

User's Guide
to
the PARI library

(version 2.19.0)

The PARI Group

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

Home Page:
<https://pari.math.u-bordeaux.fr/>

Copyright © 2000–2025 The PARI Group

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions, or translations, of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

PARI/GP is Copyright © 2000–2025 The PARI Group

PARI/GP is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation. It is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY WHATSOEVER.

Table of Contents

Chapter 4: Programming PARI in Library Mode	13
4.1 Introduction: initializations, universal objects	13
4.2 Important technical notes	14
4.2.1 Backward compatibility	14
4.2.2 Types	14
4.2.3 Type recursivity	15
4.2.4 Variations on basic functions	15
4.2.5 Portability: 32-bit / 64-bit architectures	16
4.2.6 Using <code>malloc</code> / <code>free</code>	16
4.3 Garbage collection	17
4.3.1 Why and how	17
4.3.2 Gory technical details	19
4.3.3 The basic 3 garbage collecting functions	19
4.3.4 A first detailed example	20
4.3.5 More examples: variations on a theme	21
4.3.6 Random garbage collection	22
4.3.7 Final comments	22
4.4 Creation of PARI objects, assignments, conversions	23
4.4.1 Creation of PARI objects	23
4.4.2 Sizes	24
4.4.3 Assignment and copy	25
4.4.4 Clones	25
4.4.5 Conversions	25
4.5 Implementation of the PARI types	26
4.5.1 Type <code>t_INT</code> (integer)	27
4.5.2 Type <code>t_REAL</code> (real number)	28
4.5.3 Type <code>t_INTMOD</code>	29
4.5.4 Type <code>t_FRAC</code> (rational number)	29
4.5.5 Type <code>t_FFELT</code> (finite field element)	29
4.5.6 Type <code>t_COMPLEX</code> (complex number)	29
4.5.7 Type <code>t_PADIC</code> (p -adic numbers)	29
4.5.8 Type <code>t_QUAD</code> (quadratic number)	30
4.5.9 Type <code>t_POLMOD</code> (polmod)	30
4.5.10 Type <code>t_POL</code> (polynomial)	30
4.5.11 Type <code>t_SER</code> (power series)	31
4.5.12 Type <code>t_RFRAC</code> (rational function)	31
4.5.13 Type <code>t_QFB</code> (binary quadratic form)	31
4.5.14 Type <code>t_VEC</code> and <code>t_COL</code> (vector)	32
4.5.15 Type <code>t_MAT</code> (matrix)	32
4.5.16 Type <code>t_VECSMALL</code> (vector of small integers)	32
4.5.17 Type <code>t_STR</code> (character string)	32
4.5.18 Type <code>t_ERROR</code> (error context)	32
4.5.19 Type <code>t_CLOSURE</code> (closure)	32
4.5.20 Type <code>t_INFINITY</code> (infinity)	32
4.5.21 Type <code>t_LIST</code> (list)	32
4.6 PARI variables	33

4.6.1 Multivariate objects	33
4.6.2 Creating variables	33
4.6.3 Comparing variables	35
4.7 Input and output	35
4.7.1 Input	35
4.7.2 Output to screen or file, output to string	36
4.7.3 Errors	38
4.7.4 Warnings	38
4.7.5 Debugging output	39
4.7.6 Timers and timing output	39
4.8 Iterators, Numerical integration, Sums, Products	40
4.8.1 Iterators	40
4.8.2 Iterating over primes	42
4.8.3 Parallel iterators	43
4.8.4 parallel data processing	45
4.8.5 Numerical analysis	45
4.9 Catching exceptions	46
4.9.1 Basic use	46
4.9.2 Advanced use	46
4.10 A complete program	47
Chapter 5: Technical Reference Guide: the basics	51
5.1 Initializing the library	51
5.1.1 General purpose	51
5.1.2 Technical functions	52
5.1.3 Notions specific to the GP interpreter	54
5.1.4 Public callbacks	55
5.1.5 Configuration variables	56
5.1.6 Utility functions	56
5.1.7 Saving and restoring the GP context	57
5.1.8 GP history	57
5.2 Handling GENs	58
5.2.1 Allocation	58
5.2.2 Length conversions	59
5.2.3 Read type-dependent information	60
5.2.4 Eval type-dependent information	62
5.2.5 Set type-dependent information	62
5.2.6 Type groups	63
5.2.7 Accessors and components	64
5.3 Global numerical constants	64
5.3.1 Constants related to word size	65
5.3.2 Masks used to implement the GEN type	65
5.3.3 $\log 2$, π	66
5.4 Iterating over small primes, low-level interface	66
5.5 Handling the PARI stack	67
5.5.1 Allocating memory on the stack	67
5.5.2 Stack-independent binary objects	68
5.5.3 Garbage collection	68
5.5.4 Garbage collection: advanced use	71
5.5.5 Debugging the PARI stack	72

