
</tr>
</tbody>
</table>

For instance, suppose you call a transcendental function, such as

\[
\text{GEN } \text{gexp}(\text{GEN } x, \text{long } \text{prec}).
\]

The last argument \( \text{prec} \) is an integer \( \geq 3 \), corresponding to the default floating point precision required. It is only used if \( x \) is an exact object, otherwise the relative precision is determined by the precision of \( x \). Since the parameter \( \text{prec} \) sets the size of the inexact result counted in (\text{long}) words (including codewords), the same value of \( \text{prec} \) will yield different results on 32-bit and 64-bit machines. Real numbers have two codewords (see Section 4.5), so the formula for computing the bit accuracy is

\[
\text{bit\_accuracy(} \text{prec} \text{)} = (\text{prec} - 2) \times \text{BITS\_IN\_LONG}
\]

(this is actually the definition of an inline function). The corresponding accuracy expressed in decimal digits would be

\[
\text{bit\_accuracy(} \text{prec} \text{)} \times \log(2)/\log(10).
\]

For example if the value of \( \text{prec} \) is 5, the corresponding accuracy for 32-bit machines is \((5 - 2) \times \log(2^{32})/\log(10) \approx 28 \) decimal digits, while for 64-bit machines it is \((5 - 2) \times \log(2^{64})/\log(10) \approx 57 \) decimal digits.
Thus, you must take care to change the \texttt{prec} parameter you are supplying according to the bit size, either using the default precisions given by the various \texttt{DEFPREC}s, or by using conditional constructs of the form:

\begin{verbatim}
#ifndef LONG_IS_64BIT
  prec = 4;
#else
  prec = 6;
#endif
\end{verbatim}

which is in this case equivalent to the statement \texttt{prec = MEDDEFAULTPREC;}.

Note that for parity reasons, half the accuracies available on 32-bit architectures (the odd ones) have no precise equivalents on 64-bit machines.

\subsection*{4.2.6 Using \texttt{malloc / free}}

You should make use of the PARI stack as much as possible, and avoid allocating objects using the customary functions. If you do, you should use, or at least have a very close look at, the following wrappers:

\begin{verbatim}
void* pari_malloc(size_t size) \texttt{calls malloc to allocate} size \texttt{bytes and returns a pointer to the allocated memory. If the request fails, an error is raised. The SIGINT signal is blocked until malloc returns, to avoid leaving the system stack in an inconsistent state.}

void* pari_realloc(void* ptr, size_t size) as pari_malloc but calls \texttt{realloc} instead of \texttt{malloc}.

void* pari_calloc(size_t size) as pari_malloc, setting the memory to zero.

void pari_free(void* ptr) \texttt{calls free to liberate the memory space pointed to by ptr, which must have been allocated by malloc (pari_malloc) or realloc (pari_realloc). The SIGINT signal is blocked until free returns.}
\end{verbatim}

If you use the standard \texttt{libc} functions instead of our wrappers, then your functions will be subtly incompatible with the \texttt{gp} calculator: when the user tries to interrupt a computation, the calculator may crash (if a system call is interrupted at the wrong time).

\section*{4.3 Garbage collection}

\subsection*{4.3.1 Why and how}

As we have seen, \texttt{pari_init} allocates a big range of addresses, the \texttt{stack}, that are going to be used throughout. Recall that all PARI objects are pointers. Except for a few universal objects, they all point at some part of the stack.

The stack starts at the address \texttt{bot} and ends just before \texttt{top}. This means that the quantity

\[(\texttt{top} - \texttt{bot}) / \texttt{sizeof(long)}\]

is (roughly) equal to the \texttt{size} argument of \texttt{pari_init}. The PARI stack also has a “current stack pointer” called \texttt{avma}, which stands for available memory address. These three variables are global (declared by \texttt{pari.h}). They are of type \texttt{pari_sp}, which means pari stack pointer.

The stack is oriented upside-down: the more recent an object, the closer to \texttt{bot}. Accordingly, initially \texttt{avma = top}, and \texttt{avma} gets decremented as new objects are created. As its name indicates,
avma always points just after the first free address on the stack, and (GEN)avma is always (a pointer to) the latest created object. When avma reaches bot, the stack overflows, aborting all computations, and an error message is issued. To avoid this you need to clean up the stack from time to time, when intermediate objects are not needed anymore. This is called "garbage collecting."

We are now going to describe briefly how this is done. We will see many concrete examples in the next subsection.

• First, PARI routines do their own garbage collecting, which means that whenever a documented function from the library returns, only its result(s) have been added to the stack, possibly up to a very small overhead (non-documented ones may not do this). In particular, a PARI function that does not return a GEN does not clutter the stack. Thus, if your computation is small enough (e.g. you call few PARI routines, or most of them return long integers), then you do not need to do any garbage collecting. This is probably the case in many of your subroutines. Of course the objects that were on the stack before the function call are left alone. Except for the ones listed below, PARI functions only collect their own garbage.

• It may happen that all objects that were created after a certain point can be deleted — for instance, if the final result you need is not a GEN, or if some search proved futile. Then, it is enough to record the value of avma just before the first garbage is created, and restore it upon exit:

        pari_sp av = avma; /* record initial avma */
        garbage ...
        avma = av; /* restore it */

All objects created in the garbage zone will eventually be overwritten: they should no longer be accessed after avma has been restored.

• If you want to destroy (i.e. give back the memory occupied by) the latest PARI object on the stack (e.g. the latest one obtained from a function call), you can use the function

        void cgiv(GEN z)

where z is the object you want to give back. This is equivalent to the above where the initial av is computed from z.

• Unfortunately life is not so simple, and sometimes you will want to give back accumulated garbage during a computation without losing recent data. We shall start with the lowest level function to get a feel for the underlying mechanisms, we shall describe simpler variants later:

        GEN gerepile(pari_sp ltop, pari_sp lbot, GEN q)

This function cleans up the stack between ltop and lbot, where lbot < ltop, and returns the updated object q. This means:

1) we translate (copy) all the objects in the interval [avma, lbot], so that its right extremity abuts the address ltop. Graphically

```
bot       avma  lbot  ltop  top
|-------------[+++++[-------------[[-/-/-/-/-/-/-][-/-/-/-/-/-/-]]]]| Start
free memory garbage
```

becomes:

```
bot       avma  ltop  top
|-------------[-------------[[-/-/-/-/-/-/-][-/-/-/-/-/-/-]]]]| Start
free memory
```

16
where ++ denote significant objects, -- the unused part of the stack, and --/- the garbage we remove.

2) The function then inspects all the PARI objects between avma and lbot (i.e. the ones that we want to keep and that have been translated) and looks at every component of such an object which is not a codeword. Each such component is a pointer to an object whose address is either

— between avma and lbot, in which case it is suitably updated,
— larger than or equal to ltop, in which case it does not change, or
— between lbot and ltop in which case gerepile raises an error ("significant pointers lost in gerepile").

3) avma is updated (we add ltop − lbot to the old value).

4) We return the (possibly updated) object q: if q initially pointed between avma and lbot, we return the updated address, as in 2). If not, the original address is still valid, and is returned!

As stated above, no component of the remaining objects (in particular q) should belong to the erased segment [lbot, ltop[, and this is checked within gerepile. But beware as well that the addresses of the objects in the translated zone change after a call to gerepile, so you must not access any pointer which previously pointed into the zone below ltop. If you need to recover more than one object, use the gerepileall function below.

Remark. As a consequence of the preceding explanation, if a PARI object is to be relocated by gerepile then, apart from universal objects, the chunks of memory used by its components should be in consecutive memory locations. All GENs created by documented PARI functions are guaranteed to satisfy this. This is because the gerepile function knows only about two connected zones: the garbage that is erased (between lbot and ltop) and the significant pointers that are copied and updated. If there is garbage interspersed with your objects, disaster occurs when we try to update them and consider the corresponding “pointers”. In most cases of course the said garbage is in fact a bunch of other GENs, in which case we simply waste time copying and updating them for nothing. But be wary when you allow objects to become disconnected.

In practice this is achieved by the following programming idiom:

```c
ltop = avma; garbage(); lbot = avma; q = anything();
return gerepile(ltop, lbot, q); /* returns the updated q */
```

or directly

```c
ltop = avma; garbage(); lbot = avma;
return gerepile(ltop, lbot, anything());
```

Beware that

```c
ltop = avma; garbage();
return gerepile(ltop, avma, anything())
```

might work, but should be frowned upon. We cannot predict whether avma is evaluated after or before the call to anything(): it depends on the compiler. If we are out of luck, it is after the call, so the result belongs to the garbage zone and the gerepile statement becomes equivalent to avma = ltop. Thus we return a pointer to random garbage.

17
4.3.2 Variants.

GEN gerepileupto(pari_sp ltop, GEN q). Cleans the stack between ltop and the connected object q and returns q updated. For this to work, q must have been created before all its components, otherwise they would belong to the garbage zone! Unless mentioned otherwise, documented PARI functions guarantee this.

GEN gerepilecopy(pari_sp ltop, GEN x). Functionally equivalent to, but more efficient than

gerepileupto(ltop, gcopy(x))

In this case, the GEN parameter x need not satisfy any property before the garbage collection: it may be disconnected, components created before the root, and so on. Of course, this is about twice slower than either gerepileupto or gerepile, because x has to be copied to a clean stack zone first. This function is a special case of gerepileall below, where n = 1.

void gerepileall(pari_sp ltop, int n, ...). To cope with complicated cases where many objects have to be preserved. The routine expects n further arguments, which are the addresses of the GENs you want to preserve:

    pari_sp ltop = avma; /* remember the current top of the stack */
    ...; y = ...; ... x = ...; ...
    gerepileall(ltop, 2, &x, &y);

It cleans up the most recent part of the stack (between ltop and avma), updating all the GENs added to the argument list. A copy is done just before the cleaning to preserve them, so they do not need to be connected before the call. With gerepilecopy, this is the most robust of the gerepile functions (the less prone to user error), hence the slowest.

void gerepileallsp(pari_sp ltop, pari_sp lbot, int n, ...). More efficient, but trickier than gerepileall. Cleans the stack between lbot and ltop and updates the GENs pointed at by the elements of gptr without any further copying. This is subject to the same restrictions as gerepile, the only difference being that more than one address gets updated.

4.3.3 Examples.

4.3.3.1 gerepile.

Let x and y be two preexisting PARI objects and suppose that we want to compute \( x^2 + y^2 \). This is done using the following program:

    GEN x2 = gsqr(x);
    GEN y2 = gsqr(y), z = gadd(x2,y2);

The GEN z indeed points at the desired quantity. However, consider the stack: it contains as unnecessary garbage x2 and y2. More precisely it contains (in this order) z, y2, x2. (Recall that, since the stack grows downward from the top, the most recent object comes first.)

It is not possible to get rid of x2, y2 before z is computed, since they are used in the final operation. We cannot record avma before x2 is computed and restore it later, since this would destroy z as well. It is not possible either to use the function cgiv since x2 and y2 are not at the bottom of the stack and we do not want to give back z.

But using gerepile, we can give back the memory locations corresponding to x2, y2, and move the object z upwards so that no space is lost. Specifically:

    pari_sp ltop = avma; /* remember the current top of the stack */
GEN x2 = gsqr(x);
GEN y2 = gsqr(y);

pari_sp lbot = avma; /* the bottom of the garbage pile */
GEN z = gadd(x2, y2); /* z is now the last object on the stack */
z = gerepile(ltop, lbot, z);

Of course, the last two instructions could also have been written more simply:

\[ z = \text{gerepile}(ltop, lbot, \text{gadd}(x2, y2)); \]

In fact \text{gerepileupto} is even simpler to use, because the result of \text{gadd} is the last object on the stack and \text{gadd} is guaranteed to return an object suitable for \text{gerepileupto}:

\[
\begin{align*}
\text{ltop} &= \text{avma}; \\
z &= \text{gerepileupto}(ltop, \text{gadd}(\text{gsqr}(x), \text{gsqr}(y))); \\
\end{align*}
\]

Make sure you understand exactly what has happened before you go on!

**Remark on assignments and \text{gerepile}**. When the tree structure and the size of the PARI objects which will appear in a computation are under control, one may allocate sufficiently large objects at the beginning, use assignment statements, then simply restore \text{avma}. Coming back to the above example, note that if we know that \(x\) and \(y\) are of type real fitting into \text{DEFAULTPREC} words, we can program without using \text{gerepile} at all:

\[
\begin{align*}
z &= \text{cgetr}(\text{DEFAULTPREC}); \\
ltop &= \text{avma}; \\
g\text{affect}(\text{gadd}(\text{gsqr}(x), \text{gsqr}(y)), z); \\
avma &= \text{ltop}; \\
\end{align*}
\]

This is often slower than a craftily used \text{gerepile} though, and certainly more cumbersome to use. As a rule, assignment statements should generally be avoided.

**Variations on a theme**. It is often necessary to do several \text{gerepiles} during a computation. However, the fewer the better. The only condition for \text{gerepile} to work is that the garbage be connected. If the computation can be arranged so that there is a minimal number of connected pieces of garbage, then it should be done that way.

For example suppose we want to write a function of two \text{GEN} variables \(x\) and \(y\) which creates the vector \(\lfloor x^2 + y, y^2 + x \rfloor\). Without garbage collecting, one would write:

\[
\begin{align*}
p1 &= \text{gsqr}(x); \\
p2 &= \text{gadd}(p1, y); \\
p3 &= \text{gsqr}(y); \\
p4 &= \text{gadd}(p3, x); \\
z &= \text{mkvec2}(p2, p4); /* not suitable for gerepileupto! */ \\
\end{align*}
\]

This leaves a dirty stack containing (in this order) \(z, p4, p3, p2, p1\). The garbage here consists of \(p1\) and \(p3\), which are separated by \(p2\). But if we compute \(p3\) before \(p2\) then the garbage becomes connected, and we get the following program with garbage collecting:

\[
\begin{align*}
ltop &= \text{avma}; \\
p1 &= \text{gsqr}(x); \\
p3 &= \text{gsqr}(y); \\
lbot &= \text{avma}; \\
z &= \text{cgetg}(3, \text{t_VEC}); \\
\text{gel}(z, 1) &= \text{gadd}(p1, y); \\
\text{gel}(z, 2) &= \text{gadd}(p3, x); \\
z &= \text{gerepile}(ltop, lbot, z); \\
\end{align*}
\]

Finishing by \(z = \text{gerepileupto}(ltop, z)\) would be ok as well. Beware that

\[
\begin{align*}
ltop &= \text{avma}; \\
p1 &= \text{gadd}(\text{gsqr}(x), y); \\
p3 &= \text{gadd}(\text{gsqr}(y), x); \\
z &= \text{cgetg}(3, \text{t_VEC}); \\
\end{align*}
\]
is a disaster since \( p_1 \) and \( p_3 \) are created before \( z \), so the call to \texttt{gerepileupto} overwrites them, leaving \( \text{gel}(z, 1) \) and \( \text{gel}(z, 2) \) pointing at random data! The following does work:

```plaintext
ltop = avma; p1 = gsqr(x); p3 = gsqr(y);
lbot = avma; z = mkvec2(gadd(p1,y), gadd(p3,x));
z = gerepile(ltop,lbot,z);
```

but is very subtly wrong in the sense that \( z = \text{gerepileupto}(ltop, z) \) would not work. The reason being that \texttt{mkvec2} creates the root \( z \) of the vector \textit{after} its arguments have been evaluated, creating the components of \( z \) too early; \texttt{gerepile} does not care, but the created \( z \) is a time bomb which will explode on any later \texttt{gerepileupto}. On the other hand

```plaintext
ltop = avma; z = cgetg(3, t_VEC);
gel(z, 1) = gadd(gsqr(x), y);
gel(z, 2) = gadd(gsqr(y), x); z = gerepileupto(ltop, z); /* INEFFICIENT */
```

leaves the results of \( \text{gsqr}(x) \) and \( \text{gsqr}(y) \) on the stack (and lets \texttt{gerepileupto} update them for naught). Finally, the most elegant and efficient version (with respect to time and memory use) is as follows

```plaintext
z = cgetg(3, t_VEC);
ltop = avma; gel(z, 1) = gerepileupto(ltop, gadd(gsqr(x), y));
ltop = avma; gel(z, 2) = gerepileupto(ltop, gadd(gsqr(y), x));
```

which avoids updating the container \( z \) and cleans up its components individually, as soon as they are computed.

**One last example.** Let us compute the product of two complex numbers \( x \) and \( y \), using the 3M method which requires 3 multiplications instead of the obvious 4. Let \( z = x \ast y \), and set \( x = x_r + i \ast x_i \) and similarly for \( y \) and \( z \). We compute \( p_1 = x_r \ast y_r \), \( p_2 = x_i \ast y_i \), \( p_3 = (x_r + x_i) \ast (y_r + y_i) \), and then we have \( z_r = p_1 - p_2 \), \( z_i = p_3 - (p_1 + p_2) \). The program is as follows:

```plaintext
ltop = avma;
p1 = gmul(gel(x,1), gel(y,1));
p2 = gmul(gel(x,2), gel(y,2));
p3 = gmul(gadd(gel(x,1), gel(x,2)), gadd(gel(y,1), gel(y,2)));
p4 = gadd(p1,p2);
lbot = avma; z = cgetg(3, t_COMPLEX);
gel(z, 1) = gsub(p1,p2);
gel(z, 2) = gsub(p3,p4); z = gerepile(ltop,lbot,z);
```
Exercise. Write a function which multiplies a matrix by a column vector. Hint: start with a \texttt{cgetg} of the result, and use \texttt{gerepile} whenever a coefficient of the result vector is computed. You can look at the answer in \texttt{src/basemath/RgV.c:RgM\_RgC\_mul()}.  

4.3.3.2 \texttt{gerepileall}.

Let us now see why we may need the \texttt{gerepileall} variants. Although it is not an infrequent occurrence, we do not give a specific example but a general one: suppose that we want to do a computation (usually inside a larger function) producing more than one PARI object as a result, say two for instance. Then even if we set up the work properly, before cleaning up we have a stack which has the desired results \(z_1, z_2\) (say), and then connected garbage from \texttt{lbot} to \texttt{ltop}. If we write

\[
z_1 = \text{gerepile}(\texttt{ltop}, \texttt{lbot}, z_1);
\]

then the stack is cleaned, the pointers fixed up, but we have lost the address of \(z_2\). This is where we need the \texttt{gerepileall} function:

\[
\text{gerepileall}(\texttt{ltop}, 2, &z_1, &z_2)
\]

copies \(z_1\) and \(z_2\) to new locations, cleans the stack from \texttt{ltop} to the old \texttt{avma}, and updates the pointers \(z_1\) and \(z_2\). Here we do not assume anything about the stack: the garbage can be disconnected and \(z_1, z_2\) need not be at the bottom of the stack. If all of these assumptions are in fact satisfied, then we can call \texttt{gerepilemanysp} instead, which is usually faster since we do not need the initial copy (on the other hand, it is less cache friendly).

A most important usage is “random” garbage collection during loops whose size requirements we cannot (or do not bother to) control in advance:

\[
\begin{align*}
\text{pari\_sp} \ \texttt{ltop} & = \texttt{avma}, \ \texttt{limit} = \text{stack\_lim}(\texttt{avma}, 1); \\
\text{GEN} \ x, y; \\
\text{while} (\ldots) \\
\{ \\
\text{garbage(); x = anything();} \\
\text{garbage(); y = anything(); garbage();} \\
\text{if (avma < limit) /* memory is running low (half spent since entry) */} \\
\text{gerepileall}(\texttt{ltop}, 2, &x, &y); \\
\}
\end{align*}
\]

Here we assume that only \(x\) and \(y\) are needed from one iteration to the next. As it would be costly to call \texttt{gerepile} once for each iteration, we only do it when it seems to have become necessary. The macro \texttt{stack\_lim}(\texttt{avma}, \texttt{n}) denotes an address where \(2^{n-1}/(2^{n-1}+1)\) of the remaining stack space is exhausted \((1/2 for n = 1, 2/3 for n = 2)\).

4.3.4 Comments.

First, \texttt{gerepile} has turned out to be a flexible and fast garbage collector for number-theoretic computations, which compares favorably with more sophisticated methods used in other systems. Our benchmarks indicate that the price paid for using \texttt{gerepile} and \texttt{gerepile}-related copies, when properly used, is usually less than 1\% of the total running time, which is quite acceptable!

Second, it is of course harder on the programmer, and quite error-prone if you do not stick to a consistent PARI programming style. If all seems lost, just use \texttt{gerepilecopy} (or \texttt{gerepileall}) to fix up the stack for you. You can always optimize later when you have sorted out exactly which routines are crucial and what objects need to be preserved and their usual sizes.
If you followed us this far, congratulations, and rejoice: the rest is much easier.

4.4 Creation of PARI objects, assignments, conversions.

4.4.1 Creation of PARI objects. The basic function which creates a PARI object is

GEN cgetg(long l, long t) l specifies the number of longwords to be allocated to the object, and t is the type of the object, in symbolic form (see Section 4.5 for the list of these). The precise effect of this function is as follows: it first creates on the PARI stack a chunk of memory of size length longwords, and saves the address of the chunk which it will in the end return. If the stack has been used up, a message to the effect that “the PARI stack overflows” is printed, and an error raised. Otherwise, it sets the type and length of the PARI object. In effect, it fills its first codeword (z[0]). Many PARI objects also have a second codeword (types t_INT, t_REAL, t_PADIC, t_POL, and t_SER). In case you want to produce one of those from scratch, which should be exceedingly rare, it is your responsibility to fill this second codeword, either explicitly (using the macros described in Section 4.5), or implicitly using an assignment statement (using gaffect).

Note that the length argument l is predetermined for a number of types: 3 for types t_INTMOD, t_FRAC, t_COMPLEX, t_POLMOD, t_RFRAC, 4 for type t_QUAD and t_QFI, and 5 for type t_PADIC and t_QFR. However for the sake of efficiency, cgetg does not check this: disasters will occur if you give an incorrect length for those types.

Notes. 1) The main use of this function is to create efficiently a constant object, or to prepare for later assignments (see Section 4.4.3). Most of the time you will use GEN objects as they are created and returned by PARI functions. In this case you do not need to use cgetg to create space to hold them.

2) For the creation of leaves, i.e. t_INT or t_REAL,

GEN cgeti(long length)

GEN cgetr(long length)

should be used instead of cgetg(length, t_INT) and cgetg(length, t_REAL) respectively. Finally

GEN cgetc(long prec)

creates a t_COMPLEX whose real and imaginary part are t_REALs allocated by cgetr(prec).
Examples. 1) Both \( z = \text{cgeti}(\text{DEFAULTPREC}) \) and \( \text{cgetg}(\text{DEFAULTPREC}, \text{t_INT}) \) create a \text{t_INT} whose “precision” is \( \text{bit\_accuracy} (\text{DEFAULTPREC}) = 64 \). This means \( z \) can hold rational integers of absolute value less than \( 2^{64} \). Note that in both cases, the second codeword is not filled. Of course we could use numerical values, e.g. \( \text{cgeti}(4) \), but this would have different meanings on different machines as \( \text{bit\_accuracy}(4) \) equals 64 on 32-bit machines, but 128 on 64-bit machines.

2) The following creates a complex number whose real and imaginary parts can hold real numbers of precision \( \text{bit\_accuracy} (\text{MEDDEFAULTPREC}) = 96 \) bits:

\[
\begin{align*}
    z &= \text{cgetg}(3, \text{t_COMPLEX}); \\
    \text{gel}(z, 1) &= \text{cgetr}(\text{MEDDEFAULTPREC}); \\
    \text{gel}(z, 2) &= \text{cgetr}(\text{MEDDEFAULTPREC});
\end{align*}
\]

or simply \( z = \text{cgetc}(\text{MEDDEFAULTPREC}) \).

3) To create a matrix object for \( 4 \times 3 \) matrices:

\[
\begin{align*}
    z &= \text{cgetg}(4, \text{t_MAT}); \\
    \text{for}(i=1; \ i<4; \ i++) \ \text{gel}(z, i) &= \text{cgetg}(5, \text{t_COL});
\end{align*}
\]

or simply \( z = \text{zeromatcopy}(4, 3) \), which further initializes all entries to \text{gen}_0.

These last two examples illustrate the fact that since PARI types are recursive, all the branches of the tree must be created. The function \text{cgetg} creates only the “root”, and other calls to \text{cgetg} must be made to produce the whole tree. For matrices, a common mistake is to think that \( z = \text{cgetg}(4, \text{t_MAT}) \) (for example) creates the root of the matrix: one needs also to create the column vectors of the matrix (obviously, since we specified only one dimension in the first \text{cgetg}!). This is because a matrix is really just a row vector of column vectors (hence a priori not a basic type), but it has been given a special type number so that operations with matrices become possible.

Finally, to facilitate input of constant objects when speed is not paramount, there are four \text{varargs} functions:

\text{GEN mkintn(long n, ...)} returns the non-negative \text{t_INT} whose development in base \( 2^{32} \) is given by the following \( n \) words (unsigned \text{long}). It is assumed that all such arguments are less than \( 2^{32} \) (the actual \text{sizeof}(\text{long}) is irrelevant, the behavior is also as above on 64-bit machines).

\[
\begin{align*}
    \text{mkintn}(3, a2, a1, a0); \\
    \text{returns } a2 \cdot 2^{64} + a1 \cdot 2^{32} + a0.
\end{align*}
\]

\text{GEN mkpoln(long n, ...)} Returns the \text{t_POL} whose \( n \) coefficients (\text{GEN}) follow, in order of decreasing degree.

\[
\begin{align*}
    \text{mkpoln}(3, \text{gen}_1, \text{gen}_2, \text{gen}_0); \\
    \text{returns the polynomial } X^2 + 2X \text{ (in variable 0, use } \text{setvarn} \text{ if you want other variable numbers). Beware that } n \text{ is the number of coefficients, hence } \text{one more than the degree.}
\end{align*}
\]

\text{GEN mkvecn(long n, ...)} returns the \text{t_VEC} whose \( n \) coefficients (\text{GEN}) follow.

\text{GEN mkcoln(long n, ...)} returns the \text{t_COL} whose \( n \) coefficients (\text{GEN}) follow.
Warning. Contrary to the policy of general PARI functions, the latter three functions do not copy their arguments, nor do they produce an object a priori suitable for gerepileupto. For instance

```c
/* gerepile-safe: components are universal objects */
z = mkvecn(3, gen_1, gen_0, gen_2);
/* not OK for gerepileupto: stoi(3) creates component before root */
z = mkvecn(3, stoi(3), gen_0, gen_2);
/* NO! First vector component x is destroyed */
x = gclone(gen_1);
z = mkvecn(3, x, gen_0, gen_2);
gunclone(x);
```

The following function is also available as a special case of mkintn:

```c
GEN uu32toi(ulong a, ulong b)
```

Returns the GEN equal to $2^{32}a + b$, assuming that $a, b < 2^{32}$. This does not depend on sizeof(long): the behavior is as above on both 32 and 64-bit machines.

### 4.4.2 Sizes

- **long gsizeword(GEN x)** returns the total number of BITS_IN_LONG-bit words occupied by the tree representing x.
- **long gsizebyte(GEN x)** returns the total number of bytes occupied by the tree representing x, i.e. gsizeword(x) multiplied by sizeof(long). This is normally useless since PARI functions use a number of words as input for lengths and precisions.

### 4.4.3 Assignments

Firstly, if x and y are both declared as GEN (i.e. pointers to something), the ordinary C assignment `y = x` makes perfect sense: we are just moving a pointer around. However, physically modifying either x or y (for instance, x[1] = 0) also changes the other one, which is usually not desirable.

**Very important note.** Using the functions described in this paragraph is inefficient and often awkward: one of the gerepile functions (see Section 4.3) should be preferred. See the paragraph end for one exception to this rule.

The general PARI assignment function is the function `gffect` with the following syntax:

```c
void gffect(GEN x, GEN y)
```

Its effect is to assign the PARI object x into the preexisting object y. Both x and y must be scalar types. For convenience, vector or matrices of scalar types are also allowed.

This copies the whole structure of x into y so many conditions must be met for the assignment to be possible. For instance it is allowed to assign a t_INT into a t_REAL, but the converse is forbidden. For that, you must use the truncation or rounding function of your choice, e.g. `mpfloor`.

It can also happen that y is not large enough or does not have the proper tree structure to receive the object x. For instance, let y the zero integer with length equal to 2; then y is too small to accommodate any non-zero t_INT. In general common sense tells you what is possible, keeping in mind the PARI philosophy which says that if it makes sense it is valid. For instance, the assignment of an imprecise object into a precise one does not make sense. However, a change in precision of imprecise objects is allowed, even if it increases its accuracy: we complement the
“mantissa” with infinitely many 0 digits in this case. (Mantissa between quotes, because this is not restricted to \texttt{t.REAL}s, it also applies for \(p\)-adics for instance.)

All functions ending in “\texttt{z}” such as \texttt{gaddz} (see Section 4.2.4) implicitly use this function. In fact what they exactly do is record \texttt{avma} (see Section 4.3), perform the required operation, \texttt{gaffect} the result to the last operand, then restore the initial \texttt{avma}.

You can assign ordinary C long integers into a PARI object (not necessarily of type \texttt{t.INT}) using

\begin{verbatim}
void gaffsg(long s, GEN y)
\end{verbatim}

\textbf{Note.} Due to the requirements mentioned above, it is usually a bad idea to use \texttt{gaffect} statements. There is one exception: for simple objects (e.g. leaves) whose size is controlled, they can be easier to use than \texttt{gerepile}, and about as efficient.

\textbf{Coercion.} It is often useful to coerce an inexact object to a given precision. For instance at the beginning of a routine where precision can be kept to a minimum; otherwise the precision of the input is used in all subsequent computations, which is inefficient if the latter is known to thousands of digits. One may use the \texttt{gaffect} function for this, but it is easier and more efficient to call

\begin{verbatim}
GEN gtofp(GEN x, long prec)
\end{verbatim}

converts the complex number \(x\) (\texttt{t.INT}, \texttt{t.REAL}, \texttt{t.FRAC}, \texttt{t.QUAD} or \texttt{t.COMPLEX}) to either a \texttt{t.REAL} or \texttt{t.COMPLEX} whose components are \texttt{t.REAL} of length \(prec\).

\subsection*{4.4.4 Copy.} It is also very useful to copy a PARI object, not just by moving around a pointer as in the \(y = x\) example, but by creating a copy of the whole tree structure, without pre-allocating a possibly complicated \(y\) to use with \texttt{gaffect}. The function which does this is called \texttt{gcopy}. Its syntax is:

\begin{verbatim}
GEN gcopia(GEN x)
\end{verbatim}

and the effect is to create a new copy of \(x\) on the PARI stack.

Sometimes, on the contrary, a quick copy of the skeleton of \(x\) is enough, leaving pointers to the original data in \(x\) for the sake of speed instead of making a full recursive copy. Use \texttt{GEN shallowcopy(GEN x)} for this. Note that the result is not suitable for \texttt{gerepileupto}!

Make sure at this point that you understand the difference between \(y = x\), \(y = \text{gcopia}(x)\), \(y = \text{shallowcopy}(x)\) and \texttt{gaffect}(x,y).

\subsection*{4.4.5 Clones.} Sometimes, it is more efficient to create a \textit{persistent} copy of a PARI object. This is not created on the stack but on the heap, hence unaffected by \texttt{gerepile} and friends. The function which does this is called \texttt{gclone}. Its syntax is:

\begin{verbatim}
GEN gclone(GEN x)
\end{verbatim}

A clone can be removed from the heap (thus destroyed) using

\begin{verbatim}
void gunclone(GEN x)
\end{verbatim}

No PARI object should keep references to a clone which has been destroyed!
4.4.6 Conversions. The following functions convert C objects to PARI objects (creating them on the stack as usual):

GEN stoi(long s): C long integer ("small") to t_INT.

GEN dbltor(double s): C double to t_REAL. The accuracy of the result is 19 decimal digits, i.e. a type t_REAL of length DEFAULTPREC, although on 32-bit machines only 16 of them are significant.

We also have the converse functions:

long itos(GEN x): x must be of type t_INT,

double rtodbl(GEN x): x must be of type t_REAL,

as well as the more general ones:

long gtolong(GEN x),

double gtodouble(GEN x).

4.5 Implementation of the PARI types.

We now go through each type and explain its implementation. Let z be a GEN, pointing at a PARI object. In the following paragraphs, we will constantly mix two points of view: on the one hand, z is treated as the C pointer it is, on the other, as PARI's handle on some mathematical entity, so we will shamelessly write \( z \neq 0 \) to indicate that the value thus represented is nonzero (in which case the pointer z is certainly non-NULL). We offer no apologies for this style. In fact, you had better feel comfortable juggling both views simultaneously in your mind if you want to write correct PARI programs.

Common to all the types is the first codeword z[0], which we do not have to worry about since this is taken care of by cgetg. Its precise structure depends on the machine you are using, but it always contains the following data: the internal type number associated to the symbolic type name, the length of the root in longwords, and a technical bit which indicates whether the object is a clone or not (see Section 4.4.5). This last one is used by gp for internal garbage collecting, you will not have to worry about it.

Some types have a second codeword, different for each type, which we will soon describe as we will shortly consider each of them in turn.

The first codeword is handled through the following macros:

long typ(GEN z) returns the type number of z.

void settyp(GEN z, long n) sets the type number of z to n (you should not have to use this function if you use cgetg).

long lg(GEN z) returns the length (in longwords) of the root of z.

long setlg(GEN z, long l) sets the length of z to l (you should not have to use this function if you use cgetg; however, see an advanced example in Section 4.10).

long isclone(GEN z) is z a clone?

void setisclone(GEN z) sets the clone bit.

void unsetisclone(GEN z) clears the clone bit.
Important remark. For the sake of efficiency, none of the codeword-handling macros check the types of their arguments even when there are stringent restrictions on their use. It is trivial to create invalid objects, or corrupt one of the “universal constants” (e.g. setting the sign of gen_0 to 1), and they usually provide negligible savings. Use higher level functions whenever possible.

Remark. The clone bit is there so that gunclone can check it is deleting an object which was allocated by gclone. Miscellaneous vector entries are often cloned by gp so that a GP statement like v[1] = x does not involve copying the whole of v: the component v[1] is deleted if its clone bit is set, and is replaced by a clone of x. Don’t set/unset yourself the clone bit unless you know what you are doing: in particular never set the clone bit of a vector component when the said vector is scheduled to be uncloned. Hackish code may abuse the clone bit to tag objects for reasons unrelated to the above instead of using proper data structures. Don’t do that.

4.5.1 Type t_INT (integer). this type has a second codeword z[1] which contains the following information:

- the sign of z: coded as 1, 0 or -1 if \( z > 0 \), \( z = 0 \), \( z < 0 \) respectively.
- the effective length of z, i.e. the total number of significant longwords. This means the following: apart from the integer 0, every integer is “normalized”, meaning that the most significant mantissa longword is non-zero. However, the integer may have been created with a longer length. Hence the “length” which is in \( z[0] \) can be larger than the “effective length” which is in \( z[1] \).

This information is handled using the following macros:

- long signe(GEN z) returns the sign of z.
- void setsigne(GEN z, long s) sets the sign of z to s.
- long lgefint(GEN z) returns the effective length of z.
- void setlgefint(GEN z, long l) sets the effective length of z to l.

The integer 0 can be recognized either by its sign being 0, or by its effective length being equal to 2. Now assume that \( z \neq 0 \), and let

\[
|z| = \sum_{i=0}^{n} z_i B^i, \quad \text{where } z_n \neq 0 \text{ and } B = 2^{\text{BITS}_\text{IN}_\text{LONG}}.
\]

With these notations, \( n = \text{lgefint}(z) - 3 \), and the mantissa of z may be manipulated via the following interface:

- GEN int_MSW(GEN z) returns a pointer to the most significant word of z, \( z_n \).
- GEN int_LSW(GEN z) returns a pointer to the least significant word of z, \( z_0 \).
- GEN int_W(GEN z, long i) returns the \( i \)-th significant word of z, \( z_i \). Accessing the \( i \)-th significant word for \( i > n \) yields unpredictable results.
- GEN int_W_lg(GEN z, long i, long lz) returns the \( i \)-th significant word of z, \( z_i \), assuming lgefint(z) is \( lz (= n + 3) \). Accessing the \( i \)-th significant word for \( i > n \) yields unpredictable results.
- GEN int_precW(GEN z) returns the previous (less significant) word of z, \( z_{i-1} \) assuming z points to \( z_i \).
GEN int_nextW(GEN z) returns the next (more significant) word of z, \(z_{i+1}\) assuming z points to \(z_i\).

Unnormalized integers, such that \(z_n\) is possibly 0, are explicitly forbidden. To enforce this, one may write an arbitrary mantissa then call

```c
void int_normalize(GEN z, long known0)
```

normalizes in place a non-negative integer (such that \(z_n\) is possibly 0), assuming at least the first \(known0\) words are zero.

For instance a binary `and` could be implemented in the following way:

```c
GEN AND(GEN x, GEN y) {
    long i, lx, ly, lout;
    long *xp, *yp, *outp; /* mantissa pointers */
    GEN out;
    if (!signe(x) || !signe(y)) return gen_0;
    lx = lgefint(x); xp = int_LSW(x);
    ly = lgefint(y); yp = int_LSW(y); lout = min(lx,ly); /* > 2 */
    out = cgeti(lout); out[1] = evalsigne(1) | evallgefint(lout);
    outp = int_LSW(out);
    for (i=2; i < lout; i++)
    {
        *outp = (*xp) & (*yp);
        outp = int_nextW(outp);
        xp = int_nextW(xp);
        yp = int_nextW(yp);
    }
    if ( !*int_MSW(out) ) out = int_normalize(out, 1);
    return out;
}
```

This low-level interface is mandatory in order to write portable code since PARI can be compiled using various multiprecision kernels, for instance the native one or GNU MP, with incompatible internal structures (for one thing, the mantissa is oriented in different directions).

The following further functions are available:

- `int mpodd(GEN x)` which is 1 if \(x\) is odd, and 0 otherwise.
- `long mod2(GEN x)`
- `long mod4(GEN x)`
- `long mod8(GEN x)`
- `long mod16(GEN x)`
- `long mod32(GEN x)`
- `long mod64(GEN x)` give the residue class of \(x\) modulo the corresponding power of 2, for positive \(x\).
- `ulong mod2BIL(GEN x)` returns the least significant word of \(|x|\), still assuming that \(x \neq 0\).
These functions directly access the binary data and are thus much faster than the generic modulo functions. Besides, they return long integers instead of GENs, so they do not clutter up the stack.

4.5.2 Type t_REAL (real number). This type has a second codeword z[1] which also encodes its sign, obtained or set using the same functions as for a t_INT, and a binary exponent. This exponent is handled using the following macros:

long expo(GEN z) returns the exponent of z. This is defined even when z is equal to zero, see Section 1.4.

void setexpo(GEN z, long e) sets the exponent of z to e.

Note the functions:

long gexpo(GEN z) which tries to return an exponent for z, even if z is not a real number.

long gsigne(GEN z) which returns a sign for z, even when z is neither real nor integer (a rational number for instance).

The real zero is characterized by having its sign equal to 0. If z is not equal to 0, then is is represented as $2^e M$, where $e$ is the exponent, and $M \in [1, 2]$ is the mantissa of z, whose digits are stored in $z[2], \ldots, z[\lfloor \log(z) \rfloor - 1]$.

More precisely, let $m$ be the integer ($z[2], \ldots, z[\lfloor \log(z) \rfloor - 1]$) in base $2^{BITS\_IN\_LONG}$; here, $z[2]$ is the most significant longword and is normalized, i.e. its most significant bit is 1. Then we have $M := m/2^{\text{bit\_accuracy}(\log(z)) - 1 - \text{expo}(z)}$.

GEN mantissa_real(GEN z, long *e) returns the mantissa $m$ of z, and sets *e to the exponent $\text{bit\_accuracy}(\log(z)) - 1 - \text{expo}(z)$, so that $z = m/2^e$.

Thus, the real number 3.25 to accuracy $\text{bit\_accuracy}(\log(z))$ is represented as $z[0]$ (encoding type = t_REAL, $\log(z)$), $z[1]$ (encoding sign = 1, expo = 1), $z[2] = 0xe0000000$, $z[3] = \ldots = z[\lfloor \log(z) - 1 \rfloor] = 0x0$.

4.5.3 Type t_INTMOD. z[1] points to the modulus, and z[2] at the number representing the class z. Both are separate GEN objects, and both must be t_INTs, satisfying the inequality $0 \leq z[2] < z[1]$.

4.5.4 Type t_FRAC (rational number). z[1] points to the numerator n, and z[2] to the denominator d. Both must be of type t_INT such that $n \neq 0$, $d > 0$ and $(n,d) = 1$.

4.5.5 Type t_FFELT (finite field element). (Experimental)

Components of this type should normally not be accessed directly. Instead, finite field elements should be created using ffgen.

The second codeword z[1] determines the storage format of the element, among

- t_FF_FpXQ: $A = z[2]$ and $T = z[3]$ are FpX, $p = z[4]$ is a t_INT, where $p$ is a prime number, $T$ is irreducible modulo $p$, and $\deg A < \deg T$. This represents the element $A \mod T$ in $F_p[X]/T$.

- t_FF_FlXq: $A = z[2]$ and $T = z[3]$ are FlX, $l = z[4]$ is a t_INT, where $l$ is a prime number, $T$ is irreducible modulo $l$, and $\deg A < \deg T$ This represents the element $A \mod T$ in $F_l[X]/T$.

- t_FF_F2xq: $A = z[2]$ and $T = z[3]$ are F2x, $l = z[4]$ is the t_INT 2, $T$ is irreducible modulo 2, and $\deg A < \deg T$. This represents the element $A \mod T$ in $F_2[X]/T$. 

29
4.5.6 Type \texttt{t\_COMPLEX} (complex number). \(z[1]\) points to the real part, and \(z[2]\) to the imaginary part. The components \(z[1]\) and \(z[2]\) must be of type \texttt{t\_INT}, \texttt{t\_REAL} or \texttt{t\_FRAC}. For historical reasons \texttt{t\_INTMOD} and \texttt{t\_PADIC} are also allowed (the latter for \(p = 2\) or congruent to 3 mod 4 only), but one should rather use the more general \texttt{t\_POLMOD} construction.

4.5.7 Type \texttt{t\_PADIC} (\(p\)-adic numbers). This type has a second codeword \(z[1]\) which contains the following information: the \(p\)-adic precision (the exponent of \(p\) modulo which the \(p\)-adic unit corresponding to \(z\) is defined if \(z\) is not 0), i.e. one less than the number of significant \(p\)-adic digits, and the exponent of \(z\). This information can be handled using the following functions:

\[
\text{long \ precp(GEN } z) \text{ returns the } p\text{-adic precision of } z. \text{ This is 0 if } z = 0.
\]

\[
\text{void \ setprecp(GEN } z, \text{ long } l) \text{ sets the } p\text{-adic precision of } z \text{ to } l.
\]

\[
\text{long \ valp(GEN } z) \text{ returns the } p\text{-adic valuation of } z \text{ (i.e. the exponent). This is defined even if } z \text{ is equal to 0, see Section 1.4.}
\]

\[
\text{void \ setvalp(GEN } z, \text{ long } e) \text{ sets the } p\text{-adic valuation of } z \text{ to } e.
\]

In addition to this codeword, \(z[2]\) points to the prime \(p\), \(z[3]\) points to \(p^{\text{precp}(z)}\), and \(z[4]\) points to \texttt{at\_INT} representing the \(p\)-adic unit associated to \(z\) modulo \(z[3]\) (and to zero if \(z\) is zero). To summarize, if \(z \neq 0\), we have the equality:

\[
z = p^{\text{valp}(z)} \ast (z[4] + O(z[3])), \text{ where } z[3] = O(p^{\text{precp}(z)}).
\]

4.5.8 Type \texttt{t\_QUAD} (quadratic number). \(z[1]\) points to the canonical polynomial \(P\) defining the quadratic field (as output by \texttt{quadpoly}), \(z[2]\) to the “real part” and \(z[3]\) to the “imaginary part”. The latter are of type \texttt{t\_INT}, \texttt{t\_FRAC}, \texttt{t\_INTMOD}, or \texttt{t\_PADIC} and are to be taken as the coefficients of \(z\) with respect to the canonical basis \((1, X)\) or \(\mathbb{Q}[X]/(P(X))\), see Section 1.3.6. Exact complex numbers may be implemented as quadratics, but \texttt{t\_COMPLEX} is in general more versatile (\texttt{t\_REAL} components are allowed) and more efficient.

Operations involving a \texttt{t\_QUAD} and \texttt{t\_COMPLEX} are implemented by converting the \texttt{t\_QUAD} to a \texttt{t\_REAL} (or \texttt{t\_COMPLEX} with \texttt{t\_REAL} components) to the accuracy of the \texttt{t\_COMPLEX}. As a consequence, operations between \texttt{t\_QUAD} and \texttt{exact} \texttt{t\_COMPLEX}s are not allowed.

4.5.9 Type \texttt{t\_POLMOD} (polmod). as for \texttt{t\_INTMODs}, \(z[1]\) points to the modulus, and \(z[2]\) to a polynomial representing the class of \(z\). Both must be of type \texttt{t\_POL} in the same variable, satisfying the inequality \(\deg z[2] < \deg z[1]\). However, \(z[2]\) is allowed to be a simplification of such a polynomial, e.g. a scalar. This is tricky considering the hierarchical structure of the variables; in particular, a polynomial in variable of lesser priority (see Section 2.5.3) than the modulus variable is valid, since it is considered as the constant term of a polynomial of degree 0 in the correct variable. On the other hand a variable of greater priority is not acceptable; see Section 2.5.3 for the problems which may arise.
4.5.10 Type t_POL (polynomial). This type has a second codeword. It contains a “sign”: 0 if the polynomial is equal to 0, and 1 if not (see however the important remark below) and a variable number (e.g. 0 for x, 1 for y, etc...).

These data can be handled with the following macros: signe and setsigne as for t_INT and t_REAL,

`long varn(GEN z)` returns the variable number of the object z,

`void setvarn(GEN z, long v)` sets the variable number of z to v.

The variable numbers encode the relative priorities of variables as discussed in Section 2.5.3. We will give more details in Section 4.6. Note also the function `long gvar(GEN z)` which tries to return a variable number for z, even if z is not a polynomial or power series. The variable number of a scalar type is set by definition equal to NO_VARIABLE, which has lower priority than any other variable number.

The components `z[2], z[3],...,z[lg(z)-1]` point to the coefficients of the polynomial in ascending order, with `z[2]` being the constant term and so on.

For a t_POL of non-zero sign, `degpol, leading_term, constant_term,` return its degree, and a pointer to the leading, resp. constant, coefficient with respect to the main variable. Note that no copy is made on the PARI stack so the returned value is not safe for a basic gerepile call. Applied to any other type than t_POL, the result is unspecified. Those three functions are still defined when the sign is 0, see Section 5.2.7 and Section 9.4.

`long degree(GEN x)` returns the degree of x with respect to its main variable even when x is not a polynomial (a rational function for instance). By convention, the degree of a zero polynomial is −1.

**Important remark.** The leading coefficient of a t_POL may be equal to zero:

- it is not allowed to be an exact rational 0, such as gen_0;
- an exact non-rational 0, like Mod(0,2), is possible for constant polynomials, i.e. of length 3 and no other coefficient: this carries information about the base ring for the polynomial;
- an inexact 0, like 0.E-38 or O(3^5), is always possible. Inexact zeroes do not correspond to an actual 0, but to a very small coefficient according to some metric; we keep them to give information on how much cancellation occurred in previous computations.

A polynomial disobeying any of these rules is an invalid unnormalized object. We advise not to use low-level constructions to build a t_POL coefficient by coefficient, such as

```c
GEN T = cgetg(4, t_POL);
T[1] = evalvarn(0);
gel(T, 2) = x;
gel(T, 3) = y;
```

But if you do and it is not clear whether the result will be normalized, call

```c
GEN normalizepol(GEN x)
```

applied to an unnormalized t_POL x (with all coefficients correctly set except that leading_term(x) might be zero), normalizes x correctly in place and returns x. This functions sets signe (to 0 or 1) properly.
Caveat. A consequence of the remark above is that zero polynomials are characterized by the fact that their sign is 0. It is in general incorrect to check whether $\log(x)$ is 2 or $\deg pol(x) < 0$, although both tests are valid when the coefficient types are under control: for instance, when they are guaranteed to be $t\_INT$s or $t\_FRAC$s. The same remark applies to $t\_SER$s.

4.5.11 Type $t\_SER$ (power series). This type also has a second codeword, which encodes a “sign”, i.e. 0 if the power series is 0, and 1 if not, a variable number as for polynomials, and an exponent. This information can be handled with the following functions: $signe$, $setsigne$, $vvarn$, $setvarn$ as for polynomials, and $valp$, $setvalp$ for the exponent as for $p$-adic numbers. Beware: do not use $expo$ and $setexpo$ on power series.

The coefficients $z[2]$, $z[3]$, ..., $z[\log(z)-1]$ point to the coefficients of $z$ in ascending order. As for polynomials (see remark there), the sign of a $t\_SER$ is 0 if and only all its coefficients are equal to 0. (The leading coefficient cannot be an integer 0.)

Note that the exponent of a power series can be negative, i.e. we are then dealing with a Laurent series (with a finite number of negative terms).

4.5.12 Type $t\_RFRAC$ (rational function). $z[1]$ points to the numerator $n$, and $z[2]$ on the denominator $d$. The denominator must be of type $t\_POL$, with variable of higher priority than the numerator. The numerator $n$ is not an exact 0 and $(n, d) = 1$ (see $\text{gcd}_\text{rfac2}$).

4.5.13 Type $t\_QFR$ (indefinite binary quadratic form). $z[1]$, $z[2]$, $z[3]$ point to the three coefficients of the form and are of type $t\_INT$. $z[4]$ is Shanks’s distance function, and must be of type $t\_REAL$.

4.5.14 Type $t\_QFI$ (definite binary quadratic form). $z[1]$, $z[2]$, $z[3]$ point to the three coefficients of the form. All three are of type $t\_INT$.

4.5.15 Type $t\_VEC$ and $t\_COL$ (vector). $z[1]$, $z[2]$, ..., $z[\log(z)-1]$ point to the components of the vector.

4.5.16 Type $t\_MAT$ (matrix). $z[1]$, $z[2]$,...,$z[\log(z)-1]$ point to the column vectors of $z$, i.e. they must be of type $t\_COL$ and of the same length.

4.5.17 Type $t\_VECSMALL$ (vector of small integers). $z[1]$, $z[2]$,...,$z[\log(z)-1]$ are ordinary signed long integers. This type is used instead of a $t\_VEC$ of $t\_INT$s for efficiency reasons, for instance to implement efficiently permutations, polynomial arithmetic and linear algebra over small finite fields, etc.

4.5.18 Type $t\_STR$ (character string).

c char * $GSTR(z) = (z+1)$ points to the first character of the (NULL-terminated) string.

4.5.19 Type $t\_ERROR$ (error context). This type holds error messages, as well as details about the error, as returned by the exception handling system. The second codeword $z[1]$ contains the error type (an int, as passed to $\text{pari\_err}$). The subsequent words $z[2]$,...,$z[\log(z)-1]$ are GENs containing additional data, depending on the error type.
4.5.20 Type `t_CLOSURE (closure)`. This type holds GP functions and closures, in compiled form. The internal detail of this type is subject to change each time the GP language evolves. Hence we do not describe it here and refer to the Developer’s Guide. However functions to create or to evaluate `t_CLOSURE`s are documented in Section 11.1.

`long closure_arity(GEN C)` returns the arity of the `t_CLOSURE`.

4.5.21 Type `t_LIST (list)`. This type was introduced for specific `gp` use and is rather inefficient compared to a straightforward linked list implementation (it requires more memory, as well as many unnecessary copies). Hence we do not describe it here and refer to the Developer’s Guide.

**Implementation note.** For the types including an exponent (or a valuation), we actually store a biased non-negative exponent (bit-ORing the biased exponent to the codeword), obtained by adding a constant to the true exponent: either `HIGHEXPOBIT` (for `t_REAL`) or `HIGHVALPBIT` (for `t_PADIC` and `t_SER`). Of course, this is encapsulated by the exponent/valuation-handling macros and needs not concern the library user.

4.6 PARI variables.

4.6.1 Multivariate objects.

We now consider variables and formal computations, and give the technical details corresponding to the general discussion in Section 2.5.3. As we have seen in Section 4.5, the codewords for types `t_POL` and `t_SER` encode a “variable number”. This is an integer, ranging from 0 to `MAXVARN`. Relative priorities may be ascertained using

`int varncmp(long v, long w)`

which is > 0, = 0, < 0 whenever `v` has lower, resp. same, resp. higher priority than `w`.

The way an object is considered in formal computations depends entirely on its “principal variable number” which is given by the function

`long gvar(GEN z)`

which returns a variable number for `z`, even if `z` is not a polynomial or power series. The variable number of a scalar type is set by definition equal to `NO_VARIABLE` which has lower priority than any valid variable number. The variable number of a recursive type which is not a polynomial or power series is the variable number with highest priority among its components. But for polynomials and power series only the “outermost” number counts (we directly access `varn(x)` in the codewords): the representation is not symmetrical at all.

Under `gp`, one needs not worry too much since the interpreter defines the variables as it sees them* and do the right thing with the polynomials produced (however, have a look at the remark in Section 2.3.9).

But in library mode, they are tricky objects if you intend to build polynomials yourself (and not just let PARI functions produce them, which is less efficient). For instance, it does not make

* The first time a given identifier is read by the GP parser a new variable is created, and it is assigned a strictly lower priority than any variable in use at this point. On startup, before any user input has taken place, `x` is defined in this way and has initially maximal priority (and variable number 0).
sense to have a variable number occur in the components of a polynomial whose main variable has a lower priority, even though PARI cannot prevent you from doing it; see Section 2.5.3 for a discussion of possible problems in a similar situation.

4.6.2 Creating variables. A basic difficulty is to “create” a variable. Some initializations are needed before you can use a given integer \(v\) as a variable number.

Initially, this is done for 0 (the variable \(x\) under \(\text{gp}\)), and \(\text{MAXVARN}\), which is there to address the need for a “temporary” new variable in library mode and cannot be input under \(\text{gp}\). No documented library function can create from scratch an object involving \(\text{MAXVARN}\) (of course, if the operands originally involve \(\text{MAXVARN}\), the function abides). We call the latter type a “temporary variable”. The regular variables meant to be used in regular objects, are called “user variables”.

4.6.2.1 User variables. When the program starts, \(x\) is the only user variable (number 0). To define new ones, use

```c
long fetch_user_var(char *s): inspects the user variable whose name is the string pointed to by s, creating it if needed, and returns its variable number.
```

```c
long v = fetch_user_var("y");
GEN gy = pol_x(v);
```

The function raises an exception if the name is already in use for an installed or built-in function, or an alias.

Caveat. You can use \(\text{gp_read_str}\) (see Section 4.7.1) to execute a GP command and create GP variables on the fly as needed:

```c
GEN gy = gp_read_str("'y"); /* returns \(\text{pol}_x(v)\), for some \(v\) */
long v = varn(gy);
```

But please note the quote ‘\(y\) in the above. Using \(\text{gp_read_str}("y")\) might work, but is dangerous, especially when programming functions to be used under \(\text{gp}\). The latter reads the value of \(y\), as currently known by the \(\text{gp}\) interpreter, possibly creating it in the process. But if \(y\) has been modified by previous \(\text{gp}\) commands (e.g. \(y = 1\)), then the value of \(gy\) is not what you expected it to be and corresponds instead to the current value of the \(\text{gp}\) variable (e.g. \(\text{gen}_1\)).

```c
GEN fetch_var_value(long v) returns a shallow copy of the current value of the variable numbered \(v\). Returns NULL if that variable number is unknown to the interpreter, e.g. it is a user variable. Note that this may not be the same as \(\text{pol}_x(v)\) if assignments have been performed in the interpreter.
```

4.6.2.2 Temporary variables. \(\text{MAXVARN}\) is available, but is better left to PARI internal functions (some of which do not check that \(\text{MAXVARN}\) is free for them to use, which can be considered a bug). You can create more temporary variables using

```c
long fetch_var()
```

This returns a variable number which is guaranteed to be unused by the library at the time you get it and as long as you do not delete it (we will see how to do that shortly). This has higher priority than any temporary variable produced so far (\(\text{MAXVARN}\) is assumed to be the first such). After the statement \(v = \text{fetch_var}()\), you can use \(\text{pol}_1(v)\) and \(\text{pol}_x(v)\). The variables created in this way have no identifier assigned to them though, and are printed as #<number>, except for \(\text{MAXVARN}\) which is printed as #. You can assign a name to a temporary variable, after creating it, by calling the function
void name_var(long n, char *s)

after which the output machinery will use the name s to represent the variable number n. The
GP parser will not recognize it by that name, however, and calling this on a variable known to gp
raises an error. Temporary variables are meant to be used as free variables, and you should never
assign values or functions to them as you would do with variables under gp. For that, you need a
user variable.

All objects created by fetch_var are on the heap and not on the stack, thus they are not
subject to standard garbage collecting (they are not destroyed by a gerepile or avma = ltop
statement). When you do not need a variable number anymore, you can delete it using

long delete_var()

which deletes the latest temporary variable created and returns the variable number of the previous
one (or simply returns 0 if you try, in vain, to delete MAXVARN). Of course you should make sure
that the deleted variable does not appear anywhere in the objects you use later on. Here is an
example:

long first = fetch_var();
long n1 = fetch_var();
long n2 = fetch_var(); /* prepare three variables for internal use */

/* delete all variables before leaving */
do { num = delete_var(); } while (num && num <= first);

The (dangerous) statement

while (delete_var()) /* empty */;

removes all temporary variables in use, except MAXVARN which cannot be deleted.

4.7 Input and output.

Two important aspects have not yet been explained which are specific to library mode: input and
output of PARI objects.

4.7.1 Input.

For input, PARI provides a powerful high level function which enables you to input your objects as
if you were under gp. In fact, it is essentially the GP syntactical parser, hence you can use it not
only for input but for (most) computations that you can do under gp. It has the following syntax:

GEN gp_read_str(const char *s)

Note that gp’s metacommands are not recognized.
Note. The obsolete form

```c
GEN readseq(char *t)
```

still exists for backward compatibility (assumes filtered input, without spaces or comments). Don’t use it.

To read a GEN from a file, you can use the simpler interface

```c
GEN gp_read_stream(FILE *file)
```

which reads a character string of arbitrary length from the stream `file` (up to the first complete expression sequence), applies `gp_read_str` to it, and returns the resulting GEN. This way, you do not have to worry about allocating buffers to hold the string. To interactively input an expression, use `gp_read_stream(stdin)`.

Finally, you can read in a whole file, as in GP’s `read` statement

```c
GEN gp_read_file(char *name)
```

As usual, the return value is that of the last non-empty expression evaluated. There is one technical exception: if `name` is a binary file (from `writebin`) containing more than one object, a `t_VEC` containing them all is returned. This is because binary objects bypass the parser, hence reading them has no useful side effect.

### 4.7.2 Output to screen or file, output to string.

General output functions return nothing but print a character string as a side effect. Low level routines are available to write on PARI output stream `pari_outfile` (stdout by default):

- `void pari_putchar(char c)`: write character `c` to the output stream.
- `void pari_puts(char *s)`: write `s` to the output stream.
- `void pari_flush()`: flush output stream; most streams are buffered by default, this command makes sure that all characters output so are actually written.
- `void pari_printf(const char *fmt, ...)`: the most versatile such function. `fmt` is a character string similar to the one `printf` uses. In there, `%` characters have a special meaning, and describe how to print the remaining operands. In addition to the standard format types (see the GP function `printf`), you can use the length modifier `P` (for PARI of course!) to specify that an argument is a GEN. For instance, the following are valid conversions for a GEN argument

```
%P  convert to char* (will print an arbitrary GEN)
%P.10s convert to char*, truncated to 10 chars
%P.2f convert to floating point format with 2 decimals
%P4d convert to integer, field width at least 4
```

```c
pari_printf("x[%d] = %Ps is not invertible!\n", i, gel(x,i));
```

Here `i` is an `int`, `x` a GEN which is not a leaf (presumably a vector, or a polynomial) and this would insert the value of its `i`-th GEN component: `gel(x,i)`.

Simple but useful variants to `pari_printf` are

```c
void output(GEN x) prints x in raw format, followed by a newline and a buffer flush. This is more
```

or less equivalent to

```c
pari_printf("%Ps\n", x);
```
void outmat(GEN x) as above except if $x$ is a t_MAT, in which case a multi-line display is used to display the matrix. This is prettier for small dimensions, but quickly becomes unreadable and cannot be pasted and reused for input. If all entries of $x$ are small integers, you may use the recursive features of \texttt{\%Pd} and obtain the same (or better) effect with

\begin{verbatim}
pari_printf("\%Pd\n", x);
pari_flush();
\end{verbatim}

A variant like \texttt{\%5Pd} would improve alignment by imposing 5 chars for each coefficient. Similarly if all entries are to be converted to floats, a format like \texttt{\%5.1Pf} could be useful.

These functions write on (PARI’s idea of) standard output, and must be used if you want your functions to interact nicely with \texttt{gp}. In most programs, this is not a concern and it is more flexible to write to an explicit \texttt{FILE*}, or to recover a character string:

\begin{verbatim}
void pari_fprintf(FILE *file, const char *fmt, ...)
\end{verbatim}
writes the remaining arguments to stream \texttt{file} according to the format specification \texttt{fmt}.

\begin{verbatim}
char* pari_sprintf(const char *fmt, ...)
\end{verbatim}
produces a string from the remaining arguments, according to the PARI format \texttt{fmt} (see \texttt{printf}). This is the \texttt{libpari} equivalent of \texttt{Strprintf}, and returns a \texttt{malloc}’ed string, which must be freed by the caller. Note that contrary to the analogous \texttt{sprintf} in the \texttt{libc} you do not provide a buffer (leading to all kinds of buffer overflow concerns); the function provided is actually closer to the GNU extension \texttt{asprintf}, although the latter has a different interface.

Simple variants of \texttt{pari_sprintf} convert a \texttt{GEN} to a \texttt{malloc}’ed ASCII string, which you must still \texttt{free} after use:

\begin{verbatim}
char* GENtostr(GEN x), using the current default output format (\texttt{prettymat} by default).
char* GENtoTeXstr(GEN x), suitable for inclusion in a \texttt{\LaTeX} file.
\end{verbatim}

Note that we have \texttt{va_list} analogs of the functions of \texttt{printf} type seen so far:

\begin{verbatim}
void pari_vprintf(const char *fmt, va_list ap)
void pari_vfprintf(FILE *file, const char *fmt, va_list ap)
char* pari_vsprintf(const char *fmt, va_list ap)
\end{verbatim}

4.7.3 Errors.

If you want your functions to issue error messages, you can use the general error handling routine \texttt{pari_err}. The basic syntax is

\begin{verbatim}
pari_err(e_MISC, "error message");
\end{verbatim}

This prints the corresponding error message and exit the program (in library mode; go back to the \texttt{gp} prompt otherwise). You can also use it in the more versatile guise

\begin{verbatim}
pari_err(e_MISC, format, ...);
\end{verbatim}

where \texttt{format} describes the format to use to write the remaining operands, as in the \texttt{pari_printf} function. For instance:

\begin{verbatim}
pari_err(e_MISC, "x[\%d] = \%Ps is not invertible!", i, gel(x,i));
\end{verbatim}
The simple syntax seen above is just a special case with a constant format and no remaining arguments. The general syntax is

\[
\text{void pari_err(numerr,...)}
\]

where \text{numerr} is a codeword which specifies the error class and what to do with the remaining arguments and what message to print. For instance, if \( x \) is a \text{GEN} with internal type \text{t_STR}, say, \text{pari_err(e_TYPE,"extgcd", \( x \))} prints the message:

\[
*** \text{incorrect type in extgcd (t_STR)},
\]

See Section 10.4 for details. In the libpari code itself, the general-purpose \text{e_MISC} is used sparingly: it is so flexible that the corresponding error contexts (\text{t_ERROR}) become hard to use reliably. Other more rigid error types are generally more useful: for instance the error context associated to the \text{e_TYPE} exception above is precisely documented and contains "extgcd" and \( x \) (not only its type) as readily available components.

4.7.4 Warnings.

To issue a warning, use

\[
\text{void pari_warn(warnerr,...)}
\]

In that case, of course, we do not abort the computation, just print the requested message and go on. The basic example is

\[
\text{pari_warn(warner, "Strategy 1 failed. Trying strategy 2")}
\]

which is the exact equivalent of \text{pari_err(e_MISC,...)} except that you certainly do not want to stop the program at this point, just inform the user that something important has occurred; in particular, this output would be suitably highlighted under \text{gp}, whereas a simple \text{printf} would not.

The valid \text{warning} keywords are \text{warner} (general), \text{warnprec} (increasing precision), \text{warnmem} (garbage collecting) and \text{warnfile} (error in file operation), used as follows:

\[
\text{pari_warn(warnprec, "bnfinit", newprec);} \\
\text{pari_warn(warnmem, "bnfinit");} \\
\text{pari_warn(warnfile, "close", "afile");} /* error when closing "afile" */
\]

4.7.5 Debugging output.

For debugging output, you can use the standard output functions, \text{output} and \text{pari_printf} mainly. Corresponding to the \text{gp} metacommand \( \backslash x \), you can also output the hexadecimal tree associated to an object:

\[
\text{void dbgGEN(GEN x, long nb = -1), displays the recursive structure of } x. \text{ If } nb = -1, \text{ the full structure is printed, otherwise the leaves (non-recursive components) are truncated to } nb \text{ words.}
\]

The function \text{output} is vital under debuggers, since none of them knows how to print PARI objects by default. Seasoned PARI developers add the following \text{gdb} macro to their \text{.gdbinit}:

\[
\text{define i} \\
\text{\quad call output((GEN)$arg0)} \\
\text{end}
\]

Typing \text{i \ } x \text{ at a breakpoint in \text{gdb}} then prints the value of the \text{GEN} \( x \) (provided the optimizer has not put it into a register, but it is rarely a good idea to debug optimized code).
The global variables `DEBUGLEVEL` and `DEBUGMEM` (corresponding to the default `debug` and `debugmem`, see Section 2.12) are used throughout the PARI code to govern the amount of diagnostic and debugging output, depending on their values. You can use them to debug your own functions, especially if you install the latter under `gp` (see Section 3.12.22).

**void dbg_pari_heap(void)** print debugging statements about the PARI stack, heap, and number of variables used. Corresponds to `\s` under `gp`.

**4.7.6 Timers and timing output.**

To handle timings in a reentrant way, PARI defines a dedicated data type, `pari_timer`, together with the following methods:

**void timer_start(pari_timer *T)** start (or reset) a timer.

**long timer_delay(pari_timer *T)** returns the number of milliseconds elapsed since the timer was last reset. Resets the timer as a side effect.

**long timer_get(pari_timer *T)** returns the number of milliseconds elapsed since the timer was last reset. Does not reset the timer.

**long timer_printf(pari_timer *T, char *format,...)** This diagnostics function is equivalent to the following code

```c
err_printf("Time ")
... prints remaining arguments according to format ...
err_printf(": %ld", timer_delay(T));
```

Resets the timer as a side effect.

They are used as follows:

```c
pari_timer T;
timer_start(&T); /* initialize timer */
...
printf("Total time: %ldms\n", timer_delay(&T));
or
pari_timer T;
timer_start(&T);
for (i = 1; i < 10; i++) {
...
timer_printf(&T, "for i = %ld (L[i] = %Ps)", i, gel(L,i));
}
```

The following functions provided the same functionality, in a non-reentrant way, and are now deprecated.

**long timer(void)**

**long timer2(void)**

**void msgtimer(const char *format, ...)**

The following function implements `gp`'s timer and should not be used in libpari programs:

**long gettime(void)** equivalent to `timer_delay(T)` associated to a private timer `T`. 39
4.8 Iterators, Numerical integration, Sums, Products.

4.8.1 Iterators. Since it is easier to program directly simple loops in library mode, some GP
iterators are mainly useful for GP programming. Here are the others:

- **fordiv** is a trivial iteration over a list produced by divisors.

- **forell** and **forsubgroup** are currently not implemented as an iterator but as a procedure
with callbacks.

  ```c
  void forell(void *E, long fun(void*, GEN), GEN a, GEN b)
  ```

  goes through the same curves as `forell(ell, a, b)` calling `fun(E, ell)` for each
  curve `ell`, stopping if `fun` returns a non-zero value.

  ```c
  void forsubgroup(void *E, long fun(void*, GEN), GEN G, GEN B)
  ```

  goes through the same subgroups as `forsubgroup(H = G, B)` calling `fun(E, H)`
  for each subgroup `H`, stopping if `fun` returns a non-zero value.

- **forprime**, for which we refer you to the next subsection.

- **forcomposite**, we provide an iterator over composite integers:

  ```c
  int forcomposite(forcomposite_t *T, GEN a, GEN b)
  ```

  initialize an iterator `T` over composite integers in `[a, b]`; over composites \( \geq a \) if `b`
  is `NULL`. Return 0 if the range is known to be empty from
  the start (as if `b < a` or `b < 0`), and return 1 otherwise.

  ```c
  GEN forcomposite_next(forcomposite_t *T)
  ```

  returns the next composite in the range, assuming
  that `T` was initialized by `forcomposite_init`.

- **forvec**, for which we provide a convenient iterator. To initialize the analog of `forvec(X = v, ..., flag)`, call

  ```c
  int forvec_init(forvec_t *T, GEN v, long flag)
  ```

  initialize an iterator `T` over the vectors generated by `forvec(X = v, ..., flag)`. This
  returns 0 if this vector list is empty, and 1 otherwise.

  ```c
  GEN forvec_next(forvec_t *T)
  ```

  returns the next element in the `forvec` sequence, or `NULL` if we are done. The return value
  must be used immediately or copied since the next call to the iterator
  destroys it: the relevant vector is updated in place. The iterator works hard to not use up PARI
  stack, and is more efficient when all lower bounds in the initialization vector
  is integers. In that case, the cost is linear in the number of tuples enumerated, and you can expect to run over more
  than \(10^9\) tuples per minute. If speed is critical and all integers involved would fit in
  `C longs`, write a simple direct backtracking algorithm yourself.

- **forpart** is a variant of `forvec` which iterates over partitions. See the documentation of the
  `forpart` GP function for details. This function is available as a loop with callbacks:

  ```c
  void forpart(void *data, long (*call)(void*,GEN), long k, GEN a, GEN n)
  ```

  It is also available as an iterator:

  ```c
  void forpart_init(forpart_t *T, long k, GEN a, GEN n)
  ```

  initializes an iterator over the partitions of `k`, with length restricted by `n`, and components restricted by `a`, either of which can be set to
  `NULL` to run without restriction.

  ```c
  GEN forpart_next(forpart_t *T)
  ```

  returns the next partition, or `NULL` when all partitions have been exhausted.
GEN forpart_prev(forpart_t *T) returns the previous partition, or NULL when all partitions have been exhausted.

You may not mix calls to forpart_next and forpart_prev: the first one called determines the ordering used to iterate over the partitions; you can not go back since the forpart_t structure is used in incompatible ways.

4.8.2 Iterating over primes.

The library provides a high-level iterator, which stores its (private) data in a struct forprime_t and runs over arbitrary ranges of primes, without ever overflowing.

The iterator has two flavors, one providing the successive primes as ulong, the other as GEN. They are initialized as follows, where we expect to run over primes $a \leq p \leq b$:

- int forprime_init(forprime_t *T, GEN a, GEN b) for the GEN variant, where $b = \text{NULL}$ means $+\infty$.
- int u_forprime_init(forprime_t *T, ulong a, ulong b) for the ulong variant, where $b = \text{ULONG_MAX}$ means we will run through all primes representable in a ulong type.

Both variants return 1 on success, and 0 if the iterator would run over an empty interval (if $a > b$, for instance). They allocate the forprime_t data structure on the PARI stack.

The successive primes are then obtained using

- GEN forprime_next(forprime_t *T), returns NULL if no more primes are available in the interval.
- ulong u_forprime_next(forprime_t *T), returns 0 if no more primes are available in the interval.

These two functions leave alone the PARI stack, and write their state information in the preallocated forprime_t struct. The typical usage is thus:

```plaintext
def
forprime_t T;
def
GEN p;
def
pari_sp av = avma, av2;
forprime_init(&T, gen_2, stoi(1000));
av2 = avma;
while ( (p = forprime_next(&T)) )
{
    ...
def
    if ( prime_is_OK(p) ) break;
def
    avma = av2; /* delete garbage accumulated in this iteration */
}
def
avma = av; /* delete all */
```

Of course, the final avma = av could be replaced by a gerepile call. Beware that swapping the av2 = avma and forprime_init call would be incorrect: the first avma = av2 would delete the forprime_t structure!
4.8.3 Numerical analysis.

Numerical routines code a function (to be integrated, summed, zeroed, etc.) with two parameters named

```c
void *E;
GEN (*eval)(void*, GEN)
```

The second is meant to contain all auxiliary data needed by your function. The first is such that `eval(x, E)` returns your function evaluated at `x`. For instance, one may code the family of functions \( f_t(x) = (x + t)^2 \) via

```c
GEN fun(void *t, GEN x) { return gsqr(gadd(x, (GEN)t)); }
```

One can then integrate \( f_1 \) between \( a \) and \( b \) with the call

```c
intnum((void*)stoi(1), &fun, a, b, NULL, prec);
```

Since you can set \( E \) to a pointer to any `struct` (typecast to `void*`) the above mechanism handles arbitrary functions. For simple functions without extra parameters, you may set \( E = NULL \) and ignore that argument in your function definition.

4.9 Catching exceptions.

4.9.1 Basic use.

PARI provides a mechanism to trap exceptions generated via `pari_err` using the `pari_CATCH` construction. The basic usage is as follows

```c
pari_CATCH(err_code) {
    recovery branch
}
pari_TRY {
    main branch
}
pari_ENDCATCH
```

This fragment executes the main branch, then the recovery branch if exception `err_code` is thrown, e.g. `e_TYPE`. See Section 10.4 for the description of all error classes. The special error code `CATCH_ALL` is available to catch all errors.

One can replace the `pari_TRY` keyword by `pari_RETRY`, in which case once the recovery branch is run, we run the main branch again, still catching the same exceptions.
Restrictions.

- Such constructs can be nested without adverse effect, the innermost handler catching the exception.
- It is valid to leave either branch using `pari_err`.
- It is invalid to use C flow control instructions (`break`, `continue`, `return`) to directly leave either branch without seeing the `pari_ENDCATCH` keyword. This would leave an invalid structure in the exception handler stack, and the next exception would crash.
- In order to leave using `break`, `continue` or `return`, one must precede the keyword by a call to

```c
void pari_CATCH_reset() disable the current handler, allowing to leave without adverse effect.
```

4.9.2 Advanced use.

In the recovery branch, the exception context can be examined via the following helper routines:

- `GEN pari_err_last()` returns the exception context, as a `t_ERROR`. The exception `E` returned by `pari_err_last` can be rethrown, using

```c
pari_err(0, E);
```
- `long err_get_num(GEN E)` returns the error symbolic name. E.g `e_TYPE`.
- `GEN err_get_compo(GEN E, long i)` returns `i`-th component, as documented in Section 10.4.

For instance

```c
pari_CATCH(CATCH_ALL) { /* catch everything */
    GEN x, E = pari_err_last();
    long code = err_get_num(E);
    if (code != e_INV) pari_err(0, E); /* unexpected error, rethrow */
    x = err_get_compo(E, 2);
    /* e_INV has two components, 1: function name 2: non-invertible x */
    if (typ(x) != t_INTMOD) pari_err(0, E); /* unexpected type, rethrow */
    pari_CATCH_reset();
    return x; /* leave */
...}
pari_TRY {
    /* main branch */
}
pari_ENDCATCH
```
4.10 A complete program.

Now that the preliminaries are out of the way, the best way to learn how to use the library mode is to study a detailed example. We want to write a program which computes the gcd of two integers, together with the Bezout coefficients. We shall use the standard quadratic algorithm which is not optimal but is not too far from the one used in the PARI function `bezout`.

Let \( x, y \) two integers and initially \( \left( \begin{array}{cc} s_x & s_y \\ t_x & t_y \end{array} \right) = \left( \begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right) \), so that

\[
\left( \begin{array}{cc} s_x & s_y \\ t_x & t_y \end{array} \right) \left( \begin{array}{c} x \\ y \end{array} \right) = \left( \begin{array}{c} x \\ y \end{array} \right).
\]

To apply the ordinary Euclidean algorithm to the right hand side, multiply the system from the left by \( \left( \begin{array}{cc} 0 & 1 \\ 1 & -q \end{array} \right) \), with \( q = \text{floor}(x/y) \). Iterate until \( y = 0 \) in the right hand side, then the first line of the system reads

\[ s_xx + s_yy = \gcd(x, y). \]

In practice, there is no need to update \( s_y \) and \( t_y \) since \( \gcd(x, y) \) and \( s_x \) are enough to recover \( s_y \). The following program is now straightforward. A couple of new functions appear in there, whose description can be found in the technical reference manual in Chapter 5, but whose meaning should be clear from their name and the context.

This program can be found in `examples/extgcd.c` together with a proper `Makefile`. You may ignore the first comment

```c
/*
  GP;install("extgcd", "GG&", "gcdex", "/libextgcd.so");
*/
```

which instruments the program so that `gp2c-run extgcd.c` can import the `extgcd()` routine into an instance of the `gp` interpreter (under the name `gcdex`). See the `gp2c` manual for details.
#include <pari/pari.h>

/*
GP;install("extgcd", "GG&v", "gcdex", ".m/extgcd.so");
*/

/* return d = gcd(a,b), sets u, v such that au + bv = gcd(a,b) */
GEN
extgcd(GEN A, GEN B, GEN *U, GEN *V)
{
    pari_sp av = avma;
    GEN ux = gen_1, vx = gen_0, a = A, b = B;
    if (typ(a) != t_INT) pari_err_TYPE("extgcd",a);
    if (typ(b) != t_INT) pari_err_TYPE("extgcd",b);
    if (signe(a) < 0) { a = negi(a); ux = negi(ux); }
    while (!gequal0(b))
    {
        GEN r, q = dvmdii(a, b, &r), v = vx;
        vx = subii(ux, mulii(q, vx));
        ux = v; a = b; b = r;
    }
    *U = ux;
    *V = diviiexact( subii(a, mulii(A,ux)), B );
    gerepileall(av, 3, &a, U, V); return a;
}

int
main()
{
    GEN x, y, d, u, v;
    pari_init(1000000,2);
    printf("x = "); x = gp_read_stream(stdin);
    printf("y = "); y = gp_read_stream(stdin);
    d = extgcd(x, y, &u, &v);
    pari_printf("gcd = %Ps\nu = %Ps\nv = %Ps
", d, u, v);
    pari_close();
    return 0;
}

For simplicity, the inner loop does not include any garbage collection, hence memory use is quadratic in the size of the inputs instead of linear. Here is a better version of that loop:

    pari_sp av = avma, lim = stack_lim(av,1);
    ...
    while (!gequal0(b))
    {
        GEN r, q = dvmdii(a, b, &r), v = vx;
        vx = subii(ux, mulii(q, vx));
        ux = v; a = b; b = r;
        if (low_stack(lim, stack_lim(av,1)))
            gerepileall(av, 4, &a, &b, &ux, &vx);
"
Chapter 5:
Technical Reference Guide: the basics

In the following chapters, we describe all public low-level functions of the PARI library. These include specialized functions for handling all the PARI types. Simple higher level functions, such as arithmetic or transcendental functions, are described in Chapter 3 of the GP user’s manual; we will eventually see more general or flexible versions in the chapters to come. A general introduction to the major concepts of PARI programming can be found in Chapter 4, which you should really read first.

We shall now study specialized functions, more efficient than the library wrappers, but sloppier on argument checking and damage control; besides speed, their main advantage is to give finer control about the inner workings of generic routines, offering more options to the programmer.

Important advice. Generic routines eventually call lower level functions. Optimize your algorithms first, not overhead and conversion costs between PARI routines. For generic operations, use generic routines first; do not waste time looking for the most specialized one available unless you identify a genuine bottleneck, or you need some special behavior the generic routine does not offer. The PARI source code is part of the documentation; look for inspiration there.

The type long denotes a BITS_IN_LONG-bit signed long integer (32 or 64 bits). The type ulong is defined as unsigned long. The word stack always refer to the PARI stack, allocated through an initial pari_init call. Refer to Chapters 1–2 and 4 for general background.

We shall often refer to the notion of shallow function, which means that some components of the result may point to components of the input, which is more efficient than a deep copy (full recursive copy of the object tree). Such outputs are not suitable for gerepileupto and particular care must be taken when garbage collecting objects which have been input to shallow functions: corresponding outputs also become invalid and should no longer be accessed.

A function is not stack clean if it leaves intermediate data on the stack besides its output, for efficiency reasons.

5.1 Initializing the library.

The following functions enable you to start using the PARI functions in a program, and cleanup without exiting the whole program.

5.1.1 General purpose.

void pari_init(size_t size, ulong maxprime) initialize the library, with a stack of size bytes and a prime table up to the maximum of maxprime and 2^{16}. Unless otherwise mentioned, no PARI function will function properly before such an initialization.

void pari_close(void) stop using the library (assuming it was initialized with pari_init) and frees all allocated objects.
5.1.2 Technical functions.

void pari_init_opts(size_t size, ulong maxprime, ulong opts) as pari_init, more flexible. opts is a mask of flags among the following:

INIT_JMPm: install PARI error handler. When an exception is raised, the program is terminated with exit(1).

INIT_SIGm: install PARI signal handler.

INIT_DFTm: initialize the GP_DATA environment structure. This one must be enabled once. If you close pari, then restart it, you need not reinitialize GP_DATA; if you do not, then old values are restored.

INIT_noPRIMEm: do not compute the prime table (ignore the maxprime argument). The user must call initprimetable later.

INIT_noIMTm: (technical, see pari_mt_init in the Developer’s Guide for detail). Do not call pari_mt_init to initialize the multi-thread engine. If this flag is set, pari_mt_init() will need to be called manually. See examples/pari-mt.c for an example.

void pari_close_opts(ulong init_opts) as pari_close, for a library initialized with a mask of options using pari_init_opts. opts is a mask of flags among

INIT_SIGm: restore SIG_DFL default action for signals tampered with by PARI signal handler.

INIT_DFTm: frees the GP_DATA environment structure.

INIT_noIMTm: (technical, see pari_mt_init in the Developer’s Guide for detail). Do not call pari_mt_close to close the multi-thread engine.

void pari_sig_init(void (*f)(int)) install the signal handler f (see signal(2)): the signals SIGBUS, SIGFPE, SIGINT, SIGBREAK, SIGPIPE and SIGSEGV are concerned.

void pari_stackcheck_init(void *stackbase) controls the system stack exhaustion checking code in the GP interpreter. This should be used when the system stack base address change or when the address seen by pari_init is too far from the base address. If stackbase is NULL, disable the check, else set the base address to stackbase. It is normally used this way

int thread_start (...) {
    long first_item_on_the_stack;
    ...
    pari_stackcheck_init(&first_item_on_the_stack);
}

int pari_daemon(void) fork a PARI daemon, detaching from the main process group. The function returns 1 in the parent, and 0 in the forked son.
5.1.3 Notions specific to the GP interpreter.

An **entree** is the generic object associated to an identifier (a name) in GP’s interpreter, be it a built-in or user function, or a variable. For a function, it has at least the following fields:

- **char *name**: the name under which the interpreter knows us.
- **void *value**: a pointer to the C function to call.
- **long menu**: an integer from 1 to 11 (to which group of function help do we belong).
- **char *code**: the prototype code.
- **char *help**: the help text for the function.

A routine in GP is described to the analyzer by an **entree** structure. Built-in PARI routines are grouped in **modules**, which are arrays of **entree** structs, the last of which satisfy **name** = **NULL** (sentinel).

There are currently five modules in PARI/GP: general functions (**functions_basic**, known to **libpari**), gp-specific functions (**functions_gp**), gp-specific highlevel functions (**functions_highlevel**), and two modules of obsolete functions. The function **pari.init** initializes the interpreter and declares all symbols in **functions_basic**. You may declare further functions on a case by case basis or as a whole module using

```c
void pari_add_function(entree *ep) adds a single routine to the table of symbols in the interpreter. It assumes pari_init has been called.

void pari_add_module(entree *mod) adds all the routines in module mod to the table of symbols in the interpreter. It assumes pari_init has been called.
```

For instance, gp implements a number of private routines, which it adds to the default set via the calls

```c
pari_add_module(functions_gp);
pari_add_module(functions_highlevel);
```

```c
void pari_add_oldmodule(entree *mod) adds all the routines in module mod to the table of symbols in the interpreter when running in ”PARI 1.xx compatible” mode (see default(compatible)) . It assumes that pari_init has been called.
```

A GP **default** is likewise associated to a helper routine, that is run when the value is consulted, or changed by **default0** or **setdefault**. Such routines are grouped into modules: **functions_default** containing all defaults that make sense in libpari context, **functions_gp_rl_default** containing defaults that are gp-specific and do not make sense unless we use **libreadline**, and **functions_gp_default** containing all other gp-specific defaults.

```c
void pari_add_defaults_module(entree *mod) adds all the defaults in module mod to the interpreter. It assumes that pari_init has been called. From this point on, all defaults in module mod are known to setdefault and friends.
```
5.1.4 Public callbacks.

The gp calculator associates elaborate functions (for instance the break loop handler) to the following callbacks, and so can you:

void (*cb_pari_ask_confirm)(const char *s) initialized to NULL. Called with argument s whenever PARI wants confirmation for action s, for instance in secure mode.

extern int (*cb_pari_handle_exception)(long) initialized to NULL. If not NULL, this routine is called with argument −1 on SIGINT, and argument err on error err. If it returns a non-zero value, the error or signal handler returns, in effect further ignoring the error or signal, otherwise it raises a fatal error.

void (*cb_pari_sigint)(void). Function called when we receive SIGINT. By default, raises pari_err(e_MISC, "user interrupt");

extern void (*cb_pari_err_recover)(long) initialized to NULL. If not NULL, this routine is called just before PARI cleans up from an error. It is not required to return. The error number is passed as argument, unless the PARI stack has been destroyed (allocatemem), in which case −1 is passed.

extern void (*cb_pari_err_recover)(long) initialized to pari_exit(). This callback must not return. It is called after PARI has cleaned-up from an error. The error number is passed as argument, unless the PARI stack has been destroyed, in which case it is called with argument −1.

int (*cb_pari_whatnow)(PariOUT *out, const char *s, int flag) initialized to NULL. If not NULL, must check whether s existed in older versions of pari (the gp callback checks against pari-1.39.15). All output must be done via out methods.

- flag = 0: should print verbosely the answer, including help text if available.
- flag = 1: must return 0 if the function did not change, and a non-0 result otherwise. May print a help message.

5.1.5 Configuration variables.

pari_library_path: If set, It should be a path to the libpari library. It is used by the function gpinstall to locate the PARI library when searching for symbols. This should only be useful on Windows.

Utility function.

void pari_ask_confirm(const char *s) raise an error if the callback cb_pari_ask_confirm is NULL. Otherwise calls cb_pari_ask_confirm(s);

5.1.6 Saving and restoring the GP context.

void gp_context_save(struct gp_context* rec) save the current GP context.

void gp_context_restore(struct gp_context* rec) restore a GP context. The new context must be an ancestor of the current context.
5.1.7 GP history.

These functions allow to control the GP history (the \% operator).

\begin{verbatim}
void pari_add_hist(GEN x, long t) adds x as the last history entry; t is the time used to compute it.
GEN pari_get_hist(long p), if p > 0 returns entry of index p (i.e. \%p), else returns entry of index n + p where n is the index of the last entry (used for \%, \%'t, \%'t, etc.).
long pari_get_histtime(long p) as pari_get_hist, returning the time used to compute the history entry, instead of the entry itself.
ulong pari_nb_hist(void) return the index of the last entry.
\end{verbatim}

5.2 Handling GENs.

Almost all these functions are either macros or inlined. Unless mentioned otherwise, they do not evaluate their arguments twice. Most of them are specific to a set of types, although no consistency checks are made: e.g. one may access the sign of a t_PADIC, but the result is meaningless.

5.2.1 Allocation.

\begin{verbatim}
GEN cgetg(long l, long t) allocates (the root of) a GEN of type t and length l. Sets z[0].
GEN cgeti(long l) allocates a t_INT of length l (including the 2 codewords). Sets z[0] only.
GEN cgetr(long l) allocates a t_REAL of length l (including the 2 codewords). Sets z[0] only.
GEN cgetc(long prec) allocates a t_COMPLEX whose real and imaginary parts are t_REALs of length prec.
GEN cgetg_copy(GEN x, long *lx) fast version of cgetg: allocate a GEN with the same type and length as x, setting *lx to lg(x) as a side-effect. (Only sets the first codeword.) This is a little faster than cgetg since we may reuse the bitmask in x[0] instead of recomputing it, and we do not need to check that the length does not overflow the possibilities of the implementation (since an object with that length already exists). Note that cgetg with arguments known at compile time, as in
cgetg(3, t_INTMOD)
will be even faster since the compiler will directly perform all computations and checks.
GEN vectrunc_init(long l) perform cgetg(1,t_VEC), then set the length to 1 and return the result. This is used to implement vectors whose final length is easily bounded at creation time, that we intend to fill gradually using:
\begin{verbatim}
void vectrunc_append(GEN x, GEN y) assuming x was allocated using vectrunc_init, appends y as the last element of x, which grows in the process. The function is shallow: we append y, not a copy; it is equivalent to
long lx = lg(x); gel(x,lx) = y; setlg(x, lx+1);
\end{verbatim}
Beware that the maximal size of x (the l argument to vectrunc_init) is unknown, hence unchecked, and stack corruption will occur if we append more than l − 1 elements to x. Use the safer (but slower) shallowconcat when l is not easy to bound in advance.
\end{verbatim}
An other possibility is simply to allocate using `cgetg(1, t)` then fill the components as they become available: this time the downside is that we do not obtain a correct `GEN` until the vector is complete. Almost no PARI function will be able to operate on it.

```c
void vectrunc_append_batch(GEN x, GEN y) successively apply
    vectrunc_append(x, gel(y, i))
for all elements of the vector y.
GEN vecsmalltrunc_init(long l)
void vecsmalltrunc_append(GEN x, long t) analog to the above for a `t_VECSMALL` container.
```

### 5.2.2 Length conversions.

These routines convert a non-negative length to different units. Their behavior is undefined at negative integers.

```c
long ndec2nlong(long x) converts a number of decimal digits to a number of words. Returns
1 + floor(x × BITS_IN_LONG log2 10).
long ndec2prec(long x) converts a number of decimal digits to a number of codewords. This is
equal to 2 + ndec2nlong(x).
long prec2ndec(long x) converts a number of of codewords to a number of decimal digits.
long nbits2nlong(long x) converts a number of bits to a number of words. Returns the smallest
word count containing x bits, i.e ceil(x/BITS_IN_LONG).
long nbits2prec(long x) converts a number of bits to a number of codewords. This is equal to
2 + nbits2nlong(x).
long nchar2nlong(long x) converts a number of bytes to number of words. Returns the smallest
word count containing x bytes, i.e ceil(x/sizeof(long)).
long bit_accuracy(long x) converts a `t_REAL` length into a number of significant bits. Returns
(x - 2)BITS_IN_LONG.
```

```c
double bit_accuracy_mul(long x, double y) returns (x - 2)BITS_IN_LONG × y.
```

### 5.2.3 Read type-dependent information.

```c
long typ(GEN x) returns the type number of x. The header files included through `pari.h` define
symbolic constants for the `GEN` types: `t_INT` etc. Never use their actual numerical values. E.g to
determine whether x is a `t_INT`, simply check
```
if (typ(x) == t_INT) { }
```
The types are internally ordered and this simplifies the implementation of commutative binary
operations (e.g addition, gcd). Avoid using the ordering directly, as it may change in the future;
use type grouping functions instead (Section 5.2.6).