5.5.6 Copies	72
5.5.7 Simplify	72
5.6 The PARI heap	72
5.6.1 Introduction	72
5.6.2 Public interface	73
5.6.3 Implementation note	73
5.7 Handling user and temp variables	74
5.7.1 Low-level	74
5.7.2 User variables	74
5.7.3 Temporary variables	74
5.8 Adding functions to PARI	75
5.8.1 Nota Bene	75
5.8.2 Coding guidelines	75
5.8.3 GP prototypes	76
5.8.4 Integration with gp as a shared module	78
5.8.5 Library interface for install	79
5.8.6 Integration by patching gp	79
5.9 Globals related to PARI configuration	80
5.9.1 PARI version numbers	80
5.9.2 Miscellaneous	80
Chapter 6: Arithmetic kernel: Level 0 and 1	81
6.1 Level 0 kernel (operations on ulongs)	81
6.1.1 Micro-kernel	81
6.1.2 Modular kernel	82
6.1.3 Modular kernel with “precomputed inverse”	84
6.1.4 Switching between Fl_xxx and standard operators	85
6.2 Level 1 kernel (operations on longs, integers and reals)	86
6.2.1 Creation	86
6.2.2 Assignment	87
6.2.3 Copy	87
6.2.4 Conversions	88
6.2.5 Integer parts	89
6.2.6 2-adic valuations and shifts	90
6.2.7 From t_INT to bits or digits in base 2^k and back	91
6.2.8 Integer valuation	91
6.2.9 Generic unary operators	93
6.2.10 Comparison operators	93
6.2.11 Generic binary operators	95
6.2.12 Exact division and divisibility	98
6.2.13 Division with integral operands and t_REAL result	98
6.2.14 Division with remainder	98
6.2.15 Modulo to longs	99
6.2.16 Powering, Square root	100
6.2.17 GCD, extended GCD and LCM	101
6.2.18 Continued fractions and convergents	102
6.2.19 Pseudo-random integers	102
6.2.20 Modular operations	103
6.2.21 Extending functions to vector inputs	106
6.2.22 Miscellaneous arithmetic functions	106

Chapter 7: Level 2 kernel	109
7.1 Naming scheme	109
7.2 Coefficient ring	111
7.3 Modular arithmetic	112
7.3.1 FpC / FpV, FpM	112
7.3.2 Flc / Flv, Flm	116
7.3.3 F2c / F2v, F2m	119
7.3.4 F3c / F3v, F3m	121
7.3.5 FlxqV, FlxqC, FlxqM	122
7.3.6 FpX	123
7.3.7 FpXQ, Fq	127
7.3.8 FpXQ	129
7.3.9 Fq	130
7.3.10 FpXn	132
7.3.11 FpXC, FpXM	132
7.3.12 FpXX, FpXY	132
7.3.13 FpXk	133
7.3.14 FpXQX, FqX	133
7.3.15 FpXQXn, FqXn	135
7.3.16 FpXQXQ, FqXQ	136
7.3.17 Flx	139
7.3.18 FlxV	145
7.3.19 FlxM	145
7.3.20 FlxT	145
7.3.21 Flxn	146
7.3.22 Flxq	146
7.3.23 FlxX	148
7.3.24 FlxXV, FlxXC, FlxXM	149
7.3.25 FlxqX	150
7.3.26 FlxqXQ	153
7.3.27 FlxqXn	154
7.3.28 F2x	154
7.3.29 F2xq	156
7.3.30 F2xn	157
7.3.31 F2xqV, F2xqM	157
7.3.32 F2xX	158
7.3.33 F2xXV/F2xXC	158
7.3.34 F2xqX	158
7.3.35 F2xqXQ	160
7.3.36 Functions returning objects with t_INTMOD coefficients	160
7.3.37 Slow Chinese remainder theorem over Z	162
7.3.38 Fast remainders	163
7.3.39 Fast Chinese remainder theorem over Z	164
7.3.40 Rational reconstruction	165
7.3.41 Zp	166
7.3.42 ZpM	166
7.3.43 ZpX	167
7.3.44 ZpXQ	168
7.3.45 Zq	168

7.3.46	$\mathbb{Z}_p\text{XQM}$	168
7.3.47	$\mathbb{Z}_p\text{XQX}$	169
7.3.48	$\mathbb{Z}_q\text{X}$	169
7.3.49	$\mathbb{Z}_p\text{XQXQ}$	169
7.3.50	Other p -adic functions	170
7.3.51	Conversions involving single precision objects	171
7.4	Higher arithmetic over $\mathbb{Z}$: primes, factorization	175
7.4.1	Pure powers	175
7.4.2	Factorization	176
7.4.3	Coprime factorization	178
7.4.4	Checks attached to arithmetic functions	178
7.4.5	Incremental integer factorization	179
7.4.6	Integer core, squarefree factorization	180
7.4.7	Primes, primality and compositeness tests	180
7.4.8	Iterators over primes	182
7.5	Integral, rational and generic linear algebra	182
7.5.1	$\mathbb{Z}\mathbb{C} / \mathbb{Z}\mathbb{V}, \mathbb{Z}\mathbb{M}$	182
7.5.2	$\mathbb{Q}\mathbb{M}$	187
7.5.3	Qevproj	187
7.5.4	$\mathbb{z}\mathbb{v}, \mathbb{z}\mathbb{m}$	188
7.5.5	$\mathbb{Z}\mathbb{M}\mathbb{V} / \mathbb{z}\mathbb{m}\mathbb{V}$ (vectors of $\mathbb{Z}\mathbb{M}/\mathbb{z}\mathbb{m}$)	189
7.5.6	$\mathbb{Q}\mathbb{C} / \mathbb{Q}\mathbb{V}, \mathbb{Q}\mathbb{M}$	189
7.5.7	$\text{Rg}\mathbb{C} / \text{Rg}\mathbb{V}, \text{Rg}\mathbb{M}$	189
7.5.8	$\mathbb{Z}\mathbb{G}$	195
7.5.9	Sparse and blackbox linear algebra	196
7.5.10	Obsolete functions	197
7.6	Integral, rational and generic polynomial arithmetic	198
7.6.1	$\mathbb{Z}\mathbb{X}$	198
7.6.2	Resultants	201
7.6.3	$\mathbb{Z}\mathbb{X}\mathbb{V}$	201
7.6.4	$\mathbb{Z}\mathbb{X}\mathbb{T}$	201
7.6.5	$\mathbb{Z}\mathbb{X}\mathbb{Q}$	202
7.6.6	$\mathbb{Z}\mathbb{X}\mathbb{n}$	202
7.6.7	$\mathbb{Z}\mathbb{X}\mathbb{Q}\mathbb{M}$	202
7.6.8	$\mathbb{Z}\mathbb{X}\mathbb{Q}\mathbb{X}$	202
7.6.9	$\mathbb{Z}\mathbb{X}\mathbb{X}$	203
7.6.10	$\mathbb{Z}\mathbb{X}\mathbb{k}/\mathbb{Q}\mathbb{X}\mathbb{k}$	203
7.6.11	$\mathbb{Q}\mathbb{X}$	204
7.6.12	$\mathbb{Q}\mathbb{X}\mathbb{Q}$	204
7.6.13	$\mathbb{Q}\mathbb{X}\mathbb{Q}\mathbb{X}$	205
7.6.14	$\mathbb{Q}\mathbb{X}\mathbb{Q}\mathbb{M}$	205
7.6.15	$\mathbb{z}\mathbb{x}$	206
7.6.16	$\text{Rg}\mathbb{X}$	206
7.6.17	$\text{Rg}\mathbb{X}\mathbb{n}$	212
7.6.18	$\text{Rg}\mathbb{X}\mathbb{n}\mathbb{V}$	213
7.6.19	$\text{Rg}\mathbb{X}\mathbb{Q}$	213
7.6.20	$\text{Rg}\mathbb{X}\mathbb{Q}\mathbb{V}, \text{Rg}\mathbb{X}\mathbb{Q}\mathbb{C}$	214
7.6.21	$\text{Rg}\mathbb{X}\mathbb{Q}\mathbb{M}$	214
7.6.22	$\text{Rg}\mathbb{X}\mathbb{Q}\mathbb{X}$	214