```c
const char* type_name(long t) given a type number t this routine returns a string containing
its symbolic name. E.g `type_name(t_INT)` returns "t_INT". The return value is read-only.
long lg(GEN x) returns the length of x in BITS_IN_LONG-bit words.
long lgefint(GEN x) returns the effective length of the `t_INT` x in BITS_IN_LONG-bit words.
```
long signe(GEN x) returns the sign \((-1, 0 or 1)\) of x. Can be used for t_INT, t_REAL, t_POL and t_SER (for the last two types, only 0 or 1 are possible).

long gsigne(GEN x) returns the sign of a real number x, valid for t_INT, t_REAL as signe, but also for t_FRAC. Raise a type error if typ(x) is not among those three.

long expi(GEN x) returns the binary exponent of the real number equal to the t_INT x. This is a special case of gexpo.

long expo(GEN x) returns the binary exponent of the t_REAL x.

long mpexpo(GEN x) returns the binary exponent of the t_INT or t_REAL x.

long gexpo(GEN x) same as expo, but also valid when x is not a t_REAL (returns the largest exponent found among the components of x). When x is an exact 0, this returns -HIGHEXPOBIT, which is lower than any valid exponent.

long valp(GEN x) returns the \(p\)-adic valuation (for a t_PADIC) or \(X\)-adic valuation (for a t_SER, taken with respect to the main variable) of x.

long precp(GEN x) returns the precision of the t_PADIC x.

long gvar(GEN x) returns the main variable number when any variable at all occurs in the composite object x (the smallest variable number which occurs), and NO_VARIABLE otherwise.

long gvar2(GEN x) returns the variable number for the ring over which x is defined, e.g. if \(x \in \mathbb{Z}[a][b]\) return (the variable number for) a. Return NO_VARIABLE if x has no variable or is not defined over a polynomial ring.

long degpol(GEN x) is a simple macro returning \(\lg(x) - 3\). This is the degree of the t_POL x with respect to its main variable, if its leading coefficient is non-zero (a rational 0 is impossible, but an inexact 0 is allowed, as well as an exact modular 0, e.g. \(\text{Mod}(0,2)\)). If x has no coefficients (rational 0 polynomial), its length is 2 and we return the expected \(-1\).

long lgpol(GEN x) is equal to \(\degpol(x) + 1\). Used to loop over the coefficients of a t_POL in the following situation:

```c
GEN xd = x + 2;
long i, l = lgpol(x);
for (i = 0; i < l; i++) foo( xd[i] ).
```

long precision(GEN x) If x is of type t_REAL, returns the precision of x, namely the length of x in BITS_IN_LONG-bit words if x is not zero, and a reasonable quantity obtained from the exponent of x if x is numerically equal to zero. If x is of type t_COMPLEX, returns the minimum of the precisions of the real and imaginary part. Otherwise, returns 0 (which stands for infinite precision).

long lgcols(GEN x) is equal to \(\lg(\text{gel}(x,1))\). This is the length of the columns of a t_MAT with at least one column.

long nbrows(GEN x) is equal to \(\lg(\text{gel}(x,1))-1\). This is the number of rows of a t_MAT with at least one column.

long gprecision(GEN x) as precision for scalars. Returns the lowest precision encountered among the components otherwise.

long sizedigit(GEN x) returns 0 if x is exactly 0. Otherwise, returns \(\text{gexpo}(x)\) multiplied by \(\log_{10}(2)\). This gives a crude estimate for the maximal number of decimal digits of the components of x.
5.2.4 Eval type-dependent information. These routines convert type-dependent information to bitmask to fill the codewords of GEN objects (see Section 4.5). E.g. for a t_REAL z:

\[ z[1] = \text{evalsigne}(-1) \mid \text{evalexpo}(2) \]

Compatible components of a codeword for a given type can be OR-ed as above.

ulong evaltyp(long x) convert type x to bitmask (first codeword of all GENs)
long evalallg(long x) convert length x to bitmask (first codeword of all GENs). Raise overflow error if x is so large that the corresponding length cannot be represented
long _evalallg(long x) as evalallg without the overflow check.
ulong evalvarn(long x) convert variable number x to bitmask (second codeword of t_POL and t_SER)
long evalsigne(long x) convert sign x (in \(-1, 0, 1\)) to bitmask (second codeword of t_INT, t_REAL, t_POL, t_SER)
long evalprecp(long x) convert p-adic (X-adic) precision x to bitmask (second codeword of t_PADIC, t_SER). Raise overflow error if x is so large that the corresponding precision cannot be represented.
long _evalprecp(long x) same as evalprecp without the overflow check.
long evalvalp(long x) convert p-adic (X-adic) valuation x to bitmask (second codeword of t_PADIC, t_SER). Raise overflow error if x is so large that the corresponding valuation cannot be represented.
long _evalvalp(long x) same as evalvalp without the overflow check.
long evalexpo(long x) convert exponent x to bitmask (second codeword of t_REAL). Raise overflow error if x is so large that the corresponding exponent cannot be represented
long _evalexpo(long x) same as evalexpo without the overflow check.
long evalallgefint(long x) convert effective length x to bitmask (second codeword of t_INT). This should be less or equal than the length of the t_INT, hence there is no overflow check for the effective length.

5.2.5 Set type-dependent information. Use these functions and macros with extreme care since usually the corresponding information is set otherwise, and the components and further codeword fields (which are left unchanged) may not be compatible with the new information.

void settyp(GEN x, long s) sets the type number of x to s.
void setlg(GEN x, long s) sets the length of x to s. This is an efficient way of truncating vectors, matrices or polynomials.
void setlgefint(GEN x, long s) sets the effective length of the t_INT x to s. The number s must be less than or equal to the length of x.
void setsigne(GEN x, long s) sets the sign of x to s. If x is a t_INT or t_REAL, s must be equal to \(-1, 0, 1\), and if x is a t_POL or t_SER, s must be equal to 0 or 1. No sanity check is made; in particular, setting the sign of a 0 t_INT to \(±1\) creates an invalid object.
void togglesign(GEN x) sets the sign s of x to \(-s\), in place.

54
void togglesign_safe(GEN *x) sets the sign of *x to \(-s\), in place, unless *x is one of the integer universal constants in which case replace *x by its negation (e.g. replace gen\_i by gen\_m1).

void setabssign(GEN x) sets the sign of \(|s|\) in place.

void affectsign(GEN x, GEN y) shortcut for setsigne(y, signe(x)). No sanity check is made; in particular, setting the sign of a 0 t\_INT to \pm 1 creates an invalid object.

void affectsign_safe(GEN x, GEN *y) sets the sign of *y to that of x, in place, unless *y is one of the integer universal constants in which case replace *y by its negation if needed (e.g. replace gen\_1 by gen\_m1 if x is negative). No other sanity check is made; in particular, setting the sign of a 0 t\_INT to \pm 1 creates an invalid object.

void normalize_frac(GEN z) assuming z is of the form mkfrac(a,b) with b \(\neq 0\), make sure that b > 0 by changing the sign of a in place if needed (use togglesign).

void setexpo(GEN x, long s) sets the binary exponent of the t\_REAL x to s. The value s must be a 24-bit signed number.

void setvalp(GEN x, long s) sets the \(p\)-adic or \(X\)-adic valuation of x to s, if x is a t\_PADIC or a t\_SER, respectively.

void setprecip(GEN x, long s) sets the \(p\)-adic precision of the t\_PADIC x to s.

void setvarn(GEN x, long s) sets the variable number of the t\_POL or t\_SER x to s (where 0 \(\leq s \leq\) MAXVARN).

5.2.6 Type groups. In the following functions, t denotes the type of a GEN. They used to be implemented as macros, which could evaluate their argument twice; no longer: it is not inefficient to write

\[
is\_intreal\_t(typ(x))
\]

int is_recursive_t(long t) true iff t is a recursive type (the non-recursive types are t\_INT, t\_REAL, t\_STR, t\_VECSMALL). Somewhat contrary to intuition, t\_LIST is also non-recursive, ; see the Developer’s guide for details.

int is\_intreal\_t(long t) true iff t is t\_INT or t\_REAL.

int is\_rational\_t(long t) true iff t is t\_INT or t\_FRAC.

int is\_vec\_t(long t) true iff t is t\_VEC or t\_COL.

int is\_matvec\_t(long t) true iff t is t\_MAT, t\_VEC or t\_COL.

int is\_scalar\_t(long t) true iff t is a scalar, i.e a t\_INT, a t\_REAL, a t\_INTMOD, a t\_FRAC, a t\_COMPLEX, a t\_PADIC, a t\_QUAD, or a t\_POLMOD.

int is\_extscalar\_t(long t) true iff t is a scalar (see is\_scalar\_t) or t is t\_POL.

int is\_const\_t(long t) true iff t is a scalar which is not t\_POLMOD.

int is\_noncalc\_t(long t) true if generic operations (gadd, gmul) do not make sense for t: corresponds to types t\_LIST, t\_STR, t\_VECSMALL, t\_CLOSURE.
5.2.7 Accessors and components. The first two functions return GEN components as copies on the stack:

`GEN compo(GEN x, long n)` creates a copy of the n-th true component (i.e. not counting the codewords) of the object x.

`GEN truecoeff(GEN x, long n)` creates a copy of the coefficient of degree n of x if x is a scalar, t_POL or t_SER, and otherwise of the n-th component of x.

On the contrary, the following routines return the address of a GEN component. No copy is made on the stack:

`GEN constant_term(GEN x)` returns the address of the constant coefficient of t_POL x. By convention, a 0 polynomial (whose sign is 0) has gen_0 constant term.

`GEN leading_term(GEN x)` returns the address of the leading coefficient of t_POL x, i.e. the coefficient of largest index stored in the array representing x. This may be an inexact 0. By convention, return gen_0 if the coefficient array is empty.

`GEN gel(GEN x, long i)` returns the address of the x[i] entry of x. (el stands for element.)

`GEN gcoeff(GEN x, long i, long j)` returns the address of the x[i,j] entry of t_MAT x, i.e. the coefficient at row i and column j.

`GEN gmael(GEN x, long i, long j)` returns the address of the x[i][j] entry of x. (mael stands for multidimensional array element.)

`GEN gmael2(GEN A, long x1, long x2)` is an alias for gmael. Similar macros gmael3, gmael4, gmael5 are available.

5.3 Global numerical constants.

These are defined in the various public PARI headers.

5.3.1 Constants related to word size.

`long BITS_IN_LONG = 2^TWOPOTBITS_IN_LONG` number of bits in a long (32 or 64).

`long BITS_IN_HALFULONG`: BITS_IN_LONG divided by 2.

`long LONG_MAX`: the largest positive long.

`ulong ULONG_MAX`: the largest ulong.

`long DEFAULTPREC`: the length (1g) of a t_REAL with 64 bits of accuracy

`long MEDDEFAULTPREC`: the length (1g) of a t_REAL with 128 bits of accuracy

`long BIGDEFAULTPREC`: the length (1g) of a t_REAL with 192 bits of accuracy

`ulong HIGHBIT`: the largest power of 2 fitting in an ulong.

`ulong LOWMASK`: bitmask yielding the least significant bits.

`ulong HIGHMASK`: bitmask yielding the most significant bits.

The last two are used to implement the following convenience macros, returning half the bits of their operand:
ulong LOWWORD(ulong a) returns least significant bits.

ulong HIGHWORD(ulong a) returns most significant bits.

Finally

long divsBIL(long n) returns the Euclidean quotient of n by BITS_IN_LONG (with non-negative remainder).

long remsBIL(n) returns the (non-negative) Euclidean remainder of n by BITS_IN_LONG

long dvmdbIL(long n, long *r)
ulong dvmduBIL(ulong n, ulong *r) sets r to remsBIL(n) and returns divsBIL(n).

5.3.2 Masks used to implement the GEN type.

These constants are used by higher level macros, like typ or lg:

EXPOnumBITS, LGnumBITS, SIGNnumBITS, TYPnumBITS, VALPnumBITS, VARNnumBITS: number of bits used to encode expo, lg, signe, typ, valp, varn.

PRECPSHIFT, SIGNSHIFT, TYPSHIFT, VARNSHIFT: shifts used to recover or encode precp, varn, typ, signe

CLONEBIT, EXPOBITS, LGBITS, PRECPBITS, SIGNBITS, TYPBITS, VALPBITS, VARNBITS: bitmasks used to extract isclone, expo, lg, precp, signe, typ, valp, varn from GEN codewords.

MAXVARN: the largest possible variable number.

NO_VARIABLE: sentinel returned by gvar(x) when x does not contain any polynomial; has a lower priority than any valid variable number.

HIGHEXPOBIT: a power of 2, one more that the largest possible exponent for a t_REAL.

HIGHVALPBNOT: a power of 2, one more that the largest possible valuation for a t_PADIC or a t_SER.

5.3.3 log 2, π.

These are double approximations to useful constants:

LOG2: log 2.

LOG10_2: log 2 / log 10.

LOG2_10: log 10 / log 2.

PI: π.
5.4 Iterating over small primes, low-level interface.

One of the methods used by the high-level prime iterator (see Section 4.8.2), is a precomputed table. Its direct use is deprecated, but documented here.

After \texttt{pari\_init(size, maxprime)}, a “prime table” is initialized with the successive differences of primes up to (possibly just a little beyond) \texttt{maxprime}. The prime table occupies roughly \texttt{maxprime/\log(maxprime)} bytes in memory, so be sensible when choosing \texttt{maxprime}; it is 500000 by default under \texttt{gp} and there is no real benefit in choosing a much larger value: the high-level iterator provide fast access to primes up to the square of \texttt{maxprime}. In any case, the implementation requires that \texttt{maxprime < 2^{BITS\_IN\_LONG} - 2048}, whatever memory is available.

\texttt{PARI} currently guarantees that the first 6547 primes, up to and including 65557, are present in the table, even if you set \texttt{maxprime} to zero. in the \texttt{pari\_init} call.

Some convenience functions:

\texttt{ulong maxprime()} the largest prime computable using our prime table.

\texttt{void maxprime\_check(ulong B)} raise an error if \texttt{maxprime()} is \texttt{< B}.

After the following initializations (the names \texttt{p} and \texttt{ptr} are arbitrary of course)

\begin{verbatim}
byteptr ptr = diffptr;
ulong p = 0;
\end{verbatim}

calling the macro \texttt{NEXT\_PRIME\_VIADIFF\_CHECK(p, ptr)} repeatedly will assign the successive prime numbers to \texttt{p}. Overrunning the prime table boundary will raise the error \texttt{e\_MAXPRIME}, which will just print the error message:

\begin{verbatim}
*** not enough precomputed primes, need primelimit \texttt{-c}
\end{verbatim}

(for some numerical value \texttt{c}), then abort the computation. The alternative macro \texttt{NEXT\_PRIME\_VIADIFF} operates in the same way, but will omit that check, and is slightly faster. It should be used in the following way:

\begin{verbatim}
byteptr ptr = diffptr;
ulong p = 0;
if (maxprime() < goal) pari\_err\_MAXPRIME(goal); /* not enough primes */
while (p <= goal) /* run through all primes up to goal */
{
    NEXT\_PRIME\_VIADIFF(p, ptr);
    ...
}
\end{verbatim}

Here, we use the general error handling function \texttt{pari\_err} (see Section 4.7.3), with the codeword \texttt{e\_MAXPRIME}, raising the “not enough primes” error. This could be rewritten as

\begin{verbatim}
maxprime\_check(goal);
while (p <= goal) /* run through all primes up to goal */
{
    NEXT\_PRIME\_VIADIFF(p, ptr);
    ...
}
\end{verbatim}
bytepr initprimes(ulong maxprime, long *L, ulong *lastp) computes a (malloc’ed) “prime table”, in fact a table of all prime differences for \( p < \text{maxprime} \) (and possibly a little beyond). Set \( L \) to the table length (argument to malloc), and \( lastp \) to the last prime in the table.

void initprimetable(ulong maxprime) computes a prime table (of all prime differences for \( p < \text{maxprime} \)) and assign it to the global variable diffptr. Don’t change diffptr directly, call this function instead. This calls initprimes and updates internal data recording the table size.

ulong init_primepointer_geq(ulong a, byteptr *pd) returns the smallest prime \( p \geq a \), and sets \(*pd\) to the proper offset of diffptr so that \( \text{NEXT\_PRIME\_VIADIFF}(p, *pd) \) correctly returns \( \text{unextprime}(p + 1) \).

ulong init_primepointer_gt(ulong a, byteptr *pd) returns the smallest prime \( p > a \).

ulong init_primepointer_leq(ulong a, byteptr *pd) returns the largest prime \( p \leq a \).

ulong init_primepointer_lt(ulong a, byteptr *pd) returns the largest prime \( p < a \).

5.5 Handling the PARI stack.

5.5.1 Allocating memory on the stack.

GEN cgetg(long n, long t) allocates memory on the stack for an object of length \( n \) and type \( t \), and initializes its first codeword.

GEN cgeti(long n) allocates memory on the stack for a \( t\_\text{INT} \) of length \( n \), and initializes its first codeword. Identical to cgetg(\( n, t\_\text{INT} \)).

GEN cgetr(long n) allocates memory on the stack for a \( t\_\text{REAL} \) of length \( n \), and initializes its first codeword. Identical to cgetg(\( n, t\_\text{REAL} \)).

GEN cgetc(long n) allocates memory on the stack for a \( t\_\text{COMPLEX} \), whose real and imaginary parts are \( t\_\text{REALs} \) of length \( n \).

GEN cgetp(GEN x) creates space sufficient to hold the \( t\_\text{PADIC} \) \( x \), and sets the prime \( p \) and the \( p\)-adic precision to those of \( x \), but does not copy (the \( p\)-adic unit or zero representative and the modulus of) \( x \).

GEN new_chunk(size_t n) allocates a GEN with \( n \) components, without filling the required code words. This is the low-level constructor underlying cgetg, which calls new_chunk then sets the first code word. It works by simply returning the address \(((\text{GEN})\text{avma}) - n\), after checking that it is larger than (\( \text{GEN} \))bot.

char* stack_malloc(size_t n) allocates memory on the stack for \( n \) chars (not \( n \) GENs). This is faster than using malloc, and easier to use in most situations when temporary storage is needed. In particular there is no need to free individually all variables thus allocated: a simple \( \text{avma} = \text{oldavma} \) might be enough. On the other hand, beware that this is not permanent independent storage, but part of the stack.

char* stack_calloc(size_t n) as stack_malloc, setting the memory to zero.

Objects allocated through these last three functions cannot be gerepile’d, since they are not yet valid GENs: their codewords must be filled first.

GEN cgetalloc(long t, size_t l), same as cgetg(t, l), except that the result is allocated using pari_malloc instead of the PARI stack. The resulting GEN is now impervious to garbage collecting routines, but should be freed using pari_free.
5.5.2 Stack-independent binary objects.

GENbin* copy_bin(GEN x) copies x into a malloc’ed structure suitable for stack-independent binary transmission or storage. The object obtained is architecture independent provided, sizeof(long) remains the same on all PARI instances involved, as well as the multiprecision kernel (either native or GMP).

GENbin* copy_bin_canon(GEN x) as copy_bin, ensuring furthermore that the binary object is independent of the multiprecision kernel. Slower than copy_bin.

GEN bin_copy(GENbin *p) assuming p was created by copy_bin(x) (not necessarily by the same PARI instance: transmission or external storage may be involved), restores x on the PARI stack.

The routine bin_copy transparently encapsulate the following functions:

GEN GENbinbase(GENbin *p) the GEN data actually stored in p. All addresses are stored as offsets with respect to a common reference point, so the resulting GEN is unusable unless it is a non-recursive type; private low-level routines must be called first to restore absolute addresses.

void shiftaddress(GEN x, long dec) converts relative addresses to absolute ones.

void shiftaddress_canon(GEN x, long dec) converts relative addresses to absolute ones, and converts leaves from a canonical form to the one specific to the multiprecision kernel in use. The GENbin type stores whether leaves are stored in canonical form, so bin_copy can call the right variant.

Objects containing closures are harder to e.g. copy and save to disk, since closures contain pointers to libpari functions that will not be valid in another gp instance: there is little chance for them to be loaded at the exact same address in memory. Such objects must be saved along with a linking table.

GEN copybin_unlink(GEN C) returns a linking table allowing to safely store and transmit t_CLOSURE objects in C. If C = NULL return a linking table corresponding to the content of all gp variables. C may then be dumped to disk in binary form, for instance.

void bincopy_relink(GEN C, GEN V) given a binary object C, as dumped by writebin and read back into a session, and a linking table V, restore all closures contained in C (function pointers are translated to their current value).

5.5.3 Garbage collection. See Section 4.3 for a detailed explanation and many examples.

void cgiv(GEN x) frees object x, assuming it is the last created on the stack.

GEN gerepile(pari_sp p, pari_sp q, GEN x) general garbage collector for the stack.

void gerepileall(pari_sp av, int n, ...) cleans up the stack from av on (i.e from avma to av), preserving the n objects which follow in the argument list (of type GEN*). For instance, gerepileall(av, 2, &x, &y) preserves x and y.

void gerepileallsp(pari_sp av, pari_sp ltop, int n, ...) cleans up the stack between av and ltop, updating the n elements which follow n in the argument list (of type GEN*). Check that the elements of g have no component between av and ltop, and assumes that no garbage is present between avma and ltop. Analogous to (but faster than) gerepileall otherwise.

GEN gerepilecopy(pari_sp av, GEN x) cleans up the stack from av on, preserving the object x. Special case of gerepileall (case n = 1), except that the routine returns the preserved GEN instead of updating its address through a pointer.
void gerepilemany(pari_sp av, GEN* g[], int n) alternative interface to gerepileall. The preserved GENs are the elements of the array g of length n: g[0], g[1], ..., g[n-1]. Obsolete: no more efficient than gerepileall, error-prone, and clumsy (need to declare an extra GEN *g).

void gerepilemanysp(pari_sp av, pari_sp ltop, GEN* g[], int n) alternative interface to gerepileallsp. Obsolete.

void gerepilecoeffs(pari_sp av, GEN x, int n) cleans up the stack from av on, preserving x[0], ..., x[n-1] (which are GENs).

void gerepilecoeffssp(pari_sp av, pari_sp ltop, GEN x, int n) cleans up the stack from av to ltop, preserving x[0], ..., x[n-1] (which are GENs). Same assumptions as in gerepilemanysp, of which this is a variant. For instance

```c
z = cgetg(3, t_COMPLEX);
av = avma; garbage(); ltop = avma;
z[1] = fun1();
z[2] = fun2();
gerepilecoeffssp(av, ltop, z + 1, 2);
return z;
```
cleans up the garbage between av and ltop, and connects z and its two components. This is marginally more efficient than the standard

```c
av = avma; garbage(); ltop = avma;
z = cgetg(3, t_COMPLEX);
z[1] = fun1();
z[2] = fun2(); return gerepile(av, ltop, z);
```

GEN gerepileupto(pari_sp av, GEN q) analogous to (but faster than) gerepilecopy. Assumes that q is connected and that its root was created before any component. If q is not on the stack, this is equivalent to avma = av; in particular, sentinels which are not even proper GENs such as q = NULL are allowed.

GEN gerepileuptoint(pari_sp av, GEN q) analogous to (but faster than) gerepileupto. Assumes further that q is a t_INT. The length and effective length of the resulting t_INT are equal.

GEN gerepileuptoleaf(pari_sp av, GEN q) analogous to (but faster than) gerepileupto. Assumes further that q is a leaf, i.e. a non-recursive type (is_recursive(t(typ(q))) is non-zero). Contrary to gerepileuptoint and gerepileupto, gerepileuptoleaf leaves length and effective length of a t_INT unchanged.

5.5.4 Garbage collection: advanced use.

void stackdummy(pari_sp av, pari_sp ltop) inhibits the memory area between av included and ltop excluded with respect to gerepile, in order to avoid a call to gerepile(av, ltop,...). The stack space is not reclaimed though.

More precisely, this routine assumes that av is recorded earlier than ltop, then marks the specified stack segment as a non-recursive type of the correct length. Thus gerepile will not inspect the zone, at most copy it. To be used in the following situation:

```c
av0 = avma; z = cgetg(t_VEC, 3);
gel(z,1) = HUGE(); av = avma; garbage(); ltop = avma;
```
\begin{verbatim}
gel(z, 2) = HUGE(); stackdummy(av, ltop);

Compared to the orthodox

gel(z, 2) = gerepile(av, ltop, gel(z, 2));

or even more wasteful

z = gerepilécopy(av0, z);

we temporarily lose (av − ltop) words but save a costly gerepile. In principle, a garbage collection higher up the call chain should reclaim this later anyway.

Without the stackdummy, if the [av, ltop] zone is arbitrary (not even valid GENs as could happen after direct truncation via setlg), we would leave dangerous data in the middle of z, which would be a problem for a later

gerepile(..., ..., z);

And even if it were made of valid GENs, inhibiting the area makes sure gerepile will not inspect their components, saving time.

Another natural use in low-level routines is to “shorten” an existing GEN z to its first n − 1 components:

setlg(z, n);
stackdummy((pari_sp)(z + lg(z)), (pari_sp)(z + n));

or to its last n components:

long L = lg(z) - n, tz = typ(z);
stackdummy((pari_sp)(z + L), (pari_sp)z);
z += L; z[0] = evaltyp(tz) | evallg(L);

The first scenario (safe shortening an existing GEN) is in fact so common, that we provide a function for this:

void fixlg(GEN z, long ly) a safe variant of setlg(z, ly). If ly is larger than lg(z) do nothing. Otherwise, shorten z in place, using stackdummy to avoid later gerepile problems.

GEN gcgy_avma(GEN x, pari_sp *AVMA) return a copy of x as from gcopy, except that we pretend that initially avma is *AVMA, and that *AVMA is updated accordingly (so that the total size of x is the difference between the two successive values of *AVMA). It is not necessary for *AVMA to initially point on the stack: gclone is implemented using this mechanism.

GEN icopy_avma(GEN x, pari_sp av) analogous to gcopy_avma but simpler: assume x is a t_INT and return a copy allocated as if initially we had avma equal to av. There is no need to pass a pointer and update the value of the second argument: the new (fictitious) avma is just the return value (typecast to pari_sp).

5.5.5 Debugging the PARI stack.

int chk_gerepilécupto(GEN x) returns 1 if x is suitable for gerepilécupto, and 0 otherwise. In the latter case, print a warning explaining the problem.

void dbg_gerepile(pari_sp ltop) outputs the list of all objects on the stack between avma and ltop, i.e. the ones that would be inspected in a call to gerepile(..., ltop,...).

void dbg_gerepilécupto(GEN q) outputs the list of all objects on the stack that would be inspected in a call to gerepilécupto(..., q).
\end{verbatim}
5.5.6 Copies.

\texttt{GEN gcopy(GEN x)} creates a new copy of \( x \) on the stack.

\texttt{GEN gcopy\_lg(GEN x, long l)} creates a new copy of \( x \) on the stack, pretending that \( \lg(x) \) is \( l \), which must be less than or equal to \( \lg(x) \). If equal, the function is equivalent to \texttt{gcopy(x)}.

\texttt{int isonstack(GEN x)} true iff \( x \) belongs to the stack.

\texttt{void copyifstack(GEN x, GEN y)} sets \( y = gcopy(x) \) if \( x \) belongs to the stack, and \( y = x \) otherwise. This macro evaluates its arguments once, contrary to

\[
y = \text{isonstack}(x) \text{? gcopy}(x) : x;
\]

\texttt{void icopyifstack(GEN x, GEN y)} as \texttt{copyifstack} assuming \( x \) is a \texttt{t\_INT}.

5.5.7 Simplify.

\texttt{GEN simplify(GEN x)} you should not need that function in library mode. One rather uses:

\texttt{GEN simplify\_shallow(GEN x)} shallow, faster, version of \texttt{simplify}.

5.6 The PARI heap.

5.6.1 Introduction.

It is implemented as a doubly-linked list of \texttt{malloc\_ed} blocks of memory, equipped with reference counts. Each block has type \texttt{GEN} but need not be a valid \texttt{GEN}: it is a chunk of data preceded by a hidden header (meaning that we allocate \( x \) and return \( x + \text{headersize} \)). A \textit{clone}, created by \texttt{gclone}, is a block which is a valid \texttt{GEN} and whose \textit{clone bit} is set.

5.6.2 Public interface.

\texttt{GEN newblock(size\_t n)} allocates a block of \( n \) \textit{words} (not bytes).

\texttt{void killblock(GEN x)} deletes the block \( x \) created by \texttt{newblock}. Fatal error if \( x \) not a block.

\texttt{GEN gclone(GEN x)} creates a new permanent copy of \( x \) on the heap (allocated using \texttt{newblock}). The \textit{clone bit} of the result is set.

\texttt{GEN gcloneref(GEN x)} if \( x \) is not a clone, clone it and return the result; otherwise, increase the clone reference count and return \( x \).

\texttt{void unclone(GEN x)} deletes a clone. Deletion at first only decreases the reference count by 1. If the count remains positive, no further action is taken; if the count becomes zero, then the clone is actually deleted. In the current implementation, this is an alias for \texttt{killblock}, but it is cleaner to kill clones (valid \texttt{GEN}s) using this function, and other blocks using \texttt{killblock}.

\texttt{void unclone\_deep(GEN x)} is only useful in the context of the GP interpreter which may replace arbitrary components of container types (\texttt{t\_VEC}, \texttt{t\_COL}, \texttt{t\_MAT}, \texttt{t\_LIST}) by clones. If \( x \) is such a container, the function recursively deletes all clones among the components of \( x \), then unclones \( x \). Useless in library mode: simply use \texttt{gclone}.

\texttt{void traverseheap(void(*f)(GEN, void *), void *data)} this applies \( f(x, \text{data}) \) to each object \( x \) on the PARI heap, most recent first. Mostly for debugging purposes.

63
GEN getheap() a simple wrapper around traverseheap. Returns a two-component row vector giving the number of objects on the heap and the amount of memory they occupy in long words.

GEN cgetg_block(long x, long y) as cgetg(x,y), creating the return value as a block, not on the PARI stack.

GEN cgetr_block(long prec) as cgetr(prec), creating the return value as a block, not on the PARI stack.

5.6.3 Implementation note. The hidden block header is manipulated using the following private functions:

void* bl_base(GEN x) returns the pointer that was actually allocated by malloc (can be freed).

long bl_refc(GEN x) the reference count of x: the number of pointers to this block. Incremented in killblock, decremented by the private function void gclone_refc(GEN x); block is freed when the reference count reaches 0.

long bl_num(GEN x) the index of this block in the list of all blocks allocated so far (including freed blocks). Uniquely identifies a block until \(2^{\text{BITS}_\text{IN}\text{LONG}}\) blocks have been allocated and this wraps around.

GEN bl_next(GEN x) the block after x in the linked list of blocks (NULL if x is the last block allocated not yet killed).

GEN bl_prev(GEN x) the block allocated before x (never NULL).

We documented the last four routines as functions for clarity (and type checking) but they are actually macros yielding valid lvalues. It is allowed to write bl_refc(x)++ for instance.

5.7 Handling user and temp variables.

Low-level implementation of user / temporary variables is liable to change. We describe it nevertheless for completeness. Current variables are implemented by a single array of values divided in 3 zones: 0–nvar (user variables), maxavail–MAXVARN (temporary variables), and nvar+1–maxavail-1 (pool of free variable numbers).

5.7.1 Low-level.

void pari_var_init(): a small part of pari_init. Resets variable counters nvar and maxavail, notwithstanding existing variables! In effect, this even deletes x. Don’t use it.

long pari_var_next(): returns nvar, the number of the next user variable we can create.

long pari_var_next_temp() returns maxavail, the number of the next temp variable we can create.

void pari_var_create(entree *ep) low-level initialization of an EpVAR.

The obsolete function long manage_var(long n, entree *ep) is kept for backward compatibility only. Don’t use it.

64
5.7.2 User variables.

`long fetch_user_var(char *s)` returns a user variable whose name is `s`, creating it is needed (and using an existing variable otherwise). Returns its variable number.

`entree* fetch_named_var(char *s)` as `fetch_user_var`, but returns an `entree*` suitable for inclusion in the interpreter hashlists of symbols, not a variable number. `fetch_user_var` is a trivial wrapper.

`GEN fetch_var_value(long v)` returns a shallow copy of the current value of the variable numbered `v`. Return `NULL` for a temporary variable.

`entree* is_entry(const char *s)` returns the `entree*` associated to an identifier `s` (variable or function), from the interpreter hashtables. Return `NULL` is the identifier is unknown.

5.7.3 Temporary variables.

`long fetch_var(void)` returns the number of a new temporary variable (decreasing `max_avail`).

`long delete_var(void)` delete latest temp variable created and return the number of previous one.

`void name_var(long n, char *s)` rename temporary variable number `n` to `s`; mostly useful for nicer printout. Error when trying to rename a user variable: use `fetch_named_var` to get a user variable of the right name in the first place.

5.8 Adding functions to PARI.

5.8.1 Nota Bene. As mentioned in the COPYING file, modified versions of the PARI package can be distributed under the conditions of the GNU General Public License. If you do modify PARI, however, it is certainly for a good reason, and we would like to know about it, so that everyone can benefit from your changes. There is then a good chance that your improvements are incorporated into the next release.

We classify changes to PARI into four rough classes, where changes of the first three types are almost certain to be accepted. The first type includes all improvements to the documentation, in a broad sense. This includes correcting typos or inaccuracies of course, but also items which are not really covered in this document, e.g. if you happen to write a tutorial, or pieces of code exemplifying fine points unduly omitted in the present manual.

The second type is to expand or modify the configuration routines and skeleton files (the `Configure` script and anything in the `config/` subdirectory) so that compilation is possible (or easier, or more efficient) on an operating system previously not catered for. This includes discovering and removing idiosyncrasies in the code that would hinder its portability.

The third type is to modify existing (mathematical) code, either to correct bugs, to add new functionality to existing functions, or to improve their efficiency.

Finally the last type is to add new functions to PARI. We explain here how to do this, so that in particular the new function can be called from `gp`. 

65
5.8.2 Coding guidelines. Code your function in a file of its own, using as a guide other functions
in the PARI sources. One important thing to remember is to clean the stack before exiting your
main function, since otherwise successive calls to the function clutters the stack with unnecessary
garbage, and stack overflow occurs sooner. Also, if it returns a GEN and you want it to be accessible
to gp, you have to make sure this GEN is suitable for gerepileupto (see Section 4.3).

If error messages or warnings are to be generated in your function, use pari_err and pari_warn
respectively. Recall that pari_err does not return but ends with a longjmp statement. As well,
instead of explicit printf / fprintf statements, use the following encapsulated variants:

void pari_putchar(char c): write character c to the output stream.
void pari_putstr(char *s): write s to the output stream.
void pari_printf(const char *fmt, ...): write following arguments to the output stream,
according to the conversion specifications in format fmt (see printf).
void err_printf(const char *fmt, ...): as pari_printf, writing to PARI’s current error
stream.
void err_flush(void) flush error stream.

Declare all public functions in an appropriate header file, if you want to access them from C.
The other functions should be declared static in your file.

Your function is now ready to be used in library mode after compilation and creation of the
library. If possible, compile it as a shared library (see the Makefile coming with the extgcd
example in the distribution). It is however still inaccessible from gp.

5.8.3 GP prototypes, parser codes. A GP prototype is a character string describing all
the GP parser needs to know about the function prototype. It contains a sequence of the following
atoms:

• Return type: GEN by default (must be valid for gerepileupto), otherwise the following can
appear as the first char of the code string:
  i return int
  l return long
  v return void
  m return a GEN which is not gerepile-safe.

The m code is used for member functions, to avoid unnecessary copies. A copy opcode is
generated by the compiler if the result needs to be kept safe for later use.

• Mandatory arguments, appearing in the same order as the input arguments they describe:

<table>
<thead>
<tr>
<th>char</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>GEN</td>
</tr>
<tr>
<td>&amp;</td>
<td>*GEN</td>
</tr>
<tr>
<td>L</td>
<td>long (we implicitly typecast int to long)</td>
</tr>
<tr>
<td>V</td>
<td>loop variable</td>
</tr>
<tr>
<td>n</td>
<td>variable, expects a variable number (a long, not an *entree)</td>
</tr>
<tr>
<td>W</td>
<td>a GEN which is a lvalue to be modified in place (for t_LIST)</td>
</tr>
<tr>
<td>r</td>
<td>raw input (treated as a string without quotes). Quoted args are copied as strings</td>
</tr>
<tr>
<td>s</td>
<td>expanded string. Example: Pi&quot;x&quot;2 yields &quot;3.142x2&quot;</td>
</tr>
</tbody>
</table>

Example: aa"b\n\n)c yields the string "aab\n\nc"
Unquoted components can be of any PARI type, converted to string following current output format.

I closure whose value is ignored, as in for loops, to be processed by `void closure_eval(void GEN C)`
E closure whose value is used, as in sum loops, to be processed by `void closure_evalgen(GEN C)`
J implicit function of arity 1, as in parsum loops, to be processed by `void closure_callgen1(GEN C)`

A closure is a GP function in compiled (bytecode) form. It can be efficiently evaluated using the `closure_eval.xxx` functions.

- **Automatic arguments:**
  - `f` Fake *long*. C function requires a pointer but we do not use the resulting *long*
  - `p` real precision (default `realprecision`)
  - `P` series precision (default `seriesprecision`, global variable `precdl` for the library)
  - `C` lexical context (internal, for `eval`, see `localvars_read_str`)

- **Syntax requirements, used by functions like for, sum, etc.:**
  - `=` separator `=` required at this point (between two arguments)

- **Optional arguments and default values:**
  - `E*` any number of expressions, possibly 0 (see E)
  - `s*` any number of strings, possibly 0 (see s)
  - `Dxxx` argument can be omitted and has a default value

  The `E*` code reads all remaining arguments in closure context and passes them as a single `t_VEC`. The `s*` code reads all remaining arguments in string context (see Section 2.9.1), and passes the list of strings as a single `t_VEC`. The automatic concatenation rules in string context are implemented so that adjacent strings are read as different arguments, as if they had been comma-separated. For instance, if the remaining argument sequence is: "xx", 1, "yy", the `s*` atom sends `[a, b, c]`, where `a, b, c` are GENs of type `t_STR` (content "xx"), `t_INT` (equal to 1) and `t_STR` (content "yy").

  The format to indicate a default value (atom starts with a D) is “Dvalue,type,”, where `type` is the code for any mandatory atom (previous group), `value` is any valid GP expression which is converted according to `type`, and the ending comma is mandatory. For instance, if the user-given argument reads `1 + 3` at this point, `4L` is sent to the function; and `0L` if the argument is omitted. The following special notations are available:

  - `DG` optional GEN, send NULL if argument omitted.
  - `D&` optional *GEN, send NULL if argument omitted. The argument must be prefixed by &.
  - `Dr` optional raw string, send NULL if argument omitted.
  - `Ds` optional char *, send NULL if argument omitted.
  - `DV` optional *entree, send NULL if argument omitted.
  - `D1, DE` optional closure, send NULL if argument omitted.
  - `Dn` optional variable number, −1 if omitted.
**Hardcoded limit.** C functions using more than 20 arguments are not supported. Use vectors if you really need that many parameters.

When the function is called under gp, the prototype is scanned and each time an atom corresponding to a mandatory argument is met, a user-given argument is read (gp outputs an error message if the argument was missing). Each time an optional atom is met, a default value is inserted if the user omits the argument. The “automatic” atoms fill in the argument list transparently, supplying the current value of the corresponding variable (or a dummy pointer).

For instance, here is how you would code the following prototypes, which do not involve default values:

- `GEN f(GEN x, GEN y, long prec) ----> "GGp"
- `void f(GEN x, GEN y, long prec) ----> "vGGp"
- `void f(GEN x, long y, long prec) ----> "vGLp"
- `long f(GEN x) ----> "lG"
- `int f(long x) ----> "iL"

If you want more examples, gp gives you easy access to the parser codes associated to all GP functions: just type \h function. You can then compare with the C prototypes as they stand in paridecl.h.

**Remark.** If you need to implement complicated control statements (probably for some improved summation functions), you need to know how the parser implements closures and lexicals and how the evaluator lets you deal with them, in particular the push_lex and pop_lex functions. Check their descriptions and adapt the source code in language/sumiter.c and language/intnum.c.

### 5.8.4 Integration with gp as a shared module.

In this section we assume that your Operating System is supported by install. You have written a function in C following the guidelines is Section 5.8.2; in case the function returns a GEN, it must satisfy gerepileupto assumptions (see Section 4.3).

You then succeeded in building it as part of a shared library and want to finally tell gp about your function. First, find a name for it. It does not have to match the one used in library mode, but consistency is nice. It has to be a valid GP identifier, i.e. use only alphabetic characters, digits and the underscore character (_), the first character being alphabetic.

Then figure out the correct parser code corresponding to the function prototype (as explained in Section 5.8.3) and write a GP script like the following:

```
install(libname, code, gpname, library)
addhelp(gpname, "some help text")
```

(see Section 3.12.3 and 3.12.22). The addhelp part is not mandatory, but very useful if you want others to use your module. libname is how the function is named in the library, usually the same name as one visible from C.

Read that file from your gp session, for instance from your preferences file (Section 2.14.1.2), and that’s it. You can now use the new function gpname under gp, and we would very much like to hear about it!
Example. A complete description could look like this:

```c
{  install(bnfinit0, "GD0,L,DGp", ClassGroupInit, "libpari.so");
  addhelp(ClassGroupInit, "ClassGroupInit(P,{flag=0},{data=[]}): compute the necessary data for ...");
}
```

which means we have a function `ClassGroupInit` under `gp`, which calls the library function `bnfinit0`. The function has one mandatory argument, and possibly two more (two 'D' in the code), plus the current real precision. More precisely, the first argument is a `GEN`, the second one is converted to a `long` using `itos` (0 is passed if it is omitted), and the third one is also a `GEN`, but we pass `NULL` if no argument was supplied by the user. This matches the C prototype (from `paridecl.h`):

```c
GEN bnfinit0(GEN P, long flag, GEN data, long prec)
```

This function is in fact coded in `basemath/buch2.c`, and is in this case completely identical to the GP function `bnfinit` but `gp` does not need to know about this, only that it can be found somewhere in the shared library `libpari.so`.

Important note. You see in this example that it is the function’s responsibility to correctly interpret its operands: `data = NULL` is interpreted by the function as an empty vector. Note that since `NULL` is never a valid `GEN` pointer, this trick always enables you to distinguish between a default value and actual input: the user could explicitly supply an empty vector!

5.8.5 Library interface for install.

There is a corresponding library interface for this `install` functionality, letting you expand the GP parser/evaluator available in the library with new functions from your C source code. Functions such as `gp_read_str` may then evaluate a GP expression sequence involving calls to these new function!

```c
entree * install(void *f, const char *gpname, const char *code)
```

where `f` is the (address of the) function (cast to `void*`), `gpname` is the name by which you want to access your function from within your GP expressions, and `code` is as above.

5.8.6 Integration by patching gp.

If `install` is not available, and installing Linux or a BSD operating system is not an option (why?), you have to hardcode your function in the `gp` binary. Here is what needs to be done:

- Fetch the complete sources of the PARI distribution.
- Drop the function source code module in an appropriate directory (a priori `src/modules`), and declare all public functions in `src/headers/paridecl.h`.
- Choose a help section and add a file `src/functions/section/gpname` containing the following, keeping the notation above:

```markdown
Function: gpname
Section: section
C-Name: libname
Prototype: code
Help: some help text
```
Two GP2C-related fields (Description and Wrapper) are also available to improve the code GP2C generates when compiling scripts involving your function. See the GP2C documentation for details.

- Launch Configure, which should pick up your C files and build an appropriate Makefile. At this point you can recompile gp, which will first rebuild the functions database.

**Example.** We reuse the ClassGroupInit / bnfinit0 from the preceding section. Since the C source code is already part of PARI, we only need to add a file

    functions/number_fields/ClassGroupInit

containing the following:

    Function: ClassGroupInit
    Section: number_fields
    C-Name: bnfinit0
    Prototype: GD0,L,DGp
    Help: ClassGroupInit(P,{flag=0},{tech=[]}): this routine does ...

and recompile gp.

### 5.9 Globals related to PARI configuration.

#### 5.9.1 PARI version numbers.

**paricfg_version_code** encodes in a single long, the Major and minor version numbers as well as the patchlevel.

```c
long PARI_VERSION(long M, long m, long p)
```

produces the version code associated to release $M.m.p$. Each code identifies a unique PARI release, and corresponds to the natural total order on the set of releases (bigger code number means more recent release).

**PARI_VERSION_SHIFT** is the number of bits used to store each of the integers $M$, $m$, $p$ in the version code.

**paricfg_vcsversion** is a version string related to the revision control system used to handle your sources, if any. For instance `git-commit hash` if compiled from a git repository.

The two character strings **paricfg_version** and **paricfg_buildinfo**, correspond to the first two lines printed by gp just before the Copyright message. The character string **paricfg_compiledate** is the date of compilation which appears on the next line. The character string **paricfg_mt_engine** is the name of the threading engine on the next line.

```c
GEN pari_version()
```

returns the version number as a PARI object, a t_VEC with three t_INT and one t_STR components.

#### 5.9.2 Miscellaneous.

**paricfg_datadir**: character string. The location of PARI's datadir.
6.1 Level 0 kernel (operations on ulongs).

6.1.1 Micro-kernel. The Level 0 kernel simulates basic operations of the 68020 processor on which PARI was originally implemented. They need “global” ulong variables overflow (which will contain only 0 or 1) and hiremainder to function properly. A routine using one of these lowest-level functions where the description mentions either hiremainder or overflow must declare the corresponding

```c
LOCAL_HIREMAINDER; /* provides 'hiremainder' */
LOCAL_OVERFLOW;  /* provides 'overflow' */
```

in a declaration block. Variables hiremainder and overflow then become available in the enclosing block. For instance a loop over the powers of an ulong p protected from overflows could read

```c
while (pk < lim)
{
    LOCAL_HIREMAINDER;
    ...
    pk = mulll(pk, p); if (hiremainder) break;
}
```

For most architectures, the functions mentioned below are really chunks of inlined assembler code, and the above ‘global’ variables are actually local register values.

ulong addll(ulong x, ulong y) adds x and y, returns the lower BITS_IN_LONG bits and puts the carry bit into overflow.

ulong addllx(ulong x, ulong y) adds overflow to the sum of the x and y, returns the lower BITS_IN_LONG bits and puts the carry bit into overflow.

ulong subll(ulong x, ulong y) subtracts x and y, returns the lower BITS_IN_LONG bits and put the carry (borrow) bit into overflow.

ulong subllx(ulong x, ulong y) subtracts overflow from the difference of x and y, returns the lower BITS_IN_LONG bits and puts the carry (borrow) bit into overflow.

int bfffo(ulong x) returns the number of leading zero bits in x. That is, the number of bit positions by which it would have to be shifted left until its leftmost bit first becomes equal to 1, which can be between 0 and BITS_IN_LONG − 1 for nonzero x. When x is 0, the result is undefined.

ulong mulll(ulong x, ulong y) multiplies x by y, returns the lower BITS_IN_LONG bits and stores the high-order BITS_IN_LONG bits into hiremainder.

ulong addmul(ulong x, ulong y) adds hiremainder to the product of x and y, returns the lower BITS_IN_LONG bits and stores the high-order BITS_IN_LONG bits into hiremainder.

ulong divll(ulong x, ulong y) returns the quotient of (hiremainder \* 2^BITS_IN_LONG) + x by y and stores the remainder into hiremainder. An error occurs if the quotient cannot be represented by an ulong, i.e. if initially hiremainder ≥ y.
Obsolete routines. Those functions are awkward and no longer used; they are only provided for backward compatibility:

ulong shiftl(ulong x, ulong y) returns \( x \) shifted left by \( y \) bits, i.e. \( x \ll y \), where we assume that \( 0 \leq y \leq \text{BITS}_\text{IN}_\text{LONG} \). The global variable hiremainder receives the bits that were shifted out, i.e. \( x \gg (\text{BITS}_\text{IN}_\text{LONG} - y) \).

ulong shiftr(ulong x, ulong y) returns \( x \) shifted right by \( y \) bits, i.e. \( x \gg y \), where we assume that \( 0 \leq y \leq \text{BITS}_\text{IN}_\text{LONG} \). The global variable hiremainder receives the bits that were shifted out, i.e. \( x \ll (\text{BITS}_\text{IN}_\text{LONG} - y) \).

6.1.2 Modular kernel. The following routines are not part of the level 0 kernel per se, but implement modular operations on words in terms of the above. They are written so that no overflow may occur. Let \( m \geq 1 \) be the modulus; all operands representing classes modulo \( m \) are assumed to belong to \([0, m - 1]\). The result may be wrong for a number of reasons otherwise: it may not be reduced, overflow can occur, etc.

int odd(ulong x) returns 1 if \( x \) is odd, and 0 otherwise.

int both_odd(ulong x, ulong y) returns 1 if \( x \) and \( y \) are both odd, and 0 otherwise.

ulong invmod2BIL(ulong x) returns the smallest positive representative of \( x^{-1} \mod 2^{\text{BITS}_\text{IN}_\text{LONG}} \), assuming \( x \) is odd.

ulong Fl_add(ulong x, ulong y, ulong m) returns the smallest positive representative of \( x + y \mod m \).

ulong Fl_neg(ulong x, ulong m) returns the smallest positive representative of \( -x \mod m \).

ulong Fl_sub(ulong x, ulong y, ulong m) returns the smallest positive representative of \( x - y \mod m \).

long Fl_center(ulong x, ulong m, ulong mo2) returns the representative in \([-m/2, m/2]\) of \( x \mod m \). Assume \( 0 \leq x < m \) and \( \text{mo2} = m >> 1 \).

ulong Fl_mul(ulong x, ulong y, ulong m) returns the smallest positive representative of \( xy \mod m \).

ulong Fl_double(ulong x, ulong m) returns \( 2x \mod m \).

ulong Fl_triple(ulong x, ulong m) returns \( 3x \mod m \).

ulong Fl_sqr(ulong x, ulong m) returns the smallest positive representative of \( x^2 \mod m \).

ulong Fl_inv(ulong x, ulong m) returns the smallest positive representative of \( x^{-1} \mod m \). If \( x \) is not invertible mod \( m \), raise an exception.

ulong Fl_invsafe(ulong x, ulong m) returns the smallest positive representative of \( x^{-1} \mod m \). If \( x \) is not invertible mod \( m \), return 0 (which is ambiguous if \( m = 1 \)).

ulong Fl_div(ulong x, ulong y, ulong m) returns the smallest positive representative of \( xy^{-1} \mod m \). If \( y \) is not invertible mod \( m \), raise an exception.

ulong Fl_powu(ulong x, ulong n, ulong m) returns the smallest positive representative of \( x^n \mod m \).

ulong Fl_sqrt(ulong x, ulong p) returns the square root of \( x \mod p \) (smallest positive representative). Assumes \( p \) to be prime, and \( x \) to be a square modulo \( p \).
ulong Fl_order(ulong a, ulong o, ulong p) returns the order of the Fp a. It is assumed that o is a multiple of the order of a, 0 being allowed (no non-trivial information).

ulong random_Fl(ulong p) returns a pseudo-random integer uniformly distributed in 0, 1,…p–1.

ulong pgener_Fl(ulong p) returns the smallest primitive root modulo p, assuming p is prime.

ulong pgener_Zl(ulong p) returns the smallest primitive root modulo p^k, k > 1, assuming p is an odd prime.

ulong pgener_Fl_local(ulong p, GEN L), see gener_Fp_local, L is an Flv.

6.1.3 Switching between Fl_xxx and standard operators.

Even though the Fl_xxx routines are efficient, they are slower than ordinary long operations, using the standard +, %, etc. operators. The following macro is used to choose in a portable way the most efficient functions for given operands:

int SMALL_ULONG(ulong p) true if 2p^2 < 2^BITS_IN_LONG. In that case, it is possible to use ordinary operators efficiently. If p < 2^BITS_IN_LONG, one may still use the Fl_xxx routines. Otherwise, one must use generic routines. For instance, the scalar product of the GENs x and y mod p could be computed as follows.

```c
long i, l = lg(x);
if (lgefint(p) > 3)
  { /* arbitrary */
    GEN s = gen_0;
    for (i = 1; i < l; i++) s = addii(s, mulii(gel(x,i), gel(y,i)));
    return modii(s, p).
  }
else
  {
    ulong s = 0, pp = itou(p);
    x = ZV_to_Flv(x, pp);
    y = ZV_to_Flv(y, pp);
    if (SMALL_ULONG(pp))
      { /* very small */
        for (i = 1; i < l; i++)
          {
            s += x[i] * y[i];
            if (s & HIGHBIT) s %= pp;
          }
        s %= pp;
      }
    else
      { /* small */
        for (i = 1; i < l; i++)
          s = Fl_add(s, Fl_mul(x[i], y[i], pp), pp);
        return utoi(s);
      }
    return utoi(s);
  }
```
In effect, we have three versions of the same code: very small, small, and arbitrary inputs. The very small and arbitrary variants use lazy reduction and reduce only when it becomes necessary: when overflow might occur (very small), and at the very end (very small, arbitrary).

### 6.2 Level 1 kernel (operations on longs, integers and reals)

**Note.** Some functions consist of an elementary operation, immediately followed by an assignment statement. They will be introduced as in the following example:

- GEN gadd[z](GEN x, GEN y[, GEN z]) followed by the explicit description of the function
- GEN gadd(GEN x, GEN y)

which creates its result on the stack, returning a GEN pointer to it, and the parts in brackets indicate that there exists also a function

- void gaddz(GEN x, GEN y, GEN z)

which assigns its result to the pre-existing object z, leaving the stack unchanged. These assignment variants are kept for backward compatibility but are inefficient: don’t use them.

#### 6.2.1 Creation

- GEN cgeti(long n) allocates memory on the PARI stack for a t_INT of length n, and initializes its first codeword. Identical to cgetg(n,t_INT).
- GEN cgetipos(long n) allocates memory on the PARI stack for a t_INT of length n, and initializes its two codewords. The sign of n is set to 1.
- GEN cgetineg(long n) allocates memory on the PARI stack for a negative t_INT of length n, and initializes its two codewords. The sign of n is set to −1.
- GEN cgetr(long n) allocates memory on the PARI stack for a t_REAL of length n, and initializes its first codeword. Identical to cgetg(n,t_REAL).
- GEN cgetc(long n) allocates memory on the PARI stack for a t_COMPLEX, whose real and imaginary parts are t_REALs of length n.
- GEN real_1(long prec) create a t_REAL equal to 1 to prec words of accuracy.
- GEN real_m1(long prec) create a t_REAL equal to −1 to prec words of accuracy.
- GEN real_0_bit(long bit) create a t_REAL equal to 0 with exponent −bit.
- GEN real_0(long prec) is a shorthand for
  
  ```c
  real_0_bit( -bit_accuracy(prec) )
  ```
- GEN int2n(long n) creates a t_INT equal to 1<<n (i.e 2^n if n ≥ 0, and 0 otherwise).
- GEN int2u(ulong n) creates a t_INT equal to 2^n.
- GEN real2n(long n, long prec) create a t_REAL equal to 2^n to prec words of accuracy.
- GEN real_m2n(long n, long prec) create a t_REAL equal to −2^n to prec words of accuracy.
- GEN strtoi(char *s) convert the character string s to a non-negative t_INT. The string s consists exclusively of digits: no leading sign, no whitespace. Leading zeroes are discarded.
- GEN strtor(char *s, long prec) convert the character string s to a non-negative t_REAL of precision prec. The string s consists exclusively of digits and optional decimal point and exponent (e or E): no leading sign, no whitespace. Leading zeroes are discarded.
6.2.2 Assignment. In this section, the z argument in the z-functions must be of type t_INT or t_REAL.

void mpaff(GEN x, GEN z) assigns x into z (where x and z are t_INT or t_REAL). Assumes that \( \lg(z) > 2 \).

void affii(GEN x, GEN z) assigns the t_INT x into the t_INT z.

void affiz(GEN x, GEN z) assigns t_INT x into t_INT or t_REAL z. Assumes that \( \lg(z) > 2 \).

void affsi(long s, GEN z) assigns the long s into the t_INT z. Assumes that \( \lg(z) > 2 \).

void affsz(long s, GEN z) assigns the long s into the t_INT or t_REAL z. Assumes that \( \lg(z) > 2 \).

void affui(ulong u, GEN z) assigns the ulong u into the t_INT z. Assumes that \( \lg(z) > 2 \).

void affur(ulong u, GEN z) assigns the ulong u into the t_REAL z. Assumes that \( \lg(z) > 2 \).

void affrr(GEN x, GEN z) assigns the t_REAL x into the t_REAL z.

void affgr(GEN x, GEN z) assigns the scalar x into the t_REAL z, if possible.

The function affrs and affri do not exist. So don’t use them.

void affrr_fixlg(GEN y, GEN z) a variant of affrr. First shorten z so that it is no longer than y, then assigns y to z. This is used in the following scenario: room is reserved for the result but, due to cancellation, fewer words of accuracy are available than had been anticipated; instead of appending meaningless 0s to the mantissa, we store what was actually computed.

Note that shortening z is not quite straightforward, since setlg(z, ly) would leave garbage on the stack, which gerepile might later inspect. It is done using

void fixlg(GEN z, long ly) see stackdummy and the examples that follow.

6.2.3 Copy.

GEN icopy(GEN x) copy relevant words of the t_INT x on the stack: the length and effective length of the copy are equal.

GEN rcopy(GEN x) copy the t_REAL x on the stack.

GEN leafcopy(GEN x) copy the leaf x on the stack (works in particular for t_INTs and t_REALs). Contrary to icopy, leafcopy preserves the original length of a t_INT. The obsolete form GEN mpcopy(GEN x) is still provided for backward compatibility.

This function also works on recursive types, copying them as if they were leaves, i.e. making a shallow copy in that case: the components of the copy point to the same data as the component of the source; see also shallowcopy.

GEN leafcopy_avma(GEN x, pari_sp av) analogous to gcopy_avma but simpler: assume x is a leaf and return a copy allocated as if initially we had avma equal to av. There is no need to pass a pointer and update the value of the second argument: the new (fictional) avma is just the return value (typecast to pari_sp).

GEN icopyspec(GEN x, long nx) copy the nx words x[2], ..., x[nx+1] to make up a new t_INT. Set the sign to 1.
6.2.4 Conversions.

GEN itor(GEN x, long prec) converts the t_INT x to a t_REAL of length prec and return the latter. Assumes that prec > 2.

long itos(GEN x) converts the t_INT x to a long if possible, otherwise raise an exception.

long itos_or_0(GEN x) converts the t_INT x to a long if possible, otherwise return 0.

int is_bigint(GEN n) true if itos(n) would succeed.

int is_bigint_lg(GEN n, long l) true if itos(n) would succeed. Assumes lgefint(n) is equal to 1.

ulong itou(GEN x) converts the t_INT |x| to an ulong if possible, otherwise raise an exception.

long itou_or_0(GEN x) converts the t_INT |x| to an ulong if possible, otherwise return 0.

GEN stoi(long s) creates the t_INT corresponding to the long s.

GEN stor(long s, long prec) converts the long s into a t_REAL of length prec and return the latter. Assumes that prec > 2.

GEN utoi(ulong s) converts the ulong s into a t_INT and return the latter.

GEN utoipos(ulong s) converts the non-zero ulong s into a t_INT and return the latter.

GEN utoineg(ulong s) converts the non-zero ulong s into the t_INT −s and return the latter.

GEN utor(ulong s, long prec) converts the ulong s into a t_REAL of length prec and return the latter. Assumes that prec > 2.

GEN rtor(GEN x, long prec) converts the t_REAL x to a t_REAL of length prec and return the latter. If prec < lg(x), round properly. If prec > lg(x), pad with zeroes. Assumes that prec > 2.

The following function is also available as a special case of mkintn:

GEN uu32toi(ulong a, ulong b) returns the GEN equal to \(2^{32}a + b\), assuming that \(a, b < 2^{32}\). This does not depend on sizeof(long): the behavior is as above on both 32 and 64-bit machines.

GEN uutoi(ulong a, ulong b) returns the GEN equal to \(2^{\text{BITS IN LONG}}a + b\).

GEN uutoineg(ulong a, ulong b) returns the GEN equal to \(-2^{\text{BITS IN LONG}}a + b\).

6.2.5 Integer parts. The following four functions implement the conversion from t_REAL to t_INT using standard rounding modes. Contrary to usual semantics (complement the mantissa with an infinite number of 0), they will raise an error precision loss in truncation if the t_REAL represents a range containing more than one integer.

GEN ceilr(GEN x) smallest integer larger or equal to the t_REAL x (i.e. the ceil function).

GEN floorr(GEN x) largest integer smaller or equal to the t_REAL x (i.e. the floor function).

GEN roundr(GEN x) rounds the t_REAL x to the nearest integer (towards +∞ in case of tie).

GEN truncr(GEN x) truncates the t_REAL x (not the same as floorr if x is negative).

The following four function are analogous, but can also treat the trivial case when the argument is a t_INT:

GEN mpceil(GEN x) as ceilr except that x may be a t_INT.
GEN mpfloor(GEN x) as floorr except that x may be a t_INT.

GEN mround(GEN x) as roundr except that x may be a t_INT.

GEN mpfrac(GEN x) as fracr except that x may be a t_INT.

GEN diviiround(GEN x, GEN y) if x and y are t_INTs, returns the quotient x/y of x and y, rounded to the nearest integer. If x/y falls exactly halfway between two consecutive integers, then it is rounded towards $+\infty$ (as for roundr).

GEN ceil_safe(GEN x), x being a real number (not necessarily a t_REAL) returns the smallest integer which is larger than any possible incarnation of x. (Recall that a t_REAL represents an interval of possible values.) Note that gceil raises an exception if the input accuracy is too low compared to its magnitude.

GEN floor_safe(GEN x), x being a real number (not necessarily a t_REAL) returns the largest integer which is smaller than any possible incarnation of x. (Recall that a t_REAL represents an interval of possible values.) Note that gfloor raises an exception if the input accuracy is too low compared to its magnitude.

GEN trunc_safe(GEN x), x being a real number (not necessarily a t_REAL) returns the integer with the largest absolute value, which is closer to 0 than any possible incarnation of x. (Recall that a t_REAL represents an interval of possible values.)

GEN roundr_safe(GEN x) rounds the t_REAL x to the nearest integer (towards $+\infty$). Complement the mantissa with an infinite number of 0 before rounding, hence never raise an exception.

6.2.6 2-adic valuations and shifts.

long vals(long s) 2-adic valuation of the long s. Returns $-1$ if s is equal to 0.

long vali(GEN x) 2-adic valuation of the t_INT x. Returns $-1$ if x is equal to 0.

GEN mpshift(GEN x, long n) shifts the t_INT or t_REAL x by n. If n is positive, this is a left shift, i.e. multiplication by $2^n$. If n is negative, it is a right shift by $-n$, which amounts to the truncation of the quotient of x by $2^{-n}$.

GEN shiftr(GEN x, long n) shifts the t_REAL x by n.

GEN shiftr_inplace(GEN x, long n) shifts the t_REAL x by n, in place.

GEN trunc2nr(GEN x, long n) given a t_REAL x, returns truncr(shiftr(x,n)), but faster, without leaving garbage on the stack and never raising a precision loss in truncation error. Called by gtrunc2n.

GEN trunc2nr_lg(GEN x, long lx, long n) given a t_REAL x, returns trunc2nr(x,n), pretending that the length of x is $lx$, which must be $\leq \lg(x)$.

GEN mantissa2nr(GEN x, long n) given a t_REAL x, returns the mantissa of $x2^n$ (disregards the exponent of x). Equivalent to

\[
\text{trunc2nr}(x, n=\exp(x)+\text{bit_prec}(x)-1)
\]

GEN mantissa_real(GEN z, long *e) returns the mantissa m of z, and sets *e to the exponent bit_accuracy(\lg(z)) - 1 - \exp(z), so that $z = m/2^e$. 

77
Low-level. In the following two functions, $s$(ource) and $t$(arget) need not be valid GENs (in practice, they usually point to some part of a t_REAL mantissa): they are considered as arrays of words representing some mantissa, and we shift globally $s$ by $n > 0$ bits, storing the result in $t$. We assume that $m \leq M$ and only access $s[m], s[m + 1], \ldots s[M]$ (read) and likewise for $t$ (write); we may have $s = t$ but more general overlaps are not allowed. The word $f$ is concatenated to $s$ to supply extra bits.

```c
void shift_left(GEN t, GEN s, long m, long M, ulong f, ulong n)
shifts the mantissa $s[m], s[m + 1], \ldots s[M], f$ left by $n$ bits.

void shift_right(GEN t, GEN s, long m, long M, ulong f, ulong n)
shifts the mantissa $f, s[m], s[m + 1], \ldots s[M]$ right by $n$ bits.
```

6.2.7 Integer valuation. For integers $x$ and $p$, such that $x \neq 0$ and $|p| > 1$, we define $v_p(x)$ to be the largest integer exponent $e$ such that $p^e$ divides $x$. If $p$ is prime, this is the ordinary valuation of $x$ at $p$.

```c
long Z_pvalrem(GEN x, GEN p, GEN *r) applied to t_INTs $x \neq 0$ and $|p| > 1$, returns $e := v_p(x)$ The quotient $x/p^e$ is returned in $*r$. If $|p|$ is a prime, $*r$ is the prime-to-$p$ part of $x$.

long Z_pval(GEN x, GEN p) as $Z_pvalrem$ but only returns $v_p(x)$.

long Z_lvalrem(GEN x, ulong p, GEN *r) as $Z_pvalrem$, except that $p$ is an ulong ($p > 1$).

long Z_lvalrem_stop(GEN *x, ulong p, int *stop) returns $e := v_p(x)$ and replaces $x$ by $x/p^e$. Set $stop$ to 1 if the new value of $x$ is $< p^2$ (and 0 otherwise). To be used when trial dividing $x$ by successive primes: the $stop$ condition is cheaply tested while testing whether $p$ divides $x$ (is the quotient less than $p^2$), and allows to decide that $n$ is prime if no prime $< p$ divides $n$. Not memory-clean.

long Z_lval(GEN x, ulong p) as $Z_pval$, except that $p$ is an ulong ($p > 1$).

long u_lvalrem(ulong x, ulong p, ulong *r) as $Z_pvalrem$, except the inputs/outputs are now ulongs.

long u_lvalrem_stop(ulong *n, ulong p, int *stop) as $Z_pvalrem_stop$.

long u_pvalrem(ulong x, GEN p, ulong *r) as $Z_pvalrem$, except $x$ and $r$ are now ulongs.

long u_lval(ulong x, ulong p) as $Z_pval$, except the inputs are now ulongs.

long u_pval(ulong x, GEN p) as $Z_pval$, except $x$ is now an ulong.

long Z_lval(long x, ulong p) as $u_lval$, for signed $x$.

long Z_lvalrem(long x, ulong p) as $u_lvalrem$, for signed $x$.

long Z_pval(long x, GEN p) as $Z_pval$, except $x$ is now a long.

long Z_pvalrem(long x, GEN p) as $Z_pvalrem$, except $x$ is now a long.

long Q_pval(GEN x, GEN p) valuation at the t_INT $p$ of the t_INT or t_FRAC $x$.
long factorial_lval(ulong n, ulong p) returns $v_p(n!)$, assuming $p$ is prime.

The following convenience functions generalize $\mathbb{Z}_p$ and its variants to “containers” ($\mathbb{ZV}$ and $\mathbb{ZX}$):

long $\mathbb{ZV}_p$valrem(GEN x, GEN p, GEN *r) $x$ being a $\mathbb{ZV}$ (a vector of $t_{\text{INT}}$s), return the min $v$ of the valuations of its components and set $*$r to $x/p^v$. Infinite loop if $x$ is the zero vector. This function is not stack clean.

long $\mathbb{ZV}_p$val(GEN x, GEN p) as $\mathbb{ZV}_p$valrem but only returns the “valuation”.

int $\mathbb{ZV}_Z$dvd(GEN x, GEN p) returns 1 if $p$ divides all components of $x$ and 0 otherwise. Faster than testing $\mathbb{ZV}_p$val($x$,$p$) $\geq$ 1.

long $\mathbb{ZV}_l$valrem(GEN x, ulong p, GEN *px) as $\mathbb{ZV}_p$valrem, except that $p$ is an $\text{ulong}$ ($p > 1$). This function is not stack-clean.

long $\mathbb{ZV}_l$val(GEN x, ulong p) as $\mathbb{ZV}_p$val, except that $p$ is an $\text{ulong}$ ($p > 1$).

long $\mathbb{ZX}_p$valrem(GEN x, GEN p, GEN *r) as $\mathbb{ZV}_p$valrem, for a $\mathbb{ZX} x$ (a $t_{\text{POL}}$ with $t_{\text{INT}}$ coefficients). This function is not stack-clean.

long $\mathbb{ZX}_p$val(GEN x, GEN p) as $\mathbb{ZX}_p$val for a $\mathbb{ZX} x$.

long $\mathbb{ZX}_l$valrem(GEN x, ulong p, GEN *px) as $\mathbb{ZV}_l$valrem, a $\mathbb{ZX} x$. This function is not stack-clean.

long $\mathbb{ZX}_l$val(GEN x, ulong p) as $\mathbb{ZX}_p$val, except that $p$ is an $\text{ulong}$ ($p > 1$).

6.2.8 Generic unary operators. Let “op” be a unary operation among

- $\text{neg}$: negation ($-x$).
- $\text{abs}$: absolute value ($|x|$).
- $\text{sqr}$: square ($x^2$).

The names and prototypes of the low-level functions corresponding to $op$ are as follows. The result is of the same type as $x$.

GEN $op_i$(GEN x) creates the result of $op$ applied to the $t_{\text{INT}} x$.

GEN $op_r$(GEN x) creates the result of $op$ applied to the $t_{\text{REAL}} x$.

GEN $mpop$(GEN x) creates the result of $op$ applied to the $t_{\text{INT}}$ or $t_{\text{REAL}} x$.

Complete list of available functions:

GEN $\text{absi}$(GEN x), GEN $\text{absr}$(GEN x), GEN $\text{mpabs}$(GEN x)

GEN $\text{negi}$(GEN x), GEN $\text{negr}$(GEN x), GEN $\text{mpneg}$(GEN x)

GEN $\text{sqri}$(GEN x), GEN $\text{sqrr}$(GEN x), GEN $\text{mpsqr}$(GEN x)

GEN $\text{absi\_shallow}$(GEN x) $x$ being a $t_{\text{INT}}$, returns a shallow copy of $|x|$, in particular returns $x$ itself when $x \geq 0$, and $\text{negi}(x)$ otherwise.

GEN $\text{mpabs\_shallow}$(GEN x) $x$ being a $t_{\text{INT}}$ or a $t_{\text{REAL}}$, returns a shallow copy of $|x|$, in particular returns $x$ itself when $x \geq 0$, and $\text{mpneg}(x)$ otherwise.

Some miscellaneous routines:

GEN $\text{sqrs}$(long x) returns $x^2$.

GEN $\text{sqru}$(ulong x) returns $x^2$. 79
6.2.9 Comparison operators.

long minss(long x, long y)
ulong minuu(ulong x, ulong y)
double mindd(double x, double y) returns the min of x and y.
long maxss(long x, long y)
ulong maxuu(ulong x, ulong y)
double maxdd(double x, double y) returns the max of x and y.

int mpcmp(GEN x, GEN y) compares the t\_INT or t\_REAL x to the t\_INT or t\_REAL y. The result is the sign of x − y.
int cmpii(GEN x, GEN y) compares the t\_INT x to the t\_INT y.
int cmpir(GEN x, GEN y) compares the t\_INT x to the t\_REAL y.
int cmpis(GEN x, long s) compares the t\_INT x to the long s.
int cmpsi(long s, GEN x) compares the long s to the t\_INT x.
int cmpsr(long s, GEN x) compares the long s to the t\_REAL x.
int cmpri(GEN x, GEN y) compares the t\_REAL x to the t\_INT y.
int cmprr(GEN x, GEN y) compares the t\_REAL x to the t\_REAL y.
int cmprs(GEN x, long s) compares the t\_REAL x to the long s.
int equalii(GEN x, GEN y) compares the t\_INTs x and y. The result is 1 if x = y, 0 otherwise.
int equalrr(GEN x, GEN y) compares the t\_REALs x and y. The result is 1 if x = y, 0 otherwise. Equality is decided according to the following rules: all real zeroes are equal, and different from a non-zero real; two non-zero reals are equal if all their digits coincide up to the length of the shortest of the two, and the remaining words in the mantissa of the longest are all 0.
int equalsi(long s, GEN x)
int equalis(GEN x, long s) compare the t\_INT x and the long s. The result is 1 if x = y, 0 otherwise.

The remaining comparison operators disregard the sign of their operands:
int equalui(ulong s, GEN x)
int equaliu(GEN x, ulong s) compare the absolute value of the t\_INT x and the ulong s. The result is 1 if |x| = y, 0 otherwise.
int cmpui(ulong u, GEN x)
int cmpiu(GEN x, ulong u) compare the absolute value of the t\_INT x and the ulong s.
int absi\_cmp(GEN x, GEN y) compares the t\_INTs x and y. The result is the sign of |x| − |y|.
int absi\_equal(GEN x, GEN y) compares the t\_INTs x and y. The result is 1 if |x| = |y|, 0 otherwise.
int absr\_cmp(GEN x, GEN y) compares the t\_REALs x and y. The result is the sign of |x| − |y|.
int absrnz_equal2n(GEN x) tests whether a non-zero \texttt{t\_REAL} \(x\) is equal to \(\pm 2^e\) for some integer \(e\).

int absrnz_equal1(GEN x) tests whether a non-zero \texttt{t\_REAL} \(x\) is equal to \(\pm 1\).

6.2.10 Generic binary operators. The operators in this section have arguments of C-type \texttt{GEN}, \texttt{long}, and \texttt{ulong}, and only \texttt{t\_INT} and \texttt{t\_REAL} \texttt{GEN}s are allowed. We say an argument is a real type if it is a \texttt{t\_REAL \texttt{GEN}}, and an integer type otherwise. The result is always a \texttt{t\_REAL} unless both \(x\) and \(y\) are integer types.

Let “\(op\)” be a binary operation among

- \textbf{add}: addition \((x + y)\).
- \textbf{sub}: subtraction \((x - y)\).
- \textbf{mul}: multiplication \((x \times y)\).
- \textbf{div}: division \((x / y)\). In the case where \(x\) and \(y\) are both integer types, the result is the Euclidean quotient, where the remainder has the same sign as the dividend \(x\). It is the ordinary division otherwise. A division-by-0 error occurs if \(y\) is equal to 0.

The last two generic operations are defined only when arguments have integer types; and the result is a \texttt{t\_INT}:

- \textbf{rem}: remainder (“\(x \ % \ y\)”). The result is the Euclidean remainder corresponding to \textbf{div}, i.e. its sign is that of the dividend \(x\).
- \textbf{mod}: true remainder \((x \ % \ y)\). The result is the true Euclidean remainder, i.e. non-negative and less than the absolute value of \(y\).

\textbf{Important technical note.} The rules given above fixing the output type (to \texttt{t\_REAL} unless both inputs are integer types) are subtly incompatible with the general rules obeyed by PARI’s generic functions, such as \texttt{gmul} or \texttt{gdiv} for instance: the latter return a result containing as much information as could be deduced from the inputs, so it is not true that if \(x\) is a \texttt{t\_INT} and \(y\) a \texttt{t\_REAL}, then \texttt{gmul}(\(x,y\)) is always the same as \texttt{mulir}(\(x,y\)). The exception is \(x = 0\), in that case we can deduce that the result is an exact 0, so \texttt{gmul} returns \texttt{gen\_0}, while \texttt{mulir} returns a \texttt{t\_REAL} 0. Specifically, the one resulting from the conversion of \texttt{gen\_0} to \texttt{t\_REAL} of precision \texttt{precision} of \(y\), multiplied by \(y\); this determines the exponent of the real 0 we obtain.

The reason for the discrepancy between the two rules is that we use the two sets of functions in different contexts: generic functions allow to write high-level code forgetting about types, letting PARI return results which are sensible and as simple as possible; type specific functions are used in kernel programming, where we do care about types and need to maintain strict consistency: it is much easier to compute the types of results when they are determined from the types of the inputs only (without taking into account further arithmetic properties, like being non-0).

The names and prototypes of the low-level functions corresponding to \(op\) are as follows. In this section, the \(z\) argument in the \(z\)-functions must be of type \(t\_INT\) when no \(r\) or \(mp\) appears in the argument code (no \texttt{t\_REAL} operand is involved, only integer types), and of type \texttt{t\_REAL} otherwise.

\texttt{GEN mpop[z]}(\texttt{GEN x, GEN y[, GEN z]}) applies \(op\) to the \texttt{t\_INT} or \texttt{t\_REAL} \(x\) and \(y\). The function \texttt{mpdivz} does not exist (its semantic would change drastically depending on the type of the \(z\) argument), and neither do \texttt{mprem[z]} nor \texttt{mpmod[z]} (specific to integers).
GEN opsi[z](long s, GEN x[, GEN z]) applies op to the long s and the t_INT x. These functions always return the global constant gen_0 (not a copy) when the sign of the result is 0.

GEN opsr[z](long s, GEN x[, GEN z]) applies op to the long s and the t_REAL x.

GEN opss[z](long s, long t[, GEN z]) applies op to the longs s and t. These functions always return the global constant gen_0 (not a copy) when the sign of the result is 0.

GEN opii[z](GEN x, GEN y[, GEN z]) applies op to the t_INT x and y. These functions always return the global constant gen_0 (not a copy) when the sign of the result is 0.

GEN opir[z](GEN x, GEN y[, GEN z]) applies op to the t_INT x and the t_REAL y.

GEN opis[z](GEN x, long s[, GEN z]) applies op to the t_INT x and the long s. These functions always return the global constant gen_0 (not a copy) when the sign of the result is 0.

GEN oprr[z](GEN x, GEN y[, GEN z]) applies op to the t_REAL x and the t_INT y.

GEN oprs[z](GEN x, long s[, GEN z]) applies op to the t_REAL x and the long s.

Some miscellaneous routines:

long expu(ulong x) assuming x > 0, returns the binary exponent of the real number equal to x. This is a special case of gexpo.

GEN adduu(ulong x, ulong y)
GEN addiu(GEN x, ulong y)
GEN addui(ulong x, GEN y) adds x and y.
GEN subuu(ulong x, ulong y)
GEN subiu(GEN x, ulong y)
GEN subui(ulong x, GEN y) subtracts x by y.
GEN muluu(ulong x, ulong y) multiplies x by y.
GEN mului(ulong x, GEN y) multiplies x by y.
GEN muluuui(ulong x, ulong y, GEN z) return xyz.
GEN muluui(ulong x, ulong y, GEN z) return x + yu.
GEN muliuui(ulong x, GEN y, GEN z) returns x + yz.
GEN muliu(GEN x, GEN y) multiplies x by y.

void addmului(ulong a, ulong b, GEN x) return a + b[X].
GEN addmuliu(GEN x, GEN y, ulong u) returns x + yu.
GEN addmulii(GEN x, GEN y, GEN z) returns x + yz.
GEN addmulii_inplace(GEN x, GEN y, GEN z) returns x + yz, but returns x itself and not a copy if yz = 0. Not suitable for gerepile or gerepileupto.
GEN addmuliu_inplace(GEN x, GEN y, ulong u) returns x + yu, but returns x itself and not a copy if yu = 0. Not suitable for gerepile or gerepileupto.
GEN submuliu_inplace(GEN x, GEN y, ulong u) returns x − yu, but returns x itself and not a copy if yu = 0. Not suitable for gerepile or gerepileupto.
GEN lincombii(GEN u, GEN v, GEN x, GEN y) returns ux + vy.
GEN mulsubii(GEN y, GEN z, GEN x) returns \(yz - x\).
GEN submulii(GEN x, GEN y, GEN z) returns \(x - yz\).
GEN submuliu(GEN x, GEN y, ulong u) returns \(x - yu\).
GEN mulu_interval(ulong a, ulong b) returns \(a(a + 1) \cdots b\), assuming that \(a \leq b\). Very inefficient when \(a = 0\).
GEN invr(GEN x) returns the inverse of the non-zero t_REAL \(x\).
GEN truedivii(GEN x, GEN y) returns the true Euclidean quotient (with non-negative remainder less than \(|y|\)).
GEN truedivis(GEN x, long y) returns the true Euclidean quotient (with non-negative remainder less than \(|y|\)).
GEN truedivsi(long x, GEN y) returns the true Euclidean quotient (with non-negative remainder less than \(|y|\)).
GEN centermodii(GEN x, GEN y, GEN y2), given t_INTs \(x\), \(y\), returns \(z\) congruent to \(x\) modulo \(y\), such that \(-y/2 \leq z < y/2\). The function requires an extra argument \(y2\), such that \(y2 = \text{shifti}(y, -1)\). (In most cases, \(y\) is constant for many reductions and \(y2\) need only be computed once.)
GEN remi2n(GEN x, long n) returns \(x \mod 2^n\).
GEN addii_sign(GEN x, long sx, GEN y, long sy) add the t_INTs \(x\) and \(y\) as if their signs were \(sx\) and \(sy\).
GEN addir_sign(GEN x, long sx, GEN y, long sy) add the t_INT \(x\) and the t_REAL \(y\) as if their signs were \(sx\) and \(sy\).
GEN addr_sign(GEN x, long sx, GEN y, long sy) add the t_REALs \(x\) and \(y\) as if their signs were \(sx\) and \(sy\).
GEN addsi_sign(long x, GEN y, long sy) add \(x\) and the t_INT \(y\) as if its sign was \(sy\).
GEN addui_sign(ulong x, GEN y, long sy) add \(x\) and the t_INT \(y\) as if its sign was \(sy\).

6.2.11 Exact division and divisibility.
GEN diviiexact(GEN x, GEN y) returns the Euclidean quotient \(x/y\), assuming \(y\) divides \(x\). Uses Jebelean algorithm (Jebelean-Krandick bidirectional exact division is not implemented).
GEN diviuexact(GEN x, ulong y) returns the Euclidean quotient \(x/y\), assuming \(y\) divides \(x\) and \(y\) is non-zero.
GEN diviuuexact(GEN x, ulong y, ulong z) returns the Euclidean quotient \(x/(yz)\), assuming \(yz\) divides \(x\) and \(yz \neq 0\).

The following routines return 1 (true) if \(y\) divides \(x\), and 0 otherwise. (Error if \(y\) is 0, even if \(x\) is 0.) All GEN are assumed to be t_INTS:
int dvii(GEN x, GEN y), int dvdis(GEN x, long y), int dvdiu(GEN x, ulong y),
int dvdisi(long x, GEN y), int dvdui(ulong x, GEN y).

The following routines return 1 (true) if \(y\) divides \(x\), and in that case assign the quotient to \(z\); otherwise they return 0. All GEN are assumed to be t_INTS:
int dvdiiz(GEN x, GEN y, GEN z), int dvdisz(GEN x, long y, GEN z).

int dvdiuz(GEN x, ulong y, GEN z) if y divides x, assigns the quotient |x|/y to z and returns 1 (true), otherwise returns 0 (false).

6.2.12 Division with integral operands and t_REAL result.

GEN rdivii(GEN x, GEN y, long prec), assuming x and y are both of type t_INT, return the quotient x/y as a t_REAL of precision prec.

GEN rdiviiiz(GEN x, GEN y, GEN z), assuming x and y are both of type t_INT, and z is a t_REAL, assign the quotient x/y to z.

GEN rdivis(GEN x, long y, long prec), assuming x is of type t_INT, return the quotient x/y as a t_REAL of precision prec.

GEN rdivsi(long x, GEN y, long prec), assuming y is of type t_INT, return the quotient x/y.

GEN rdivss(long x, long y, long prec), return the quotient x/y as a t_REAL of precision prec.

6.2.13 Division with remainder. The following functions return two objects, unless specifically asked for only one of them — a quotient and a remainder. The quotient is returned and the remainder is returned through the variable whose address is passed as the r argument. The term true Euclidean remainder refers to the non-negative one (mod), and Euclidean remainder by itself to the one with the same sign as the dividend (rem). All GENs, whether returned directly or through a pointer, are created on the stack.

GEN dvmdii(GEN x, GEN y, GEN *r) returns the Euclidean quotient of the t_INT x by a t_INT y and puts the remainder into *r. If r is equal to NULL, the remainder is not created, and if r is equal to ONLYREM, only the remainder is created and returned. In the generic case, the remainder is created after the quotient and can be disposed of individually with a cgiv(r). The remainder is always of the sign of the dividend x. If the remainder is 0 set r = gen_0.

void dvmdiiiz(GEN x, GEN y, GEN z, GEN t) assigns the Euclidean quotient of the t_INTs x and y into the t_INT z, and the Euclidean remainder into the t_INT t.

Analogous routines dvmdis[z], dvmdsi[z], dvmdss[z] are available, where s denotes a long argument. But the following routines are in general more flexible:

long sdivss_rem(long s, long t, long *r) computes the Euclidean quotient and remainder of the longs s and t. Puts the remainder into *r, and returns the quotient. The remainder is of the sign of the dividend s, and has strictly smaller absolute value than t.

long sdivsi_rem(long s, GEN x, long *r) computes the Euclidean quotient and remainder of the long s by the t_INT x. As sdivss_rem otherwise.

long sdivsi(long s, GEN x) as sdivsi_rem, without remainder.

GEN divis_rem(GEN x, long s, long *r) computes the Euclidean quotient and remainder of the t_INT x by the long s. As sdivss_rem otherwise.

GEN diviu_rem(GEN x, ulong s, ulong *r) computes the Euclidean quotient and remainder of absolute value of the t_INT x by the ulong s. As sdivss_rem otherwise.

ulong udiviu_rem(GEN n, ulong d, ulong *r) as diviu_rem, assuming that |n|/d fits into an ulong.
ulong udivui_rem(ulong x, GEN y, ulong *rem) computes the Euclidean quotient and remainder of $x$ by $y$. As sdivss_rem otherwise.

ulong udivuu_rem(ulong x, ulong y, ulong *rem) computes the Euclidean quotient and remainder of $x$ by $y$. As sdivss_rem otherwise.

GEN divsi_rem(long s, GEN y, long *r) computes the Euclidean quotient and remainder of the t_long $s$ by the GEN $y$. As sdivss_rem otherwise.

GEN divss_rem(long x, long y, long *r) computes the Euclidean quotient and remainder of the t_long $x$ by the long $y$. As sdivss_rem otherwise.

GEN truedvmdii(GEN x, GEN y, GEN *r), as dvmdii but with a non-negative remainder.

GEN truedvmdis(GEN x, long y, GEN *z), as dvmdis but with a non-negative remainder.

GEN truedvmdsi(long x, GEN y, GEN *z), as dvmdsi but with a non-negative remainder.

6.2.14 Modulo to longs. The following variants of modii do not clutter the stack:

long smodis(GEN x, long y) computes the true Euclidean remainder of the t_INT $x$ by the long $y$. This is the non-negative remainder, not the one whose sign is the sign of $x$ as in the div functions.

long smodss(long x, long y) computes the true Euclidean remainder of the long $x$ by a long $y$.

ulong umodiu(GEN x, ulong y) computes the true Euclidean remainder of the t_INT $x$ by the ulong $y$.

ulong umodui(ulong x, GEN y) computes the true Euclidean remainder of the ulong $x$ by the t_INT $|y|$.

The routine smodi does not exist, since it would not always be defined: for a negative $x$, if the quotient is $\pm 1$, the result $x + |y|$ would in general not fit into a long. Use either umodui or modsi.

6.2.15 Powering, Square root.

GEN powii(GEN x, GEN n), assumes $x$ and $n$ are t_INTs and returns $x^n$.

GEN powuu(ulong x, ulong n), returns $x^n$.

GEN powiu(GEN x, ulong n), assumes $x$ is a t_INT and returns $x^n$.

GEN powis(GEN x, long n), assumes $x$ is a t_INT and returns $x^n$ (possibly a t_FRAC if $n < 0$).

GEN powrs(GEN x, long n), assumes $x$ is a t_REAL and returns $x^n$. This is considered as a sequence of mulrr, possibly empty: as such the result has type t_REAL, even if $n = 0$. Note that the generic function gpowgs(x,0) would return gen_1, see the technical note in Section 6.2.10.

GEN powru(GEN x, ulong n), assumes $x$ is a t_REAL and returns $x^n$ (always a t_REAL, even if $n = 0$).

GEN powruvec(GEN e, ulong n). Given a t_REAL $e$, return the vector of all $e^i$, $1 \leq i \leq n$.

GEN powrshalf(GEN x, long n), assumes $x$ is a t_REAL and returns $x^{n/2}$ (always a t_REAL, even if $n = 0$).
GEN powruhalf(GEN x, ulong n), assumes $x$ is a t_REAL and returns $x^{n/2}$ (always a t_REAL, even if $n = 0$).

GEN powrfrac(GEN x, long n, long d), assumes $x$ is a t_REAL and returns $x^{n/d}$ (always a t_REAL, even if $n = 0$).

GEN powIs(long n) returns $I^n \in \{1, I, -1, -I\}$ (t_INT for even $n$, t_COMPLEX otherwise).

ulong upowuu(ulong x, ulong n), returns $x^n$ when $\lt 2^{\text{BITS_IN_LONG}}$, and 0 otherwise (overflow).

GEN sqrtremi(GEN N, GEN *r), returns the integer square root $S$ of the non-negative t_INT $N$ (rounded towards 0) and puts the remainder $R$ into *r. Precisely, $N = S^2 + R$ with $0 \leq R \leq 2S$. If *r is equal to NULL, the remainder is not created. In the generic case, the remainder is created after the quotient and can be disposed of individually with cgiv(R). If the remainder is 0 set $R = \text{gen}_0$.

Uses a divide and conquer algorithm (discrete variant of Newton iteration) due to Paul Zimmermann ("Karatsuba Square Root", INRIA Research Report 3805 (1999)).

GEN sqrti(GEN N), returns the integer square root $S$ of the non-negative t_INT $N$ (rounded towards 0). This is identical to sqrtremi(N, NULL).

6.2.16 GCD, extended GCD and LCM.

long cgcd(long x, long y) returns the GCD of $x$ and $y$.

ulong cgcd(ulong x, ulong y) returns the GCD of $x$ and $y$.

long clcm(long x, long y) returns the LCM of $x$ and $y$, provided it fits into a long. Silently overflows otherwise.

GEN gcdii(GEN x, GEN y), returns the GCD of the t_INTs $x$ and $y$.

GEN lcmii(GEN x, GEN y), returns the LCM of the t_INTs $x$ and $y$.

GEN bezout(GEN a,GEN b, GEN *u,GEN *v), returns the GCD $d$ of t_INTs $a$ and $b$ and sets $u$, $v$ to the Bezout coefficients such that $au + bv = d$.

long cbbezout(long a,long b, long *u,long *v), returns the GCD $d$ of $a$ and $b$ and sets $u$, $v$ to the Bezout coefficients such that $au + bv = d$.

GEN ZV_gcdext(GEN A) given a vector of $n$ integers $A$, returns $[d,U]$, where $d$ is the GCD of the $A[i]$ and $U$ is a matrix in $\text{GL}_n(\mathbb{Z})$ such that $AU = [0, \ldots, 0, D]$.

6.2.17 Pseudo-random integers. These routine return pseudo-random integers uniformly distributed in some interval. The all use the same underlying generator which can be seeded and restarted using getrand and setrand.

void setrand(GEN seed) reseeds the random number generator using the seed $n$. The seed is either a technical array output by getrand or a small positive integer, used to generate deterministically a suitable state array. For instance, running a randomized computation starting by setrand(1) twice will generate the exact same output.

GEN getrand(void) returns the current value of the seed used by the pseudo-random number generator random. Useful mainly for debugging purposes, to reproduce a specific chain of computations. The returned value is technical (reproduces an internal state array of type t_VECSMALL), and can only be used as an argument to setrand.
ulong pari_rand(void) returns a random $0 \leq x < 2^\text{BITS\_IN\_LONG}$.

long random_bits(long k) returns a random $0 \leq x < 2^k$. Assumes that $0 \leq k \leq \text{BITS\_IN\_LONG}$.

ulong random_Fl(ulong p) returns a pseudo-random integer in $0, 1, \ldots, p - 1$.

GEN randomi(GEN n) returns a random $\text{t\_INT}$ between $0$ and $n - 1$.

GEN randomr(long prec) returns a random $\text{t\_REAL}$ in $[0, 1[$, with precision $\text{prec}$.

6.2.18 Modular operations. In this subsection, all GENs are $\text{t\_INT}$

GEN Fp_red(GEN a, GEN m) returns a modulo $m$ (smallest non-negative residue). (This is identical to modii).

GEN Fp_neg(GEN a, GEN m) returns $-a$ modulo $m$ (smallest non-negative residue).

GEN Fp_add(GEN a, GEN b, GEN m) returns the sum of $a$ and $b$ modulo $m$ (smallest non-negative residue).

GEN Fp_sub(GEN a, GEN b, GEN m) returns the difference of $a$ and $b$ modulo $m$ (smallest non-negative residue).

GEN Fp_center(GEN a, GEN p, GEN pov2) assuming that $pov2$ is shifti($p, -1$) and that $a$ is between $0$ and $p - 1$ and, returns the representative of $a$ in the symmetric residue system.

GEN Fp_mul(GEN a, GEN b, GEN m) returns the product of $a$ by $b$ modulo $m$ (smallest non-negative residue).

GEN Fp_addmul(GEN x, GEN y, GEN z, GEN p) returns $x + yz$.

GEN Fp_mulu(GEN a, ulong b, GEN m) returns the product of $a$ by $b$ modulo $m$ (smallest non-negative residue).

GEN Fp_muls(GEN a, long b, GEN m) returns the product of $a$ by $b$ modulo $m$ (smallest non-negative residue).

GEN Fp_sqr(GEN a, GEN m) returns $a^2$ modulo $m$ (smallest non-negative residue).

ulong Fp_powu(GEN x, ulong n, GEN m) raises $x$ to the $n$-th power modulo $m$ (smallest non-negative residue). Not memory-clean, but suitable for gerepileupto.

ulong Fp_pows(GEN x, long n, GEN m) raises $x$ to the $n$-th power modulo $m$ (smallest non-negative residue). A negative $n$ is allowed Not memory-clean, but suitable for gerepileupto.

GEN Fp_inv(GEN a, GEN m) returns an inverse of $a$ modulo $m$ (smallest non-negative residue). Raise an error if $a$ is not invertible.

GEN Fp_invsafe(GEN a, GEN m) as Fp_inv, but return NULL if $a$ is not invertible.

GEN FpV_inv(GEN x, GEN m) $x$ being a vector of $\text{t\_INTs}$, return the vector of inverses of the $x[i]$ mod $m$. The routine uses Montgomery’s trick, and involves a single inversion mod $m$, plus $3(N - 1)$ multiplications for $N$ entries. The routine is not stack-clean: $2N$ integers mod $m$ are left on stack, besides the $N$ in the result.

GEN Fp_div(GEN a, GEN b, GEN m) returns the quotient of $a$ by $b$ modulo $m$ (smallest non-negative residue). Raise an error if $b$ is not invertible.
int invmod(GEN a, GEN m, GEN *g), return 1 if a modulo m is invertible, else return 0 and set g = gcd(a,m).

GEN Fp_log(GEN a, GEN g, GEN ord, GEN p) Let g such that \(g^{ord} \equiv 1 \pmod{p}\). Return an integer e such that \(a^e \equiv g \pmod{p}\). If e does not exist, the result is currently undefined.

GEN Fp_order(GEN a, GEN N, GEN p) returns the order of the Fp a. If N is non-NULL, it is assumed that N is a multiple of the order of a, as a t_INT or a factorization matrix.