Chapter 8: Black box algebraic structures	215
8.1 Black box groups	215
8.1.1 Black box groups with pairing	217
8.1.2 Functions returning black box groups	217
8.2 Black box fields	218
8.2.1 Functions returning black box fields	219
8.3 Black box algebra	219
8.3.1 Functions returning black box algebras	220
8.4 Black box ring	220
8.5 Black box free $\mathbf{Z}_p$ -modules	221
Chapter 9: Operations on general PARI objects	223
9.1 Conversions	223
9.1.1 Scalars	223
9.1.2 Modular objects / lifts	225
9.1.3 Between polynomials and coefficient arrays	225
9.2 Constructors	227
9.2.1 Clean constructors	227
9.2.2 Unclean constructors	230
9.2.3 From roots to polynomials	234
9.3 Integer parts	234
9.4 Valuation and shift	234
9.5 Comparison operators	235
9.5.1 Generic	235
9.5.2 Comparison with a small integer	235
9.6 Miscellaneous Boolean functions	236
9.6.1 Obsolete	237
9.7 Sorting	237
9.7.1 Basic sort	237
9.7.2 Indirect sorting	237
9.7.3 Generic sort and search	238
9.7.4 Further useful comparison functions	239
9.8 Division	239
9.9 Divisibility, Euclidean division	240
9.10 GCD, content and primitive part	241
9.10.1 Generic	241
9.10.2 Over the rationals	241
9.11 Generic arithmetic operators	243
9.11.1 Unary operators	243
9.11.2 Binary operators	243
9.12 Generic operators: product, powering, factorback	244
9.13 Matrix and polynomial norms	245
9.14 Substitution and evaluation	246
Chapter 10: Miscellaneous mathematical functions	249
10.1 Fractions	249
10.2 Binomials	249
10.3 Real numbers	249
10.4 Complex numbers	250
10.5 Low accuracy input or output	250
10.6 Quadratic numbers and binary quadratic forms	251

10.7	Polynomials	251
10.7.1	Obsolete functions	252
10.8	Power series	252
10.9	Functions to handle <code>t_FFELT</code>	253
10.9.1	FFX	256
10.9.2	FFM	257
10.9.3	FFXQ	258
10.10	Transcendental functions	258
10.10.1	Transcendental functions with <code>t_REAL</code> arguments	258
10.10.2	Other complex transcendental functions	260
10.10.3	Modular functions	261
10.10.4	Transcendental functions with <code>t_PADIC</code> arguments	262
10.10.5	Cached constants	262
10.10.6	Obsolete functions	263
10.11	Permutations	263
10.12	Small groups	264
Chapter 11:	Standard data structures	269
11.1	Character strings	269
11.1.1	Functions returning a <code>char *</code>	269
11.1.2	Functions returning a <code>t_STR</code>	270
11.1.3	Dynamic strings	270
11.2	Output	271
11.2.1	Output contexts	271
11.2.2	Default output context	271
11.2.3	PARI colors	272
11.2.4	Obsolete output functions	272
11.3	Files	273
11.3.1	pariFILE	273
11.3.2	Temporary files	274
11.4	Errors	274
11.4.1	Internal errors, “system” errors	274
11.4.2	Syntax errors, type errors	275
11.4.3	Overflows	276
11.4.4	Errors triggered intentionally	277
11.4.5	Mathematical errors	278
11.4.6	Miscellaneous functions	279
11.5	Hashtables	279
11.6	Dynamic arrays	282
11.6.1	Initialization	282
11.6.2	Adding elements	282
11.6.3	Accessing elements	282
11.6.4	Stack of stacks	282
11.6.5	Public interface	283
11.7	Vectors and Matrices	283
11.7.1	Access and extract	283
11.7.2	Componentwise operations	285
11.7.3	Low-level vectors and columns functions	285
11.8	Vectors of small integers	286
11.8.1	<code>t_VECSMALL</code>	286