GEN Fp_factored_order(GEN a, GEN N, GEN p) returns \((o, F)\), where o is the multiplicative order of the Fp a in \(F_p^*\), and F is the factorization of o. If N is non-NULL, it is assumed that N is a multiple of the order of a, as a t_INT or a factorization matrix.

int Fp_issquare(GEN x, GEN p) returns 1 if x is a square modulo p, and 0 otherwise.

int Fp_ispower(GEN x, GEN n, GEN p) returns 1 if x is an n-th power modulo p, and 0 otherwise.

GEN Fp_sqrt(GEN x, GEN p) returns a square root of x modulo p (the smallest non-negative residue), where x, p are t_INTs, and p is assumed to be prime. Return NULL if x is not a quadratic residue modulo p.

GEN Fp_sqrtn(GEN x, GEN n, GEN p, GEN *zn) returns an n-th root of x modulo p (smallest non-negative residue), where x, n, p are t_INTs, and p is assumed to be prime. Return NULL if x is not an n-th power residue. Otherwise, if zn is non-NULL set it to a primitive n-th root of 1.

GEN Zn_sqrt(GEN x, GEN n) returns one of the square roots of x modulo n (possibly not prime), where x is a t_INT and n is either a t_INT or is given by its factorisation matrix. Return NULL if no such square root exist.

long kross(long x, long y) returns the Kronecker symbol \((x|y)\), i.e. \(-1, 0 or 1\). If y is an odd prime, this is the Legendre symbol. (Contrary to krouu, kross also supports y = 0)

long krouu(ulong x, ulong y) returns the Kronecker symbol \((x|y)\), i.e. \(-1, 0 or 1\). Assumes y is non-zero. If y is an odd prime, this is the Legendre symbol.

long krois(GEN x, long y) returns the Kronecker symbol \((x|y)\) of t_INT x and long y. As kross otherwise.

long kroiu(GEN x, ulong y) returns the Kronecker symbol \((x|y)\) of t_INT x and non-zero ulong y. As krouu otherwise.

long krosi(long x, GEN y) returns the Kronecker symbol \((x|y)\) of long x and t_INT y. As kross otherwise.

long kronecker(GEN x, GEN y) returns the Kronecker symbol \((x|y)\) of t_INTs x and y. As kross otherwise.

GEN pgener_Fp(GEN p) returns the smallest primitive root modulo p, assuming p is prime.

GEN pgener_Zp(GEN p) returns the smallest primitive root modulo \(p^k\), \(k > 1\), assuming p is an odd prime.

long Zp_issquare(GEN x, GEN p) returns 1 if the t_INT x is a p-adic square, 0 otherwise.

long Zn_issquare(GEN x, GEN n) returns 1 if t_INT x is a square modulo n (possibly not prime), where n is either a t_INT or is given by its factorisation matrix. Return 0 otherwise.
long Zn_ispower(GEN x, GEN n, GEN K, GEN *py) returns 1 if t_INT x is a K-th power modulo n (possibly not prime), where n is either a t_INT or is given by its factorisation matrix. Return 0 otherwise. If py is not NULL, set it to y such that \( y^K = x \) modulo n.

GEN pgener_Fp_local(GEN p, GEN L), L being a vector of primes dividing \( p-1 \), returns the smallest integer \( x > 1 \) which is a generator of the \( \ell \)-Sylow of \( F_p^* \) for every \( \ell \) in L. In other words, \( x^{(p-1)/\ell} \neq 1 \) for all such \( \ell \). In particular, returns \( \text{pgener_Fp}(p) \) if L contains all primes dividing \( p-1 \). It is not necessary, and in fact slightly inefficient, to include \( \ell = 2 \), since 2 is treated separately in any case, i.e. the generator obtained is never a square.

GEN rootsof1_Fp(GEN n, GEN p) returns a primitive n-th root modulo the prime p.

GEN rootsoflu_Fp(ulong n, GEN p) returns a primitive n-th root modulo the prime p.

ulong rootsof1_Fl(ulong n, ulong p) returns a primitive n-th root modulo the prime p.

6.2.19 Extending functions to vector inputs.

The following functions apply \( f \) to the given arguments, recursively if they are of vector / matrix type:

GEN map_proto_G(GEN (*f)(GEN), GEN x) For instance, if \( x \) is a t_VEC, return a t_VEC whose components are the \( f(x[i]) \).

GEN map_proto_lG(long (*f)(GEN), GEN x) As above, applying the function \text{stoi}( f() ).

GEN map_proto_GL(GEN (*f)(GEN,long), GEN x, long y)

GEN map_proto_lGL(long (*f)(GEN,long), GEN x, long y)

In the last function, \( f \) implements an associative binary operator, which we extend naturally to an \( n \)-ary operator \( f_n \) for any \( n \): by convention, \( f_0() = 1, f_1(x) = x \), and

\[
f_n(x_1, \ldots, x_n) = f(f_{n-1}(x_1, \ldots, x_{n-1}), x_n),
\]

for \( n \geq 2 \).

GEN gassoc_proto(GEN (*f)(GEN,GEN), GEN x, GEN y) If \( y \) is not NULL, return \( f(x, y) \). Otherwise, \( x \) must be of vector type, and we return the result of \( f \) applied to its components, computed using a divide-and-conquer algorithm. More precisely, return

\[
f(f(x_1, \text{NULL}), f(x_2, \text{NULL})),
\]

where \( x_1, x_2 \) are the two halves of \( x \).
6.2.20 Miscellaneous arithmetic functions.

ulong coreu(ulong n), unique squarefree integer $d$ dividing $n$ such that $n/d$ is a square.
ulong eulerphiu(ulong n), Euler’s totient function of $n$.
ulong eulerphiu_fact(GEN fa), Euler’s totient function of the ulong $n$, where $fa$ is factoru(n).
long moebiusu(ulong n), Moebius $\mu$-function of $n$.
GEN divisorsu(ulong n), returns the divisors of $n$ in a t_VECSMALL, sorted by increasing order.
long uissquarefree(ulong n) returns 1 if $n$ is square-free, and 0 otherwise.
ulong uissquarefree_fact(GEN fa) returns uissquarefree(n), where fa is factoru(n).
long uposisfundamental(ulong x) return 1 is $x$ is a fundamental discriminant, and 0 otherwise.
long unegisfundamental(ulong x) return 1 is $-x$ is a fundamental discriminant, and 0 otherwise.
int uis_357_power(ulong x, ulong *pt, ulong *mask) as is_357_power for ulong $x$.
int uis_357_powermod(ulong x, ulong *mask) as uis_357_power, but only check for 3rd, 5th or 7th powers modulo $211 \times 209 \times 61 \times 203 \times 117 \times 31 \times 43 \times 71$.
long uisprimepower(ulong n, ulong *p) as isprimepower, for ulong $n$.
int uislucaspsp(ulong n) returns 1 if the ulong $n$ fails Lucas compositeness test (it thus may be prime or composite), and 0 otherwise (proving that $n$ is composite).
ulong sumdigitsu(ulong n) returns the sum of decimal digits of $u$.
GEN usumdivkvec(ulong n, GEN K) $K$ being a t_VECSMALL of positive integers. Returns the vector of sumdivk($n, K[i]$).
GEN hilbertii(GEN x, GEN y, GEN p), returns the Hilbert symbol $(x, y)$ at the prime $p$ (NULL for the place at infinity); $x$ and $y$ are t_INTs.
GEN sumdedekind(GEN h, GEN k) returns the Dedekind sum associated to the t_INT $h$ and $k$, $k > 0$.
GEN sumdedekind_coprime(GEN h, GEN k) as sumdedekind, except that $h$ and $k$ are assumed to be coprime t_INTs.
GEN u_sumdedekind_coprime(long h, long k) Let $k > 0$, $0 \leq h < k$, $(h, k) = 1$. Returns $[s_1, s_2]$ in a t_VECSMALL, such that $s(h, k) = (s_2 + ks_1)/(12k)$. Requires $max(h + k/2, k) < \text{LONG\_MAX}$ to avoid overflow, in particular $k \leq (2/3)\text{LONG\_MAX}$ is fine.
Chapter 7:  
Level 2 kernel

These functions deal with modular arithmetic, linear algebra and polynomials where assumptions can be made about the types of the coefficients.

7.1 Naming scheme.

A function name is built in the following way: $A_1 \ldots A_n \text{fun}$ for an operation $\text{fun}$ with $n$ arguments of class $A_1, \ldots, A_n$. A class name is given by a base ring followed by a number of code letters. Base rings are among

- $\mathbb{Z}/l\mathbb{Z}$ where $l < 2^{\text{BITS-IN-LONG}}$ is not necessarily prime. Implemented using u1ongs
- $\mathbb{Z}/p\mathbb{Z}$ where $p$ is a t_INT, not necessarily prime. Implemented as t_INTs $z$, preferably satisfying $0 \leq z < p$. More precisely, any t_INT can be used as an Fp, but reduced inputs are treated more efficiently. Outputs from Fpxxx routines are reduced.
- $\mathbb{Z}[X]/(p, T(X))$, $p$ a t_INT, $T$ a t_POL with Fp coefficients or NULL (in which case no reduction modulo $T$ is performed). Implemented as t_POLs $z$ with Fp coefficients, $\deg(z) < \deg T$, although $z$ a t_INT is allowed for elements in the prime field.
- $\mathbb{Z}$: the integers $\mathbb{Z}$, implemented as t_INTs.
- $\mathbb{Z}$: the integers $\mathbb{Z}$, implemented using (signed) longs.
- $\mathbb{Q}$: the rational numbers $\mathbb{Q}$, implemented as t_INTs and t_FRACs.
- $\mathbb{Rg}$: a commutative ring, whose elements can be gadd-ed, gmul-ed, etc.

Possible letters are:

- $X$: polynomial in $X$ (t_POL in a fixed variable), e.g. FpX means $\mathbb{Z}/p\mathbb{Z}[X]$
- $Y$: polynomial in $Y \neq X$. This is used to resolve ambiguities. E.g. FpXY means $((\mathbb{Z}/p\mathbb{Z})[X])[Y]$.
- $V$: vector (t_VEC or t_COL), treated as a line vector (independently of the actual type). E.g. ZV means $\mathbb{Z}^k$ for some $k$.
- $C$: vector (t_VEC or t_COL), treated as a column vector (independently of the actual type). The difference with $V$ is purely semantic: if the result is a vector, it will be of type t_COL unless mentioned otherwise. For instance the function ZC_add receives two integral vectors (t_COL or t_VEC, possibly different types) of the same length and returns a t_COL whose entries are the sums of the input coefficients.
- $M$: matrix (t_MAT). E.g. QM means a matrix with rational entries
- $T$: Trees. Either a leaf or a t_VEC of trees.
- $E$: point over an elliptic curve, represented as two-component vectors $[x, y]$, except for the represented by the one-component vector [0]. Not all curve models are supported.
- $Q$: representative (t_POL) of a class in a polynomial quotient ring. E.g. an FpXQ belongs to $(\mathbb{Z}/p\mathbb{Z})[X]/(T(X))$, FpXQV means a vector of such elements, etc.
x, y, m, v, c, q: as their uppercase counterpart, but coefficient arrays are implemented using t_VECSMALLs, which coefficient understood as ulongs.

x and y (and q) are implemented by a t_VECSMALL whose first coefficient is used as a code-word and the following are the coefficients, similarly to a t_POL. This is known as a 'POLSMALL'.

m are implemented by a t_MAT whose components (columns) are t_VECSMALLs. This is known as a 'MATSMALL'.

v and c are regular t_VECSMALLs. Difference between the two is purely semantic. Omitting the letter means the argument is a scalar in the base ring. Standard functions fun are

add: add
sub: subtract
mul: multiply
sqr: square
div: divide (Euclidean quotient)
rem: Euclidean remainder
divrem: return Euclidean quotient, store remainder in a pointer argument. Three special values of that pointer argument modify the default behavior: NULL (do not store the remainder, used to implement div), ONLY_REM (return the remainder, used to implement rem), ONLY_DIVIDES (return the quotient if the division is exact, and NULL otherwise).
gcd: GCD
extgcd: return GCD, store Bezout coefficients in pointer arguments
pow: exponentiate
eval: evaluation / composition

7.2 Modular arithmetic.

These routines implement univariate polynomial arithmetic and linear algebra over finite fields, in fact over finite rings of the form \((Z/pZ)[X]/(T)\), where \(p\) is not necessarily prime and \(T \in (Z/pZ)[X]\) is possibly reducible; and finite extensions thereof. All this can be emulated with t_INTMOD and t_POLMOD coefficients and using generic routines, at a considerable loss of efficiency. Also, specialized routines are available that have no obvious generic equivalent.

7.2.1 FpC/ FpV, FpM. A ZV (resp. a ZM) is a t_VEC or t_COL (resp. t_MAT) with t_INT coefficients. An FpV or FpM, with respect to a given t_INT \(p\), is the same with Fp coordinates; operations are understood over \(Z/pZ\).
7.2.1.1 Conversions.

int Rg_is_Fp(GEN z, GEN *p), checks if \( z \) can be mapped to \( \mathbb{Z}/p\mathbb{Z} \): a t_INT or a t_INTMOD whose modulus is equal to \(*p\), (if \(*p\) not NULL), in that case return 1, else 0. If a modulus is found it is put in \(*p\), else \(*p\) is left unchanged.

int RgV_is_FpV(GEN z, GEN *p), \( z \) a t_VEC (resp. t_COL), checks if it can be mapped to a FpV (resp. FpC), by checking Rg_is_Fp coefficientwise.

int RgM_is_FpM(GEN z, GEN *p), \( z \) a t_MAT, checks if it can be mapped to a FpM, by checking RgV_is_FpV columnwise.

GEN Rg_to_Fp(GEN z, GEN p), \( z \) a scalar which can be mapped to \( \mathbb{Z}/p\mathbb{Z} \): a t_INT, a t_INTMOD whose modulus is divisible by \( p \), a t_FRAC whose denominator is coprime to \( p \), or a t_PADIC with underlying prime \( \ell \) satisfying \( p = \ell^n \) for some \( n \) (less than the accuracy of the input). Returns lift(\( z \bullet \text{Mod}(1,p) \)), normalized.

GEN padic_to_Fp(GEN x, GEN p) special case of Rg_to_Fp, for a \( x \) a t_PADIC.

GEN RgV_to_FpV(GEN z, GEN p), \( z \) a t_VEC or t_COL, returns the FpV (as a t_VEC) obtained by applying Rg_to_Fp coefficientwise.

GEN RgC_to_FpC(GEN z, GEN p), \( z \) a t_VEC or t_COL, returns the FpC (as a t_COL) obtained by applying Rg_to_Fp coefficientwise.

GEN RgM_to_FpM(GEN z, GEN p), \( z \) a t_MAT, returns the FpM obtained by applying RgC_to_FpC columnwise.

GEN RgM_Fp_init(GEN z, GEN p, ulong *pp), given an RgM \( z \), whose entries can be mapped to \( \mathbb{F}_p \) (as per Rg_to_Fp), and a prime number \( p \). This routine returns a normal form of \( z \): either an F2m (\( p = 2 \)), an Flm (\( p \) fits into an ulong) or an FpM. In the first two cases, \( pp \) is set to itou(\( p \)), and to 0 in the last.

The functions above are generally used as follow:

GEN add(GEN x, GEN y)
{
    GEN p = NULL;
    if (Rg_is_Fp(x, &p) && Rg_is_Fp(y, &p) && p)
    {
        x = Rg_to_Fp(x, p); y = Rg_to_Fp(y, p);
        z = Fp_add(x, y, p);
        return Fp_to_mod(z);
    }
    else return gadd(x, y);
}

GEN FpC_red(GEN z, GEN p), \( z \) a ZC. Returns lift(Col(\( z \)) * Mod(1,p)), hence a t_COL.

GEN FpV_red(GEN z, GEN p), \( z \) a ZV. Returns lift(Vec(\( z \)) * Mod(1,p)), hence a t_VEC

GEN FpM_red(GEN z, GEN p), \( z \) a ZM. Returns lift(\( z \bullet \text{Mod}(1,p) \)), which is an FpM.
7.2.1.2 Basic operations.

GEN FpC_center(GEN z, GEN p, GEN pov2) returns a t_COL whose entries are the Fp_center of the gel(z,i).

GEN FpM_center(GEN z, GEN p, GEN pov2) returns a matrix whose entries are the Fp_center of the gcoeff(z,i,j).

GEN FpC_add(GEN x, GEN y, GEN p) adds the ZC x and y and reduce modulo p to obtain an FpC.

GEN FpV_add(GEN x, GEN y, GEN p) same as FpC_add, returning and FpV.

GEN FpC_sub(GEN x, GEN y, GEN p) subtracts the ZC y to the ZC x and reduce modulo p to obtain an FpC.

GEN FpV_sub(GEN x, GEN y, GEN p) same as FpC_sub, returning and FpV.

GEN FpC_Fp_mul(GEN x, GEN y, GEN p) multiplies the ZC x (seen as a column vector) by the t_INT y and reduce modulo p to obtain an FpC.

GEN FpC_FpV_mul(GEN x, GEN y, GEN p) multiplies the ZC x (seen as a column vector) by the ZV y (seen as a row vector, assumed to have compatible dimensions), and reduce modulo p to obtain an FpM.

GEN FpM_mul(GEN x, GEN y, GEN p) multiplies the two ZM s x and y (assumed to have compatible dimensions), and reduce modulo p to obtain an FpM.

GEN FpM_powu(GEN x, ulong n, GEN p) computes $x^n$ where x is a square FpM.

GEN FpM_FpC_mul(GEN x, GEN y, GEN p) multiplies the ZM x by the ZC y (seen as a column vector, assumed to have compatible dimensions), and reduce modulo p to obtain an FpC.

GEN FpM_FpC_mul_FpX(GEN x, GEN y, GEN p, long v) is a memory-clean version of

GEN tmp = FpM_FpC_mul(x,y,p);
return RgV_to_RgX(tmp, v);

GEN FpV_FpC_mul(GEN x, GEN y, GEN p) multiplies the ZV x (seen as a row vector) by the ZC y (seen as a column vector, assumed to have compatible dimensions), and reduce modulo p to obtain an Fp.

GEN FpV_dotproduct(GEN x, GEN y, GEN p) scalar product of x and y (assumed to have the same length).

GEN FpV_dotsquare(GEN x, GEN p) scalar product of x with itself. has t_INT entries.

7.2.1.3 Fp-linear algebra. The implementations are not asymptotically efficient ($O(n^3)$ standard algorithms).

GEN FpM_deplin(GEN x, GEN p) returns a non-trivial kernel vector, or NULL if none exist.

GEN FpM_det(GEN x, GEN p) as det

GEN FpM_gauss(GEN a, GEN b, GEN p) as gauss, where b is a FpM.

GEN FpM_FpC_gauss(GEN a, GEN b, GEN p) as gauss, where b is a FpC.

GEN FpM_image(GEN x, GEN p) as image

GEN FpM_intersect(GEN x, GEN y, GEN p) as intersect
**F_{pM} inv(GEN x, GEN p)** returns a left inverse of \( x \) (the inverse if \( x \) is square), or NULL if \( x \) is not invertible.

**F_{pM}_F_{pC} invimage(GEN m, GEN v, GEN p)** given an \( F_{pM} \) \( x \) and an \( F_{pC} \) \( y \), returns an \( x \) such that \( Ax = y \), or NULL if no such vector exist.

**F_{pM} invimage(GEN m, GEN v, GEN p)** given two \( F_{pM} \) \( x \) and \( y \), returns \( x \) such that \( Ax = y \), or NULL if no such matrix exist.

**F_{pM} ker(GEN x, GEN p)** as ker

**long F_{pM} rank(GEN x, GEN p)** as rank

**F_{pM} indexrank(GEN x, GEN p)** as indexrank

**F_{pM} suppl(GEN x, GEN p)** as suppl

**F_{pM} hess(GEN x, GEN p)** upper Hessenberg form of \( x \) over \( F_p \).

**F_{pM} charpoly(GEN x, GEN p)** characteristic polynomial of \( x \).

### 7.2.1.4 F_{qc}, F_{qm} and F_{q}-linear algebra.

An \( F_{qm} \) (resp. \( F_{qc} \)) is a matrix (resp a \( t\_COL \)) with \( F_q \) coefficients (with respect to given \( T \), \( p \)), not necessarily reduced (i.e arbitrary \( t\_INTs \) and \( ZXs \) in the same variable as \( T \)).

**F_{qc} add(GEN a, GEN b, GEN T, GEN p)**

**F_{qc} sub(GEN a, GEN b, GEN T, GEN p)**

**F_{qc} F_{q} mul(GEN a, GEN b, GEN T, GEN p)**

**F_{qm} deplin(GEN x, GEN T, GEN p)** returns a non-trivial kernel vector, or NULL if none exist.

**F_{qm} gauss(GEN a, GEN b, GEN T, GEN p)** as gauss, where \( b \) is a \( F_{qm} \).

**F_{qm} F_{qc} gauss(GEN a, GEN b, GEN T, GEN p)** as gauss, where \( b \) is a \( F_{qc} \).

**F_{qm} F_{qc} mul(GEN a, GEN b, GEN T, GEN p)**

**F_{qm} ker(GEN x, GEN T, GEN p)** as ker

**F_{qm} image(GEN x, GEN T, GEN p)** as image

**F_{qm} inv(GEN x, GEN T, GEN p)** returns the inverse of \( x \), or NULL if \( x \) is not invertible.

**F_{qm} mul(GEN a, GEN b, GEN T, GEN p)**

**long F_{qm} rank(GEN x, GEN T, GEN p)** as rank

**F_{qm} suppl(GEN x, GEN T, GEN p)** as suppl

**F_{qm} det(GEN x, GEN T, GEN p)** as det
7.2.2 Flc / Flv, Flm. See FpV, FpM operations.

GEN Flv_copy(GEN x) returns a copy of x.

GEN Flv_center(GEN z, ulong p, ulong ps2)

GEN Flm_copy(GEN x) returns a copy of x.

GEN matid_Flm(long n) returns an Flm which is an $n \times n$ identity matrix.

GEN scalar_Flm(long s, long n) returns an Flm which is $s$ times the $n \times n$ identity matrix.

GEN Flm_center(GEN z, ulong p, ulong ps2)

GEN Flm_Fl_add(GEN x, ulong y, ulong p) returns $x + y \cdot \text{Id}$ ($x$ must be square).

GEN Flm_Flc_mul(GEN x, GEN y, ulong p) multiplies $x$ and $y$ (assumed to have compatible dimensions).

GEN Flm_Fl_mul(GEN x, ulong y, ulong p) multiplies the Flm $x$ by $y$.

GEN Flm_neg(GEN x, ulong p) negates the Flm $x$.

void Flm_Fl_mul_inplace(GEN x, ulong y, ulong p) replaces the Flm $x$ by $x \cdot y$.

GEN Flc_Fl_mul(GEN x, ulong y, ulong p) multiplies the Flv $x$ by $y$.

void Flc_Fl_mul_inplace(GEN x, ulong y, ulong p) replaces the Flc $x$ by $x \cdot y$.

void Flc_Fl_mul_part_inplace(GEN x, ulong y, ulong p, long l) multiplies $x[1..l]$ by $y$ modulo $p$. In place.

GEN Flc_Fl_div(GEN x, ulong y, ulong p) divides the Flv $x$ by $y$.

void Flc_Fl_div_inplace(GEN x, ulong y, ulong p) replaces the Flv $x$ by $x/y$.

void Flc_lincomb1_inplace(GEN X, GEN Y, ulong v, ulong q) sets $X \leftarrow X + vY$, where $X, Y$ are Flc. Memory efficient (e.g. no-op if $v = 0$), and gerepile-safe.

GEN Flv_add(GEN x, GEN y, ulong p) adds two Flv.

void Flv_add_inplace(GEN x, GEN y, ulong p) replaces $x$ by $x + y$.

GEN Flv_sub(GEN x, GEN y, ulong p) subtracts $y$ to $x$.

void Flv_sub_inplace(GEN x, GEN y, ulong p) replaces $x$ by $x - y$.

ulong Flv_dotproduct(GEN x, GEN y, ulong p) returns the scalar product of $x$ and $y$

ulong Flv_sum(GEN x, ulong p) returns the sums of the components of $x$.

GEN zero_Flm(long m, long n) creates a Flm with $m \times n$ components set to 0. Note that the result allocates a single column, so modifying an entry in one column modifies it in all columns.

GEN zero_Flm_copy(long m, long n) creates a Flm with $m \times n$ components set to 0.

GEN zero_Flv(long n) creates a Flv with $n$ components set to 0.

GEN row_Flm(GEN A, long x0) return $A[i,]$ the $i$-th row of the Flm (or zm) $A$.

GEN Flm_mul(GEN x, GEN y, ulong p) multiplies $x$ and $y$ (assumed to have compatible dimensions).

GEN Flm_powu(GEN x, ulong n, ulong p) computes $x^n$ where $x$ is a square Flm.

96
GEN Flm_charpoly(GEN x, ulong p) return the characteristic polynomial of the square Flm x, as a Flx.

GEN Flm_deplin(GEN x, ulong p)
ulong Flm_det(GEN x, ulong p)
ulong Flm_det_sp(GEN x, ulong p), as Flm_det, in place (destroys x).
GEN Flm_gauss(GEN a, GEN b, ulong p) as gauss, where b is a Flm.
GEN Flm_Flc_gauss(GEN a, GEN b, ulong p) as gauss, where b is a Flc.
GEN Flm_indexrank(GEN x, ulong p)
GEN Flm_inv(GEN x, ulong p)
GEN Flm_Flc_invimage(GEN A, GEN y, ulong p) given an Flm x and an Flc y, returns an x such that Ax = y, or NULL if no such vector exist.
GEN Flm_invimage(GEN x, GEN y, ulong p) given two Flm x and y, returns x such that Ax = y, or NULL if no such matrix exist.
GEN Flm_ker(GEN x, ulong p)
GEN Flm_ker_sp(GEN x, ulong p, long deplin), as Flm_ker (if deplin=0) or Flm_deplin (if deplin=1), in place (destroys x).
long Flm_rank(GEN x, ulong p)
long Flm_suppl(GEN x, ulong p)
GEN Flm_image(GEN x, ulong p)
GEN Flm_transpose(GEN x)
GEN Flm_hess(GEN x, ulong p) upper Hessenberg form of x over \( F_p \).

7.2.3 F2c / F2v, F2m. An F2v v is a t_VECSMALL representing a vector over \( F_2 \). Specifically \( z[0] \) is the usual codeword, \( z[1] \) is the number of components of \( v \) and the coefficients are given by the bits of remaining words by increasing indices.

ulong F2v_coeff(GEN x, long i) returns the coefficient \( i \geq 1 \) of \( x \).
void F2v_clear(GEN x, long i) sets the coefficient \( i \geq 1 \) of \( x \) to 0.
void F2v_flip(GEN x, long i) adds 1 to the coefficient \( i \geq 1 \) of \( x \).
void F2v_set(GEN x, long i) sets the coefficient \( i \geq 1 \) of \( x \) to 1.
void F2v_copy(GEN x) returns a copy of \( x \).
GEN F2v_slice(GEN x, long a, long b) returns the F2v with entries \( x[a], \ldots, x[b] \). Assumes \( a \leq b \).
ulong F2m_coeff(GEN x, long i, long j) returns the coefficient \( (i,j) \) of \( x \).
void F2m_clear(GEN x, long i, long j) sets the coefficient \( (i,j) \) of \( x \) to 0.
void F2m_flip(GEN x, long i, long j) adds 1 to the coefficient \( (i,j) \) of \( x \).
void F2m_set(GEN x, long i, long j) sets the coefficient \( (i,j) \) of \( x \) to 1.
void F2m_copy(GEN x) returns a copy of x.

GEN F2m_rowslice(GEN x, long a, long b) returns the F2m built from the a-th to b-th rows of the F2m x. Assumes a ≤ b.

GEN F2m_F2c_mul(GEN x, GEN y) multiplies x and y (assumed to have compatible dimensions).

GEN F2m_image(GEN x) gives a subset of the columns of x that generate the image of x.

GEN F2m_invimage(GEN A, GEN B)

GEN F2m_F2c_invimage(GEN A, GEN y)

GEN F2m_gauss(GEN a, GEN b) as gauss, where b is a F2m.

GEN F2m_F2c_gauss(GEN a, GEN b) as gauss, where b is a F2c.

GEN F2m_indexrank(GEN x) x being a matrix of rank r, returns a vector with two t_VECSMALL components y and z of length r giving a list of rows and columns respectively (starting from 1) such that the extracted matrix obtained from these two vectors using vecextract(x, y, z) is invertible.

GEN F2m_mul(GEN x, GEN y) multiplies x and y (assumed to have compatible dimensions).

GEN F2m_powu(GEN x, ulong n) computes x^n where x is a square F2m.

long F2m_rank(GEN x) as rank.

long F2m_suppl(GEN x) as suppl.

GEN matid_F2m(long n) returns an F2m which is an n × n identity matrix.

GEN zero_F2v(long n) creates a F2v with n components set to 0.

GEN F2v_ei(long n, long i) creates a F2v with n components set to 0, but for the i-th one, which is set to 1 (i-th vector in the canonical basis).

GEN zero_F2m(long m, long n) creates a F2m with m x n components set to 0. Note that the result allocates a single column, so modifying an entry in one column modifies it in all columns.

GEN F2m_to_Flm(GEN x)

GEN F2m_to_ZM(GEN x)

GEN RgM_to_F2m(GEN x)

void F2v_add_inplace(GEN x, GEN y) replaces x by x + y. It is allowed for y to be shorter than x.
ulong F2m_det(GEN x)
ulong F2m_det_sp(GEN x), as $F_{2m}$ det, in place (destroys x).
GEN F2m_deplin(GEN x)
ulong F2v_dotproduct(GEN x, GEN y) returns the scalar product of x and y
GEN F2m_inv(GEN x)
GEN F2m_ker(GEN x)
GEN F2m_ker_sp(GEN x, long deplin), as $F_{2m}$ ker (if deplin=0) or F2m_deplin (if deplin=1), in place (destroys x).

7.2.4 FlxqV, FlxqM. See FqV, FqM operations.
GEN FlxqV_dotproduct(GEN x, GEN y, GEN T, ulong p) as FpV_dotproduct.
GEN FlxM_Flx_add_shallow(GEN x, GEN y, GEN T, ulong p) as RgM_Rg_add_shallow.
GEN FlxM_gauss(GEN a, GEN b, GEN T, ulong p)
GEN FlxM_FlxqC_gauss(GEN a, GEN b, GEN T, ulong p)
GEN FlxqM_FlxqC_mul(GEN a, GEN b, GEN T, ulong p)
GEN FlxqM_ker(GEN x, GEN T, ulong p)
GEN FlxqM_image(GEN x, GEN T, ulong p)
GEN FlxqM_det(GEN a, GEN T, ulong p)
GEN FlxqM_inv(GEN x, GEN T, ulong p)
GEN FlxqM_mul(GEN a, GEN b, GEN T, ulong p)
long FlxqM_rank(GEN x, GEN T, ulong p)
GEN matid_FlxqM(long n, GEN T, ulong p)

7.2.5 Zlm. GEN Zlm_gauss(GEN a, GEN b, ulong p, long e, GEN C) as gauss with the following peculiarities: $a$ and $b$ are $ZM$, such that $a$ is invertible modulo $p$. Optional $C$ is an Flm that is an inverse of $a \bmod p$ or NULL. Return the matrix $x$ such that $ax = b \bmod p^e$ and all elements of $x$ are in $[0, p^e - 1]$. For efficiency, it is better to reduce $a$ and $b \bmod p^e$ first.

7.2.6 FpX. Let $p$ an understood t_INT, to be given in the function arguments; in practice $p$ is not assumed to be prime, but be wary. Recall than an Fp object is a t_INT, preferably belonging to $[0, p - 1]$; an FpX is a t_POL in a fixed variable whose coefficients are Fp objects. Unless mentioned otherwise, all outputs in this section are FpXs. All operations are understood to take place in $(\mathbb{Z}/p\mathbb{Z})[X]$. 
7.2.6.1 Conversions. In what follows p is always a t_INT, not necessarily prime.

int RgX_is_FpX(GEN z, GEN *p), z a t_POL, checks if it can be mapped to a FpX, by checking Rg_is_Fp coefficientwise.

GEN RgX_to_FpX(GEN z, GEN p), z a t_POL, returns the FpX obtained by applying Rg_to_Fp coefficientwise.

GEN FpX_red(GEN z, GEN p), z a ZX, returns lift(z * Mod(1,p)), normalized.

GEN FpXV_red(GEN z, GEN p), z a t_VEC of ZX. Applies FpX_red componentwise and returns the result (and we obtain a vector of FpXs).

GEN FpXT_red(GEN z, GEN p), z a tree of ZX. Applies FpX_red to each leaf and returns the result (and we obtain a tree of FpXs).

7.2.6.2 Basic operations. In what follows p is always a t_INT, not necessarily prime.

Now, except for p, the operands and outputs are all FpX objects. Results are undefined on other inputs.

GEN FpX_add(GEN x,GEN y, GEN p) adds x and y.

GEN FpX_neg(GEN x,GEN p) returns −x, the components are between 0 and p if this is the case for the components of x.

GEN FpX_renormalize(GEN x, long l), as normalizepol, where l = lg(x), in place.

GEN FpX_sub(GEN x,GEN y,GEN p) returns x − y.

GEN FpX_mul(GEN x,GEN y,GEN p) returns xy.

GEN FpX_mulspec(GEN a, GEN b, GEN p, long na, long nb) see ZX_mulspec

GEN FpX_sqr(GEN x,GEN p) returns x^2.

GEN FpX_divrem(GEN x, GEN y, GEN p, GEN *pr) returns the quotient of x by y, and sets pr to the remainder.

GEN FpX_div(GEN x, GEN y, GEN p) returns the quotient of x by y.

GEN FpX_div_by_X_x(GEN A, GEN a, GEN p, GEN *r) returns the quotient of the FpX A by (X − a), and sets r to the remainder A(a).

GEN FpX_rem(GEN x, GEN y, GEN p) returns the remainder x mod y.

long FpX_valrem(GEN x, GEN t, GEN p, GEN *r) The arguments x and e being non-zero FpX returns the highest exponent e such that t^e divides x. The quotient x/t^e is returned in *r. In particular, if t is irreducible, this returns the valuation at t of x, and *r is the prime-to-t part of x.

GEN FpX_deriv(GEN x, GEN p) returns the derivative of x. This function is not memory-clean, but nevertheless suitable for gerepileupto.

GEN FpX_translate(GEN P, GEN c, GEN p) let c be an Fp and let P be an FpX; returns the translated FpX of P(X + c).

GEN FpX_gcd(GEN x, GEN y, GEN p) returns a (not necessarily monic) greatest common divisor of x and y.

GEN FpX_halfgcd(GEN x, GEN y, GEN p) returns a two-by-two FpX M with determinant ±1 such that the image (a, b) of (x, y) by M has the property that deg a ≥ \frac{\deg x}{2} > deg b.
GEN FpX_extgcd(GEN x, GEN y, GEN p, GEN *u, GEN *v) returns $d = \text{GCD}(x, y)$ (not necessarily monic), and sets $*u, *v$ to the Bezout coefficients such that $*ux + *vy = d$. If $*u$ is set to NULL, it is not computed which is a bit faster. This is useful when computing the inverse of $y$ modulo $x$.

GEN FpX_center(GEN z, GEN p, GEN pov2) returns the polynomial whose coefficient belong to the symmetric residue system. Assumes the coefficients already belong to $[0, p - 1]$ and pov2 is shifti($p, -1$).

### 7.2.6.3 Mixed operations

The following functions implement arithmetic operations between FpX and Fp operands, the result being of type FpX. The integer $p$ need not be prime.

GEN Z_to_FpX(GEN x, GEN p, long v) converts a t_INT to a scalar polynomial in variable $v$, reduced modulo $p$.

GEN FpX_Fp_add(GEN y, GEN x, GEN p) adds $FpX y$ to the $FpX x$.

GEN FpX_Fp_add_shallow(GEN y, GEN x, GEN p) adds $FpX x$ to the $FpX y$, using a shallow copy (result not suitable for gerepileupto).

GEN FpX_Fp_sub(GEN y, GEN x, GEN p) subtracts $FpX x$ from the $FpX y$.

GEN FpX_Fp_sub_shallow(GEN y, GEN x, GEN p) subtracts $FpX x$ from the $FpX y$, using a shallow copy (result not suitable for gerepileupto).

GEN Fp_FpX_sub(GEN x, GEN y, GEN p) returns $x - y$, where $x$ is a t_INT and $y$ an FpX.

GEN FpX_Fp_mul(GEN x, GEN y, GEN p) multiplies the $FpX x$ by the $FpX y$.

GEN FpX_Fp_mulspec(GEN x, GEN y, GEN p, long lx) see ZX_mulspec.

GEN FpX_mulu(GEN x, ulong y, GEN p) multiplies the $FpX x$ by $y$.

GEN FpX_Fp_mul_to_monic(GEN y, GEN x, GEN p) returns $yx$ assuming the result is monic of the same degree as $y$ (in particular $x \neq 0$).

### 7.2.6.4 Miscellaneous operations

GEN FpX_normalize(GEN z, GEN p) divides the FpX $z$ by its leading coefficient. If the latter is 1, $z$ itself is returned, not a copy. If not, the inverse remains uncollected on the stack.

GEN FpX_invBarrett(GEN T, GEN p), returns the Barrett inverse $M$ of $T$ defined by $M(x) = x^n \times T(1/x) \equiv 1 \pmod{x^n - 1}$ where $n$ is the degree of $T$.

GEN FpX_rescale(GEN P, GEN h, GEN p) returns $h^{\deg(P)}P(x/h)$. $P$ is an FpX and $h$ is a non-zero Fp (the routine would work with any non-zero t_INT but is not efficient in this case).

GEN FpX_eval(GEN x, GEN y, GEN p) evaluates the FpX $x$ at the Fp $y$. The result is an Fp.

GEN FpXV_FpC_mul(GEN V, GEN W, GEN p) multiplies a non-empty line vector of FpX by a column vector of Fp of compatible dimensions. The result is an FpX.

GEN FpXV_FpC_mulspec(GEN V, GEN W, GEN p) multiplies the FpX $V$ by the Fp $W$.

GEN FpXV_prod(GEN V, GEN p), $V$ being a vector of FpX, returns their product.

GEN FpV_roots_to_pol(GEN V, GEN p, long v), $V$ being a vector of INTs, returns the monic FpX $\prod_i\{\text{pol}_x[v] - V[i]\}$.

GEN FpX_chinese_coprime(GEN x, GEN y, GEN Tx, GEN Ty, GEN Tz, GEN p): returns an FpX, congruent to $x \text{ mod } Tx$ and to $y \text{ mod } Ty$. Assumes $Tx$ and $Ty$ are coprime, and $Tz = Tx \ast Ty$ or NULL (in which case it is computed within).
GEN FpV_polint(GEN x, GEN y, GEN p) returns the FpX interpolation polynomial with value $y[i]$ at $x[i]$. Assumes lengths are the same, components are t_INTs, and the $x[i]$ are distinct modulo $p$.

int FpX_is_squarefree(GEN f, GEN p) returns 1 if the FpX $f$ is squarefree, 0 otherwise.

int FpX_is_irred(GEN f, GEN p) returns 1 if the FpX $f$ is irreducible, 0 otherwise. Assumes that $p$ is prime. If $f$ has few factors, FpX_nbfact($f$, $p$) == 1 is much faster.

int FpX_is_totally_split(GEN f, GEN p) returns 1 if the FpX $f$ splits into a product of distinct linear factors, 0 otherwise. Assumes that $p$ is prime.

GEN FpX_factor(GEN f, GEN p), factors the FpX $f$. Assumes that $p$ is prime. The returned value $v$ is a t_VEC with two components: $v[1]$ is a vector of distinct irreducible (FpX) factors, and $v[2]$ is a t_VECSMALL of corresponding exponents. The order of the factors is deterministic (the computation is not).

long FpX_nbfact(GEN f, GEN p), assuming the FpX $f$ is squarefree, returns the number of its irreducible factors. Assumes that $p$ is prime.

long FpX_degfact(GEN f, GEN p), as FpX_factor, but the degrees of the irreducible factors are returned instead of the factors themselves (as a t_VECSMALL). Assumes that $p$ is prime.

long FpX_nbroots(GEN f, GEN p) returns the number of distinct roots in $\mathbb{Z}/p\mathbb{Z}$ of the FpX $f$. Assumes that $p$ is prime.

GEN FpX_oneroot(GEN f, GEN p) returns one root in $\mathbb{Z}/p\mathbb{Z}$ of the FpX $f$. Return NULL if no root exists. Assumes that $p$ is prime.

GEN FpX_roots(GEN f, GEN p) returns the roots in $\mathbb{Z}/p\mathbb{Z}$ of the FpX $f$ (without multiplicity, as a vector of Fps). Assumes that $p$ is prime.

GEN random_FpX(long d, long v, GEN p) returns a random FpX in variable $v$, of degree less than $d$.

GEN FpX_resultant(GEN x, GEN y, GEN p) returns the resultant of $x$ and $y$, both FpX. The result is a t_INT belonging to $[0,p-1]$.

GEN FpX_disc(GEN x, GEN p) returns the discriminant of the FpX $x$. The result is a t_INT belonging to $[0,p-1]$.

GEN FpX_FpXY_resultant(GEN a, GEN b, GEN p), a a t_POL of t_INTs (say in variable $X$), b a t_POL (say in variable $X$) whose coefficients are either t_POLs in $\mathbb{Z}[Y]$ or t_INTs. Returns $\text{Res}_X(a,b)$ in $\mathbb{F}_p[Y]$. The function assumes that $X$ has lower priority than $Y$.

7.2.7 FpXQ, Fq. Let $p$ a t_INT and $T$ an FpX for $p$, both to be given in the function arguments; an FpXQ object is an FpX whose degree is strictly less than the degree of $T$. An Fq is either an FpXQ or an Fp. Both represent a class in $(\mathbb{Z}/p\mathbb{Z})[X]/(T)$, in which all operations below take place. In addition, Fq routines also allow $T = \text{NULL}$, in which case no reduction mod $T$ is performed on the result.

For efficiency, the routines in this section may leave small unused objects behind on the stack (their output is still suitable for gerepileupto). Besides $T$ and $p$, arguments are either FpXQ or Fq depending on the function name. (All Fq routines accept FpXQs by definition, not the other way round.)
7.2.7.1 Preconditioned reduction.

For faster reduction, the modulus \( T \) can be replaced by an extended modulus, which is an \( \mathbb{F}_p \mathbb{X} T \), in all \( \mathbb{F}_p \mathbb{X} Q \)- and \( \mathbb{F}_q \)-classes functions, and in \( \mathbb{F}_p \mathbb{X} \text{rem} \) and \( \mathbb{F}_p \mathbb{X} \text{divrem} \).

\[
\text{GEN } \mathbb{F}_p \mathbb{X} \text{get_red} (\text{GEN} \ T, \text{GEN} \ p) \text{ returns the extended modulus } eT.
\]

To write code that works both with plain and extended moduli, the following accessors are defined:

\[
\begin{align*}
\text{GEN } & \text{ get_FpX_mod (GEN } eT) \text{ returns the underlying modulus } T. \\
\text{GEN } & \text{ get_FpX_var (GEN } eT) \text{ returns the variable number of the modulus.} \\
\text{GEN } & \text{ get_FpX_degree (GEN } eT) \text{ returns the degree of the modulus.}
\end{align*}
\]

Furthermore, \( \mathbb{Z}X_T \) to \( \mathbb{F}pX_T \) allows to convert an extended modulus for a \( \mathbb{F}_p \mathbb{X} \) to an extended modulus for the corresponding \( \mathbb{F}_lX \).

7.2.7.2 Conversions.

\[
\begin{align*}
\text{GEN } & \text{Rg_is_FpXQ (GEN } z, \text{GEN } *T, \text{GEN } *p) \text{ checks if } z \text{ is a GEN which can be mapped to } \mathbb{F}_p [X] / (T): \text{ anything for which Rg_is_Fp return 1, a t_POL for which RgX_to_FpX return 1, a t_POLMOD whose modulus is equal to } *T \text{ if } *T \text{ is not NULL (once mapped to a FpX). If an integer modulus is found it is put in } *p, \text{ else } *p \text{ is left unchanged. If a polynomial modulus is found it is put in } *T, \text{ else } *T \text{ is left unchanged.} \\
\text{int } & \text{RgX_is_FpXQ (GEN } z, \text{GEN } *T, \text{GEN } *p), \text{ z a t_POL, checks if it can be mapped to a FpXQX, by checking Rg_is_FpXQ coefficientwise.} \\
\text{GEN } & \text{Rg_to_FpXQ (GEN } z, \text{GEN } T, \text{GEN } p), \text{ z a GEN which can be mapped to } \mathbb{F}_p [X] / (T): \text{ anything Rg_to_Fp can be applied to, a t_POL to which RgX_to_FpX can be applied to, a t_POLMOD whose modulus is divisible by } T \text{ (once mapped to a FpX), a suitable t_RFRAC. Returns } z \text{ as an FpXQ, normalized.} \\
\text{GEN } & \text{RgX_to_FpXQ (GEN } z, \text{GEN } T, \text{GEN } p), \text{ z a t_POL, returns the FpXQ obtained by applying Rg_to_FpXQ coefficientwise.} \\
\text{GEN } & \text{RgX_to_FpX (GEN } z, \text{GEN } T, \text{GEN } p): \text{ let } z \text{ be a t_POL; returns the FpXQ obtained by applying Rg_to_FpXQ coefficientwise and simplifying scalars to t_INTs.} \\
\text{GEN } & \text{Fq_red (GEN } x, \text{GEN } T, \text{GEN } p), \text{ x a ZX or t_INT, reduce it to an Fq (T = NULL is allowed iff } x \text{ is a t_INT).} \\
\text{GEN } & \text{FqX_red (GEN } x, \text{GEN } T, \text{GEN } p), \text{ x a t_POL whose coefficients are ZXs or t_INTs, reduce them to Fqs. (If } T = \text{NULL, as FpXX_red(x, p)).} \\
\text{GEN } & \text{FqV_red (GEN } x, \text{GEN } T, \text{GEN } p), \text{ x a vector of ZXs or t_INTs, reduce them to Fqs. (If } T = \text{NULL, only reduce components mod } p \text{ to FpXs or Fps.)} \\
\text{GEN } & \text{FpXQ_red (GEN } x, \text{GEN } T, \text{GEN } p), \text{ x a t_POL whose coefficients are t_INTs, reduce them to FpXQs.}
\end{align*}
\]
7.2.8 FpXQ.

GEN FpXQ_add(GEN x, GEN y, GEN T, GEN p)
GEN FpXQ_sub(GEN x, GEN y, GEN T, GEN p)
GEN FpXQ_mul(GEN x, GEN y, GEN T, GEN p)
GEN FpXQ_sqr(GEN x, GEN T, GEN p)
GEN FpXQ_div(GEN x, GEN y, GEN T, GEN p)
GEN FpXQ_inv(GEN x, GEN T, GEN p) computes the inverse of x
GEN FpXQ_invsafe(GEN x, GEN T, GEN p), as FpXQ_inv, returning NULL if x is not invertible.
GEN FpXQX_renormalize(GEN x, long lx)
GEN FpXQ_pow(GEN x, GEN n, GEN T, GEN p) computes $x^n$.
GEN FpXQ_powu(GEN x, ulong n, GEN T, GEN p) computes $x^n$ for small n.
GEN FpXQ_log(GEN a, GEN g, GEN ord, GEN T, GEN p) Let g be of order ord in the finite field $F_p[X]/(T)$, return e such that $a^e = g$. If e does not exists, the result is currently undefined. Assumes that T is irreducible mod p.
GEN Fp_FpXQ_log(GEN a, GEN g, GEN ord, GEN T, GEN p) As FpXQ_log, a being a Fp.
GEN FpXQ_order(GEN a, GEN ord, GEN T, GEN p) returns the order of the FpXQ a. If o is non-NULL, it is assumed that o is a multiple of the order of a, either as a t_INT or a factorization matrix. Assumes that T is irreducible mod p.
int FpXQ_issquare(GEN x, GEN T, GEN p) returns 1 if x is a square and 0 otherwise. Assumes that T is irreducible mod p.
GEN FpXQ_sqrt(GEN x, GEN T, GEN p) returns a square root of x. Return NULL if x is not a square.
GEN FpXQ_sqrtn(GEN x, GEN n, GEN T, GEN p, GEN *zn) returns an n-th root of x. Return NULL if x is not an n-th power residue. Otherwise, if zn is non-NULL set it to a primitive n-th root of the unity. Assumes that T is irreducible mod p.

7.2.9 Fq.

GEN Fq_add(GEN x, GEN y, GEN T/*unused*/, GEN p)
GEN Fq_sub(GEN x, GEN y, GEN T/*unused*/, GEN p)
GEN Fq_mul(GEN x, GEN y, GEN T, GEN p)
GEN Fq_Fp_mul(GEN x, GEN y, GEN T, GEN p) multiplies the Fq x by the t_INT y.
GEN Fq_mulu(GEN x, ulong y, GEN T, GEN p) multiplies the Fq x by the scalar y.
GEN Fq_sqr(GEN x, GEN T, GEN p)
GEN Fq_neg(GEN x, GEN T, GEN p)
GEN Fq_neg_inv(GEN x, GEN T, GEN p) computes $-x^{-1}$
GEN Fq_inv(GEN x, GEN pol, GEN p) computes $x^{-1}$, raising an error if x is not invertible.
GEN Fq_invsafe(GEN x, GEN pol, GEN p) as Fq_inv, but returns NULL if x is not invertible.

GEN Fq_div(GEN x, GEN y, GEN T, GEN p)

GEN FqV_inv(GEN x, GEN T, GEN p)
x being a vector of Fqs, return the vector of inverses of the x[i]. The routine uses Montgomery’s trick, and involves a single inversion, plus 3(N - 1) multiplications for N entries. The routine is not stack-clean: 2N FpXQ are left on stack, besides the N in the result.

GEN Fq_pow(GEN x, GEN n, GEN pol, GEN p)
returns x^n.

GEN Fq_powu(GEN x, ulong n, GEN pol, GEN p)
returns x^n for small n.

int Fq_issquare(GEN x, GEN T, GEN p)
returns 1 if x is a square and 0 otherwise. Assumes that T is irreducible mod p. T = NULL is forbidden unless x is an Fp.

GEN Fq_sqrt(GEN x, GEN T, GEN p)
returns a square root of x. Return NULL if x is not a square.

GEN Fq_sqrtn(GEN x, GEN n, GEN T, GEN p, GEN *zn)
returns an n-th root of x. Return NULL if x is not an n-th power residue. Otherwise, if zn is non-NULL set it to a primitive n-th root of the unity. Assumes that T is irreducible mod p.

GEN FpXQ_charpoly(GEN x, GEN T, GEN p)
returns the characteristic polynomial of x

GEN FpXQ_minpoly(GEN x, GEN T, GEN p)
returns the minimal polynomial of x

GEN FpXQ_norm(GEN x, GEN T, GEN p)
returns the norm of x

GEN FpXQ_trace(GEN x, GEN T, GEN p)
returns the trace of x

GEN FpXQ_conjvec(GEN x, GEN T, GEN p)
returns the vector of conjugates [x, x^p, x^{p^2}, ..., x^{p^{n-1}}] where n is the degree of T.

GEN gener_FpXQ(GEN T, GEN p, GEN *po)
returns a primitive root modulo (T,p). T is an FpX assumed to be irreducible modulo the prime p. If po is not NULL it is set to [o, fa], where o is the order of the multiplicative group of the finite field, and fa is its factorization.

GEN gener_FpXQ_local(GEN T, GEN p, GEN L)
returns an element of G := F_p[X]/(T) which is a generator of the ℓ-Sylow of G for every ℓ in L. It is not necessary, and in fact slightly inefficient, to include ℓ = 2, since 2 is treated separately in any case, i.e. the generator obtained is never a square if p is odd.

GEN FpXQ_powers(GEN x, long n, GEN T, GEN p)
returns [x^0, ..., x^n] as a t_VEC of FpXQs.

GEN FpXQ_matrix_pow(GEN x, long m, long n, GEN T, GEN p)
returns the powers as a an m × n matrix. Usually, we have m = n = deg T.

GEN FpXQ_autpow(GEN a, ulong n, GEN T, GEN p)
computes σ^n(X) assuming a = σ(X) where σ is an automorphism of the algebra F_p[X]/T(X).

GEN FpXQ_autsum(GEN a, ulong n, GEN T, GEN p)
returns the vector [σ^n(X), bσ(b)...σ^{n-1}(b)] where b = a[2].

GEN FpXQ_autpowers(GEN S, long n, GEN T, GEN p)
returns [x, S(x), S(S(x)), ..., S^{(n)}(x)] as a t_VEC of FpXQs.

GEN FpX_FpXQ_eval(GEN f,GEN x,GEN T,GEN p)
returns f(x).

GEN FpX_FpXQV_eval(GEN f,GEN V,GEN T,GEN p)
returns f(x), assuming that V was computed by FpXQ_powers(x,n,T,p).
7.2.10 FpXX, FpXY. Contrary to what the name implies, an FpXX is a t_POL whose coefficients are either t_INTs or FpXs. This reduces memory overhead at the expense of consistency. The prefix FpXY is an alias for FpXX when variables matters.

GEN FpXX_red(GEN z, GEN p), z a t_POL whose coefficients are either t_INTs or FpXs. Returns the t_POL equal to z with all components reduced modulo p.

GEN FpXX_renormalize(GEN x, long l), as normalizepol, where l = lg(x), in place.

GEN FpXX_add(GEN x, GEN y, GEN p) adds x and y.

GEN FpXX_sub(GEN x, GEN y, GEN p) returns x − y.

GEN FpXX_neg(GEN x, GEN p) returns −x.

GEN FpXX_Fp_mul(GEN x, GEN y, GEN p) multiplies the FpXX x by the Fp y.

GEN FpXX_FpX_mul(GEN x, GEN y, GEN p) multiplies the coefficients of the FpXX x by the FpX y.

GEN FpXX_mulu(GEN x, ulong y, GEN p) multiplies the FpXX x by the scalar y.

GEN FpXX_Fq_mul(GEN x, GEN y, GEN T, GEN p) multiplies the FpXX x by the t_INT y.

7.2.11 FpXQX, FqX. Contrary to what the name implies, an FpXQX is a t_POL whose coefficients are either t_INTs or FpXs. This reduces memory overhead at the expense of consistency. The prefix FpXY is an alias for FpXX when variables matters.

GEN FpXX_red(GEN z, GEN p), z a t_POL whose coefficients are either ZXs or t_INTs. Returns the t_POL equal to z with all components reduced modulo p.

GEN FpXX_renormalize(GEN x, long l), as normalizepol, where l = lg(x), in place.

GEN FpXX_add(GEN x, GEN y, GEN p) adds x and y.

GEN FpXX_sub(GEN x, GEN y, GEN p) returns x − y.

GEN FpXX_neg(GEN x, GEN p) returns −x.

GEN FpXX_Fp_mul(GEN x, GEN y, GEN p) multiplies the FpXX x by the Fp y.

GEN FpXX_FpX_mul(GEN x, GEN y, GEN p) multiplies the coefficients of the FpXX x by the FpX y.

GEN FpXX_mulu(GEN x, ulong y, GEN p) multiplies the FpXX x by the scalar y.

GEN FpXX_Fq_mul(GEN x, GEN y, GEN T, GEN p) multiplies the FpXX x by the t_INT y.

7.2.11.1 Basic operations.

GEN FqX_add(GEN x, GEN y, GEN T, GEN p)

GEN FqX_Fq_add(GEN x, GEN y, GEN T, GEN p) adds the Fq y to the FqX x.

GEN FqX_neg(GEN x, GEN T, GEN p)

GEN FqX_sub(GEN x, GEN y, GEN T, GEN p)

GEN FqX_mulu(GEN x, GEN y, GEN T, GEN p) multiplies the FqX x by the scalar y.

GEN FqX_Fp_mul(GEN x, GEN y, GEN T, GEN p) multiplies the FqX x by the t_INT y.
GEN FqX_Fq_mul_to_monic(GEN x, GEN y, GEN T, GEN p) returns $xy$ assuming the result is monic of the same degree as $x$ (in particular $y \neq 0$).

GEN FqX_normalize(GEN z, GEN T, GEN p) divides the FqX $z$ by its leading term. The leading coefficient becomes 1 as a t_INT.

GEN FqX_sqr(GEN x, GEN T, GEN p)

GEN FqX_divrem(GEN x, GEN y, GEN T, GEN p, GEN *z)

GEN FqX_rem(GEN x, GEN y, GEN T, GEN p)

GEN FqX_deriv(GEN x, GEN T, GEN p) returns the derivative of $x$. (This function is suitable for gerepilupto but not memory-clean.)

GEN FqX_translate(GEN P, GEN c, GEN T, GEN p) let $c$ be an Fq defined modulo $(p,T)$, and let $P$ be an FqX; returns the translated FqX of $P(X+c)$.

GEN FqX_gcd(GEN P, GEN Q, GEN T, GEN p) returns a (not necessarily monic) greatest common divisor of $x$ and $y$.

GEN FqX_extgcd(GEN x, GEN y, GEN T, GEN p, GEN *ptu, GEN *ptv) returns $d = \text{GCD}(x,y)$ (not necessarily monic), and sets $*u, *v$ to the Bezout coefficients such that $*ux + *vy = d$.

GEN FqX_eval(GEN x, GEN y, GEN T, GEN p) evaluates the FqX $x$ at the Fq $y$. The result is an Fq.

GEN FqXY_eval(GEN Q, GEN y, GEN x, GEN T, GEN p) $Q$ an FqXY, i.e. a t_POL with Fq or FqX coefficients representing an element of $F_q[X][Y]$. Returns the Fq $Q(x,y)$.

GEN FqXY_evalx(GEN Q, GEN x, GEN T, GEN p) $Q$ being an FqXY, returns the FqX $Q(x,Y)$, where $Y$ is the main variable of $Q$.

GEN FpXQX_red(GEN z, GEN T, GEN p) z a t_POL whose coefficients are ZXs or t_INTs, reduce them to FpXQs.

GEN FpXQX_FpXQ_mul(GEN x, GEN y, GEN T, GEN p)

GEN FpXQX_sqr(GEN x, GEN T, GEN p)

GEN FpXQX_divrem(GEN x, GEN y, GEN T, GEN p, GEN *pr)

GEN FpXQX_div(GEN x, GEN y, GEN T, GEN p)

GEN FpXQX_rem(GEN x, GEN y, GEN T, GEN p)

GEN FpXQX_invBarrett(GEN y, GEN T, GEN p) returns the Barrett inverse of the FpXQ $y$, namely a lift of $1/\text{polrecip}(y) + O(x^{\text{deg}(y)-1})$. 

107
GEN FpXQX_rem_Barrett(GEN x, GEN iy, GEN y, GEN T, GEN p) returns \( x \mod y \), assuming \( iy \) is the Barrett inverse of \( y \).

GEN FpXQX_divrem_Barrett(GEN x, GEN iy, GEN y, GEN T, GEN p, GEN *pr) performs the Euclidean division of \( x \) by \( y \), assuming \( iy \) is the Barrett inverse of \( y \). Returns the quotient and set *pr to the remainder.

GEN FpXQV_prod(GEN V, GEN T, GEN p), \( V \) being a vector of \( FpXQ \), returns their product.

GEN FpXQ_gcd(GEN x, GEN y, GEN T, GEN p)
GEN FpXQ_extgcd(GEN x, GEN y, GEN T, GEN p, GEN *ptu, GEN *ptv)
GEN FpXQ_FpXQXQ_eval(GEN f, GEN x, GEN S, GEN T, GEN p) returns \( f(x) \).

GEN FpXQ_FpXQXQV_eval(GEN f, GEN V, GEN S, GEN T, GEN p) returns \( f(x) \), assuming that \( V \) was computed by \( FpXQXQ \) powers of \( x, n, S, T, p \).

GEN FpXQXQ_div(GEN x, GEN y, GEN S, GEN T, GEN p), \( x \), \( y \) and \( S \) being \( FpXQX \)s, returns \( x \times y^{-1} \) modulo \( S \).

GEN FpXQXQ_inv(GEN x, GEN S, GEN T, GEN p), \( x \) and \( S \) being \( FpXQX \)s, returns \( x^{-1} \) modulo \( S \).

GEN FpXQXQ_inv_safe(GEN x, GEN S, GEN T, GEN p) as FpXQXQ_inv, returning NULL if \( x \) is not invertible.

GEN FpXQXQ_mul(GEN x, GEN y, GEN S, GEN T, GEN p), \( x \), \( y \) and \( S \) being \( FpXQX \)s, returns \( xy \) modulo \( S \).

GEN FpXQXQ_sqr(GEN x, GEN S, GEN T, GEN p), \( x \) and \( S \) being \( FpXQX \)s, returns \( x^2 \) modulo \( S \).

GEN FpXQXQ_pow(GEN x, GEN n, GEN S, GEN T, GEN p), \( x \) and \( S \) being \( FpXQX \)s, returns \( x^n \) modulo \( S \).

GEN FpXQXQ_powers(GEN x, long n, GEN S, GEN T, GEN p), \( x \) and \( S \) being \( FpXQX \)s, returns \([x^0, \ldots, x^n]\) as a \( \text{t}_\text{VEC} \) of \( FpXQXs \).

GEN FpXQXQ_matrix_pow(GEN x, long m, long n, GEN S, GEN T, GEN p) returns the same powers of \( x \) as \( FpXQXQ \) powers of \( x, n-1, S, T, p \), but as an \( m \times n \) matrix.

GEN FpXQXQV_autpow(GEN a, long n, GEN S, GEN T, GEN p) \( \sigma \) being the automorphism defined by \( \sigma(X) = a[1] \mod T(X) \), \( \sigma(Y) = a[2] \mod S(X, Y) \), \( T(X) \), returns \([\sigma^n(X), \sigma^n(Y)]\).

GEN FpXQXQV_autsum(GEN a, long n, GEN S, GEN T, GEN p) \( \sigma \) being the automorphism defined by \( \sigma(X) = a[1] \mod T(X) \), \( \sigma(Y) = a[2] \mod S(X, Y) \), \( T(X) \), returns the vector \([\sigma^n(X), \sigma^n(Y), b\sigma(b) \ldots \sigma^{n-1}(b)]\) where \( b = a[3] \).

GEN FqXQ_add(GEN x, GEN y, GEN S, GEN T, GEN p), \( x \), \( y \) and \( S \) being \( FqXs \), returns \( x + y \) modulo \( S \).

GEN FqXQ_sub(GEN x, GEN y, GEN S, GEN T, GEN p), \( x \), \( y \) and \( S \) being \( FqXs \), returns \( x - y \) modulo \( S \).

GEN FqXQ_mul(GEN x, GEN y, GEN S, GEN T, GEN p), \( x \), \( y \) and \( S \) being \( FqXs \), returns \( xy \) modulo \( S \).

GEN FqXQ_div(GEN x, GEN y, GEN S, GEN T, GEN p), \( x \) and \( S \) being \( FqXs \), returns \( x/y \) modulo \( S \).

GEN FqXQ_inv(GEN x, GEN S, GEN T, GEN p), \( x \) and \( S \) being \( FqXs \), returns \( x^{-1} \) modulo \( S \).
GEN FqXQ_invsafe(GEN x, GEN S, GEN T, GEN p), as FqXQ_inv, returning NULL if x is not invertible.

GEN FqXQ_sqr(GEN x, GEN S, GEN T, GEN p), x and S being FqXs, returns $x^2$ modulo S.

GEN FqXQ_pow(GEN x, GEN n, GEN S, GEN T, GEN p), x and S being FqXs, returns $x^n$ modulo S.

GEN FqXQ_powers(GEN x, long n, GEN S, GEN T, GEN p), x and S being FqXs, returns $[x^0, \ldots, x^n]$ as a t_VEC of FqXQs.

GEN FqXQ_matrix_pow(GEN x, long m, long n, GEN S, GEN T, GEN p) returns the same powers of $x$ as FqXQ_powers($x$, $n - 1$, $S, T, p$), but as an $m \times n$ matrix.

GEN FqVQ_roots_to_pol(GEN V, GEN T, GEN p, long v), V being a vector of Fqs, returns the monic FqXQ $\prod_i (pol_x[v_i] - V[i])$.

7.2.11.2 Miscellaneous operations.

GEN init_Fq(GEN p, long n, long v) returns an irreducible polynomial of degree $n > 0$ over $\mathbb{F}_p[v]$ in variable v.

int FqXQ_is_squarefree(GEN P, GEN T, GEN p)

GEN FqXQ_roots(GEN x, GEN T, GEN p) return the roots of $x$ in $\mathbb{F}_p[X]/(T)$. Assumes $p$ is prime and $T$ irreducible in $\mathbb{F}_p[X]$.

GEN FqXQ_factor(GEN x, GEN T, GEN p) same output convention as FpXQ_factor. Assumes $p$ is prime and $T$ irreducible in $\mathbb{F}_p[X]$.

GEN FpXQ_factorff(GEN P, GEN T, GEN p). Assumes $p$ prime and $T$ irreducible in $\mathbb{F}_p[X]$. Factor the FpX P over the finite field $\mathbb{F}_p[Y]/(T(Y))$. See FpXQ_factorff_irred if P is known to be irreducible of $\mathbb{F}_p$.


GEN FpXQ_factorff_irred(GEN P, GEN T, GEN p). Assumes $p$ prime and $T$ irreducible in $\mathbb{F}_p[X]$. Factors the irreducible FpX P over the finite field $\mathbb{F}_p[Y]/(T(Y))$ and returns the vector of irreducible FqXs factors (the exponents, being all equal to 1, are not included).

GEN FpXQ_ffisom(GEN P, GEN Q, GEN p). Assumes $p$ prime, $P, Q$ are ZXs, both irreducible mod $p$, and deg($P$) | deg Q. Outputs a monomorphism between $\mathbb{F}_p[X]/(P)$ and $\mathbb{F}_p[X]/(Q)$, as a polynomial $R$ such that $Q | P(R)$ in $\mathbb{F}_p[X]$. If $P$ and $Q$ have the same degree, it is of course an isomorphism.

void FpXQ_ffintersect(GEN P, GEN Q, long n, GEN p, GEN *SP, GEN *SQ, GEN MA, GEN MB) Assumes $p$ is prime, $P, Q$ are ZXs, both irreducible mod $p$, and $n$ divides both the degree of $P$ and $Q$. Compute SP and SQ such that the subfield of $\mathbb{F}_p[X]/(P)$ generated by SP and the subfield of $\mathbb{F}_p[X]/(Q)$ generated by SQ are isomorphic of degree $n$. The polynomials $P$ and $Q$ do not need to be of the same variable. If MA (resp. MB) is not NULL, it must be the matrix of the Frobenius map in $\mathbb{F}_p[X]/(P)$ (resp. $\mathbb{F}_p[X]/(Q)$).

GEN FpXQQ_ffisom_inv(GEN S, GEN T, GEN p). Assumes $p$ is prime, $T$ a ZX, which is irreducible modulo $p$, $S$ a ZX representing an automorphism of $\mathbb{F}_q := \mathbb{F}_p[X]/(T)$, $S(X)$ is the image of $X$ by the automorphism.) Returns the inverse automorphism of $S$, in the same format, i.e. an FpX H such that $H(S) \equiv X$ modulo ($T, p$).

long FpXQQ_nbfact(GEN S, GEN T, GEN p) returns the number of irreducible factors of the polynomial $S$ over the finite field $\mathbb{F}_q$ defined by $T$ and $p$. 

109
long \texttt{FqX_{nbfact}}(\texttt{GEN S}, \texttt{GEN T}, \texttt{GEN p}) as above but accept \texttt{T=NULL}.

long \texttt{FpXQX_{nbroots}}(\texttt{GEN S}, \texttt{GEN T}, \texttt{GEN p}) returns the number of roots of the polynomial \texttt{S} over the finite field \( \mathbb{F}_q \) defined by \texttt{T} and \texttt{p}.

long \texttt{FqX_{nbroots}}(\texttt{GEN S}, \texttt{GEN T}, \texttt{GEN p}) as above but accept \texttt{T=NULL}.

\texttt{GEN \texttt{FpXQX_{Frobenius}}(GEN S, GEN T, GEN p)} returns \( X^q \mod S(X) \) over the finite field \( \mathbb{F}_q \) defined by \texttt{T} and \texttt{p}, thus \( q = p^n \) where \( n \) is the degree of \texttt{T}.

\texttt{GEN \texttt{FpXQX_{halfFrobenius}}(A, S, T, p)} returns \( A(X)^{(q-1)/2} \mod S(X) \) over the finite field \( \mathbb{F}_q \) defined by \texttt{T} and \texttt{p}, thus \( q = p^n \) where \( n \) is the degree of \texttt{T}.

\textbf{7.2.12 Flx.} Let \texttt{p} an understood \texttt{ulong}, assumed to be prime, to be given the the function arguments; an \texttt{Fl} is an \texttt{ulong} belonging to \([0, p - 1]\), an \texttt{Flx z} is a \texttt{t_VECSMALL} representing a polynomial with small integer coefficients. Specifically \( z[0] \) is the usual codeword, \( z[1] = \texttt{evalvarn}(v) \) for some variable \( v \), then the coefficients by increasing degree. An \texttt{Flx} is a \texttt{t_POL} whose coefficients are \texttt{Flx}.

In the following, an argument called \texttt{sv} is of the form \texttt{evalvarn}(\texttt{v}) for some variable number \texttt{v}.

\textbf{7.2.12.1 Preconditioned reduction.}

For faster reduction, the modulus \texttt{T} can be replaced by an extended modulus, which is an \texttt{FlxT}, in all \texttt{Flxq}-classes functions, and in \texttt{Flx_{divrem}}.

\texttt{GEN Flx_{get_red}(GEN T, ulong p)} returns the extended modulus \texttt{eT}.

To write code that works both with plain and extended moduli, the following accessors are defined:

\texttt{GEN get_Flx_mod(GEN eT)} returns the underlying modulus \texttt{T}.

\texttt{GEN get_Flx_var(GEN eT)} returns the variable number of the modulus.

\texttt{GEN get_Flx_degree(GEN eT)} returns the degree of the modulus.

Furthermore, \texttt{ZXT_to_FlxT} allows to convert an extended modulus for a \texttt{FpX} to an extended modulus for the corresponding \texttt{Flx}.

\textbf{7.2.12.2 Basic operations.}

\texttt{ulong Flx_{lead}(GEN x)} returns the leading coefficient of \texttt{x} as a \texttt{ulong} (return 0 for the zero polynomial).

\texttt{GEN Flx_{red}(GEN z, ulong p)} converts from \texttt{zx} with non-negative coefficients to \texttt{Flx} (by reducing them mod p).

\texttt{int Flx_{equal1}(GEN x)} returns 1 (true) if the \texttt{Flx} \texttt{x} is equal to 1, 0 (false) otherwise.

\texttt{int Flx_{equal}(GEN x, GEN y)} returns 1 (true) if the \texttt{Flx} \texttt{x} and \texttt{y} are equal, and 0 (false) otherwise.

\texttt{GEN Flx_{copy}(GEN x)} returns a copy of \texttt{x}.

\texttt{GEN Flx_{add}(GEN x, GEN y, ulong p)}

\texttt{GEN Flx_{Fl_{add}}(GEN y, ulong x, ulong p)}

\texttt{GEN Flx_{neg}(GEN x, ulong p)}

110
GEN Flx_neg_inplace(GEN x, ulong p), same as Flx_neg, in place (x is destroyed).

GEN Flx_sub(GEN x, GEN y, ulong p)
GEN Flx_mul(GEN x, GEN y, ulong p)
GEN Flx_Fl_mul(GEN y, ulong x, ulong p)
GEN Flx_double(GEN y, ulong p) returns 2y.
GEN Flx_triple(GEN y, ulong p) returns 3y.
GEN Flx_mulu(GEN y, ulong x, ulong p) as Flx_Fl_mul but do not assume that x < p.
GEN Flx_Fl_mul_to_monic(GEN y, ulong x, ulong p) returns yx assuming the result is monic of the same degree as y (in particular x ≠ 0).
GEN Flx_sqr(GEN x, ulong p)
GEN Flx_divrem(GEN x, GEN y, ulong p, GEN *pr)
GEN Flx_div(GEN x, GEN y, ulong p)
GEN Flx_rem(GEN x, GEN y, ulong p)
GEN Flx_deriv(GEN z, ulong p)
GEN Flx_gcd(GEN a, GEN b, ulong p) returns a (not necessarily monic) greatest common divisor of x and y.
GEN Flx_halfgcd(GEN x, GEN y, GEN p) returns a two-by-two FlxM M with determinant ±1 such that the image (a, b) of (x, y) by M has the property that \( \deg a \geq \frac{\deg x}{2} > \deg b \).
GEN Flx_extgcd(GEN a, GEN b, ulong p, GEN *ptu, GEN *ptv)
GEN Flx_pow(GEN x, long n, ulong p)
GEN Flx_roots(GEN f, ulong p) returns the vector of roots of f (without multiplicity, as a t_VECSMALL). Assumes that p is prime.
ulong Flx_oneroot(GEN f, ulong p) returns one root 0 ≤ r < p of the Flx f in \( \mathbb{Z}/p\mathbb{Z} \). Return p if no root exists. Assumes that p is prime.
GEN Flx_roots_naive(GEN f, ulong p) returns the vector of roots of f as a t_VECSMALL (multiple roots are not repeated), found by an exhaustive search. Efficient for very small p!
GEN Flx_factor(GEN f, ulong p)
GEN Flx_mod_Xn1(GEN T, ulong n, ulong p) return T modulo \( X^n + 1, p \). Shallow function.
GEN Flx_mod_Xnm1(GEN T, ulong n, ulong p) return T modulo \( X^n - 1, p \). Shallow function.
GEN Flx_degfact(GEN f, ulong p) as FpX_degfact.
GEN Flx_factorff_irred(GEN P, GEN Q, ulong p) as FpX_factorff_irred.
GEN Flx_ffisom(GEN P, GEN Q, ulong l) as FpX_ffisom.
7.2.12.3 Miscellaneous operations.

GEN pol0_Flx(long sv) returns a zero Flx in variable \( v \).

GEN zero_Flx(long sv) alias for pol0_Flx

GEN pol1_Flx(long sv) returns the unit Flx in variable \( v \).

GEN polx_Flx(long sv) returns the variable \( v \) as degree 1 Flx.

GEN Flx_normalize(GEN z, ulong p), as FpX_normalize.

GEN random_Flx(long d, long sv, ulong p) returns a random Flx in variable \( v \), of degree less than \( d \).

GEN Flx_recip(GEN x), returns the reciprocal polynomial

ulong Flx_resultant(GEN a, GEN b, ulong p), returns the resultant of \( a \) and \( b \)

ulong Flx_extresultant(GEN a, GEN b, ulong p, GEN *ptU, GEN *ptV) given two Flx \( a \) and \( b \), returns their resultant and sets Bezout coefficients (if the resultant is 0, the latter are not set).

GEN Flx_invBarrett(GEN T, ulong p), returns the Barrett inverse \( M \) of \( T \) defined by \( M(x) \times x^n T(1/x) \equiv 1 \left( \text{mod } x^{n-1} \right) \) where \( n \) is the degree of \( T \).

GEN Flx_renormalize(GEN x, long l), as FpX_renormalize, where \( l = lg(x) \), in place.

GEN Flx_shift(GEN T, long n) returns \( T \times x^n \) if \( n \geq 0 \), and \( T \div x^{-n} \) otherwise.

long Flx_val(GEN x) returns the valuation of \( x \), i.e. the multiplicity of the 0 root.

long Flx_valrem(GEN x, GEN *Z) as RgX_valrem, returns the valuation of \( x \). In particular, if the valuation is 0, set \( *Z \) to \( x \), not a copy.

GEN Flx_div_by_X_x(GEN A, ulong a, ulong p, ulong *rem), returns the Euclidean quotient of the Flx \( A \) by \( X - a \), and sets \( *rem \) to the remainder \( A(a) \).

ulong Flx_eval(GEN x, ulong y, ulong p), as FpX_eval.

GEN Flx_deflate(GEN P, long d) assuming \( P \) is a polynomial of the form \( Q(X^d) \), return \( Q \).

GEN Flx_splitting(GEN p, long k), as RgX_splitting.

GEN Flx_inflate(GEN P, long d) returns \( P(X^d) \).

int Flx_is_squarefree(GEN z, ulong p)

int Flx_is_irred(GEN f, ulong p), as FpX_is_irred.

int Flx_is_smooth(GEN f, long r, ulong p) return 1 if all irreducible factors of \( f \) are of degree at most \( r \), 0 otherwise.

long Flx_nbroots(GEN f, ulong p), as FpX_nbroots.

long Flx_nbfact(GEN z, ulong p), as FpX_nbfact.

GEN Flx_degfact(GEN f, ulong p), as FpX_degfact.

GEN Flx_nbfact_by_degree(GEN z, long *nb, ulong p) Assume that the Flx \( z \) is squarefree mod the prime \( p \). Returns a t_VECSMALL \( D \) with \( deg z \) entries, such that \( D[i] \) is the number of irreducible factors of degree \( i \). Set \( nb \) to the total number of irreducible factors (the sum of the \( D[i] \)).
void Flx_ffintersect(GEN P, GEN Q, long n, ulong p, GEN* SP, GEN* SQ, GEN MA, GEN MB), as FpX_ffintersect

GEN Flv_polint(GEN x, GEN y, ulong p, long sv) as FpV_polint, returning an Flx in variable v.

GEN Flv_roots_to_pol(GEN a, ulong p, long sv) as FpV_roots_to_pol returning an Flx in variable v.

7.2.13 FlxV. See FpXV operations.

GEN FlxV_Flq_mul(GEN V, GEN W, ulong p), as FpXV_Flq_mul.

GEN FlxV_red(GEN V, ulong p) reduces each components with Flx_red.

7.2.14 FlxT. See FpXT operations.

GEN FlxT_red(GEN V, ulong p) reduces each leaf with Flx_red.

7.2.15 Flxq. See FpXQ operations.

GEN Flxq_add(GEN x, GEN y, GEN T, ulong p)
GEN Flxq_sub(GEN x, GEN y, GEN T, ulong p)
GEN Flxq_mul(GEN x, GEN y, GEN T, ulong p)
GEN Flxq_sqr(GEN y, GEN T, ulong p)
GEN Flxq_inv(GEN x, GEN T, ulong p)
GEN Flxq_invsafe(GEN x, GEN T, ulong p)
GEN Flxq_div(GEN x, GEN y, GEN T, ulong p)
GEN Flxq_pow(GEN x, GEN n, GEN T, ulong p)
GEN Flxq_powu(GEN x, ulong n, GEN T, ulong p)
GEN Flxq_powers(GEN x, long n, GEN T, ulong p)
GEN Flxq_matrix_pow(GEN x, long m, long n, GEN T, ulong p), see FpXQ_matrix_pow.
GEN Flxq_autpow(GEN a, long n, GEN T, ulong p) see FpXQ_autpow.
GEN Flxq_autsum(GEN a, long n, GEN T, GEN p) see Flxq_autsum.
GEN Flxq_ffisom_inv(GEN S, GEN T, ulong p), as FpXQ_ffisom_inv.
GEN Flx_Flxq_eval(GEN f, GEN x, GEN T, ulong p) returns f(x).
GEN Flx_FlxqV_eval(GEN f, GEN x, GEN T, ulong p), see FpX_FpXQV_eval.
GEN FlxqV_roots_to_pol(GEN V, GEN T, ulong p, long v) as Fvq_roots_to_pol returning an FlxqX in variable v.
GEN Flxq_order(GEN a, GEN ord, GEN T, ulong p) returns the order of the t_Flxq a. If o is non-NULL, it is assumed that o is a multiple of the order of a, either as a t_INT or a factorization matrix.
int Flxq_issquare(GEN x, GEN T, ulong p) returns 1 if x is a square and 0 otherwise. Assume that T is irreducible mod p.
int Flxq_is2npower(GEN x, long n, GEN T, ulong p) returns 1 if \( x \) is a \( 2^n \)-th power and 0 otherwise. Assume that \( T \) is irreducible mod \( p \).

GEN Flxq_log(GEN a, GEN g, GEN ord, GEN T, ulong p) Let \( g \) of exact order \( \text{ord} \) in the field \( F_p[X]/(T) \). Return \( e \) such that \( a^e = g \). If \( e \) does not exists, the result is currently undefined. Assumes that \( T \) is irreducible mod \( p \).

GEN Flxq_sqrt(GEN x, GEN T, ulong p) returns a square root of \( x \). Return NULL if \( x \) is not a square.

GEN Flxq_lroot(GEN a, GEN T, ulong p) returns \( x \) such that \( x^p = a \).

GEN Flxq_lroot_fast(GEN a, GEN V, GEN T, ulong p) assuming that \( V = \text{Flxq_powers(s, p-1, T, p)} \) where \( s(x)^p \equiv x \pmod{T(x), p} \), returns \( b \) such that \( b^p = a \). Only useful if \( p \) is less than the degree of \( T \).

GEN Flxq_charpoly(GEN x, GEN T, ulong p) returns the characteristic polynomial of \( x \)

GEN Flxq_minpoly(GEN x, GEN T, ulong p) returns the minimal polynomial of \( x \)

ulong Flxq_norm(GEN x, GEN T, ulong p) returns the norm of \( x \)

ulong Flxq_trace(GEN x, GEN T, ulong p) returns the trace of \( x \)

GEN Flxq_conjvec(GEN x, GEN T, ulong p) returns the conjugates \( [x, x^p, x^{p^2}, \ldots, x^{p^{n-1}}] \) where \( n \) is the degree of \( T \).

GEN gener_Flxq(GEN T, ulong p, GEN *po) returns a primitive root modulo \( (T, p) \). \( T \) is an Flx assumed to be irreducible modulo the prime \( p \). If \( po \) is not NULL it is set to \([o, fa]\), where \( o \) is the order of the multiplicative group of the finite field, and \( fa \) is its factorization.

7.2.16 FlxX. See FpXX operations.

GEN poll_FlxX(long vX, long sx) returns the unit FlxX as a t_POL in variable \( vX \) which only coefficient is \( \text{pol1_Flx(sx)} \).

GEN polx_FlxX(long vX, long sx) returns the variable \( X \) as a degree 1 t_POL with Flx coefficients in the variable \( x \).

GEN FlxX_add(GEN P, GEN Q, ulong p)
GEN FlxX_sub(GEN P, GEN Q, ulong p)
GEN FlxX_Fl_mul(GEN x, ulong y, ulong p)
GEN FlxX_double(GEN x, ulong p)
GEN FlxX_triple(GEN x, ulong p)
GEN FlxX_neg(GEN x, ulong p)
GEN FlxX_Flx_add(GEN y, GEN x, ulong p)
GEN FlxX_Flx_mul(GEN x, GEN y, ulong p)
GEN FlxY_Flx_div(GEN x, GEN y, ulong p) divides the coefficients of \( x \) by \( y \) using Flx_div.
GEN FlxY_evalx(GEN x, ulong y, ulong p)
GEN FlxY_Flxq_evalx(GEN x, GEN y, ulong p)
GEN FlxX_renormalize(GEN x, long l), as normalizepol, where
l = lg(x), in place.
GEN FlxX_resultant(GEN u, GEN v, ulong p, long sv)
Returns ResX(u, v), which is an Flx.
The coefficients of u and v are assumed to be in the variable v.
GEN Flx_FlxY_resultant(GEN a, GEN b, ulong p)
Returns ResX(a, b), which is an Flx in the
main variable of b.
GEN FlxX_shift(GEN a, long n)
GEN FlxX_swap(GEN x, long n, long ws), as RgXY_swap.
GEN FlxYqq_pow(GEN x, GEN n, GEN S, GEN T, ulong p), as FpXYQQ_pow.

7.2.17 FlxqX. See FpXQX operations.
GEN zxX_to_Kronecker(GEN P, GEN Q) assuming
P(X, Y) is a polynomial of degree in X strictly
less than n, returns P(X, X^2n−1), the Kronecker form of P.
GEN Kronecker_to_FlxqX(GEN z, GEN T, ulong p). Let n = deg T and let
P(X, Y) ∈ Z[X, Y] lift a polynomial in K[Y], where K := F_p[X]/(T) and degX P < 2n−1 — such as would result from
multiplying minimal degree lifts of two polynomials in K[Y]. Let z = P(t, t^2n−1) be a Kronecker
form of P, this function returns Q ∈ Z[X, t] such that Q is congruent to P(X, t) mod (p, T(X)),
degX Q < n, and all coefficients are in [0, p]. Not stack-clean. Note that t need not be the same
variable as Y!
GEN FlxqX_red(GEN z, GEN T, ulong p)
GEN FlxqX_normalize(GEN z, GEN T, ulong p)
GEN FlxqX_mul(GEN x, GEN y, GEN T, ulong p)
GEN FlxqX_Flxq_mul(GEN P, GEN U, GEN T, ulong p)
GEN FlxqX_Flxq_mul_to_monic(GEN P, GEN U, GEN T, ulong p) returns P * U assuming the
result is monic of the same degree as P (in particular U ≠ 0).
GEN FlxqX_sqr(GEN x, GEN T, ulong p)
GEN FlxqX_pow(GEN x, long n, GEN T, ulong p)
GEN FlxqX_divrem(GEN x, GEN y, GEN T, ulong p, GEN *pr)
GEN FlxqX_div(GEN x, GEN y, GEN T, ulong p)
GEN FlxqX_rem(GEN x, GEN y, GEN T, ulong p)
GEN FlxqX_invBarrett(GEN T, GEN Q, ulong p)
GEN FlxqX_rem_Barrett(GEN x, GEN mg, GEN T, GEN Q, ulong p)
GEN FlxqX_gcd(GEN x, GEN y, ulong p) returns a (not necessarily monic) greatest common di-
visor of x and y.
GEN FlxqX_extgcd(GEN x, GEN y, GEN T, ulong p, GEN *ptu, GEN *ptv)
GEN FlxqXV_prod(GEN V, GEN T, ulong p)

115
GEN FlxqX_safegcd(GEN P, GEN Q, GEN T, ulong p) Returns the monic GCD of P and Q if Euclid’s algorithm succeeds and NULL otherwise. In particular, if \( p \) is not prime or \( T \) is not irreducible over \( \mathbb{F}_p[X] \), the routine may still be used (but will fail if non-invertible leading terms occur).

GEN FlxqX_Frobenius(GEN S, GEN T, GEN p), as FpXQX_Frobenius

GEN FlxqXQ_halfFrobenius(GEN A, GEN S, GEN T, GEN p), as FpXQXQ_halfFrobenius

long FlxqX_nroots(GEN S, GEN T, GEN p), as FpX_nroots.

GEN FlxqX_FlxqXQ_eval(GEN Q, GEN x, GEN S, GEN T, ulong p) as FpX_FpXQQ_eval.

GEN FlxqX_FlxqXQV_eval(GEN P, GEN V, GEN S, GEN T, ulong p) as FpX_FpXQVQV_eval.

7.2.18 FlxqXQ. See FpXQXQ operations.

GEN FlxqXQ_mul(GEN x, GEN y, GEN S, GEN T, ulong p)

GEN FlxqXQ sqr(GEN x, GEN S, GEN T, ulong p)

GEN FlxqXQ_inv(GEN x, GEN S, GEN T, ulong p)

GEN FlxqXQ_invsafe(GEN x, GEN S, GEN T, ulong p)

GEN FlxqXQ_div(GEN x, GEN y, GEN S, GEN T, ulong p)

GEN FlxqXQ_pow(GEN x, GEN n, GEN S, GEN T, ulong p)

GEN FlxqXQ_powers(GEN x, long n, GEN S, GEN T, ulong p)

GEN FlxqXQ_matrix_pow(GEN x, long n, long m, GEN S, GEN T, ulong p)

GEN FlxqXQV_autpow(GEN a, long n, GEN S, GEN T, ulong p) as FpXQXQV_autpow

GEN FlxqXQV_autsum(GEN a, long n, GEN S, GEN T, ulong p) as FpXQXQV_autsum

7.2.19 F2x. An F2x z is a t_VECSMALL representing a polynomial over \( \mathbb{F}_2[X] \). Specifically \( z[0] \) is the usual codeword, \( z[1] = \text{evalvarn}(v) \) for some variable \( v \) and the coefficients are given by the bits of remaining words by increasing degree.

7.2.19.1 Basic operations.

ulong F2x_coeff(GEN x, long i) returns the coefficient \( i \geq 0 \) of \( x \).

void F2x_clear(GEN x, long i) sets the coefficient \( i \geq 0 \) of \( x \) to 0.

void F2x_flip(GEN x, long i) adds 1 to the coefficient \( i \geq 0 \) of \( x \).

void F2x_set(GEN x, long i) sets the coefficient \( i \geq 0 \) of \( x \) to 1.

GEN Flx_to_F2x(GEN x)

GEN Z_to_F2x(GEN x, long v)

GEN ZX_to_F2x(GEN x)

GEN F2v_to_F2x(GEN x, long sv)

GEN ZXX_to_F2x(GEN x, long v)

GEN F2x_to_Flx(GEN x)

116
GEN F2x_to_ZX(GEN x)
GEN pol0_F2x(long sv) returns a zero F2x in variable v.
GEN zero_F2x(long sv) alias for pol0_F2x.
GEN pol1_F2x(long sv) returns the F2x in variable v constant to 1.
GEN polx_F2x(long sv) returns the variable v as degree 1 F2x.
GEN random_F2x(long d, long sv) returns a random F2x in variable v, of degree less than d.
long F2x_degree(GEN x) returns the degree of the F2x x. The degree of 0 is defined as −1.
int F2x_equal1(GEN x)
int F2x_equal(GEN x, GEN y)
GEN F2x_1_add(GEN y) returns y+1 where y is a Flx.
GEN F2x_add(GEN x, GEN y)
GEN F2x_mul(GEN x, GEN y)
GEN F2x_sqr(GEN x)
GEN F2x_divrem(GEN x, GEN y, GEN *pr)
GEN F2x_rem(GEN x, GEN y)
GEN F2x_div(GEN x, GEN y)
GEN F2x_renormalize(GEN x, long lx)
GEN F2x_deriv(GEN x)
GEN F2x_deflate(GEN x, long d)
void F2x_shift(GEN x, long d) as RgX_shift
void F2x_even_odd(GEN p, GEN *pe, GEN *po) as RgX_even_odd
long F2x_valrem(GEN x, GEN *Z)
GEN F2x_extgcd(GEN a, GEN b, GEN *ptu, GEN *ptv)
GEN F2x_gcd(GEN a, GEN b)
GEN F2x_halfgcd(GEN a, GEN b)
int F2x_issquare(GEN x) returns 1 if x is a square of a F2x and 0 otherwise.
int F2x_is_irred(GEN f), as FpX_is_irred.
GEN F2x_sqrt(GEN x) returns the squareroot of x, assuming x is a square of a F2x.
GEN F2x_factor(GEN f)
7.2.20 F2xq. See FpXQ operations.

GEN F2xq_mul(GEN x, GEN y, GEN pol)
GEN F2xq_sqr(GEN x, GEN pol)
GEN F2xq_div(GEN x, GEN y, GEN T)
GEN F2xq_inv(GEN x, GEN T)
GEN F2xq_invsafe(GEN x, GEN T)
GEN F2xq_pow(GEN x, GEN n, GEN pol)
GEN F2xq_powu(GEN x, ulong n, GEN pol)
ulong F2xq_trace(GEN x, GEN T)
GEN F2xq_conjvec(GEN x, GEN T) returns the vector of conjugates $[x, x^2, x^2^2, \ldots, x^2^{n-1}]$ where $n$ is the degree of $T$.
GEN F2xq_log(GEN a, GEN g, GEN ord, GEN T)
GEN F2xq_order(GEN a, GEN ord, GEN T)
GEN F2xq_Artin_Schreier(GEN a, GEN T) returns a solution of $x^2 + x = a$, assuming it exists.
GEN F2xq_sqrt(GEN a, GEN T)
GEN F2xq_sqrt_fast(GEN a, GEN s, GEN T) assuming that $s^2 \equiv x \pmod{T(x)}$, computes $b \equiv a(s) \pmod{T}$ so that $b^2 = a$.
GEN F2xq_conjvec(GEN x, GEN T, GEN *zeta)
GEN gener_F2xq(GEN T, GEN *po)
GEN F2xq_powers(GEN x, long n, GEN T)
GEN F2x_F2xq_eval(GEN f, GEN x, GEN T)
GEN F2x_F2xqV_eval(GEN f, GEN x, GEN T), see FpX_FpXV_eval.

7.2.21 F2xqV, F2xqM. See FqV, FqM operations.

GEN F2xqM_F2xqC_mul(GEN a, GEN b, GEN T)
GEN F2xqM_ker(GEN x, GEN T)
GEN F2xqM_det(GEN a, GEN T)
GEN F2xqM_image(GEN x, GEN T)
GEN F2xqM_inv(GEN a, GEN T)
GEN F2xqM_mul(GEN a, GEN b, GEN T)
long F2xqM_rank(GEN x, GEN T)
GEN matid_F2xqM(long n, GEN T)

118
7.2.22 Functions returning objects with \texttt{t\_INTMOD} coefficients.

Those functions are mostly needed for interface reasons: \texttt{t\_INTMODs} should not be used in library mode since the modular kernel is more flexible and more efficient, but GP users do not have access to the modular kernel. We document them for completeness:

- **\texttt{GEN Fp\_to\_mod(GEN z, GEN p)}**, \texttt{z} a \texttt{t\_INT}. Returns \( z \times \text{Mod}(1,p) \), normalized. Hence the returned value is a \texttt{t\_INTMOD}.

- **\texttt{GEN FpX\_to\_mod(GEN z, GEN p)}**, \texttt{z} a \texttt{ZX}. Returns \( z \times \text{Mod}(1,p) \), normalized. Hence the returned value has \texttt{t\_INTMOD} coefficients.

- **\texttt{GEN FpC\_to\_mod(GEN z, GEN p)}**, \texttt{z} a \texttt{ZC}. Returns \( \text{Col}(z) \times \text{Mod}(1,p) \), a \texttt{t\_COL} with \texttt{t\_INTMOD} coefficients.

- **\texttt{GEN FpV\_to\_mod(GEN z, GEN p)}**, \texttt{z} a \texttt{ZV}. Returns \( \text{Vec}(z) \times \text{Mod}(1,p) \), a \texttt{t\_VEC} with \texttt{t\_INTMOD} coefficients.

- **\texttt{GEN FpVV\_to\_mod(GEN z, GEN p)}**, \texttt{z} a \texttt{ZVV}. Returns \( \text{Vec}(z) \times \text{Mod}(1,p) \), a \texttt{t\_VEC} of \texttt{t\_VEC} with \texttt{t\_INTMOD} coefficients.

- **\texttt{GEN FpM\_to\_mod(GEN z, GEN p)}**, \texttt{z} a \texttt{ZM}. Returns \( z \times \text{Mod}(1,p) \), with \texttt{t\_INTMOD} coefficients.

- **\texttt{GEN F2c\_to\_mod(GEN x)}**

- **\texttt{GEN F2m\_to\_mod(GEN x)}**

- **\texttt{GEN Flc\_to\_mod(GEN z)}**

- **\texttt{GEN Flm\_to\_mod(GEN x)}**

- **\texttt{GEN FpXQC\_to\_mod(GEN V, GEN T, GEN p)}**, \( V \) being a vector of \texttt{FpXQ}, converts each entry to a \texttt{t\_POLMOD} with \texttt{t\_INTMOD} coefficients, and return a \texttt{t\_COL}.

- **\texttt{GEN QXQV\_to\_mod(GEN V, GEN T)}**, \( V \) a vector of \texttt{QXQ}, which are lifted representatives of elements of \( \mathbb{Q}[X]/(T) \) (number field elements in most applications) and \( T \) is in \( \mathbb{Z}[X] \). Return a vector where all non-rational entries are converted to \texttt{t\_POLMOD} modulo \( T \); no reduction mod \( T \) is attempted: the representatives should be already reduced. Used to normalize the output of \texttt{nfroots}.

- **\texttt{GEN QXQXV\_to\_mod(GEN V, GEN T)}**, \( V \) a vector of polynomials whose coefficients are \texttt{QXQ}. Analogous to \texttt{QXQV\_to\_mod}. Used to normalize the output of \texttt{nffactor}.

- **\texttt{GEN QXQX\_to\_mod\_shallow(GEN z, GEN T)}**, \( v \) a polynomial with \texttt{QXQ} coefficients; replace them by \texttt{mkpolmod(.,T)}. Shallow function.

The following functions are obsolete and should not be used: they receive a polynomial with arbitrary coefficients, apply \texttt{RgX\_to\_FpX}, a function from the modular kernel, then \texttt{\_to\_mod}:

- **\texttt{GEN rootmod(GEN f, GEN p)}**, applies \texttt{FpX\_roots}.

- **\texttt{GEN rootmod2(GEN f, GEN p)}**, applies \texttt{ZX\_to\_flx} then \texttt{Flx\_roots\_naive}.

- **\texttt{GEN factmod(GEN f, GEN p)}** applies \texttt{FpX\_factor}.

- **\texttt{GEN simplefactmod(GEN f, GEN p)}** applies \texttt{FpX\_degfact}.
7.2.23 Chinese remainder theorem over \( \mathbb{Z} \).

**GEN Z_chinese(GEN a, GEN b, GEN A, GEN B)** returns the integer in \([0, \text{lcm}(A, B)]\) congruent to \(a \mod A\) and \(b \mod B\), assuming it exists; in other words, that \(a\) and \(b\) are congruent mod \(\gcd(A, B)\).

**GEN Z_chinese_all(GEN a, GEN b, GEN A, GEN B, GEN *pC)** as **Z_chinese**, setting \(*pC\) to the \(\text{lcm}\) of \(A\) and \(B\).

**GEN Z_chinese_coprime(GEN a, GEN b, GEN A, GEN B, GEN C)**, as **Z_chinese**, assuming that \(\gcd(A, B) = 1\) and that \(C = \text{lcm}(A, B) = AB\).

void **Z_chinese_pre(GEN A, GEN B, GEN *pC, GEN *pU, GEN *pd)** initializes chinese remainder computations modulo \(A\) and \(B\). Sets \(*pC\) to \(\text{lcm}(A, B)\), \(*pd\) to \(\gcd(A, B)\), \(*pU\) to an integer congruent to 0 mod \((A/d)\) and 1 mod \((B/d)\). It is allowed to set \(pd = \text{NULL}\), in which case, \(d\) is still computed, but not saved.

**GEN Z_chinese_post(GEN a, GEN b, GEN C, GEN U, GEN d)** returns the solution to the chinese remainder problem \(x\) congruent to \(a\) mod \(A\) and \(b\) mod \(B\), where \(C, U, d\) were set in **Z_chinese_pre**. If \(d\) is \(\text{NULL}\), assume the problem has a solution. Otherwise, return NULL if it has no solution.

The following pair of functions is used in homomorphic imaging schemes, when reconstructing an integer from its images modulo pairwise coprime integers. The idea is as follows: we want to discover an integer \(H\) which satisfies \(|H| < B\) for some known bound \(B\); we are given pairs \((H_p, p)\) with \(H\) congruent to \(H_p\) modulo \(p\) and all \(p\) pairwise coprime.

Given \(H\) congruent to \(H_p\) modulo a number of \(p\), whose product is \(q\), and a new pair \((H_p, p)\), \(p\) coprime to \(q\), the following incremental functions use the chinese remainder theorem (CRT) to find a new \(H\), congruent to the preceding one modulo \(q\), but also to \(H_p\) modulo \(p\). It is defined uniquely modulo \(qp\), and we choose the centered representative. When \(P\) is larger than \(2B\), we have \(H = H_p\), but of course, the value of \(H\) may stabilize sooner. In many applications it is possible to directly check that such a partial result is correct.

**GEN Z_init_CRT(ulong Hp, ulong p)** given a Fl \(Hp\) in \([0, p - 1]\), returns the centered representative \(H\) congruent to \(Hp\) modulo \(p\).

int **Z_incremental_CRT(GEN *H, ulong Hp, GEN *q, ulong p)** given a t_INT \(*H\), centered modulo \(*q\), a new pair \((Hp, p)\) with \(p\) coprime to \(q\), this function updates \(*H\) so that it also becomes congruent to \((Hp, p)\), and \(*q\) to the product \(qp = p \cdot *q\). It returns 1 if the new value is equal to the old one, and 0 otherwise.

**GEN chinese1_coprime_Z(GEN v)** an alternative divide-and-conquer implementation: \(v\) is a vector of t_INTMOD with pairwise coprime moduli. Return the t_INTMOD solving the corresponding chinese remainder problem. This is a streamlined version of

**GEN chinese1(GEN v)**, which solves a general chinese remainder problem (not necessarily over \(\mathbb{Z}\), moduli not assumed coprime).

As above, for \(H\) a ZX: we assume that \(H\) and all \(Hp\) have dimension > 0. The original \(*H\) is destroyed.

**GEN ZM_init_CRT(GEN Hp, ulong p)**

int **ZM_incremental_CRT(GEN *H, GEN Hp, GEN *q, ulong p)**

As above for \(H\) a ZX: note that the degree may increase or decrease. The original \(*H\) is destroyed.
GEN ZX_init_CRT(GEN Hp, ulong p, long v)
int ZX_incremental_CRT(GEN *H, GEN Hp, GEN *q, ulong p)

7.2.24 Rational reconstruction.

int Fp_ratlift(GEN x, GEN m, GEN amax, GEN bmax, GEN *a, GEN *b). Assuming that 0 ≤ x < m, amax ≥ 0, and bmax > 0 are t_Ints, and that 2amaxbmax < m, attempts to recognize x as a rational a/b, i.e. to find t_Ints a and b such that

• a ≡ bx modulo m,
• |a| ≤ amax, 0 < b ≤ bmax,
• gcd(m, b) = gcd(a, b).

If unsuccessful, the routine returns 0 and leaves a, b unchanged; otherwise it returns 1 and sets a and b.

In almost all applications, we actually know that a solution exists, as well as a non-zero multiple B of b, and m = pℓ is a prime power, for a prime p chosen coprime to B hence to b. Under the single assumption gcd(m, b) = 1, if a solution a, b exists satisfying the three conditions above, then it is unique.

GEN FpM_ratlift(GEN M, GEN m, GEN amax, GEN bmax, GEN denom) given an FpM modulo m with reduced or Fp_center-ed entries, reconstructs a matrix with rational coefficients by applying Fp_ratlift to all entries. Assume that all preconditions for Fp_ratlift are satisfied, as well gcd(m, b) = 1 (so that the solution is unique if it exists). Return NULL if the reconstruction fails, and the rational matrix otherwise. If denom is not NULL check further that all denominators divide denom.

The functions is not stack clean if one coefficients of M is negative (centered residues), but still suitable for gerepileup_to.

GEN FpX_ratlift(GEN P, GEN m, GEN amax, GEN bmax, GEN denom) as FpM_ratlift, where P is an FpX.

GEN FpC_ratlift(GEN P, GEN m, GEN amax, GEN bmax, GEN denom) as FpM_ratlift, where P is an FpC.

7.2.25 Hensel lifts.

GEN Zp_sqrtlift(GEN b, GEN a, GEN p, long e) let a, b, p be t_Ints, with p > 1 odd, such that a² ≡ b mod p. Returns a t_INT A such that A² ≡ b mod p^e. Special case of Zp_sqrtnlift.

GEN Zp_sqrtnlift(GEN b, GEN n, GEN a, GEN p, long e) let a, b, n, p be t_Ints, with n, p > 1, and p coprime to n, such that a^n ≡ b mod p. Returns a t_INT A such that A^n ≡ b mod p^e. Special case of ZpXQ_invliftroot.

GEN Zp_teichmuller(GEN x, GEN p, long e, GEN pe) for p an odd prime, x a t_INT coprime to p, and pe = p^r, returns the (p − 1)-th root of 1 congruent to x modulo p, modulo p^r. For convenience, p = 2 is also allowed and we return 1 (x is 1 mod 4) or 2^r − 1 (x is 3 mod 4).

GEN ZpXQ_invlift(GEN b, GEN a, GEN T, GEN p, long e) let p be a prime t_INT and a, b be FpXQs (modulo T) such that ab ≡ 1 mod (p, T). Returns an FpXQ A such that Ab ≡ 1 mod (p^r, T). Special case of ZpXQ_invroot.
GEN ZpXQ_inv(GEN b, GEN T, GEN p, long e) let \( p \) be a prime \( t \_INT \) and \( b \) be a \( \text{FpXQ} \) (modulo \( T, p^e \)). Returns an \( \text{FpXQ} \) \( A \) such that \( Ab \equiv 1 \mod (p^e, T) \).

GEN ZpXQ_sqrtnlift(GEN b, GEN n, GEN a, GEN T, GEN p, long e) let \( n, p \) be \( t \_INTs \), with \( n, p > 1 \) and \( p \) coprime to \( n \), and \( a, b \) be \( \text{FpXQs} \) (modulo \( T \)) such that \( a^n \equiv b \mod (p, T) \). Returns an \( \text{FqX} \) \( A \) such that \( A^n \equiv b \mod (p^e, T) \). Special case of \( \text{ZpXQ} \_liftroot \).

GEN rootpadicfast(GEN f, GEN p, long e) \( f \) a \( \text{ZX} \) with leading term prime to \( p \), and without multiple roots mod \( p \). Return a vector of \( t \_INTs \) which are the roots of \( f \mod p^e \). This is a very important special case of \( \text{rootpadic} \).

GEN ZpX \_liftroot(GEN f, GEN a, GEN p, long e) \( f \) a \( \text{ZX} \) with leading term prime to \( p \), and \( a \) a root mod \( p \) such that \( v_p(f'(a)) = 0 \). Return a \( t \_VEC \) of \( t \_INTs \) which are the \( p \)-adic roots of \( f \) congruent to \( a \mod p \) (given modulo \( p^e \)).

GEN ZpX \_liftroots(GEN f, GEN S, GEN p, long e) \( f \) a \( \text{ZX} \) with leading term prime to \( p \), and \( S \) a vector of simple roots mod \( p \). Return a vector of \( t \_INTs \) which are the root of \( f \mod p^e \) congruent to the \( S[i] \mod p \).

GEN ZpXQX \_liftroot(GEN f, GEN a, GEN T, GEN p, long e) as \( \text{ZpX} \_liftroot \), but \( f \) is now a polynomial in \( \mathbb{Z}[X, Y] \) and we find a root in the unramified extension of \( \mathbb{Q}_p \) with residue field \( \mathbb{F}_p[Y]/(T) \), assuming \( v_p(f(a)) > 0 \) and \( v_p(f'(a)) = 0 \).

GEN ZpXQX \_liftroot \_valid(GEN f, GEN a, long v, GEN T, GEN p, long e) returns the roots of \( f \) as \( \text{ZpXQX} \_liftroot \), where \( v \) is the valuation of the content of \( f' \) and it is required that \( v_p(f(a)) > v \) and \( v_p(f'(a)) = v \).

GEN ZpX \_liftfact(GEN A, GEN B, GEN T, GEN p, long e, GEN pe) is the routine underlying \( \text{polhensellift} \). Here, \( p \) is prime, \( T(Y) \) defines a finite field \( \mathbb{F}_q \), either \( \mathbb{F}_q = \mathbb{F}_p \ (T \ \text{NULL}) \) or a non-prime finite field \( T \) an \( \text{FpX} \). \( A \) is a polynomial in \( \mathbb{Z}[X] \) \( (T \ \text{NULL}) \) or \( \mathbb{Z}[X, Y] \), whose leading coefficient is non-zero in \( \mathbb{F}_p \). \( B \) is a vector of monic \( \text{FpX} \) \( (T \ \text{NULL}) \) or \( \text{FpX} \), pairwise coprime in \( \mathbb{F}_q[X] \), whose product is congruent to \( A/\text{lcm}(A) \) in \( \mathbb{F}_q[X] \). Lifts the elements of \( B \mod \text{pe} = p^e \), such that the congruence now holds mod \( (T, p^e) \).

The following technical function returns an optimal sequence of \( p \)-adic accuracies, for a given target accuracy:

\[ \text{ulong quadratic \_prec \_mask(ulong n)} \] we want to reach accuracy \( n \geq 1 \), starting from accuracy 1, using a quadratically convergent, self-correcting, algorithm; in other words, from inputs correct to accuracy \( l \) one iteration outputs a result correct to accuracy \( 2l \). For instance, to reach \( n = 9 \), we want to use accuracies \([1, 2, 3, 5, 9]\) instead of \([1, 2, 4, 8, 9]\). The idea is to essentially double the accuracy at each step, and not overshoot in the end.

Let \( a_0 = 1 \), \( a_1 = 2 \), \ldots, \( a_k = n \), be the desired sequence of accuracies. To obtain it, we work backwards and set

\[ a_k = n, \quad a_{i-1} = (a_i + 1) \backslash 2. \]

This is in essence what the function returns. But we do not want to store the \( a_i \) explicitly, even as a \( t \_VEC \text{SMALL} \), since this would leave an object on the stack. Instead, we store \( a_i \) implicitly in a bitmask \( \text{MASK} \): let \( a_0 = 1 \), if the \( i \)-th bit of the mask is set, set \( a_{i+1} = 2a_i - 1 \), and \( 2a_i \) otherwise;
in short the bits indicate the places where we do something special and do not quite double the accuracy (which would be the straightforward thing to do).

In fact, to avoid returning separately the mask and the sequence length $k + 1$, the function returns $\text{MASK} + 2^{k+1}$, so the highest bit of the mask indicates the length of the sequence, and the following ones give an algorithm to obtain the accuracies. This is much simpler than it sounds, here is what it looks like in practice:

```c
ulong mask = quadratic_prec_mask(n);
long l = 1;
while (mask > 1) { /* here, the result is known to accuracy l */
  l = 2*l; if (mask & 1) l--; /* new accuracy l for the iteration */
  mask >>= 1; /* pop low order bit */
  /* ... lift to the new accuracy ... */
}
/* we are done. At this point l = n */
```

We just pop the bits in `mask` starting from the low order bits, stop when `mask` is 1 (that last bit corresponds to the $2^{k+1}$ that we added to the mask proper). Note that there is nothing specific to Hensel lifts in that function; it would work equally well for an Archimedean Newton iteration.

Note that in practice, we rather use an infinite loop, and insert an

```c
if (mask == 1) break;
```

in the middle of the loop: the loop body usually includes preparations for the next iterations (e.g. lifting Bezout coefficients in a quadratic Hensel lift), which are costly and useless in the last iteration.

### 7.2.26 Other $p$-adic functions.

`long ZpX_disc_val(GEN f, GEN p)` returns the valuation at $p$ of the discriminant of $f$. Assume that $f$ is a monic separable $\mathbb{Z}[X]$ and that $p$ is a prime number. Proceeds by dynamically increasing the $p$-adic accuracy; infinite loop if the discriminant of $f$ is 0.

`long ZpX_resultant_val(GEN f, GEN g, GEN p, long M)` returns the valuation at $p$ of $\text{Res}(f,g)$. Assume $f,g$ are both $\mathbb{Z}[X]$, and that $p$ is a prime number coprime to the leading coefficient of $f$. Proceeds by dynamically increasing the $p$-adic accuracy. To avoid an infinite loop when the resultant is 0, we return $M$ if the Sylvester matrix mod $p^M$ still does not have maximal rank.

`GEN ZpX_gcd(GEN f, GEN g, GEN p, GEN pm)` $f$ a monic $\mathbb{Z}[X]$, $g$ a $\mathbb{Z}[X]$, $pm = p^m$ a prime power. There is a unique integer $r \geq 0$ and a monic $h \in \mathbb{Q}_p[X]$ such that

$$p^r h \mathbb{Z}_p[X] + p^m \mathbb{Z}_p[X] = f \mathbb{Z}_p[X] + g \mathbb{Z}_p[X] + p^m \mathbb{Z}_p[X].$$

Return the 0 polynomial if $r \geq m$ and a monic $h \in \mathbb{Z}[1/p][X]$ otherwise (whose valuation at $p$ is $> -m$).

`GEN ZpX_reduced_resultant(GEN f, GEN g, GEN p, GEN pm)` $f$ a monic $\mathbb{Z}[X]$, $g$ a $\mathbb{Z}[X]$, $pm = p^m$ a prime power. The $p$-adic reduced resultant of $f$ and $g$ is 0 if $f$, $g$ not coprime in $\mathbb{Z}_p[X]$, and otherwise the generator of the form $p^d$ of

$$(f \mathbb{Z}_p[X] + g \mathbb{Z}_p[X]) \cap \mathbb{Z}_p.$$

Return the reduced resultant modulo $p^m$. 

123
GEN ZpX_reduced_resultant_fast(GEN f, GEN g, GEN p, long M) f a monic ZX, g a ZX, p a prime. Returns the the p-adic reduced resultant of f and g modulo $p^M$. This function computes resultants for a sequence of increasing p-adic accuracies (up to $M p$-adic digits), returning as soon as it obtains a non-zero result. It is very inefficient when the resultant is 0, but otherwise usually more efficient than computations using a priori bounds.

GEN ZpM_echelon(GEN M, long early_abort, GEN p, GEN pm) given a $Zp$ M, a prime p and pm = $p^m$, returns an echelon form $E$ for $M \bmod p^m$. I.e. there exist a square integral matrix $U$ with det $U$ coprime to $p$ such that $E = MU \bmod p^m$. I early_abort is non-zero, return NULL as soon as one pivot in the echelon form is divisible by $p^m$. The echelon form is an upper triangular HNF, we do not waste time to reduce it to Gauss-Jordan form.

GEN zlm_echelon(GEN M, long early_abort, ulong p, ulong pm) variant of ZpM_echelon, for a $Zlm$ M.

GEN ZpXQ_log(GEN a, GEN T, GEN p, long e) T being a ZpX irreducible modulo p, return the logarithm of a in $Z_p[X]/(T)$ to precision e, assuming that a ≡ 1 (mod $pZ_p[X]$) if p odd or a ≡ 1 (mod 4$Z_2[X]$) if p = 2.

GEN padic_to_Q(GEN x) truncate the t_PADIC to a t_INT or t_FRAC.

GEN padic_to_Q_shallow(GEN x) shallow version of padic_to_Q

GEN QpV_to_QV(GEN v) apply padic_to_Q_shallow

long padicprec(GEN x, GEN p) returns the absolute p-adic precision of the object x, by definition the minimum precision of the components of x. For a non-zero t_PADIC, this returns $\text{valp}(x) + \text{precp}(x)$.

long padicprec_relative(GEN x) returns the relative p-adic precision of the t_INT, t_FRAC, or t_PADIC x (minimum precision of the components of x for t_POL or vector/matrices). For a t_PADIC, this returns $\text{precp}(x)$ if x ≠ 0, and 0 for x = 0.

7.2.27 Conversions involving single precision objects.

7.2.27.1 To single precision.

ulong Rg_to_Fl(GEN z, ulong p) z which can be mapped to $Z/pZ$: a t_INT, a t_INTMOD whose modulus is divisible by p, a t_FRAC whose denominator is coprime to p, or a t_PADIC with underlying prime $\ell$ satisfying $p = \ell^n$ for some n (less than the accuracy of the input). Returns lift(z * Mod(1,p)), normalized, as an Fl.

ulong Rg_to_F2(GEN z), as Rg_to_Fl for p = 2.

ulong padic_to_Fl(GEN x, ulong p) special case of Rg_to_Fl, for a x a t_PADIC.

GEN RgX_to_F2x(GEN x), x a t_POL, returns the F2x obtained by applying Rg_to_Fl coefficientwise.

GEN RgX_to_Flx(GEN x, ulong p), x a t_POL, returns the Flx obtained by applying Rg_to_Fl coefficientwise.

GEN Rg_to_F2xq(GEN z, GEN T), z a GEN which can be mapped to $F_2[X]/(T)$: anything Rg_to_Fl can be applied to, a t_POL to which RgX_to_F2x can be applied to, a t_POLMOD whose modulus is divisible by T (once mapped to a F2x), a suitable t_RFRAC. Returns z as an F2xq, normalized.

GEN Rg_to_Flxq(GEN z, GEN T, ulong p), z a GEN which can be mapped to $F_p[X]/(T)$: anything Rg_to_Fl can be applied to, a t_POL to which RgX_to_Flx can be applied to, a t_POLMOD whose
modulus is divisible by $T$ (once mapped to a Flx), a suitable $t_{RFRAC}$. Returns $z$ as an Flxq, normalized.

GEN ZX_to_Flx(GEN x, ulong p) reduce ZX x modulo p (yielding an Flx). Faster than RgX_to_Flx.

GEN ZV_to_Flv(GEN x, ulong p) reduce ZV x modulo p (yielding an Flv).

GEN ZXV_to_FlxV(GEN v, ulong p), as ZX_to_Flx, repeatedly called on the vector’s coefficients.

GEN ZXX_to_FlxX(GEN B, ulong p, long v), as ZX_to_Flx, repeatedly called on the polynomial’s coefficients.

GEN ZXXV_to_FlxXV(GEN V, ulong p, long v), as ZXX_to_FlxX, repeatedly called on the vector’s coefficients.

GEN RgC_to_Flc(GEN x, ulong p) reduce the t_VEC/t_COL x modulo p, yielding a t_VECSMALL.

GEN RgM_to_Flm(GEN x, ulong p) reduce the t_MAT x modulo p.

GEN ZM_to_Flm(GEN x, ulong p) reduce ZM x modulo p (yielding an Flm).

GEN ZV_to_zv(GEN z), converts coefficients using itos

GEN ZV_to_nv(GEN z), converts coefficients using itou

GEN ZM_to_zm(GEN z), converts coefficients using itos

GEN FqC_to_FlxC(GEN x, GEN T, GEN p), converts coefficients in Fq to coefficient in Flx, result being a column vector.

GEN FqV_to_FlxV(GEN x, GEN T, GEN p), converts coefficients in Fq to coefficient in Flx, result being a line vector.

GEN FqM_to_FlxM(GEN x, GEN T, GEN p), converts coefficients in Fq to coefficient in Flx.

7.2.27.2 From single precision.

GEN Flx_to_ZX(GEN z), converts to ZX (t_POL of non-negative t_INTs in this case)

GEN Flx_to_FlxX(GEN z), converts to FlxX (t_POL of constant Flx in this case).

GEN Flx_to_ZX_inplace(GEN z), same as Flx_to_ZX, in place (z is destroyed).

GEN FlxX_to_ZXX(GEN B), converts an FlxX to a polynomial with ZX or t_INT coefficients (repeated calls to Flx_to_ZX).

GEN FlxC_to_ZXC(GEN x), converts a vector of Flx to a column vector of polynomials with t_INT coefficients (repeated calls to Flx_to_ZX).

GEN FlxV_to_ZXV(GEN x), as above but return a t_VEC.

void F2xV_to_FlxV_inplace(GEN v) v is destroyed.

void F2xV_to_ZXV_inplace(GEN v) v is destroyed.

void FlxV_to_ZXV_inplace(GEN v) v is destroyed.

GEN FlxM_to_ZXM(GEN z), converts a matrix of Flx to a matrix of polynomials with t_INT coefficients (repeated calls to Flx_to_ZX).
GEN zx_to_ZX(GEN z), as Flx_to_ZX, without assuming coefficients are non-negative.

GEN Flc_to_ZC(GEN z), converts to ZC (t_COL of non-negative t_INTs in this case)

GEN Flv_to_ZV(GEN z), converts to ZV (t_VEC of non-negative t_INTs in this case)

GEN Flm_to_ZM(GEN z), converts to ZM (t_MAT with non-negative t_INTs coefficients in this case)

GEN zc_to_ZC(GEN z) as Flc_to_ZC, without assuming coefficients are non-negative.

GEN zv_to_ZV(GEN z) as Flv_to_ZV, without assuming coefficients are non-negative.

GEN zm_to_ZM(GEN z) as Flm_to_ZM, without assuming coefficients are non-negative.

GEN zv_to_Flv(GEN z, ulong p)

GEN zm_to_Flm(GEN z, ulong p)

7.2.27.3 Mixed precision linear algebra. Assumes dimensions are compatible. Multiply a multiprecision object by a single-precision one.

GEN RgM_zc_mul(GEN x, GEN y)
GEN RgM_zm_mul(GEN x, GEN y)
GEN RgV_zc_mul(GEN x, GEN y)
GEN RgV_zm_mul(GEN x, GEN y)
GEN ZM_zc_mul(GEN x, GEN y)
GEN ZM_zm_mul(GEN x, GEN y)
GEN ZC_z_mul(GEN x, long y)
GEN ZM_nm_mul(GEN x, GEN y) the entries of y are ulongs.
GEN nm_Z_mul(GEN y, GEN c) the entries of y are ulongs.

7.2.27.4 Miscellaneous involving Fl.

GEN Fl_to_Flx(ulong x, long evx) converts a unsigned long to a scalar Flx. Assume that evx = evalvarn(vx) for some variable number vx.

GEN Z_to_Flx(GEN x, ulong p, long v) converts a t_INT to a scalar polynomial in variable v.

GEN Flx_to_Flv(GEN x, long n) converts from Flx to Flv with n components (assumed larger than the number of coefficients of x).

GEN zx_to_zv(GEN x, long n) as Flx_to_Flv.

GEN Flv_to_Flx(GEN x, long sv) converts from vector (coefficient array) to (normalized) polynomial in variable v.

GEN zv_to_zx(GEN x, long n) as Flv_to_Flx.

GEN Flm_to_FlxV(GEN x, long sv) converts the columns of Flm x to an array of Flx in the variable v (repeated calls to Flv_to_Flx).

GEN zm_to_zxV(GEN x, long n) as Flm_to_FlxV.

GEN Flm_to_FlxX(GEN x, long sw, long sv) same as Flm_to_FlxV(x,sv) but returns the result as a (normalized) polynomial in variable w.
GEN FlxV_to_Flm(GEN v, long n) reverse Flm_to_FlxV, to obtain an Flm with n rows (repeated calls to Flx_to_Flv).

GEN FlxX_to_Flm(GEN v, long n) reverse Flm_to_FlxX, to obtain an Flm with n rows (repeated calls to Flx_to_Flv).

GEN FlxX_to_FlxC(GEN B, long n, long sv) see RgX_to_RgV. The coefficients of B are assumed to be in the variable v.

GEN FlxV_to_FlxM(GEN V, long n, long sv) see RgXV_to_RgM. The coefficients of V[i] are assumed to be in the variable v.

GEN Fly_to_FlxY(GEN a, long sv) convert coefficients of a to constant Flx in variable v.

7.2.27.5 Miscellaneous involving F2x.

GEN F2x_to_F2v(GEN x, long n) converts from F2x to F2v with n components (assumed larger than the number of coefficients of x).

GEN F2xC_to_ZXC(GEN x), converts a vector of F2x to a column vector of polynomials with t_INT coefficients (repeated calls to F2x_to_ZX).

GEN F2xV_to_F2m(GEN v, long n) F2x_to_F2v to each polynomial to get an F2m with n rows.

7.3 Higher arithmetic over Z: primes, factorization.

7.3.1 Pure powers.

long Z_issquare(GEN n) returns 1 if the t_INT n is a square, and 0 otherwise. This is tested first modulo small prime powers, then sqrtremi is called.

long Z_issquareall(GEN n, GEN *sqrtn) as Z_issquare. If n is indeed a square, set sqrtn to its integer square root. Uses a fast congruence test mod 64 × 63 × 65 × 11 before computing an integer square root.

long Z_ispow2(GEN x) returns 1 if the t_INT x is a power of 2, and 0 otherwise.

long uissquare(ulong n) as Z_issquare, for an ulong operand n.

long uissquareall(ulong n, ulong *sqrtn) as Z_issquareall, for an ulong operand n.

ulong usqrt(ulong a) returns the floor of the square root of a.

ulong usqrtn(ulong a, ulong n) returns the floor of the n-th root of a.

long Z_ispower(GEN x, ulong k) returns 1 if the t_INT n is a k-th power, and 0 otherwise; assume that k > 1.

long Z_ispowerall(GEN x, ulong k, GEN *pt) as Z_ispower. If n is indeed a k-th power, set *pt to its integer k-th root.

long Z_isanypower(GEN x, GEN *ptn) returns the maximal k ≥ 2 such that the t_INT x = n^k is a perfect power, or 0 if no such k exist; in particular ispower(1), ispower(0), ispower(-1) all return 0. If the return value k is not 0 (so that x = n^k) and ptn is not NULL, set *ptn to n.

The following low-level functions are called by Z_isanypower but can be directly useful:
int is_357_power(GEN x, GEN *ptn, ulong *pmask) tests whether the integer \( x > 0 \) is a 3-rd, 5-th or 7-th power. The bits of *mask initially indicate which test is to be performed; bit 0: 3-rd, bit 1: 5-th, bit 2: 7-th (e.g. *pmask = 7 performs all tests). They are updated during the call: if the "i-th power" bit is set to 0 then \( x \) is not a \( k \)-th power. The function returns 0 (not a 3-rd, 5-th or 7-th power), 3 (3-rd power, not a 5-th or 7-th power), 5 (5-th power, not a 7-th power), or 7 (7-th power): if an i-th power bit is initially set to 0, we take it at face value and assume \( x \) is not an i-th power without performing any test. If the return value \( k \) is non-zero, set *ptn to \( n \) such that \( x = n^k \).

int is_pth_power(GEN x, GEN *ptn, forprime_t *T, ulong cutoff) let \( x > 0 \) be an integer, \( \text{cutoff} > 0 \) and \( T \) be an iterator over primes \( \geq 11 \), we look for the smallest prime \( p \) such that \( x = n^p \) (advancing \( T \) as we go along). The 11 is due to the fact that is_357_power and issquare are faster than the generic version for \( p < 11 \).

Fail and return 0 when the existence of \( p \) would imply \( 2^\text{cutoff} > x^{1/p} \), meaning that a possible \( n \) is so small that it should have been found by trial division; for maximal speed, you should start by a round of trial division, but the cut-off may also be set to 1 for a rigorous result without any trial division.

Otherwise returns the smallest suitable prime power \( p^i \) and set *ptn to the \( p^i \)-th root of \( x \) (which is now not a \( p \)-th power). We may immediately recall the function with the same parameters after setting \( x = *ptn \): it will start at the next prime.

### 7.3.2 Factorization.

GEN Z_factor(GEN n) factors the t_INT \( n \). The “primes” in the factorization are actually strong pseudoprimes.

GEN absi_factor(GEN n) returns Z_factor(absi(n)).

long Z_issmooth(GEN n, ulong lim) returns 1 if all the prime factors of the t_INT \( n \) are less or equal to \( \text{lim} \).

GEN Z_issmooth_fact(GEN n, ulong lim) returns NULL if a prime factor of the t_INT \( n \) is \( > \text{lim} \), and returns the factorization of \( n \) otherwise, as a t_MAT with t_VECSMALL columns (word-size primes and exponents). Neither memory-clean nor suitable for gerepileupto.

GEN Z_factor_until(GEN n, GEN lim) as Z_factor, but stop the factorization process as soon as the unfactored part is smaller than \( \text{lim} \). The resulting factorization matrix only contains the factors found. No other assumptions can be made on the remaining factors.

GEN Z_factor_limit(GEN n, ulong lim) trial divide \( n \) by all primes \( p < \text{lim} \) in the precomputed list of prime numbers and return the corresponding factorization matrix. In this case, the last “prime” divisor in the first column of the factorization matrix may well be a proven composite.

If \( \text{lim} = 0 \), the effect is the same as setting \( \text{lim} = \text{maxprime}() + 1 \): use all precomputed primes.

GEN absi_factor_limit(GEN n, ulong all) returns Z_factor_limit(absi(n)).

GEN boundfact(GEN x, ulong lim) as Z_factor_limit, applying to t_INT or t_FRAC inputs.

GEN Z_smoothen(GEN n, GEN L, GEN *pP, GEN *pE) given a t_VECSMALL \( L \) containing a list of small primes and a t_INT \( n \), trial divide \( n \) by the elements of \( L \) and return the cofactor. Return NULL if the cofactor is \( \pm 1 \). *P and *E contain the list of prime divisors found and their exponents, as t_VECSMALLs. Neither memory-clean, nor suitable for gerepileupto.

128
GEN Z_factor_listP(GEN N, GEN L) given a t_INT N, a vector or primes L containing all prime divisors of N (and possibly others). Return factor(N). Neither memory-clean, nor suitable for gerepileupto.

GEN factor_pn_1(GEN p, ulong n) returns the factorization of $p^n - 1$, where p is prime and n is a positive integer.

GEN factor_pn_1_limit(GEN p, ulong n, ulong B) returns a partial factorization of $p^n - 1$, where p is prime and n is a positive integer. Don’t actively search for prime divisors $p > B$, but we may find still find some due to Aurifeuillian factorizations. Any entry $> B^2$ in the output factorization matrix is a priori not a prime (but may well be).

GEN factor_Aurifeuille_prime(GEN p, long n) an Aurifeuillian factor of $\phi_n(p)$, assuming p prime and an Aurifeuillian factor exists ($p^{\phi_n} \equiv 1 \pmod{\mathbb{Z}}$).

GEN factor_Aurifeuille(GEN a, long d) an Aurifeuillian factor of $\phi_n(a)$, assuming a is a non-zero integer and $n > 2$. Returns 1 if no Aurifeuillian factor exists.

GEN odd_prime_divisors(GEN a) t_VEC of all prime divisors of the t_INT a.

GEN factoru(ulong n), returns the factorization of n. The result is a 2-component vector $[P, E]$, where P and E are t_VECSMALL containing the prime divisors of n, and the $v_p(n)$.

GEN factoru_pow(ulong n), returns the factorization of n. The result is a 3-component vector $[P, E, C]$, where P, E and C are t_VECSMALL containing the prime divisors of n, the $v_p(n)$ and the $p^{v_p(n)}$.

ulong tridiv_bound(GEN n) returns the trial division bound used by Z_factor(n).

7.3.3 Checks associated to arithmetic functions.

Arithmetic functions accept arguments of the following kind: a plain positive integer N (t_INT), the factorization fa of a positive integer (a t_MAT with two columns containing respectively primes and exponents), or a vector $[N, fa]$. A few functions accept non-zero integers (e.g. omega), and some others arbitrary integers (e.g. factorint, ...).

int is_Z_factorpos(GEN f) returns 1 if f looks like the factorization of a positive integer, and 0 otherwise. Useful for sanity checks but not 100% foolproof. Specifically, this routine checks that f is a two-column matrix all of whose entries are positive integers. It does not check that entries in the first column (“primes”) are prime, or even pairwise coprime, nor that they are strictly increasing.

int is_Z_factornon0(GEN f) returns 1 if f looks like the factorization of a non-zero integer, and 0 otherwise. Useful for sanity checks but not 100% foolproof, analogous to is_Z_factorpos. (Entries in the first column need only be non-zero integers.)

int is_Z_factor(GEN f) returns 1 if f looks like the factorization of an integer, and 0 otherwise. Useful for sanity checks but not 100% foolproof. Specifically, this routine checks that f is a two-column matrix all of whose entries are integers. Entries in the second column (“exponents”) are all positive. Either it encodes the “factorization” $0^e$, $e > 0$, or entries in the first column (“primes”) are all non-zero.

GEN clean_Z_factor(GEN f) assuming f is the factorization of an integer n, return the factorization of $|n|$, i.e. remove $-1$ from the factorization. Shallow function.

In the following two routines, f is the name of an arithmetic function, and n a supplied argument. They all raise exceptions if n does not correspond to an integer or an integer factorization of the expected shape.
GEN check_arith_pos(GEN n, const char *f) check whether $n$ is associated to the factorization of a positive integer, and return NULL (plain t_INT) or a factorization extracted from $n$ otherwise. May raise an e_DOMAIN ($n \leq 0$) or an e_TYPE exception (other failures).

GEN check_arith_non0(GEN n, const char *f) check whether $n$ is associated to the factorization of a non-0 integer, and return NULL (plain t_INT) or a factorization extracted from $n$ otherwise. May raise an e_TYPE exception.

GEN check_arith_all(GEN n, const char *f) is associated to the factorization of an integer, and return NULL (plain t_INT) or a factorization extracted from $n$ otherwise.

7.3.4 Incremental integer factorization.

Routines associated to the dynamic factorization of an integer $n$, iterating over successive prime divisors. This is useful to implement high-level routines allowed to take shortcuts given enough partial information: e.g. moebius($n$) can be trivially computed if we hit $p$ such that $p^2 \mid n$. For efficiency, trial division by small primes should have already taken place. In any case, the functions below assume that no prime $< 2^{14}$ divides $n$.

GEN ifac_start(GEN n, int moebius) schedules a new factorization attempt for the integer $n$. If moebius is non-zero, the factorization will be aborted as soon as a repeated factor is detected (Moebius mode). The function assumes that $n > 1$ is a composite t_INT whose prime divisors satisfy $p > 2^{14}$ and that one can write to $n$ in place.

This function stores data on the stack, no gerepile call should delete this data until the factorization is complete. Returns partial, a data structure recording the partial factorization state.

int ifac_next(GEN *partial, GEN *p, long *e) deletes a primary factor $p^e$ from partial and sets $p$ (prime) and $e$ (exponent), and normally returns 1. Whatever remains in the partial structure is now coprime to $p$.

Returns 0 if all primary factors have been used already, so we are done with the factorization. In this case $p$ is set to NULL. If we ran in Moebius mode and the factorization was in fact aborted, we have $e = 1$, otherwise $e = 0$.

int ifac_read(GEN part, GEN *k, long *e) peeks at the next integer to be factored in the list $k^e$, where $k$ is not necessarily prime and can be a perfect power as well, but will be factored by the next call to ifac_next. You can remove this factorization from the schedule by calling:

void ifac_skip(GEN part) removes the next scheduled factorization.

int ifac_isprime(GEN n) given $n$ whose prime divisors are $> 2^{14}$, returns the decision the factoring engine would take about the compositeness of $n$: 0 if $n$ is a proven composite, and 1 if we believe it to be prime; more precisely, $n$ is a proven prime if factor_proven is set, and only a BPSW-pseudoprime otherwise.
7.3.5 Integer core, squarefree factorization.

long Z_issquarefree(GEN n) returns 1 if the t_INT n is square-free, and 0 otherwise.

long Z_isfundamental(GEN x) returns 1 if the t_INT x is a fundamental discriminant, and 0 otherwise.

GEN core(GEN n) unique squarefree integer $d$ dividing $n$ such that $n/d$ is a square. The core of 0 is defined to be 0.

GEN core2(GEN n) return $[d, f]$ with $d$ squarefree and $n = df^2$.

GEN corepartial(GEN n, long lim) as core, using boundfact(n,lim) to partially factor n. The result is not necessarily squarefree, but $p^2 | n$ implies $p > \text{lim}$.

GEN core2partial(GEN n, long lim) as core2, using boundfact(n,lim) to partially factor n. The resulting $d$ is not necessarily squarefree, but $p^2 | n$ implies $p > \text{lim}$.

7.3.6 Primes, primality and compositeness tests.

7.3.6.1 Chebyshev’s $\pi$ function, bounds.

ulong uprimepi(ulong n), returns the number of primes $p \leq n$ (Chebyshev’s $\pi$ function).

double primepi_upper_bound(double x) return a quick upper bound for $\pi(x)$, using Dusart bounds.

GEN gprimepi_upper_bound(GEN x) as primepi_upper_bound, returns a t_REAL.

double primepi_lower_bound(double x) return a quick lower bound for $\pi(x)$, using Dusart bounds.

GEN gprimepi_lower_bound(GEN x) as primepi_lower_bound, returns a t_REAL or gen_0.

7.3.6.2 Primes, primes in intervals.

ulong unextprime(ulong n), returns the smallest prime $\geq n$. Return 0 if it cannot be represented as an ulong ($n$ bigger than $2^{64} - 59$ or $2^{32} - 5$ depending on the word size).

ulong uprecprime(ulong n), returns the largest prime $\leq n$. Return 0 if $n \leq 1$.

ulong uprime(long n) returns the $n$-th prime, assuming it fits in an ulong (overflow error otherwise).

GEN prime(long n) same as utoi(uprime(n)).

GEN primes_zv(long m) returns the first $m$ primes, in a t_VECSMALL.

GEN primes(long m) return the first $m$ primes, as a t_VEC of t_INTs.

GEN primes_interval(GEN a, GEN b) return the primes in the interval $[a, b]$, as a t_VEC of t_INTs.

GEN primes_interval_zv(ulong a, ulong b) return the primes in the interval $[a, b]$, as a t_VECSMALL of ulongss.

GEN primes_upto_zv(ulong b) return the primes in the interval $[2, b]$, as a t_VECSMALL of ulongss.
7.3.6.3 Tests.

int uisprime(ulong p), returns 1 if p is a prime number and 0 otherwise.

int isprime(GEN n), returns 1 if the t_INT n is a (fully proven) prime number and 0 otherwise.

long isprimeAPRCL(GEN n), returns 1 if the t_INT n is a prime number and 0 otherwise, using only the APRCL test — not even trial division or compositeness tests. The workhorse isprime should be faster on average, especially if non-primes are included!

long BPSW_psp(GEN n), returns 1 if the t_INT n is a Baillie-Pomerance-Selfridge-Wagstaff pseudoprime, and 0 otherwise (proven composite).

int BPSW_isprime(GEN x) assuming x is a BPSW-pseudoprime, rigorously prove its primality. The function isprime is currently implemented as

    BPSW_psp(x) && BPSW_isprime(x)

long millerrabin(GEN n, long k) performs k strong Rabin-Miller compositeness tests on the t_INT n, using k random bases. This function also caches square roots of −1 that are encountered during the successive tests and stops as soon as three distinct square roots have been produced; we have in principle factored n at this point, but unfortunately, there is currently no way for the factoring machinery to become aware of it. (It is highly implausible that hard to find factors would be exhibited in this way, though.) This should be slower than BPSW_psp for k ≥ 4 and we would expect it to be less reliable.

7.3.7 Iterators over primes.

int forprime_init(forprime_t *T, GEN a, GEN b) initialize an iterator T over primes in [a, b]; over primes ≥ a if b = NULL. Return 0 if the range is known to be empty from the start (as if b < a or b < 0), and return 1 otherwise.

GEN forprime_next(forprime_t *T) returns the next prime in the range, assuming that T was initialized by forprime_init.

int u_forprime_init(forprime_t *T, ulong a, ulong b)
ulong u_forprime_next(forprime_t *T)

void u_forprime_restrict(forprime_t *T, ulong c) let T an iterator over primes initialized via u_forprime_init(kT, a, b), possibly followed by a number of calls to u_forprime_next, and a ≤ c ≤ b. Restrict the range of primes considered to [a, c].

int u_forprime_arith_init(forprime_t *T, ulong a,ulong b, ulong c,ulong q) initialize an iterator over primes in [a, b], congruent to c modulo q. Assume 0 ≤ c < q and (c,q) = 1. Subsequent calls to u_forprime_next will only return primes congruent to c modulo q.
7.4 Integral, rational and generic linear algebra.

7.4.1 ZC / ZV, ZM. A ZV (resp. a ZM, resp. a ZX) is a t_VEC or t_COL (resp. t_MAT, resp. t_POL) with t_INT coefficients.

7.4.1.1 ZC / ZV.

void RgV_check_ZV(GEN x, const char *s) Assuming x is a t_VEC or t_COL raise an error if it is not a ZV (s should point to the name of the caller).

int RgV_is_ZV(GEN x) Assuming x is a t_VEC or t_COL return 1 if it is a ZV, and 0 otherwise.

int RgV_is_QV(GEN P) return 1 if the RgV P has only t_INT and t_FRAC coefficients, and 0 otherwise.

int ZV_equal0(GEN x) returns 1 if all entries of the ZV x are zero, and 0 otherwise.

int ZV_cmp(GEN x, GEN y) compare two ZV, which we assume have the same length (lexicographic order, comparing absolute values).

int ZV_abscmp(GEN x, GEN y) compare two ZV, which we assume have the same length (lexicographic order).

int ZV_equal(GEN x, GEN y) returns 1 if the two ZV are equal and 0 otherwise. A t_COL and a t_VEC with the same entries are declared equal.

GEN ZC_add(GEN x, GEN y) adds x and y.

GEN ZC_sub(GEN x, GEN y) subtracts x and y.

GEN ZC_Z_add(GEN x, GEN y) adds y to x[1].

GEN ZC_Z_sub(GEN x, GEN y) subtracts y to x[1].

GEN ZC_copy(GEN x) returns a (t_COL) copy of x.

GEN ZC_neg(GEN x) returns −x as a t_COL.

void ZV_neg_inplace(GEN x) negates the ZV x in place, by replacing each component by its opposite (the type of x remains the same, t_COL or t_COL). If you want to save even more memory by avoiding the implicit component copies, use ZV_togglesign.

void ZV_togglesign(GEN x) negates x in place, by toggling the sign of its integer components. Universal constants gen_1, gen_m1, gen_2 and gen_m2 are handled specially and will not be corrupted. (We use togglesign_safe.)

GEN ZC_Z_mul(GEN x, GEN y) multiplies the ZC or ZV x (which can be a column or row vector) by the t_INT y, returning a ZC.

GEN ZC_Z_divexact(GEN x, GEN y) returns x/y assuming all divisions are exact.

GEN ZV_dotproduct(GEN x, GEN y) as RgV_dotproduct assuming x and y have t_INT entries.

GEN ZV_dotsquare(GEN x) as RgV_dotsquare assuming x has t_INT entries.

GEN ZC_lincomb(GEN u, GEN v, GEN x, GEN y) returns ux + vy, where u, v are t_INT and x, y are ZC or ZV. Return a ZC.

void ZC_lincomb1_inplace(GEN X, GEN Y, GEN v) sets X ← X + vY, where v is a t_INT and X, Y are ZC or ZV. (The result has the type of X.) Memory efficient (e.g. no-op if v = 0), but not gerepile-safe.

133
GEN ZC_ZV_mul(GEN x, GEN y, GEN p) multiplies the ZC x (seen as a column vector) by the ZV y (seen as a row vector, assumed to have compatible dimensions).

GEN ZV_content(GEN x) returns the GCD of all the components of x.

GEN ZV_gcdext(GEN A) given a vector of n integers A, returns [d, U], where d is the content of A and U is a matrix in GL_n(Z) such that AU = [D, 0, ..., 0].

GEN ZV_prod(GEN x) returns the product of all the components of x (1 for the empty vector).

GEN ZV_sum(GEN x) returns the sum of all the components of x (0 for the empty vector).

long ZV_max_lg(GEN x) returns the effective length of the longest entry in x.

int ZV_dvd(GEN x, GEN y) assuming x, y are two ZVs of the same length, return 1 if y[i] divides x[i] for all i and 0 otherwise. Error if one of the y[i] is 0.

GEN ZV_sort(GEN L) sort the ZV L. Returns a vector with the same type as L.

GEN ZV_sort_uniq(GEN L) sort the ZV L, removing duplicate entries. Returns a vector with the same type as L.

long ZV_search(GEN L, GEN y) look for the t_INT y in the sorted ZV L. Return an index i such that L[i] = y, and 0 otherwise.

GEN ZV_indexsort(GEN L) returns the permutation which, applied to the ZV L, would sort the vector. The result is a t_VECSMALL.

GEN ZV_union_shallow(GEN x, GEN y) given two sorted ZV (as per ZV_sort, returns the union of x and y. Shallow function. In case two entries are equal in x and y, include the one from x.

7.4.1.2 ZM.

void RgM_check_ZM(GEN A, const char *s) Assuming x is a t_MAT raise an error if it is not a ZM (s should point to the name of the caller).

GEN ZM_copy(GEN x) returns a copy of x.

int ZM_equal(GEN A, GEN B) returns 1 if the two ZM are equal and 0 otherwise.

GEN ZM_add(GEN x, GEN y) returns x + y (assumed to have compatible dimensions).

GEN ZM_sub(GEN x, GEN y) returns x − y (assumed to have compatible dimensions).

GEN ZM_neg(GEN x) returns −x.

void ZM_togglesign(GEN x) negates x in place, by toggling the sign of its integer components. Universal constants gen1, gen_m1, gen_2 and gen_m2 are handled specially and will not be corrupted. (We use togglesign_safe.)

GEN ZM_mul(GEN x, GEN y) multiplies x and y (assumed to have compatible dimensions).

GEN ZM_Z_mul(GEN x, GEN y) multiplies the ZM x by the t_INT y.

GEN ZM_ZC_mul(GEN x, GEN y) multiplies the ZM x by the ZC y (seen as a column vector, assumed to have compatible dimensions).

GEN ZM_multosym(GEN x, GEN y)
GEN ZM_transmultosym(GEN x, GEN y)
GEN ZMrow_ZC_mul(GEN x, GEN y, long i) multiplies the i-th row of ZM x by the ZC y (seen as a column vector, assumed to have compatible dimensions). Assumes that x is non-empty and $0 < i < \text{lg}(x[1])$.

GEN ZV_ZM_mul(GEN x, GEN y) multiplies the ZV x by the ZM y. Returns a t_VEC.

GEN ZM_Z_divexact(GEN x, GEN y) returns $x/y$ assuming all divisions are exact.

GEN ZM_pow(GEN x, GEN n) returns $x^n$, assuming x is a square ZM and $n \geq 0$.

GEN ZM_powu(GEN x, ulong n) returns $x^n$, assuming x is a square ZM and $n \geq 0$.

GEN ZM_det(GEN M) if M is a ZM, returns the determinant of M. This is the function underlying matdet whenever M is a ZM.

GEN ZM_dettmult(GEN M) if M is a ZM, returns a multiple of the determinant of the lattice generated by its columns. This is the function underlying detint.

GEN ZM_supnorm(GEN x) return the sup norm of the ZM x.

GEN ZM_charpoly(GEN M) returns the characteristic polynomial (in variable 0) of the ZM M.

GEN ZM_imagecompl(GEN x) returns matimagecompl(x).

long ZM_rank(GEN x) returns matrank(x).

GEN ZM_indexrank(GEN x) returns matindexrank(x).

GEN ZM_indeximage(GEN x) returns gel(ZM_indexrank(x), 2).

long ZM_max_lg(GEN x) returns gel(ZM_indexrank(x), 2).

GEN ZM_inv(GEN M, GEN d) if M is a ZM and d is a t_INT such that $M' := dM^{-1}$ is integral, return $M'$. It is allowed to set d = NULL, in which case, the determinant of M is used instead.

GEN QM_inv(GEN M, GEN d) as above, with M a QM. We still assume that $M'$ has integer coefficients.

GEN ZM_det_triangular(GEN x) returns the product of the diagonal entries of x (its determinant if it is indeed triangular).

int ZM_isidentity(GEN x) return 1 is the ZM x is the identity matrix, and 0 otherwise.

int ZM_ishnf(GEN x) return 1 if x is in HNF form, i.e. is upper triangular with positive diagonal coefficients, and for $j > i$, $x_{i,i} > x_{i,j} \geq 0$.

7.4.2 zv, zm.

GEN zv_neg(GEN x) return $-x$. No check for overflow is done, which occurs in the fringe case where an entry is equal to $2^\text{BITS}_\text{IN}_\text{LONG}-1$.

GEN zv_neg_inplace(GEN x) negates x in place and return it. No check for overflow is done, which occurs in the fringe case where an entry is equal to $2^\text{BITS}_\text{IN}_\text{LONG}-1$.

GEN zm_zc_mul(GEN x, GEN y)

GEN zm_mul(GEN x, GEN y)

GEN zv_z_mul(GEN x, long n) return $nx$. No check for overflow is done.

long zv_content(GEN x) returns the gcd of the entries of x.

long zv_dotproduct(GEN x, GEN y)
long zv_prod(GEN x) returns the product of all the components of x (assumes no overflow occurs).

GEN zv_prod_Z(GEN x) returns the product of all the components of x; consider all x[i] as ulongs.

long zv_sum(GEN x) returns the sum of all the components of x (assumes no overflow occurs).

int zv_cmp0(GEN x) returns 1 if all entries of the zv x are 0, and 0 otherwise.

int zv_equal(GEN x, GEN y) returns 1 if the two zv are equal and 0 otherwise.

int zv_equal0(GEN x) returns 1 if all entries are 0, and return 0 otherwise.

long zv_search(GEN L, long y) look for y in the sorted zv L. Return an index i such that L[i] = y, and 0 otherwise.

GEN zv_copy(GEN x) as Flv_copy.

GEN zm_transpose(GEN x) as Flm_transpose.

GEN zm_copy(GEN x) as Flm_copy.

GEN zero_zm(long m, long n) as zero_Flm.

GEN zero_zv(long n) as zero_Flv.

GEN row_zm(GEN A, long x0) as row_Flm.

int zvV_equal(GEN x, GEN y) returns 1 if the two zvV (vectors of zv) are equal and 0 otherwise.

7.4.3 ZMV / zmV (vectors of ZM/zm).

int RgV_is_ZMV(GEN x) Assuming x is a t_VEC or t_COL return 1 if its components are ZM, and 0 otherwise.

GEN ZMV_to_zmV(GEN z)

GEN zmV_to_ZMV(GEN z)

7.4.4 RgC / RgV, RgM.

RgC and RgV routines assume the inputs are VEC or COL of the same dimension. RgM assume the inputs are MAT of compatible dimensions.

7.4.4.1 Matrix arithmetic.

void RgM_dimensions(GEN)x, long *m, long *n sets m, resp. n, to the number of rows, resp. columns of the t_MAT x.

GEN RgC_add(GEN x, GEN y) returns x + y as a t_COL.

GEN RgC_neg(GEN x) returns −x as a t_COL.

GEN RgC_sub(GEN x, GEN y) returns x − y as a t_COL.

GEN RgV_add(GEN x, GEN y) returns x + y as a t_VEC.

GEN RgV_neg(GEN x) returns −x as a t_VEC.

GEN RgV_sub(GEN x, GEN y) returns x − y as a t_VEC.

GEN RgM_add(GEN x, GEN y) return x + y.

GEN RgM_neg(GEN x) returns −x.

136
GEN RgM_sub(GEN x, GEN y) returns $x - y$.

GEN RgM_Rg_add(GEN x, GEN y) assuming $x$ is a square matrix and $y$ a scalar, returns the square matrix $x + y \cdot \text{Id}$.

GEN RgM_Rg_add_shallow(GEN x, GEN y) as RgM_Rg_add with much fewer copies. Not suitable for gerepileupto.

GEN RgM_Rg_sub(GEN x, GEN y) assuming $x$ is a square matrix and $y$ a scalar, returns the square matrix $x - y \cdot \text{Id}$.

GEN RgM_Rg_sub_shallow(GEN x, GEN y) as RgM_Rg_sub with much fewer copies. Not suitable for gerepileupto.

GEN RgC_Rg_add(GEN x, GEN y) assuming $x$ is a non-empty column vector and $y$ a scalar, returns $[x_1 + y, x_2, \ldots, x_n]$.

GEN RgC_Rg_div(GEN x, GEN y)

GEN RgC_Rg_mul(GEN x, GEN y)

GEN RgV_Rg_mul(GEN x, GEN y)

GEN RgM_Rg_mul(GEN x, GEN y) returns $x \times y$ ($y$ treated as a scalar).

GEN RgM_RgC_mul(GEN x, GEN y) returns $x \times y$.

GEN RgM_transmul(GEN x, GEN y) returns $x^\top \times y$.

GEN RgM_multosym(GEN x, GEN y) returns $x \times y$, assuming the result is a symmetric matrix (about twice faster than a generic matrix multiplication).

GEN RgM_transmultosym(GEN x, GEN y) returns $x^\top \times y$, assuming the result is a symmetric matrix (about twice faster than a generic matrix multiplication).

GEN RgMrow_RgC_mul(GEN x, GEN y, long i) multiplies the $i$-th row of RgM x by the RgC y (seen as a column vector, assumed to have compatible dimensions). Assumes that x is non-empty and $0 < i < \lg(x[1])$.

GEN RgM_mulreal(GEN x, GEN y) returns the real part of $x \times y$ (whose entries are t_INT, t_FRAC, t_REAL or t_COMPLEX).

GEN RgC_RgV_mul(GEN x, GEN y) returns $x \times y$ (the square matrix $(x_iy_j)$).

The following two functions are not well defined in general and only provided for convenience in specific cases:

GEN RgC_RgM_mul(GEN x, GEN y) returns $x \times y[1,\ldots,1]$ if $y$ is a row matrix $1 \times n$, error otherwise.

GEN RgM_RgV_mul(GEN x, GEN y) returns $x \times y[1,\ldots,1]$ if $y$ is a column matrix $n \times 1$, error otherwise.

GEN RgM_powers(GEN x, long n) returns $[x^0, \ldots, x^n]$ as a t_VEC of RgMs.
GEN RgV_sum(GEN v) sum of the entries of v

GEN RgV_sumpart(GEN v, long n) returns the sum $v[1] + \ldots + v[n]$ (assumes that $\log(v) > n$).

GEN RgV_sumpart2(GEN v, long m, long n) returns the sum $v[m] + \ldots + v[n]$ (assumes that $\log(v) > n$ and $m > 0$). Returns gen_0 when $m > n$.

GEN RgV_dotproduct(GEN x, GEN y) returns the scalar product of $x$ and $y$

GEN RgV_dotsquare(GEN x) returns the scalar product of $x$ with itself.

GEN gram_matrix(GEN v) returns the Gram matrix ($v_i \cdot v_j$) associated to the entries of $v$ (matrix, or vector of vectors).

GEN RgV_polint(GEN X, GEN Y, long v) $X$ and $Y$ being two vectors of the same length, returns the polynomial $T$ in variable $v$ such that $T(X[i]) = Y[i]$ for all $i$. The special case $X = \text{NULL}$ corresponds to $X = [1, 2, \ldots, n]$, where $n$ is the length of $Y$.

7.4.4.2 Special shapes

The following routines check whether matrices or vectors have a special shape, using gequal1 and gequal0 to test components. (This makes a difference when components are inexact.)

int RgV_isscalar(GEN x) return 1 if all the entries of $x$ are 0 (as per gequal0), except possibly the first one. The name comes from vectors expressing polynomials on the standard basis $1, T, \ldots, T^{n-1}$, or on $\text{nf.zk}$ (whose first element is 1).

int QV_isscalar(GEN x) as RgV_isscalar, assuming $x$ is a QV ($\text{t_INT}$ and $\text{t_FRAC}$ entries only).

int ZV_isscalar(GEN x) as RgV_isscalar, assuming $x$ is a ZV ($\text{t_INT}$ entries only).

int RgM_isscalar(GEN x, GEN s) return 1 if $x$ is the scalar matrix equal to $s$ times the identity, and 0 otherwise. If $s$ is NULL, test whether $x$ is an arbitrary scalar matrix.

int RgM_isidentity(GEN x) return 1 is the $\text{t_MAT}$ $x$ is the identity matrix, and 0 otherwise.

int RgM_isdiagonal(GEN x) return 1 is the $\text{t_MAT}$ $x$ is a diagonal matrix, and 0 otherwise.

int RgM_is_ZM(GEN x) return 1 is the $\text{t_MAT}$ $x$ has only $\text{t_INT}$ coefficients, and 0 otherwise.

long RgV_isin(GEN v, GEN x) return the first index $i$ such that $v[i] = x$ if it exists, and 0 otherwise. Naive search in linear time, does not assume that $v$ is sorted.

GEN RgM_diagonal(GEN m) returns the diagonal of $m$ as a $\text{t_VEC}$.

GEN RgM_diagonal_shallow(GEN m) shallow version of RgM_diagonal

7.4.4.3 Conversion to floating point entries

GEN RgC_gtofp(GEN x, GEN prec) returns the $\text{t_COL}$ obtained by applying gtofp(gel(x,i), prec) to all coefficients of $x$.

GEN RgC_gtomp(GEN x, long prec) returns the $\text{t_COL}$ obtained by applying gtomp(gel(x,i), prec) to all coefficients of $x$.

GEN RgC_fpnorml2(GEN x, long prec) returns (a stack-clean variant of) gnorml2( RgC_gtofp(x, prec) )

GEN RgM_gtofp(GEN x, GEN prec) returns the $\text{t_MAT}$ obtained by applying gtofp(gel(x,i), prec) to all coefficients of $x$.

138
GEN RgM_gtomp(GEN x, long prec) returns the t_MAT obtained by applying gtomp(gel(x,i), prec) to all coefficients of x.

GEN RgM_fpnorml2(GEN x, long prec) returns (a stack-clean variant of) gnorml2( RgM_gtofp(x, prec) )

### 7.4.4.4 Linear algebra, linear systems.

GEN RgM_inv(GEN a) returns a left inverse of a (which needs not be square), or NULL if this turns out to be impossible. The latter happens when the matrix does not have maximal rank (or when rounding errors make it appear so).

GEN RgM_inv_upper(GEN a) as RgM_inv, assuming that a is a non-empty invertible upper triangular matrix, hence a little faster.

GEN RgM_RgC_invimage(GEN A, GEN B) returns a t_COL X such that AX = B if one such exists, and NULL otherwise.

GEN RgM_invimage(GEN A, GEN B) returns a t_MAT X such that AX = B if one such exists, and NULL otherwise.

GEN RgM_Hadamard(GEN a) returns a upper bound for the absolute value of det(a). The bound is a t_INT.

GEN RgM_solve(GEN a, GEN b) returns a−1b where a is a square t_MAT and b is a t_COL or t_MAT. Returns NULL if a−1 cannot be computed, see RgM_inv.

If b = NULL, the matrix a need no longer be square, and we strive to return a left inverse for a (NULL if it does not exist).

GEN RgM_solve_realimag(GEN M, GEN b) M being a t_MAT with r1+r2 rows and r1+2r2 columns, y a t_COL or t_MAT such that the equation Mx = y makes sense, returns x under the following simplifying assumptions: the first r1 rows of M and y are real (the r2 others are complex), and x is real. This is stabler and faster than calling RgM_solve(M, b) over C. In most applications, M approximates the complex embeddings of an integer basis in a number field, and x is actually rational.

GEN split_realimag(GEN x, long r1, long r2) x is a t_COL or t_MAT with r1+r2 rows, whose first r1 rows have real entries (the r2 others are complex). Return an object of the same type as x and r1+2r2 rows, such that the first r1 + r2 rows contain the real part of x, and the r2 following ones contain the imaginary part of the last r2 rows of x. Called by RgM_solve_realimag.

GEN RgM_det_triangular(GEN x) returns the product of the diagonal entries of x (its determinant if it is indeed triangular).

GEN Frobeniusform(GEN V, long n) given the vector V of elementary divisors for M−xId, where M is an n × n square matrix. Returns the Frobenius form of M. Used by matfrobenius.

int RgM_QR_init(GEN x, GEN *pB, GEN *pQ, GEN *pL, long prec) QR-decomposition of a square invertible t_MAT x with real coefficients. Sets *pB to the vector of squared lengths of the xi[i], *pL to the Gram-Schmidt coefficients and *pQ to a vector of successive Householder transforms. If R denotes the transpose of L and Q is the result of applying *pQ to the identity matrix, then x = QR is the QR decomposition of x. Returns 0 is x is not invertible or we hit a precision problem, and 1 otherwise.

int QR_init(GEN x, GEN *pB, GEN *pQ, GEN *pL, long prec) as RgM_QR_init, assuming further that x has t_INT or t-REAL coefficients.
GEN R_from_QR(GEN x, long prec) assuming that x is a square invertible t_MAT with t_INT or t_REAL coefficients, return the upper triangular R from the QR decomposition of x. Not memory clean. If the matrix is not known to have t_INT or t_REAL coefficients, apply RgM_gtomp first.

GEN gaussred_from_QR(GEN x, long prec) assuming that x is a square invertible t_MAT with t_INT or t_REAL coefficients, returns qfgaussred(x ~* x); this is essentially the upper triangular R matrix from the QR decomposition of x, renormalized to accommodate qfgaussred conventions. Not memory clean.

7.4.5 Blackbox linear algebra.

A sparse column zCs v is a t_COL with two components C and E which are t_VECSMALL of the same length, representing $\sum_i E[i] * e_{C[i]}$, where (e_j) is the canonical basis. A sparse matrix (zMs) is a t_VEC of zCs.

FpCs and FpMs are identical to the above, but E[i] is now interpreted as a signed C long integer representing an element of F_p. This is important since p can be so large that $p + E[i]$ would not fit in a C long.

RgCs and RgMs are similar, except that the type of the components of E is now unspecified. Functions handling those later objects must not depend on the type of those components.

It is not possible to derive the space dimension (number of rows) from the above data. Thus most functions take an argument nbrow which is the number of rows of the corresponding column/matrix in dense representation.

GEN zCs_to_ZC(GEN C, long nbrow) convert the sparse vector C to a dense ZC of dimension nbrow.

GEN zMs_to_ZM(GEN M, long nbrow) convert the sparse matrix M to a dense ZM whose columns have dimension nbrow.

GEN FpMs_FpC_mul(GEN M, GEN B, GEN p) multiply the sparse matrix M (over F_p) by the sparse vector B. The result is an FpC, i.e. a dense vector.

GEN zMs_ZC_mul(GEN M, GEN B, GEN p) multiply the sparse matrix M by the sparse vector B (over Z). The result is an ZC, i.e. a dense vector.

GEN FpV_FpMs_mul(GEN B, GEN M, GEN p) multiply the sparse vector B by the sparse matrix M (over F_p). The result is an FpV, i.e. a dense vector.

GEN ZV_zMs_mul(GEN B, GEN M, GEN p) multiply the sparse vector B (over Z) by the sparse matrix M. The result is an ZV, i.e. a dense vector.

void RgMs_structelim(GEN M, long nbrow, GEN A, GEN *p_col, GEN *p_row) M being a RgMs with nbrow rows, A being a list of row indices, Perform structured elimination on M by removing some rows and columns until the number of effectively present rows is equal to the number of columns. the result is stored in two t_VECSMALLs, *p_col and *p_row: *p_col is a map from the new columns indices to the old one. *p_row is a map from the old rows indices to the new one (0 if removed).

GEN FpMs_leftkernel_elt(GEN M, long nbrow, GEN p) M being a sparse matrix over F_p, return a non-zero kbdFpV X such that XM components are almost all 0.

GEN FpMs_FpCs_solve(GEN M, GEN B, long nbrow, GEN p) solve the equation MX = B, where M is a sparse matrix and B is a sparse vector, both over F_p. Return either a solution as a t_COL

140
(dense vector), the index of a column which is linearly dependent from the others as a \texttt{t_VECSMALL} with a single component, or \texttt{NULL} (can happen if $B$ is not in the image of $M$).

\texttt{GEN FpMs\_FpCs\_solve\_safe(GEN M, GEN B, long nbrow, GEN p)} as above, but in the event that $p$ is not a prime and an impossible division occurs, return \texttt{NULL}.

\texttt{GEN ZpMs\_ZpCs\_solve(GEN M, GEN B, long nbrow, GEN p, long e)} solve the equation $MX = B$, where $M$ is a sparse matrix and $B$ is a sparse vector, both over $\mathbb{Z}/p^e\mathbb{Z}$. Return either a solution as a \texttt{t\_COL} (dense vector), or the index of a column which is linearly dependent from the others as a \texttt{t_VECSMALL} with a single component.

\texttt{GEN gen\_FpM\_Wiedemann(void *E, GEN (*f)(void*, GEN), GEN B, GEN p)} solve the equation $f(X) = B$ over $\mathbb{F}_p$, where $B$ is a $\mathbb{F}_pV$, and $f$ is a blackbox endomorphism, where $f(E, X)$ computes the value of $f$ at the (dense) column vector $X$. Returns either a solution \texttt{t\_COL}, or a kernel vector as a \texttt{t\_VEC}.

\texttt{GEN gen\_ZpM\_Dixon(void *E, GEN (*f)(void*, GEN), GEN B, GEN p, long e)} solve equation $f(X) = B$ over $\mathbb{Z}/p^e\mathbb{Z}$, where $B$ is a $\mathbb{Z}V$, and $f$ is a blackbox endomorphism, where $f(E, X)$ computes the value of $f$ at the (dense) column vector $X$. Returns either a solution \texttt{t\_COL}, or a kernel vector as a \texttt{t\_VEC}.

7.4.6 Obsolete functions.

The functions in this section are kept for backward compatibility only and will eventually disappear.

\texttt{GEN image2(GEN x)} compute the image of $x$ using a very slow algorithm. Use \texttt{image} instead.

7.5 Integral, rational and generic polynomial arithmetic.

7.5.1 ZX.

\texttt{void RgX\_check\_ZX(GEN x, const char *s)} Assuming $x$ is a \texttt{t\_POL} raise an error if it is not a ZX ($s$ should point to the name of the caller).

\texttt{GEN ZX\_copy(GEN x, GEN p)} returns a copy of $x$.

\texttt{long ZX\_max\_lg(GEN x)} returns the effective length of the longest component in $x$.

\texttt{GEN scalar\_ZX(GEN x, long v)} returns the constant ZX in variable $v$ equal to the \texttt{t\_INT} $x$.

\texttt{GEN scalar\_ZX\_shallow(GEN x, long v)} returns the constant ZX in variable $v$ equal to the \texttt{t\_INT} $x$. Shallow function not suitable for gerepile and friends.

\texttt{GEN ZX\_renormalize(GEN x, long l)} as \texttt{normalizepol}, where $l = \lg(x)$, in place.

\texttt{int ZX\_equal(GEN x, \texttt{\_\_1\_y}) \texttt{\_\_1\_y}} returns 1 if the two ZX have the same \texttt{degpol} and their coefficients are equal. Variable numbers are not checked.

\texttt{int ZX\_equal1(GEN x)} returns 1 if the ZX is equal to 1 and 0 otherwise.

\texttt{GEN ZX\_add(GEN x, GEN y)} adds $x$ and $y$.

\texttt{GEN ZX\_sub(GEN x, GEN y)} subtracts $x$ and $y$.

\texttt{GEN ZX\_neg(GEN x, GEN p)} returns $-x$. 141
GEN ZX_Z_add(GEN x, GEN y) adds the integer y to the ZX x.

GEN ZX_Z_add_shallow(GEN x, GEN y) shallow version of ZX_Z_add.

GEN ZX_Z_sub(GEN x, GEN y) subtracts the integer y to the ZX x.

GEN Z_ZX_sub(GEN x, GEN y) subtracts the ZX y to the integer x.

GEN ZX_Z_mul(GEN x, GEN y) multiplies the ZX x by the integer y.

GEN ZX_mulu(GEN x, ulong y) multiplies x by the integer y.

GEN ZX_shifti(GEN x, long n) shifts all coefficients of x by n bits, which can be negative.

GEN ZX_Z_divexact(GEN x, GEN y) returns x/y assuming all divisions are exact.

GEN ZX_remi2n(GEN x, long n) reduces all coefficients of x to n bits, using remi2n.

GEN ZX_mul(GEN x, GEN y) multiplies x and y.

GEN ZX_sqr(GEN x, GEN p) returns x^2.

GEN ZX_mulspec(GEN a, GEN b, long na, long nb). Internal routine: a and b are arrays of coefficients representing polynomials \(\sum_{i=0}^{na-1} a[i]X^i\) and \(\sum_{i=0}^{nb-1} b[i]X^i\). Returns their product (as a true GEN).

GEN ZX_sqrspec(GEN a, long na). Internal routine: a is an array of coefficients representing polynomial \(\sum_{i=0}^{na-1} a[i]X^i\). Return its square (as a true GEN).

GEN ZX_rem(GEN x, GEN y) returns the remainder of the Euclidean division of x mod y. Assume that x, y are two ZX and that y is monic.

GEN ZX_mod_Xnm1(GEN T, ulong n) return T modulo \(X^n-1\). Shallow function.

GEN ZX_gcd(GEN x, GEN y) returns a gcd of the ZX x and y. Not memory-clean, but suitable for gerepileupto.

GEN ZX_gcd_all(GEN x, GEN y, GEN *pX). returns a gcd d of x and y. If pX is not NULL, set *pX to a (non-zero) integer multiple of x/d. If x and y are both monic, then d is monic and *pX is exactly x/d. Not memory clean if the gcd is 1 (in that case *pX is set to x).

GEN ZX_content(GEN x) returns the content of the ZX x.

long ZX_val(GEN P) as RgX_val, but assumes P has t_INT coefficients.

long ZX_valrem(GEN P, GEN *z) as RgX_valrem, but assumes P has t_INT coefficients.

GEN ZX_to_monic(GEN q GEN *L) given q a non-zero ZX, returns a monic integral polynomial Q such that Q(x) = Cq(x/L), for some rational C and positive integer L > 0. If L is not NULL, set *L to L; if L = 1, *L is set to gen_1. Not suitable for gerepileupto.

GEN ZX_primitive_to_monic(GEN q, GEN *L) as ZX_to_monic except q is assumed to have trivial content, which avoids recomputing it. The result is suboptimal if q is not primitive (L larger than necessary), but remains correct.

GEN ZX_Z_normalize(GEN q, GEN *L) a restricted version of ZX_primitive_to_monic, where q is a monic ZX of degree > 0. Finds the largest integer L > 0 such that Q(X) := L^{-\deg q}(Lx) is integral and return Q; this is not well-defined if q is a monomial, in that case, set L = 1 and Q = q. If L is not NULL, set *L to L.
GEN ZX_Q_normalize(GEN q, GEN *L) a variant of ZX_Z_normalize where \( L > 0 \) is allowed to be rational, the monic \( Q \in \mathbb{Z}[X] \) has possibly smaller coefficients.

GEN ZX_rescale(GEN P, GEN h) returns \( h^{\text{deg}(P)} P(x/h) \). \( P \) is a \( \mathbb{Z} \) and \( h \) is a non-zero integer. Neither memory-clean nor suitable for gerepileupto.

GEN ZX_rescale_lt(GEN P) returns the monic integral polynomial \( h^{\text{deg}(P)-1} P(x/h) \), where \( P \) is a non-zero \( \mathbb{Z} \) and \( h \) is its leading coefficient. Neither memory-clean nor suitable for gerepileupto.

GEN ZX_translate(GEN P, GEN c) assume \( P \) is a \( \mathbb{Z} \) and \( c \) an integer. Returns \( P(X+c) \) (optimized for \( c = \pm 1 \)).

GEN ZX_unscale(GEN P, GEN h) given a \( \mathbb{Z} \) \( P \) and a t_INT \( h \), returns \( P(hx) \). Not memory clean.

GEN ZX_unscale_div(GEN P, GEN h) given a \( \mathbb{Z} \) \( P \) and a t_INT \( h \) such that \( h | P(0) \), returns \( P(hx)/h \). Not memory clean.

GEN ZX_eval1(GEN P) returns the integer \( P(1) \).

GEN ZX_graeffe(GEN p) returns the Graeffe transform of \( p \), i.e. the \( \mathbb{Z} \) \( q \) such that \( p(x)p(-x) = q(x^2) \).

GEN ZX_deriv(GEN x) returns the derivative of \( x \).

GEN ZX_resultant(GEN A, GEN B) returns the resultant of the \( \mathbb{Z} \) \( A \) and \( B \).

GEN ZX_disc(GEN T) returns the discriminant of the \( \mathbb{Z} \) \( T \).

GEN ZX_factor(GEN T) returns the factorization of the primitive part of \( T \) over \( \mathbb{Q}[X] \) (the content is lost).

int ZX_is_squarefree(GEN T) returns 1 if the \( \mathbb{Z} \) \( T \) is squarefree, 0 otherwise.

long ZX_is_irred(GEN T) returns 1 it \( T \) is irreducible, and 0 otherwise.

GEN ZX_squff(GEN T, GEN *E) write \( T \) as a product \( \prod T_i^{e_i} \) with the \( e_1 < e_2 < \cdots \) all distinct and the \( T_i \) pairwise coprime. Return the vector of the \( T_i \), and set *E to the vector of the \( e_i \), as a t_VECsmall.

7.5.2 \( \mathbb{Z}XQ \).

GEN ZXQ_mul(GEN x, GEN y, GEN T) returns \( x \ast y \mod T \), assuming that all inputs are \( \mathbb{Z}X \)s and that \( T \) is monic.

GEN ZXQ_sqr(GEN x, GEN T) returns \( x^2 \mod T \), assuming that all inputs are \( \mathbb{Z}X \)s and that \( T \) is monic.

GEN ZXQ_charpoly(GEN A, GEN T, long v): let \( T \) and \( A \) be \( \mathbb{Z}X \)s, returns the characteristic polynomial of \( \text{Mod}(A, T) \). More generally, \( A \) is allowed to be a \( \mathbb{Q}X \), hence possibly has rational coefficients, assuming the result is a \( \mathbb{Z}X \), i.e. the algebraic number \( \text{Mod}(A, T) \) is integral over \( \mathbb{Z} \).

GEN ZX_ZXY_resultant(GEN A, GEN B) under the assumption that \( A \) in \( \mathbb{Z}[Y] \), \( B \) in \( \mathbb{Q}[Y][X] \), and \( R = \text{Res}_Y(A,B) \in \mathbb{Z}[X] \), returns the resultant \( R \).

GEN ZX_compositum_disjoint(GEN A, GEN B) given two irreducible \( \mathbb{Z}X \) defining linearly disjoint extensions, returns a \( \mathbb{Z}X \) defining their compositum.

GEN ZX_ZXY_rnfequation(GEN A, GEN B, long *lambda), assume \( A \) in \( \mathbb{Z}[Y] \), \( B \) in \( \mathbb{Q}[Y][X] \), and \( R = \text{Res}_Y(A,B) \in \mathbb{Z}[X] \). If \( \lambda = \text{NULL} \), returns \( R \) as in \( \text{ZYZXY}\text{resultant} \). Otherwise, \( \lambda \)
must point to some integer, e.g. 0 which is used as a seed. The function then finds a small \( \lambda \in \mathbb{Z} \) (starting from \(*\lambda\)) such that \( R_\lambda(X) := \text{Res}_Y(A, B(X + \lambda Y)) \) is squarefree, resets \(*\lambda\) to the chosen value and returns \( R_\lambda \).

### 7.5.3 ZXV.

- **GEN ZXV_equal(GEN x,GEN y)** returns 1 if the two vectors of ZX are equal, as per ZX_equal (variables are not checked to be equal) and 0 otherwise.

- **GEN ZXV_Z_mul(GEN x,GEN y)** multiplies the vector of ZX \( x \) by the integer \( y \).

- **GEN ZXV_remi2n(GEN x, long n)** applies ZX_remi2n to all coefficients of \( x \).

- **GEN ZXV_dotproduct(GEN x,GEN y)** as RgV_dotproduct assuming \( x \) and \( y \) have ZX entries.

### 7.5.4 ZXT.

- **GEN ZXT_remi2n(GEN x, long n)** applies ZX_remi2n to all leaves of the tree \( x \).

### 7.5.5 ZZX.

- **void RgX_check_ZXX(GEN x, const char *s)** Assuming \( x \) is a t_POL raise an error if it one of its coefficients is not an integer or a ZX (\( s \) should point to the name of the caller).

- **GEN ZXX_renormalize(GEN x, long l)**, as normalizepol, where \( l = \lg(x) \), in place.

- **long ZXX_max_lg(GEN x)** returns the effective length of the longest component in \( x \); assume all coefficients are t_INT or ZXs.

- **GEN ZXX_Z_divexact(GEN x, GEN y)** returns \( x/y \) assuming all integer divisions are exact.

- **GEN ZXX_to_Kronecker(GEN P, long n)** Assuming \( P(X,Y) \) is a polynomial of degree in \( X \) strictly less than \( n \), returns \( P(X,X^{2^n-1}) \), the Kronecker form of \( P \). Shallow function.

- **GEN ZXX_to_Kronecker_spec(GEN Q, long lQ, long n)** return ZXX_to_Kronecker \((P,n)\), where \( P \) is the polynomial \( \sum_{i=0}^{lQ-1} Q[i]x^i \). To be used when splitting the coefficients of genuine polynomials into blocks. Shallow function.

- **GEN Kronecker_to_ZXX(GEN z, long n, long v)** recover \( P(X,Y) \) from its Kronecker form \( P(X,X^{2^n-1}) \), \( v \) is the variable number corresponding to \( Y \). Shallow function.

- **GEN ZXX_mul_Kronecker(GEN P, GEN Q, long n)** return ZX_mul applied to the Kronecker forms \( P(X,X^{2^n-1}) \) and \( Q(X,X^{2^n-1}) \) of \( P \) and \( Q \). Not memory clean.

### 7.5.6 QX.

- **void RgX_check_QX(GEN x, const char *s)** Assuming \( x \) is a t_POL raise an error if it is not a QX (\( s \) should point to the name of the caller).

- **GEN QX_gcd(GEN x,GEN y)** returns a gcd of the QX \( x \) and \( y \).

- **GEN QX_disc(GEN T)** returns the discriminant of the QX \( T \).

- **GEN QX_factor(GEN T)** as ZX_factor.

- **GEN QX_resultant(GEN A, GEN B)** returns the resultant of the QX \( A \) and \( B \).

- **GEN QX_complex_roots(GEN p, long l)** returns the complex roots of the QX \( p \) at accuracy \( l \), where real roots are returned as t_REALs. More efficient when \( p \) is irreducible and primitive. Special case of cleanroots.
7.5.7 QXQ.

GEN QXQ_norm(GEN A, GEN B) A being a QX and B being a ZX, returns the norm of the algebraic number \(A \mod B\), using a modular algorithm. To ensure that B is a ZX, one may replace it by \(Q\text{\_primpart}(B)\), which of course does not change the norm.

If \(A\) is not a ZX — it has a denominator —, but the result is nevertheless known to be an integer, it is much more efficient to call \(QXQ\text{\_intnorm}\) instead.

GEN QXQ_intnorm(GEN A, GEN B) A being a QX and B being a ZX, returns the norm of the algebraic number \(A \mod B\), assuming that the result is an integer, which is for instance the case is \(A \mod B\) is an algebraic integer, in particular if \(A\) is a ZX. To ensure that \(B\) is a ZX, one may replace it by \(Q\text{\_primpart}(B)\) (which of course does not change the norm).

If the result is not known to be an integer, you must use \(QXQ\text{\_norm}\) instead, which is slower.

GEN QXQ_inv(GEN A, GEN B) returns the inverse of \(A\) modulo \(B\) where \(A\) is a QX and \(B\) is a ZX.

Should you need this for a QX \(B\), just use

\[
QXQ\text{\_inv}(A, Q\text{\_primpart}(B));
\]

But in all cases where modular arithmetic modulo \(B\) is desired, it is much more efficient to replace \(B\) by \(Q\text{\_primpart}(B)\) once and for all.

GEN QXQ_powers(GEN x, long n, GEN T) returns \([x^0, \ldots, x^n]\) as \(\text{RgXQ}\) would, but in a more efficient way when \(x\) has a huge integer denominator (we start by removing that denominator). Meant to be much more efficient to precompute powers of algebraic integers in \(Q[t]/(T)\). The current implementation does not require \(x\) to be a QX: any polynomial to which \(Q\text{\_remove\_denom}\) can be applied is fine.

GEN QXQ_reverse(GEN f, GEN T) as \(\text{RgXQ\_reverse}\), assuming \(f\) is a QX.

GEN QX_ZXQV_eval(GEN f, GEN nV, GEN dV) as \(\text{RgX\_RgXQV\_eval}\), except that \(f\) is assumed to be a QX, \(V\) is given implicitly by a numerator \(nV\) (ZV) and denominator \(dV\) (a positive t\_INT or NULL for trivial denominator). Not memory clean, but suitable for gerepileupto.

GEN QXX_QXQ_eval(GEN P, GEN a, GEN T) \(P(X,Y)\) is a t\_POL with QX coefficients (possibly scalars, i.e. rational numbers, for convenience), \(a\) and \(T\) both QX. Return the QX \(P(X,a \mod T)\). Not memory clean, nor suitable for gerepileupto.

GEN nfgcd(GEN P, GEN Q, GEN T, GEN den) given \(P\) and \(Q\) in \(Z[X,Y]\), \(T\) monic irreducible in \(Z[Y]\), returns the primitive \(d\) in \(Z[X,Y]\) which is a gcd of \(P, Q\) in \(K[X]\), where \(K\) is the number field \(Q[Y]/(T)\). If not NULL, \(\text{den}\) is a multiple of the integral denominator of the (monic) gcd of \(P, Q\) in \(K[X]\).

GEN nfgcd_all(GEN P, GEN Q, GEN T, GEN den, GEN *Pnew) as nfgcd. If \(P\text{\_new}\) is not NULL, set \(*\text{\_new}\) to a non-zero integer multiple of \(P/d\). If \(P\) and \(Q\) are both monic, then \(d\) is monic and \(*\text{\_new}\) is exactly \(P/d\). Not memory clean if the gcd is 1 (in that case \(*\text{\_new}\) is set to \(P\)).
7.5.8 \(\text{zx}\).

\text{GEN} \text{zero_zx}(\text{long} \ sv) \text{ returns a zero } \text{zx} \text{ in variable } v.

\text{GEN} \text{polx_zx}(\text{long} \ sv) \text{ returns the variable } v \text{ as degree 1 Flx}.

\text{GEN} \text{zx_renormalize}(\text{GEN} \ x, \text{long} \ 1), \text{as Flx_renormalize, where } 1 = \log(x), \text{in place}.

\text{GEN} \text{zx_shift}(\text{GEN} \ T, \text{long} \ n) \text{ returns } T \text{ multiplied by } x^n, \text{assuming } n \geq 0.

7.5.9 \(RgX\).

\text{long} \text{RgX_type}(\text{GEN} \ x, \ \text{GEN} *\text{ptp}, \ \text{GEN} *\text{ptpol}, \ \text{long} *\text{ptprec}) \text{ returns the “natural” base ring over which the polynomial } x \text{ is defined. Raise an error if it detects consistency problems in modular objects: incompatible rings (e.g. } F_p \text{ and } F_q \text{ for primes } p \neq q, F_p[X]/(T) \text{ and } F_q[X]/(U) \text{ for } T \neq U). \text{ Minor discrepancies are supported if they make general sense (e.g. } F_p \text{ and } F_p[k], \text{but not } F_p \text{ and } \mathbb{Q}_p); \text{t_FFELT and t_POLMOD of t_INTMODs are considered inconsistent, even if they define the same field: if you need to use simultaneously these different finite field implementations, multiply the polynomial by a t_FFELT equal to 1 first.}

\begin{itemize}
  \item 0: none of the others (presumably multivariate, possibly inconsistent).
  \item t_INT: defined over \(\mathbb{Q}\) (not necessarily \(\mathbb{Z}\)).
  \item t_INTEMD: defined over \(\mathbb{Z}/p\mathbb{Z}\), where \*ptp is set to \(p\). It is not checked whether \(p\) is prime.
  \item t_COMPLEX: defined over \(\mathbb{C}\) (at least one t_COMPLEX with at least one inexact floating point \text{t_REAL component}). Set \*ptprec to the minimal accuracy (as per \text{precision}) of inexact components.
  \item t_REAL: defined over \(\mathbb{R}\) (at least one inexact floating point \text{t_REAL component}). Set \*ptprec to the minimal accuracy (as per \text{precision}) of inexact components.
  \item t_PADIC: defined over \(\mathbb{Q}_p\), where \*ptp is set to \(p\) and \*ptprec to the \(p\)-adic accuracy.
  \item t_FFELT: defined over a finite field \(F_{p^k}\), where \*ptp is set to the field characteristic \(p\) and \*ptpol is set to a t_FFELT belonging to the field.
  \item other values are composite corresponding to quotients \(R[X]/(T)\), with one primary type \(t1\), describing the form of the quotient, and a secondary type \(t2\), describing \(R\). If \(t\) is the \text{RgX_type}, \(t1\) and \(t2\) are recovered using
\end{itemize}

\text{void RgX_type_decode(long t, long \*t1, long \*t2)}

\begin{itemize}
  \item \(t1\) is one of
    \begin{itemize}
      \item t_POLMOD: at least one t_POLMOD component, set \*ppol to the modulus,
      \item t_QUAD: no t_POLMOD, at least one t_QUAD component, set \*ppol to the modulus (-.pol) of the t_QUAD,
      \item t_COMPLEX: no t_POLMOD or t_QUAD, at least one t_COMPLEX component, set \*ppol to \(y^2 + 1\).
    \end{itemize}
    \text{and the underlying base ring } R \text{ is given by } t2, \text{which is one of t_INT, t_INTEMD (set \*ptp) or t_PADIC (set \*ptp and \*ptprec), with the same meaning as above.}
\end{itemize}

\text{int RgX_type_is_composite(long t) \(t\) as returned by RgX_type, return 1 if } t \text{ is a composite type, and 0 otherwise.}

146
GEN RgX_get_0(GEN x) returns 0 in the base ring over which x is defined, to the proper accuracy (e.g. 0, Mod(0,3), 0(5^10)).

GEN RgX_get_1(GEN x) returns 1 in the base ring over which x is defined, to the proper accuracy (e.g. 0, Mod(0,3)).

int RgX_isscalar(GEN x) return 1 if all the coefficients of x of degree > 0 are 0 (as per gequal0).

int RgX_blocks(GEN P, long n, long m) writes \( P(X) = a_0(X) + X^n a_1(X) + \ldots + X^{m-1} a_{m-1}(X) \), where the \( a_i \) are polynomial of degree at most \( n-1 \) (except possibly for the last one) and returns \([a_0(X), a_1(X), \ldots, a_{m-1}(X)]\). This is a shallow function.

void RgX_even_odd(GEN p, GEN *pe, GEN *po) write \( p(X) = E(X^2) + XO(X^2) \) and set *pe = E, *po = O.

GEN RgX_splitting(GEN P, long k) write \( P(X) = a_0(X^k) + Xa_1(X^k) + \ldots + X^{k-1} a_{k-1}(X^k) \) and return \([a_0(X), a_1(X), \ldots, a_{k-1}(X)]\). This is a shallow function.

GEN RgX_copy(GEN x) returns (a deep copy of) x.

GEN RgX_add(GEN x, GEN y) adds x and y.

GEN RgX_sub(GEN x, GEN y) subtracts x and y.

GEN RgX_neg(GEN x) returns \(-x\).

GEN RgX_Rg_add(GEN y, GEN x) returns \( x + y \).

GEN RgX_Rg_add_shallow(GEN y, GEN x) returns \( x + y \); shallow function.

GEN RgX_Rg_sub(GEN y, GEN x) returns \( x - y \).

GEN RgX_mul(GEN x, GEN y) multiplies the two t_POL (in the same variable) x and y. Uses Karatsuba algorithm.

GEN RgX_mul_normalized(GEN A, long a, GEN B, long b) returns \( (X^a + A)(X^b + B) - X^{a+b} \), where we assume that deg \( A < a \) and deg \( B < b \) are polynomials in the same variable X.

GEN RgX_mulspec(GEN a, GEN b, long na, long nb). Internal routine: a and b are arrays of coefficients representing polynomials \( \sum_{i=0}^{na-1} a[i] X^i \) and \( \sum_{i=0}^{nb-1} b[i] X^i \). Returns their product (as a true GEN).

GEN RgX_mullow(GEN a, GEN b, long n) returns \( ab \) modulo \( X^n \), where a, b are two t_POL in the same variable X and \( n \geq 0 \). Uses Karatsuba algorithm (Mulders, Hanrot-Zimmermann variant).

GEN RgX_sqr(GEN x) squares the t_POL x. Uses Karatsuba algorithm.

GEN RgX_sqrspec(GEN a, long na). Internal routine: a is an array of coefficients representing polynomial \( \sum_{i=0}^{na-1} a[i] X^i \). Return its square (as a true GEN).

GEN RgX_sqrlow(GEN a, long n) returns \( a^2 \) modulo \( X^n \), where a is a t_POL in the variable X and \( n \geq 0 \). Uses Karatsuba algorithm (Mulders, Hanrot-Zimmermann variant).

GEN RgX_divrem(GEN x, GEN y, GEN *r) by default, returns the Euclidean quotient and store the remainder in r. Three special values of r change that behavior • NULL: do not store the remainder, used to implement RgX_div,

• ONLY_REM: return the remainder, used to implement RgX_rem.
• **ONLY_DIVIDES**: return the quotient if the division is exact, and NULL otherwise.

```c
GEN RgX_div(GEN x, GEN y)
GEN RgX_div_by_X_x(GEN A, GEN a, GEN *r) returns the quotient of the RgX A by (X − a), and
sets r to the remainder A(a).

GEN RgX_rem(GEN x, GEN y)
GEN RgX_pseudodivrem(GEN x, GEN y, GEN *ptr) compute a pseudo-quotient q and pseudo-
remainder r such that lc(y)^deg(x)−deg(y)+1 x = qy + r. Return q and set *ptr to r.
GEN RgX_pseudorem(GEN x, GEN y) return the remainder in the pseudo-division of x by y.
long RgX_degree(GEN x, long v) x being a t_POL and v ≥ 0, returns the degree in v of x. Error
if x is not a polynomial in v.
GEN RgXQX_pseudorem(GEN x, GEN y, GEN T) return the remainder in the pseudo-division of x
by y over \( \mathbb{R}[X]/(T) \).
int ZXQX_dvd(GEN x, GEN y, GEN T) let T be a monic irreducible ZX, let x, y be t_POL whose
coefficients are either t_INTs or ZX in the same variable as T. Assume further that the leading
coefficient of y is an integer. Return 1 if y|x in (\mathbb{Z}[Y]/(T))[X], and 0 otherwise.
GEN RgXQX_pseudodivrem(GEN x, GEN y, GEN T, GEN *ptr) compute a pseudo-quotient q and
pseudo-remainder r such that lc(y)^deg(x)−deg(y)+1 x = qy + r in \( \mathbb{R}[X]/(T) \). Return q and set *ptr
to r.
GEN RgX_mulXn(GEN x, long n) returns x ∗ t^n. This may be a t_FRAC if n < 0 and the valuation
of x is not large enough.
GEN RgX_shift(GEN x, long n) returns x ∗ t^n if n ≥ 0, and x/t^n otherwise.
GEN RgX_shift_shallow(GEN x, long n) as RgX_shift, but shallow (coefficients are not copied).
GEN RgX_rotate_shallow(GEN P, long k, long p) returns P ∗ X^k (mod X^p−1), assuming the
degree of P is strictly less than p, and k ≥ 0.
void RgX_shift_inplace_init(long v) v ≥ 0, prepare for a later call to RgX_shift_inplace. Reserves v
words on the stack.
GEN RgX_shift_inplace(GEN x, long v) v ≥ 0, assume that RgX_shift_inplace_init(v) has
been called (reserving v words on the stack), immediately followed by a t_POL x. Return
RgX_shift(x, v) by shifting x in place. To be used as follows
   RgX_shift_inplace_init(v);
   av = avma;
   ...
   x = gerepileupto(av, ...); /* a t_POL */
   return RgX_shift_inplace(x, v);
```

```c
long RgX_valrem(GEN P, GEN *pz) returns the valuation v of the t_POL P with respect to its main
variable X. Check whether coefficients are 0 using gequal0. Set *pz to RgX_shift_shallow(P, −v).
long RgX_val(GEN P) returns the valuation v of the t_POL P with respect to its main variable X.
Check whether coefficients are 0 using gequal0.
long RgX_valrem_inexact(GEN P, GEN *z) as RgX_valrem, using isexactzero instead of
gequal0.
```
GEN RgX_deriv(GEN x) returns the derivative of x with respect to its main variable.

GEN RgX_integ(GEN x) returns the primitive of x vanishing at 0, with respect to its main variable.

GEN RgX_gcd(GEN x, GEN y) returns the GCD of x and y, assumed to be t_POLs in the same variable.

GEN RgX_gcd_simple(GEN x, GEN y) as RgX_gcd using a standard extended Euclidean algorithm. Usually slower than RgX_gcd.

GEN RgX_extgcd(GEN x, GEN y, GEN *u, GEN *v) returns \( d = \text{GCD}(x, y) \), and sets \(*u, *v\) to the Bezout coefficients such that \( *ux + *vy = d \). Uses a generic subresultant algorithm.

GEN RgX_extgcd_simple(GEN x, GEN y, GEN *u, GEN *v) as RgX_extgcd using a standard extended Euclidean algorithm. Usually slower than RgX_extgcd.

GEN RgX_disc(GEN x) returns the discriminant of the t_POL x with respect to its main variable.

GEN RgX_resultant_all(GEN x, GEN y, GEN *sol) returns \( \text{resultant}(x,y) \). If \( \text{sol} \) is not NULL, sets it to the last non-constant remainder in the polynomial remainder sequence if it exists and to \( \text{gen}_0 \) otherwise (e.g. one polynomial has degree 0). Compared to resultant_all, this function always uses the generic subresultant algorithm, hence always computes \( \text{sol} \).

GEN RgX_modXn_shallow(GEN x, long n) returns \( x \mod t^n \), where \( n \geq 0 \). Shallow function.

GEN RgX_modXn_eval(GEN Q, GEN x, long n) special case of RgX_RgXQ_eval, when the modulus is a monomial: returns \( Q(x) \mod t^n \), where \( x \in R[t] \).

GEN RgX_renormalize(GEN x) remove leading terms in x which are equal to (necessarily inexact) zeros.

GEN RgX_renormalize_lg(GEN x, long lx) as setlg(x, lx) followed by RgX_renormalize(x). Assumes that \( 1x \leq 1g(x) \).

GEN RgX_gtofp(GEN x, GEN prec) returns the polynomial obtained by applying \( \text{gtofp}(g(x,i), \text{prec}) \) to all coefficients of x.

GEN RgX_fpnorml2(GEN x, long prec) returns (a stack-clean variant of)

\[
\text{gnorml2( RgX_gtofp(x, \text{prec}) )}
\]

GEN RgX_recip(GEN P) returns the reverse of the polynomial P, i.e. \( X^{\text{deg}P}P(1/X) \).

GEN RgX_recip_shallow(GEN P) shallow function of RgX_recip.

GEN RgX_deflate(GEN P, long d) assuming P is a polynomial of the form \( Q(X^d) \), return Q. Shallow function, not suitable for gerepileupto.

\text{long RgX_deflate_max(GEN P, long *d) sets d to the largest exponent such that P is of the form \( P(x^d) \) (use gequal0 to check whether coefficients are 0), 0 if P is the zero polynomial. Returns RgX_deflate(P,d).}

GEN RgX_inflate(GEN P, long d) return \( P(X^d) \). Shallow function, not suitable for gerepileupto.

GEN RgX_rescale(GEN P, GEN h) returns \( h^{\text{deg}(P)}P(x/h) \). P is an RgX and h is non-zero. (Leaves small objects on the stack. Suitable but inefficient for gerepileupto.)
GEN RgX_unscale(GEN P, GEN h) returns $P(hx)$. (Leaves small objects on the stack. Suitable but inefficient for gerepileupto.)

GEN RgXV_unscale(GEN v, GEN h) apply RgX_unscale to a vector of RgX.

int RgX_is_rational(GEN P) return 1 is the RgX $P$ has only rational coefficients (t_INT and t_FRAC), and 0 otherwise.

int RgX_is_QX(GEN P) return 1 is the RgX $P$ has only t_INT and t_FRAC coefficients, and 0 otherwise.

int RgX_is_ZX(GEN P) return 1 is the RgX $P$ has only t_INT coefficients, and 0 otherwise.

int RgX_is_monomial(GEN x) returns 1 (true) if $x$ is a non-zero monomial in its main variable, 0 otherwise.

long RgX_equal(GEN x, GEN y) returns 1 if the t_POLs $x$ and $y$ have the same degpol and their coefficients are equal (as per gequal). Variable numbers are not checked. Note that this is more stringent than gequal(x,y), which only checks whether $x−y$ satisfies gequal0; in particular, they may have different apparent degrees provided the extra leading terms are 0.

long RgX_equal_var(GEN x, GEN y) returns 1 if $x$ and $y$ have the same variable number and RgX_equal(x,y) is 1.

GEN RgXQ_mul(GEN y, GEN x, GEN T) computes $xy \mod T$

GEN RgXQ_sqr(GEN x, GEN T) computes $x^2 \mod T$

GEN RgXQ_inv(GEN x, GEN T) return the inverse of $x \mod T$.

GEN RgXQ_pow(GEN x, GEN n, GEN T) computes $x^n \mod T$

GEN RgXQ_powu(GEN x, ulong n, GEN T) computes $x^n \mod T$, $n$ being an ulong.

GEN RgXQ_powers(GEN x, long n, GEN T) returns $[x^0,...,x^n]$ as a t_VEC of RgXQs.

int RgXQ_ratlift(GEN x, GEN T, long amax, long bmax, GEN *P, GEN *Q) Assuming that amax + bmax < deg T, attempts to recognize $x$ as a rational function $a/b$, i.e. to find t_POLs $P$ and $Q$ such that

- $P \equiv Qx \mod T$,
- $\deg P \leq \text{amax}$, $\deg Q \leq \text{bmax}$,
- $\gcd(T,P) = \gcd(P,Q)$.

If unsuccessful, the routine returns 0 and leaves $P$, $Q$ unchanged; otherwise it returns 1 and sets $P$ and $Q$.

GEN RgXQ_reverse(GEN f, GEN T) returns a t_POL $g$ of degree $\leq n = \deg T$ such that $T(x)$ divides $(g \circ f)(x) − x$, by solving a linear system. Low-level function underlying modreverse: it returns a lift of $\text{modreverse}(f,T)$; faster than the high-level function since it needs not compute the characteristic polynomial of $f \mod T$ (often already known in applications). In the trivial case where $n \leq 1$, returns a scalar, not a constant t_POL.

GEN RgXQ_matrix_pow(GEN y, long n, long m, GEN P) returns RgXQ_powers(y,m−1,P), as a matrix of dimension $n \geq \deg P$.

GEN RgXQ_norm(GEN x, GEN T) returns the norm of Mod(x, T).
GEN RgXQ_charpoly(GEN x, GEN T, long v) returns the characteristic polynomial of Mod(x, T), in variable v.

GEN RgX_RgXQ_eval(GEN f, GEN x, GEN T) returns f(x) modulo T.

GEN RgX_RgXQV_eval(GEN f, GEN V, GEN T) as RgX_RgXQ_eval(f, x, T), assuming V was output by RgXQ_powers(x, n, T) for some n ≥ 1.

GEN RgX_translate(GEN P, GEN c) assume c is a scalar or a polynomials whose main variable has lower priority than the main variable X of P. Returns P(X + c) (optimized for c = ±1).

GEN RgXQX_translate(GEN P, GEN c, GEN T) assume the main variable X of P has higher priority than the main variable Y of T and c. Return a lift of P(X + Mod(c(Y), T(Y))).

GEN RgXQC_red(GEN z, GEN T) z a vector whose coefficients are RgXs (arbitrary GENs in fact), reduce them to RgXQs (applying grem coefficientwise) in a t_COL.

GEN RgXQV_red(GEN z, GEN T) z a t_POL whose coefficients are RgXs (arbitrary GENs in fact), reduce them to RgXQs (applying grem coefficientwise) in a t_VEC.

GEN RgXQX_mul(GEN x, GEN y, GEN T) multiplies the RgX y by the scalar x.

GEN RgXQX_sqr(GEN x, GEN T)

GEN RgXQX_div(GEN x, GEN y, GEN T, GEN *r) divides the RgX y by the scalar x.

GEN RgXQX_rem(GEN x, GEN y, GEN T, GEN *r)

GEN RgXQX_divrem(GEN x, GEN y, GEN T, GEN *pr)

GEN Kronecker_to_mod(GEN z, GEN T) z ∈ R[X] represents an element P(X,Y) in R[X,Y] mod T(Y) in Kronecker form, i.e. z = P(X, X2n-1).

Let R be some commutative ring, n = deg T and let P(X,Y) ∈ R[X,Y] lift a polynomial in K[Y], where K := R[X]/(T) and deg_x P < 2n − 1 — such as would result from multiplying minimal degree lifts of two polynomials in K[Y]. Let z = P(t, t2n-1) be a Kronecker form of P, this function returns the image of P(X,t) in K[t], with t_POLMOD coefficients. Not stack-clean. Note that t need not be the same variable as Y!
Chapter 8:
Operations on general PARI objects

8.1 Assignment.

It is in general easier to use a direct conversion, e.g. \( y = \text{stoi}(s) \), than to allocate a target of correct type and sufficient size, then assign to it:

\[
\text{GEN } y = \text{cgeti}(3); \text{affsi}(s, y);
\]

These functions can still be moderately useful in complicated garbage collecting scenarios but you will be better off not using them.

\[
\text{void gaffsg(long s, GEN x) assigns the long s into the object x.}
\]

\[
\text{void g affect(GEN x, GEN y) assigns the object x into the object y. Both x and y must be scalar types. Type conversions (e.g. from t_INT to t_REAL or t_INTMOD) occur if legitimate.}
\]

\[
\text{int is universal constant(GEN x) returns 1 if x is a global PARI constant you should never assign to (such as gen_1), and 0 otherwise.}
\]

8.2 Conversions.

8.2.1 Scalars.

\[
\text{double rtodbl(GEN x) applied to a t_REAL x, converts x into a double if possible.}
\]

\[
\text{GEN dbltor(double x) converts the double x into a t_REAL.}
\]

\[
\text{long dblexpo(double x) returns \text{expo(dbltor(x))}, but faster and without cluttering the stack.}
\]

\[
\text{ulong dblmantissa(double x) returns the most significant word in the mantissa of dbltor(x).}
\]

\[
\text{double gtodouble(GEN x) if x is a real number (not necessarily a t_REAL), converts x into a double if possible.}
\]

\[
\text{long gtos(GEN x) converts the t_INT x to a small integer if possible, otherwise raise an exception. This function is similar to its, slightly slower since it checks the type of x.}
\]

\[
\text{double dbllog2r(GEN x) assuming x is a non-zero t_REAL, returns an approximation to log2(|x|)).}
\]

\[
\text{long gtolong(GEN x) if x is an integer (not necessarily a t_INT), converts x into a long if possible.}
\]

\[
\text{GEN fractor(GEN x, long l) applied to a t_FRAC x, converts x into a t_REAL of length prec.}
\]

\[
\text{GEN quadtofp(GEN x, long l) applied to a t_QUAD x, converts x into a t_REAL or t_COMPLEX depending on the sign of the discriminant of x, to precision l BITS_IN_LONG-bit words.}
\]

\[
\text{GEN cxtofp(GEN x, long prec) converts the t_COMPLEX x to a a complex whose real and imaginary parts are t_REAL of length prec (special case of gtofp).}
\]
GEN cxcomptor(GEN x, long prec) converts the t_INT, t_REAL or t_FRAC x to a t_REAL of length prec. These are all the real types which may occur as components of a t_COMPLEX; special case of gtofp (introduced so that the latter is not recursive and can thus be inlined).

GEN gtofp(GEN x, long prec) converts the complex number x (t_INT, t_REAL, t_FRAC, t_QUAD or t_COMPLEX) to either a t_REAL or t_COMPLEX whose components are t_REAL of precision prec; not necessarily of length prec: a real 0 may be given as real 0(...). If the result is a t_COMPLEX extra care is taken so that its modulus really has accuracy prec: there is a problem if the real part of the input is an exact 0; indeed, converting it to real 0(prec) would be wrong if the imaginary part is tiny, since the modulus would then become equal to 0, as in 1.E−100+0.E−28 = 0.E−28.

GEN gtopm(GEN z, long prec) converts the real number x (t_INT, t_REAL, t_FRAC, real t_QUAD) to either a t_INT or a t_REAL of precision prec. Not memory clean if x is a t_INT: we return x itself and not a copy.

GEN cvtop(GEN x, GEN p, long l) converts x into a t_PADIC of precision l. Works component-wise on recursive objects, e.g. t_POL or t_VEC. Converting 0 yields \(O(p^l)\); converting a non-zero number yield a result well defined modulo \(p^v(x)+l\).

GEN cvtop2(GEN x, GEN y) y being a p-adic, converts the scalar x to a p-adic of the same accuracy. Shallow function.

GEN cvstop2(long s, GEN y) y being a p-adic, converts the scalar s to a p-adic of the same accuracy. Shallow function.

GEN gprec(GEN x, long l) returns a copy of x whose precision is changed to l digits. The precision change is done recursively on all components of x. Digits means decimal, p-adic and X-adic digits for t_REAL, t_SER, t_PADIC components, respectively.

GEN gprec_w(GEN x, long l) returns a shallow copy of x whose t_REAL components have their precision changed to l words. This is often more useful than gprec.

GEN gprec_wtrunc(GEN x, long l) returns a shallow copy of x whose t_REAL components have their precision truncated to l words. Contrary to gprec_w, this function may never increase the precision of x.

8.2.2 Modular objects / lifts.

GEN gmodulo(GEN x, GEN y) creates the object Mod(x,y) on the PARI stack, where x and y are either both t_INTs, and the result is a t_INTMOD, or x is a scalar or a t_POL and y a t_POL, and the result is a t_POLMOD.

GEN gmodulgs(GEN x, long y) same as gmodulo except y is a long.

GEN gmodulsg(long x, GEN y) same as gmodulo except x is a long.

GEN gmodulss(long x, long y) same as gmodulo except both x and y are longs.

GEN liftall_shallow(GEN x) shallow version of liftall

GEN liftint_shallow(GEN x) shallow version of liftint

GEN liftpol_shallow(GEN x) shallow version of liftpol

GEN centerlift0(GEN x, long v) DEPRECATED, kept for backward compatibility only: use either lift0(x,v) or centerlift(x).
8.2.3 Between polynomials and coefficient arrays.

`GEN gtopoly(GEN x, long v)` converts or truncates the object `x` into a `t_POL` with main variable number `v`. A common application would be the conversion of coefficient vectors (coefficients are given by decreasing degree). E.g. `[2,3]` goes to `2*v + 3`.

`GEN gtopolyrev(GEN x, long v)` converts or truncates the object `x` into a `t_POL` with main variable number `v`, but vectors are converted in reverse order compared to `gtopoly` (coefficients are given by increasing degree). E.g. `[2,3]` goes to `3*v + 2`. In other words the vector represents a polynomial in the basis `(1,v,v^2,v^3,...)`.

`GEN normalizepol(GEN x)` applied to an unnormalized `t_POL x` (with all coefficients correctly set except that `leading_term(x)` might be zero), normalizes `x` correctly in place and returns `x`. For internal use. Normalizing means deleting all leading exact zeroes (as per `isexactzero`), except if the polynomial turns out to be 0, in which case we try to find a coefficient `c` which is a non-rational zero, and return the constant polynomial `c`. (We do this so that information about the base ring is not lost.)

`GEN normalizepol_lg(GEN x, long l)` applies `normalizepol` to `x`, pretending that `lg(x)` is `l`, which must be less than or equal to `lg(x)`. If equal, the function is equivalent to `normalizepol(x)`.

`GEN normalizepol_approx(GEN x, long lx)` as `normalizepol_lg`, with the difference that we just delete all leading zeroes (as per `gequal0`). This rougher normalization is used when we have no other choice, for instance before attempting a Euclidean division by `x`.

The following routines do not copy coefficients on the stack (they only move pointers around), hence are very fast but not suitable for `gerepile` calls. Recall that an `RgV` (resp. an `RgX`, resp. an `RgM`) is a `t_VEC` or `t_COL` (resp. a `t_POL`, resp. a `t_MAT`) with arbitrary components. Similarly, an `RgXV` is a `t_VEC` or `t_COL` with `RgX` components, etc.

`GEN RgV_to_RgX(GEN x, long v)` converts the `RgV x` to a (normalized) polynomial in variable `v` (as `gtopolyrev`, without copy).

`GEN RgV_to_RgX_reverse(GEN x, long v)` converts the `RgV x` to a (normalized) polynomial in variable `v` (as `gtopoly`, without copy).

`GEN RgX_to_RgV(GEN x, long N)` converts the `t_POL x` to a `t_COL` `v` with `N` components. Coefficients of `x` are listed by increasing degree, so that `y[i]` is the coefficient of the term of degree `i - 1` in `x`.

`GEN Rg_to_RgV(GEN x, long N)` as `RgX_to_RgV`, except that other types than `t_POL` are allowed for `x`, which is then considered as a constant polynomial.

`GEN RgM_to_RgXV(GEN x, long v)` converts the `RgM x` to a `t_VEC` of `RgX`, by repeated calls to `RgV_to_RgX`.

`GEN RgV_to_RgM(GEN v, long N)` converts the vector `v` to a `t_MAT` with `N` rows, by repeated calls to `Rg_to_RgV`.

`GEN RgXV_to_RgM(GEN v, long N)` converts the vector of `RgX v` to a `t_MAT` with `N` rows, by repeated calls to `RgX_to_RgV`.

`GEN RgM_to_RgXX(GEN x, long v, long w)` converts the `RgM x` into a `t_POL` in variable `v`, whose coefficients are `t_POLs` in variable `w`. This is a shortcut for

```
RgV_to_RgX( RgM_to_RgXV(x, w), v );
```
There are no consistency checks with respect to variable priorities: the above is an invalid object if \( \text{varncmp}(v, w) \geq 0. \)

\[
\text{GEN RxXX_to_RgM(GEN x, long N)} \quad \text{converts the t_POL x with RgX (or constant) coefficients to a matrix with N rows.}
\]

\[
\text{GEN RxXY_swap(GEN P, long n, long w)} \quad \text{converts the bivariate polynomial P(u,v) (a t_POL with t_POL or scalar coefficients) to P(\text{pol_x[w]}, u), assuming n is an upper bound for deg_v(P).}
\]

\[
\text{GEN RxXY_swapspec(GEN C, long n, long w, long 1P)} \quad \text{as RxXY_swap where the coefficients of P are given by gel(C,0),...,gel(C,1P-1).}
\]

\[
\text{GEN Rx_to_ser(GEN x, long l)} \quad \text{applied to a t_POL x, creates a shallow t_SER of length l \geq 2 starting with x. Unless the polynomial is an exact zero, the coefficient of lowest degree T^d of the result is not an exact zero (as per isexactzero). The remainder is } O(T^{d+l}).
\]

\[
\text{GEN Rx_to_ser_inexact(GEN x, long l)} \quad \text{applied to a t_POL x, creates a shallow t_SER of length l starting with x. Unless the polynomial is zero, the coefficient of lowest degree T^d of the result is not zero (as per gequal0). The remainder is } O(T^{d+l}).
\]

\[
\text{GEN rfrac_to_ser(GEN x, long l)} \quad \text{applied to a t_RFRAC x, creates a t_SER of length l congruent to x. Not memory-clean but suitable for gerepileupto.}
\]

\[
\text{GEN gtoser(GEN s, long v, long d)} \quad \text{converts the object s into a t_SER with main variable number v and d > 0 significant terms. More precisely}
\]

\[
\quad \text{• if s is a scalar, we return a constant power series with d significant terms.}
\]

\[
\quad \text{• if s is a t_POL, it is truncated to d terms if needed.}
\]

\[
\quad \text{• If s is a vector, the coefficients of the vector are understood to be the coefficients of the power series starting from the constant term (as in Polrev), and the precision d is ignored.}
\]

\[
\quad \text{• If s is already a power series in v, we return a copy, and the precision d is again ignored.}
\]

\[
\text{GEN gtocol(GEN x)} \quad \text{converts the object x into a t_COL}
\]

\[
\text{GEN gtomat(GEN x)} \quad \text{converts the object x into a t_MAT}
\]

\[
\text{GEN gtovec(GEN x)} \quad \text{converts the object x into a t_VEC}
\]

\[
\text{GEN gtovecsmall(GEN x)} \quad \text{converts the object x into a t_VECSMALL}
\]

\[
\text{GEN normalize(GEN x)} \quad \text{applied to an unnormalized t_SER x (i.e. type t_SER with all coefficients correctly set except that x[2] might be zero), normalizes x correctly in place. Returns x. For internal use.}
\]

\[
\text{GEN serchop0(GEN s)} \quad \text{given a t_SER of the form } x^v s(x), \text{ with } s(0) \neq 0, \text{ return } x^v (s - s(0)). \text{ Shallow function.}
\]
8.3 Constructors.

8.3.1 Clean constructors.

\texttt{GEN zeropadic(GEN p, long n)} creates a 0 \texttt{t\_PADIC} equal to \(O(p^n)\).

\texttt{GEN zeronser(long v, long n)} creates a 0 \texttt{t\_SER} variable \(v\) equal to \(O(X^n)\).

\texttt{GEN scalarser(GEN x, long v, long prec)} creates a constant \texttt{t\_SER} variable \(v\) and precision \(prec\), whose constant coefficient is (a copy of) \(x\), in other words \(x + O(v^{\text{prec}})\). Assumes that \(x\) is non-zero.

\texttt{GEN pol\_0(long v)} Returns the constant polynomial 0 in variable \(v\).

\texttt{GEN pol\_1(long v)} Returns the constant polynomial 1 in variable \(v\).

\texttt{GEN pol\_x(long v)} Returns the monomial of degree 1 in variable \(v\).

\texttt{GEN pol\_x\_powers(long N, long v)} returns the powers of \texttt{pol\_x(v)}, of degree 0 to \(N\), in a vector with \(N + 1\) components.

\texttt{GEN scalarpol(GEN x, long v)} creates a constant \texttt{t\_POL} variable \(v\), whose constant coefficient is (a copy of) \(x\).

\texttt{GEN deg\_1pol(GEN a, GEN b, long v)} creates the degree 1 \texttt{t\_POL} \(a \text{pol}\_x(v) + b\)

\texttt{GEN zeropol(long v)} is identical \texttt{pol\_0}.

\texttt{GEN zero\_col(long n)} creates a \texttt{t\_COL} with \(n\) components set to \texttt{gen\_0}.

\texttt{GEN zero\_vec(long n)} creates a \texttt{t\_VEC} with \(n\) components set to \texttt{gen\_0}.

\texttt{GEN col\_ei(long n, long i)} creates a \texttt{t\_COL} with \(n\) components set to \texttt{gen\_0}, but for the \(i\)-th one which is set to \texttt{gen\_1} (i-th vector in the canonical basis).

\texttt{GEN vec\_ei(long n, long i)} creates a \texttt{t\_VEC} with \(n\) components set to \texttt{gen\_0}, but for the \(i\)-th one which is set to \texttt{gen\_1} (i-th vector in the canonical basis).

\texttt{GEN trivial\_fact(void)} returns the trivial (empty) factorization \texttt{Mat([],[])}

\texttt{GEN prime\_fact(GEN x)} returns the factorization \texttt{Mat([x], [1])}

\texttt{GEN Rg\_col\_ei(GEN x, long n, long i)} creates a \texttt{t\_COL} with \(n\) components set to \texttt{gen\_0}, but for the \(i\)-th one which is set to \(x\).

\texttt{GEN vec\_small\_ei(long n, long i)} creates a \texttt{t\_VECSMALL} with \(n\) components set to 0, but for the \(i\)-th one which is set to 1 (i-th vector in the canonical basis).

\texttt{GEN scalar\_col(GEN x, long n)} creates a \texttt{t\_COL} with \(n\) components set to \texttt{gen\_0}, but the first one which is set to a copy of \(x\). (The name comes from \texttt{Rg\_V\_isscalar}.)

\texttt{GEN mkint\_modu(ulong x, ulong y)} creates the \texttt{t\_INTMOD \text{Mod}(x, y)}). The inputs must satisfy \(x < y\).

\texttt{GEN zerom\_mat(long m, long n)} creates a \texttt{t\_MAT} with \(m \times n\) components set to \texttt{gen\_0}. Note that the result allocates a \emph{single} column, so modifying an entry in one column modifies it in all columns. To fully allocate a matrix initialized with zero entries, use \texttt{zeromat\_copy}.

\texttt{GEN zeromat\_copy(long m, long n)} creates a \texttt{t\_MAT} with \(m \times n\) components set to \texttt{gen\_0}.
GEN matid(long n) identity matrix in dimension n (with components gen_1 and gen_0).

GEN scalarmat(GEN x, long n) scalar matrix, x times the identity.

GEN scalarmat_s(long x, long n) scalar matrix, stoi(x) times the identity.

GEN vecrange(GEN a, GEN b) returns the t_VEC [a..b].

GEN vecrangess(long a, long b) returns the t_VEC [a..b].

See also next section for analogs of the following functions:

GEN mkfraccopy(GEN x, GEN y) creates the t_FRAC x/y. Assumes that y > 1 and (x, y) = 1.

GEN mkrfraccopy(GEN x, GEN y) creates the t_RFRAC x/y. Assumes that y is a t_POL, x a compatible type whose variable has lower or same priority, with (x, y) = 1.

GEN mkcolcopy(GEN x) creates a 1-dimensional t_COL containing x.

GEN mkmatcopy(GEN x) creates a 1-by-1 t_MAT wrapping the t_COL x.

GEN mkveccopy(GEN x) creates a 1-dimensional t_VEC containing x.

GEN mkvec2copy(GEN x, GEN y) creates a 2-dimensional t_VEC equal to [x,y].

GEN mkcolss(long x) creates a 1-dimensional t_COL containing stoi(x).

GEN mkcolss(long x, long y) creates a 2-dimensional t_COL containing [stoi(x), stoi(y)].

GEN mkcolss(long x, long y, long z) creates a 3-dimensional t_COL containing [stoi(x), stoi(y), stoi(z)].

GEN mkcolss(long x, long y, long z, long t) creates a 4-dimensional t_COL containing [stoi(x), stoi(y), stoi(z), stoi(t)].

GEN mkvecsss(long x) creates a 1-dimensional t_VEC containing stoi(x).

GEN mkvecsss(long x, long y) creates a 2-dimensional t_VEC containing [stoi(x), stoi(y)].

GEN mkvecsss(long x, long y, long z) creates a 3-dimensional t_VEC containing [stoi(x), stoi(y), stoi(z)].

GEN mkvecsss(long x, long y, long z, long t) creates a 4-dimensional t_VEC containing [stoi(x), stoi(y), stoi(z), stoi(t)].

GEN mkvecsmallss(long x) creates a 1-dimensional t_VECSMALL containing x.

GEN mkvecsmallss(long x, long y) creates a 2-dimensional t_VECSMALL containing [x, y].

GEN mkvecsmallss(long x, long y, long z) creates a 3-dimensional t_VECSMALL containing [x, y, z].

GEN mkvecsmallss(long x, long y, long z, long t) creates a 4-dimensional t_VECSMALL containing [x, y, z, t].

GEN mkvecsmallss(long n, ...) returns the t_VECSMALL whose n coefficients (long) follow.
8.3.2 Unclean constructors.

Contrary to the policy of general PARI functions, the functions in this subsection do not copy their arguments, nor do they produce an object a priori suitable for gerepileupto. In particular, they are faster than their clean equivalent (which may not exist). If you restrict their arguments to universal objects (e.g. gen_0), then the above warning does not apply.

GEN mkcomplex(GEN x, GEN y) creates the t_COMPLEX $x + iy$.

GEN mulcXi(GEN x) creates the t_COMPLEX $ix$. The result in general contains data pointing back to the original $x$. Use gcopy if this is a problem. But in most cases, the result is to be used immediately, before $x$ is subject to garbage collection.

GEN mulcXmI(GEN x), as mulcXi, but returns the t_COMPLEX $-ix$.

GEN mkquad(GEN n, GEN x, GEN y) creates the t_QUAD $x + yw$, where $w$ is a root of $n$, which is of the form quadpoly(D).

GEN mkfrac(GEN x, GEN y) creates the t_FRAC $x/y$. Assumes that $y > 1$ and $(x, y) = 1$.

GEN mkfracx(GEN x, GEN y) creates the t_RFRAC $x/y$. Assumes that $y$ is a t_POL, $x$ a compatible type whose variable has lower or same priority, with $(x, y) = 1$.

GEN mkcol(GEN x) creates a 1-dimensional t_COL containing $x$.

GEN mkcol2(GEN x, GEN y) creates a 2-dimensional t_COL equal to $[x, y]$.

GEN mkcol3(GEN x, GEN y, GEN z) creates a 3-dimensional t_COL equal to $[x, y, z]$.

GEN mkcol4(GEN x, GEN y, GEN z, GEN t) creates a 4-dimensional t_COL equal to $[x, y, z, t]$.

GEN mkcol5(GEN a1, GEN a2, GEN a3, GEN a4, GEN a5) creates the 5-dimensional t_COL equal to $[a_1, a_2, a_3, a_4, a_5]$.

GEN mkintmod(GEN x, GEN y) creates the t_INTMOD $\text{Mod}(x, y)$. The inputs must be t_INTs satisfying $0 \leq x < y$.

GEN mkpolmod(GEN x, GEN y) creates the t_POLMOD $\text{Mod}(x, y)$. The input must satisfy $\deg x < \deg y$ with respect to the main variable of the t_POL $y$. $x$ may be a scalar.

GEN mkmat(GEN x) creates a 1-column t_MAT with column $x$ (a t_COL).

GEN mkmat2(GEN x, GEN y) creates a 2-column t_MAT with columns $x, y$ (t_COLS of the same length).

GEN mkmat3(GEN x, GEN y, GEN z) creates a 3-column t_MAT with columns $x, y, z$ (t_COLS of the same length).

GEN mkmat4(GEN x, GEN y, GEN z, GEN t) creates a 4-column t_MAT with columns $x, y, z, t$ (t_COLS of the same length).

GEN mkmat5(GEN x, GEN y, GEN z, GEN t, GEN u) creates a 5-column t_MAT with columns $x, y, z, t, u$ (t_COLS of the same length).

GEN mkvec(GEN x) creates a 1-dimensional t_VEC containing $x$.

GEN mkvec2(GEN x, GEN y) creates a 2-dimensional t_VEC equal to $[x, y]$.

GEN mkvec3(GEN x, GEN y, GEN z) creates a 3-dimensional t_VEC equal to $[x, y, z]$.

GEN mkvec4(GEN x, GEN y, GEN z, GEN t) creates a 4-dimensional t_VEC equal to $[x, y, z, t]$.
GEN mkvec5(GEN a1, GEN a2, GEN a3, GEN a4, GEN a5) creates the 5-dimensional \textit{t\_VEC} equal to \([a_1, a_2, a_3, a_4, a_5]\).

GEN mkqfi(GEN x, GEN y, GEN z) creates \textit{t\_QFI} equal to \texttt{Qfb(x,y,z)}, assuming that \(y^2 - 4xz < 0\).

GEN mkerr(long n) returns a \textit{t\_ERROR} with error code \(n\) (\texttt{enum err\_list}).

It is sometimes useful to return such a container whose entries are not universal objects, but nonetheless suitable for \texttt{gerepiloup\_te}. If the entries can be computed at the time the result is returned, the following macros achieve this effect:

GEN retmkvec(GEN x) returns a vector containing the single entry \(x\), where the vector root is created just before the function argument \(x\) is evaluated. Expands to

\begin{verbatim}
{ 
    GEN res = cgetg(2, t\_VEC);
    gel(res, 1) = x; /* or rather, the expansion of \(x\) */
    return res;
}
\end{verbatim}

For instance, the \texttt{retmkvec(gc\_copy(x))} returns a clean object, just like \texttt{return mkvec\_copy(x)} would.

GEN retmkvec2(GEN x, GEN y) returns the 2-dimensional \textit{t\_VEC} \([x,y]\).

GEN retmkvec3(GEN x, GEN y, GEN z) returns the 3-dimensional \textit{t\_VEC} \([x,y,z]\).

GEN retmkvec4(GEN x, GEN y, GEN z, GEN t) returns the 4-dimensional \textit{t\_VEC} \([x,y,z,t]\).

GEN retmkvec5(GEN x, GEN y, GEN z, GEN t, GEN u) returns the 5-dimensional row vector \([x,y,z,t,u]\).

GEN retconst\_vec(long n, GEN x) returns the \(n\)-dimensional \textit{t\_VEC} whose entries are constant and all equal to \(x\).

GEN retmkcol(GEN x) returns the 1-dimensional \textit{t\_COL} \([x]\).

GEN retmkcol2(GEN x, GEN y) returns the 2-dimensional \textit{t\_COL} \([x,y]\).

GEN retmkcol3(GEN x, GEN y, GEN z) returns the 3-dimensional \textit{t\_COL} \([x,y,z]\).

GEN retmkcol4(GEN x, GEN y, GEN z, GEN t) returns the 4-dimensional \textit{t\_COL} \([x,y,z,t]\).

GEN retmkcol5(GEN x, GEN y, GEN z, GEN t, GEN u) returns the 5-dimensional column vector \([x,y,z,t,u]\).

GEN retconst\_col(long n, GEN x) returns the \(n\)-dimensional \textit{t\_COL} whose entries are constant and all equal to \(x\).

GEN retmkmat(GEN x) returns the 1-column \textit{t\_MAT} with column \(x\).

GEN retmkmat2(GEN x, GEN y) returns the 2-column \textit{t\_MAT} with columns \(x, y\).

GEN retmkmat3(GEN x, GEN y, GEN z) returns the 3-column \textit{t\_MAT} with columns \(x, y, z\).

GEN retmkmat4(GEN x, GEN y, GEN z, GEN t) returns the 4-column \textit{t\_MAT} with columns \(x, y, z, t\).
GEN retmkmat5(GEN x, GEN y, GEN z, GEN t, GEN u) returns the 5-dimensional t_MAT with columns x, y, z, t, u.

GEN retmkcomplex(GEN x, GEN y) returns the t_COMPLEX x + I*y.

GEN retmkfrac(GEN x, GEN y) returns the t_FRAC x / y. Assume x and y are coprime and y > 1.

GEN retmkintmod(GEN x, GEN y) returns the t_INTMOD Mod(x, y).

GEN retmkqfi(GEN a, GEN b, GEN c).

GEN retmkqfr(GEN a, GEN b, GEN c, GEN d).

GEN retmkquad(GEN n, GEN a, GEN b).

GEN retmkpmod(GEN x, GEN y) returns the t_POLMOD Mod(x, y).

GEN mkinxn(long n, ...) returns the non-negative t_INT whose development in base $2^{32}$ is given by the following n words (unsigned long). It is assumed that all such arguments are less than $2^{32}$ (the actual size of (long) is irrelevant, the behavior is also as above on 64-bit machines).

\[ \text{mkinxn}(3, a2, a1, a0) \]

returns $a_2 2^{64} + a_1 2^{32} + a_0$.

GEN mkpoln(long n, ...) Returns the t_POL whose n coefficients (GEN) follow, in order of decreasing degree.

\[ \text{mkpoln}(3, \text{gen}_1, \text{gen}_2, \text{gen}_0) \]

returns the polynomial $X^2 + 2X$ (in variable 0, use setvarn if you want other variable numbers). Beware that n is the number of coefficients, hence one more than the degree.

GEN mkvecn(long n, ...) returns the t_VEC whose n coefficients (GEN) follow.

GEN mkcoln(long n, ...) returns the t_COL whose n coefficients (GEN) follow.

GEN scalarscol_shallow(GEN x, long n) creates a t_COL with n components set to gen_0, but the first one which is set to a shallow copy of x. (The name comes from RgV_iscalar.)

GEN scalarmat_shallow(GEN x, long n) creates an $n \times n$ scalar matrix whose diagonal is set to shallow copies of the scalar x.

GEN diagonal_shallow(GEN x) returns a diagonal matrix whose diagonal is given by the vector x. Shallow function.

GEN scalarpol_shallow(GEN a, long v) returns the degree 0 t_POL apol_x(v)^0.

GEN degipol_shallow(GEN a, GEN b, long v) returns the degree 1 t_POL apol_x(v) + b.

GEN zeropadic_shallow(GEN p, long n) returns a (shallow) 0 t_PADIC equal to $O(p^n)$. 

161
8.3.3 From roots to polynomials.

GEN deg1_from_roots(GEN L, long v) given a vector \( L \) of scalars, returns the vector of monic linear polynomials in variable \( v \) whose roots are the \( L[i] \), i.e. the \( x - L[i] \).

GEN roots_from_deg1(GEN L) given a vector \( L \) of monic linear polynomials, return their roots, i.e. the \( -L[i](0) \).

GEN roots_to_pol(GEN L, long v) given a vector of scalars \( L \), returns the monic polynomial in variable \( v \) whose roots are the \( L[i] \). Calls divide_conquer_prod, so leaves some garbage on stack, but suitable for gerepileupto.

GEN roots_to_pol_r1(GEN L, long v, long r1) as roots_to_pol assuming the first \( r_1 \) roots are “real”, and the following ones are representatives of conjugate pairs of “complex” roots. So if \( L \) has \( r_1 + r_2 \) elements, we obtain a polynomial of degree \( r_1 + 2r_2 \). In most applications, the roots are indeed real and complex, but the implementation assumes only that each “complex” root \( z \) introduces a quadratic factor \( X^2 - \text{trace}(z)X + \text{norm}(z) \). Calls divide_conquer_prod. Calls divide_conquer_prod, so leaves some garbage on stack, but suitable for gerepileupto.

8.4 Integer parts.

GEN gfloor(GEN x) creates the floor of \( x \), i.e. the (true) integral part.

GEN gfrac(GEN x) creates the fractional part of \( x \), i.e. \( x \) minus the floor of \( x \).

GEN gceil(GEN x) creates the ceiling of \( x \).

GEN ground(GEN x) rounds towards \( +\infty \) the components of \( x \) to the nearest integers.

GEN grndtoi(GEN x, long *e) same as ground, but in addition sets \(*e\) to the binary exponent of \( x - \text{ground}(x) \). If this is positive, all significant bits are lost. This kind of situation raises an error message in ground but not in grndtoi.

GEN gtrunc(GEN x) truncates \( x \). This is the false integer part if \( x \) is a real number (i.e. the unique integer closest to \( x \) among those between 0 and \( x \)). If \( x \) is a t_SER, it is truncated to a t_POL; if \( x \) is a t_RFRAC, this takes the polynomial part.

GEN gtrunc2n(GEN x, long n) creates the floor of \( 2^n x \), this is only implemented for t_INT, t_REAL, t_FRAC and t_COMPLEX of those.

GEN gcvtoi(GEN x, long *e) analogous to grndtoi for t_REAL inputs except that rounding is replaced by truncation. Also applies componentwise for vector or matrix inputs; otherwise, sets \(*e\) to -HIGHEXPOBIT (infinite real accuracy) and return gtrunc(\( x \)).
8.5 Valuation and shift.

GEN gshift[z](GEN x, long n[, GEN z]) yields the result of shifting (the components of) x left by n (if n is non-negative) or right by −n (if n is negative). Applies only to t_INT and vectors/matrices of such. For other types, it is simply multiplication by 2^n.

GEN gmul2n[z](GEN x, long n[, GEN z]) yields the product of x and 2^n. This is different from gshift when n is negative and x is a t_INT: gshift truncates, while gmul2n creates a fraction if necessary.

long gvaluation(GEN x, GEN p) returns the greatest exponent e such that p^e divides x, when this makes sense.

long gval(GEN x, long v) returns the highest power of the variable number v dividing the t_POL x.

8.6 Comparison operators.

8.6.1 Generic.

long gcmp(GEN x, GEN y) comparison of x with y: returns 1 (x > y), 0 (x = y) or −1 (x < y). Two t_STR are compared using the standard lexicographic ordering; a t_STR is considered strictly larger than any non-string type. If neither x nor y is a t_STR, their allowed types are t_INT, t_REAL or t_FRAC. Used cmp_universal to compare arbitrary GENs.

long lexcmp(GEN x, GEN y) comparison of x with y for the lexicographic ordering; when comparing objects of different lengths whose components are all equal up to the smallest of their length, consider that the longest is largest. Consider scalars as 1-component vectors. Return gcmp(x,y) if both arguments are scalars.

int gequalX(GEN x) return 1 (true) if x is a variable (monomial of degree 1 with t_INT coefficients equal to 1 and 0), and 0 otherwise

long gequal(GEN x, GEN y) returns 1 (true) if x is equal to y, 0 otherwise. A priori, this makes sense only if x and y have the same type, in which case they are recursively compared component-wise. When the types are different, a true result means that x - y was successfully computed and that gequal0 found it equal to 0. In particular

gequal(cgetg(1, t_VEC), gen_0)

is true, and the relation is not transitive. E.g. an empty t_COL and an empty t_VEC are not equal but are both equal to gen_0.

long gidentical(GEN x, GEN y) returns 1 (true) if x is identical to y, 0 otherwise. In particular, the types and length of x and y must be equal. This test is much stricter than gequal, in particular, t_REAL with different accuracies are tested different. This relation is transitive.
8.6.2 Comparison with a small integer.

`int isexactzero(GEN x)` returns 1 (true) if `x` is exactly equal to 0 (including `t_INTMOD`s like `Mod(0,2)`), and 0 (false) otherwise. This includes recursive objects, for instance vectors, whose components are 0.

`int isrationalzero(GEN x)` returns 1 (true) if `x` is equal to an integer 0 (excluding `t_INTMOD`s like `Mod(0,2)`), and 0 (false) otherwise. Contrary to `isintzero`, this includes recursive objects, for instance vectors, whose components are 0.

`int ismpzero(GEN x)` returns 1 (true) if `x` is a `t_INT` or a `t_REAL` equal to 0.

`int isintzero(GEN x)` returns 1 (true) if `x` is a `t_INT` equal to 0.

`int isint1(GEN x)` returns 1 (true) if `x` is a `t_INT` equal to 1.

`int isintm1(GEN x)` returns 1 (true) if `x` is a `t_INT` equal to 1.

`int equali1(GEN n)` Assuming that `x` is a `t_INT`, return 1 (true) if `x` is equal to 1, and return 0 (false) otherwise.

`int equalim1(GEN n)` Assuming that `x` is a `t_INT`, return 1 (true) if `x` is equal to −1, and return 0 (false) otherwise.

`int is_pm1(GEN x)` Assuming that `x` is a non-zero `t_INT`, return 1 (true) if `x` is equal to −1 or 1, and return 0 (false) otherwise.

`int gequal0(GEN x)` returns 1 (true) if `x` is equal to 0, 0 (false) otherwise.

`int gequal1(GEN x)` returns 1 (true) if `x` is equal to 1, 0 (false) otherwise.

`int gequalm1(GEN x)` returns 1 (true) if `x` is equal to −1, 0 (false) otherwise.

`long gcmpsg(long s, GEN x)`

`long gcmpgs(GEN x, long s)` comparison of `x` with the `long s`.

`GEN gmaxsg(long s, GEN x)`

`GEN gmaxgs(GEN x, long s)` returns the largest of `x` and the `long s` (converted to `GEN`)

`GEN gmaxsg(long s, GEN x)`

`GEN gminsg(long s, GEN x)`

`GEN gmaxgs(GEN x, long s)` returns the smallest of `x` and the `long s` (converted to `GEN`)

`long gequalsg(long s, GEN x)`

`long gequalgs(GEN x, long s)` returns 1 (true) if `x` is equal to the `long s`, 0 otherwise.

164
8.7 Miscellaneous Boolean functions.

int isrationalzeroscalar(GEN x) equivalent to, but faster than,

\[
\text{is\_scalar\_t(typ(x)) \&\& isrationalzero(x)}
\]

int isinexact(GEN x) returns 1 (true) if \( x \) has an inexact component, and 0 (false) otherwise.

int isinexactreal(GEN x) return 1 if \( x \) has an inexact \text{t\_REAL} component, and 0 otherwise.

int isrealappr(GEN x, long e) applies (recursively) to complex inputs; returns 1 if \( x \) is approximately real to the bit accuracy \( e \), and 0 otherwise. This means that any \text{t\_COMPLEX} component must have imaginary part \( t \) satisfying \( \text{gexpo}(t) < e. \)

int isint(GEN x, GEN *n) returns 0 (false) if \( x \) does not round to an integer. Otherwise, returns 1 (true) and set \( n \) to the rounded value.

int issmall(GEN x, long *n) returns 0 (false) if \( x \) does not round to a small integer (suitable for \text{itos}). Otherwise, returns 1 (true) and set \( n \) to the rounded value.

long iscomplex(GEN x) returns 1 (true) if \( x \) is a complex number (of component types embeddable into the reals) but is not itself real, 0 if \( x \) is a real (not necessarily of type \text{t\_REAL}), or raises an error if \( x \) is not embeddable into the complex numbers.

8.7.1 Obsolete.

The following less convenient comparison functions and Boolean operators were used by the historical GP interpreter. They are provided for backward compatibility only and should not be used:

GEN gle(GEN x, GEN y)
GEN glt(GEN x, GEN y)
GEN gge(GEN x, GEN y)
GEN ggt(GEN x, GEN y)
GEN geq(GEN x, GEN y)
GEN gne(GEN x, GEN y)
GEN gor(GEN x, GEN y)
GEN gand(GEN x, GEN y)
GEN gnot(GEN x, GEN y)
8.8 Sorting.

8.8.1 Basic sort.
GEN sort(GEN x) sorts the vector x in ascending order using a mergesort algorithm, and gcmp as the underlying comparison routine (returns the sorted vector). This routine copies all components of x, use gen_sort_inplace for a more memory-efficient function.
GEN lexsort(GEN x), as sort, using lexcmp instead of gcmp as the underlying comparison routine.
GEN vecsort(GEN x, GEN k), as sort, but sorts the vector x in ascending lexicographic order, according to the entries of the t_VECSMALL k. For example, if k = [2, 1, 3], sorting will be done with respect to the second component, and when these are equal, with respect to the first, and when these are equal, with respect to the third.

8.8.2 Indirect sorting.
GEN indexsort(GEN x) as sort, but only returns the permutation which, applied to x, would sort the vector. The result is a t_VECSMALL.
GEN indexlexsort(GEN x), as indexsort, using lexcmp instead of gcmp as the underlying comparison routine.
GEN indexvecsort(GEN x, GEN k), as vecsort, but only returns the permutation that would sort the vector x.

long vecindexmin(GEN x) returns the index for a maximal element of x (t_VEC, t_COL or t_VECSMALL).
long vecindexmax(GEN x) returns the index for a maximal element of x (t_VEC, t_COL or t_VECSMALL).
long vecindexmax(GEN x)

8.8.3 Generic sort and search. The following routines allow to use an arbitrary comparison function int (*cmp)(void* data, GEN x, GEN y), such that cmp(data,x,y) returns a negative result if x < y, a positive one if x > y and 0 if x = y. The data argument is there in case your cmp requires additional context.
GEN gen_sort(GEN x, void *data, int (*cmp)(void *,GEN,GEN)), as sort, with an explicit comparison routine.
GEN gen_sort_uniq(GEN x, void *data, int (*cmp)(void *,GEN,GEN)), as gen_sort, removing duplicate entries.
GEN gen_indexsort(GEN x, void *data, int (*cmp)(void*,GEN,GEN)), as indexsort.
GEN gen_indexsort_uniq(GEN x, void *data, int (*cmp)(void*,GEN,GEN)), as indexsort, removing duplicate entries.
void gen_sort_inplace(GEN x, void *data, int (*cmp)(void*,GEN,GEN), GEN *perm) sort x in place, without copying its components. If perm is non-NULL, it is set to the permutation that would sort the original x.
GEN gen_setminus(GEN A, GEN B, int (*cmp)(GEN,GEN)) given two sorted vectors A and B, returns the vector of elements of A not belonging to B.

166
GEN sort_factor(GEN y, void *data, int (*cmp)(void *,GEN,GEN)): assuming y is a factorization matrix, sorts its rows in place (no copy is made) according to the comparison function cmp applied to its first column.

GEN merge_sort_uniq(GEN x, GEN y, void *data, int (*cmp)(void *,GEN,GEN)) assuming x and y are sorted vectors, with respect to the cmp comparison function, return a sorted concatenation, with duplicates removed.

GEN merge_factor(GEN fx, GEN fy, void *data, int (*cmp)(void *,GEN,GEN)) let fx and fy be factorization matrices for X and Y sorted with respect to the comparison function cmp (see sort_factor), returns the factorization of X * Y. Zero exponents in the latter factorization are preserved, e.g. when merging the factorization of 2 and 1/2, the result is 2^0.

long gen_search(GEN v, GEN y, long flag, void *data, int (*cmp)(void*,GEN,GEN)). Let v be a vector sorted according to cmp(data,a,b); look for an index i such that v[i] is equal to y. flag has the same meaning as in setsearch: if flag is 0, return i if it exists and 0 otherwise; if flag is non-zero, return 0 if i exists and the index where y should be inserted otherwise.

long tablesearch(GEN T, GEN x, int (*cmp)(GEN,GEN)) is a faster implementation for the common case gen_search(T,x,0,cmp,cmp_nodata).

8.8.4 Further useful comparison functions.

int cmp_universal(GEN x, GEN y) a somewhat arbitrary universal comparison function, devoid of sensible mathematical meaning. It is transitive, and returns 0 if and only if gidentical(x,y) is true. Useful to sort and search vectors of arbitrary data.

int cmp_nodata(void *data, GEN x, GEN y). This function is a hack used to pass an existing basic comparison function lacking the data argument, i.e. with prototype int (*cmp)(GEN x, GEN y). Instead of gen_sort(x, NULL, cmp) which may or may not work depending on how your compiler handles typecasts between incompatible function pointers, one should use gen_sort(x, (void*)cmp, cmp_nodata).

Here are a few basic comparison functions, to be used with cmp_nodata:

int ZV_cmp(GEN x, GEN y) compare two ZV, which we assume have the same length (lexicographic order).

int cmp_RgX(GEN x, GEN y) compare two polynomials, which we assume have the same main variable (lexicographic order). The coefficients are compared using gcmp.

int cmp_prime_over_p(GEN x, GEN y) compare two prime ideals, which we assume divide the same prime number. The comparison is ad hoc but orders according to increasing residue degrees.

int cmp_prime_ideal(GEN x, GEN y) compare two prime ideals in the same nf. Orders by increasing primes, breaking ties using cmp_prime_over_p.

Finally a more elaborate comparison function:

int gen_cmp_RgX(void *data, GEN x, GEN y) compare two polynomials, ordering first by increasing degree, then according to the coefficient comparison function:

int (*cmp_coeff)(GEN,GEN) = (int(*)(GEN,GEN)) data;

167
8.9 Divisibility, Euclidean division.

\[
\text{GEN gdivexact(GEN x, GEN y) returns the quotient } x/y, \text{ assuming } y \text{ divides } x. \text{ Not stack clean if } y = 1 (\text{we return } x, \text{ not a copy}).
\]

\[
\text{int gdvd(GEN x, GEN y) returns 1 (true) if } y \text{ divides } x, 0 \text{ otherwise.}
\]

\[
\text{GEN gdiventres(GEN x, GEN y) creates a 2-component vertical vector whose components are the true Euclidean quotient and remainder of } x \text{ and } y.
\]

\[
\text{GEN gdivent}[z](GEN x, GEN y[, GEN z]) \text{ yields the true Euclidean quotient of } x \text{ and the t_INT or t_POL } y.
\]

\[
\text{GEN gdiventsg(long s, GEN y[, GEN z]), as gdivent except that } x \text{ is a long.}
\]

\[
\text{GEN gdiventgs}[z](GEN x, long s[, GEN z]), as gdivent except that } y \text{ is a long.}
\]

\[
\text{GEN gmod}[z](GEN x, GEN y[, GEN z]) \text{ yields the remainder of } x \text{ modulo the t_INT or t_POL } y.
\]

\[
\text{A t_REAL or t_FRAC } y \text{ is also allowed, in which case the remainder is the unique real } r \text{ such that } 0 \leq r < |y| \text{ and } y = qx + r \text{ for some (in fact unique) integer } q.
\]

\[
\text{GEN gmodsg(long s, GEN y[, GEN z]) as gmod, except } x \text{ is a long.}
\]

\[
\text{GEN gmodgs(GEN x, long s[, GEN z]) as gmod, except } y \text{ is a long.}
\]

\[
\text{GEN gdivmod(GEN x, GEN y, GEN *r) If } r \text{ is not equal to NULL or ONLY_Rem, creates the (false) Euclidean quotient of } x \text{ and } y, \text{ and puts (the address of) the remainder into *r. If } r \text{ is equal to NULL, do not create the remainder, and if } r \text{ is equal to ONLY_Rem, create and output only the remainder. The remainder is created after the quotient and can be disposed of individually with a cgiv(r).}
\]

\[
\text{GEN poldivrem(GEN x, GEN y, GEN *r) same as gdivmod but specifically for t_POLs } x \text{ and } y, \text{ not necessarily in the same variable. Either of } x \text{ and } y \text{ may also be scalars, treated as polynomials of degree 0.}
\]

\[
\text{GEN gdeuc(GEN x, GEN y) creates the Euclidean quotient of the t_POLs } x \text{ and } y. \text{ Either of } x \text{ and } y \text{ may also be scalars, treated as polynomials of degree 0.}
\]

\[
\text{GEN grem(GEN x, GEN y) creates the Euclidean remainder of the t_POL } x \text{ divided by the t_POL } y. \text{ Either of } x \text{ and } y \text{ may also be scalars, treated as polynomials of degree 0.}
\]

\[
\text{GEN gdivround(GEN x, GEN y) if } x \text{ and } y \text{ are t_INT, as diviiround. Operate componentwise if } x \text{ is a t_COL, t_VEC or t_MAT. Otherwise as gdivent.}
\]

\[
\text{GEN centermod_i(GEN x, GEN y, GEN y2), as centermodii, componentwise.}
\]

\[
\text{GEN centermod(GEN x, GEN y), as centermod_i, except that } y2 \text{ is computed (and left on the stack for efficiency).}
\]

\[
\text{GEN ginvmmod(GEN x, GEN y) creates the inverse of } x \text{ modulo } y \text{ when it exists. } y \text{ must be of type t_INT (in which case } x \text{ is of type t_INT) or t_POL (in which case } x \text{ is either a scalar type or a t_POL).}
\]
8.10 GCD, content and primitive part.

8.10.1 Generic.

GEN resultant(GEN x, GEN y) creates the resultant of the t_POLs x and y computed using Sylvester’s matrix (inexact inputs), a modular algorithm (inputs in \( \mathbb{Q}[X] \)) or the subresultant algorithm, as optimized by Lazard and Ducos. Either of x and y may also be scalars (treated as polynomials of degree 0).

GEN ggcd(GEN x, GEN y) creates the GCD of x and y.

GEN g lcm(GEN x, GEN y) creates the LCM of x and y.

GEN g bezout(GEN x, GEN y, GEN *u, GEN *v) returns the GCD of x and y, and puts (the addresses of) objects u and v such that \( ux + vy = \gcd(x, y) \) into *u and *v.

GEN subres ext(GEN x, GEN y, GEN *U, GEN *V) returns the resultant of x and y, and puts (the addresses of) polynomials u and v such that \( ux + vy = \text{Res}(x, y) \) into *U and *V.

GEN content(GEN x) returns the GCD of all the components of x.

GEN primitive_part(GEN x, GEN *c) sets c to content(x) and returns the primitive part \( x / c \). A trivial content is set to NULL.

GEN primpart(GEN x) as above but the content is lost. (For efficiency, the content remains on the stack.)

8.10.2 Over the rationals.

long Q_pval(GEN x, GEN p) valuation at the t_INT p of the t_INT or t_FRAC x.

long Q_pvalrem(GEN x, GEN p, GEN *r) returns the valuation e at the t_INT p of the t_INT or t_FRAC x. The quotient \( x/p^e \) is returned in *r.

GEN Q_abs(GEN x) absolute value of the t_INT or t_FRAC x.

GEN Q_abs_shallow(GEN x) x being a t_INT or a t_FRAC, returns a shallow copy of |x|, in particular returns x itself when \( x \geq 0 \), and gneg(x) otherwise.

GEN Q_gcd(GEN x, GEN y) gcd of the t_INT or t_FRAC x and y.

In the following functions, arguments belong to a \( M \otimes \mathbb{Z} \mathbb{Q} \) for some natural \( \mathbb{Z} \)-module \( M \), e.g. multivariate polynomials with integer coefficients (or vectors/matrices recursively built from such objects), and an element of \( M \) is said to be \( \text{integral} \). We are interested in contents, denominators, etc. with respect to this canonical integral structure; in particular, contents belong to \( \mathbb{Q} \), denominators to \( \mathbb{Z} \). For instance the \( \mathbb{Q} \)-content of \( (1/2)xy \) is \( (1/2) \), and its \( \mathbb{Q} \)-denominator is 2, whereas content would return \( y/2 \) and denom 1.

GEN Q_content(GEN x) the \( \mathbb{Q} \)-content of x

GEN Q_denom(GEN x) the \( \mathbb{Q} \)-denominator of x. Shallow function.

GEN Q_primitive_part(GEN x, GEN *c) sets c to the \( \mathbb{Q} \)-content of x and returns \( x / c \), which is integral.

GEN Q_primpart(GEN x) as above but the content is lost. (For efficiency, the content remains on the stack.)
GEN Q_remove_denom(GEN x, GEN *ptd) sets d to the Q-denominator of x and returns x * d, which is integral. Shallow function.

GEN Q_div_to_int(GEN x, GEN c) returns x / c, assuming c is a rational number (t_INT or t_FRAC) and the result is integral.

GEN Q_mul_to_int(GEN x, GEN c) returns x * c, assuming c is a rational number (t_INT or t_FRAC) and the result is integral.

GEN Q_muli_to_int(GEN x, GEN d) returns x * c, assuming c is a t_INT and the result is integral.

GEN mul_content(GEN cx, GEN cy) cx and cy are as set by primitive_part: either a GEN or NULL representing the trivial content 1. Returns their product (either a GEN or NULL).

GEN mul_denom(GEN dx, GEN dy) dx and dy are as set by Q_remove_denom: either a t_INT or NULL representing the trivial denominator 1. Returns their product (either a t_INT or NULL).

8.11 Generic arithmetic operators.

8.11.1 Unary operators.

GEN gneg[z](GEN x[, GEN z]) yields −x.

GEN gneg_i(GEN x) shallow function yielding −x.

GEN gabs[z](GEN x[, GEN z]) yields |x|.

GEN gsqr(GEN x) creates the square of x.

GEN ginv(GEN x) creates the inverse of x.

8.11.2 Binary operators.

Let “op” be a binary operation among

op=add: addition (x + y).

op=sub: subtraction (x - y).

op=mul: multiplication (x * y).

op=div: division (x / y).

The names and prototypes of the functions corresponding to op are as follows:

GEN gop(GEN x, GEN y)

GEN gopgs(GEN x, long s)

GEN gopsg(long s, GEN y)

Explicitly

GEN gadd(GEN x, GEN y), GEN gaddgs(GEN x, long s), GEN gaddsg(GEN s, GEN x)

GEN gmul(GEN x, GEN y), GEN gmulgs(GEN x, long s), GEN gmulsG(GEN s, GEN x)

GEN gsub(GEN x, GEN y), GEN gsubgs(GEN x, long s), GEN gsubsg(GEN s, GEN x)
GEN gdiv(GEN x, GEN y), GEN gdivgs(GEN x, long s), GEN gdivsg(GEN s, GEN x)

GEN gpow(GEN x, GEN y, long l) creates $x^y$. If $y$ is a t_INT, return powgi(x,y) (the precision 1 is not taken into account). Otherwise, the result is $\exp(y \times \log(x))$ where exact arguments are converted to floats of precision 1 in case of need; if there is no need, for instance if $x$ is a t_REAL, $l$ is ignored. Indeed, if $x$ is a t_REAL, the accuracy of $\log x$ is determined from the accuracy of $x$, it is no problem to multiply by $y$, even if it is an exact type, and the accuracy of the exponential is determined, exactly as in the case of the initial $\log x$.

GEN gpowgs(GEN x, long n) creates $x^n$ using binary powering. To treat the special case $n = 0$, we consider gpowgs as a series of gmul, so we follow the rule of returning result which is as exact as possible given the input. More precisely, we return  • gen_1 if $x$ has type t_INT, t_REAL, t_FRAC, or t_PADIC
  • Mod(1,N) if $x$ is a t_INTMOD modulo $N$.
  • gen_1 for t_COMPLEX, t_QUAD unless one component is a t_INTMOD, in which case we return Mod(1, N) for a suitable $N$ (the gcd of the moduli that appear).
  • FF_1(x) for a t_FFELT.
  • RgX_get_1(x) for a t_POL.
  • qfi_1(x) and qfr_1(x) for t_QFI and t_QFR.
  • the identity permutation for t_VECSMALL.
  • etc. Of course, the only practical use of this routine for $n = 0$ is to obtain the multiplicative neutral element in the base ring (or to treat marginal cases that should be special cased anyway if there is the slightest doubt about what the result should be).

GEN powgi(GEN x, GEN y) creates $x^y$, where $y$ is a t_INT, using left-shift binary powering. The case where $y = 0$ (as all cases where $y$ is small) is handled by gpowgs(x, 0).

In addition we also have the obsolete forms:

void gaddz(GEN x, GEN y, GEN z)
void gsubz(GEN x, GEN y, GEN z)
void gmulz(GEN x, GEN y, GEN z)
void gdivz(GEN x, GEN y, GEN z)
8.12 Generic operators: product, powering, factorback.

GEN divide_conquer_prod(GEN v, GEN (*mul)(GEN,GEN)) v is a vector of objects, which can be “multiplied” using the mul function. Return the “product” of the v[i] using a product tree: by convention return gen_1 if v is the empty vector, a copy of v[1] if it has a single entry; and otherwise apply the function recursively on the vector (twice smaller)

mul(v[1],v[2]), mul(v[3],v[4]), ...

Only requires that mul is an associative binary operator, which need not correspond to a true multiplication. D is meant to encode an arbitrary evaluation context, set it to NULL in simple cases where you do not need this. Leaves some garbage on stack, but suitable for gerepileupto if mul is.

To describe the following functions, we use the following private typedefs to simplify the description:

typedef (*F0)(void *);
typedef (*F1)(void *, GEN);
typedef (*F2)(void *, GEN, GEN);

They correspond to generic functions with one and two arguments respectively (the void* argument provides some arbitrary evaluation context).

GEN divide_conquer_assoc(GEN v, void *D, F2 op) general version of divide_conquer_prod. Given two objects x,y, assume that op(D, x, y) implements an associative binary operator. If v has k entries, return

v[1] op v[2] op ... op v[k];

returns gen_1 if k = 0 and a copy of v[1] if k = 1.

GEN gen_pow(GEN x, GEN n, void *D, F1 sqr, F2 mul) n > 0 a t_INT, returns x^n; mul(D, x, y) implements the multiplication in the underlying monoid; sqr is a (presumably optimized) shortcut for mul(D, x, x).

GEN gen_powu(GEN x, ulong n, void *D, F1 sqr, F2 mul) n > 0, returns x^n. See gen_pow.

GEN gen_pow_i(GEN x, GEN n, void *E, F1 sqr, F2 mul) internal variant of gen_pow, not memory-clean.

GEN gen_powu_i(GEN x, ulong n, void *E, F1 sqr, F2 mul) internal variant of gen_powu, not memory-clean.

GEN gen_pow_fold(GEN x, GEN n, void *D, F1 sqr, F1 msqr) variant of gen_pow, where mul is replaced by msqr, with msqr(D, y) returning xy^2. In particular D must implicitly contain x.

GEN gen_pow_fold_i(GEN x, GEN n, void *E, F1 sqr, F1 msqr) internal variant of the function gen_pow_fold, not memory-clean.

GEN gen_powu_fold(GEN x, ulong n, void *D, F1 sqr, F1 msqr), see gen_pow_fold.

GEN gen_powu_fold_i(GEN x, ulong n, void *E, F1 sqr, F1 msqr) see gen_pow_fold_i.

GEN gen_powers(GEN x, long n, long usesqr, void *D, F1 sqr, F2 mul, F0 one) returns [x^0,...,x^n] as a t_VEC; mul(D, x, y) implements the multiplication in the underlying monoid; sqr is a (presumably optimized) shortcut for mul(D, x, x); one returns the monoid unit. The flag usesqr should be set to 1 if squaring are faster than multiplication by x.
GEN gen_factorback(GEN L, GEN e, F2 mul, F2 pow, void *D) generic form of factorback. The pair \([L, e]\) is of the form

- \([fa, \text{NULL}]\), \(fa\) a two-column factorization matrix: expand it.
- \([v, \text{NULL}]\), \(v\) a vector of objects: return their product.
- or \([v, e]\), \(v\) a vector of objects, \(e\) a vector of integral exponents: return the product of the \(v[i]e[i]\).

\(\text{mul}(D, x, y)\) and \(\text{pow}(D, x, n)\) return \(xy\) and \(x^n\) respectively.

8.13 Matrix and polynomial norms.

This section concerns only standard norms of \(\mathbb{R}\) and \(\mathbb{C}\) vector spaces, not algebraic norms given by the determinant of some multiplication operator. We have already seen type-specific functions like \(\text{ZM\_supnorm}\) or \(\text{RgM\_fpnorml2}\) and limit ourselves to generic functions assuming nothing about their GEN argument; these functions allow the following scalar types: \(\text{t\_INT}\), \(\text{t\_FRAC}\), \(\text{t\_REAL}\), \(\text{t\_COMPLEX}\), \(\text{t\_QUAD}\) and are defined recursively (in terms of norms of their components) for the following “container” types: \(\text{t\_POL}\), \(\text{t\_VEC}\), \(\text{t\_COL}\) and \(\text{t\_MAT}\). They raise an error if some other type appears in the argument.

GEN gnorml2(GEN x) The norm of a scalar is the square of its complex modulus, the norm of a recursive type is the sum of the norms of its components. For polynomials, vectors or matrices of complex numbers one recovers the square of the usual \(L^2\) norm. In most applications, the missing square root computation can be skipped.

GEN gnorml1(GEN x, long prec) The norm of a scalar is its complex modulus, the norm of a recursive type is the sum of the norms of its components. For polynomials, vectors or matrices of complex numbers one recovers the usual \(L^1\) norm. One must include a real precision \(\text{prec}\) in case the inputs include \(\text{t\_COMPLEX}\) or \(\text{t\_QUAD}\) with exact rational components: a square root must be computed and we must choose an accuracy.

GEN gnorml1_fake(GEN x) as gnorml1, except that the norm of a \(\text{t\_QUAD}\) \(x + wy\) or \(\text{t\_COMPLEX}\) \(x + Iy\) is defined as \(|x| + |y|\), where we use the ordinary real absolute value. This is still a norm of \(\mathbb{R}\) vector spaces, which is easier to compute than gnorml1 and can often be used in its place.

GEN gsupnorm(GEN x, long prec) The norm of a scalar is its complex modulus, the norm of a recursive type is the max of the norms of its components. A precision \(\text{prec}\) must be included for the same reason as in gnorml1.

void gsupnorm_aux(GEN x, GEN *m, GEN *m2, long prec) Low-level function underlying gsupnorm, used as follows:

\[
\text{GEN m = NULL, m2 = NULL;}
\text{gsupnorm\_aux(x, &m, &m2);} \]

After the call, the sup norm of \(x\) is the min of \(m\) and the square root of \(m2\); one or both of \(m\), \(m2\) may be NULL, in which case it must be omitted. You may initially set \(m\) and \(m2\) to non-NULL values, in which case, the above procedure yields the max of (the initial) \(m\), the square root of (the initial) \(m2\), and the sup norm of \(x\).

The strange interface is due to the fact that \(|z|^2\) is easier to compute than \(|z|\) for a \(\text{t\_QUAD}\) or \(\text{t\_COMPLEX}\) \(z\): \(m2\) is the max of those \(|z|^2\), and \(m\) is the max of the other \(|z|\).
8.14 Substitution and evaluation.

GEN gsubst(GEN x, long v, GEN y) substitutes the object y into x for the variable number v.

GEN poleval(GEN q, GEN x) evaluates the \texttt{t\_POL} or \texttt{t\_RFRAC} \( q \) at \( x \). For convenience, a \texttt{t\_VEC} or \texttt{t\_COL} is also recognized as the \texttt{t\_POL gtovecrev(q)}.

GEN RgX_RgM_eval(GEN q, GEN x) evaluates the \texttt{t\_POL} \( q \) at the square matrix \( x \).

GEN RgX_RgMV_eval(GEN f, GEN V) returns the evaluation \( f(x) \), assuming that \( V \) was computed by \texttt{FpXQ\_powers(x,n)} for some \( n > 1 \).

GEN RgX_RgM_eval_col(GEN q, GEN x, long c) evaluates the \texttt{t\_POL} \( q \) at the square matrix \( x \) but only returns the \( c \)-th column of the result.

GEN qfeval(GEN q, GEN x) evaluates the quadratic form \( q \) (symmetric matrix) at \( x \) (column vector of compatible dimensions).

GEN qfevalb(GEN q, GEN x, GEN y) evaluates the polar bilinear form associated to the quadratic form \( q \) (symmetric matrix) at \( x, y \) (column vectors of compatible dimensions).

GEN hqfeval(GEN q, GEN x) evaluates the Hermitian form \( q \) (a Hermitian complex matrix) at \( x \).

GEN qf_apply_RgM(GEN q, GEN M) \( q \) is a symmetric \( n \times n \) matrix, \( M \) an \( n \times k \) matrix, return \( M'qM \).

GEN qf_apply_ZM(GEN q, GEN M) as above assuming that both \( q \) and \( M \) have integer entries.
Chapter 9:
Miscellaneous mathematical functions

9.1 Fractions.

GEN absfrac(GEN x) returns the absolute value of the t_FRAC $x$.

GEN absfrac_shallow(GEN x) $x$ being a t_FRAC, returns a shallow copy of $|x|$, in particular returns $x$ itself when $x \geq 0$, and $\text{gneg}(x)$ otherwise.

GEN sqrfrac(GEN x) returns the square of the t_FRAC $x$.

9.2 Complex numbers.

GEN imag(GEN x) returns a copy of the imaginary part of $x$.

GEN real(GEN x) returns a copy of the real part of $x$. If $x$ is a t_QUAD, returns the coefficient of 1 in the “canonical” integral basis $(1, \omega)$.

The last two functions are shallow, and not suitable for \texttt{gerepileupto}:

GEN imag_i(GEN x) as \texttt{gimag}, returns a pointer to the imaginary part. GEN real_i(GEN x) as \texttt{greal}, returns a pointer to the real part.

GEN mulreal(GEN x, GEN) returns the real part of $xy$; $x, y$ have type t_INT, t_FRAC, t_REAL or t_COMPLEX. See also \texttt{RgM_mulreal}.

GEN cxnorm(GEN x) norm of the t_COMPLEX $x$ (modulus squared).

GEN cxexpm1(GEN x) returns $\exp(x) - 1$, for a t_COMPLEX $x$.

9.3 Quadratic numbers and binary quadratic forms.

GEN quad_disc(GEN x) returns the discriminant of the t_QUAD $x$.

GEN quadnorm(GEN x) norm of the t_QUAD $x$.

GEN qfb_disc(GEN x) returns the discriminant of the t_QFI or t_QFR $x$.

GEN qfb_disc3(GEN x, GEN y, GEN z) returns $y^2 - 4xz$ assuming all inputs are t_INTs. Not stack-clean.
9.4 Polynomials.

**GEN truecoeff(GEN x, long n)** returns `polcoeff0(x, n, -1)`, i.e. the coefficient of the term of degree `n` in the main variable.

**GEN polcoeff_i(GEN x, long n, long v)** is an internal shallow function. Rewrite `x` as a Laurent polynomial in the variable `v` and returns its coefficient of degree `n` (`gen_0` if this falls outside the coefficient array). Allow `t_POL`, `t_SER`, `t_RFRAC` and scalars.

**long degree(GEN x)** returns `poldegree(x, -1)`, the degree of `x` with respect to its main variable, with the usual meaning if the leading coefficient of `x` is non-zero. If the sign of `x` is 0, this function always returns −1. Otherwise, we return the index of the leading coefficient of `x`, i.e. the coefficient of largest index stored in `x`. For instance the “degrees” of

```
0. E-38 * x^4 + 0.E-19 * x + 1
Mod(0,2) * x^0 \ sign is 0 !
```

are 4 and −1 respectively.

**long degpol(GEN x)** is a simple macro returning `lg(x) - 3`. This is the degree of the `t_POL` `x` with respect to its main variable, if its leading coefficient is non-zero (a rational 0 is impossible, but an inexact 0 is allowed, as well as an exact modular 0, e.g. `Mod(0,2)`). If `x` has no coefficients (rational 0 polynomial), its length is 2 and we return the expected −1.

**GEN characteristic(GEN x)** returns the characteristic of the base ring over which the polynomial is defined (as defined by `t_INTMOD` and `t_FFELT` components). The function raises an exception if incompatible primes arise from `t_FFELT`s and `t_PADIC`s. Shallow function.

**GEN residual_characteristic(GEN x)** returns a kind of “residual characteristic” of the base ring over which the polynomial is defined. This is defined as the gcd of all moduli `t_INTMOD`s occurring in the structure, as well as primes `p` arising from `t_PADIC`s or `t_FFELT`s. The function raises an exception if incompatible primes arise from `t_FFELT`s and `t_PADIC`s. Shallow function.

**GEN resultant(GEN x, GEN y)** resultant of `x` and `y`, with respect to the main variable of highest priority. Uses either the subresultant algorithm (generic case), a modular algorithm (inputs in \( \mathbb{Q}[X] \)) or Sylvester’s matrix (inexact inputs).

**GEN resultant2(GEN x, GEN y)** resultant of `x` and `y`, with respect to the main variable of highest priority. Computes the determinant of Sylvester’s matrix.

**GEN resultant_all(GEN u, GEN v, GEN *sol)** returns `resultant(x, y)`. If `sol` is not NULL, sets it to the last non-constant remainder in the polynomial remainder sequence if such a sequence was computed, and to `gen_0` otherwise (e.g. polynomials of degree 0, `u, v` in \( \mathbb{Q}[X] \)).

**GEN cleanroots(GEN x, long prec)** returns the complex roots of the complex polynomial `x` (with coefficients `t_INT`, `t_FRAC`, `t_REAL` or `t_COMPLEX` of the above). The roots are returned as `t_REAL` or `t_COMPLEX` of `t_REAL`s of precision `prec` (guaranteeing a non-0 imaginary part). See \( \text{QX\_complex\_roots} \).

**GEN polmod_to_embed(GEN x, long prec)** return the vector of complex embeddings of the `t_POLMOD` `x` (with complex coefficients). Shallow function, simple complex variant of `conjvec`. 

---

176
9.5 Power series.

GEN derivser(GEN x) returns the derivative of the t_SER x with respect to its main variable.

GEN integser(GEN x) returns the primitive of the t_SER x with respect to its main variable.

GEN truecoeff(GEN x, long n) returns polcoeff0(x,n, -1), i.e. the coefficient of the term of degree n in the main variable.

GEN ser_unscale(GEN P, GEN h) returns $P(hx)$, not memory clean.

GEN ser_normalize(GEN x) divide x by its “leading term” so that the series is either 0 or equal to $t^n(1 + O(t))$. Shallow function if the “leading term” is 1.

9.6 Functions to handle t_FFELT.

These functions define the public interface of the t_FFELT type to use in generic functions. However, in specific functions, it is better to use the functions class FpXQ and/or Flxq as appropriate.

GEN FF_p(GEN a) returns the characteristic of the definition field of the t_FFELT element a.

long FF_f(GEN a) returns the dimension of the definition field over its prime field; the cardinality of the dimension field is thus $p^f$.

GEN FF_p_i(GEN a) shallow version of FF_p.

GEN FF_q(GEN a) returns the cardinal of the definition field of the t_FFELT element a.

GEN FF_mod(GEN a) returns the polynomial (with reduced t_INT coefficients) defining the finite field, in the variable used to display a.

GEN FF_to_FpXQ(GEN a) converts the t_FFELT a to a polynomial $P$ with reduced t_INT coefficients such that $a = P(g)$ where $g$ is the generator of the finite field returned by ffgen, in the variable used to display $g$.

GEN FF_to_FpXQ_i(GEN a) shallow version of FF_to_FpXQ.

GEN FF_to_F2xq(GEN a) converts the t_FFELT a to a F2x $P$ such that $a = P(g)$ where $g$ is the generator of the finite field returned by ffgen, in the variable used to display $g$. This only work if the characteristic is 2.

GEN FF_to_F2xq_i(GEN a) shallow version of FF_to_F2xq.

GEN FF_to_Flxq(GEN a) converts the t_FFELT a to a Flx $P$ such that $a = P(g)$ where $g$ is the generator of the finite field returned by ffgen, in the variable used to display $g$. This only work if the characteristic is small enough.

GEN FF_to_Flxq_i(GEN a) shallow version of FF_to_Flxq.

GEN p_to_FF(GEN p, long v) returns a t_FFELT equal to 1 in the finite field $\mathbb{Z}/p\mathbb{Z}$. Useful for generic code that wants to handle (inefficiently) $\mathbb{Z}/p\mathbb{Z}$ as if it were not a prime field.

GEN FF_1(GEN a) returns the unity in the definition field of the t_FFELT element a.

GEN FF_zero(GEN a) returns the zero element of the definition field of the t_FFELT element a.

int FF_equal0(GEN a), int FF_equal1(GEN a), int FF_equalm1(GEN a) returns 1 if the t_FFELT a is equal to 0 (resp. 1, resp. $-1$) else 0.
int FF_equal(GEN a, GEN b) return 1 if the t_FFELT a and b have the same definition field and are equal, else 0.

int FF_samefield(GEN a, GEN b) return 1 if the t_FFELT a and b have the same definition field, else 0.

int Rg_is_FF(GEN c, GEN *ff) to be called successively on many objects, setting *ff = NULL (unset) initially. Returns 1 as long as c is a t_FFELT defined over the same field as *ff (setting *ff = c if unset), and 0 otherwise.

int RgC_is_FFC(GEN x, GEN *ff) apply Rg_is_FF successively to all components of the t_VEC or t_COL x. Return 0 if one call fails, and 1 otherwise.

int RgM_is_FFM(GEN x, GEN *ff) apply Rg_is_FF to all components of the t_MAT. Return 0 if one call fails, and 1 otherwise.

GEN FF_add(GEN a, GEN b) returns \(a + b\) where a and b are t_FFELT having the same definition field.

GEN FF_Z_add(GEN a, GEN x) returns \(a + x\), where a is a t_FFELT, and x is a t_INT, the computation being performed in the definition field of a.

GEN FF_Q_add(GEN a, GEN x) returns \(a + x\), where a is a t_FFELT, and x is a t_RFRAC, the computation being performed in the definition field of a.

GEN FF_sub(GEN a, GEN b) returns \(a - b\) where a and b are t_FFELT having the same definition field.

GEN FF_mul(GEN a, GEN b) returns \(ab\) where a and b are t_FFELT having the same definition field.

GEN FF_Z_mul(GEN a, GEN b) returns \(ab\), where a is a t_FFELT, and b is a t_INT, the computation being performed in the definition field of a.

GEN FF_div(GEN a, GEN b) returns \(a/b\) where a and b are t_FFELT having the same definition field.

GEN FF_neg(GEN a) returns \(-a\) where a is a t_FFELT.

GEN FF_neg_i(GEN a) shallow function returning \(-a\) where a is a t_FFELT.

GEN FF_inv(GEN a) returns \(a^{-1}\) where a is a t_FFELT.

GEN FF_sqr(GEN a) returns \(a^2\) where a is a t_FFELT.

GEN FF_pow(GEN x, GEN n) returns \(a^n\) where a is a t_FFELT and n is a t_INT.

GEN FF_Z_Z_muldiv(GEN a, GEN x, GEN y) returns \(ay/z\), where a is a t_FFELT, and x and y are t_INT, the computation being performed in the definition field of a.

GEN Z_FF_div(GEN x, GEN a) returns \(x/a\) where a is a t_FFELT, and x is a t_INT, the computation being performed in the definition field of a.

GEN FF_norm(GEN a) returns the norm of the t_FFELT a with respect to its definition field.

GEN FF_trace(GEN a) returns the trace of the t_FFELT a with respect to its definition field.

GEN FF_conjvec(GEN a) returns the vector of conjugates \([a, a^p, a^{p^2}, \ldots, a^{p^{n-1}}]\) where the t_FFELT a belong to a field with \(p^n\) elements.
GEN FF_charpoly(GEN a) returns the characteristic polynomial of the t_FFELT a with respect to its definition field.

GEN FF_minpoly(GEN a) returns the minimal polynomial of the t_FFELT a.

GEN FF_sqrt(GEN a) returns an t_FFELT b such that $a = b^2$ if it exist, where a is a t_FFELT.

long FF_issquareall(GEN x, GEN *pt) returns 1 if x is a square, and 0 otherwise. If x is indeed a square, set pt to its square root.

long FF_issquare(GEN x) returns 1 if x is a square and 0 otherwise.

long FF_ispower(GEN x, GEN K, GEN *pt) Given K a positive integer, returns 1 if x is a K-th power, and 0 otherwise. If x is indeed a K-th power, set pt to its K-th root.

GEN FF_sqrtn(GEN a, GEN n, GEN *zn) returns an n-th root of a if it exist. If zn is non-NULL set it to a primitive n-th root of the unity.

GEN FF_log(GEN a, GEN g, GEN o) the t_FFELT g being a generator for the definition field of the t_FFELT a, returns a t_INT e such that $a^e = g$. If e does not exists, the result is currently undefined. If o is not NULL it is assumed to be a factorization of the multiplicative order of g (as set by FF_primroot).

GEN FF_order(GEN a, GEN o) returns the order of the t_FFELT a. If o is non-NULL, it is assumed that o is a multiple of the order of a.

GEN FF_primroot(GEN a, GEN *o) returns a generator of the multiplicative group of the definition field of the t_FFELT a. If o is not NULL, set it to the factorization of the order of the primitive root (to speed up FF_log).

GEN FFX_factor(GEN f, GEN a) returns the factorization of the univariate polynomial f over the definition field of the t_FFELT a. The coefficients of f must be of type t_INT, t_INTMOD or t_FFELT and compatible with a.

GEN FFX_roots(GEN f, GEN a) returns the roots (t_FFELT) of the univariate polynomial f over the definition field of the t_FFELT a. The coefficients of f must be of type t_INT, t_INTMOD or t_FFELT and compatible with a.

GEN FFM_FFC_mul(GEN M, GEN C, GEN ff) returns the product of the matrix M (t_MAT) and the column vector C (t_COL) over the finite field given by ff (t_FFELT).

GEN FFM_ker(GEN M, GEN ff) returns the kernel of the t_MAT M defined over the finite field given by the t_FFELT ff (obtained by RgM_is_FFM(M, &ff)).

GEN FFM_det(GEN M, GEN ff)
GEN FFM_image(GEN M, GEN ff)
GEN FFM_inv(GEN M, GEN ff)
GEN FFM_mul(GEN M, GEN N, GEN ff) returns the product of the matrices M and N (t_MAT) over the finite field given by ff (t_FFELT).

long FFM_rank(GEN M, GEN ff)
9.7 Transcendental functions.

The following two functions are only useful when interacting with gp, to manipulate its internal default precision (expressed as a number of decimal digits, not in words as used everywhere else):

long getrealprecision(void) returns realprecision.

long setrealprecision(long n, long *prec) sets the new realprecision to n, which is returned. As a side effect, set prec to the corresponding number of words ndec2prec(n).

9.7.1 Transcendental functions with t_REAL arguments.

In the following routines, \( x \) is assumed to be a t_REAL and the result is a t_REAL (sometimes a t_COMPLEX with t_REAL components), with the largest accuracy which can be deduced from the input. The naming scheme is inconsistent here, since we sometimes use the prefix mp even though t_INT inputs are forbidden:

GEN sqrtr(GEN x) returns the square root of \( x \).

GEN sqrtnr(GEN x, long n) returns the \( n \)-th root of \( x \), assuming \( n \geq 1 \) and \( x > 0 \). Not stack clean.

GEN mpcos[z](GEN x[, GEN z]) returns \( \cos(x) \).

GEN mpsin[z](GEN x[, GEN z]) returns \( \sin(x) \).

GEN mplog[z](GEN x[, GEN z]) returns \( \log(x) \). We must have \( x > 0 \) since the result must be a t_REAL. Use glog for the general case, where you want such computations as \( \log(-1) = i \).

GEN mpexp[z](GEN x[, GEN z]) returns \( \exp(x) \).

GEN mpexpm1(GEN x) returns \( \exp(x) - 1 \), but is more accurate than subrs(mpexp(x), 1), which suffers from catastrophic cancellation if \( |x| \) is very small.

void mpsincosm1(GEN x, GEN *s, GEN *c) sets \( s \) and \( c \) to \( \sin(x) \) and \( \cos(x) - 1 \) respectively, where \( x \) is a t_REAL; the latter is more accurate than subrs(mpcos(y), 1), which suffers from catastrophic cancellation if \( |x| \) is very small.

GEN mpveceint1(GEN C, GEN eC, long n) as veceint1; assumes that \( C > 0 \) is a t_REAL and that eC is NULL or mpexp(C).

GEN mpeint1(GEN x, GEN expx) returns eint1(x), for a t_REAL \( x \geq 0 \), assuming that expx is mpexp(x).

GEN szeta(long s, long prec) returns the value of Riemann’s zeta function at the (possibly negative) integer \( s \neq 1 \), in relative accuracy prec.

GEN mplambertW(GEN y) solution \( x \) of the implicit equation \( x \exp(x) = y \), for \( y > 0 \) a t_REAL.

Useful low-level functions which disregard the sign of \( x \):

GEN sqrtr_abs(GEN x) returns \( \sqrt{|x|} \) assuming \( x \neq 0 \).

GEN exp1r_abs(GEN x) returns \( \exp(|x|) - 1 \), assuming \( x \neq 0 \).

GEN logr_abs(GEN x) returns \( \log(|x|) \), assuming \( x \neq 0 \).

A few variants on sin and cos:
void mpsincos(GEN x, GEN *s, GEN *c) sets \( s \) and \( c \) to \( \sin(x) \) and \( \cos(x) \) respectively, where \( x \) is a \texttt{t_REAL}

\texttt{GEN explr(GEN x)} returns \( \exp(ix) \), where \( x \) is a \texttt{t_REAL}. The return type is \texttt{t_COMPLEX} unless the imaginary part is equal to 0 to the current accuracy (its sign is 0).

\texttt{GEN expIxy(GEN x, GEN y, long prec)} returns \( \exp(ify) \). Efficient when \( x \) is real and \( y \) pure imaginary.

void gsinicos(GEN x, GEN *s, GEN *c, long prec) general case.

A generalization of \texttt{affrr\_fixlg}

\texttt{GEN affc\_fixlg(GEN x, GEN res)} assume \( res \) was allocated using \texttt{cgetc}, and that \( x \) is either a \texttt{t_REAL} or a \texttt{t_COMPLEX} with \texttt{t_REAL} components. Assign \( x \) to \( res \), first shortening the components of \( res \) if needed (in a \texttt{gerepile-safe} way). Further convert \( res \) to a \texttt{t_REAL} if \( x \) is a \texttt{t_REAL}.

\texttt{GEN trans\_eval(const char *fun, GEN (*f) (GEN, long), GEN x, long prec)} evaluates transcendental function \( f \) (named "\texttt{fun}" at the argument \( x \) and precision \( prec \). This is a quick way to implement a transcendental function to be made available under GP, starting from a \texttt{C} function handling only \texttt{t_REAL} and \texttt{t_COMPLEX} arguments. This routine first converts \( x \) to a suitable type:

- \texttt{t_INT/t_FRAC} to \texttt{t_REAL} of precision \( prec \), \texttt{t_QUAD} to \texttt{t_REAL} or \texttt{t_COMPLEX} of precision \( prec \).
- \texttt{t_POLMOD} to a \texttt{t_COL} of complex embeddings (as in \texttt{conjvec})

Then evaluates the function at \texttt{t_VEC}, \texttt{t_COL}, \texttt{t_MAT} arguments coefficientwise.

9.7.2 Transcendental functions with \texttt{t_PADIC} arguments.

\texttt{GEN Qp\_exp(GEN x)} shortcut for \texttt{gexp(x, /*ignored*/prec)}

\texttt{GEN Qp\_gamma(GEN x)} shortcut for \texttt{ggamma(x, /*ignored*/prec)}

\texttt{GEN Qp\_log(GEN x)} shortcut for \texttt{glog(x, /*ignored*/prec)}

\texttt{GEN Qp\_sqrt(GEN x)} shortcut for \texttt{gsqrt(x, /*ignored*/prec)} Return NULL if \( x \) is not a square.

\texttt{GEN Qp\_sqrtn(GEN x, GEN n, GEN *z)} shortcut for \texttt{gsqrtn(x, n, z, /*ignored*/prec)}. Return NULL if \( x \) is not an \texttt{n}-th power.

9.7.3 Cached constants.

The cached constant is returned at its current precision, which may be larger than \( prec \). One should always use the \texttt{mp\_xxx} variant: \texttt{mppi}, \texttt{mpeuler}, or \texttt{mplog2}.

\texttt{GEN consteuler(long prec)} precomputes Euler-Mascheroni’s constant at precision \( prec \).

\texttt{GEN constcatalan(long prec)} precomputes Catalan’s constant at precision \( prec \).

\texttt{GEN constpi(long prec)} precomputes \( \pi \) at precision \( prec \).

\texttt{GEN constlog2(long prec)} precomputes \( \log(2) \) at precision \( prec \).

\texttt{void mpbern(long n, long prec)} precomputes the \( n \) even Bernoulli numbers \( B_2, \ldots, B_{2n} \) as \texttt{t_FRAC} or \texttt{t_REALs} of precision \( prec \). For any \( 2 \leq k \leq 2n \), if a floating point approximation of \( B_k \) to accuracy \( prec \) is enough to reconstruct it exactly, a \texttt{t_FRAC} is stored; otherwise a \texttt{t_REAL}.
at the requested accuracy. No more than \( n \) Bernoulli numbers will ever be stored (by \texttt{bernfrac}
or \texttt{bernreal}), unless a subsequent call to \texttt{mpbern} increases the cache. If \texttt{prec} is 0, the \( B_k \) are computed exactly.

The following functions use cached data if \texttt{prec} is smaller than the precision of the cached value; otherwise the newly computed data replaces the old cache.

- \texttt{GEN mppi(long prec)} returns \( \pi \) at precision \texttt{prec}.
- \texttt{GEN Pi2n(long n, long prec)} returns \( 2^n\pi \) at precision \texttt{prec}.
- \texttt{GEN PiI2(long n, long prec)} returns the complex number \( 2\pi i \) at precision \texttt{prec}.
- \texttt{GEN PiI2n(long n, long prec)} returns the complex number \( 2^n\pi i \) at precision \texttt{prec}.
- \texttt{GEN mpeuler(long prec)} returns Euler-Mascheroni’s constant at precision \texttt{prec}.
- \texttt{GEN mpeuler(long prec)} returns Catalan’s number at precision \texttt{prec}.
- \texttt{GEN mplog2(long prec)} returns \( \log 2 \) at precision \texttt{prec}.
- \texttt{GEN bernreal(long i, long prec)} returns the Bernoulli number \( B_i \) as a \texttt{t_REAL} at precision \texttt{prec}. If \texttt{mpbern(n, p)} was called previously with \( n \geq i \) and \( p \geq \text{prec} \), then the cached value is (converted to a \texttt{t_REAL} of accuracy \texttt{prec} then) returned. Otherwise, the missing value is computed. In the latter case, if \( n \geq i \), the cached table is updated.
- \texttt{GEN bernfrac(long i)} returns the Bernoulli number \( B_i \) as a rational number (\texttt{t_FRAC} or \texttt{t_INT}). If a cached table includes \( B_i \) as a rational number, the latter is returned. Otherwise, the missing value is computed. In the latter case, the cached Bernoulli table may be updated.

### 9.8 Permutations

Permutation are represented in two different ways

- \texttt{(perm)} a \texttt{t_VECSMALL} \( p \) representing the bijection \( i \mapsto p[i] \); unless mentioned otherwise, this is the form used in the functions below for both input and output,

- \texttt{(cyc)} a \texttt{t_VEC} of \texttt{t_VECSMALLs} representing a product of disjoint cycles.

- \texttt{GEN identity_perm(long n)} return the identity permutation on \( n \) symbols.
- \texttt{GEN cyclic_perm(long n, long d)} return the cyclic permutation mapping \( i \) to \( i + d \mod n \) in \( S_n \). Assume that \( d \leq n \).
- \texttt{GEN perm_mul(GEN s, GEN t)} multiply \( s \) and \( t \) (composition \( s \circ t \))
- \texttt{GEN perm_conj(GEN s, GEN t)} return \( sts^{-1} \).
- \texttt{int perm_commute(GEN p, GEN q)} return 1 if \( p \) and \( q \) commute, 0 otherwise.
- \texttt{GEN perm_inv(GEN p)} returns the inverse of \( p \).
- \texttt{GEN perm_pow(GEN p, long n)} returns \( p^n \)
- \texttt{GEN cyc_pow_perm(GEN p, long n)} the permutation \( p \) is given as a product of disjoint cycles (\texttt{cyc}); return \( p^n \) (as a \texttt{perm}).
- \texttt{GEN cyc_pow(GEN p, long n)} the permutation \( p \) is given as a product of disjoint cycles (\texttt{cyc}); return \( p^n \) (as a \texttt{cyc}).
GEN perm_cycles(GEN p) return the cyclic decomposition of p.

long perm_order(GEN p) returns the order of the permutation p (as the lcm of its cycle lengths).

GEN vecperm_orbits(GEN p, long n) the permutation p ∈ S_n being given as a product of disjoint cycles, return the orbits of the subgroup generated by p on {1,2,…,n}.

9.9 Small groups.

The small (finite) groups facility is meant to deal with subgroups of Galois groups obtained by galoisinit and thus is currently limited to weakly super-solvable groups.

A group grp of order n is represented by its regular representation (for an arbitrary ordering of its element) in S_n. A subgroup of such group is represented by the restriction of the representation to the subgroup. A small group can be either a group or a subgroup. Thus it is embedded in some S_n, where n is the multiple of the order. Such n is called the domain of the small group. The domain of a trivial subgroup cannot be derived from the subgroup data, so some functions require the subgroup domain as argument.

The small group grp is represented by a t_VEC with two components:

grp[1] is a generating subset [s_1,...,s_g] of grp expressed as a vector of permutation of length n.

grp[2] contains the relative orders [o_1,...,o_g] of the generators grp[1].

See galoisinit for the technical details.

GEN checkgroup(GEN gal, GEN *elts) checks whether gal is a small group or a Galois group. Returns the underlying small group and set elts to the list of elements or to NULL if it is not known.

GEN galois_group(GEN gal) return the underlying small group of the Galois group gal.

GEN cyclicgroup(GEN g, long s) returns the cyclic group with generator g of order s.

GEN trivialgroup(void) returns the trivial group.

GEN dicyclicgroup(GEN g1, GEN g2, long s1, long s2) returns the group with generators g1, g2 with respecting relative orders s1, s2.

GEN abelian_group(GEN v) let v be a t_VECSMALL seen as the SNF of a small abelian group, return its regular representation.

long group_domain(GEN grp) returns the domain of the non-trivial small group grp. Return an error if grp is trivial.

GEN group_elts(GEN grp, long n) returns the list of elements of the small group grp of domain n as permutations.

GEN group_set(GEN grp, long n) returns a F2^v b such that b[i] is set if and only if the small group grp of domain n contains a permutation sending 1 to i.

GEN groupelts_set(GEN elts, long n), where elts is the list of elements of a small group of domain n, returns a F2^v b such that b[i] is set if and only if the small group contains a permutation sending 1 to i.

long group_order(GEN grp) returns the order of the small group grp (which is the product of the relative orders).
long group_isabelian(GEN grp) returns 1 the the small group *grp* is Abelian, else 0.

GEN group_abelianHNF(GEN grp, GEN elts) if *grp* is not Abelian, returns NULL, else returns the HNF matrix of *grp* with respect to the generating family *grp*[1]. If *elts* is no NULL, it must be the list of elements of *grp*.

GEN group_abelianSNF(GEN grp, GEN elts) if *grp* is not Abelian, returns NULL, else returns its cyclic decomposition. If *elts* is no NULL, it must be the list of elements of *grp*.

long group_subgroup_isnormal(GEN G, GEN H), H being a subgroup of the small group G, returns 1 if H is normal in G, else 0.

long group_isA4S4(GEN grp) returns 1 if the small group *grp* is isomorphic to A₄, 2 if it is isomorphic to S₄ and 0 else. This is mainly to deal with the idiosyncrasy of the format.

GEN group_leftcoset(GEN G, GEN g) where G is a small group and g a permutation of the same domain, the the left coset gG as a vector of permutations.

GEN group_rightcoset(GEN G, GEN g) where G is a small group and g a permutation of the same domain, the the right coset Gg as a vector of permutations.

long group_perm_normalize(GEN G, GEN g) where G is a small group and g a permutation of the same domain, return 1 if gG^{-1} = G, else 0.

GEN group_quotient(GEN G, GEN H), where G is a small group and H is a subgroup of G, returns the quotient map G → G/H as an abstract data structure.

GEN quotient_perm(GEN C, GEN g) where C is the quotient map G → G/H for some subgroup H of G and g an element of G, return the image of g by C (i.e. the coset gH).

GEN quotient_group(GEN C, GEN G) where C is the quotient map G → G/H for some normal subgroup H of G, return the quotient group G/H as a small group.

GEN quotient_subgroup_lift(GEN C, GEN H, GEN S) where C is the quotient map G → G/H for some group G normalizing H and S is a subgroup of G/H, return the inverse image of S by C.

GEN group_subgroups(GEN grp) returns the list of subgroups of the small group *grp* as a t_VEC.

GEN subgroups_tableset(GEN S, long n) where S is a vector of subgroups of domain n, returns a table which matches the set of elements of the subgroups against the index of the subgroups.

long tablesset_find_index(GEN tbl, GEN set) searches the set *set* in the table *tbl* and returns its associated index, or 0 if not found.

GEN groupelts_abelian_group(GEN elts) where *elts* is the list of elements of an Abelian small group, returns the corresponding small group.

GEN groupelts_center(GEN elts) where *elts* is the list of elements of a small group, returns the list of elements of the center of the group.

GEN group_export(GEN grp, long format) exports a small group to another format, see galoissexport.

long group_ident(GEN grp, GEN elts) returns the index of the small group *grp* in the GAP4 Small Group library, see galoisidentify. If *elts* is not NULL, it must be the list of elements of *grp*.

long group_ident_trans(GEN grp, GEN elts) returns the index of the regular representation of the small group *grp* in the GAP4 Transitive Group library, see polgalois. If *elts* is no NULL, it must be the list of elements of *grp*.
Chapter 10:
Standard data structures

10.1 Character strings.

10.1.1 Functions returning a char *.

char* pari_strdup(const char *s) returns a malloc’ed copy of s (uses pari_malloc).

char* pari_strndup(const char *s, long n) returns a malloc’ed copy of at most n chars from s (uses pari_malloc). If s is longer than n, only n characters are copied and a terminal null byte is added.

char* stack_strdup(const char *s) returns a copy of s, allocated on the PARI stack (uses stack_malloc).

char* stack_strcat(const char *s, const char *t) returns the concatenation of s and t, allocated on the PARI stack (uses stack_malloc).

char* stack_sprintf(const char *fmt, ...) runs pari_sprintf on the given arguments, returning a string allocated on the PARI stack.

char* itostr(GEN x) writes the t_INT x to a stack_malloc’ed string.

char* GENtostr(GEN x), using the current default output format (GP_DATA->fmt, which contains the output style and the number of significant digits to print), converts x to a malloc’ed string. Simple variant of pari_sprintf.

char* GENtostr_unquoted(GEN x) as GENtostr with the following differences: 1) a t_STR x is printed without enclosing quotes (to be used by print); 2) the result is allocated on the stack and must not be freed.

char* GENtoTeXstr(GEN x), as GENtostr, except that f_TEX overrides the output format from GP_DATA->fmt.

char* RgV_to_str(GEN g, long flag) g being a vector of GENs, returns a malloc’ed string, the concatenation of the GENtostr applied to its elements, except that t_STR are printed without enclosing quotes. flag determines the output format: f_RAW, fPRETTYMAT or f_TEX.
10.1.2 Functions returning a t_STR.

GEN strtoGENstr(const char *s) returns a t_STR with content s.
GEN strntoGENstr(const char *s, long n) returns a t_STR containing the first n characters of s.
GEN chartoGENstr(char c) returns a t_STR containing the character c.
GEN GENtoGENstr(GEN x) returns a t_STR containing the printed form of x (in raw format). This is often easier to use than GENtostr (which returns a malloc-ed char*) since there is no need to free the string after use.
GEN GENtoGENstr_nospace(GEN x) as GENtoGENstr, removing all spaces from the output.
GEN Str(GEN g) as RgV_to_str with output format f_RAW, but returns a t_STR, not a malloc’ed string.
GEN Strtex(GEN g) as RgV_to_str with output format f_TEX, but returns a t_STR, not a malloc’ed string.
GEN Strexpand(GEN g) as RgV_to_str with output format f_RAW, performing tilde and environment expansion on the result. Returns a t_STR, not a malloc’ed string.
GEN gsprintf(const char *fmt, ...) equivalent to pari_sprintf(fmt,...), followed by strtoGENstr. Returns a t_STR, not a malloc’ed string.
GEN gvsprintf(const char *fmt, va_list ap) variadic version of gsprintf.

10.2 Output.

10.2.1 Output contexts.

An output context, of type PariOUT, is a struct that models a stream and contains the following function pointers:

void (*putch)(char); /* fputc()-alike */
void (*puts)(const char*); /* fputs()-alike */
void (*flush)(void); /* fflush()-alike */

The methods putch and puts are used to print a character or a string respectively. The method flush is called to finalize a messages.

The generic functions pari_putc, pari_puts, pari_flush and pari_printf print according to a default output context, which should be sufficient for most purposes. Lower level functions are available, which take an explicit output context as first argument:

void out_putc(PariOUT *out, char c) essentially equivalent to out->putc(c). In addition, registers whether the last character printed was a \n.
void out_puts(PariOUT *out, const char *s) essentially equivalent to out->puts(s). In addition, registers whether the last character printed was a \n.
void out_printf(PariOUT *out, const char *fmt, ...) void out_vprintf(PariOUT *out, const char *fmt, va_list ap)
N.B. The function `out_flush` does not exist since it would be identical to `out->flush()`

```c
int pari_last_was_newline(void) returns a non-zero value if the last character printed via
out_putchar or `out_puts` was \n, and 0 otherwise.
```

```c
void pari_set_last_newline(int last) sets the boolean value to be returned by the function
`pari_last_was_newline` to `last`.
```

### 10.2.2 Default output context

They are defined by the global variables `pariOut` and `pariErr` for normal outputs and warnings/errors, and you probably do not want to change them. If you *do* change them, diverting output in non-trivial ways, this probably means that you are rewriting gp. For completeness, we document in this section what the default output contexts do.

- **pariOut**. writes output to the `FILE* pari_outfile`, initialized to `stdout`. The low-level methods are actually the standard `putc` / `fputs`, plus some magic to handle a log file if one is open.

- **pariErr**. prints to the `FILE* pari_errfile`, initialized to `stderr`. The low-level methods are as above.

You can stick with the default `pariOut` output context and change PARI’s standard output, redirecting `pari_outfile` to another file, using

```c
void switchout(const char *name) where name is a character string giving the name of the file
you want to write to; the output is *appended* at the end of the file. To close the file and revert to
outputting to `stdout`, call `switchout(NULL)`.
```

### 10.2.3 PARI colors

In this section we describe the low-level functions used to implement GP’s color scheme, associated to the `colors` default. The following symbolic names are associated to gp’s output strings:

- `c_ERR` an error message
- `c_HIST` a history number (as in `%1 = ...`)
- `c_PROMPT` a prompt
- `c_INPUT` an input line (minus the prompt part)
- `c_OUTPUT` an output
- `c_HELP` a help message
- `c_TIME` a timer
- `c_NONE` everything else

*If* the `colors` default is set to a non-empty value, before gp outputs a string, it first outputs an ANSI colors escape sequence — understood by most terminals —, according to the `colors` specifications. As long as this is in effect, the following strings are rendered in color, possibly in bold or underlined.

```c
void term_color(long c) prints (as if using `pari_puts`) the ANSI color escape sequence associated
to output object `c`. If `c` is `c_NONE`, revert to default printing style.
```

```c
void out_term_color(PariOUT *out, long c) as `term_color`, using output context `out`.
```
char* term_get_color(char *s, long c) returns as a character string the ANSI color escape sequence associated to output object c. If c is c_NONE, the value used to revert to default printing style is returned. The argument s is either NULL (string allocated on the PARI stack), or preallocated storage (in which case, it must be able to hold at least 16 chars, including the final \0).

### 10.2.4 Obsolete output functions.

These variants of `void output(GEN x)`, which prints x, followed by a newline and a buffer flush are complicated to use and less flexible than what we saw above, or than the `pari_printf` variants. They are provided for backward compatibility and are scheduled to disappear.

```c
void brute(GEN x, char format, long dec)
void matbrute(GEN x, char format, long dec)
void texe(GEN x, char format, long dec)
```

### 10.3 Files.

The following routines are trivial wrappers around system functions (possibly around one of several functions depending on availability). They are usually integrated within PARI’s diagnostics system, printing messages if DEBUGFILES is high enough.

```c
int pari_is_dir(const char *name) returns 1 if name points to a directory, 0 otherwise.
int pari_is_file(const char *name) returns 1 if name points to a directory, 0 otherwise.
int file_is_binary(FILE *f) returns 1 if the file f is a binary file (in the writebin sense), 0 otherwise.
void pari_unlink(const char *s) deletes the file named s. Warn if the operation fails.
void pari_fread_chars(void *b, size_t n, FILE *f) read n chars from stream f, storing the result in pre-allocated buffer b (assumed to be large enough).
char* path_expand(const char *s) perform tilde and environment expansion on s. Returns a malloc'ed buffer.
void strftime_expand(const char *s, char *buf, long max) perform time expansion on s, storing the result (at most max chars) in buffer buf. Trivial wrapper around
time_t t = time(NULL);
strftime(buf, max, s, localtime(&t));
char* pari_get_homedir(const char *user) expands ~user constructs, returning the home directory of user user, or NULL if it could not be determined (in particular if the operating system has no such concept). The return value may point to static area and may be overwritten by subsequent system calls: use immediately or strdup it.
int pari_stdin_isatty(void) returns 1 if our standard input stdin is attached to a terminal. Trivial wrapper around isatty.
```
10.3.1 pariFILE.

PARI maintains a linked list of open files, to reclaim resources (file descriptors) on error or interrupts. The corresponding data structure is a pariFILE, which is a wrapper around a standard FILE*, containing further the file name, its type (regular file, pipe, input or output file, etc.). The following functions create and manipulate this structure; they are integrated within PARI’s diagnostics system, printing messages if DEBUGFILES is high enough.

- pariFILE* pari_fopen(const char *s, const char *mode) wrapper around fopen(s, mode), return NULL on failure.
- pariFILE* pari_fopen_or_fail(const char *s, const char *mode) simple wrapper around fopen(s, mode); error on failure.
- pariFILE* pari_fopengz(const char *s) opens the file whose name is s, and associates a (read-only) pariFILE with it. If s is a compressed file (.gz suffix), it is uncompressed on the fly. If s cannot be opened, also try to open s.gz. Returns NULL on failure.
- void pari_fclose(pariFILE *f) closes the underlying file descriptor and deletes the pariFILE struct.
- pariFILE* pari_safefopen(const char *s, const char *mode) creates a new file s (a priori for writing) with 600 permissions. Error if the file already exists. To avoid symlink attacks, a symbolic link exists, regardless of where it points to.

10.3.2 Temporary files.

PARI has its own idea of the system temp directory derived from from an environment variable ($GPTMPDIR, else $TMPDIR), or the first writable directory among /tmp, /var/tmp and ..

- char* pari_unique_dir(const char *s) creates a “unique directory” and return its name built from the string s, the user id and process pid (on Unix systems). This directory is itself located in the temp directory mentioned above. The name returned is malloc’ed.
- char* pari_unique_filename(const char *s) creates a new empty file in the temp directory, whose name contains the id-string s (truncated to its first 8 chars), followed by a system-dependent suffix (incorporating the ids of both the user and the running process, for instance). The function returns the tempfile name. The name returned is malloc’ed.

10.4 Errors.

This section documents the various error classes, and the corresponding arguments to pari_err. The general syntax is

void pari_err(numerr,...)

In the sequel, we mostly use sequences of arguments of the form

   const char *s
   const char *fmt, ...

where fmt is a PARI format, producing a string s from the remaining arguments. Since providing the correct arguments to pari_err is quite error-prone, we also provide specialized routines pari_err_ERRORCLASS(… ) instead of pari_err(e_ERRORCLASS, ...) so that the C compiler can check their arguments.

We now inspect the list of valid keywords (error classes) for numerr, and the corresponding required arguments.

189
10.4.1 Internal errors, “system” errors.

10.4.1.1 e_ARCH. A requested feature \( s \) is not available on this architecture or operating system.

\[
\text{pari\_err(e\_ARCH)}
\]

prints the error message: sorry, ’\( s \)’ not available on this system.

10.4.1.2 e_BUG. A bug in the PARI library, in function \( s \).

\[
\text{pari\_err(e\_BUG, const char *s)}
\]
\[
\text{pari\_err\_BUG(const char *s)}
\]

prints the error message: Bug in \( s \), please report.

10.4.1.3 e_FILE. Error while trying to open a file.

\[
\text{pari\_err(e\_FILE, const char *what, const char *name)}
\]
\[
\text{pari\_err\_FILE(const char *what, const char *name)}
\]

prints the error message: error opening \( \text{what} \): ‘\( \text{name} \)’.

10.4.1.4 e_IMPL. A requested feature \( s \) is not implemented.

\[
\text{pari\_err(e\__IMPL, const char *s)}
\]
\[
\text{pari\_err\_IMPLEMENT(const char *s)}
\]

prints the error message: sorry, \( s \) is not yet implemented.

10.4.1.5 e_PACKAGE. Missing optional package \( s \).

\[
\text{pari\_err(e\_PACKAGE, const char *s)}
\]
\[
\text{pari\_err\_PACKAGE(const char *s)}
\]

prints the error message: package \( s \) is required, please install it

10.4.2 Syntax errors, type errors.

10.4.2.1 e_DIM. Arguments submitted to function \( s \) have inconsistent dimensions. E.g., when solving a linear system, or trying to compute the determinant of a non-square matrix.

\[
\text{pari\_err(e\_DIM, const char *s)}
\]
\[
\text{pari\_err\_DIM(const char *s)}
\]

prints the error message: inconsistent dimensions in \( s \).

10.4.2.2 e_FLAG. A flag argument is out of bounds in function \( s \).

\[
\text{pari\_err(e\_FLAG, const char *s)}
\]
\[
\text{pari\_err\_FLAG(const char *s)}
\]

prints the error message: invalid flag in \( s \).

10.4.2.3 e_NOTFUNC. Generated by the PARI evaluator; tried to use a \text{GEN} which is not a \text{t\_CLOSURE} in a function call syntax (as in \( f = 1; f(2); \)).

\[
\text{pari\_err(e\_NOTFUNC, GEN fun)}
\]

prints the error message: not a function in a function call.
10.4.2.4 e_OP. Impossible operation between two objects than cannot be typecast to a sensible common domain for deeper reasons than a type mismatch, usually for arithmetic reasons. As in $O(2) + O(3)$: it is valid to add two $t_{PADIC}$s, provided the underlying prime is the same; so the addition is not forbidden a priori for type reasons, it only becomes so when inspecting the objects and trying to perform the operation.

```
pari_err(e_OP, const char *op, GEN x, GEN y)
pari_err_OP(const char *op, GEN x, GEN y)
```

As e_TYPE2, replacing forbidden by inconsistent.

10.4.2.5 e_PRIORITY. object $o$ in function $s$ contains variables whose priority is incompatible with the expected operation. E.g. $\text{Pol}([x,1], 'y)$: this raises an error because it’s not possible to create a polynomial whose coefficients involve variables with higher priority than the main variable.

```
pari_err(e_PRIORITY, const char *s, GEN o, const char *op, long v)
pari_err_PRIORITY(const char *s, GEN o, const char *op, long v)
```

prints the error message: incorrect priority in s, variable $v_o$ op $v$, were $v_o$ is gvar($o$).

10.4.2.6 e_SYNTAX. Syntax error, generated by the PARI parser.

```
pari_err(e_SYNTAX, const char *msg, const char *e, const char *entry)
```

where msg is a complete error message, and $e$ and entry point into the same character string, which is the input that was incorrectly parsed: $e$ points to the character where the parser failed, and entry $\leq e$ points somewhat before.

Prints the error message: msg, followed by a colon, then a part of the input character string (in general entry itself, but an initial segment may be truncated if $e – entry$ is large); a caret points at $e$, indicating where the error took place.

10.4.2.7 e_TYPE. An argument $x$ of function $s$ had an unexpected type. (As in $\text{factor}("blah")$.)

```
pari_err(e_TYPE, const char *s, GEN x)
pari_err_TYPE(const char *s, GEN x)
```

prints the error message: incorrect type in s ($t_x$), where $t_x$ is the type of $x$.

10.4.2.8 e_TYPE2. Forbidden operation between two objects than cannot be typecast to a sensible common domain, because their types do not match up. (As in $\text{Mod}(1,2) + \pi$.)

```
pari_err(e_TYPE2, const char *op, GEN x, GEN y)
pari_err_TYPE2(const char *op, GEN x, GEN y)
```

prints the error message: forbidden $s$ $t_x$ op $t_y$, where $t_z$ denotes the type of $z$. Here, $s$ denotes the spelled out name of the operator $op \in \{+, *, /, \%, =\}$, e.g. addition for "+" or assignment for ":=". If $op$ is not in the above operator, list, it is taken to be the already spelled out name of a function, e.g. "gcd", and the error message becomes forbidden $op$ $t_x$, $t_y$.

10.4.2.9 e_VAR. polynomials $x$ and $y$ submitted to function $s$ have inconsistent variables. E.g., considering the algebraic number $\text{Mod}(t,t^2+1)$ in $\text{nfinit}(x^2+1)$.

```
pari_err(e_VAR, const char *s, GEN x, GEN y)
pari_err_VAR(const char *s, GEN x, GEN y)
```

prints the error message: inconsistent variables in s $X \neq Y$, where $X$ and $Y$ are the names of the variables of $x$ and $y$, respectively.
10.4.3 Overflows.

10.4.3.1 e_COMPONENT. Trying to access an inexistent component in a vector/matrix/list in a function: the index is less than 1 or greater than the allowed length.

```c
pari_err(e_COMPONENT, const char *f, const char *op, GEN lim, GEN x)
pari_err_COMPONENT(const char *f, const char *op, GEN lim, GEN x)
```

prints the error message: non-existent component in f: index op lim. Special case: if f is the empty string (no meaningful public function name can be used), we ignore it and print the message: non-existent component: index op lim.

10.4.3.2 e_DOMAIN. An argument x is not in the function’s domain (as in moebius(0) or zeta(1)).

```c
pari_err(e_DOMAIN, char *f, char *v, char *op, GEN lim, GEN x)
pari_err_DOMAIN(char *f, char *v, char *op, GEN lim, GEN x)
```

prints the error message: domain error in f: v op lim. Special case: if op is the empty string, we ignore lim and print the error message: domain error in f: v out of range.

10.4.3.3 e_MAXPRIME. A function using the precomputed list of prime numbers ran out of primes.

```c
pari_err(e_MAXPRIME, ulong c)
pari_err_MAXPRIME(ulong c)
```

prints the error message: not enough precomputed primes, need primelimit ~c if c is non-zero. And simply not enough precomputed primes otherwise.

10.4.3.4 e_MEM. A call to pari_malloc or pari_realloc failed.

```c
pari_err(e_MEM)
```

prints the error message: not enough memory.

10.4.3.5 e_OVERFLOW. An object in function s becomes too large to be represented within PARI’s hardcoded limits. (As in \(2^{2^{2^{10}}}\) or \(\exp(1e100)\), which overflow in lg and expo.)

```c
pari_err(e_OVERFLOW, const char *s)
pari_err_OVERFLOW(const char *s)
```

prints the error message: overflow in s.

10.4.3.6 e_PREC. Function s fails because input accuracy is too low. (As in floor(1e100) at default accuracy.)

```c
pari_err(e_PREC, const char *s)
pari_err_PREC(const char *s)
```

prints the error message: precision too low in s.

10.4.3.7 e_STACK. The PARI stack overflows.

```c
pari_err(e_STACK)
```

prints the error message: the PARI stack overflows! as well as some statistics concerning stack usage.
10.4.4 Errors triggered intentionally.

10.4.4.1 e_ALARM. A timeout, generated by the alarm function.

\[
\text{pari}_\text{err}(\text{e_ALARM}, \text{const char *fmt}, \ldots)
\]
prints the error message: \textit{s}.

10.4.4.2 e_USER. A user error, as triggered by \texttt{error}(g_1, \ldots, g_n) in GP.

\[
\text{pari}_\text{err}(\text{e_USER}, \text{GEN } g)
\]
prints the error message: \texttt{user error:}, then the entries of the vector \textit{g}.

10.4.5 Mathematical errors.

10.4.5.1 e_CONSTPOL. An argument of function \textit{s} is a constant polynomial, which does not make sense. (As in \texttt{galoisinit(Pol(1)).})

\[
\text{pari}_\text{err}(\text{e_CONSTPOL}, \text{const char *s})
\]
prints the error message: constant polynomial in \textit{s}.

10.4.5.2 e_COPRIME. Function \textit{s} expected two coprime arguments, and did receive \textit{x, y} which were not.

\[
\text{pari}_\text{err}(\text{e_COPRIME}, \text{const char *s, GEN } x, \text{GEN } y)
\]
prints the error message: elements not coprime in \textit{s}: \textit{x,y}.

10.4.5.3 e_INV. Tried to invert a non-invertible object \textit{x}.

\[
\text{pari}_\text{err}(\text{e_INV, GEN } x)
\]
prints the error message: impossible inverse: \textit{x}. If \textit{x} = \texttt{Mod}\texttt{(a,b)} is a \texttt{t_INTMOD} and \textit{a} is not 0 mod \textit{b}, this allows to factor the modulus, as \texttt{gcd}(\textit{a,b}) is a non-trivial divisor of \textit{b}.

10.4.5.4 e_IRREDPOL. Function \textit{s} expected an irreducible polynomial, and did not receive one. (As in \texttt{nfinit(x^2-1).})

\[
\text{pari}_\text{err}(\text{e_IRREDPOL, const char *s, GEN } x)
\]
prints the error message: not an irreducible polynomial in \textit{s}: \textit{x}.

10.4.5.5 e_MISC. Generic uncategorized error.

\[
\text{pari}_\text{err}(\text{e_MISC, const char *fmt, \ldots})
\]
prints the error message: \textit{s}.

10.4.5.6 e_MODULUS. moduli \textit{x} and \textit{y} submitted to function \textit{s} are inconsistent. E.g., considering the algebraic number Mod\texttt{(t,t^2+1)} in \texttt{nfinit(t^3-2)}.

\[
\text{pari}_\text{err}(\text{e_MODULUS, const char *s, GEN } x, \text{GEN } y)
\]
prints the error message: inconsistent moduli in \textit{s}, then the moduli.
10.4.5.7 `e_PRIME`. Function $s$ expected a prime number, and did receive $p$, which was not. (As in `idealprimedec(nf, 4)`.)

```c
pari_err(e_PRIME, const char *s, GEN x)
pari_err_PRIME(const char *s, GEN x)
```
prints the error message: **not a prime in** $s$: $x$.

10.4.5.8 `e_ROOTS0`. An argument of function $s$ is a zero polynomial, and we need to consider its roots. (As in `polroots(0)`.)

```c
pari_err(e_ROOTS0, const char *s)
pari_err_ROOTS0(const char *s)
```
prints the error message: **zero polynomial in** $s$.

10.4.5.9 `e_SQRTN`. Tried to compute an $n$-th root of $x$, which does not exist, in function $s$. (As in `sqrt(Mod(-1,3))`.)

```c
pari_err(e_SQRTN, GEN x)
pari_err_SQRTN(GEN x)
```
prints the error message: **not an n-th power residue in** $s$: $x$.

10.4.6 Miscellaneous functions.

`long name_numerr(const char *s)` return the error number corresponding to an error name. E.g. `name_numerr("e_DIM")` returns `e_DIM`.

`const char* numerr_name(long errnum)` returns the error name corresponding to an error number. E.g. `name_numerr(e_DIM)` returns "e_DIM".

`char* pari_err2str(GEN err)` returns the error message that would be printed on `t_ERROR err`. The name is allocated on the PARI stack and must not be freed.

10.5 Hashtables.

A **hashtable**, or associative array, is a set of pairs $(k, v)$ of keys and values. PARI implements general extensible hashtables for fast data retrieval: when creating a table, we may either choose to use the PARI stack, or `malloc` so as to be stack-independent. A hashtable is implemented as a table of linked lists, each list containing all entries sharing the same hash value. The table length is a prime number, which roughly doubles as the table overflows by gaining new entries; both the current number of entries and the threshold before the table grows are stored in the table. Finally the table remembers the functions used to hash the entries’s keys and to test for equality two entries hashed to the same value.

An entry, or **hashentry**, contains

- a key/value pair $(k, v)$, both of type `void*` for maximal flexibility,
- the hash value of the key, for the table hash function. This hash is mapped to a table index (by reduction modulo the table length), but it contains more information, and is used to bypass costly general equality tests if possible,
- a link pointer to the next entry sharing the same table cell.
typedef struct {
    void *key, *val;
    ulong hash; /* hash(key) */
    struct hashentry *next;
} hashentry;

typedef struct {
    ulong len; /* table length */
    hashentry **table; /* the table */
    ulong nb, maxnb; /* number of entries stored and max nb before enlarging */
    ulong pindex; /* prime index */
    ulong (*hash)(void *k); /* hash function */
    int (*eq)(void *k1, void *k2); /* equality test */
    int use_stack; /* use the PARI stack, resp. malloc */
} hashtable;

hashtable* hash_create(size, hash, eq, use_stack)
    ulong size;
    ulong (*hash)(void*);
    int (*eq)(void*,void*);
    int use_stack;
creates a hashtable with enough room to contain size entries. The functions hash and eq compute
the hash value of keys and test keys for equality, respectively. If use_stack is non zero, the resulting
table will use the PARI stack; otherwise, we use malloc.

void hash_insert(hashtable *h, void *k, void *v) inserts (k,v) in hashtable h. No copy is
made: k and v themselves are stored. The implementation does not prevent one to insert two
entries with equal keys k, but which of the two is affected by later commands is undefined.

hashentry* hash_search(hashtable *h, void *k) look for an entry with key k in h. Return it
if it one exists, and NULL if not.

hashentry* hash_remove(hashtable *h, void *k) deletes an entry (k,v) with key k from h
and return it. (Return NULL if none was found.) Only the linking structures are freed, memory
associated to k and v is not reclaimed.

void hash_destroy(hashtable *h) deletes the hashtable, by removing all entries.

Some interesting hash functions are available:

ulong hash_str(const char *s)

ulong hash_str2(const char *s) is the historical PARI string hashing function and seems to be
generally inferior to hash_str.

ulong hash_GEN(GEN x)

195
10.6 Dynamic arrays.

A dynamic array is a generic way to manage stacks of data that need to grow dynamically. It allocates memory using pari_malloc, and is independent of the PARI stack; it even works before the pari_init call.

10.6.1 Initialization.

To create a stack of objects of type foo, we proceed as follows:

```c
foo *t_foo;
pari_stack s_foo;
pari_stack_init(&s_foo, sizeof(*t_foo), (void**)t_foo);
```

Think of s_foo as the controlling interface, and t_foo as the (dynamic) array tied to it. The value of t_foo may be changed as you add more elements.

10.6.2 Adding elements. The following function pushes an element on the stack.

```c
/* access globals t_foo and s_foo */
void push_foo(foo x)
{
    long n = pari_stack_new(&s_foo);
    t_foo[n] = x;
}
```

10.6.3 Accessing elements.

Elements are accessed naturally through the t_foo pointer. For example this function swaps two elements:

```c
void swapfoo(long a, long b)
{
    foo x;
    if (a > s_foo.n || b > s_foo.n) pari_err_BUG("swapfoo");
    x = t_foo[a];
    t_foo[a] = t_foo[b];
    t_foo[b] = x;
}
```

10.6.4 Stack of stacks. Changing the address of t_foo is not supported in general. In particular realloc()’ed array of stacks and stack of stacks are not supported.
10.6.5 Public interface. Let s be a pari_stack and data the data linked to it. The following public fields are defined:

- **s.alloc** is the number of elements allocated for data.
- **s.n** is the number of elements in the stack and data[s.n-1] is the topmost element of the stack. s.n can be changed as long as 0 ≤ s.n ≤ s.alloc holds.

void pari_stack_init(pari_stack *s, size_t size, void **data) links *s to the data pointer *data, where size is the size of data element. The pointer *data is set to NULL, s->n and s->alloc are set to 0: the array is empty.

void pari_stack_alloc(pari_stack *s, long nb) makes room for nb more elements, i.e. makes sure that s.alloc ≥ s.n + nb, possibly reallocating data.

long pari_stack_new(pari_stack *s) increases s.n by one unit, possibly reallocating data, and returns s.n − 1.

Caveat. The following construction is incorrect because stack_new can change the value of t_foo:

t_foo[ pari_stack_new(&s_foo) ] = x;

void pari_stack_delete(pari_stack *s) frees data and resets the stack to the state immediately following stack_init (s->n and s->alloc are set to 0).

void * pari_stack_pushp(pari_stack *s, void *u) This function assumes that *data is of pointer type. Pushes the element u on the stack s.

void ** pari_stack_base(pari_stack *s) returns the address of data, typecast to a void **.

10.7 Vectors and Matrices.

10.7.1 Access and extract. See Section 8.3.1 and Section 8.3.2 for various useful constructors. Coefficients are accessed and set using gel, gcoeff, see Section 5.2.7. There are many internal functions to extract or manipulate subvectors or submatrices but, like the accessors above, none of them are suitable for gerepileupto. Worse, there are no type verification, nor bound checking, so use at your own risk.

GEN shallowcopy(GEN x) returns a GEN whose components are the components of x (no copy is made). The result may now be used to compute in place without destroying x. This is essentially equivalent to

GEN y = cgetg(lg(x), typ(x));
for (i = 1; i < lg(x); i++) y[i] = x[i];
return y;

except that t_MAT is treated specially since shallow copies of all columns are made. The function also works for non-recursive types, but is useless in that case since it makes a deep copy. If x is known to be a t_MAT, you may call RgM_shallowcopy directly; if x is known not to be a t_MAT, you may call leafcopy directly.

GEN RgM_shallowcopy(GEN x) returns shallowcopy(x), where x is a t_MAT.

GEN shallowtrans(GEN x) returns the transpose of x, without copying its components, i.e., it returns a GEN whose components are (physically) the components of x. This is the internal function underlying gtrans.
GEN shallowconcat(GEN x, GEN y) concatenate $x$ and $y$, without copying components, i.e., it returns a GEN whose components are (physically) the components of $x$ and $y$.

GEN shallowconcat1(GEN x) $x$ must be t_VEC or t_LIST, concatenate its elements from left to right. Shallow version of concat1.

GEN shallowmatconcat(GEN v) shallow version of matconcat.

GEN shallowextract(GEN x, GEN y) extract components of the vector or matrix $x$ according to the selection parameter $y$. This is the shallow analog of extract0($x$, $y$, NULL), see vecextract.

GEN RgM_minor(GEN A, long i, long j) given a square t_MAT A, return the matrix with $i$-th row and $j$-th column removed.

GEN vconcat(GEN A, GEN B) concatenate vertically the two t_MAT $A$ and $B$ of compatible dimensions. A NULL pointer is accepted for an empty matrix. See shallowconcat.

GEN row(GEN A, long i) return $A[i]$, the $i$-th row of the t_MAT $A$.

GEN row_i(GEN A, long i, long j1, long j2) return part of the $i$-th row of t_MAT $A$: $A[i,j_1], A[i,j_1+1],\ldots,A[i,j_2]$. Assume $j_1 \leq j_2$.

GEN rowcopy(GEN A, long i) return the row $A[i]$ of the t_MAT $A$. This function is memory clean and suitable for gerepileupto. See row for the shallow equivalent.

GEN rowslice(GEN A, long i1, long i2) return the t_MAT formed by the $i_1$-th through $i_2$-th rows of t_MAT $A$. Assume $i_1 \leq i_2$.

GEN rowpermute(GEN A, GEN p), $p$ being a t_VECSMALL representing a list $[p_1,\ldots,p_n]$ of rows of t_MAT $A$, returns the matrix whose rows are $A[p_1],\ldots,A[p_n]$.

GEN rowslicepermute(GEN A, GEN p, long x1, long x2), short for rowslice(rowpermute(A,p), x1, x2) (more efficient).

GEN vecslice(GEN A, long j1, long j2), return $A[j_1],\ldots,A[j_2]$. If $A$ is a t_MAT, these correspond to columns of $A$. The object returned has the same type as $A$ (t_VEC, t_COL or t_MAT). Assume $j_1 \leq j_2$.

GEN vecsplice(GEN A, long j) return $A$ with $j$-th entry removed (t_VEC, t_COL) or $j$-th column removed (t_MAT).

GEN vecreverse(GEN A). Returns a GEN which has the same type as $A$ (t_VEC, t_COL or t_MAT), and whose components are the $A[n],\ldots,A[1]$. If $A$ is a t_MAT, these are the columns of $A$.

GEN vecpermutEGEN A, GEN p) $p$ is a t_VECSMALL representing a list $[p_1,\ldots,p_n]$ of indices. Returns a GEN which has the same type as $A$ (t_VEC, t_COL or t_MAT), and whose components are $A[p_1],\ldots,A[p_n]$. If $A$ is a t_MAT, these are the columns of $A$.

GEN vecslicepermute(GEN A, GEN p, long y1, long y2) short for vecslice(vecpermuteg(A,p), y1, y2) (more efficient).
10.7.2 Componentwise operations.

The following convenience routines automate trivial loops of the form

\[
\text{for } (i = 1; i < \log(a); i++) \text{ gel(v,i) = } f(\text{gel(a,i), gel(b,i)})
\]

for suitable \(f\):

**GEN vecinv(GEN a).** Given a vector \(a\), returns the vector whose \(i\)-th component is \(\text{ginv}(a[i])\).

**GEN vecmul(GEN a, GEN b).** Given \(a\) and \(b\) two vectors of the same length, returns the vector whose \(i\)-th component is \(\text{gmul}(a[i], b[i])\).

**GEN vecdiv(GEN a, GEN b).** Given \(a\) and \(b\) two vectors of the same length, returns the vector whose \(i\)-th component is \(\text{gdiv}(a[i], b[i])\).

**GEN vecpow(GEN a, GEN n).** Given \(n\) a \(\text{t_INT}\), returns the vector whose \(i\)-th component is \(a[i]^n\).

**GEN vecmodii(GEN a, GEN b).** Assuming \(a\) and \(b\) are two \(\text{ZV}\) of the same length, returns the vector whose \(i\)-th component is \(\text{modii}(a[i], b[i])\).

Note that \text{vecadd} or \text{vecsub} do not exist since \text{gadd} and \text{gsub} have the expected behavior. On the other hand, \text{ginv} does not accept vector types, hence \text{vecinv}.

10.7.3 Low-level vectors and columns functions.

These functions handle \(\text{t_VEC}\) as an abstract container type of \(\text{GENs}\). No specific meaning is attached to the content. They accept both \(\text{t_VEC}\) and \(\text{t_COL}\) as input, but \text{col} functions always return \(\text{t_COL}\) and \text{vec} functions always return \(\text{t_VEC}\).

**Note.** All the functions below are shallow.

**GEN const_col(long n, GEN x)** returns a \(\text{t_COL}\) of \(n\) components equal to \(x\).

**GEN const_vec(long n, GEN x)** returns a \(\text{t_VEC}\) of \(n\) components equal to \(x\).

**int vec_isconst(GEN v)** Returns 1 if all the components of \(v\) are equal, else returns 0.

**void vec_setconst(GEN v, GEN x)** \(v\) a pre-existing vector. Set all its components to \(x\).

**int vec_is1to1(GEN v)** Returns 1 if the components of \(v\) are pair-wise distinct, i.e. if \(i \mapsto v[i]\) is a 1-to-1 mapping, else returns 0.

**GEN vec_shorten(GEN v, long n)** shortens the vector \(v\) to \(n\) components.

**GEN vec_lengthen(GEN v, long n)** lengthens the vector \(v\) to \(n\) components. The extra components are not initialized.

**GEN vec_insert(GEN v, long n, GEN x)** inserts \(x\) at position \(n\) in the vector \(v\).
10.8 Vectors of small integers.

10.8.1 t_VECSMALL.

These functions handle t_VECSMALL as an abstract container type of small signed integers. No specific meaning is attached to the content.

**GEN const_vecsmall(long n, long c)** returns a t_VECSMALL of n components equal to c.

**GEN vec_to_vecsmall(GEN z)** identical to ZV_to_zv(z).

**GEN vecsmall_to_vec(GEN z)** identical to zv_to_ZV(z).

**GEN vecsmall_to_col(GEN z)** identical to zv_to_ZC(z).

**GEN vecsmall_copy(GEN x)** makes a copy of x on the stack.

**GEN vecsmall_shorten(GEN v, long n)** shortens the t_VECSMALL v to n components.

**GEN vecsmall_lengthen(GEN v, long n)** lengthens the t_VECSMALL v to n components. The extra components are not initialized.

**GEN vecsmall_indexsort(GEN x)** performs an indirect sort of the components of the t_VECSMALL x and return a permutation stored in a t_VECSMALL.

**void vecsmall_sort(GEN v)** sorts the t_VECSMALL v in place.

**long vecsmall_max(GEN v)** returns the maximum of the elements of t_VECSMALL v, assumed non-empty.

**long vecsmall_indexmax(GEN v)** returns the index of the largest element of t_VECSMALL v, assumed non-empty.

**long vecsmall_min(GEN v)** returns the minimum of the elements of t_VECSMALL v, assumed non-empty.

**long vecsmall_indexmin(GEN v)** returns the index of the smallest element of t_VECSMALL v, assumed non-empty.

**long vecsmall_isin(GEN v, long x)** returns the first index i such that v[i] is equal to x. Naive search in linear time, does not assume that v is sorted.

**GEN vecsmall_uniq(GEN v)** given a t_VECSMALL v, return the vector of unique occurrences.

**GEN vecsmall_uniq_sorted(GEN v)** same as vecsmall_uniq, but assumes v sorted.

**long vecsmall_duplicate(GEN v)** given a t_VECSMALL v, return 0 if there is no duplicates, or the index of the first duplicate (vecsmall_duplicate([1,1]) returns 2).

**long vecsmall_duplicate_sorted(GEN v)** same as vecsmall_duplicate, but assume v sorted.

**int vecsmall_lexcmp(GEN x, GEN y)** compares two t_VECSMALL lexically.

**int vecsmall_prefixcmp(GEN x, GEN y)** truncate the longest t_VECSMALL to the length of the shortest and compares them lexicographically.

**GEN vecsmall_prepend(GEN V, long s)** prepend s to the t_VECSMALL V.

**GEN vecsmall_append(GEN V, long s)** append s to the t_VECSMALL V.

**GEN vecsmall_concat(GEN u, GEN v)** concat the t_VECSMALL u and v.
long vecsmall_coincidence(GEN u, GEN v) returns the numbers of indices where u and v agree.

long vecsmall_pack(GEN v, long base, long mod) handles the t_VECSMALL v as the digit of a number in base base and return this number modulo mod. This can be used as an hash function.

10.8.2 Vectors of t_VECSMALL. These functions manipulate vectors of t_VECSMALL (vecvecsmall).

GEN vecvecsmall_sort(GEN x) sorts lexicographically the components of the vector x.

GEN vecvecsmall_sort_uniq(GEN x) sorts lexicographically the components of the vector x, removing duplicates entries.

GEN vecvecsmall_indexsort(GEN x) performs an indirect lexicographic sorting of the components of the vector x and return a permutation stored in a t_VECSMALL.

long vecvecsmall_search(GEN x, GEN y, long flag) x being a sorted vecvecsmall and y a t_VECSMALL, search y inside x. flag has the same meaning as for setsearch.
11.1 Handling closures.

### 11.1.1 Functions to evaluate t_CLOSURE.

- **void closure_disassemble(GEN C)** print the t_CLOSURE C in GP assembly format.

- **GEN closure_callgenall(GEN C, long n, ...)** evaluate the t_CLOSURE C with the n arguments (of type GEN) following n in the function call. Assumes C has arity ≥ n.

- **GEN closure_callgenvec(GEN C, GEN args)** evaluate the t_CLOSURE C with the arguments supplied in the vector args. Assumes C has arity ≥ \( \lg(\text{args}) - 1 \).

- **GEN closure_callgen1(GEN C, GEN x)** evaluate the t_CLOSURE C with argument x. Assumes C has arity ≥ 1.

- **GEN closure_callgen2(GEN C, GEN x, GEN y)** evaluate the t_CLOSURE C with argument x, y. Assumes C has arity ≥ 2.

- **void closure_callvoid1(GEN C, GEN x)** evaluate the t_CLOSURE C with argument x and discard the result. Assumes C has arity ≥ 1.

The following technical functions are used to evaluate *inline* closures and closures of arity 0.

The control flow statements (break, next and return) will cause the evaluation of the closure to be interrupted; this is called below a *flow change*. When that occurs, the functions below generally return NULL. The caller can then adopt three positions:

- raises an exception (*closure_evalnobrk*).
- passes through (by returning NULL itself).
- handles the flow change.

- **GEN closure_evalgen(GEN code)** evaluates a closure and returns the result, or NULL if a flow change occurred.

- **GEN closure_evalnobrk(GEN code)** as *closure_evalgen* but raise an exception if a flow change occurs. Meant for iterators where interrupting the closure is meaningless, e.g. `intnum` or `sumnum`.

- **void closure_evalvoid(GEN code)** evaluates a closure whose return value is ignored. The caller has to deal with eventual flow changes by calling `loop_break`.

The remaining functions below are for exceptional situations:

- **GEN closure_evalres(GEN code)** evaluates a closure and returns the result. The difference with *closure_evalgen* being that, if the flow end by a return statement, the result will be the returned value instead of NULL. Used by the main GP loop.
GEN closure_evalbrk(GEN code, long *status) as closure_evalres but set status to a non-zero value if a flow change occurred. This variant is not stack clean. Used by the break loop.

GEN closure_trapgen(long numerr, GEN code) evaluates closure, while trapping error numerr. Return (GEN)1L if error trapped, and the result otherwise, or NULL if a flow change occurred. Used by trap.

11.1.2 Functions to handle control flow changes.

long loop_break(void) processes an eventual flow changes inside an iterator. If this function return 1, the iterator should stop.

11.1.3 Functions to deal with lexical local variables.

Function using the prototype code ‘V’ need to manually create and delete a lexical variable for each code ‘V’, which will be given a number −1, −2,....

void push_lex(GEN a, GEN code) creates a new lexical variable whose initial value is a on the top of the stack. This variable get the number −1, and the number of the other variables is decreased by one unit. When the first variable of a closure is created, the argument code must be the closure that references this lexical variable. The argument code must be NULL for all subsequent variables (if any). (The closure contains the debugging data for the variable).

void pop_lex(long n) deletes the n topmost lexical variables, increasing the number of other variables by n. The argument n must match the number of variables allocated through push_lex.

GEN get_lex(long vn) get the value of the variable with number vn.

void set_lex(long vn, GEN x) set the value of the variable with number vn.

11.1.4 Functions returning new closures.

GEN compile_str(const char *s) returns the closure corresponding to the GP expression s.

GEN closure_deriv(GEN code) returns a closure corresponding to the numerical derivative of the closure code.

GEN snm_closure(entree *ep, GEN data) Let data be a vector of length m, ep be an entree pointing to a C function f of arity n + m, returns a t_CLOSURE object g of arity n such that $g(x_1,...,x_n) = f(x_1,...,x_n, gel(data, 1),..., gel(data, m))$. If data is NULL, then m = 0 is assumed. This function has a low overhead since it does not copy data.

GEN strtofunction(char *str) returns a closure corresponding to the built-in or install’ed function named str.

GEN strtoclosure(char *str, long n, ...) returns a closure corresponding to the built-in or install’ed function named str with the n last parameters set to the n GENs following n, see snm_closure. This function has an higher overhead since it copies the parameters and does more input validation.

In the example code below, agm1 is set to the function x->agm(x,1) and res is set to agm(2,1).

    GEN agm1 = strtoclosure("agm", 1, gen_1);
    GEN res = closure_callgen1(agm1, gen_2);
11.1.5 Functions used by the gp debugger (break loop). long closure_context(long s) restores the compilation context starting at frame s+1, and returns the index of the topmost frame. This allow to compile expressions in the topmost lexical scope.

void closure_err(void) prints a backtrace of the last 20 stack frames.

11.1.6 Standard wrappers for iterators. Two families of standard wrappers are provided to interface iterators like intnum or sumnum with GP.

11.1.6.1 Standard wrappers for inline closures. These wrappers are used to implement GP functions taking inline closures as input. The object (GEN)E must be an inline closure which is evaluated with the lexical variable number −1 set to x.

GEN gp_eval(void *E, GEN x) is used for the prototype code ‘E’.

long gp_evalvoid(void *E, GEN x) is used for the prototype code ‘I’. The resulting value is discarded. Return a non-zero value if a control-flow instruction request the iterator to terminate immediately.

long gp_evalbool(void *E, GEN x) returns the boolean gp_eval(E, x) evaluates to (i.e. true if the value is non-zero).

GEN gp_evalupto(void *E, GEN x) memory-safe version of gp_eval, gcopy-ing the result, when the evaluator returns components of previously allocated objects (e.g. member functions).

11.1.6.2 Standard wrappers for true closures. These wrappers are used to implement GP functions taking true closures as input.

GEN gp_call(void *E, GEN x) evaluates the closure (GEN)E on x.

long gp_callbool(void *E, GEN x) evaluates the closure (GEN)E on x, returns 1 if its result is non-zero, and 0 otherwise.

long gp_callvoid(void *E, GEN x) evaluates the closure (GEN)E on x, discarding the result. Return a non-zero value if a control-flow instruction request the iterator to terminate immediately.

11.2 Defaults.

int pari_is_default(const char *s) return 1 if s is the name of a default, 0 otherwise.

GEN setdefault(const char *s, const char *v, long flag) is the low-level function underlying default0. If s is NULL, call all default setting functions with string argument NULL and flag d_ACKNOWLEDGE. Otherwise, check whether s corresponds to a default and call the corresponding default setting function with arguments v and flag.

We shall describe these functions below: if v is NULL, we only look at the default value (and possibly print or return it, depending on flag); otherwise the value of the default to v, possibly after some translation work. The flag is one of

- d_INITRC called while reading the gprc: print and return gnil, possibly defer until gp actually starts.
- d_RETURN return the current value, as a t_INT if possible, as a t_STR otherwise.
- d_ACKNOWLEDGE print the current value, return gnil.
• d_SILENT print nothing, return gnil.

Low-level functions called by setdefault:

GEN sd_TeXstyle(const char *v, long flag)
GEN sd_colors(const char *v, long flag)
GEN sd_compatible(const char *v, long flag)
GEN sd_datadir(const char *v, long flag)
GEN sd_debug(const char *v, long flag)
GEN sd_debugfiles(const char *v, long flag)
GEN sd_debugmem(const char *v, long flag)
GEN sd_factor_add_primes(const char *v, long flag)
GEN sd_factor_proven(const char *v, long flag)
GEN sd_format(const char *v, long flag)
GEN sd_histsize(const char *v, long flag)
GEN sd_log(const char *v, long flag)
GEN sd_logfile(const char *v, long flag)
GEN sd_nbthreads(const char *v, long flag)
GEN sd_new_galois_format(const char *v, long flag)
GEN sd_output(const char *v, long flag)
GEN sd_parisize(const char *v, long flag)
GEN sd_path(const char *v, long flag)
GEN sd_prettyprinter(const char *v, long flag)
GEN sd_primerlimit(const char *v, long flag)
GEN sd_realprecision(const char *v, long flag)
GEN sd_recover(const char *v, long flag)
GEN sd_secure(const char *v, long flag)
GEN sd_seriesprecision(const char *v, long flag)
GEN sd_simplify(const char *v, long flag)
GEN sd_sopath(char *v, int flag)
GEN sd_strictargs(const char *v, long flag)
GEN sd_strictmatch(const char *v, long flag)
GEN sd_threadsize(const char *v, long flag)

Generic functions used to implement defaults: most of the above routines are implemented in terms of the following generic ones. In all routines below
• v and flag are the arguments passed to default: v is a new value (or the empty string: no change), and flag is one of d_INITRC, d_RETURN, etc.

• s is the name of the default being changed, used to display error messages or acknowledgments.

GEN sd_toggle(const char *v, long flag, const char *s, int *ptn)

• if v is neither "0" nor "1", an error is raised using pari_err.

• ptn points to the current numerical value of the toggle (1 or 0), and is set to the new value (when v is non-empty).

For instance, here is how the timer default is implemented internally:

GEN
sd_timer(const char *v, long flag)
{ return sd_toggle(v,flag,"timer", &GP_DATA->chrono); }

The exact behavior and return value depends on flag:

• d_RETURN: returns the new toggle value, as a GEN.

• d_ACKNOWLEDGE: prints a message indicating the new toggle value and return gnil.

• other cases: print nothing and return gnil.

GEN sd_ulong(const char *v, long flag, const char *s, ulong *ptn, ulong Min, ulong Max, const char **msg)

• ptn points to the current numerical value of the toggle, and is set to the new value (when v is non-empty).

• Min and Max point to the minimum and maximum values allowed for the default.

• v must translate to an integer in the allowed ranger, a suffix among k/K (×10^3), m/M (×10^6), or g/G (×10^9) is allowed, but no arithmetic expression.

• msg is a [NULL]-terminated array of messages or NULL (ignored). If msg is not NULL, msg[i] contains a message associated to the value i of the default. The last entry in the msg array is used as a message associated to all subsequent ones.

The exact behavior and return value depends on flag:

• d_RETURN: returns the new toggle value, as a GEN.

• d_ACKNOWLEDGE: prints a message indicating the new value, possibly a message associated to it via the msg argument, and return gnil.

• other cases: print nothing and return gnil.

GEN sd_string(const char *v, long flag, const char *s, char **pstr)

• v is subject to environment expansion, then time expansion.

• pstr points to the current string value, and is set to the new value (when v is non-empty).
11.3 Records and Lazy vectors.

The functions in this section are used to implement ell structures and analogous objects, which are vectors some of whose components are initialized to dummy values, later computed on demand. We start by initializing the structure:

GEN obj_init(long d, long n) returns an obj S, a t_VEC with d regular components, accessed as gel(S,1), ..., gel(S,d); together with a record of n members, all initialized to 0. The arguments d and n must be non-negative.

After \( S = \text{obj}\_\text{init}(d, n) \), the prototype of our other functions are of the form

GEN obj_do(GEN S, long tag, ...)

The first argument \( S \) holds the structure to be managed. The second argument tag is the index of the struct member (from 1 to \( n \)) we operate on. We recommend to define an enum and use descriptive names instead of hardcoded numbers. For instance, if \( n = 3 \), after defining

```
enum { TAG_p = 1, TAG_list, TAG_data };
```

one may use TAG_list or 2 indifferently as a tag. The former being preferred, of course.

**Technical note.** In the current implementation, \( S \) is a t_VEC with \( d + 1 \) entries. The first \( d \) components are ordinary t_GEN entries, which you can read or assign to in the customary way. But the last component gel(S,d+1), a t_VEC of length \( n \) initialized to zerovec(n), must be handled in a special way: you should never access or modify its components directly, only through the API we are about to describe. Indeed, its entries are meant to contain dynamic data, which will be stored, retrieved and replaced (for instance by a value computed to a higher accuracy), while interacting safely with intermediate gerepile calls. This mechanism allows to simulate C structs, in a simpler way than with general hashtables, while remaining compatible with the GP language, which knows neither structs nor hashtables. It also serialize the structure in an ordinary GEN, which facilitates copies and garbage collection (use gcobj or gerepile), rather than having to deal with individual components of actual C structs.

GEN obj_check(GEN S, long tag) if the tag-component in \( S \) is non empty, return it. Otherwise return NULL. The t_INT 0 (initial value) is used as a sentinel to indicated an empty component.

GEN obj_insert(GEN S, long tag, GEN O) insert (a clone of) \( O \) as tag-component of \( S \). Any previous value is deleted, and data pointing to it become invalid.

GEN obj_insert_shallow(GEN S, long K, GEN O) as obj_insert, inserting \( O \) as-is, not via a clone.

GEN obj_checkbuild(GEN S, long tag, GEN (*build)(GEN)) if the tag-component of \( S \) is non empty, return it. Otherwise insert (a clone of) build(S) as tag-component in \( S \), and return it.

GEN obj_checkbuild_padicprec(GEN S, long tag, GEN (*build)(GEN,long), long prec) if the tag-component of \( S \) is non empty and has relative p-adic precision \( \geq \) prec, return it. Otherwise insert (a clone of) build(S, prec) as tag-component in \( S \), and return it.

GEN obj_checkbuild_prec(GEN S, long tag, GEN (*build)(GEN,long), long prec) if the tag-component of \( S \) is non empty and has gprecision \( \geq \) prec, return it. Otherwise insert (a clone of) build(S, prec) as tag-component in \( S \), and return it.

void obj_free(GEN S) destroys all clones stored in the \( n \) tagged components, and replace them by the initial value 0. The regular entries of \( S \) are unaffected, and \( S \) remains a valid object. This is used to avoid memory leaks.
Chapter 12:
Technical Reference Guide for Algebraic Number Theory

12.1 General Number Fields.

12.1.1 Number field types.

None of the following routines thoroughly check their input: they distinguish between bona fide
structures as output by PARI routines, but designing perverse data will easily fool them. To give
an example, a square matrix will be interpreted as an ideal even though the \( \mathbb{Z} \)-module generated by
its columns may not be an \( \mathbb{Z}_K \)-module (i.e. the expensive \texttt{nfisideal} routine will not be called).

\texttt{long nftyp(GEN x)}. Returns the type of number field structure stored in \( x \), \texttt{typ_NF}, \texttt{typ_BNF}, or
\texttt{typ_BNR}. Other answers are possible, meaning \( x \) is not a number field structure.

\texttt{GEN get_nf(GEN x, long *t)}. Extract an \( nf \) structure from \( x \) if possible and return it, otherwise
return \texttt{NULL}. Sets \( t \) to the \texttt{nftyp} of \( x \) in any case.

\texttt{GEN get_bnf(GEN x, long *t)}. Extract a \( bnf \) structure from \( x \) if possible and return it, otherwise
return \texttt{NULL}. Sets \( t \) to the \texttt{nftyp} of \( x \) in any case.

\texttt{GEN get_nfpol(GEN x, GEN *nf)} try to extract an \( nf \) structure from \( x \), and sets \( *nf \) to \texttt{NULL}
(failure) or to the \( nf \). Returns the (monic, integral) polynomial defining the field.

\texttt{GEN get_bnfpol(GEN x, GEN *bnf, GEN *nf)} try to extract a \( bnf \) and an \( nf \) structure from \( x \),
and sets \( *bnf \) and \( *nf \) to \texttt{NULL} (failure) or to the corresponding structure. Returns the (monic,
integral) polynomial defining the field.

\texttt{GEN checknf(GEN x)} if an \( nf \) structure can be extracted from \( x \), return it; otherwise raise an
exception. The more general \texttt{get_nf} is often more flexible.

\texttt{GEN checkbnf(GEN x)} if an \( bnf \) structure can be extracted from \( x \), return it; otherwise raise an
exception. The more general \texttt{get_bnf} is often more flexible.

\texttt{void checkbnr(GEN bnr)} Raise an exception if the argument is not a \( bnr \) structure.

\texttt{void checkbnrgen(GEN bnr)} Raise an exception if the argument is not a \( bnr \) structure, complete
with explicit generators for the ray class group. This is normally useless and \texttt{checkbnr}
should be instead, unless you are absolutely certain that the generators will be needed at a later point,
and you are about to embark in a costly intermediate computation. PARI functions do check that
generators are present in \( bnr \) before accessing them: they will raise an error themselves; many
functions that may require them, e.g. \texttt{bnrconductor}, often do not actually need them.

\texttt{void checkrnf(GEN rnf)} Raise an exception if the argument is not an \( rnf \) structure.

\texttt{void checkbid(GEN bid)} Raise an exception if the argument is not a \( bid \) structure.

\texttt{GEN checkgal(GEN x)} if a \texttt{galoisinit} structure can be extracted from \( x \), return it; otherwise raise
an exception.
void checksqmat(GEN x, long N) check whether x is a square matrix of dimension N. May be used to check for ideals if N is the field degree.

void checkprim(GEN pr) Raise an exception if the argument is not a prime ideal structure.

GEN get_prim(GEN ideal) return the underlying prime ideal structure if one can be extracted from ideal (ideal or extended ideal), and return NULL otherwise.

void checkabgrp(GEN v) Raise an exception if the argument is not an abelian group structure, i.e. a t_VEC with either 2 or 3 entries: [N, cyc] or [N, cyc, gen].

GEN abgrp_get_no(GEN x) extract the cardinality N from an abelian group structure.

GEN abgrp_get_cyc(GEN x) extract the elementary divisors cyc from an abelian group structure.

GEN abgrp_get_gen(GEN x) extract the generators gen from an abelian group structure.

void checkmodpr(GEN modpr) Raise an exception if the argument is not a prime ideal structure.

GEN checknfelt_mod(GEN nf, GEN x, const char *s) given an nf structure nf and a t_POLMOD x, return the associated polynomial representative (shallow) if x and nf are compatible. Raise an exception otherwise. Set s to the name of the caller for a meaningful error message.

GEN check_ZKmodule(GEN x, const char *s) check whether x looks like Z_K-module (a pair [A,I], where A is a matrix and I is a list of ideals; A has as many columns as I has elements. Otherwise raises an exception. Set s to the name of the caller for a meaningful error message.

12.1.2 Extracting info from a nf structure.

These functions expect a true nf argument associated to a number field $K = \mathbb{Q}[x]/(T)$, e.g. a bnf will not work. Let $n = [K : \mathbb{Q}]$ be the field degree.

GEN nf_get_pol(GEN nf) returns the polynomial $T$ (monic, in $\mathbb{Z}[x]$).

long nf_get_varn(GEN nf) returns the variable number of the number field defining polynomial.

long nf_get_r1(GEN nf) returns the number of real places $r_1$.

long nf_get_r2(GEN nf) returns the number of complex places $r_2$.

void nf_get_sign(GEN nf, long *r1, long *r2) sets $r_1$ and $r_2$ to the number of real and complex places respectively. Note that $r_1 + 2r_2$ is the field degree.

long nf_get_degree(GEN nf) returns the number field degree, $n = r_1 + 2r_2$.

GEN nf_get_disc(GEN nf) returns the field discriminant.

GEN nf_get_index(GEN nf) returns the index of $T$, i.e. the index of the order generated by the power basis $(1, x, \ldots, x^{n-1})$ in the maximal order of $K$.
GEN nf_get_zk(GEN nf) returns a basis \((w_1, w_2, \ldots, w_n)\) for the maximal order of \(K\). Those are polynomials in \(\mathbb{Q}[x]\) of degree < \(n\); it is guaranteed that \(w_1 = 1\).

GEN nf_get_invzk(GEN nf) returns the matrix \((m_{i,j}) \in M_n(\mathbb{Z})\) giving the power basis \((x^i)\) in terms of the \((w_j)\), i.e. such that \(x^{j-1} = \sum_{i=1}^{n} m_{i,j} w_i\) for all \(1 \leq j \leq n\); since \(w_1 = 1 = x^0\), we have \(m_{i,1} = \delta_{i,1}\) for all \(i\). The conversion functions in the \texttt{algtobasis} family essentially amount to a left multiplication by this matrix.

GEN nf_get_roots(GEN nf) returns the \(r_1\) real roots of the polynomial defining the number fields: first the \(r_1\) real roots (as \texttt{t_REAL}s), then the \(r_2\) representatives of the pairs of complex conjugates.

GEN nf_get_allroots(GEN nf) returns all the complex roots of \(T\): first the \(r_1\) real roots (as \texttt{t_REAL}s), then the \(r_2\) pairs of complex conjugates.

GEN nf_get_M(GEN nf) returns the \((r_1 + r_2) \times n\) matrix \(M\) giving the embeddings of \(K\): \(M[i,j]\) contains \(w_j(\alpha_i)\), where \(\alpha_i\) is the \(i\)-th element of \(\texttt{nf_get_roots(nf)}\). In particular, if \(v\) is an \(n\)-th dimensional \texttt{t_COL} representing the element \(\sum_{i=1}^{n} v[i] w_i\) of \(K\), then \(\text{RgM}_{\text{RgC}} \text{mul}(M, v)\) represents the embeddings of \(v\).

GEN nf_get_G(GEN nf) returns a \(n \times n\) real matrix \(G\) such that \(G v \cdot G v = T_2(v)\), where \(v\) is an \(n\)-th dimensional \texttt{t_COL} representing the element \(\sum_{i=1}^{n} v[i] w_i\) of \(K\) and \(T_2\) is the standard Euclidean form on \(K \otimes \mathbb{R}\), i.e. \(T_2(v) = \sum_{\sigma} |\sigma(v)|^2\), where \(\sigma\) runs through all \(n\) complex embeddings of \(K\).

GEN nf_get_roundG(GEN nf) returns a rescaled version of \(G\), rounded to nearest integers, specifically \(\text{RM\_round\_maxrank}(G)\).

GEN nf_get_Tr(GEN nf) returns the matrix of the Trace quadratic form on the basis \((w_1, \ldots, w_n)\): its \((i,j)\) entry is \(T(w_i w_j)\).

GEN nf_get_diff(GEN nf) returns the primitive part of the inverse of the above Trace matrix.

long nf_get_prec(GEN nf) returns the precision (in words) to which the \(nf\) was computed.

### 12.1.3 Extracting info from a \texttt{bnf} structure.

These functions expect a true \texttt{bnf} argument, e.g. a \texttt{bnr} will not work.

GEN bnf_get_nf(GEN bnf) returns the underlying \(nf\).

GEN bnf_get_clgp(GEN bnf) returns the class group in \(bnf\), which is a 3-component vector \([h, cyc, gen]\).

GEN bnf_get_cyc(GEN bnf) returns the elementary divisors of the class group (cyclic components) \([d_1, \ldots, d_k]\), where \(d_k | \ldots | d_1\).

GEN bnf_get_gen(GEN bnf) returns the generators \([g_1, \ldots, g_k]\) of the class group. Each \(g_i\) has order \(d_i\), and the full module of relations between the \(g_i\) is generated by the \(d_ig_i = 0\).

GEN bnf_get_no(GEN bnf) returns the class number.

GEN bnf_get_reg(GEN bnf) returns the regulator.

GEN bnf_get_logfu(GEN bnf) returns (complex floating point approximations to) the logarithms of the complex embeddings of our system of fundamental units.

GEN bnf_get_fu(GEN bnf) returns the fundamental units. Raise an error if the \texttt{bnf} does not contain units in algebraic form.

211
GEN bnf_get_fu_nocheck(GEN bnf) as bnf_get_fu without checking whether units are present. Do not use this unless you initialize the bnf yourself!

GEN bnf_get_tuU(GEN bnf) returns a generator of the torsion part of $\mathbb{Z}_K^\times$.

long bnf_get_tuN(GEN bnf) returns the order of the torsion part of $\mathbb{Z}_K^\times$, i.e. the number of roots of unity in $K$.

### 12.1.4 Extracting info from a bnr structure.

These functions expect a true bnr argument.

GEN bnr_get_bnf(GEN bnr) returns the underlying bnf.

GEN bnr_get_nf(GEN bnr) returns the underlying nf.

GEN bnr_get_clgp(GEN bnr) returns the ray class group.

GEN bnr_get_no(GEN bnr) returns the ray class number.

GEN bnr_get_cyc(GEN bnr) returns the elementary divisors of the ray class group (cyclic components) $[d_1, \ldots, d_k]$, where $d_k | \ldots | d_1$.

GEN bnr_get_gen(GEN bnr) returns the generators $[g_1, \ldots, g_k]$ of the ray class group. Each $g_i$ has order $d_i$, and the full module of relations between the $g_i$ is generated by the $d_ig_i = 0$. Raise a generic error if the bnr does not contain the ray class group generators.

GEN bnr_get_gen_nocheck(GEN bnr) as bnr_get_gen without checking whether generators are present. Do not use this unless you initialize the bnr yourself!

GEN bnr_get_bid(GEN bnr) returns the bid associated to the bnr modulus.

GEN bnr_get_mod(GEN bnr) returns the modulus associated to the bnr.

### 12.1.5 Extracting info from an rnf structure.

These functions expect a true rnf argument, associated to an extension $L/K$, $K = \mathbb{Q}[y]/(T)$, $L = K[x]/(P)$.

long rnf_get_degree(GEN rnf) returns the relative degree $[L : K]$.

long rnf_get_absdegree(GEN rnf) returns the absolute degree $[L : \mathbb{Q}]$.

long rnf_get_nfdegree(GEN rnf) returns the degree of the base field $[K : \mathbb{Q}]$.

GEN rnf_get_nf(GEN rnf) returns the base field $K$, an nf structure.

GEN rnf_get_nfpol(GEN rnf) returns the polynomial $T$ defining the base field $K$.

long rnf_get_nfvarn(GEN rnf) returns the variable $y$ associated to the base field $K$.

void rnf_get_nfzk(GEN rnf, GEN *b, GEN *cb) returns the integer basis of the base field $K$.

GEN rnf_get_pol(GEN rnf) returns the relative polynomial defining $L/K$.

long rnf_get_varn(GEN rnf) returns the variable $x$ associated to $L$.

GEN rnf_get_zk(GEN nf) returns the relative integer basis generating $\mathbb{Z}_L$ as a $\mathbb{Z}_K$-module, as a pseudo-matrix $(A, I)$ in HNF.

GEN rnf_get_disc(GEN rnf) is the output $[d, s]$ of rnfdisc.
GEN rnf_get_index(GEN rnf) is the index ideal \( \mathfrak{J} \).

GEN rnf_get_polabs(GEN rnf) returns an absolute polynomial defining \( L/\mathbb{Q} \).

GEN rnf_get_invzk(GEN rnf) contains \( A^{-1} \), where \((A,I)\) is the chosen pseudo-basis for \( Z_L \) over \( Z_K \).

GEN rnf_get_map(GEN rnf) returns technical data associated to the map \( K \to L \). Currently, this contains data from rnfequation, as well as the polynomials \( T \) and \( P \).

12.1.6 Extracting info from a bid structure.

These functions expect a true bid argument, associated to a modulus \( I \) in a number field \( K \).

GEN bid_get_mod(GEN bid) returns the modulus associated to the bid.

GEN bid_get_grp(GEN bid) returns the Abelian group associated to \((Z_K/I)^*\) modulo associated to the bid.

GEN bid_get_ideal(GEN bid) return the finite part of the bid modulus (an integer ideal).

GEN bid_get_arch(GEN bid) return the Archimedean part of the bid modulus (a vector of real places).

GEN bid_get_no(GEN bid) returns the cardinality of the group \((Z_K/I)^*\).

GEN bid_get_cyc(GEN bid) returns the elementary divisors of the group \((Z_K/I)^* \) (cyclic components) \([d_1, \ldots, d_k] \), where \( d_k | \ldots | d_1 \).

GEN bid_get_gen(GEN bid) returns the generators of \((Z_K/I)^* \) contained in bid. Raise a generic error if bid does not contain generators.

GEN bid_get_gen_nocheck(GEN bid) as bid_get_gen without checking whether generators are present. Do not use this unless you initialize the bid yourself!

12.1.7 Increasing accuracy.

GEN nfnewprec(GEN x, long prec). Raise an exception if x is not a number field structure (nf, bnf or bnr). Otherwise, sets its accuracy to prec and return the new structure. This is mostly useful with prec larger than the accuracy to which x was computed, but it is also possible to decrease the accuracy of x (truncating relevant components, which may speed up later computations). This routine may modify the original x (see below).

This routine is straightforward for nf structures, but for the other ones, it requires all principal ideals corresponding to the bnf relations in algebraic form (they are originally only available via floating point approximations). This in turn requires many calls to bnfisprincipal0, which is often slow, and may fail if the initial accuracy was too low. In this case, the routine will not actually fail but recomputes a bnf from scratch!

Since this process may be very expensive, the corresponding data is cached (as a clone) in the original x so that later precision increases become very fast. In particular, the copy returned by nfnewprec also contains this additional data.

GEN bnfnewprec(GEN x, long prec). As nfnewprec, but extracts a bnf structure form x before increasing its accuracy, and returns only the latter.

GEN bnnrnewprec(GEN x, long prec). As nfnewprec, but extracts a bnr structure form x before increasing its accuracy, and returns only the latter.

213
GEN nfnewprec_shallow(GEN nf, long prec)
GEN bnfnewprec_shallow(GEN bnf, long prec)
GEN bnrnewprec_shallow(GEN bnr, long prec) Shallow functions underlying the above, except that the first argument must now have the corresponding number field type. I.e. one cannot call nfnewprec_shallow(nf, prec) if nf is actually a bnf.

12.1.8 Number field arithmetic. The number field $K = \mathbb{Q}(X)/(T)$ is represented by an nf (or bnf or bnr structure). An algebraic number belonging to $K$ is given as

- a t_INT, t_FRAC or t_POL (implicitly modulo $T$), or
- a t_POLMOD (modulo $T$), or
- a t_COL $v$ of dimension $N = [K : \mathbb{Q}]$, representing the element in terms of the computed integral basis $(e_i)$, as

  $\sum(i = 1, N, v[i] \times nf.zk[i])$

The preferred forms are t_INT and t_COL of t_INT. Routines can handle denominators but it is much more efficient to remove denominators first (Q_remove_denom) and take them into account at the end.

Safe routines. The following routines do not assume that their nf argument is a true nf (it can be any number field type, e.g. a bnf), and accept number field elements in all the above forms. They return their result in t_COL form.

GEN nfadd(GEN nf, GEN x, GEN y) returns $x + y$.
GEN nfdiv(GEN nf, GEN x, GEN y) returns $x/y$.
GEN nfinv(GEN nf, GEN x) returns $x^{-1}$.
GEN nfmul(GEN nf, GEN x, GEN y) returns $xy$.
GEN nfpow(GEN nf, GEN x, GEN y) returns $x^k$, $k$ is in $\mathbb{Z}$.
GEN nfpow_u(GEN nf, GEN x, ulong k) returns $x^k$, $k \geq 0$.
GEN nfsqr(GEN nf, GEN x) returns $x^2$.

long nfval(GEN nf, GEN x, GEN pr) returns the valuation of $x$ at the maximal ideal $p$ associated to the prid $pr$. Returns LONG_MAX is $x$ is 0.

GEN nfnorm(GEN nf, GEN x) absolute norm of $x$.
GEN ntrace(GEN nf, GEN x) absolute trace of $x$.
GEN npoleval(GEN nf, GEN pol, GEN a) evaluate the t_POL pol (with coefficients in nf) on the algebraic number $a$ (also in nf).
GEN FpX_FpC_nfpoleval(GEN nf, GEN pol, GEN a, GEN p) evaluate the FpX pol on the algebraic number $a$ (also in nf).

The following three functions implement trivial functionality akin to Euclidean division for which we currently have no real use. Of course, even if the number field is actually Euclidean, these do not in general implement a true Euclidean division.
GEN nfdivuc(GEN nf, GEN a, GEN b) returns the algebraic integer closest to \( x/y \). Functionally identical to ground( nfdiv(nf,x,y) ).

GEN nfdivrem(GEN nf, GEN a, GEN b) returns the vector \([q,r]\), where
\[
q = \text{nfdiveuc}(nf, a, b);
\]
\[
r = \text{nfadd}(nf, a, \text{nfmul}(nf, q, \text{gneg}(b))); \quad \text{or} \quad r = \text{nmod}(nf,a,b);
\]

GEN nfmod(GEN nf, GEN a, GEN b) returns \( r \) such that
\[
q = \text{nfdiveuc}(nf, a,b);
\]
\[
r = \text{nfadd}(nf, a, \text{nfmul}(nf, q, \text{gneg}(b)));
\]

GEN nf_to_scalar_or_basis(GEN nf, GEN x) let \( x \) be a number field element. If it is a rational scalar, i.e. can be represented by a t_INT or t_FRAC, return the latter. Otherwise returns its basis representation (nfalgtobasis). Shallow function.

GEN nf_to_scalar_or_alg(GEN nf, GEN x) let \( x \) be a number field element. If it is a rational scalar, i.e. can be represented by a t_INT or t_FRAC, return the latter. Otherwise returns its lifted t_POLMOD representation (lifted nfbasistoalg). Shallow function.

GEN RgX_to_nfX(GEN nf, GEN x) let \( x \) be a t_POL whose coefficients are number field elements; apply nf to scalar or basis to each coefficient and return the resulting new polynomial. Shallow function.

GEN RgM_to_nfM(GEN nf, GEN x) let \( x \) be a t_MAT whose coefficients are number field elements; apply nf to scalar or basis to each coefficient and return the resulting new matrix. Shallow function.

GEN RgC_to_nfC(GEN nf, GEN x) let \( x \) be a t_COL or t_VEC whose coefficients are number field elements; apply nf to scalar or basis to each coefficient and return the resulting new t_COL. Shallow function.

Unsafe routines. The following routines assume that their nf argument is a true nf (e.g. a bnf is not allowed) and their argument are restricted in various ways, see the precise description below.

GEN nfinvmodideal(GEN nf, GEN x, GEN A) given an algebraic integer \( x \) and a non-zero integral ideal \( A \) in HNF, returns a \( y \) such that \( xy \equiv 1 \) modulo \( A \).

GEN npowmodideal(GEN nf, GEN x, GEN n, GEN ideal) given an algebraic integer \( x \), an integer \( n \), and a non-zero integral ideal \( A \) in HNF, returns an algebraic integer congruent to \( x^n \) modulo \( A \).

GEN nmuli(GEN nf, GEN x, GEN y) returns \( x \times y \) assuming that both \( x \) and \( y \) are either t_INTs or ZVs of the correct dimension.

GEN nsqri(GEN nf, GEN x) returns \( x^2 \) assuming that \( x \) is a t_INT or a ZV of the correct dimension.

GEN nC_nf_mul(GEN nf, GEN v, GEN x) given a t_VEC or t_COL \( v \) of elements of \( K \) in t_INT, t_FRAC or t_COL form, multiply it by the element \( x \) (arbitrary form). This is faster than multiplying coordinatewise since pre-computations related to \( x \) (computing the multiplication table) are done only once. The components of the result are in most cases t_COLS but are allowed to be t_INTs or t_FRACs. Shallow function.

GEN zk_multable(GEN nf, GEN x) given a ZC \( x \) (implicitly representing an algebraic integer), returns the ZM giving the multiplication table by \( x \). Shallow function (the first column of the result points to the same data as \( x \)).
GEN zk_scalar_or_multable(GEN nf, GEN x) given a t_INT or ZC x, returns a t_INT equal to x if the latter is a scalar (t_INT or ZV_isscalar(x) is 1) and zk_multable(nf, x) otherwise. Shallow function.

The following routines implement multiplication in a commutative $R$-algebra, generated by $(e_1 = 1, \ldots, e_n)$, and given by a multiplication table $M$: elements in the algebra are $n$-dimensional t_COLs, and the matrix $M$ is such that for all $1 \leq i, j \leq n$, its column with index $(i - 1)n + j$, say $(c_k)$, gives $e_i \cdot e_j = \sum c_k e_k$. It is assumed that $e_1$ is the neutral element for the multiplication (a convenient optimization, true in practice for all multiplications we needed to implement). If $x$ has any other type than t_COL where an algebra element is expected, it is understood as $xe_1$.

GEN multable(GEN M, GEN x) given a column vector $x$, representing the quantity $\sum_{i=1}^{N} x_i e_i$, returns the multiplication table by $x$. Shallow function.

GEN ei_multable(GEN M, long i) returns the multiplication table by the $i$-th basis element $e_i$. Shallow function.

GEN tablemul(GEN M, GEN x, GEN y) returns $x \cdot y$.

GEN tablesqr(GEN M, GEN x) returns $x^2$.

GEN tablemul_ei(GEN M, GEN x, long i) returns $x \cdot e_i$.

GEN tablemul_ei_ej(GEN M, long i, long j) returns $e_i \cdot e_j$.

GEN tablemulvec(GEN M, GEN x, GEN v) given a vector $v$ of elements in the algebra, returns the $x \cdot v[i]$.

12.1.9 Elements in factored form.

Computational algebraic theory performs extensively linear algebra on $\mathbb{Z}$-modules with a natural multiplicative structure ($K^*$, fractional ideals in $K$, $\mathbb{Z}_K^*$, ideal class group), thereby raising elements to horrendously large powers. A seemingly innocuous elementary linear algebra operation like $C_i \leftarrow C_i - 10000 C_1$ involves raising entries in $C_1$ to the 10000-th power. Understandably, it is often more efficient to keep elements in factored form rather than expand every such expression. A factorization matrix (or famat) is a two column matrix, the first column containing elements (arbitrary objects which may be repeated in the column), and the second one contains exponents (t_INTs, allowed to be 0). By abuse of notation, the empty matrix cgetg(1, t_MAT) is recognized as the trivial factorization (no element, no exponent).

Even though we think of a famat with columns $g$ and $e$ as one meaningful object when fully expanded as $\prod g[i]^{e[i]}$, famats are basically about concatenating information to keep track of linear algebra: the objects stored in a famat need not be operation-compatible, they will not even be compared to each other (with one exception: famat_reduce). Multiplying two famats just concatenates their elements and exponents columns. In a context where a famat is expected, an object $x$ which is not of type t_MAT will be treated as the factorization $x^1$. The following functions all return famats:

GEN famat_mul(GEN f, GEN g) $f, g$ are famat, or objects whose type is not t_MAT (understood as $f^1$ or $g^1$). Returns $fg$. The empty factorization is the neutral element for famat multiplication.

GEN famat_mul_shallow(GEN f, GEN g) $f, g$ are famat, returns $fg$. Shallow function.

GEN famat_pow(GEN f, GEN n) $n$ is a t_INT. If $f$ is a t_MAT, assume it is a famat and return $f^n$ (multiplies the exponent column by $n$). Otherwise, understand it as an element and returns the 1-line famat $f^n$. 

216
GEN famat_sqr(GEN f) returns \( f^2 \).
GEN famat_inv(GEN f) returns \( f^{-1} \).
GEN famat_inv_shallow(GEN f) shallow version of famat_inv.
GEN to_famat(GEN x, GEN k) given an element \( x \) and an exponent \( k \), returns the \( \text{famat} \ x^k \).
GEN to_famat_shallow(GEN x, GEN k) same, as a shallow function.

Note that it is trivial to break up a famat into its two constituent columns: \( \text{gel}(f,1) \) and \( \text{gel}(f,2) \) are the elements and exponents respectively. Conversely, \text{mkmat2} \ builds a (shallow) famat from two \text{t_COL}s of the same length.

The last two functions makes an assumption about the elements: they must be regular algebraic numbers (not famats) over a given number field:
GEN famat_reduce(GEN f) given a famat \( f \), returns a famat \( g \) without repeated elements or 0 exponents, such that the expanded forms of \( f \) and \( g \) would be equal. Shallow function.
GEN ZM_famat_limit(GEN f, GEN limit) given a famat \( f \) with \text{t_INT} \ entries, returns a famat \( g \) with all factors larger than limit multiplied out as the last entry (with exponent 1).
GEN famat_to_nf(GEN nf, GEN f) You normally never want to do this! This is a simplified form of \text{nffactorback}, where we do not check the user input for consistency.

The description of famat_to_nf says that you do not want to use this function. Then how do we recover genuine number field elements? Well, in most cases, we do not need to: most of the functions useful in this context accept \text{famats} as inputs, for instance \text{nfsign}, \text{nfsign_arch}, \text{ideallog} and \text{bnfisunit}. Otherwise, we can generally make good use of a quotient operation (modulo a fixed conductor, modulo \ell-th powers); see the end of Section 12.1.19.

Caveat. Receiving a famat input, \text{bnfisunit} assumes that it is an algebraic integer, since this is expensive to check, and normally easy to ensure from the user’s side; do not feed it ridiculous inputs.
GEN famatsmall_reduce(GEN f) as famat_reduce, but for exponents given by a \text{t_VECSMALL}.

12.1.10 Ideal arithmetic.

Conversion to HNF.
GEN idealhnf(GEN nf, GEN x) returns the HNF of the ideal defined by \( x \): \( x \) may be an algebraic number (defining a principal ideal), a maximal ideal (as given by \text{idealprimedec} or \text{idealfactor}), or a matrix whose columns give generators for the ideal. This last format is complicated, but useful to reduce general modules to the canonical form once in a while:

- if strictly less than \( N = [K : Q] \) generators are given, \( x \) is the \( \mathbb{Z}_K \)-module they generate,
- if \( N \) or more are given, it is assumed that they form a \( \mathbb{Z} \)-basis (that the matrix has maximal rank \( N \)). This acts as \text{mathnf} since the \( \mathbb{Z}_K \)-module structure is (taken for granted hence) not taken into account in this case.

Extended ideals are also accepted, their principal part being discarded.
GEN idealhnf0(GEN nf, GEN x, GEN y) returns the HNF of the ideal generated by the two algebraic numbers \( x \) and \( y \).
The following low-level functions underlie the above two: they all assume that \( nf \) is a true \( nf \) and perform no type checks:

\[
\text{GEN idealhnf_principal(GEN nf, GEN x)} \quad \text{returns the ideal generated by the algebraic number } x.
\]

\[
\text{GEN idealhnf_shallow(GEN nf, GEN x)} \quad \text{is idealhnf except that the result may not be suitable for gerepile: if } x \text{ is already in HNF, we return } x, \text{ not a copy!}
\]

\[
\text{GEN idealhnf_two(GEN nf, GEN v)} \quad \text{assuming } a = v[1] \text{ is a non-zero t_INT and } b = v[2] \text{ is an algebraic integer, possibly given in regular representation by a t_MAT (the multiplication table by } b, \text{ see zk_multable), returns the HNF of } a\mathbb{Z}_K + b\mathbb{Z}_K.
\]

Operations.

The basic ideal routines accept all \( nf s (nf, bnf, bnr) \) and ideals in any form, including extended ideals, and return ideals in HNF, or an extended ideal when that makes sense:

\[
\text{GEN idealadd(GEN nf, GEN x, GEN y)} \quad \text{returns } x + y.
\]

\[
\text{GEN idealdiv(GEN nf, GEN x, GEN y)} \quad \text{returns } x/y. \text{ Returns an extended ideal if } x \text{ or } y \text{ is an extended ideal.}
\]

\[
\text{GEN idealmul(GEN nf, GEN x, GEN y)} \quad \text{returns } xy. \text{ Returns an extended ideal if } x \text{ or } y \text{ is an extended ideal.}
\]

\[
\text{GEN idealsqr(GEN nf, GEN x)} \quad \text{returns } x^2. \text{ Returns an extended ideal if } x \text{ is an extended ideal.}
\]

\[
\text{GEN idealinv(GEN nf, GEN x)} \quad \text{returns } x^{-1}. \text{ Returns an extended ideal if } x \text{ is an extended ideal.}
\]

\[
\text{GEN idealpow(GEN nf, GEN x, GEN n)} \quad \text{returns } x^n. \text{ Returns an extended ideal if } x \text{ is an extended ideal.}
\]

\[
\text{GEN idealpows(GEN nf, GEN ideal, long n)} \quad \text{returns } x^n. \text{ Returns an extended ideal if } x \text{ is an extended ideal.}
\]

\[
\text{GEN idealprodprime(GEN nf, GEN P)} \quad \text{given a list } P \text{ of prime ideals in prid form, return their product.}
\]

\[
\text{GEN idealmul_HNF(GEN nf, GEN x, GEN y)} \quad \text{returns } xy, \text{ assuming than } nf \text{ is a true } nf, x \text{ is an integral ideal in HNF and } y \text{ is an integral ideal in HNF or precompiled form (see below). For}
\]

218
maximal speed, the second ideal \( y \) may be given in precompiled form \( y = [a, b] \), where \( a \) is a non-zero \( t \_\text{INT} \) and \( b \) is an algebraic integer in regular representation (a \( t \_\text{MAT} \) giving the multiplication table by the fixed element): very useful when many ideals \( x \) are going to be multiplied by the same ideal \( y \). This essentially reduces each ideal multiplication to an \( N \times N \) matrix multiplication followed by a \( N \times 2N \) modular HNF reduction (modulo \( xy \cap \mathbb{Z} \)).

Approximation.

\[
\text{GEN idealaddtoone(GEN } \text{nf, GEN } A, \text{ GEN } B) \text{ given to coprime integer ideals } A, B, \text{ returns } [a, b] \text{ with } a \in A, \ b \in B, \text{ such that } a + b = 1 \text{ The result is reduced mod } AB, \text{ so } a, b \text{ will be small.}
\]

\[
\text{GEN hnfmerge_get_1(GEN } \text{nf, GEN } A, \text{ GEN } B) \text{ as idealaddtoone except that } nf \text{ must be a true } nf, \text{ and only } a \text{ is returned.}
\]

\[
\text{GEN hnfmerge_get_1(GEN } A, \text{ GEN } B) \text{ given two square upper HNF integral matrices } A, B \text{ of the same dimension } n > 0, \text{ return } a \text{ in the image of } A \text{ such that } 1 - a \text{ is in the image of } B. \text{ (By abuse of notation we denote } 1 \text{ the column vector } [1, 0, \ldots, 0].) \text{ If such an } a \text{ does not exist, return NULL. This is the function underlying idealaddtoone.}
\]

\[
\text{GEN idealaddmultoone(GEN } \text{nf, GEN } v) \text{ given a list of } n \text{ (globally) coprime integer ideals } (v[i]) \text{ returns an } n\text{-dimensional vector } a \text{ such that } a[i] \in v[i] \text{ and } \sum a[i] = 1. \text{ If } [K : Q] = N, \text{ this routine computes the HNF reduction (with } GL_{nN}(\mathbb{Z}) \text{ base change) of an } N \times nN \text{ matrix; so it is well worth pruning "useless" ideals from the list (as long as the ideals remain globally coprime).}
\]

\[
\text{GEN idealappr(GEN } \text{nf, GEN } x) \text{ given a fractional ideal } x, \text{ returns an algebraic number } \alpha \text{ such that } v(x) = v(\alpha) \text{ for all valuations such that } v(x) > 0, \text{ and } v(\alpha) \geq 0 \text{ at all others.}
\]

\[
\text{GEN idealapprfact(GEN } \text{nf, GEN } fx) \text{ same as idealappr, } x \text{ being given in factored form, as after } fx = \text{ idealfactor(nf,x), except that we allow 0 exponents in the factorization. Returns an algebraic number } \alpha \text{ such that } v(x) = v(\alpha) \text{ for all valuations associated to the prime ideal decomposition of } x, \text{ and } v(\alpha) \geq 0 \text{ at all others.}
\]

\[
\text{GEN idealcoprime(GEN } \text{nf, GEN } x, \text{ GEN } y) \text{. Given } 2 \text{ integral ideals } x \text{ and } y, \text{ returns an algebraic number } \alpha \text{ such that } \alpha x \text{ is an integral ideal coprime to } y.
\]

\[
\text{GEN idealcoprimefact(GEN } \text{nf, GEN } x, \text{ GEN } fy) \text{ same as idealcoprime, except that } y \text{ is given in factored form, as from idealfactor.}
\]

\[
\text{GEN idealchinese(GEN } \text{nf, GEN } x, \text{ GEN } y) \text{ } x \text{ being a prime ideal factorization (i.e. a } 2 \text{ by } 2 \text{ matrix whose first column contain prime ideals, and the second column integral exponents), } y \text{ a vector of elements in } nf \text{ indexed by the ideals in } x, \text{ computes an element } b \text{ such that } v_\varphi(b - y_\varphi) \geq v_\varphi(x) \text{ for all prime ideals in } x \text{ and } v_\varphi(b) \geq 0 \text{ for all other } \varphi.
\]

12.1.11 Maximal ideals.

The PARI structure associated to maximal ideals is a \textit{prid} (for \textit{prime ideal}), usually produced by \texttt{idealprimedec} and \texttt{idealfactor}. In this section, we describe the format; other sections will deal with their daily use.

A \textit{prid} associated to a maximal ideal \( p \) stores the following data: the underlying rational prime \( p \), the ramification degree \( e \geq 1 \), the residue field degree \( f \geq 1 \), a \( p \)-uniformizer \( \pi \) with valuation \( 1 \) at \( p \) and valuation \( 0 \) at all other primes dividing \( p \) and a rescaled “anti-uniformizer” \( \tau \) used to compute valuations. This \( \tau \) is an algebraic integer such that \( \tau/p \) has valuation \( -1 \) at \( p \) and valuation \( 0 \) at all other primes dividing \( p \); in particular, the valuation of \( x \in \mathbb{Z}_K \) is positive if and only if the algebraic integer \( xt \) is divisible by \( p \) (easy to check for elements in \( t \_\text{COL} \) form).
The following functions are shallow and return directly components of the prid pr:

- **GEN pr_get_p(GEN pr)** returns \( p \). Shallow function.
- **GEN pr_get_gen(GEN pr)** returns \( \pi \). Shallow function.
- **long pr_get_e(GEN pr)** returns \( e \).
- **long pr_get_f(GEN pr)** returns \( f \).
- **GEN pr_get_tau(GEN pr)** returns \( z[k] \), scalar or multable (\( nf, \tau \)), which is the t_INT 1 iff \( p \) is inert, and a ZM otherwise. Shallow function.
- **int pr_is_inert(GEN pr)** returns 1 if \( p \) is inert, 0 otherwise.
- **GEN pr_norm(GEN pr)** returns the norm \( p^f \) of the maximal ideal.

### 12.1.12 Reducing modulo maximal ideals.

- **GEN nfmodprinit(GEN nf, GEN pr)** returns an abstract modpr structure, associated to reduction modulo the maximal ideal \( pr \), in idealprimedec format. From this data we can quickly project any \( pr \)-integral number field element to the residue field. This function is almost useless in library mode, we rather use:

  - **GEN nf_to_Fq_init(GEN nf, GEN *ppr, GEN *pT, GEN *pp)** concrete version of nfmodprinit: \( nf \) and \( *ppr \) are the inputs, the return value is a modpr and \( *ppr, *pT \) and \( *pp \) are set as side effects.

  The input \( *ppr \) is either a maximal ideal or already a modpr (in which case it is replaced by the underlying maximal ideal). The residue field is realized as \( F_p[X]/(T) \) for some monic \( T \in F_p[X] \), and we set \( *pT \) to \( T \) and \( *pp \) to \( p \). Set \( T = \text{NULL} \) if the prime has degree 1 and the residue field is \( F_p \).

  In short, this receives (or initializes) a modpr structure, and extracts from it \( T, p \) and \( p \).

- **GEN nf_to_Fq(GEN nf, GEN x, GEN modpr)** returns an \( F_q \) congruent to \( x \) modulo the maximal ideal associated to modpr. The output is canonical: all elements in a given residue class are represented by the same \( F_q \).

- **GEN Fq_to_nf(GEN x, GEN modpr)** returns an \( nf \) element lifting the residue field element \( x \), either a t_INT or an algebraic integer in algtobasis format.

- **GEN modpr_genFq(GEN modpr)** Returns an \( nf \) element whose image by \( nf\_to\_Fq \) is \( X \pmod T \), if \( \deg T > 1 \), else 1.

- **GEN zkmodprinit(GEN nf, GEN pr)** as nfmodprinit, but we assume we will only reduce algebraic integers, hence do not initialize data allowing to remove denominators. More precisely, we can in fact still handle an \( x \) whose rational denominator is not 0 in the residue field (i.e. if the valuation of \( x \) is non-negative at all primes dividing \( p \)).

- **GEN zk_to_Fq_init(GEN nf, GEN *pr, GEN *T, GEN *p)** as \( nf\_to\_Fq\_init \), able to reduce only \( p \)-integral elements.

- **GEN zk_to_Fq(GEN x, GEN modpr)** as \( nf\_to\_Fq \), for a \( p \)-integral \( x \).

- **GEN nfM_to_FqM(GEN M, GEN nf, GEN modpr)** reduces a matrix of \( nf \) elements to the residue field; returns an \( FqM \).

- **GEN FqM_to_nfM(GEN M, GEN modpr)** lifts an \( FqM \) to a matrix of \( nf \) elements.
GEN nfV_to_FqV(GEN A, GEN nf, GEN modpr) reduces a vector of nf elements to the residue field; returns an FqV with the same type as A (t_VEC or t_COL).

GEN FqV_to_nfV(GEN A, GEN modpr) lifts an FqV to a vector of nf elements (same type as A).

GEN nfX_to_FqX(GEN Q, GEN nf, GEN modpr) reduces a polynomial with nf coefficients to the residue field; returns an FqX.

GEN FqX_to_nfX(GEN Q, GEN modpr) lifts an FqX to a polynomial with coefficients in nf.

12.1.13 Valuations.

long nfval(GEN nf, GEN x, GEN P) return \( v_P(x) \)

Unsafe functions. assume \( nf \) is a genuine nf structure, that \( P, Q \) are prid.

long ZC_nfval(GEN nf, GEN x, GEN P) returns \( v_P(x) \), assuming \( x \) is a ZC, representing a non-zero algebraic integer.

long ZC_nfvalrem(GEN nf, GEN x, GEN pr, GEN *newx) returns \( v_P = v_P(x) \), assuming \( x \) is a ZC, representing a non-zero algebraic integer, and sets \( *newx \) to \( x \tau^v \) which is an algebraic integer coprime to \( p \).

int ZC_prdvd(GEN nf, GEN x, GEN P) returns 1 is \( P \) divides \( x \) and 0 otherwise. Assumes that \( x \) is a ZC, representing an algebraic integer. Faster than computing \( v_P(x) \).

int pr_equal(GEN nf, GEN P, GEN Q) returns 1 is \( P \) and \( Q \) represent the same maximal ideal: they must lie above the same \( p \) and share the same \( e, f \) invariants, but the \( p \)-uniformizer and \( \tau \) element may differ. Returns 0 otherwise.

12.1.14 Signatures.

"Signs" of the real embeddings of number field element are represented in additive notation, using the standard identification \( (\mathbb{Z}/2\mathbb{Z}, +) \rightarrow \{-1, 1\}, \times, s \mapsto (-1)^s \).

With respect to a fixed \( nf \) structure, a selection of real places (a divisor at infinity) is normally given as a t_VECSMALL of indices of the roots \( nf.roots \) of the defining polynomial for the number field. For compatibility reasons, in particular under GP, the (obsolete) vec01 form is also accepted: a t_VEC with gen 0 or gen 1 entries.

The following internal functions go back and forth between the two representations for the Archimedean part of divisors (GP: 0/1 vectors, library: list of indices):

GEN vec01_to_indices(GEN v) given a t_VEC v with t_INT entries equal to 0 or 1, return as a t_VECSMALL the list of indices \( i \) such that \( v[i] = 1 \). If \( v \) is already a t_VECSMALL, return it (not suitable for gerepile in this case).

GEN indices_to_vec01(GEN p, long n) return the 0/1 vector of length \( n \) with ones exactly at the positions \( p[1], p[2], \ldots \)

GEN nfsign(GEN nf, GEN x) \( x \) being a number field element and \( nf \) any form of number field, return the 0 \(-\) 1-vector giving the signs of the \( r_1 \) real embeddings of \( x \), as a t_VECSMALL. Linear algebra functions like Flv_add_inplace then allow keeping track of signs in series of multiplications.

If \( x \) is a t_VEC of number field elements, return the matrix whose columns are the signs of the \( x[i] \).
GEN nfsign_arch(GEN nf, GEN x, GEN arch) arch being a list of distinct real places, either in vec01(t_VEC with gen_0 or gen_1 entries) or indices (t_VECSMALL) form (see vec01_to_indices), returns the signs of x at the corresponding places. This is the low-level function underlying nfsign.

GEN nfsign_units(GEN bnf, GEN archp, int add_tu) archp being a divisor at infinity in indices form (or NULL for the divisor including all real places), return the signs at archp of a system of fundamental units for the field, in the same order as bnf.tufu if add_tu is set; and in the same order as bnf.fu otherwise.

GEN nfsign_from_logarch(GEN L, GEN invpi, GEN archp) given L the vector of the log σ(x), where σ runs through the (real or complex) embeddings of some number field, invpi being a floating point approximation to 1/π, and archp being a divisor at infinity in indices form, return the signs of x at the corresponding places. This is the low-level function underlying nfsign_units; the latter is actually a trivial wrapper bnf structures include the log σ(x) for a system of fundamental units of the field.

GEN set_sign_mod_divisor(GEN nf, GEN x, GEN y, GEN module, GEN sarch) let $f = f_0 f_\infty$ be the divisor represented by module, x, y two number field elements. Returns $yt$ with $t \equiv 1 \mod f_\infty$ such that $x$ and $ty$ have the same signs at $f_\infty$; if $x = \text{NULL}$, make $ty$ totally positive at $f_\infty$. sarch is the output of nfarchstar(nf, f0, finf).

GEN nfarchstar(GEN nf, GEN f0, GEN finf) for a divisor $f = f_0 f_\infty$ represented by the integral ideal $f_0$ in HNF and the $f_\infty$ in indices form, returns $(\mathbb{Z}_K/f_\infty)^*$ in a form suitable for computations mod $f$. More precisely, returns $[c,g,M]$, where $c = [2,...,2]$ gives the cyclic structure of that group ($#f_\infty$ copies of $\mathbb{Z}/2\mathbb{Z}$), $g$ a minimal system of independent generators, which are furthermore congruent to 1 mod $f_0$ (no condition if $f_0 = \mathbb{Z}_K$), and $M$ is the matrix of signs of the $g[i]$ at $f_\infty$, which is square and invertible over $\mathbb{F}_2$.

GEN idealprincipalunits(GEN nf, GEN pr, long e) returns the multiplicative group $(1 + pr)/(1 + pr^e)$ as an abelian group. Faster than idealstar when the norm of pr is large, since it avoids (useless) work in the multiplicative group of the residue field.

12.1.15 Maximal order and discriminant.

A number field $K = \mathbb{Q}[X]/(T)$ is defined by a monic $T \in \mathbb{Z}[X]$. The low-level function computing a maximal order is

void nfmaxord(nfmaxord_t *S, GEN T0, long flag), where the polynomial $T_0$ is squarefree with integer coefficients. Let $K$ be the étale algebra $\mathbb{Q}[X]/(T_0)$ and let $T = ZX.Q.normalize(T_0)$, i.e. $T = CT_0(X/L)$ is monic and integral for some $C, Q \in \mathbb{Q}$.

The structure nfmaxord_t is initialized by the call; it has the following fields:

GEN T0, T, dT, dK; /* T0, T, discriminants of T and K */
GEN unscale; /* the integer L */
GEN index; /* index of power basis in maximal order */
GEN dTP, dTE; /* factorization of |dT|, primes / exponents */
GEN dKP, dKE; /* factorization of |dK|, primes / exponents */
GEN basis; /* Z-basis for maximal order of Q[X]/(T) */

The exponent vectors are t_VECSMALL. The primes in dTP and dKP are pseudoprimes, not proven primes. We recommend restricting to $T = T_0$, i.e. either to pass the input polynomial through ZX.Q.normalize before the call, or to forget about $T_0$ and go on with the polynomial $T$; otherwise
unscale \neq 1$, all data is expressed in terms of $T \neq T_0$, and needs to be converted to $T_0$. For instance to convert the basis to $\mathbb{Q}[X]/(T_0)$:

$$\text{RgXV unscale}(S.\text{basis}, S.\text{unscale})$$

Instead of passing $T$, one can use the format $[T, \text{listP}]$ as in \texttt{nfbasis} or \texttt{nfini}t, which computes an order which is maximal at a set of primes, but need not be the maximal order.

The \texttt{flag} is an or-ed combination of the binary flags:

\texttt{nf\_PARTIALFACT}: do not try to fully factor $dT$ and only look for primes less than \texttt{primelimit}. In that case, the elements in $dT$ and $dKP$ need not all be primes. But the resulting $dK$, \texttt{index} and \texttt{basis} are correct provided there exists no prime $p > \texttt{primelimit}$ such that $p^2$ divides the field discriminant $dK$. This flag is \texttt{deprecated}: the $[T, \text{listP}]$ is safer and more flexible.

\texttt{nf\_ROUND2}: use the \texttt{ROUND2} algorithm instead of the default \texttt{ROUND4}. This flag is \texttt{deprecated}: this algorithm is consistently slower.

$T$ is the input polynomial (monic ZX). The format $[T, \text{listP}]$ is also recognized, where \texttt{listP} is as in \texttt{nfbasis} and is used to compute a local integral basis with respect to a specific set of primes.

\texttt{GEN indexpartial} (\texttt{GEN T}, \texttt{GEN dT}) $T$ a monic separable ZX, $dT$ is either \texttt{NULL} (no information) or a multiple of the discriminant of $T$. Let $K = \mathbb{Q}[X]/(T)$ and $Z_K$ its maximal order. Returns a multiple of the exponent of the quotient group $Z_K/(Z[X]/(T))$. In other word, a denominator $d$ such that $dx \in Z[X]/(T)$ for all $x \in Z_K$.

12.1.16 Computing in the class group.

We compute with arbitrary ideal representatives (in any of the various formats seen above), and call

\texttt{GEN bnfisprincipal0} (\texttt{GEN bnf}, \texttt{GEN x}, \texttt{long flag}). The \texttt{bnf} structure already contains information about the class group in the form $\oplus_{i=1}^n (Z/d_iZ)g_i$ for canonical integers $d_i$ (with $d_n \mid \ldots \mid d_1$ all > 1) and essentially random generators $g_i$, which are ideals in HNF. We normally do not need the value of the $g_i$, only that they are fixed once and for all and that any (non-zero) fractional ideal $x$ can be expressed uniquely as $x = (t) \prod_{i=1}^n g_i^{e_i}$, where $0 \leq e_i < d_i$, and $(t)$ is some principal ideal. Computing $e$ is straightforward, but $t$ may be very expensive to obtain explicitly. The routine returns (possibly partial) information about the pair $[e, t]$, depending on \texttt{flag}, which is an or-ed combination of the following symbolic flags:

- \texttt{nf\_GEN} tries to compute $t$. Returns $[e, t]$, with $t$ an empty vector if the computation failed. This flag is normally useless in non-trivial situations since the next two serve analogous purposes in more efficient ways.

- \texttt{nf\_GENMAT} tries to compute $t$ in factored form, which is much more efficient than \texttt{nf\_GEN} if the class group is moderately large; imagine a small ideal $x = (t)g^{10000}$; the norm of $t$ has 10000 as many digits as the norm of $g$; do we want to see it as a vector of huge meaningless integers? The idea is to compute $e$ first, which is easy, then compute $(t)$ as $x \prod g_i^{-e_i}$ using successive \texttt{idealmulred}, where the ideal reduction extracts small principal ideals along the way, eventually raised to large powers because of the binary exponentation technique; the point is to keep this principal part in factored \texttt{unexpanded} form. Returns $[e, t]$, with $t$ an empty vector if the computation failed; this should be exceedingly rare, unless the initial accuracy to which \texttt{bnf} was computed was ridiculously low (and then \texttt{bnfinit} should not have succeeded either). Setting/unsetting \texttt{nf\_GEN} has no effect when this flag is set.
• \texttt{nf\_GEN\_IF\_PRINCIPAL} tries to compute \( t \) only if the ideal is principal \((e = 0)\). Returns \texttt{gen\_0} if the ideal is not principal. Setting/unsetting \texttt{nf\_GEN} has no effect when this flag is set, but setting/unsetting \texttt{nf\_GENMAT} is possible.

• \texttt{nf\_FORCE} in the above, insist on computing \( t \), even if it requires recomputing a \texttt{bnf} from scratch. This is a last resort, and normally the accuracy of a \texttt{bnf} can be increased without trouble, but it may be that some algebraic information simply cannot be recovered from what we have: see \texttt{bnfnewprec}. It should be very rare, though.

In simple cases where you do not care about \( t \), you may use

\[
\text{GEN isprincipal(GEN bnf, GEN x), which is a shortcut for bnfisprincipal0(bnf, x, 0).}
\]

The following low-level functions are often more useful:

\[
\begin{align*}
\text{GEN isprincipalfact(GEN bnf, GEN C, GEN L, GEN f, long flag) is about the same as bnfisprincipal0 applied to } & C \prod L[i]/f[i], \text{ where the } L[i] \text{ are ideals, the } f[i] \text{ integers and } C \text{ is either an ideal or NULL (omitted). Make sure to include nf\_GENMAT in flag!} \\
\text{GEN isprincipalfact\_or\_fail(GEN bnf, GEN C, GEN L, GEN f) is for delicate cases, where we must be more clever than nf\_FORCE (it is used when trying to increase the accuracy of a bnf, for instance). If performs } & \\
isprincipalfact(bnf,C, L, f, nf\_GENMAT); \\
\text{but if it fails to compute } t, \text{ it just returns a } t\_\text{INT}, \text{ which is the estimated precision (in words, as usual) that would have been sufficient to complete the computation. The point is that nf\_FORCE does exactly this internally, but goes on increasing the accuracy of the bnf, then discarding it, which is a major inefficiency if you intend to compute lots of discrete logs and have selected a precision which is just too low. (It is sometimes not so bad since most of the really expensive data is cached in bnf anyway, if all goes well.) With this function, the caller may decide to increase the accuracy using bnfnewprec (and keep the resulting bnf!), or avoid the computation altogether. In any case the decision can be taken at the place where it is most likely to be correct.}
\end{align*}
\]

12.1.17 Floating point embeddings, the \( T_2 \) quadratic form.

We assume the \( nf \) is a true \( nf \) structure, associated to a number field \( K \) of degree \( n \) and signature \((r_1, r_2)\). We saw that

\[
\text{GEN nf\_get\_M(GEN nf) returns the } (r_1 + r_2) \times n \text{ matrix } M \text{ giving the embeddings of } K, \text{ so that if } v \text{ is an } n\text{-th dimensional } t\_\text{COL} \text{ representing the element } \sum_{i=1}^n v[i]w_i \text{ of } K, \text{ then } RgM.RgC\_mul(M, v) \text{ represents the embeddings of } v. \text{ Its first } r_1 \text{ components are real numbers (} t\_\text{INT}, t\_\text{FRAC} \text{ or } t\_\text{REAL}, \text{ usually the latter), and the last } r_2 \text{ are complex numbers (usually of } t\_\text{COMPLEX}, \text{ but not necessarily for embeddings of rational numbers).}
\]

\[
\text{GEN embed\_T2(GEN x, long r1) assuming } x \text{ is the vector of floating point embeddings of some algebraic number } v, \text{ i.e.}
\]

\[
x = RgM.RgC\_mul(nf\_get\_M(nf), \text{algtobasis}(nf,v));
\]

returns \( T_2(v) \). If the floating point embeddings themselves are not needed, but only the values of \( T_2 \), it is more efficient to restrict to real arithmetic and use

\[
\text{gnorml2( RgM.RgC\_mul(nf\_get\_G(nf), \text{algtobasis}(nf,v))));}
\]

\[
\text{GEN embednorm\_T2(GEN x, long r1) analogous to embed\_T2, applied to the \text{gnorm of the floating point embeddings. Assuming that}
}\]

224
\[ x = \text{gnorm( RgM_RgC_mul(nf_get_M(nf), algtobasis(nf,v)) )}; \]

returns \( T_2(v) \).

**GEN embed_roots(GEN z, long r1)** given a vector \( z \) of \( r_1 + r_2 \) complex embeddings of the algebraic number \( v \), return the \( r_1 + 2r_2 \) roots of its characteristic polynomial. Shallow function.

**GEN embed_disc(GEN z, long r1, long prec)** given a vector \( z \) of \( r_1 + r_2 \) complex embeddings of the algebraic number \( v \), return a floating point approximation of the discriminant of its characteristic polynomial as a \( \text{t_REAL} \) of precision \( \text{prec} \).

**GEN embed_norm(GEN x, long r1)** given a vector \( z \) of \( r_1 + r_2 \) complex embeddings of the algebraic number \( v \), return (a floating point approximation of) the norm of \( v \).

### 12.1.18 Ideal reduction, low level.

In the following routines \( nf \) is a true \( nf \), associated to a number field \( K \) of degree \( n \):

**GEN nf_get_Gtwist(GEN nf, GEN v)** assuming \( v \) is a \( \text{t_VECSMALL} \) with \( r_1 + r_2 \) entries, let

\[ ||x||_v^2 = \sum_{i=1}^{r_1+r_2} 2^{r_i} \varepsilon_i |\sigma_i(x)|^2, \]

where as usual the \( \sigma_i \) are the (real and) complex embeddings and \( \varepsilon_i = 1, \text{ resp. } 2 \), for a real, resp. complex place. This is a twisted variant of the \( T_2 \) quadratic form, the standard Euclidean form on \( K \otimes \mathbb{R} \). In applications, only the relative size of the \( v_i \) will matter.

Let \( G_v \in M_n(\mathbb{R}) \) be a square matrix such that if \( x \in K \) is represented by the column vector \( X \) in terms of the fixed \( \mathbb{Z} \)-basis of \( \mathbb{Z}_K \) in \( nf \), then

\[ ||x||_v^2 = \langle G_v X, G_v X \rangle. \]

(This is a kind of Cholesky decomposition.) This function returns a rescaled copy of \( G_v \), rounded to nearest integers, specifically \( \text{RM_round_maxrank}(G_v) \). Suitable for \text{gerepileupto}, but does not collect garbage.

**GEN nf_get_Gtwist1(GEN nf, long i)**. Simple special case. Returns the twisted \( G \) matrix associated to the vector \( v \) whose entries are all 0 except the \( i \)-th one, which is equal to 10.

**GEN idealpseudomin(GEN x, GEN G)**. Let \( x, G \) be two \( \mathbb{Z} \)-Vs, such that the product \( Gx \) is well-defined. This returns a “small” integral linear combinations of the columns of \( x \), given by the \text{LLL-algorithm} applied to the lattice \( Gx \). Suitable for \text{gerepileupto}, but does not collect garbage.

In applications, \( x \) is an integral ideal, \( G \) approximates a Cholesky form for the \( T_2 \) quadratic form as returned by \text{nf_get_Gtwist}, and we return a small element \( a \) in the lattice \( (x, T_2) \). This is used to implement \text{idealred}.

**GEN idealpseudomin_nonscalar(GEN x, GEN G)**. As \text{idealpseudomin}, but we insist of returning a non-scalar \( a \) (\( \text{ZV_isscalar} \) is false), if the dimension of \( x \) is \( > 1 \).

In the interpretation where \( x \) defines an integral ideal on a fixed \( \mathbb{Z}_K \) basis whose first element is 1, this means that \( a \) is not rational.

**GEN idealred_elt(GEN nf, GEN x)** shortcut for

\[ \text{idealpseudomin}(x, \text{nf_get_roundG}(nf)) \]
12.1.19 Ideal reduction, high level.

Given an ideal \( x \) this means finding a "simpler" ideal in the same ideal class. The public GP function is of course available

\[ \text{GEN idealred0}(\text{GEN nf}, \text{GEN } x, \text{GEN } v) \]

finds a small \( a \in x \) and returns the primitive part of \( x/a \), as an ideal in HNF. What "small" means depends on the parameter \( v \), see the GP description. More precisely, \( a \) is returned by \( \text{idealpseudomin}(x, G) \), where \( G = \text{nf_get_Gtwist}(nf, v) \) for \( v \neq \text{NULL} \) and \( \text{nf_get_roundG}(nf) \) otherwise.

Usually one sets \( v = \text{NULL} \) to obtain an element of small \( T_2 \) norm in \( x \):

\[ \text{GEN idealred}(\text{GEN nf}, \text{GEN } x) \]

is a shortcut for \( \text{idealred0}(nf,x,\text{NULL}) \).

The function \( \text{idealred} \) remains complicated to use: in order not to lose information \( x \) must be an extended ideal, otherwise the value of \( a \) is lost. There is a subtlety here: the principal ideal \( (a) \) is easy to recover, but \( a \) itself is an instance of the principal ideal problem which is very difficult given only an \( nf \) (once a \( bnf \) structure is available, \( \text{bnfisprincipal0} \) will recover it). It is in general simpler to use directly \( \text{idealred_elt} \).

\[ \text{GEN idealmoddivisor}(\text{GEN bnr}, \text{GEN } x) \]

A proof-of-concept implementation, useless in practice. If \( bnr \) is associated to some modulus \( f \), returns a "small" ideal in the same class as \( x \) in the ray class group modulo \( f \). The reason why this is useless is that using extended ideals with principal part in a computation, there is a simple way to reduce them: simply reduce the generator of the principal part in \( (\mathbb{Z}_K/f)^* \).

\[ \text{GEN famat_to_nf_moddivisor}(\text{GEN nf}, \text{GEN } g, \text{GEN } e, \text{GEN } bid) \]

given a true \( nf \) associated to a number field \( K \), a \( bid \) structure associated to a modulus \( f \), and an algebraic number in factored form \( \prod g[i]^{e[i]} \), such that \((g[i],f) = 1\) for all \( i \), returns a small element in \( \mathbb{Z}_K \) congruent to it mod \( f \). Note that if \( f \) contains places at infinity, this includes sign conditions at the specified places.

A simpler case when the conductor has no place at infinity:

\[ \text{GEN famat_to_nf_modideal_coprime}(\text{GEN nf}, \text{GEN } g, \text{GEN } e, \text{GEN } f, \text{GEN } expo) \]

as above except that the ideal \( f \) is now integral in HNF (no need for a full \( bid \)), and we pass the exponent of the group \( (\mathbb{Z}_K/f)^* \) as \( expo \); any multiple will also do, at the expense of efficiency. Of course if a \( bid \) for \( f \) is available, it is easy to extract \( f \) and the exact value of \( expo \) from it (the latter is the first elementary divisor in the group structure). A useful trick: if you set \( expo \) to any positive integer, the result is correct up to \( expo \)-th powers, hence exact if \( expo \) is a multiple of the exponent; this is useful when trying to decide whether an element is a square in a residue field for instance! (take \( expo = 2 \)).

What to do when the \( g[i] \) are not coprime to \( f \), but only \( \prod g[i]^{e[i]} \) is? Then the situation is more complicated, and we advise to solve it one prime divisor of \( f \) at a time. Let \( v \) the valuation associated to a maximal ideal \( pr \) and assume \( v(f) = k > 0 \):

\[ \text{GEN famat_makecoprime}(\text{GEN nf}, \text{GEN } g, \text{GEN } e, \text{GEN } pr, \text{GEN } prk, \text{GEN } expo) \]

returns an element in \( (\mathbb{Z}_K/pr^k)^* \) congruent to the product \( \prod g[i]^{e[i]} \), assumed to be globally coprime to \( f \). As above, \( expo \) is any positive multiple of the exponent of \( (\mathbb{Z}_K/pr^k)^* \), for instance \((Nv - 1)p^{k-1}\), if \( p \) is the underlying rational prime. You may use other values of \( expo \) (see the useful trick in \text{famat_to_nf_modideal_coprime}).
12.1.20 Class field theory.

Under GP, a class-field theoretic description of a number field is given by a triple \(A, B, C\), where the defining set \([A, B, C]\) can have any of the following forms: \([bnr], [bnr, subgroup], [bnf, modulus], [bnf, modulus, subgroup]\). You can still use directly all of (libpari’s routines implementing) GP’s functions as described in Chapter 3, but they are often awkward in the context of libpari programming. In particular, it does not make much sense to always input a triple \(A, B, C\) because of the fringe \([bnf, modulus, subgroup]\). The first routine to call, is thus

\[
\text{GEN Buchray(GEN bnf, GEN mod, long flag)}\]

initializes a \(bnr\) structure from \(bnf\) and modulus \(mod\). \(flag\) is an or-ed combination of \(nf\_GEN\) (include generators) and \(nf\_INIT\) (if omitted, do not return a \(bnr\), only the ray class group as an abelian group). In fact, a single value of \(flag\) actually makes sense: \(nf\_GEN | nf\_INIT\) to initialize a proper \(bnr\): removing \(nf\_GEN\) saves very little time, but the corresponding crippled \(bnr\) structure will raise errors in most class field theoretic functions. Possibly also 0 to quickly compute the ray class group structure; \(bnrclassno\) is faster if we only need the order of the ray class group.

Now we have a proper \(bnr\) encoding a \(bnf\) and a modulus, we no longer need the \([bnf, modulus]\) and \([bnf, modulus, subgroup]\) forms, which would internally call \(Buchray\) anyway. Recall that a subgroup \(H\) is given by a matrix in HNF, whose column express generators of \(H\) on the fixed generators of the ray class group that stored in our \(bnr\). You may also code the trivial subgroup by \text{NULL}.

\[
\text{GEN bnrconductor(GEN bnr, GEN H, long flag)}\]

returns 1 is the class field defined by the subgroup \(H\) (of the ray class group mod \(f\) coded in \(bnr\)) has conductor \(f\). Returns 0 otherwise.

\[
\text{GEN bnrisconductor(GEN bnr, GEN H)}\]

returns 1 is the class field defined by the subgroup \(H\) (of the ray class group mod \(f\) coded in \(bnr\)) has conductor \(f\). Returns 0 otherwise.

\[
\text{GEN bnrdisc(GEN bnr, GEN H, long flag)}\]

returns the discriminant and signature of the class field defined by \(bnr\) and \(H\). See the description of the GP function for details. \(flag\) is an or-ed combination of the flags \(rnf\_REL\) (output relative data) and \(rnf\_COND\) (return 0 unless the modulus is the conductor).

\[
\text{GEN bnrsurjection(GEN BNR, GEN bnr)}\]

\(BNR\) and \(bnr\) defined over the same field \(K\), for moduli \(F\) and \(f\) with \(F | f\), returns the matrix of the canonical surjection \(Cl_K(F) \rightarrow Cl_K(f)\) (giving the image of the fixed ray class group generators of \(BNR\) in terms of the ones in \(bnr\)). \(BNR\) must include the ray class group generators.

\[
\text{GEN ABC_to_bnr(GEN A, GEN B, GEN C, GEN \*H, int addgen)}\]

This is a quick conversion function designed to go from the too general (inefficient) \(A, B, C\) form to the preferred \(bnr\), \(H\) form for class fields. Given \(A, B, C\) as explained above (omitted entries coded by \text{NULL}), return the associated \(bnr\), and set \(H\) to the associated subgroup. If \text{addgen} is 1, make sure that if the \(bnr\) needed to be computed, then it contains generators.

12.1.21 Relative equations, Galois conjugates.

\[
\text{GEN rnfequationall(GEN A, GEN B, long \*pk, GEN \*pLPRS)}\]

\(A\) is either an \(nf\) type (corresponding to a number field \(K\)) or an irreducible \(ZX\) defining a number field \(K\). \(B\) is an irreducible polynomial in \(K[X]\). Returns an absolute equation \(C\) (over \(Q\)) for the number field \(K[X]/(B)\). \(C\) is the characteristic polynomial of \(b + ka\) for some roots \(a\) of \(A\) and \(b\) of \(B\), and \(k\) is a small rational integer. Set \(\*pk\) to \(k\).

If \(pLPRS\) is not \text{NULL} set it to \([h_0, h_1], h_i \in Q[X]\), where \(h_0 + h_1 Y\) is the last non-constant polynomial in the pseudo-Euclidean remainder sequence associated to \(A(Y)\) and \(B(X - kY)\), leading
to $C = \text{Res}_Y(A(Y), B(Y-kX))$. In particular $a := -h_0/h_1$ is a root of $A$ in $\mathbb{Q}[X]/(C)$, and $X-ka$ is a root of $B$.

GEN $\text{nf rnfeq}(\text{GEN } A, \text{GEN } B)$ wrapper around $\text{rnfequationall}$ to allow mapping $K \rightarrow L$ (eltup) and converting elements of $L$ between absolute and relative form ($\text{reltoabs, abstorel}$), without computing a full $\text{rnf}$ structure, which is useful if the relative integral basis is not required. In fact, since $A$ may be a $\text{t_POL}$ or an $\text{nf}$, the integral basis of the base field is not needed either. The return value is the same as $\text{rnf get map}$. Shallow function.

GEN $\text{nf rnfeqsimple}(\text{GEN } nf, \text{GEN } relpol)$ as $\text{nf rnfeq}$ except some fields are omitted, so that only the $\text{abstorel}$ operation is supported. Shallow function.

GEN $\text{eltabstorel}(\text{GEN } \text{rnfeq}, \text{GEN } x)$ $\text{rnfeq}$ is as given by $\text{rnf get map}$ (but in this case $\text{nfeltabstorel}$ is more robust), $\text{nf rnfeq}$ or $\text{nf rnfeqsimple}$, return $x$ as an element of $L/K$, i.e. as a $\text{t_POLMOD}$ with $\text{t_POLMOD}$ coefficients. Shallow function.

GEN $\text{eltabstorel lift}(\text{GEN } \text{rnfeq}, \text{GEN } x)$ same as $\text{eltabstorel}$, except that $x$ is returned in partially lifted form, i.e. as a $\text{t_POL}$ with $\text{t_POLMOD}$ coefficients.

GEN $\text{eltreltoabs}(\text{GEN } \text{rnfeq}, \text{GEN } x)$ $\text{rnfeq}$ is as given by $\text{rnf get map}$ (but in this case $\text{nfeltreltoabs}$ is more robust) or $\text{nf rnfeq}$, return $x$ in absolute form.

void $\text{nf_nefzk}(\text{GEN } \text{nf}, \text{GEN } \text{rnfeq}, \text{GEN } *zknf, \text{GEN } *czknf)$ $\text{rnfeq}$ as given by $\text{nf rnfeq}$, $\text{nf}$ a true $\text{nf}$ structure, set $*zknf$ and $*czknf$ to a suitable representation of $\text{nf.zk}$ allowing quick computation of the map $K \rightarrow L$ by the function $\text{nfeltup}$, without computing a full $\text{rnf}$ structure, which is useful if the relative integral basis is not required. The computed values are the same as in $\text{rnf get nfzk}$. Shallow function.

GEN $\text{nfeltup}(\text{GEN } \text{nf}, \text{GEN } x, \text{GEN } \text{zknf}, \text{GEN } \text{czknf})$ $\text{zknf}$ and $\text{czknf}$ are initialized by $\text{nf_nefzk}$ or $\text{rnf get nfzk}$ (but in this case $\text{nfeltup}$ is more robust); $\text{nf}$ is a true $\text{nf}$ structure for $K$, returns $x \in K$ as a (lifted) element of $L$ in absolute form.

GEN $\text{Rg nffix}(\text{const char } f, \text{GEN } T, \text{GEN } c, \text{int } lift)$ given a $\text{ZX} T$ and a “coefficient” $c$ supposedly belonging to $\mathbb{Q}[y]/(T)$, check whether this is the case and return a cleaned up version of $c$. The string $f$ is the calling function name, used to report errors.

This means that $c$ must be one of $\text{t_INT, t_FRAC, t_POL}$ in the variable $y$ with rational coefficients, or $\text{t_POLMOD}$ modulo $T$ which lift to a rational $\text{t_POL}$ as above. The cleanup consists in the following improvements:

- $\text{t_POL}$ coefficients are reduced modulo $T$.
- $\text{t_POL}$ and $\text{t_POLMOD}$ belonging to $\mathbb{Q}$ are converted to rationals, $\text{t_INT}$ or $\text{t_FRAC}$.
- if $\text{lift}$ is non-zero, convert $\text{t_POLMOD}$ to $\text{t_POL}$, and otherwise convert $\text{t_POL}$ to $\text{t_POLMOD}$s modulo $T$.

GEN $\text{RgX_nffix}(\text{const char } f, \text{GEN } T, \text{GEN } P, \text{int } lift)$ check whether $P$ is a polynomials with coefficients in the number field defined by the absolute equation $T(y) = 0$, where $T$ is a $\text{ZX}$ and returns a cleaned up version of $P$. This checks whether $P$ is indeed a $\text{t_POL}$ with variable compatible with coefficients in $\mathbb{Q}[y]/(T)$, i.e.

$$\text{varncmp(varn(P), varn(T)) < 0}$$

and applies $\text{Rg nffix}$ to each coefficient.

228
GEN RgV_nffix(const char *f, GEN T, GEN P, int lift) as RgX_nffix for a vector of coefficients.

GEN polmod_nffix(const char *f, GEN rnf, GEN x, int lift) given a t_POLMOD x supposedly defining an element of rnf, check this and perform Rg_nffix cleanups.

GEN polmod_nffix2(const char *f, GEN T, GEN P, GEN x, int lift) as polmod_nffix, where the relative extension is explicitly defined as $L = \mathbb{Q}[y]/(T(x)/P)$, instead of by an rnf structure.

long numberofconjugates(GEN T, long pinit) returns a quick multiple for the number of $\mathbb{Q}$-automorphism of the (integral, monic) t POL T, from modular factorizations, starting from prime pinit (you can set it to 2). This upper bounds often coincides with the actual number of conjugates. Of course, you should use nfgaloisconj to be sure.

12.1.22 Obsolete routines.

Still provided for backward compatibility, but should not be used in new programs. They will eventually disappear.

GEN zidealstar(GEN nf, GEN x) short for Idealstar(nf, x, nf_GEN)
GEN zidealstarinit(GEN nf, GEN x) short for Idealstar(nf, x, nf_INIT)
GEN zidealstarinitgen(GEN nf, GEN x) short for Idealstar(nf, x, nf_GEN|nf_INIT)
GEN buchimag(GEN D, GEN c1, GEN c2, GEN gCO) short for

\[ \text{Buchquad}(D, \text{gtodouble}(c1), \text{gtodouble}(c2), /*\text{ignore}*/0) \]
GEN buchreal(GEN D, GEN gsens, GEN c1, GEN c2, GEN RELSUP, long prec) short for

\[ \text{Buchquad}(D, \text{gtodouble}(c1), \text{gtodouble}(c2), \text{prec}) \]

The following use a naming scheme which is error-prone and not easily extensible; besides, they compute generators as per nf_GEN and not nf_GENMAT. Don’t use them:

GEN isprincipalforce(GEN bnf, GEN x)
GEN isprincipalgen(GEN bnf, GEN x)
GEN isprincipalgenforce(GEN bnf, GEN x)
GEN isprincipalraygen(GEN bnr, GEN x), use bnrisprincipal.

Variants on polred: use polredbest.

GEN factoredpolred(GEN x, GEN fa)
GEN factoredpolred2(GEN x, GEN fa)
GEN smallpolred(GEN x)
GEN smallpolred2(GEN x), use Polred.
GEN polredabs(GEN x)
GEN polredabs2(GEN x)
GEN polredabsall(GEN x, long flun)
nfmaxord wrappers implementing the old nfbasis interface: use the $[T, listP$ format.
12.2 Galois extensions of $\mathbb{Q}$.

This section describes the data structure output by the function `galoisinit`. This will be called a `gal` structure in the following.

12.2.1 Extracting info from a `gal` structure.

The functions below expect a `gal` structure and are shallow. See the documentation of `galoisinit` for the meaning of the member functions.

- `GEN gal_get_pol(GEN gal)` returns `gal.pol`.
- `GEN gal_get_p(GEN gal)` returns `gal.p`.
- `GEN gal_get_e(GEN gal)` returns the integer `e` such that `gal.mod==gal.p^e`.
- `GEN gal_get_mod(GEN gal)` returns `gal.mod`.
- `GEN gal_get_roots(GEN gal)` returns `gal.roots`.
- `GEN gal_get_group(GEN gal)` returns `gal.group`.
- `GEN gal_get_gen(GEN gal)` returns `gal.gen`.
- `GEN gal_get_orders(GEN gal)` returns `gal.orders`.

Superseded by `bnrdisc`:

- `GEN discrayabs(GEN bnr, GEN subgroup)`
- `GEN discrayabscond(GEN bnr, GEN subgroup)`
- `GEN discrayrel(GEN bnr, GEN subgroup)`
- `GEN discrayrelcond(GEN bnr, GEN subgroup)`

Superseded by `bnrdisclist0`:

- `GEN discrayablist(GEN bnf, GEN listes)`
- `GEN discrayablistarch(GEN bnf, GEN arch, long bound)`
- `GEN discrayablistlong(GEN bnf, long bound)`
12.2.2 Miscellaneous functions.

GEN nfgaloismatrix(GEN nf, GEN s) returns the ZM associated to the automorphism s, seen as a linear operator expressed on the number field integer basis. This allows to use

\[
M = \text{nfgaloismatrix}(nf, s);
sx = \text{ZM\_ZC\_mul}(M, x); /* or RgM\_RgC\_mul(M, x) if x is not integral */
\]

instead of

\[
sx = \text{nfgaloisapply}(nf, s, x);
\]

for an algebraic integer x.

12.3 Quadratic number fields and quadratic forms.

12.3.1 Checks.

void check_quaddisc(GEN x, long *s, long *mod4, const char *f) checks whether the GEN x is a quadratic discriminant (t\_INT, not a square, congruent to 0,1 modulo 4), and raise an exception otherwise. Set *s to the sign of x and *mod4 to x modulo 4 (0 or 1).

void check_quaddisc_real(GEN x, long *mod4, const char *f) as check_quaddisc; check that signe(x) is positive.

void check_quaddisc_imag(GEN x, long *mod4, const char *f) as check_quaddisc; check that signe(x) is negative.

12.3.2 t\_QFI, t\_QFR.

GEN qfi(GEN x, GEN y, GEN z) creates the t\_QFI (x,y,z).

GEN qfr(GEN x, GEN y, GEN z, GEN d) creates the t\_QFR (x,y,z) with distance component d.

GEN qfr\_1(GEN q) given a t\_QFR q, return the unit form q\^0.

GEN qfi\_1(GEN q) given a t\_QFI q, return the unit form q\^0.

12.3.2.1 Composition.

GEN qficomp(GEN x, GEN y) compose the two t\_QFI x and y, then reduce the result. This is the same as gmul(x,y).

GEN qfrcomp(GEN x, GEN y) compose the two t\_QFR x and y, then reduce the result. This is the same as gmul(x,y).

GEN qfisqr(GEN x) as qficomp(x,y).

GEN qfrsqr(GEN x) as qfrcomp(x,y).

Same as above, without reducing the result:

GEN qficompraw(GEN x, GEN y)

GEN qfrcompraw(GEN x, GEN y)

GEN qfisqrraw(GEN x)

GEN qfrsqrraw(GEN x)

GEN qfbcompraw(GEN x, GEN y) compose two t\_QFIs or two t\_QFRs, without reduce the result.
12.3.2.2 Powering.

GEN powgi(GEN x, GEN n) computes $x^n$ (will work for many more types than t_QFI and t_QFR, of course). Reduce the result.

GEN qfrpow(GEN x, GEN n) computes $x^n$ for a t_QFR $x$, reducing along the way. If the distance component is initially 0, leave it alone; otherwise update it.

GEN qfbpowraw(GEN x, long n) compute $x^n$ (pure composition, no reduction), for a t_QFI or t_QFR $x$.

GEN qfipowraw(GEN x, long n) as qfbpowraw, for a t_QFI $x$.

GEN qfrpowraw(GEN x, long n) as qfbpowraw, for a t_QFR $x$.

12.3.2.3 Solve, Cornacchia.

The following functions underly qfbsolve; $p$ denotes a prime number.

GEN qfisolvep(GEN Q, GEN p) solves $Q(x, y) = p$ over the integers, for a t_QFI $Q$. Return gen_0 if there are no solutions.

GEN qfrsolvep(GEN Q, GEN p) solves $Q(x, y) = p$ over the integers, for a t_QFR $Q$. Return gen_0 if there are no solutions.

long cornacchia(GEN d, GEN p, GEN *px, GEN *py) solves $x^2 + dy^2 = p$ over the integers, where $d > 0$. Return 1 if there is a solution (and store it in *x and *y), 0 otherwise.

long cornacchia2(GEN d, GEN p, GEN *px, GEN *py) as cornacchia, for the equation $x^2 + dy^2 = 4p$.

12.3.2.4 Prime forms.

GEN primeform_u(GEN x, ulong p) t_QFI whose first coefficient is the prime $p$.

GEN primeform(GEN x, GEN p, long prec)

12.3.3 Efficient real quadratic forms. Unfortunately, t_QFRs are very inefficient, and are only provided for backward compatibility.

- they do not contain needed quantities, which are thus constantly recomputed (the discriminant $D$, $\sqrt{D}$ and its integer part),
- the distance component is stored in logarithmic form, which involves computing one extra logarithm per operation. It is much more efficient to store its exponential, computed from ordinary multiplications and divisions (taking exponent overflow into account), and compute its logarithm at the very end.

Internally, we have two representations for real quadratic forms:

- qfr3, a container $[a, b, c]$ with at least 3 entries: the three coefficients; the idea is to ignore the distance component.
- qfr5, a container with at least 5 entries $[a, b, c, e, d]$: the three coefficients a t_REAL $d$ and a t_INT $e$ coding the distance component $2^{Ne}d$, in exponential form, for some large fixed $N$.

It is a feature that qfr3 and qfr5 have no specified length or type. It implies that a qfr5 or t_QFR will do whenever a qfr3 is expected. Routines using these objects all require a global context, provided by a struct qfr_data *:
struct qfr_data {
    GEN D; /* discriminant, t_INT */
    GEN sqrtD; /* sqrt(D), t_REAL */
    GEN isqrtD; /* floor(sqrt(D)), t_INT */
};

void qfr_data_init(GEN D, long prec, struct qfr_data *S) given a discriminant \( D > 0 \), initialize \( S \) for computations at precision \( \text{prec} \) (\( \sqrt{D} \) is computed to that initial accuracy).

All functions below are shallow, and not stack clean.

GEN qfr3_comp(GEN x, GEN y, struct qfr_data *S) compose two qfr3, reducing the result.

GEN qfr3_pow(GEN x, GEN n, struct qfr_data *S) compute \( x^n \), reducing along the way.

GEN qfr3_red(GEN x, struct qfr_data *S) reduce \( x \).

GEN qfr3_rho(GEN x, struct qfr_data *S) perform one reduction step; \( \text{qfr3}_\text{red} \) just performs reduction steps until we hit a reduced form.

GEN qfr3_to_qfr(GEN x, GEN d) recover an ordinary t_QFR from the qfr3 \( x \), adding distance component \( d \).

Before we explain \( \text{qfr5} \), recall that it corresponds to an ideal, that reduction corresponds to multiplying by a principal ideal, and that the distance component is a clever way to keep track of these principal ideals. More precisely, reduction consists in a number of reduction steps, going from the form \((a, b, c) \) to \( \rho(a, b, c) = (c, -b \mod 2c, *) \); the distance component is multiplied by (a floating point approximation to) \( (b + \sqrt{D})/(b - \sqrt{D}) \).

GEN qfr5_comp(GEN x, GEN y, struct qfr_data *S) compose two qfr5, reducing the result, and updating the distance component.

GEN qfr5_pow(GEN x, GEN n, struct qfr_data *S) compute \( x^n \), reducing along the way.

GEN qfr5_red(GEN x, struct qfr_data *S) reduce \( x \).

GEN qfr5_rho(GEN x, struct qfr_data *S) perform one reduction step.

GEN qfr5_dist(GEN e, GEN d, long prec) decode the distance component from exponential (qfr5-specific) to logarithmic form (as in a t_QFR).

GEN qfr_to_qfr5(GEN x, long prec) convert a t_QFR to a qfr5 with initial trivial distance component (= 1).

GEN qfr5_to_qfr(GEN x, GEN d), assume \( x \) is a qfr5 and \( d \) was the original distance component of some t_QFR that we converted using \( \text{qfr5}_\text{to}_\text{qfr} \) to perform efficiently a number of operations. Convert \( x \) to a t_QFR with the correct (logarithmic) distance component.
12.4 Linear algebra over \( \mathbb{Z} \).

12.4.1 Hermite and Smith Normal Forms.

\[ \text{GEN } \text{ZM}_hnf(\text{GEN } x) \text{ returns the upper triangular Hermite Normal Form of the ZM } x \text{ (removing 0 columns), using the ZM}_hnfall algorithm. If you want the true HNF, use ZM}_hnfall(x, \text{NULL}, 0). \]

\[ \text{GEN } \text{ZM}_hnfmod(\text{GEN } x, \text{GEN } d) \text{ returns the HNF of the ZM } x \text{ (removing 0 columns), assuming the t_INT } d \text{ is a multiple of the determinant of } x. \text{ This is usually faster than ZM}_hnf (and uses less memory) if the dimension is large, > 50 say. \]

\[ \text{GEN } \text{ZM}_hnfmodid(\text{GEN } x, \text{GEN } d) \text{ returns the HNF of the matrix } (x \mid d \text{Id}) \text{ (removing 0 columns), for a ZM } x \text{ and a t_INT } d. \]

\[ \text{GEN } \text{ZM}_hnfmodall(\text{GEN } x, \text{GEN } d, \text{long flag}) \text{ low-level function underlying the ZM}_hnfmod variants. If flag is 0, calls ZM}_hnfmod(x,d); flag is an or-ed combination of: }\]

- \( \text{hnf}_\text{MODID} \) call ZM}_hnfmodid instead of ZM}_hnfmod,

- \( \text{hnf}_\text{PART} \) return as soon as we obtain an upper triangular matrix, saving time. The pivots are non-negative and give the diagonal of the true HNF, but the entries to the right of the pivots need not be reduced, i.e. they may be large or negative.

- \( \text{hnf}_\text{CENTER} \) returns the centered HNF, where the entries to the right of a pivot \( p \) are centered residues in \([-p/2,p/2]\), hence smallest possible in absolute value, but possibly negative.

\[ \text{GEN } \text{ZM}_hnfall(\text{GEN } x, \text{GEN } *U, \text{int remove}) \text{ returns the upper triangular HNF } H \text{ of the ZM } x; \text{ if } U \text{ is not NULL, set if to the matrix } U \text{ such that } xU = H. \text{ If remove = 0, } H \text{ is the true HNF, including 0 columns; if remove = 1, delete the 0 columns from } H \text{ but do not update } U \text{ accordingly (so that the integer kernel may still be recovered): we no longer have } xU = H; \text{ if remove = 2, remove 0 columns from } H \text{ and update } U \text{ so that } xU = H. \text{ The matrix } U \text{ is square and invertible unless remove = 2.} \]

This routine uses a naive algorithm which is potentially exponential in the dimension (due to coefficient explosion) but is fast in practice, although it may require lots of memory. The base change matrix \( U \) may be very large, when the kernel is large.

\[ \text{GEN } \text{ZM}_hnfperm(\text{GEN } A, \text{GEN } *ptU, \text{GEN } *ptperm) \text{ returns the hnf } H = PAU \text{ of the matrix } PA, \text{ where } P \text{ is a suitable permutation matrix, and } U \in \text{Gl}_n(\mathbb{Z}). \text{ } P \text{ is chosen so as to (heuristically) minimize the size of } U; \text{ in this respect it is less efficient than ZM}_hnfull but usually faster. Set *ptU to } U \text{ and *ptperm to a t_VECSMALL representing the row permutation associated to } P = (\delta_{i,\text{perm}[i]}). \text{ If ptU is set to NULL, } U \text{ is not computed, saving some time; although useless, setting ptperm to NULL is also allowed.} \]

\[ \text{GEN } \text{ZM}_hnfull1(\text{GEN } x, \text{GEN } *U, \text{int remove}) \text{ returns the HNF } H \text{ of the ZM } x; \text{ if } U \text{ is not NULL, set if to the matrix } U \text{ such that } xU = H. \text{ The meaning of remove is the same as in ZM}_hnfall. \]

This routine uses the LLL variant of Havas, Majewski and Mathews, which is polynomial time, but rather slow in practice because it uses an exact LLL over the integers instead of a floating point variant; it uses polynomial space but lots of memory is needed for large dimensions, say larger than 300. On the other hand, the base change matrix \( U \) is essentially optimally small with respect to the \( L_2 \) norm.

\[ \text{GEN } \text{ZM}_hnfcenter(\text{GEN } M). \text{ Given a ZM in HNF } M, \text{ update it in place so that non-diagonal entries belong to a system of centered residues. Not suitable for gerepile.} \]
Some direct applications: the following routines apply to upper triangular integral matrices; in practice, these come from HNF algorithms.

\texttt{GEN hnf\_divscale(GEN A, GEN B, GEN t)} A an upper triangular \texttt{ZM}, \texttt{B} a \texttt{ZM}, \texttt{t} an integer, such that \texttt{C} := \texttt{tA}^{-1}\texttt{B} is integral. Return \texttt{C}.

\texttt{GEN hnf\_solve(GEN A, GEN B)} A a \texttt{ZM} in upper HNF (not necessarily square), \texttt{B} a \texttt{ZM} or \texttt{ZC}. Return \texttt{A}^{-1}\texttt{B} if it is integral, and \texttt{NULL} if it is not.

\texttt{GEN hnf\_invgimage(GEN A, GEN b)} A a \texttt{ZM} in upper HNF (not necessarily square), \texttt{b} a \texttt{ZC}. Return \texttt{A}^{-1}\texttt{B} if it is integral, and \texttt{NULL} if it is not.

\texttt{int hnfdivide(GEN A, GEN B)} \texttt{A} and \texttt{B} are two upper triangular \texttt{ZM}. Return 1 if \texttt{A}^{-1}\texttt{B} is integral, and 0 otherwise.

**Smith Normal Form.**

\texttt{GEN ZM\_snf(GEN x)} returns the Smith Normal Form (vector of elementary divisors) of the \texttt{ZM} \texttt{x}.

\texttt{GEN ZM\_snfall(GEN x, GEN *U, GEN *V)} returns \texttt{ZM\_smith(x)} and sets \texttt{U} and \texttt{V} to unimodular matrices such that \texttt{UxV} = \texttt{D} (diagonal matrix of elementary divisors). Either (or both) \texttt{U} or \texttt{V} may be \texttt{NULL} in which case the corresponding matrix is not computed.

\texttt{GEN ZM\_snfall\_i(GEN x, GEN *U, GEN *V, int returnvec)} same as \texttt{ZM\_snfall}, except that, depending on the value of \texttt{returnvec}, we either return a diagonal matrix (as in \texttt{ZM\_snfall}, \texttt{returnvec} is 0) or a vector of elementary divisors (as in \texttt{ZM\_snf}, \texttt{returnvec} is 1).

\texttt{void ZM\_snfclean(GEN d, GEN U, GEN V)} assuming \texttt{d}, \texttt{U}, \texttt{V} come from \texttt{d} = \texttt{ZM\_snfall(x, &U, &V)}, where \texttt{U} or \texttt{V} may be \texttt{NULL}, cleans up the output in place. This means that elementary divisors equal to 1 are deleted and \texttt{U}, \texttt{V} are updated. The output is not suitable for \texttt{gerepileupto}.

\texttt{GEN ZM\_snf\_group(GEN H, GEN *U, GEN *Uninv)} this function computes data to go back and forth between an abelian group (of finite type) given by generators and relations, and its canonical SNF form. Given an abstract abelian group with generators \(g = (g_1, \ldots, g_n)\) and a vector \(X = (x_i) \in \mathbb{Z}^n\), we write \(gX\) for the group element \(\sum_i x_i g_i\); analogously if \(M\) is an \(n \times r\) integer matrix \(gM\) is a vector containing \(r\) group elements. The group neutral element is 0; by abuse of notation, we still write 0 for a vector of group elements all equal to the neutral element. The input is a full relation matrix \(H\) among the generators, i.e. a \texttt{ZM} (not necessarily square) such that \(gX = 0\) for some \(X \in \mathbb{Z}^n\) if and only if \(X\) is in the integer image of \(H\), so that the abelian group is isomorphic to \(\mathbb{Z}^n/\text{Im} H\). The routine assumes that \(H\) is in HNF; replace it by its HNF if it is not the case. (Of course this defines the same group.)

Let \(G\) a minimal system of generators in SNF for our abstract group: if the \(d_i\) are the elementary divisors \((\ldots | d_2 | d_1)\), each \(G_i\) has either infinite order \((d_i = 0)\) or order \(d_i > 1\). Let \(D\) the matrix with diagonal \((d_i)\), then

\[
GD = 0, \quad G = gU_{\text{inv}}, \quad g = GU,
\]

for some integer matrices \(U\) and \(U_{\text{inv}}\). Note that these are not even square in general; even if square, there is no guarantee that these are unimodular: they are chosen to have minimal entries given the known relations in the group and only satisfy \(D | (U_{\text{inv}} I - \text{Id})\) and \(H | (U_{\text{inv}} U - \text{Id})\).

The function returns the vector of elementary divisors \((d_i)\); if \(U\) is not \texttt{NULL}, it is set to \(U\); if \(U_{\text{inv}}\) is not \texttt{NULL} it is set to \(U_{\text{inv}}\). The function is not memory clean.

The following 3 routines underly the various \texttt{matrixqz} variants. In all case the \(m \times n\) \texttt{t\_MAT} \(x\) is assumed to have rational \(\text{t\_INT}\) and \(\text{t\_FRAC}\) coefficients.
GEN `QM_ImQ_hnf(GEN x)` returns an HNF basis for \( \text{Im}_Q x \cap \mathbb{Z}^n \).

GEN `QM_ImZ_hnf(GEN x)` returns an HNF basis for \( \text{Im}_Z x \cap \mathbb{Z}^n \).

GEN `QM_minors_coprime(GEN x, GEN D)`, assumes \( m \geq n \), and returns a matrix in \( M_{m,n}(\mathbb{Z}) \) with the same \( \mathbb{Q} \)-image as \( x \), such that the GCD of all \( n \times n \) minors is coprime to \( D \); if \( D \) is NULL, we want the GCD to be 1.

The following routines are simple wrappers around the above ones and are normally useless in library mode:

GEN `hnf(GEN x)` checks whether \( x \) is a \( \mathbb{Z}M \), then calls \( \text{ZM}_\text{hnf} \). Normally useless in library mode.

GEN `hnfmod(GEN x, GEN d)` checks whether \( x \) is a \( \mathbb{Z}M \), then calls \( \text{ZM}_\text{hnfmod} \). Normally useless in library mode.

GEN `hnfmodid(GEN x, GEN d)` checks whether \( x \) is a \( \mathbb{Z}M \), then calls \( \text{ZM}_\text{hnfmodid} \). Normally useless in library mode.

GEN `hnfall(GEN x)` calls \( \text{ZM}_\text{hnfall}(x, &U, 1) \) and returns \([H, U]\). Normally useless in library mode.

GEN `hnflll(GEN x)` calls \( \text{ZM}_\text{hnflll}(x, &U, 1) \) and returns \([H, U]\). Normally useless in library mode.

GEN `hnfperm(GEN x)` calls \( \text{ZM}_\text{hnfperm}(x, &U, &P) \) and returns \([H, U, P]\). Normally useless in library mode.

GEN `smith(GEN x)` checks whether \( x \) is a \( \mathbb{Z}M \), then calls \( \text{ZM}_\text{smith} \). Normally useless in library mode.

GEN `smithall(GEN x)` checks whether \( x \) is a \( \mathbb{Z}M \), then calls \( \text{ZM}_\text{smithall}(x, &U, &V) \) and returns \([U, V, D]\). Normally useless in library mode.

Some related functions over \( K[X] \), \( K \) a field:

GEN `gsmith(GEN A)` the input matrix must be square, returns the elementary divisors.

GEN `gsmithall(GEN A)` the input matrix must be square, returns the \([U, V, D]\), \( D \) diagonal, such that \( UAV = D \).

GEN `RgM_hnfall(GEN A, GEN *pB, long remove)` analogous to \( \text{ZM}_\text{hnfall} \).

GEN `smithclean(GEN z)` cleanup the output of \( \text{smithall} \) or \( \text{gsmithall} \) (delete elementary divisors equal to 1, updating base change matrices).

### 12.4.2 The LLL algorithm.

The basic GP functions and their immediate variants are normally not very useful in library mode. We briefly list them here for completeness, see the documentation of \( \text{qflll} \) and \( \text{qflllgram} \) for details:

- GEN `qflll0(GEN x, long flag)`
- GEN `lll(GEN x) flag = 0`
- GEN `lllint(GEN x) flag = 1`
- GEN `lllkerim(GEN x) flag = 4`

236
GEN lllkerimgen(GEN x) flag= 5
GEN lllgen(GEN x) flag= 8

- GEN qflllgram0(GEN x, long flag)

GEN lllgram(GEN x) flag= 0
GEN lllgramint(GEN x) flag= 1
GEN lllgramkerim(GEN x) flag= 4
GEN lllgramkerimgen(GEN x) flag= 5
GEN lllgramgen(GEN x) flag= 8

The basic workhorse underlying all integral and floating point LLLs is

GEN ZM_lll(GEN x, double D, long flag), where

- $x$ is a ZM;
- $D \in [1/4, 1]$ is the Lovász constant
determining the frequency of swaps during the algorithm: a larger values means better guarantees
for the basis (in principle smaller basis vectors) but longer running times (suggested value: $D = 0.99$).

**Important.** This function does not collect garbage and its output is not suitable for either
gerepile or gerepileupto. We expect the caller to do something simple with the output (e.g.
matrix multiplication), then collect garbage immediately.

flag is an or-ed combination of the following flags:

- LLL_GRAM. If set, the input matrix $x$ is the Gram matrix $\langle v, v \rangle$ of some lattice vectors $v$.
- LLL_INPLACE. If unset, we return the base change matrix $U$, otherwise the transformed
matrix $xU$ or $\langle Ux, U \rangle$ (LLL_GRAM). Implies LLL_IM (see below).
- LLL_KEEP_FIRST. The first vector in the output basis is the same one as was originally input.
Provided this is a shortest non-zero vector of the lattice, the output basis is still LLL-reduced. This
is used to reduce maximal orders of number fields with respect to the $T_2$ quadratic form, to ensure
that the first vector in the output basis corresponds to 1 (which is a shortest vector).

The last three flags are mutually exclusive, either 0 or a single one must be set:

- LLL_KER If set, only return a kernel basis $K$ (not LLL-reduced).
- LLL_IM If set, only return an LLL-reduced lattice basis $T$. (This is implied by LLL_INPLACE).
- LLL_ALL If set, returns a 2-component vector $[K, T]$ corresponding to both kernel and image.

GEN lllfp(GEN x, double D, long flag) is a variant for matrices with inexact entries: $x$ is a matrix with real coefficients (types t_INT, t_FRAC and t_REAL), $D$ and flag are as in ZM_lll. The matrix is rescaled, rounded to nearest integers, then fed to ZM_lll. The flag LLL_INPLACE is still accepted but less useful (it returns an LLL-reduced basis associated to rounded input, instead of
an exact base change matrix).

GEN ZM_lll_norms(GEN x, double D, long flag, GEN *ptB) slightly more general version of ZM_lll, setting *ptB to a vector containing the squared norms of the Gram-Schmidt vectors ($b_i^*$)
associated to the output basis ($b_i$), $b_i^* = b_i + \sum_{j<i} \mu_{i,j} b_j^*$.

GEN lllintpartial_inplace(GEN x) given a ZM $x$ of maximal rank, returns a partially reduced
basis ($b_i$) for the space spanned by the columns of $x$: $|b_i \pm b_j| \geq |b_i|$ for any two distinct basis vectors $b_i$, $b_j$. This is faster than the LLL algorithm, but produces much larger bases.

237
GEN lllintpartial(GEN x) as lllintpartial_inplace, but returns the base change matrix \( U \) from the canonical basis to the \( b_i \), i.e. \( xU \) is the output of lllintpartial_inplace.

### 12.4.3 Reduction modulo matrices.

GEN ZC_hnfremdiv(GEN x, GEN y, GEN *Q) assuming \( y \) is an invertible ZM in HNF and \( x \) is a ZC, returns the ZC \( R \) equal to \( x \mod y \) (whose \( i \)-th entry belongs to \([-y_i/2, y_i/2]\). Stack clean unless \( x \) is already reduced (in which case, returns \( x \) itself, not a copy). If \( Q \) is not NULL, set it to the ZC such that \( x = yQ + R \).

GEN ZM_hnfdivrem(GEN x, GEN y, GEN *Q) reduce each column of the ZM \( x \) using ZC_hnfremdiv. If \( Q \) is not NULL, set it to the ZM such that \( x = yQ + R \).

GEN ZC_hnfrem(GEN x, GEN y) alias for ZC_hnfremdiv(x,y,NULL).

GEN ZM_hnfrem(GEN x, GEN y) alias for ZM_hnfremdiv(x,y,NULL).

GEN ZC_reducemodmatrix(GEN v, GEN y) Let \( y \) be a ZM, not necessarily square, which is assumed to be LLL-reduced (otherwise, very poor reduction is expected). Size-reduces the ZC \( v \) modulo the Z-module \( Y \) spanned by \( y \): if the columns of \( y \) are denoted by \((y_1, \ldots, y_{n-1})\), we return \( y_n \equiv v \mod Y \), such that the Gram-Schmidt coefficients \( \mu_{n,j} \) are less than 1/2 in absolute value for all \( j < n \). In short, \( y_n \) is almost orthogonal to \( Y \).

GEN ZM_reducemodmatrix(GEN v, GEN y) Let \( y \) be as in ZC_reducemodmatrix, and \( v \) be a ZM. This returns a matrix \( v \) which is congruent to \( v \) modulo the Z-module spanned by \( y \), whose columns are size-reduced. This is faster than repeatedly calling ZC_reducemodmatrix on the columns since most of the Gram-Schmidt coefficients can be reused.

GEN ZC_reducemodlll(GEN v, GEN y) Let \( y \) be an arbitrary ZM, LLL-reduce it then call ZC_reducemodmatrix.

GEN ZM_reducemodlll(GEN v, GEN y) Let \( y \) be an arbitrary ZM, LLL-reduce it then call ZM_reducemodmatrix.

Besides the above functions, which were specific to integral input, we also have:

GEN reducemodinvertible(GEN x, GEN y) \( y \) is an invertible matrix and \( x \) a t_COL or t_MAT of compatible dimension. Returns \( x - y \lfloor y^{-1} x \rfloor \), which has small entries and differs from \( x \) by an integral linear combination of the columns of \( y \). Suitable for gerepileupto, but does not collect garbage.

GEN closemodinvertible(GEN x, GEN y) returns \( x - \text{reducemodinvertible}(x,y) \), i.e. an integral linear combination of the columns of \( y \), which is close to \( x \).

GEN reducemodlll(GEN x, GEN y) LLL-reduce the non-singular ZM \( y \) and call reducemodinvertible to find a small representative of \( x \mod y \mathbb{Z}^n \). Suitable for gerepileupto, but does not collect garbage.

### 12.4.4 Miscellaneous.

GEN RM_round_maxrank(GEN G) given a matrix \( G \) with real floating point entries and independent columns, let \( G_e \) be the rescaled matrix \( 2^e G \) rounded to nearest integers, for \( e \geq 0 \). Finds a small \( e \) such that the rank of \( G_e \) is equal to the rank of \( G \) (its number of columns) and return \( G_e \). This is useful as a preconditioning step to speed up LLL reductions, see nf_get_Gtwist. Suitable for gerepileupto, but does not collect garbage.
13.1 Elliptic curves.

Elliptic curves are represented in the Weierstrass model

\[(E) : y^2z + a_1xyz + a_3yz = x^3 + a_2x^2z + a_4xz^2 + a_6z^3,\]

by the 5-tuple \([a_1, a_2, a_3, a_4, a_6]\). Points in the projective plane are represented as follows: the point at infinity \((0 : 1 : 0)\) is coded as \([0]\), a finite point \((x : y : 1)\) outside the projective line at infinity \(z = 0\) is coded as \([x, y]\). Note that other points at infinity than \((0 : 1 : 0)\) cannot be represented; this is harmless, since they do not belong to any of the elliptic curves \(E\) above.

Points on the curve are just projective points as described above, they are not tied to a curve in any way: the same point may be used in conjunction with different curves, provided it satisfies their equations (if it does not, the result is usually undefined). In particular, the point at infinity belongs to all elliptic curves.

As with \texttt{factor} for polynomial factorization, the 5-tuple \([a_1, a_2, a_3, a_4, a_6]\) implicitly defines a base ring over which the curve is defined. Point coordinates must be operation-compatible with this base ring (\texttt{gadd, gmul, gdiv} involving them should not give errors).

13.1.1 Types of elliptic curves.

We call a 5-tuple as above an \texttt{ell5}; most functions require an \texttt{ell} structure, as returned by \texttt{ellinit}, which contains additional data (usually dynamically computed as needed), depending on the base field.

\[
\text{GEN} \ ell\text{init}(\text{GEN } E, \text{GEN } D, \text{long } prec)\]
returns an \texttt{ell} structure, associated to the elliptic curve \(E\) : either an \texttt{ell5}, a pair \([a_4, a_6]\) or a \texttt{t_STR} in Cremona’s notation, e.g. "11a1". The optional \(D\) (\texttt{NULL} to omit) describes the domain over which the curve is defined.

13.1.2 Type checking.

\[
\begin{align*}
\text{void checkell}(\text{GEN } e) & \text{ raise an error unless } e \text{ is a } \texttt{ell}. \\
\text{void checkell5}(\text{GEN } e) & \text{ raise an error unless } e \text{ is an } \texttt{ell} \text{ or an } \texttt{ell5}. \\
\text{void checkellpt}(\text{GEN } z) & \text{ raise an error unless } z \text{ is a point (either finite or at infinity).} \\
\text{long ell}\_\text{get}\_\text{type}(\text{GEN } e) & \text{ returns the domain type over which the curve is defined, one of} \\
& \text{t}\_\texttt{ELL}\_Q \text{ the field of rational numbers;} \\
& \text{t}\_\texttt{ELL}\_\text{Qp} \text{ the field of } p\text{-adic numbers, for some prime } p; \\
& \text{t}\_\texttt{ELL}\_Fp \text{ a prime finite field, base field elements are represented as } \text{Fp} (\text{t}\_\texttt{INT} \text{ reduced modulo } p); \\
\end{align*}
\]
t_ELL_Fq a non-prime finite field (a prime finite field can also be represented by this subtype, but this is inefficient), base field elements are represented as t_FFELT;

t_ELL_Rg none of the above.

void checkell_Fq(GEN e) checks whether e is an ell, defined over a finite field (either prime or non-prime), raises pari_err_TYPE otherwise.

void checkell_Q(GEN e) checks whether e is an ell, defined over Q, raises pari_err_TYPE otherwise.

void checkell_Qp(GEN e) checks whether e is an ell, defined over some Q_p, raises pari_err_TYPE otherwise.

13.1.3 Extracting info from an ell structure.

These functions expect an ell argument. If the required data is not part of the structure, it is computed then inserted, and the new value is returned.

13.1.3.1 All domains.

GEN ell_get_a1(GEN e)
GEN ell_get_a2(GEN e)
GEN ell_get_a3(GEN e)
GEN ell_get_a4(GEN e)
GEN ell_get_a6(GEN e)
GEN ell_get_b2(GEN e)
GEN ell_get_b4(GEN e)
GEN ell_get_b6(GEN e)
GEN ell_get_b8(GEN e)
GEN ell_get_c4(GEN e)
GEN ell_get_c6(GEN e)
GEN ell_get_disc(GEN e)
GEN ell_get_j(GEN e)

13.1.3.2 Curves over Q.

GEN ellQ_get_N(GEN e) returns the curve conductor

void ellQ_get_Nfa(GEN e, GEN *N, GEN *faN) sets N to the conductor and faN to its factorization

long ellrootno_global(GEN e) returns \([c, [c_{p_1}, \ldots, c_{p_k}]],\) where the t_INT c ∈ \([-1, 1]\) is the global root number, and the \(c_{p_i}\) are the local root numbers at all prime divisors of the conductor, ordered as in faN above.

GEN elldatagenerators(GEN E) returns generators for \(E(Q)\) extracted from Cremona’s table.

GEN ellanal_globalred(GEN e, GEN *v) takes an ell over Q and returns a global minimal model \(E\) (in ellinit form, over Q) for e suitable for analytic computations related to the curve L series:
it contains `ellglobalred` data, as well as global and local root numbers. If `v` is not NULL, set `*v` to the needed change of variable: NULL if `e` was already the standard minimal model, such that \( E = \text{ellchangecurve}(e, v) \) otherwise. Compared to the direct use of `ellchangecurve` followed by `ellrootno`, this function avoids converting unneeded dynamic data and avoids potential memory leaks (the changed curve would have had to be deleted using `obj_free`). The original curve `e` is updated as well with the same information.

13.1.3.3 Curves over \( \mathbb{Q}_p \).

```c
GEN ellQp_get_p(GEN E) returns \( p \)
long ellQp_get_prec(GEN E) returns the default \( p \)-adic accuracy to which we must compute approximate results associated to \( E \).
GEN ellQp_get_zero(GEN x) returns \( O(p^n) \), where \( n \) is the default \( p \)-adic accuracy as above.
```

The following functions are only defined when \( E \) has multiplicative reduction (Tate curves):

```c
GEN ellQp_Tate_uniformization(GEN E, long prec) returns a `t_VEC` containing \( u^2, u, q, [a, b] \), at \( p \)-adic precision `prec`.
GEN ellQp_u(GEN E, long prec) returns \( u \).
GEN ellQp_u2(GEN E, long prec) returns \( u^2 \).
GEN ellQp_q(GEN E, long prec) returns the Tate period \( q \).
GEN ellQp_ab(GEN E, long prec) returns \( [a, b] \).
GEN ellQp_root(GEN E, long prec) returns \( e_1 \).
```

13.1.3.4 Curves over a finite field \( \mathbb{F}_q \).

```c
GEN ellff_get_p(GEN E) returns the characteristic
GEN ellff_get_field(GEN E) returns \( \mathbb{F}_q \) if \( E \) is a prime field, and a `t_FFELT` belonging to \( \mathbb{F}_q \) otherwise.
GEN ellff_get_card(GEN E) returns \( \#E(\mathbb{F}_q) \)
GEN ellff_get_gens(GEN E) returns a minimal set of generators for \( E(\mathbb{F}_q) \).
GEN ellff_get_group(GEN E) returns `ellgroup(E)`.
GEN ellff_get_o(GEN E) returns \([d, \text{factord}]\), where \( d \) is the exponent of \( E(\mathbb{F}_q) \).
GEN ellff_get_a4a6(GEN E) returns a canonical "short model" for \( E \), and the corresponding change of variable \([u, r, s, t]\). For \( p \neq 2, 3 \), this is \([A_4, A_6, [u, r, s, t]]\), corresponding to \( y^2 = x^3 + A_4 x + A_6 \), where \( A_4 = -27c_4, A_6 = -54c_6, [u, r, s, t] = [6, 3b_2, 3a_1, 108a_3] \).

- If \( p = 3 \) and the curve is ordinary \( (b_2 \neq 0) \), this is \([-b_2, A_6, [1, v, -a_1, -a_3]]\), corresponding to
  \[
y^2 = x^3 + b_2 x + A_6,
\]
where \( v = b_4/b_2, A_6 = b_6 - v(b_4 + v^2) \).

- If \( p = 3 \) and the curve is supersingular \( (b_2 = 0) \), this is \([-b_4, b_6, [1, 0, -a_1, -a_3]]\), corresponding to
  \[
y^2 = x^3 + 2b_4 x + b_6.
\]
• If \( p = 2 \) and the curve is ordinary \((a_1 \neq 0)\), return \([A_2, A_6, [a_1^{-1}, d a_1^{-2}, 0, (a_4 + d^2) a_1^{-1}]]\),
corresponding to
\[
y^2 + xy = x^3 + A_2 x^2 + A_6,
\]
where \( d = a_3/a_1, a_1^2 A_2 = (a_2 + d) \) and
\[
a_1^2 A_6 = d^3 + a_2 d^2 + a_4 d + a_6 + (a_4^2 + d^4) a_1^{-2}.
\]

• If \( p = 2 \) and the curve is supersingular \((a_1 = 0, a_3 \neq 0)\), return \([[a_3, A_4, 1/a_3], A_6, [1, a_2, 0, 0]]\),
corresponding to
\[
y^2 + 3a_3 y = x^3 + A_4 x + A_6,
\]
where \( A_4 = a_2^2 + a_4, A_6 = a_2 a_4 + a_6 \). The value \( 1/a_3 \) is included in the vector since it is frequently
needed in computations.

13.1.3.5 Curves over \( \mathbb{C} \). (This includes curves over \( \mathbb{Q} \! \!)\)
\[
\text{long ellR_get_prec(GEN E)} \quad \text{returns the default accuracy to which we must compute approximate results associated to} \ E.
\]
\[
\text{GEN ellR_ab(GEN E, long prec)} \quad \text{returns} \ [a, b]
\]
\[
\text{GEN ellR_omega(GEN x, long prec)} \quad \text{returns periods} \ [\omega_1, \omega_2].
\]
\[
\text{GEN ellR_eta(GEN E, long prec)} \quad \text{returns quasi-periods} \ [\eta_1, \eta_2].
\]
\[
\text{GEN ellR_roots(GEN E, long prec)} \quad \text{returns} \ [e_1, e_2, e_3]. \text{ If} \ E \text{ is defined over} \ \mathbb{R}, \text{ then} \ e_1 \text{ is real. If furthermore} \ \text{disc} E > 0, \text{ then} \ e_1 > e_2 > e_3.
\]
\[
\text{long ellR_get_sign(GEN E)} \quad \text{if} \ E \text{ is defined over} \ \mathbb{R} \text{ returns the signe of its discriminant, otherwise return} 0.
\]

13.1.4 Points.
\[
\text{int ell_is_inf(GEN z)} \quad \text{tests whether the point} \ z \text{ is the point at infinity.}
\]
\[
\text{GEN ellinf()} \quad \text{returns the point at infinity} \ [0].
\]

13.1.5 Change of variables. \text{GEN ellchangeinvert(GEN w)} \text{ given a change of variables} \ w = [u, r, s, t], \text{ returns the inverse change of variables} \ w', \text{ such that if} \ E' = \text{ellchangecurve}(E, w), \text{ then} \ E = \text{ellchangecurve}(E, w').
\]

13.1.6 Functions to handle elliptic curves over finite fields.

13.1.6.1 Tolerant routines.
\[
\text{GEN ellap(GEN E, GEN p)} \quad \text{given a prime number} \ p \text{ and an elliptic curve defined over} \ \mathbb{Q} \text{ or} \ \mathbb{Q}_p, \text{ computes the trace of Frobenius} \ a_p = p + 1 - \#E(F_p). \text{ If} \ E \text{ is defined over a non-prime finite field} \ F_q, \text{ ignore} \ p \text{ and return} q + 1 - \#E(F_q). \text{ When} \ p \text{ is implied} \ (E \text{ defined over} \ \mathbb{Q}_q \text{ or a finite field}), \ p \text{ can be omitted (set to} \ \text{NULL}).
\]
\[
\text{GEN ellsea(GEN E, GEN p, long s)} \text{ available if the} \ \text{seadata} \text{ package is installed. This function returns} \ \#E(F_p), \text{ using the Schoof-Elkies-Atkin algorithm; it is called by} \ \text{ellap} \text{ same conditions as above for} \ E, \text{ except that} \ t_{\text{ELL}_Fq} \text{ are not allowed. The extra flag} \ s, \text{ if set to a non-zero value, causes the computation to return} \ \text{gen}_0 \text{ (an impossible cardinality) if one of the small primes} \ \ell > s \text{ divides the curve order. For cryptographic applications, where one is usually interested in curves of prime order, setting} \ s = 1 \text{ efficiently weeds out most uninteresting curves; if curves of order a power of} 2 \text{ times a prime are acceptable, set} s = 2. \text{ There is no guarantee that the resulting cardinality is prime, only that it has no small prime divisor larger than} s.
\]

242
13.1.6.2 Curves defined a non-prime finite field. In this subsection, we assume that `ell_get_type(E)` is `t_ELL_Fq`. (As noted above, a curve defined over \( \mathbb{Z}/p\mathbb{Z} \) can be represented as a `t_ELL_Fq`.)

**GEN FF_ellmul(GEN E, GEN P, GEN n)** returns \([n]P\) where \(n\) is an integer and \(P\) is a point on the curve \(E\).

**GEN FF_ellrandom(GEN E)** returns a random point in \(E(F_q)\). This function never returns the point at infinity, unless this is the only point on the curve.

**GEN FF_ellorder(GEN E, GEN P, GEN o)** returns the order of the point \(P\), where \(o\) is a multiple of the order of \(P\), or its factorization.

**GEN FF_ellcard(GEN E)** returns \(#E(F_q)\).

**GEN FF_ellgens(GEN E)** returns the generators of the group \(E(F_q)\).

**GEN FF_elllog(GEN E, GEN P, GEN G, GEN o)** Let \(G\) be a point of order \(o\), return \(e\) such that \([e]P = G\). If \(e\) does not exists, the result is undefined.

**GEN FF_ellgroup(GEN E)** returns the Abelian group \(E(F_q)\) in the form \([h, \text{cyc}, \text{gen}]\).

**GEN FF_ellweilpairing(GEN E, GEN P, GEN Q, GEN m)** returns the Weil pairing of the points of \(m\)-torsion \(P\) and \(Q\).

**GEN FF_elltatepairing(GEN E, GEN P, GEN Q, GEN m)** returns the Tate pairing of \(P\) and \(Q\), where \([m]P = 0\).

13.2 Arithmetic on elliptic curve over a finite field in simple form.

The functions in this section no longer operate on elliptic curve structures, as seen up to now. They are used to implement those higher-level functions without using cached information and thus require suitable explicitly enumerated data.

13.2.1 Helper functions.

**GEN elltrace_extension(GEN t, long n, GEN q)** Let \(E\) some elliptic curve over \(F_q\) such that the trace of the Frobenius is \(t\), returns the trace of the Frobenius over \(F_nq\).

13.2.2 Elliptic curves over \(F_p\), \(p > 3\).

Let \(p\) a prime number and \(E\) the elliptic curve given by the equation \(E : y^2 = x^3 + a_4x + a_6\), with \(a_4\) and \(a_6\) in \(F_p\). A \(FpE\) is a point of \(E(F_p)\). Since an affine point and \(a_4\) determine an unique \(a_6\), most functions do not take \(a_6\) as an argument. A \(FpE\) is either the point at infinity (\(ellinf()\)) or a \(FpV\) with two components. The parameters \(a_4\) and \(a_6\) are given as \(t\_INTs\) when required.

**GEN Fp_ellj(GEN a4, GEN a6, GEN p)** returns the \(j\)-invariant of the curve \(E\).

**GEN Fp_ellcard(GEN a4, GEN a6, GEN p)** returns the cardinal of the group \(E(F_p)\).

**GEN Fp_ellcard_SEA(GEN a4, GEN a6, GEN p, long s)** same as `ellsea` when only \([a_4, a_6]\) are given.

**GEN Fq_ellcard_SEA(GEN a4, GEN a6, GEN q, GEN T, GEN p, long s)** same as `ellsea` when only \([a_4, a_6]\) are given, over \(F_p[t]/(T)\). Assume \(p \neq 2, 3\).
GEN Fp_ffellcard(GEN a4, GEN a6, GEN q, long n, GEN p) returns the cardinal of the group $E(F_q)$ where $q = p^n$.

GEN Fp_ellgroup(GEN a4, GEN a6, GEN N, GEN p, GEN *pt_m) returns the group structure $D$ of the group $E(F_p)$, which is assumed to be of order $N$ and set $pt_m = m$.

GEN Fp_ellgens(GEN a4, GEN a6, GEN ch, GEN D, GEN m, GEN p) returns generators of the group $E(F_p)$ with the base change $ch$ (see FpE_changepoint), where $D$ and $m$ are as returned by Fp_ellgroup.

GEN Fp_elldivpol(GEN a4, GEN a6, long n, GEN p) returns the $n$-division polynomial of the elliptic curve $E$.

13.2.3 FpE.

GEN FpE_add(GEN P, GEN Q, GEN a4, GEN p) returns the sum $P + Q$ in the group $E(F_p)$, where $E$ is defined by $E : y^2 = x^3 + a_4x + a_6$, for any value of $a_6$ compatible with the points given.

GEN FpE_sub(GEN P, GEN Q, GEN a4, GEN p) returns $P - Q$.

GEN FpE_dbl(GEN P, GEN a4, GEN p) returns $2.P$.

GEN FpE_neg(GEN P, GEN p) returns $-P$.

GEN FpE_mul(GEN P, GEN n, GEN a4, GEN p) returns $n.P$.

GEN FpE_changepoint(GEN P, GEN m, GEN a4, GEN p) returns the image $Q$ of the point $P$ on the curve $E : y^2 = x^3 + a_4x + a_6$ by the coordinate change $m$ (which is a FpV).

GEN FpE_changepointinv(GEN P, GEN m, GEN a4, GEN p) returns the image $Q$ on the curve $E : y^2 = x^3 + a_4x + a_6$ of the point $P$ by the inverse of the coordinate change $m$ (which is a FpV).

GEN random_FpE(GEN a4, GEN a6, GEN p) returns a random point on $E(F_p)$, where $E$ is defined by $E : y^2 = x^3 + a_4x + a_6$.

GEN FpE_order(GEN P, GEN o, GEN a4, GEN p) returns the order of $P$ in the group $E(F_p)$, where $o$ is a multiple of the order of $P$, or its factorization.

GEN FpE_log(GEN P, GEN G, GEN o, GEN a4, GEN p) Let $G$ be a point of order $o$, return $e$ such that $e.P = G$. If $e$ does not exists, the result is currently undefined.

GEN FpE_tatepairing(GEN P, GEN Q, GEN m, GEN a4, GEN p) returns the Tate pairing of the point of $m$-torsion $P$ and the point $Q$.

GEN FpE_weilpairing(GEN P, GEN Q, GEN m, GEN a4, GEN p) returns the Weil pairing of the points of $m$-torsion $P$ and $Q$.

GEN FpE_to_mod(GEN P, GEN p) returns $P$ as a vector of t_INTMODs.

GEN RgE_to_FpE(GEN P, GEN p) returns the FpE obtained by applying Rg_to_Fp coefficientwise.
13.2.4 Fl	ext{e}. Let \( p \) be a prime ulong, and \( E \) the elliptic curve given by the equation \( E : y^2 = x^3 + a_4 x + a_6 \), where \( a_4 \) and \( a_6 \) are ulong. A Fl	ext{e} is either the point at infinity (\text{ellinf()}), or a Fl	ext{e} with two components.

long Fl_elltrace(ulong a4, ulong a6, ulong p) returns the trace \( t \) of the Frobenius of \( E(\mathbb{F}_p) \). The cardinal of \( E(\mathbb{F}_p) \) is thus \( p + 1 - t \), which might not fit in a ulong.

GEN Fl	ext{e}_add(GEN P, GEN Q, ulong a4, ulong p)
GEN Fl	ext{e}_db1(GEN P, ulong a4, ulong p)
GEN Fl	ext{e}_sub(GEN P, GEN Q, ulong a4, ulong p)
GEN Fl	ext{e}_mul(GEN P, GEN n, ulong a4, ulong p)
GEN Fl	ext{e}_mulu(GEN P, ulong n, ulong a4, ulong p)
GEN Fl	ext{e}_order(GEN P, GEN o, ulong a4, ulong p)
GEN random_Fl	ext{e}(ulong a4, ulong a6, ulong p)

13.2.5 Elliptic curves over \( \mathbb{F}_{2^n} \). Let \( T \) be an irreducible \( \mathbb{F}_2 x \) and \( E \) the elliptic curve given by either the equation \( E : y^2 + x y = x^3 + a_2 x^2 + a_6 \), where \( a_2, a_6 \) are \( \mathbb{F}_2 x \) in \( \mathbb{F}_2 [X]/(T) \) (ordinary case) or \( E : y^2 + a_3 y = x^3 + a_4 x + a_6 \), where \( a_3, a_4, a_6 \) are \( \mathbb{F}_2 x \) in \( \mathbb{F}_2 [X]/(T) \) (supersingular case).

A \( \mathbb{F}_2 x q E \) is a point of \( E(\mathbb{F}_2[X]/(T)) \). In the supersingular case, the parameter \( a_2 \) is actually the \( t \_ \_ \_ V [a_3, a_4, a_3^{-1}] \).

GEN F2xq_ellcard(GEN a2, GEN a6, GEN T) Return the order of the group \( E(\mathbb{F}_2[X]/(T)) \).
GEN F2xq_ellgroup(GEN a2, GEN a6, GEN N, GEN T, GEN *pt_m) Return the group structure \( D \) of the group \( E(\mathbb{F}_2[X]/(T)) \), which is assumed to be of order \( N \) and set \( *pt_m = m \).
GEN F2xq_ellgens(GEN a2, GEN a6, GEN ch, GEN D, GEN m, GEN T) Returns generators of the group \( E(\mathbb{F}_2[X]/(T)) \) with the base change \( ch \) (see \text{F2xqE_changepoint}), where \( D \) and \( m \) are as returned by \text{F2xqE_ellgroup}.

13.2.6 F2xqE.
GEN F2xqE_changepoint(GEN P, GEN m, GEN a2, GEN T) returns the image \( Q \) of the point \( P \) on the curve \( E : y^2 + x y = x^3 + a_4 x + a_6 \) by the coordinate change \( m \) (which is a \( \mathbb{F}_2 x q V \)).
GEN F2xqE_changepointinv(GEN P, GEN m, GEN a2, GEN T) returns the image \( Q \) on the curve \( E : y^2 = x^3 + a_4 x + a_6 \) of the point \( P \) by the inverse of the coordinate change \( m \) (which is a \( \mathbb{F}_2 x q V \)).
GEN F2xqE_add(GEN P, GEN Q, GEN a2, GEN T)
GEN F2xqE_sub(GEN P, GEN Q, GEN a2, GEN T)
GEN F2xqE_db1(GEN P, GEN a2, GEN T)
GEN F2xqE_neg(GEN P, GEN a2, GEN T)
GEN F2xqE_mul(GEN P, GEN n, GEN a2, GEN T)
GEN random_F2xqE(GEN a2, GEN a6, GEN T)
GEN F2xqE_order(GEN P, GEN o, GEN a2, GEN T) returns the order of \( P \) in the group \( E(\mathbb{F}_2[X]/(T)) \), where \( o \) is a multiple of the order of \( P \), or its factorization.

245
GEN F2xqE_log(GEN P, GEN G, GEN o, GEN a2, GEN T) Let G be a point of order o, return e such that $e.P = G$. If e does not exists, the result is currently undefined.

GEN F2xqE_tatepairing(GEN P, GEN Q, GEN m, GEN a2, GEN T) returns the Tate pairing of the point of $m$-torsion $P$ and the point $Q$.

GEN F2xqE_weilpairing(GEN Q, GEN Q, GEN m, GEN a2, GEN T) returns the Weil pairing of the points of $m$-torsion $P$ and $Q$.

GEN RgE_to_F2xqE(GEN P, GEN T) returns the $F_{2xqE}$ obtained by applying $Rg$ to $F_{2xq}$ coefficient-wise.

13.2.7 Elliptic curves over $F_q$, small characteristic $p > 2$. Let $p$ be a prime ulong, $T$ an irreducible $F_{lx}$ mod $p$, and $E$ the elliptic curve given by the equation $E : y^2 = x^3 + a_4 x + a_6$, where $a_4$ and $a_6$ are $F_{lx}$ in $F_p[X]/(T)$. A $F_{lxqE}$ is a point of $E(F_{p[X]}/(T))$. In that case, the parameter $a_2$ is actually stored as a t_VEC, $[a_2]$, to avoid ambiguities.

GEN Flxq_ellj(GEN a4, GEN a6, GEN T, ulong p) returns the $j$-invariant of the curve $E$.

GEN Flxq_ellcard(GEN a4, GEN a6, GEN T, ulong p) returns the order of $E(F_{p[X]}/(T))$.

GEN Flxq_ellgroup(GEN a4, GEN a6, GEN N, GEN T, ulong p, GEN *pt_m) returns the group structure $D$ of the group $E(F_{p[X]}/(T))$, which is assumed to be of order $N$ and set $*pt_m = m$.

GEN Flxq_ellgens(GEN a4, GEN a6, GEN ch, GEN D, GEN m, GEN T, ulong p) returns generators of the group $E(F_{p[X]}/(T))$ with the base change $ch$ (see FlxqE_changepoint), where $D$ and $m$ are as returned by Flxq_ellgroup.

13.2.8 FlxqE.

GEN FlxqE_changepoint(GEN P, GEN m, GEN a4, GEN T, ulong p) returns the image $Q$ of the point $P$ on the curve $E : y^2 = x^3 + a_4 x + a_6$ by the coordinate change $m$ (which is a $FlxqV$).

GEN FlxqE_changepointinv(GEN P, GEN m, GEN a4, GEN T, ulong p) returns the image $Q$ on the curve $E : y^2 = x^3 + a_4 x + a_6$ of the point $P$ by the inverse of the coordinate change $m$ (which is a $FlxqV$).

GEN FlxqE_add(GEN P, GEN Q, GEN a4, GEN T, ulong p)
GEN FlxqE_sub(GEN P, GEN Q, GEN a4, GEN T, ulong p)
GEN FlxqE_dbl(GEN P, GEN a4, GEN T, ulong p)
GEN FlxqE_neg(GEN P, GEN T, ulong p)
GEN FlxqE_mul(GEN P, GEN n, GEN a4, GEN T, ulong p)
GEN random_FlxqE(GEN a4, GEN a6, GEN T, ulong p)
GEN FlxqE_order(GEN P, GEN o, GEN a4, GEN T, ulong p) returns the order of $P$ in the group $E(F_{p[X]}/(T))$, where $o$ is a multiple of the order of $P$, or its factorization.

GEN FlxqE_log(GEN P, GEN G, GEN o, GEN a4, GEN T, ulong p) Let $G$ be a point of order $o$, return $e$ such that $e.P = G$. If $e$ does not exists, the result is currently undefined.
GEN FlxqE_tatepairing(GEN P, GEN Q, GEN m, GEN a4, GEN T, ulong p) returns the Tate pairing of the point of $m$-torsion $P$ and the point $Q$.

GEN FlxqE_weilpairing(GEN P, GEN Q, GEN m, GEN a4, GEN T, ulong p) returns the Weil pairing of the points of $m$-torsion $P$ and $Q$.

GEN RgE_to_FlxqE(GEN P, GEN T, ulong p) returns the FlxqE obtained by applying $Rg$ to Flxq coefficientwise.

13.2.9 Elliptic curves over $F_q$, large characteristic.

Let $p$ be a prime number, $T$ an irreducible polynomial mod $p$, and $E$ the elliptic curve given by the equation $E : y^2 = x^3 + a_4 x + a_6$ with $a_4$ and $a_6$ in $F_p[X]/(T)$. A FpXQE is a point of $E(F_p[X]/(T))$.

GEN FpXQ_ellj(GEN a4, GEN a6, GEN T, GEN p) returns the $j$-invariant of the curve $E$.

GEN FpXQ_ellcard(GEN a4, GEN a6, GEN T, GEN p) returns the order of $E(F_p[X]/(T))$.

GEN FpXQ_ellgroup(GEN a4, GEN a6, GEN N, GEN T, GEN p, GEN *pt_m) returns the group structure $D$ of the group $E(F_p[X]/(T))$, which is assumed to be of order $N$ and set $pt_m = m$.

GEN FpXQ_ellgens(GEN a4, GEN a6, GEN ch, GEN D, GEN m, GEN T, GEN p) returns generators of the group $E(F_p[X]/(T))$ with the base change $ch$ (see FpXQE_changepoint), where $D$ and $m$ are as returned by FpXQ_ellgroup.

GEN FpXQ_elldivpol(GEN a4, GEN a6, long n, GEN T, GEN p) returns the $n$-division polynomial of the elliptic curve $E$.

GEN FpXQ_elldivpolmod(GEN a4, GEN a6, long n, GEN h, GEN T, GEN p) returns the $n$-division polynomial of the elliptic curve $E$ modulo the polynomial $h$.

13.2.10 FpXQE.

GEN FpXQE_changepoint(GEN P, GEN m, GEN a4, GEN T, GEN p) returns the image $Q$ of the point $P$ on the curve $E : y^2 = x^3 + a_4 x + a_6$ by the coordinate change $m$ (which is a FpXQE).

GEN FpXQE_changepointinv(GEN P, GEN m, GEN a4, GEN T, GEN p) returns the image $Q$ on the curve $E : y^2 = x^3 + a_4 x + a_6$ of the point $P$ by the inverse of the coordinate change $m$ (which is a FpXQE).

GEN FpXQE_add(GEN P, GEN Q, GEN a4, GEN T, GEN p)
GEN FpXQE_sub(GEN P, GEN Q, GEN a4, GEN T, GEN p)
GEN FpXQE_dbl(GEN P, GEN a4, GEN T, GEN p)
GEN FpXQE_neg(GEN P, GEN T, GEN p)
GEN FpXQE_mul(GEN P, GEN n, GEN a4, GEN T, GEN p)
GEN random_FpXQE(GEN a4, GEN a6, GEN T, GEN p)
GEN FpXQE_log(GEN P, GEN G, GEN o, GEN a4, GEN T, GEN p) Let $G$ be a point of order $o$, return $e$ such that $e.P = G$. If $e$ does not exists, the result is currently undefined.

GEN FpXQE_order(GEN P, GEN o, GEN a4, GEN T, GEN p) returns the order of $P$ in the group $E(F_p[X]/(T))$, where $o$ is a multiple of the order of $P$, or its factorization.

247
GEN FpXQE_tatepairing(GEN P, GEN Q, GEN m, GEN a4, GEN T, GEN p) returns the Tate pairing of the point of $m$-torsion $P$ and the point $Q$.

GEN FpXQE_weilpairing(GEN P, GEN Q, GEN m, GEN a4, GEN T, GEN p) returns the Weil pairing of the points of $m$-torsion $P$ and $Q$.

GEN RgE_to_FpXQE(GEN P, GEN T, GEN p) returns the $FpXQE$ obtained by applying $Rg$ to $FpXQ$ coefficientwise.

13.3 Other curves.

The following functions deal with hyperelliptic curves in weighted projective space $\mathbb{P}_{(1,d,1)}$, with coordinates $(x,y,z)$ and a model of the form $y^2 = T(x,z)$, where $T$ is homogeneous of degree $2d$, and squarefree. Thus the curve is nonsingular of genus $d - 1$.

long hyperell_locally_soluble(GEN T, GEN p) assumes that $T \in \mathbb{Z}[X]$ is integral. Returns 1 if the curve is locally soluble over $\mathbb{Q}_p$, 0 otherwise.

long nf_hyperell_locally_soluble(GEN nf, GEN T, GEN pr) let $K$ be a number field, associated to $nf$, pr a prid associated to some maximal ideal $p$; assumes that $T \in \mathbb{Z}_K[X]$ is integral. Returns 1 if the curve is locally soluble over $K_p$. 

248
Appendix A:
A Sample program and Makefile

We assume that you have installed the PARI library and include files as explained in Appendix A or in the installation guide. If you chose differently any of the directory names, change them accordingly in the Makefiles.

If the program example that we have given is in the file extgcd.c, then a sample Makefile might look as follows. Note that the actual file examples/Makefile is more elaborate and you should have a look at it if you intend to use install() on custom made functions, see Section 3.12.22.

```
CC = cc
INCDIR = /home/kb/PARI/pari/../GP/include
LIBDIR = /home/kb/PARI/pari/../GP/lib
CFLAGS = -O -I$(INCDIR) -L$(LIBDIR)
all: extgcd
extgcd: extgcd.c
    $(CC) $(CFLAGS) -o extgcd extgcd.c -lpari -lm
```

We then give the listing of the program examples/extgcd.c seen in detail in Section 4.10.

```
#include <pari/pari.h>
/*
GP;install("extgcd", "GG&k", "gcdex", "/.libextgcd.so");
*/
/* return d = gcd(a,b), sets u, v such that au + bv = gcd(a,b) */
GEN
extgcd(GEN A, GEN B, GEN *U, GEN *V)
{
    pari_sp av = avma;
    GEN ux = gen_1, vx = gen_0, a = A, b = B;
    if (typ(a) != t_INT) pari_err_TYPE("extgcd",a);
    if (typ(b) != t_INT) pari_err_TYPE("extgcd",b);
    if (signe(a) < 0) { a = negi(a); ux = negi(ux); }
    while (!gequal0(b))
    {
        GEN r, q = dvmdii(a, b, &r), v = vx;
        vx = subii(ux, mulii(q, vx));
        ux = v; a = b; b = r;
    }
    *U = ux;
    *V = diviexact( subii(a, mulii(A,ux)), B );
gerepileall(av, 3, &a, U, V); return a;
}
int
```
main()
{
    GEN x, y, d, u, v;
    pari_init(1000000, 2);
    printf("x = "); x = gp_read_stream(stdin);
    printf("y = "); y = gp_read_stream(stdin);
    d = extgcd(x, y, &u, &v);
    pari_printf("gcd = %Ps\nu = %Ps\nv = %Ps\n", d, u, v);
    pari_close();
    return 0;
}
Appendix B:

PARI and threads

To use PARI in multi-threaded programs, you must configure it using `Configure --enable-tls`. Your system must implement the _thread storage class. As a major side effect, this breaks the `libpari` ABI: the resulting library is not compatible with the old one, and `-tls` is appended to the PARI library `soname`. On the other hand, this library is now thread-safe.

PARI provides some functions to set up PARI subthreads. In our model, each concurrent thread needs its own PARI stack. The following scheme is used:

**Child thread:**

```c
void *child_thread(void *arg)
{
    GEN data = pari_thread_start((struct pari_thread*)arg);
    GEN result = ...; /* Compute result from data */
    pari_thread_close();
    return (void*)result;
}
```

**Parent thread:**

```c
pthread_t th;
struct pari_thread pth;
GEN data, result;
pari_thread_alloc(&pth, s, data);
pthread_create(&th, NULL, &child_thread, (void*)&pth); /* start child */
... /* do stuff in parent */
pthread_join(th, (void*)&result); /* wait until child terminates */
result = gcopy(result); /* copy result from thread stack to main stack */
pari_thread_free(&pth); /* ... and clean up */
```

`void pari_thread_alloc(struct pari_thread *pth, size_t s, GEN arg)` Allocate a PARI stack of size `s` and associate it, together with the argument `arg`, with the PARI thread data `pth`.

`void pari_thread_free(struct pari_thread *pth)` Free the PARI stack associated with the PARI thread data `pth`. This is called after the child thread terminates, i.e. after `pthread_join` in the parent. Any GEN objects returned by the child in the thread stack need to be saved before running this command.

`void pari_thread_init(void)` Initialize the thread-local PARI data structures. This function is called by `pari_thread_start`.

`GEN pari_thread_start(struct pari_thread *t)` Initialize the thread-local PARI data structures and set up the thread stack using the PARI thread data `pth`. This function returns the thread argument `arg` that was given to `pari_thread_alloc`. 

251
void pari_thread_close(void) Free the thread-local PARI data structures, but keeping the thread stack, so that a GEN returned by the thread remains valid.

Under this model, some PARI states are reset in new threads. In particular

- the random number generator is reset to the starting seed;
- the system stack exhaustion checking code, meant to catch infinite recursions, is disabled (use pari_stackcheck_init() to reenable it);
- cached real constants (returned by mmpi, mpeuler and mplog2) are not shared between threads and will be recomputed as needed;

The following sample program can be compiled using

    cc thread.c -o thread.o -lpari -lpthread

(Add -I/-L paths as necessary.)

```c
#include <pari/pari.h> /* Include PARI headers */
#include <pthread.h> /* Include POSIX threads headers */

void *
mydet(void *arg)
{
    GEN F, M;
    /* Set up thread stack and get thread parameter */
    M = pari_thread_start((struct pari_thread*) arg);
    F = det(M);
    /* Free memory used by the thread */
    pari_thread_close();
    return (void*)F;
}

void *
myfactor(void *arg) /* same principle */
{
    GEN F, N;
    N = pari_thread_start((struct pari_thread*) arg);
    F = factor(N);
    pari_thread_close();
    return (void*)F;
}

int
main(void)
{
    GEN M, N1, N2, F1, F2, D;
    pthread_t th1, th2, th3; /* POSIX-thread variables */
    struct pari_thread pth1, pth2, pth3; /* pari thread variables */
    /* Initialise the main PARI stack and global objects (gen_0, etc.) */
    pari_init(4000000, 5000000);
    /* Compute in the main PARI stack */
    N1 = addis(int2n(256), 1); /* 2^256 + 1 */
```
N2 = subis(int2n(193), 1); /* 2^193 - 1 */
M = mathilbert(80);

/* Allocate pari thread structures */
pari_thread_alloc(&pth1, 4000000, N1);
pari_thread_alloc(&pth2, 4000000, N2);
pari_thread_alloc(&pth3, 4000000, M);

/* pthread_create() and pthread_join() are standard POSIX-thread
 * functions to start and get the result of threads. */
pthread_create(&th1, NULL, &myfactor, (void*) &pth1);
pthread_create(&th2, NULL, &myfactor, (void*) &pth2);
pthread_create(&th3, NULL, &mydet, (void*) &pth3); /* Start 3 threads */
pthread_join(th1, (void*) &F1);
pthread_join(th2, (void*) &F2);
pthread_join(th3, (void*) &D); /* Wait for termination, get the results */
pari_printf("F1=%Ps\nF2=%Ps\nlog(D)=%Ps\n", F1, F2, glog(D, 3));
pari_thread_free(&pth1);
pari_thread_free(&pth2);
pari_thread_free(&pth3); /* clean up */
return 0;
Index

*SomeWord* refers to PARI-GP concepts.
*SomeWord* is a PARI-GP keyword.
*SomeWord* is a generic index entry.

### A

<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC_to_bnr</td>
<td>227</td>
</tr>
<tr>
<td>abelian_group</td>
<td>183</td>
</tr>
<tr>
<td>abgrp_get_cyc</td>
<td>210</td>
</tr>
<tr>
<td>abgrp_get_gen</td>
<td>210</td>
</tr>
<tr>
<td>abgrp_get_no</td>
<td>210</td>
</tr>
<tr>
<td>absfrac</td>
<td>175</td>
</tr>
<tr>
<td>absfrac_shallow</td>
<td>175</td>
</tr>
<tr>
<td>absi</td>
<td>79</td>
</tr>
<tr>
<td>absi_cmp</td>
<td>80</td>
</tr>
<tr>
<td>absi_equal</td>
<td>80</td>
</tr>
<tr>
<td>absi_factor</td>
<td>128</td>
</tr>
<tr>
<td>absi_factor_limit</td>
<td>128</td>
</tr>
<tr>
<td>absi_shallow</td>
<td>79</td>
</tr>
<tr>
<td>absr</td>
<td>79</td>
</tr>
<tr>
<td>absrnz_equal1</td>
<td>80</td>
</tr>
<tr>
<td>absrnz_equal12n</td>
<td>80</td>
</tr>
<tr>
<td>absr_cmp</td>
<td>80</td>
</tr>
<tr>
<td>abstorel</td>
<td>228</td>
</tr>
<tr>
<td>addhelp</td>
<td>68</td>
</tr>
<tr>
<td>addi</td>
<td>13</td>
</tr>
<tr>
<td>addii_sign</td>
<td>83</td>
</tr>
<tr>
<td>addir</td>
<td>13</td>
</tr>
<tr>
<td>addir_sign</td>
<td>83</td>
</tr>
<tr>
<td>addis</td>
<td>13</td>
</tr>
<tr>
<td>addiu</td>
<td>82</td>
</tr>
<tr>
<td>addil</td>
<td>71</td>
</tr>
<tr>
<td>addilx</td>
<td>71</td>
</tr>
<tr>
<td>addmul</td>
<td>71</td>
</tr>
<tr>
<td>addmulii</td>
<td>82</td>
</tr>
<tr>
<td>addmulii_inplace</td>
<td>82</td>
</tr>
<tr>
<td>addmuliu</td>
<td>82</td>
</tr>
<tr>
<td>addmuliu_inplace</td>
<td>82</td>
</tr>
<tr>
<td>addri</td>
<td>13</td>
</tr>
<tr>
<td>addrr</td>
<td>13</td>
</tr>
<tr>
<td>addrr_sign</td>
<td>83</td>
</tr>
<tr>
<td>addsi_sign</td>
<td>83</td>
</tr>
<tr>
<td>addui</td>
<td>82</td>
</tr>
<tr>
<td>addui_sign</td>
<td>83</td>
</tr>
<tr>
<td>addumului</td>
<td>82</td>
</tr>
<tr>
<td>adduu</td>
<td>82</td>
</tr>
<tr>
<td>affc_fixlg</td>
<td>181</td>
</tr>
<tr>
<td>affectsign</td>
<td>54</td>
</tr>
<tr>
<td>affectsign_safe</td>
<td>55</td>
</tr>
<tr>
<td>affr</td>
<td>75</td>
</tr>
<tr>
<td>affii</td>
<td>75</td>
</tr>
<tr>
<td>affir</td>
<td>75</td>
</tr>
<tr>
<td>affiz</td>
<td>75</td>
</tr>
<tr>
<td>affrr</td>
<td>75</td>
</tr>
<tr>
<td>affrr_fixlg</td>
<td>75, 181</td>
</tr>
<tr>
<td>affsi</td>
<td>75</td>
</tr>
<tr>
<td>affsr</td>
<td>75</td>
</tr>
<tr>
<td>affsz</td>
<td>75</td>
</tr>
<tr>
<td>affui</td>
<td>75</td>
</tr>
<tr>
<td>affur</td>
<td>75</td>
</tr>
<tr>
<td>alarm</td>
<td>192</td>
</tr>
<tr>
<td>assignment</td>
<td>24</td>
</tr>
<tr>
<td>avma</td>
<td>15, 24</td>
</tr>
</tbody>
</table>

### B

<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bernfrac</td>
<td>181, 182</td>
</tr>
<tr>
<td>Bernoulli</td>
<td>181, 182</td>
</tr>
<tr>
<td>bernreal</td>
<td>181, 182</td>
</tr>
<tr>
<td>bezout</td>
<td>43, 86</td>
</tr>
<tr>
<td>bfffo</td>
<td>71</td>
</tr>
<tr>
<td>bid_get_arch</td>
<td>213</td>
</tr>
<tr>
<td>bid_get_cyc</td>
<td>213</td>
</tr>
<tr>
<td>bid_get_gen</td>
<td>213</td>
</tr>
<tr>
<td>bid_get_gen_nocheck</td>
<td>213</td>
</tr>
<tr>
<td>bid_get_grp</td>
<td>213</td>
</tr>
<tr>
<td>bid_get_ideal</td>
<td>213</td>
</tr>
<tr>
<td>bid_get_mod</td>
<td>213</td>
</tr>
<tr>
<td>bid_get_no</td>
<td>213</td>
</tr>
<tr>
<td>BIGDEFAULTPREC</td>
<td>14, 56</td>
</tr>
<tr>
<td>BIL</td>
<td>47</td>
</tr>
<tr>
<td>bin_copy</td>
<td>60</td>
</tr>
<tr>
<td>bin_copy_relink</td>
<td>60</td>
</tr>
<tr>
<td>BITS_IN_HALFULONG</td>
<td>56</td>
</tr>
<tr>
<td>BITS_IN_LONG</td>
<td>14, 47, 56</td>
</tr>
<tr>
<td>bit_accuracy</td>
<td>14, 52</td>
</tr>
<tr>
<td>bit_accuracy_mul</td>
<td>52</td>
</tr>
<tr>
<td>bl_base</td>
<td>64</td>
</tr>
<tr>
<td>bl_next</td>
<td>64</td>
</tr>
<tr>
<td>bl_num</td>
<td>64</td>
</tr>
<tr>
<td>bl_prev</td>
<td>64</td>
</tr>
<tr>
<td>bl_refc</td>
<td>64</td>
</tr>
<tr>
<td>bnfisprincipal10</td>
<td>213, 223, 226</td>
</tr>
<tr>
<td>bnfisunit</td>
<td>217</td>
</tr>
<tr>
<td>bnfnewprec</td>
<td>213, 224</td>
</tr>
<tr>
<td>bnfnewprec_shallow</td>
<td>213</td>
</tr>
<tr>
<td>bnf_get_clgp</td>
<td>211</td>
</tr>
<tr>
<td>bnf_get_cyc</td>
<td>211</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
</tr>
<tr>
<td>bnf_get_fu</td>
<td>211</td>
</tr>
<tr>
<td>bnf_get_fu_nocheck</td>
<td>211</td>
</tr>
<tr>
<td>bnf_get_gen</td>
<td>211</td>
</tr>
<tr>
<td>bnf_get_logfu</td>
<td>211</td>
</tr>
<tr>
<td>bnf_get_nf</td>
<td>211</td>
</tr>
<tr>
<td>bnf_get_no</td>
<td>211</td>
</tr>
<tr>
<td>bnf_get_reg</td>
<td>211</td>
</tr>
<tr>
<td>bnf_get_tuW</td>
<td>211</td>
</tr>
<tr>
<td>bnf_get_tuU</td>
<td>211</td>
</tr>
<tr>
<td>bnrclassno</td>
<td>227</td>
</tr>
<tr>
<td>bnrcaracter</td>
<td>227</td>
</tr>
<tr>
<td>bnrclassno</td>
<td>227</td>
</tr>
<tr>
<td>bnrdisc</td>
<td>227, 230</td>
</tr>
<tr>
<td>bnrdisclist0</td>
<td>230</td>
</tr>
<tr>
<td>bnrisconductor</td>
<td>227</td>
</tr>
<tr>
<td>bnrisprincipal</td>
<td>229</td>
</tr>
<tr>
<td>bnrnewprec</td>
<td>213</td>
</tr>
<tr>
<td>bnrnewprec_shallow</td>
<td>213</td>
</tr>
<tr>
<td>bnrsurjection</td>
<td>227</td>
</tr>
<tr>
<td>bnr_get_bid</td>
<td>212</td>
</tr>
<tr>
<td>bnr_get_bnf</td>
<td>212</td>
</tr>
<tr>
<td>bnr_get_clgp</td>
<td>212</td>
</tr>
<tr>
<td>bnr_get_cyc</td>
<td>212</td>
</tr>
<tr>
<td>bnr_get_gen</td>
<td>212</td>
</tr>
<tr>
<td>bnr_get_gen_nocheck</td>
<td>212</td>
</tr>
<tr>
<td>bnr_get_mod</td>
<td>212</td>
</tr>
<tr>
<td>bnr_get_nf</td>
<td>212</td>
</tr>
<tr>
<td>bnr_get_no</td>
<td>212</td>
</tr>
<tr>
<td>both_odd</td>
<td>72</td>
</tr>
<tr>
<td>boundfact</td>
<td>128</td>
</tr>
<tr>
<td>BPSW_isprime</td>
<td>132</td>
</tr>
<tr>
<td>BPSW_psp</td>
<td>131, 132</td>
</tr>
<tr>
<td>brute</td>
<td>188</td>
</tr>
<tr>
<td>buchimag</td>
<td>229</td>
</tr>
<tr>
<td>Buchray</td>
<td>227</td>
</tr>
<tr>
<td>buchreal</td>
<td>229</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATCH_ALL</td>
<td>42</td>
</tr>
<tr>
<td>cbezout</td>
<td>86</td>
</tr>
<tr>
<td>cb pari_ask_confirm</td>
<td>49, 50</td>
</tr>
<tr>
<td>cb pari_err_recover</td>
<td>50</td>
</tr>
<tr>
<td>cb pari_handle_exception</td>
<td>49</td>
</tr>
<tr>
<td>cb pari_pre_recover</td>
<td>50</td>
</tr>
<tr>
<td>cb pari_simgnt</td>
<td>50</td>
</tr>
<tr>
<td>cb pari_whatnow</td>
<td>50</td>
</tr>
<tr>
<td>ceilr</td>
<td>76</td>
</tr>
<tr>
<td>ceil_safe</td>
<td>77</td>
</tr>
<tr>
<td>centerlift</td>
<td>154</td>
</tr>
<tr>
<td>centerlift0</td>
<td>154</td>
</tr>
<tr>
<td>centermod</td>
<td>168</td>
</tr>
<tr>
<td>centermodi</td>
<td>83</td>
</tr>
<tr>
<td>centermod_i</td>
<td>168</td>
</tr>
<tr>
<td>cgcdf</td>
<td>86</td>
</tr>
<tr>
<td>cgetalloc</td>
<td>59</td>
</tr>
<tr>
<td>cgetc</td>
<td>22, 51, 59, 74, 181</td>
</tr>
<tr>
<td>cgetg</td>
<td>22, 23, 51, 59</td>
</tr>
<tr>
<td>cgetg_block</td>
<td>63</td>
</tr>
<tr>
<td>cgetg_copy</td>
<td>51</td>
</tr>
<tr>
<td>cgeti</td>
<td>22, 51, 59, 74</td>
</tr>
<tr>
<td>cgetineg</td>
<td>74</td>
</tr>
<tr>
<td>cgetipos</td>
<td>74</td>
</tr>
<tr>
<td>cgetp</td>
<td>59</td>
</tr>
<tr>
<td>cgetr</td>
<td>22, 51, 59, 74</td>
</tr>
<tr>
<td>cgetr_block</td>
<td>63</td>
</tr>
<tr>
<td>cgiv</td>
<td>16, 60</td>
</tr>
<tr>
<td>character string</td>
<td>32</td>
</tr>
<tr>
<td>characteristic</td>
<td>176</td>
</tr>
<tr>
<td>chartoGENstr</td>
<td>185</td>
</tr>
<tr>
<td>checkabgrp</td>
<td>210</td>
</tr>
<tr>
<td>checkbid</td>
<td>209</td>
</tr>
<tr>
<td>checkbnf</td>
<td>209</td>
</tr>
<tr>
<td>checkbnr</td>
<td>209</td>
</tr>
<tr>
<td>checkbnrgen</td>
<td>209</td>
</tr>
<tr>
<td>checkell</td>
<td>239</td>
</tr>
<tr>
<td>checkell5</td>
<td>239</td>
</tr>
<tr>
<td>checkellpt</td>
<td>239</td>
</tr>
<tr>
<td>checkell_Fq</td>
<td>239</td>
</tr>
<tr>
<td>checkell_Q</td>
<td>240</td>
</tr>
<tr>
<td>checkell_Qp</td>
<td>240</td>
</tr>
<tr>
<td>checkgal</td>
<td>209</td>
</tr>
<tr>
<td>checkgroup</td>
<td>183</td>
</tr>
<tr>
<td>checkmodpr</td>
<td>210</td>
</tr>
<tr>
<td>checknf</td>
<td>209</td>
</tr>
<tr>
<td>checknfelt_mod</td>
<td>210</td>
</tr>
<tr>
<td>checkprim</td>
<td>209</td>
</tr>
<tr>
<td>checkrnf</td>
<td>209</td>
</tr>
<tr>
<td>checksqmat</td>
<td>209</td>
</tr>
<tr>
<td>check_arith_all</td>
<td>130</td>
</tr>
<tr>
<td>check_arith_non0</td>
<td>130</td>
</tr>
<tr>
<td>check_arith_pos</td>
<td>129</td>
</tr>
<tr>
<td>check_quaddisc</td>
<td>231</td>
</tr>
<tr>
<td>check_quaddisc_imag</td>
<td>231</td>
</tr>
<tr>
<td>check_quaddisc_real</td>
<td>231</td>
</tr>
<tr>
<td>check_zKmodule</td>
<td>210</td>
</tr>
<tr>
<td>chinese</td>
<td>120</td>
</tr>
<tr>
<td>chinese_coprime_Z</td>
<td>120</td>
</tr>
<tr>
<td>chk gerepileupto</td>
<td>62</td>
</tr>
<tr>
<td>Function/Command</td>
<td>Page</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
</tr>
<tr>
<td>dblexpo</td>
<td>153</td>
</tr>
<tr>
<td>dblog2r</td>
<td>153</td>
</tr>
<tr>
<td>dblmantissa</td>
<td>153</td>
</tr>
<tr>
<td>dbltor</td>
<td>25, 153</td>
</tr>
<tr>
<td>debug</td>
<td>38</td>
</tr>
<tr>
<td>debugging</td>
<td>38</td>
</tr>
<tr>
<td>DEBUGLEVEL</td>
<td>38</td>
</tr>
<tr>
<td>DEBUGMEM</td>
<td>38</td>
</tr>
<tr>
<td>debugmem</td>
<td>38</td>
</tr>
<tr>
<td>default0</td>
<td>49</td>
</tr>
<tr>
<td>DEFAULTPREC</td>
<td>14, 56</td>
</tr>
<tr>
<td>definite binary quadratic form</td>
<td>32</td>
</tr>
<tr>
<td>deg1pol</td>
<td>157</td>
</tr>
<tr>
<td>deg1pol_shallow</td>
<td>161</td>
</tr>
<tr>
<td>deg1_from_roots</td>
<td>161</td>
</tr>
<tr>
<td>degpol</td>
<td>31, 53, 176</td>
</tr>
<tr>
<td>degree</td>
<td>31, 175</td>
</tr>
<tr>
<td>delete_var</td>
<td>35, 65</td>
</tr>
<tr>
<td>derivser</td>
<td>176</td>
</tr>
<tr>
<td>destruction</td>
<td>16</td>
</tr>
<tr>
<td>detint</td>
<td>135</td>
</tr>
<tr>
<td>diagonal_shallow</td>
<td>161</td>
</tr>
<tr>
<td>dicyclicgroup</td>
<td>183</td>
</tr>
<tr>
<td>diffptr</td>
<td>12</td>
</tr>
<tr>
<td>discrayabs</td>
<td>230</td>
</tr>
<tr>
<td>discrayabscond</td>
<td>230</td>
</tr>
<tr>
<td>discrayabslist</td>
<td>230</td>
</tr>
<tr>
<td>discrayabslistarch</td>
<td>230</td>
</tr>
<tr>
<td>discrayabslistlong</td>
<td>230</td>
</tr>
<tr>
<td>discrayrel</td>
<td>230</td>
</tr>
<tr>
<td>discrayrelcond</td>
<td>230</td>
</tr>
<tr>
<td>divide_conquer_assoc</td>
<td>172</td>
</tr>
<tr>
<td>divide_conquer_prod</td>
<td>161, 171, 172</td>
</tr>
<tr>
<td>diviexact</td>
<td>83</td>
</tr>
<tr>
<td>diviiround</td>
<td>76</td>
</tr>
<tr>
<td>divisors</td>
<td>40</td>
</tr>
<tr>
<td>divisorsu</td>
<td>89</td>
</tr>
<tr>
<td>divisrem</td>
<td>84</td>
</tr>
<tr>
<td>diviuexact</td>
<td>83</td>
</tr>
<tr>
<td>diviuuexact</td>
<td>83</td>
</tr>
<tr>
<td>diviu_rem</td>
<td>84</td>
</tr>
<tr>
<td>dvi1</td>
<td>71</td>
</tr>
<tr>
<td>divaBIL</td>
<td>56</td>
</tr>
<tr>
<td>divai_rem</td>
<td>85</td>
</tr>
<tr>
<td>divassrem</td>
<td>85</td>
</tr>
<tr>
<td>dvdi</td>
<td>83</td>
</tr>
<tr>
<td>dvdiu</td>
<td>83</td>
</tr>
<tr>
<td>dvdiuz</td>
<td>83</td>
</tr>
<tr>
<td>dvdsi</td>
<td>83</td>
</tr>
<tr>
<td>dvdui</td>
<td>83</td>
</tr>
<tr>
<td>dvmdii</td>
<td>84</td>
</tr>
<tr>
<td>dvmdiiiz</td>
<td>84</td>
</tr>
<tr>
<td>dvmdis</td>
<td>84</td>
</tr>
<tr>
<td>dvmdsBIL</td>
<td>57</td>
</tr>
<tr>
<td>dvmdsi</td>
<td>84</td>
</tr>
<tr>
<td>dvmdss</td>
<td>84</td>
</tr>
<tr>
<td>dvmdubIL</td>
<td>57</td>
</tr>
<tr>
<td>dynamic array</td>
<td>195</td>
</tr>
<tr>
<td>d_ACKNOWLEDGE</td>
<td>205, 207</td>
</tr>
<tr>
<td>d_INITRC</td>
<td>205, 206</td>
</tr>
<tr>
<td>d_RETURN</td>
<td>205, 206, 207</td>
</tr>
<tr>
<td>d_SILENT</td>
<td>205</td>
</tr>
<tr>
<td>effective length</td>
<td>27</td>
</tr>
<tr>
<td>ei_multable</td>
<td>216</td>
</tr>
<tr>
<td>ell</td>
<td>239</td>
</tr>
<tr>
<td>ellanal_globalred</td>
<td>240</td>
</tr>
<tr>
<td>ellap</td>
<td>242</td>
</tr>
<tr>
<td>ellchangeinvert</td>
<td>242</td>
</tr>
<tr>
<td>elldatagenerators</td>
<td>240</td>
</tr>
<tr>
<td>ellff_get_a4a6</td>
<td>241</td>
</tr>
<tr>
<td>ellff_get_card</td>
<td>241</td>
</tr>
<tr>
<td>ellff_get_field</td>
<td>241</td>
</tr>
<tr>
<td>ellff_get_gens</td>
<td>241</td>
</tr>
<tr>
<td>ellff_get_group</td>
<td>241</td>
</tr>
<tr>
<td>ellff_get_o</td>
<td>241</td>
</tr>
<tr>
<td>ellff_get_p</td>
<td>241</td>
</tr>
<tr>
<td>ellinf</td>
<td>242</td>
</tr>
<tr>
<td>elllimit</td>
<td>239</td>
</tr>
<tr>
<td>ellQp_ab</td>
<td>241</td>
</tr>
<tr>
<td>ellQp_get_p</td>
<td>241</td>
</tr>
<tr>
<td>ellQp_get_prec</td>
<td>241</td>
</tr>
<tr>
<td>ellQp_get_zero</td>
<td>241</td>
</tr>
<tr>
<td>ellQp_q</td>
<td>241</td>
</tr>
<tr>
<td>ellQp_root</td>
<td>241</td>
</tr>
<tr>
<td>ellQp Tate_uniformization</td>
<td>241</td>
</tr>
<tr>
<td>ellQp_u</td>
<td>241</td>
</tr>
<tr>
<td>ellQp_u2</td>
<td>241</td>
</tr>
<tr>
<td>ellQ_get_N</td>
<td>240</td>
</tr>
<tr>
<td>ellQ_get_Nfa</td>
<td>240</td>
</tr>
<tr>
<td>ellrootno_global</td>
<td>240</td>
</tr>
<tr>
<td>ellR_ab</td>
<td>242</td>
</tr>
<tr>
<td>ellR_eta</td>
<td>242</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
</tr>
<tr>
<td>F2m_to_ZM</td>
<td>98</td>
</tr>
<tr>
<td>F2v_add_inplace</td>
<td>98</td>
</tr>
<tr>
<td>F2v_clear</td>
<td>97</td>
</tr>
<tr>
<td>F2v_coeff</td>
<td>97</td>
</tr>
<tr>
<td>F2v_copy</td>
<td>97</td>
</tr>
<tr>
<td>F2v_dotproduct</td>
<td>98</td>
</tr>
<tr>
<td>F2v_ei</td>
<td>98</td>
</tr>
<tr>
<td>F2v_flip</td>
<td>97</td>
</tr>
<tr>
<td>F2v_set</td>
<td>97</td>
</tr>
<tr>
<td>F2v_slice</td>
<td>97</td>
</tr>
<tr>
<td>F2v_to_F2x</td>
<td>116</td>
</tr>
<tr>
<td>F2xC_to_ZXC</td>
<td>127</td>
</tr>
<tr>
<td>F2xE_add</td>
<td>245</td>
</tr>
<tr>
<td>F2xE_changepoint</td>
<td>245</td>
</tr>
<tr>
<td>F2xE_changepointinv</td>
<td>245</td>
</tr>
<tr>
<td>F2xE_db1</td>
<td>245</td>
</tr>
<tr>
<td>F2xE_log</td>
<td>245</td>
</tr>
<tr>
<td>F2xE_mul</td>
<td>245</td>
</tr>
<tr>
<td>F2xE_neg</td>
<td>245</td>
</tr>
<tr>
<td>F2xE_order</td>
<td>245</td>
</tr>
<tr>
<td>F2xE_sub</td>
<td>245</td>
</tr>
<tr>
<td>F2xE_tatepairing</td>
<td>245</td>
</tr>
<tr>
<td>F2xE_weilpairing</td>
<td>245</td>
</tr>
<tr>
<td>F2xM_det</td>
<td>118</td>
</tr>
<tr>
<td>F2xM_F2xqC_mul</td>
<td>118</td>
</tr>
<tr>
<td>F2xM_image</td>
<td>118</td>
</tr>
<tr>
<td>F2xM_inv</td>
<td>118</td>
</tr>
<tr>
<td>F2xM_ker</td>
<td>118</td>
</tr>
<tr>
<td>F2xM_mul</td>
<td>118</td>
</tr>
<tr>
<td>F2xM_rank</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_Artin_Schreier</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_autpow</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_conjvec</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_div</td>
<td>117</td>
</tr>
<tr>
<td>F2xq_ellcard</td>
<td>245</td>
</tr>
<tr>
<td>F2xq_ellgens</td>
<td>245</td>
</tr>
<tr>
<td>F2xq_ellgroup</td>
<td>245</td>
</tr>
<tr>
<td>F2xq_inv</td>
<td>117</td>
</tr>
<tr>
<td>F2xq_invsafe</td>
<td>117</td>
</tr>
<tr>
<td>F2xq_log</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_matrix_pow</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_mul</td>
<td>117</td>
</tr>
<tr>
<td>F2xq_order</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_pow</td>
<td>117</td>
</tr>
<tr>
<td>F2xq_powers</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_powu</td>
<td>117</td>
</tr>
<tr>
<td>F2xq_sqr</td>
<td>117</td>
</tr>
<tr>
<td>F2xq_sqrt</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_sqrtin</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_sqrt_fast</td>
<td>118</td>
</tr>
<tr>
<td>F2xq_trace</td>
<td>118</td>
</tr>
<tr>
<td>F2xV_to_F2m</td>
<td>127</td>
</tr>
<tr>
<td>F2xV_to_FlxV_inplace</td>
<td>125</td>
</tr>
<tr>
<td>F2xV_to_ZXV_inplace</td>
<td>125</td>
</tr>
<tr>
<td>F2x_1_add</td>
<td>117</td>
</tr>
<tr>
<td>F2x_add</td>
<td>117</td>
</tr>
<tr>
<td>F2x_clear</td>
<td>116</td>
</tr>
<tr>
<td>F2x_coeff</td>
<td>116</td>
</tr>
<tr>
<td>F2x_deflate</td>
<td>117</td>
</tr>
<tr>
<td>F2x_degree</td>
<td>117</td>
</tr>
<tr>
<td>F2x_deriv</td>
<td>117</td>
</tr>
<tr>
<td>F2x_div</td>
<td>117</td>
</tr>
<tr>
<td>F2x_divrem</td>
<td>117</td>
</tr>
<tr>
<td>F2x_equal</td>
<td>117</td>
</tr>
<tr>
<td>F2x_equal1</td>
<td>117</td>
</tr>
<tr>
<td>F2x_even_odd</td>
<td>117</td>
</tr>
<tr>
<td>F2x_extgcd</td>
<td>117</td>
</tr>
<tr>
<td>F2xF2xqV_eval</td>
<td>118</td>
</tr>
<tr>
<td>F2xF2xq_eval</td>
<td>118</td>
</tr>
<tr>
<td>F2x_factor</td>
<td>117</td>
</tr>
<tr>
<td>F2x_factor</td>
<td>117</td>
</tr>
<tr>
<td>F2x_flip</td>
<td>116</td>
</tr>
<tr>
<td>F2x_gcd</td>
<td>117</td>
</tr>
<tr>
<td>F2x_halfgcd</td>
<td>117</td>
</tr>
<tr>
<td>F2x_issquare</td>
<td>117</td>
</tr>
<tr>
<td>F2x_is_irred</td>
<td>117</td>
</tr>
<tr>
<td>F2x_mul</td>
<td>117</td>
</tr>
<tr>
<td>F2x_rem</td>
<td>117</td>
</tr>
<tr>
<td>F2x_renormalize</td>
<td>117</td>
</tr>
<tr>
<td>F2x_set</td>
<td>116</td>
</tr>
<tr>
<td>F2x_shift</td>
<td>117</td>
</tr>
<tr>
<td>F2x_sqr</td>
<td>117</td>
</tr>
<tr>
<td>F2x_sqrt</td>
<td>117</td>
</tr>
<tr>
<td>F2x_to_F2v</td>
<td>127</td>
</tr>
<tr>
<td>F2x_to_Flx</td>
<td>116</td>
</tr>
<tr>
<td>F2x_to_ZX</td>
<td>116</td>
</tr>
<tr>
<td>F2x_valrem</td>
<td>117</td>
</tr>
<tr>
<td>factmod</td>
<td>119</td>
</tr>
<tr>
<td>factor</td>
<td>239</td>
</tr>
<tr>
<td>factorback</td>
<td>172</td>
</tr>
<tr>
<td>factoredpolred</td>
<td>229</td>
</tr>
<tr>
<td>factoredpolred2</td>
<td>229</td>
</tr>
<tr>
<td>factorial_lval</td>
<td>78</td>
</tr>
<tr>
<td>factorint</td>
<td>129</td>
</tr>
<tr>
<td>factorpadic0</td>
<td>229</td>
</tr>
<tr>
<td>factor_Aurifeuille</td>
<td>129</td>
</tr>
<tr>
<td>factor_Aurifeuille_prime</td>
<td>129</td>
</tr>
</tbody>
</table>
Fle_order .................................. 245
Fle_sub ................................... 244
Fim_center ................................. 95
Fim_charpoly ............................... 96
Fim_copy ................................... 95
Fim_deplin ................................ 96
Fim_det .................................... 96
Fim_det_sp ................................ 97
Fim_Flc_gauss .............................. 97
Fim_Flc_invimage ........................... 97
Fim_Flc_mul ................................ 95
Fim_Fl_mul .................................. 96
Fim_Fl_mul_inplace ......................... 96
Fim_gauss .................................. 97
Fim_hess ................................... 97
Fim_image .................................. 97
Fim_indexrank .............................. 97
Fim_inv .................................... 97
Fim_invimage ............................... 97
Fim_ker .................................... 97
Fim_ker_sp ................................ 97
Fim_mul .................................... 96
Fim_neg .................................... 96
Fim_powu .................................. 96
Fim_rank ................................... 97
Fim_suppl .................................. 97
Fim_to_F2m ................................ 98
Fim_to_FlxV ................................ 126
Fim_to_FlxX ................................ 126
Fim_to_mod ................................ 119
Fim_to_ZM .................................. 125
Fim_transpose ............................. 97
floor ....................................... 76
floor_safe .................................. 77
flush ...................................... 186
Flv_add .................................... 96
Flv_add_inplace ............................ 96, 221
Flv_center ................................ 95
Flv_copy ................................... 95
Flv_dotproduct ............................. 96
Flv_frobn .................................. 112
Flv_roots_to_pol ........................... 113
Flv_sub .................................... 96
Flv_sub_inplace ............................ 96
Flv_sum .................................... 96
Flv_to_F2 .................................. 98
Flv_to_Flx .................................. 126
Flv_to_ZV .................................. 125
<table>
<thead>
<tr>
<th>Flx_mod_Xmml</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flx_mul</td>
<td>110</td>
</tr>
<tr>
<td>Flx_mulu</td>
<td>111</td>
</tr>
<tr>
<td>Flx_nbfact</td>
<td>112</td>
</tr>
<tr>
<td>Flx_nbfact_by_degree</td>
<td>112</td>
</tr>
<tr>
<td>Flx_nbroots</td>
<td>112</td>
</tr>
<tr>
<td>Flx_neg</td>
<td>110</td>
</tr>
<tr>
<td>Flx_neg_inplace</td>
<td>110</td>
</tr>
<tr>
<td>Flx_normalize</td>
<td>111</td>
</tr>
<tr>
<td>Flx_oneroot</td>
<td>111</td>
</tr>
<tr>
<td>Flx_pow</td>
<td>111</td>
</tr>
<tr>
<td>Flx_recip</td>
<td>112</td>
</tr>
<tr>
<td>Flx_red</td>
<td>110</td>
</tr>
<tr>
<td>Flx_rem</td>
<td>111</td>
</tr>
<tr>
<td>Flx_renormalize</td>
<td>112</td>
</tr>
<tr>
<td>Flx_resultant</td>
<td>112</td>
</tr>
<tr>
<td>Flx_roots</td>
<td>111</td>
</tr>
<tr>
<td>Flx_roots_naive</td>
<td>111</td>
</tr>
<tr>
<td>Flx_shift</td>
<td>112</td>
</tr>
<tr>
<td>Flx_splitting</td>
<td>112</td>
</tr>
<tr>
<td>Flx_sqr</td>
<td>111</td>
</tr>
<tr>
<td>Flx_sub</td>
<td>110</td>
</tr>
<tr>
<td>Flx_to_F2x</td>
<td>116</td>
</tr>
<tr>
<td>Flx_to_Flv</td>
<td>126</td>
</tr>
<tr>
<td>Flx_to_FlxX</td>
<td>125</td>
</tr>
<tr>
<td>Flx_to_ZX</td>
<td>125</td>
</tr>
<tr>
<td>Flx_to_ZX_inplace</td>
<td>125</td>
</tr>
<tr>
<td>Flx_triple</td>
<td>111</td>
</tr>
<tr>
<td>Flx_val</td>
<td>112</td>
</tr>
<tr>
<td>Flx_valrem</td>
<td>112</td>
</tr>
<tr>
<td>Fly_to_FlxY</td>
<td>127</td>
</tr>
<tr>
<td>Fl_add</td>
<td>72</td>
</tr>
<tr>
<td>Fl_center</td>
<td>72</td>
</tr>
<tr>
<td>Fl_div</td>
<td>72</td>
</tr>
<tr>
<td>Fl_double</td>
<td>72</td>
</tr>
<tr>
<td>Fl_elltrace</td>
<td>244</td>
</tr>
<tr>
<td>Fl_inv</td>
<td>72</td>
</tr>
<tr>
<td>Fl_invsafe</td>
<td>72</td>
</tr>
<tr>
<td>Fl_mul</td>
<td>72</td>
</tr>
<tr>
<td>Fl_neg</td>
<td>72</td>
</tr>
<tr>
<td>Fl_order</td>
<td>72</td>
</tr>
<tr>
<td>Fl_powu</td>
<td>72</td>
</tr>
<tr>
<td>Fl_sqr</td>
<td>72</td>
</tr>
<tr>
<td>Fl_sqrt</td>
<td>72</td>
</tr>
<tr>
<td>Fl_sub</td>
<td>72</td>
</tr>
<tr>
<td>Fl_to_Flx</td>
<td>126</td>
</tr>
<tr>
<td>Fl_triple</td>
<td>72</td>
</tr>
<tr>
<td>forcomposite</td>
<td>40</td>
</tr>
<tr>
<td>forcomposite_init</td>
<td>40</td>
</tr>
</tbody>
</table>

| forcomposite_next | 40 |
| fordiv            | 40 |
| forell            | 40 |
| forell(ell,a,b,)  | 40 |
| format            | 38 |
| forpart           | 40 |
| forpart_init      | 40 |
| forpart_next      | 40 |
| forpart_prev      | 40 |
| forpart_t         | 40 |
| forprime          | 40 |
| forprime_init     | 41, 132 |
| forprime_next     | 41, 132 |
| forpart_init      | 40 |
| forvec            | 40 |
| forvec_next       | 40 |
| FpC_add           | 93 |
| FpC_center        | 93 |
| FpC_FpV_mul       | 94 |
| FpC_Fp_mul        | 94 |
| FpC_ratlift       | 121 |
| FpC_red           | 93 |
| FpC_sub           | 94 |
| FpC_to_mod        | 119 |
| FpE_add           | 244 |
| FpE_changepoint   | 244 |
| FpE_changepointinv | 244 |
| FpE dbl           | 244 |
| FpE_log           | 244 |
| FpE_mul           | 244 |
| FpE_neg           | 244 |
| FpE_order         | 244 |
| FpE_sub           | 244 |
| FpE_tatepairing   | 244 |
| FpE_to_mod        | 244 |
| FpM_deplin        | 94 |
| FpM_det           | 94 |
| FpM_FpCs_solve    | 140 |
| FpM_FpCs_solve_safe | 140 |
| FpM_FpC_mul       | 140 |
| FpM_leftkernel_elt | 140 |
| FpM_center        | 93 |
| FpM_charpoly      | 95 |
| FpM_FpC_gauss     | 94 |
| FpM_FpC_invimage  | 94 |

263
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FpX_sqrt</td>
<td>104</td>
<td>FpX_gcd</td>
<td>100</td>
</tr>
<tr>
<td>FpX_sqrtnt</td>
<td>104</td>
<td>FpX_get_red</td>
<td>102</td>
</tr>
<tr>
<td>FpX_sub</td>
<td>103</td>
<td>FpX_halfgcd</td>
<td>100</td>
</tr>
<tr>
<td>FpX_trace</td>
<td>105</td>
<td>FpX_invBarrett</td>
<td>101</td>
</tr>
<tr>
<td>FpXT_red</td>
<td>99</td>
<td>FpX_is_irred</td>
<td>102, 117</td>
</tr>
<tr>
<td>FpXV_FpC_mul</td>
<td>101</td>
<td>FpX_is_squarefree</td>
<td>101</td>
</tr>
<tr>
<td>FpXV_prod</td>
<td>101</td>
<td>FpX_is_totally_split</td>
<td>102</td>
</tr>
<tr>
<td>FpXV_red</td>
<td>99</td>
<td>FpX_mul</td>
<td>100</td>
</tr>
<tr>
<td>FpXX_add</td>
<td>106</td>
<td>FpX_mulspec</td>
<td>100</td>
</tr>
<tr>
<td>FpXX_FpX_mul</td>
<td>106</td>
<td>FpX_mul</td>
<td>101</td>
</tr>
<tr>
<td>FpXX_Fp_mul</td>
<td>106</td>
<td>FpX_nbfact</td>
<td>102</td>
</tr>
<tr>
<td>FpXX_mul</td>
<td>106</td>
<td>FpX_nbroots</td>
<td>102</td>
</tr>
<tr>
<td>FpXX_neg</td>
<td>106</td>
<td>FpX_neg</td>
<td>100</td>
</tr>
<tr>
<td>FpXX_red</td>
<td>106</td>
<td>FpX_normalize</td>
<td>101</td>
</tr>
<tr>
<td>FpXX_renormalize</td>
<td>106</td>
<td>FpX_oneroot</td>
<td>102</td>
</tr>
<tr>
<td>FpXX_sub</td>
<td>106</td>
<td>FpX_ratlift</td>
<td>121</td>
</tr>
<tr>
<td>FpXYQ_pow</td>
<td>106</td>
<td>FpX_red</td>
<td>99</td>
</tr>
<tr>
<td>FpXY_eval</td>
<td>106</td>
<td>FpX_rem</td>
<td>100</td>
</tr>
<tr>
<td>FpXY_evalx</td>
<td>106</td>
<td>FpX_renormalize</td>
<td>100</td>
</tr>
<tr>
<td>FpXY_evaly</td>
<td>106</td>
<td>FpX_rescale</td>
<td>101</td>
</tr>
<tr>
<td>FpXY_FpX_evalix</td>
<td>106</td>
<td>FpX_resultant</td>
<td>102</td>
</tr>
<tr>
<td>FpXY_FQ_evaly</td>
<td>106</td>
<td>FpX_roots</td>
<td>102</td>
</tr>
<tr>
<td>FpX_add</td>
<td>100</td>
<td>FpX_rootsff</td>
<td>109</td>
</tr>
<tr>
<td>FpX_center</td>
<td>101</td>
<td>FpX_sqr</td>
<td>100</td>
</tr>
<tr>
<td>FpX_chinese_copime</td>
<td>101</td>
<td>FpX_sub</td>
<td>100</td>
</tr>
<tr>
<td>FpX_degfact</td>
<td>102, 111</td>
<td>FpX_to_mod</td>
<td>119</td>
</tr>
<tr>
<td>FpX_deriv</td>
<td>100</td>
<td>FpX_translate</td>
<td>100</td>
</tr>
<tr>
<td>FpX_disc</td>
<td>102</td>
<td>FpX_valrem</td>
<td>100</td>
</tr>
<tr>
<td>FpX_div</td>
<td>100</td>
<td>Fp_add</td>
<td>13, 87</td>
</tr>
<tr>
<td>FpX_divrem</td>
<td>100</td>
<td>Fp_adddmul</td>
<td>87</td>
</tr>
<tr>
<td>FpX_div_by_X_x</td>
<td>100</td>
<td>Fp_center</td>
<td>87</td>
</tr>
<tr>
<td>FpX_eval</td>
<td>101</td>
<td>Fp_div</td>
<td>87</td>
</tr>
<tr>
<td>FpX_extgcd</td>
<td>100</td>
<td>Fp_ellcard</td>
<td>243</td>
</tr>
<tr>
<td>FpX_factor</td>
<td>102</td>
<td>Fp_ellcard_SEA</td>
<td>243</td>
</tr>
<tr>
<td>FpX_factorff</td>
<td>109</td>
<td>Fp_elldivpol</td>
<td>244</td>
</tr>
<tr>
<td>FpX_factorff_irred</td>
<td>109, 111</td>
<td>Fp_ellgens</td>
<td>243</td>
</tr>
<tr>
<td>FpX_ffintersect</td>
<td>109</td>
<td>Fp_ellgroup</td>
<td>243</td>
</tr>
<tr>
<td>FpX_ffisom</td>
<td>109, 111</td>
<td>Fp_ellj</td>
<td>243</td>
</tr>
<tr>
<td>FpX_ffisom_0p</td>
<td>214</td>
<td>Fp_factored_order</td>
<td>88</td>
</tr>
<tr>
<td>FpX_FpC_npoleval</td>
<td>105</td>
<td>Fp_ffellcard</td>
<td>243</td>
</tr>
<tr>
<td>FpX_FpXQV_eval</td>
<td>105</td>
<td>Fp_FpXQV_log</td>
<td>104</td>
</tr>
<tr>
<td>FpX_FpXQ_eval</td>
<td>105</td>
<td>Fp_FpX_sub</td>
<td>101</td>
</tr>
<tr>
<td>FpX_FpXY_resultant</td>
<td>102</td>
<td>Fp_inv</td>
<td>87</td>
</tr>
<tr>
<td>FpX_Fp_add</td>
<td>101</td>
<td>Fp_invsafe</td>
<td>87</td>
</tr>
<tr>
<td>FpX_Fp_add_shallow</td>
<td>101</td>
<td>Fp_ispower</td>
<td>88</td>
</tr>
<tr>
<td>FpX_Fp_mul</td>
<td>101</td>
<td>Fp_ispower</td>
<td>88</td>
</tr>
<tr>
<td>FpX_Fp_mulspec</td>
<td>101</td>
<td>Fp_isquare</td>
<td>88</td>
</tr>
<tr>
<td>FpX_Fp_mul_to_monic</td>
<td>101</td>
<td>Fp_log</td>
<td>87</td>
</tr>
<tr>
<td>FpX_Fp_sub</td>
<td>101</td>
<td>Fp_mul</td>
<td>87</td>
</tr>
<tr>
<td>FpX_Fp_sub_shallow</td>
<td>101</td>
<td>Fp_muls</td>
<td>87</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_mulu</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_neg</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_order</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_pow</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_pows</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_powu</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_ratlift</td>
<td>121</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_red</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_sqr</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_sqrt</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_sqrtn</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_sub</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fp_to_mod</td>
<td>119</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqC_add</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqC_Fq_mul</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqC_sub</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqC_to_FlxC</td>
<td>125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_deplin</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_det</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_FqC_gauss</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_FqC_mul</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_gauss</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_image</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_inv</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_ker</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_mulu</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_rank</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_suppl</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_to_FlxM</td>
<td>125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqM_to_nfM</td>
<td>220</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqV_inv</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqV_red</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqV_roots_to_pol</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqV_to_FlxV</td>
<td>125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqV_to_nfV</td>
<td>220</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXQ_add</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXQ_div</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXQ_inv</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXQ_invsafe</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXQ_matrix_pow</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXQ_mulu</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXQ_pow</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXQ_powers</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXQ_sqr</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXQ_sub</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXY_eval</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqXY_evalx</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_add</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_deriv</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_div</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_divrem</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_eval</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_extgcd</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_factor</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_Fq_add</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_Fq_mul</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_Fq_mul_to_monic</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_gcd</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_is_squarefree</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_mul</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_mulu</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_nbact</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_nbfact</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_nbroots</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_normalize</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_red</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_roots</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_sqr</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_sub</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_to_nfX</td>
<td>221</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FqX_translate</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_add</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_div</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_ellcard_SEA</td>
<td>243</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_elldivpolmod</td>
<td>247</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_Fp_mul</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_inv</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_invsafe</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_isquare</td>
<td>105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_mulu</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_mul</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_neg</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_neg_inv</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_pow</td>
<td>105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_powu</td>
<td>105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_red</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_sqr</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_sqrtn</td>
<td>105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_sub</td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fq_to_nf</td>
<td>220</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frobeniusform</td>
<td>139</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fun(E, ell)</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fun(E, H)</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>functions_basic</td>
<td>49</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

266
functions_default .................................. 49
functions_gp ...................................... 49
functions_gp_default ................................ 49
functions_gp_rl_default ................................ 49
functions_highlevel .................................. 49
fPRETTYMAT ........................................... 185
f_RAW .................................................. 185, 186
f_TEX .................................................. 185, 186

G

gabs[z] ............................................... 170
gadd .................................................. 74, 170
gaddgs ............................................... 13, 170
gaddg ............................................... 13, 170
gaddz ............................................... 13, 24, 74, 171
gaddz[z] ............................................... 74
gaffect .............................................. 24, 25, 153
gaffsg ............................................... 25, 153
galoisexport ....................................... 184
galoisidentify ................................... 184
galoisinit ......................................... 183, 230
galois_group ..................................... 183
gal_get_den ....................................... 230
gal_get_e ........................................... 230
gal_get_gen ........................................ 230
gal_get_group ...................................... 230
gal_get_invvdm ................................... 230
gal_get_mod ........................................ 230
gal_get_orders .................................... 230
gal_get_p .......................................... 230
gal_get_pol ........................................ 230
gal_get_roots ..................................... 230
gand .................................................. 165
garbage collecting .................................. 15
gassoc_proto ...................................... 89
gaussred_from_QR ................................ 140
gbezout ............................................. 169
gcdii ............................................... 86
gceil ............................................... 162
gclone ............................................. 25, 62, 63
gcloneref ......................................... 63
gclone_refc ....................................... 64
gcmp ............................................... 163
gcmgs ............................................. 164
gcmpsg ............................................ 164
gcoeff ............................................. 13, 56, 197
 gcop ............................................... 25, 62
gcopy ............................................... 62
gcopy_avma ....................................... 62
gcopy_lg .......................................... 63
gcvtoi ............................................. 162
gcvtop ............................................. 154
gdeuc ............................................. 168
gdiv ............................................... 170
gdiventgsz ....................................... 168
gdiventres ....................................... 167
gdiventsg ....................................... 168
gdiventz .......................................... 168
gdivexact ......................................... 167
gdivgs ............................................ 170
gdivmod ........................................... 168
gdivround ......................................... 168
gdivsg ............................................ 170
gdivz ............................................. 171
gvd ............................................... 167
gel .................................................. 12, 13, 56, 197
GEN ............................................... 11
GENbinbase ...................................... 60
gener_F2xq ....................................... 118
gener_Flxq ....................................... 114
gener_FPQ ......................................... 105
gener_FPQXQ_local ................................ 105
GENtoGENstr .................................... 185
GENtoGENstr_nospace ................................ 185
GENtostr ........................................... 37, 185
GENtostr_unquoted ................................ 185
GENtoTeXstr ....................................... 37, 185
gen_0 ............................................... 11, 31
gen_1 ............................................... 11
gen_2 ............................................... 11
gen_cmp_RgX ....................................... 167
gen_factorback .................................... 172
gen_FPQWiedemann ................................. 141
gen_indexsort ..................................... 166
gen_indexsort_uniq ................................ 166
gen_m1 ............................................. 11
gen_m2 ............................................. 11
gen_pow ........................................... 172
gen_powers ........................................ 172
gen_powu .......................................... 172
gen_powu_fold .................................... 172
gen_powu_fold_i .................................. 172
gen_powu_i ........................................ 172
gen_pow_fold ..................................... 172
gen_pow_fold_i .................................. 172
gen_pow_i .......................................... 172
gen_search ......................................... 167
gen_setminus ...................................... 166
gp_context_save ..........................  50
gp_eval  ..................................  205
gp_evalbool  ...............................  205
gp_evalupto ................................ 205
gp_evalvoid ................................ 205
gp_read_file ................................ 36
gp_read_str ................................ 35
Gram matrix ................................ 138
gram_matrix ................................ 138
gred_rfac2 ..................................  32
grem ......................................... 168
grndtoi ...................................... 162
ground ...................................... 162
groupelts_abelian_group .....................  84
groupelts_center .........................  84
groupelts_set ...............................  83
group_abelianHNF .........................  83
group_abelianSNF .........................  83
group_domain ...............................  83
group_elts ................................  83
group_export ...............................  83
group_ident .................................  83
group_ident_trans ..........................  83
group_isA4S4 ................................ 84
group_isabelian ................................ 83
group_leftcoset .............................  83
group_order ................................  83
group_perm_normalize ......................  83
group_quotient ..............................  83
group_rightcoset ...........................  83
group_set ..................................  83
group_subgroups .............................. 83
group_subgroup_isnormal ...................  83
gshift[x] ..................................  62
gsigne ......................................  29, 52
gsincos .....................................  81
gsizebyte ...................................  24
gsizeword ...................................  24
gsmith .......................................  236
gsmithall ...................................  236
gsprintf ....................................  186
gsqr .........................................  170
GSTR .......................................  170
gsub .........................................  170
gsubbs ......................................  170
gsubb ......................................  173
gsubz .......................................  171
gsupnorm .................................... 173
gsupnorm_aux .........................  173
gtocol .......................................  156
gtodouble .................................  26, 153
gtodecimal ..................................  25, 153
gtolong ....................................  26, 153
gtomat .......................................  156
gtomp .......................................  154
gtopoly ......................................  154
gtopolyrev ..................................  155
gtos .........................................  153
gtoser .......................................  156
gtovect .....................................  156
gtovcsmall .................................  156
gtrans ........................................ 197
gtrunc ........................................ 162
gtrunc2n ....................................  77, 162
gunclone .....................................  25, 63
gunclone_deep ................................  63
gval .........................................  162
gvaluation ...................................  162
gvar .........................................  30, 33, 53
gvar2 ........................................  53
gvprintf .....................................  186

H

hashentry ...................................  194
hashtable ...................................  194
hash_create ................................  195
hash_destroy ...............................  195
hash_GEN ...................................  195
hash_insert ................................  195
hash_remove ................................. 195
hash_search ................................  195
hash_str ....................................  195
hash_str2 ...................................  195
heap .........................................  12
hexadecimal tree .........................  38
HIGHBIT ...................................  56
HIGHEXPOBIT ................................  57
HIGHMASK ...................................  56
HIGHVALBIT ................................  57
HIGHWORD ..................................  56
hilbertii ...................................  90
hnf .........................................  236
hnfcall ....................................  236
hnfdivide ...................................  235
hnflll .....................................  236

269
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkvecs</td>
<td>158</td>
</tr>
<tr>
<td>mkvecsmall</td>
<td>158</td>
</tr>
<tr>
<td>mkvecsmall2</td>
<td>158</td>
</tr>
<tr>
<td>mkvecsmall3</td>
<td>158</td>
</tr>
<tr>
<td>mkvecsmall4</td>
<td>158</td>
</tr>
<tr>
<td>mkvecsmallln</td>
<td>158</td>
</tr>
<tr>
<td>mod16</td>
<td>28</td>
</tr>
<tr>
<td>mod2</td>
<td>28</td>
</tr>
<tr>
<td>mod2BIL</td>
<td>28</td>
</tr>
<tr>
<td>mod32</td>
<td>28</td>
</tr>
<tr>
<td>mod4</td>
<td>28</td>
</tr>
<tr>
<td>mod64</td>
<td>28</td>
</tr>
<tr>
<td>mod8</td>
<td>28</td>
</tr>
<tr>
<td>modpr_genFq</td>
<td>220</td>
</tr>
<tr>
<td>modreverse</td>
<td>150</td>
</tr>
<tr>
<td>moebiusu</td>
<td>89</td>
</tr>
<tr>
<td>mpabs</td>
<td>79</td>
</tr>
<tr>
<td>mpabs_shallow</td>
<td>79</td>
</tr>
<tr>
<td>mpadd</td>
<td>13</td>
</tr>
<tr>
<td>mpaff</td>
<td>75</td>
</tr>
<tr>
<td>mpbern</td>
<td>181</td>
</tr>
<tr>
<td>mpceil</td>
<td>76</td>
</tr>
<tr>
<td>mpcmp</td>
<td>80</td>
</tr>
<tr>
<td>mpcopy</td>
<td>75</td>
</tr>
<tr>
<td>mpcos[z]</td>
<td>180</td>
</tr>
<tr>
<td>mpeint1</td>
<td>180</td>
</tr>
<tr>
<td>mpeuler</td>
<td>182</td>
</tr>
<tr>
<td>mpxpm1</td>
<td>180</td>
</tr>
<tr>
<td>mpxpo</td>
<td>53</td>
</tr>
<tr>
<td>mpxp[z]</td>
<td>180</td>
</tr>
<tr>
<td>mpfloor</td>
<td>76</td>
</tr>
<tr>
<td>mplambertW</td>
<td>180</td>
</tr>
<tr>
<td>mplog2</td>
<td>182</td>
</tr>
<tr>
<td>mplog[z]</td>
<td>180</td>
</tr>
<tr>
<td>mpneg</td>
<td>79</td>
</tr>
<tr>
<td>mpodd</td>
<td>28</td>
</tr>
<tr>
<td>mmpi</td>
<td>182</td>
</tr>
<tr>
<td>mpround</td>
<td>76</td>
</tr>
<tr>
<td>mpsin</td>
<td>77</td>
</tr>
<tr>
<td>mpsinbni</td>
<td>180</td>
</tr>
<tr>
<td>mpsinbni1</td>
<td>180</td>
</tr>
<tr>
<td>mpsinbni2</td>
<td>180</td>
</tr>
<tr>
<td>mpsinbni3</td>
<td>180</td>
</tr>
<tr>
<td>mpsgr</td>
<td>79</td>
</tr>
<tr>
<td>mptrunc</td>
<td>76</td>
</tr>
<tr>
<td>mpveceint1</td>
<td>180</td>
</tr>
<tr>
<td>msgtimer</td>
<td>39</td>
</tr>
<tr>
<td>mulfci</td>
<td>158,159</td>
</tr>
<tr>
<td>mulfcm1</td>
<td>159</td>
</tr>
<tr>
<td>muli</td>
<td>82</td>
</tr>
<tr>
<td>mul1</td>
<td>71</td>
</tr>
<tr>
<td>mulreal</td>
<td>175</td>
</tr>
<tr>
<td>mulsubii</td>
<td>82</td>
</tr>
<tr>
<td>multable</td>
<td>216</td>
</tr>
<tr>
<td>mului</td>
<td>82</td>
</tr>
<tr>
<td>muluu</td>
<td>82</td>
</tr>
<tr>
<td>muluui</td>
<td>82</td>
</tr>
<tr>
<td>mulu_interval</td>
<td>82</td>
</tr>
<tr>
<td>mul_content</td>
<td>170</td>
</tr>
<tr>
<td>mul_denom</td>
<td>170</td>
</tr>
<tr>
<td>name_numerr</td>
<td>194</td>
</tr>
<tr>
<td>name_var</td>
<td>34,65</td>
</tr>
<tr>
<td>nbits2nlong</td>
<td>52</td>
</tr>
<tr>
<td>nbits2prec</td>
<td>52</td>
</tr>
<tr>
<td>nbrows</td>
<td>53</td>
</tr>
<tr>
<td>nchar2nlong</td>
<td>52</td>
</tr>
<tr>
<td>ndec2nlong</td>
<td>52</td>
</tr>
<tr>
<td>ndec2prec</td>
<td>52</td>
</tr>
<tr>
<td>negi</td>
<td>79</td>
</tr>
<tr>
<td>negr</td>
<td>79</td>
</tr>
<tr>
<td>newblock</td>
<td>63</td>
</tr>
<tr>
<td>new_chunk</td>
<td>59</td>
</tr>
<tr>
<td>NEXT_PRIME_VIADIFF</td>
<td>58</td>
</tr>
<tr>
<td>NEXT_PRIME_VIADIFF_CHECK</td>
<td>58</td>
</tr>
<tr>
<td>nfadd</td>
<td>214</td>
</tr>
<tr>
<td>nfalgtobasis</td>
<td>215</td>
</tr>
<tr>
<td>nfarchstar</td>
<td>222</td>
</tr>
<tr>
<td>nfbasis</td>
<td>223</td>
</tr>
<tr>
<td>nfbasis0</td>
<td>229</td>
</tr>
<tr>
<td>nfbasistoalg</td>
<td>215</td>
</tr>
<tr>
<td>nC_nf_mul</td>
<td>215</td>
</tr>
<tr>
<td>nfdisc0</td>
<td>229</td>
</tr>
<tr>
<td>nfdiv</td>
<td>214</td>
</tr>
<tr>
<td>nfdivfz</td>
<td>214</td>
</tr>
<tr>
<td>nfdivrem</td>
<td>214</td>
</tr>
<tr>
<td>nfeltup</td>
<td>228</td>
</tr>
<tr>
<td>nffactorback</td>
<td>217</td>
</tr>
<tr>
<td>nfgaloisconj</td>
<td>229</td>
</tr>
<tr>
<td>nfgaloismatrix</td>
<td>230</td>
</tr>
<tr>
<td>nfgcd</td>
<td>145</td>
</tr>
<tr>
<td>nfgcd_all</td>
<td>145</td>
</tr>
<tr>
<td>nfenv</td>
<td>214</td>
</tr>
<tr>
<td>nfinvmoduleal</td>
<td>215</td>
</tr>
<tr>
<td>nfinvmodideal</td>
<td>222</td>
</tr>
<tr>
<td>nmaxord</td>
<td>222</td>
</tr>
<tr>
<td>nmaxord_t</td>
<td>222</td>
</tr>
<tr>
<td>nfmod</td>
<td>215</td>
</tr>
<tr>
<td>random_bits</td>
<td>86</td>
</tr>
<tr>
<td>---------------</td>
<td>-----</td>
</tr>
<tr>
<td>random_F2x</td>
<td>117</td>
</tr>
<tr>
<td>random_F2xE</td>
<td>245</td>
</tr>
<tr>
<td>random_Fl</td>
<td>72, 86</td>
</tr>
<tr>
<td>random_Fle</td>
<td>245</td>
</tr>
<tr>
<td>random_FlX</td>
<td>111</td>
</tr>
<tr>
<td>random_FlxE</td>
<td>246</td>
</tr>
<tr>
<td>random_FpE</td>
<td>244</td>
</tr>
<tr>
<td>random_FpX</td>
<td>102</td>
</tr>
<tr>
<td>random_FpXQE</td>
<td>247</td>
</tr>
<tr>
<td>rational function</td>
<td>32</td>
</tr>
<tr>
<td>rational number</td>
<td>29</td>
</tr>
<tr>
<td>raw</td>
<td>185</td>
</tr>
<tr>
<td>rcopy</td>
<td>75</td>
</tr>
<tr>
<td>rdivii</td>
<td>84</td>
</tr>
<tr>
<td>rdiviiiz</td>
<td>84</td>
</tr>
<tr>
<td>rdivis</td>
<td>84</td>
</tr>
<tr>
<td>rdivss</td>
<td>84</td>
</tr>
<tr>
<td>read</td>
<td>36</td>
</tr>
<tr>
<td>readseq</td>
<td>35</td>
</tr>
<tr>
<td>real number</td>
<td>29</td>
</tr>
<tr>
<td>real</td>
<td>175</td>
</tr>
<tr>
<td>real2n</td>
<td>74</td>
</tr>
<tr>
<td>real_0</td>
<td>74</td>
</tr>
<tr>
<td>real_0_bit</td>
<td>74</td>
</tr>
<tr>
<td>real_1</td>
<td>74</td>
</tr>
<tr>
<td>real_i</td>
<td>175</td>
</tr>
<tr>
<td>real_m1</td>
<td>74</td>
</tr>
<tr>
<td>real_m2n</td>
<td>74</td>
</tr>
<tr>
<td>reducemodinv</td>
<td>238</td>
</tr>
<tr>
<td>reducemodinv11</td>
<td>238</td>
</tr>
<tr>
<td>remi2n</td>
<td>83, 142</td>
</tr>
<tr>
<td>remBIL</td>
<td>57</td>
</tr>
<tr>
<td>residual_characteristic</td>
<td>176</td>
</tr>
<tr>
<td>resultant (reduced)</td>
<td>123</td>
</tr>
<tr>
<td>resultant</td>
<td>168, 176</td>
</tr>
<tr>
<td>resultant2</td>
<td>176</td>
</tr>
<tr>
<td>resultant_all</td>
<td>176</td>
</tr>
<tr>
<td>retconst_col</td>
<td>160</td>
</tr>
<tr>
<td>retconst_vec</td>
<td>160</td>
</tr>
<tr>
<td>retmkcol</td>
<td>160</td>
</tr>
<tr>
<td>retmkcol2</td>
<td>160</td>
</tr>
<tr>
<td>retmkcol3</td>
<td>160</td>
</tr>
<tr>
<td>retmkcol14</td>
<td>160</td>
</tr>
<tr>
<td>retmkcol15</td>
<td>160</td>
</tr>
<tr>
<td>retmkcomplex</td>
<td>160</td>
</tr>
<tr>
<td>retmkfrac</td>
<td>160</td>
</tr>
<tr>
<td>retmkintmod</td>
<td>160</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
</tr>
<tr>
<td>sd_ulong</td>
<td>207</td>
</tr>
<tr>
<td>secure</td>
<td>49</td>
</tr>
<tr>
<td>ser_chop0</td>
<td>156</td>
</tr>
<tr>
<td>ser_normalize</td>
<td>177</td>
</tr>
<tr>
<td>ser_unscale</td>
<td>177</td>
</tr>
<tr>
<td>setabssign</td>
<td>54</td>
</tr>
<tr>
<td>setdefault</td>
<td>49, 205</td>
</tr>
<tr>
<td>setexpo</td>
<td>29, 31, 55</td>
</tr>
<tr>
<td>setisclone</td>
<td>26</td>
</tr>
<tr>
<td>setlg</td>
<td>26, 54</td>
</tr>
<tr>
<td>setigefint</td>
<td>27, 54</td>
</tr>
<tr>
<td>setprec</td>
<td>30, 55</td>
</tr>
<tr>
<td>setrealprecision</td>
<td>179</td>
</tr>
<tr>
<td>setsign</td>
<td>27, 30, 31, 54</td>
</tr>
<tr>
<td>settype</td>
<td>26, 54</td>
</tr>
<tr>
<td>setvalp</td>
<td>30, 31, 55</td>
</tr>
<tr>
<td>setvarn</td>
<td>23, 30, 31, 55, 161</td>
</tr>
<tr>
<td>set_lex</td>
<td>204</td>
</tr>
<tr>
<td>set_sign_mod_divisor</td>
<td>222</td>
</tr>
<tr>
<td>shallow</td>
<td>47</td>
</tr>
<tr>
<td>shallowconcat</td>
<td>197, 198</td>
</tr>
<tr>
<td>shallowconcat1</td>
<td>197</td>
</tr>
<tr>
<td>shallowcopy</td>
<td>25, 197</td>
</tr>
<tr>
<td>shallowextract</td>
<td>198</td>
</tr>
<tr>
<td>shallowmatconcat</td>
<td>198</td>
</tr>
<tr>
<td>shallowtrans</td>
<td>197</td>
</tr>
<tr>
<td>shiftaddress</td>
<td>60</td>
</tr>
<tr>
<td>shiftaddress_canon</td>
<td>60</td>
</tr>
<tr>
<td>shifti</td>
<td>77</td>
</tr>
<tr>
<td>shiftl</td>
<td>72</td>
</tr>
<tr>
<td>shiftr</td>
<td>72</td>
</tr>
<tr>
<td>shiftr_inplace</td>
<td>77</td>
</tr>
<tr>
<td>shift_left</td>
<td>78</td>
</tr>
<tr>
<td>shift_right</td>
<td>78</td>
</tr>
<tr>
<td>SIGNBITS</td>
<td>57</td>
</tr>
<tr>
<td>signe</td>
<td>27, 30, 31, 52</td>
</tr>
<tr>
<td>SIGNnumBITS</td>
<td>57</td>
</tr>
<tr>
<td>SIMPLESHIFT</td>
<td>57</td>
</tr>
<tr>
<td>simplefactmod</td>
<td>119</td>
</tr>
<tr>
<td>simplify</td>
<td>63</td>
</tr>
<tr>
<td>simplify_shallow</td>
<td>63</td>
</tr>
<tr>
<td>sizedigit</td>
<td>53</td>
</tr>
<tr>
<td>smallpoired</td>
<td>229</td>
</tr>
<tr>
<td>smallpoired2</td>
<td>229</td>
</tr>
<tr>
<td>SMALL ULONG</td>
<td>73</td>
</tr>
<tr>
<td>smith</td>
<td>236</td>
</tr>
<tr>
<td>smithall</td>
<td>236</td>
</tr>
<tr>
<td>simplification_shallow</td>
<td>166, 167</td>
</tr>
<tr>
<td>simplify_factmod</td>
<td>139</td>
</tr>
<tr>
<td>sprintf</td>
<td>37</td>
</tr>
<tr>
<td>sqrfrac</td>
<td>175</td>
</tr>
<tr>
<td>sqri</td>
<td>79</td>
</tr>
<tr>
<td>sqrr</td>
<td>79</td>
</tr>
<tr>
<td>sqrs</td>
<td>79</td>
</tr>
<tr>
<td>sqrti</td>
<td>86</td>
</tr>
<tr>
<td>sqtr</td>
<td>86</td>
</tr>
<tr>
<td>sqtrr</td>
<td>180</td>
</tr>
<tr>
<td>sqtrt</td>
<td>180</td>
</tr>
<tr>
<td>sqtremi</td>
<td>86</td>
</tr>
<tr>
<td>sqtr_abs</td>
<td>180</td>
</tr>
<tr>
<td>sqru</td>
<td>79</td>
</tr>
<tr>
<td>stack</td>
<td>11, 15</td>
</tr>
<tr>
<td>stackdefault</td>
<td>61, 75</td>
</tr>
<tr>
<td>stackdummy</td>
<td>59</td>
</tr>
<tr>
<td>stack_callalloc</td>
<td>21</td>
</tr>
<tr>
<td>stack_limit</td>
<td>59, 185</td>
</tr>
<tr>
<td>stack_malloc</td>
<td>185</td>
</tr>
<tr>
<td>stack_sprintf</td>
<td>185</td>
</tr>
<tr>
<td>stack_strcat</td>
<td>185</td>
</tr>
<tr>
<td>stack_strdup</td>
<td>185</td>
</tr>
<tr>
<td>stderr</td>
<td>187</td>
</tr>
<tr>
<td>stdout</td>
<td>36, 187</td>
</tr>
<tr>
<td>stoi</td>
<td>25, 76</td>
</tr>
<tr>
<td>stor</td>
<td>76</td>
</tr>
<tr>
<td>Str</td>
<td>185</td>
</tr>
<tr>
<td>Strexpand</td>
<td>186</td>
</tr>
<tr>
<td>strftime_expanded</td>
<td>188</td>
</tr>
<tr>
<td>string_context</td>
<td>67</td>
</tr>
<tr>
<td>strntoGENstr</td>
<td>185</td>
</tr>
<tr>
<td>Strprintf</td>
<td>37</td>
</tr>
<tr>
<td>Strtexp</td>
<td>185</td>
</tr>
<tr>
<td>strtfclosure</td>
<td>204</td>
</tr>
<tr>
<td>strtoclosure</td>
<td>204</td>
</tr>
<tr>
<td>subiu</td>
<td>82</td>
</tr>
<tr>
<td>subll</td>
<td>71</td>
</tr>
<tr>
<td>subllx</td>
<td>71</td>
</tr>
<tr>
<td>submulii</td>
<td>82</td>
</tr>
<tr>
<td>submuliu</td>
<td>82</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>submuliu_inplace</td>
<td>82</td>
</tr>
<tr>
<td>subresext</td>
<td>169</td>
</tr>
<tr>
<td>subui</td>
<td>82</td>
</tr>
<tr>
<td>subuq</td>
<td>82</td>
</tr>
<tr>
<td>sumdedekind</td>
<td>90</td>
</tr>
<tr>
<td>sumdedekind_coprime</td>
<td>90</td>
</tr>
<tr>
<td>sumdigitsu</td>
<td>90</td>
</tr>
<tr>
<td>switchout</td>
<td>187</td>
</tr>
<tr>
<td>szeta</td>
<td>180</td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>tablemul</td>
<td>216</td>
</tr>
<tr>
<td>tablemulvec</td>
<td>216</td>
</tr>
<tr>
<td>tablemul_ei</td>
<td>216</td>
</tr>
<tr>
<td>tablemul_ei_ej</td>
<td>216</td>
</tr>
<tr>
<td>tablesearch</td>
<td>167</td>
</tr>
<tr>
<td>tablesset_find_index</td>
<td>184</td>
</tr>
<tr>
<td>tablesqr</td>
<td>216</td>
</tr>
<tr>
<td>term_color</td>
<td>187</td>
</tr>
<tr>
<td>term_get_color</td>
<td>187</td>
</tr>
<tr>
<td>texe</td>
<td>188</td>
</tr>
<tr>
<td>threads</td>
<td>251</td>
</tr>
<tr>
<td>timer</td>
<td>39</td>
</tr>
<tr>
<td>timer2</td>
<td>39</td>
</tr>
<tr>
<td>timer_delay</td>
<td>39</td>
</tr>
<tr>
<td>timer_get</td>
<td>39</td>
</tr>
<tr>
<td>timer_printf</td>
<td>39</td>
</tr>
<tr>
<td>timer_start</td>
<td>39</td>
</tr>
<tr>
<td>togglesign</td>
<td>54</td>
</tr>
<tr>
<td>togglesign_safe</td>
<td>54, 133, 134</td>
</tr>
<tr>
<td>to_famat</td>
<td>217</td>
</tr>
<tr>
<td>to_famat_shallow</td>
<td>217</td>
</tr>
<tr>
<td>trans_eval</td>
<td>181</td>
</tr>
<tr>
<td>traverseheap</td>
<td>63</td>
</tr>
<tr>
<td>tridiv_bound</td>
<td>129</td>
</tr>
<tr>
<td>trivialgroup</td>
<td>183</td>
</tr>
<tr>
<td>trivial_fact</td>
<td>157</td>
</tr>
<tr>
<td>truecoeff</td>
<td>55, 175, 176</td>
</tr>
<tr>
<td>truedivii</td>
<td>83</td>
</tr>
<tr>
<td>truedivis</td>
<td>83</td>
</tr>
<tr>
<td>truedivsi</td>
<td>83</td>
</tr>
<tr>
<td>truedvmdii</td>
<td>85</td>
</tr>
<tr>
<td>truedvmdis</td>
<td>85</td>
</tr>
<tr>
<td>truedvmsdi</td>
<td>85</td>
</tr>
<tr>
<td>trunc2nr</td>
<td>77</td>
</tr>
<tr>
<td>trunc2nr_lg</td>
<td>77</td>
</tr>
<tr>
<td>truncr</td>
<td>76</td>
</tr>
<tr>
<td>trunc_safe</td>
<td>77</td>
</tr>
<tr>
<td>twopotbits_in_long</td>
<td>56</td>
</tr>
<tr>
<td>typ</td>
<td>26, 52</td>
</tr>
<tr>
<td>typbits</td>
<td>57</td>
</tr>
<tr>
<td>type number</td>
<td>26</td>
</tr>
<tr>
<td>type</td>
<td>12</td>
</tr>
<tr>
<td>type_name</td>
<td>52</td>
</tr>
<tr>
<td>typnumBITS</td>
<td>57</td>
</tr>
<tr>
<td>typ_SHIFT</td>
<td>57</td>
</tr>
<tr>
<td>typ_BNF</td>
<td>209</td>
</tr>
<tr>
<td>typ_BNR</td>
<td>209</td>
</tr>
<tr>
<td>typ_NF</td>
<td>209</td>
</tr>
<tr>
<td>_CLOSEURE</td>
<td>32</td>
</tr>
<tr>
<td>_COL</td>
<td>32</td>
</tr>
<tr>
<td>_COMPLEX</td>
<td>29</td>
</tr>
<tr>
<td>_ELL_Fp</td>
<td>239</td>
</tr>
<tr>
<td>_ELL_Fq</td>
<td>239, 242</td>
</tr>
<tr>
<td>_ELL_Q</td>
<td>239</td>
</tr>
<tr>
<td>_ELL_Qp</td>
<td>239</td>
</tr>
<tr>
<td>_ELL_Rg</td>
<td>239</td>
</tr>
<tr>
<td>_ERROR</td>
<td>32</td>
</tr>
<tr>
<td>_FFELT</td>
<td>29</td>
</tr>
<tr>
<td>_FF_F2xq</td>
<td>29</td>
</tr>
<tr>
<td>_FF_F1xq</td>
<td>29</td>
</tr>
<tr>
<td>_FF_FpXq</td>
<td>29</td>
</tr>
<tr>
<td>_FRAC</td>
<td>29</td>
</tr>
<tr>
<td>_INT</td>
<td>27</td>
</tr>
<tr>
<td>_INTMOD</td>
<td>29</td>
</tr>
<tr>
<td>_LIST</td>
<td>33</td>
</tr>
<tr>
<td>_MAT</td>
<td>32</td>
</tr>
<tr>
<td>_PADIC</td>
<td>30</td>
</tr>
<tr>
<td>_POL</td>
<td>30</td>
</tr>
<tr>
<td>_POLMOD</td>
<td>30</td>
</tr>
<tr>
<td>_QFI</td>
<td>32</td>
</tr>
<tr>
<td>_QFR</td>
<td>32</td>
</tr>
<tr>
<td>_QUAD</td>
<td>30</td>
</tr>
<tr>
<td>_REAL</td>
<td>29</td>
</tr>
<tr>
<td>_RFRAC</td>
<td>32</td>
</tr>
<tr>
<td>_SER</td>
<td>31</td>
</tr>
<tr>
<td>_STR</td>
<td>32</td>
</tr>
<tr>
<td>_VEC</td>
<td>32</td>
</tr>
<tr>
<td>_VECSMALL</td>
<td>32</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>udiviu_rem</td>
<td>84</td>
</tr>
<tr>
<td>udivui_rem</td>
<td>84</td>
</tr>
<tr>
<td>udivuui_rem</td>
<td>84</td>
</tr>
<tr>
<td>ugcd</td>
<td>86</td>
</tr>
<tr>
<td>uislucaasp stupid</td>
<td>90</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ZV_prod</td>
<td>134</td>
</tr>
<tr>
<td>zv_prod</td>
<td>135</td>
</tr>
<tr>
<td>zv_prod_Z</td>
<td>135</td>
</tr>
<tr>
<td>ZV_pval</td>
<td>79</td>
</tr>
<tr>
<td>ZV_pvalrem</td>
<td>78</td>
</tr>
<tr>
<td>ZV_search</td>
<td>134</td>
</tr>
<tr>
<td>zv_search</td>
<td>136</td>
</tr>
<tr>
<td>ZV_sort</td>
<td>134</td>
</tr>
<tr>
<td>ZV_sort_uniq</td>
<td>134</td>
</tr>
<tr>
<td>ZV_sum</td>
<td>134</td>
</tr>
<tr>
<td>zv_sum</td>
<td>135</td>
</tr>
<tr>
<td>ZV_togglesign</td>
<td>133</td>
</tr>
<tr>
<td>ZV_to_F2v</td>
<td>98</td>
</tr>
<tr>
<td>ZV_to_F1v</td>
<td>125</td>
</tr>
<tr>
<td>zv_to_F1v</td>
<td>126</td>
</tr>
<tr>
<td>ZV_to_nv</td>
<td>125</td>
</tr>
<tr>
<td>ZV_to_zv</td>
<td>125</td>
</tr>
<tr>
<td>zv_to_ZV</td>
<td>126</td>
</tr>
<tr>
<td>zv_to_zx</td>
<td>126</td>
</tr>
<tr>
<td>ZV_union_shallow</td>
<td>134</td>
</tr>
<tr>
<td>ZV_zMs_mul</td>
<td>140</td>
</tr>
<tr>
<td>ZV_Zm_mul</td>
<td>134</td>
</tr>
<tr>
<td>ZV_Z_dvd</td>
<td>79</td>
</tr>
<tr>
<td>zv_z_mul</td>
<td>135</td>
</tr>
<tr>
<td>ZVX_dvd</td>
<td>148</td>
</tr>
<tr>
<td>ZXQ_charpoly</td>
<td>143</td>
</tr>
<tr>
<td>ZXQ_mul</td>
<td>143</td>
</tr>
<tr>
<td>ZXQ_sqr</td>
<td>143</td>
</tr>
<tr>
<td>ZXT_rem12n</td>
<td>144</td>
</tr>
<tr>
<td>ZXT_to_F1xT</td>
<td>125</td>
</tr>
<tr>
<td>ZVX_dotproduct</td>
<td>144</td>
</tr>
<tr>
<td>ZXV_equal</td>
<td>144</td>
</tr>
<tr>
<td>ZXV_rem</td>
<td>142</td>
</tr>
<tr>
<td>ZXV_rem2n</td>
<td>142</td>
</tr>
<tr>
<td>ZXV_to_F1xV</td>
<td>125</td>
</tr>
<tr>
<td>ZVX_Z_mul</td>
<td>144</td>
</tr>
<tr>
<td>ZVXX_to_F1xX</td>
<td>125</td>
</tr>
<tr>
<td>ZXX_max_lg</td>
<td>144</td>
</tr>
<tr>
<td>ZXX_mul_Kronecker</td>
<td>144</td>
</tr>
<tr>
<td>ZXX_renormalize</td>
<td>144</td>
</tr>
<tr>
<td>ZXX_to_F2xX</td>
<td>116</td>
</tr>
<tr>
<td>ZXX_to_F2xX</td>
<td>116</td>
</tr>
<tr>
<td>ZXX_to_FlxX</td>
<td>125</td>
</tr>
<tr>
<td>ZXX_to_Kronecker</td>
<td>115</td>
</tr>
<tr>
<td>ZXX_to_Kronecker_spec</td>
<td>144</td>
</tr>
<tr>
<td>ZXX_Z_divexact</td>
<td>144</td>
</tr>
<tr>
<td>ZX_add</td>
<td>141</td>
</tr>
<tr>
<td>ZX_compositum_disjoint</td>
<td>143</td>
</tr>
<tr>
<td>ZX_content</td>
<td>142</td>
</tr>
<tr>
<td>ZX_copy</td>
<td>141</td>
</tr>
<tr>
<td>ZX_deriv</td>
<td>143</td>
</tr>
<tr>
<td>ZX_disc</td>
<td>143</td>
</tr>
<tr>
<td>ZX_equal</td>
<td>141</td>
</tr>
<tr>
<td>ZX_equal1</td>
<td>141</td>
</tr>
<tr>
<td>ZX_eval1</td>
<td>143</td>
</tr>
<tr>
<td>ZX_factor</td>
<td>143</td>
</tr>
<tr>
<td>ZX_gcd</td>
<td>142</td>
</tr>
<tr>
<td>ZX_gcd_all</td>
<td>142</td>
</tr>
<tr>
<td>ZX_graeffe</td>
<td>143</td>
</tr>
<tr>
<td>ZX_incremental_CRT</td>
<td>120</td>
</tr>
<tr>
<td>ZX_init_CRT</td>
<td>120</td>
</tr>
<tr>
<td>ZX_is_irred</td>
<td>143</td>
</tr>
<tr>
<td>ZX_is_squarefree</td>
<td>143</td>
</tr>
<tr>
<td>ZX_lval</td>
<td>79</td>
</tr>
<tr>
<td>ZX_lvalrem</td>
<td>79</td>
</tr>
<tr>
<td>ZX_neg</td>
<td>141</td>
</tr>
<tr>
<td>ZX_primitive_to_monic</td>
<td>142</td>
</tr>
<tr>
<td>ZX_pval</td>
<td>79</td>
</tr>
<tr>
<td>ZX_pvalrem</td>
<td>79</td>
</tr>
<tr>
<td>ZX_mul</td>
<td>142</td>
</tr>
<tr>
<td>ZX_mulspec</td>
<td>142</td>
</tr>
<tr>
<td>ZX_rem</td>
<td>142</td>
</tr>
<tr>
<td>ZX_rem2n</td>
<td>142</td>
</tr>
<tr>
<td>ZX_renormalize</td>
<td>141</td>
</tr>
<tr>
<td>ZX_renormalize</td>
<td>145</td>
</tr>
<tr>
<td>ZX_rescale</td>
<td>142</td>
</tr>
<tr>
<td>ZX_rescale_lt</td>
<td>142</td>
</tr>
<tr>
<td>ZX_resultant</td>
<td>143</td>
</tr>
<tr>
<td>ZX_shift</td>
<td>145</td>
</tr>
<tr>
<td>ZX_shifti</td>
<td>142</td>
</tr>
<tr>
<td>ZX_sqr</td>
<td>142</td>
</tr>
<tr>
<td>ZX_sgrspspec</td>
<td>142</td>
</tr>
<tr>
<td>ZX_sqrspec</td>
<td>143</td>
</tr>
<tr>
<td>ZX_sqrspec</td>
<td>143</td>
</tr>
<tr>
<td>ZX_sub</td>
<td>141</td>
</tr>
<tr>
<td>ZX_to_F2x</td>
<td>116</td>
</tr>
<tr>
<td>ZX_to_Flx</td>
<td>124</td>
</tr>
<tr>
<td>ZX_to_FlxX</td>
<td>125</td>
</tr>
<tr>
<td>zx_to_zv</td>
<td>126</td>
</tr>
<tr>
<td>zx_to_ZX</td>
<td>125</td>
</tr>
<tr>
<td>ZX_translate</td>
<td>143</td>
</tr>
<tr>
<td>ZX_unscale</td>
<td>143</td>
</tr>
<tr>
<td>ZX_unscale_div</td>
<td>143</td>
</tr>
<tr>
<td>ZX_val</td>
<td>142</td>
</tr>
<tr>
<td>ZX_valrem</td>
<td>142</td>
</tr>
<tr>
<td>ZX_Zp_root</td>
<td>122</td>
</tr>
</tbody>
</table>