11.8.2 Vectors of t_VECSMALL	288
Chapter 12: Functions related to the GP interpreter	289
12.1 Handling closures	289
12.1.1 Functions to evaluate t_CLOSURE	289
12.1.2 Control flow functions	290
12.1.3 Functions to handle control flow changes	291
12.1.4 Functions to deal with lexical local variables	291
12.1.5 Functions returning new closures	292
12.1.6 Functions used by the gp debugger (break loop)	292
12.1.7 Standard wrappers for iterators	292
12.2 Defaults	293
12.3 Records and Lazy vectors	296
Chapter 13: Algebraic Number Theory	299
13.1 General Number Fields	299
13.1.1 Number field types	299
13.1.2 Extracting info from a nf structure	301
13.1.3 Extracting info from a bnf structure	302
13.1.4 Extracting info from a bnr structure	303
13.1.5 Extracting info from an rnf structure	304
13.1.6 Extracting info from a bid structure	305
13.1.7 Extracting info from a znstar structure	306
13.1.8 Inserting info in a number field structure	306
13.1.9 Increasing accuracy	307
13.1.10 Number field arithmetic	307
13.1.11 Number field arithmetic for linear algebra	310
13.1.12 Cyclotomic field arithmetic for linear algebra	311
13.1.13 Cyclotomic trace	311
13.1.14 Elements in factored form	311
13.1.15 Ideal arithmetic	313
13.1.16 Maximal ideals	316
13.1.17 Decomposition groups	318
13.1.18 Reducing modulo maximal ideals	318
13.1.19 Valuations	319
13.1.20 Signatures	320
13.1.21 Complex embeddings	321
13.1.22 Maximal order and discriminant, conversion to nf structure	322
13.1.23 Computing in the class group	323
13.1.24 Floating point embeddings, the T_2 quadratic form	325
13.1.25 Ideal reduction, low level	325
13.1.26 Ideal reduction, high level	326
13.1.27 Class field theory	327
13.1.28 Abelian maps	329
13.1.29 Grunwald–Wang theorem	330
13.1.30 Relative equations, Galois conjugates	330
13.1.31 Units	332
13.1.32 Obsolete routines	332
13.2 Galois extensions of Q	333
13.2.1 Extracting info from a gal structure	333
13.2.2 Miscellaneous functions	334

13.3 Quadratic number fields and quadratic forms	334
13.3.1 Checks	334
13.3.2 Class number	334
13.3.3 <code>t_QFB</code>	335
13.3.4 Efficient real quadratic forms	337
13.4 Linear algebra over $\mathbf{Z}$	339
13.4.1 Hermite and Smith Normal Forms	339
13.4.2 The LLL algorithm	343
13.4.3 Linear dependencies	345
13.4.4 Reduction modulo matrices	345
13.5 Finite abelian groups and characters	346
13.5.1 Abstract groups	346
13.5.2 Dirichlet characters	347
13.6 Hecke characters	348
13.7 Central simple algebras	349
13.7.1 Initialization	349
13.7.2 Type checks	349
13.7.3 Shallow accessors	350
13.7.4 Other low-level functions	351
Chapter 14: Elliptic curves and arithmetic geometry	353
14.1 Elliptic curves	353
14.1.1 Types of elliptic curves	353
14.1.2 Type checking	353
14.1.3 Extracting info from an <code>ell</code> structure	354
14.1.4 Points	358
14.1.5 Change of variables	358
14.1.6 Generic helper functions	358
14.1.7 Functions to handle elliptic curves over finite fields	359
14.2 Arithmetic on elliptic curve over a finite field in simple form	359
14.2.1 Helper functions	359
14.2.2 Elliptic curves over $\mathbf{F}_p$, $p > 3$	360
14.2.3 <code>FpE</code>	360
14.2.4 <code>Fle</code>	361
14.2.5 <code>FpJ</code>	362
14.2.6 <code>Flj</code>	363
14.2.7 Elliptic curves over $\mathbf{F}_{2^n}$	363
14.2.8 <code>F2xqE</code>	363
14.2.9 Elliptic curves over $\mathbf{F}_q$, small characteristic $p > 2$	364
14.2.10 <code>FlxqE</code>	364
14.2.11 Elliptic curves over $\mathbf{F}_q$, large characteristic	365
14.2.12 <code>FpXQE</code>	366
14.3 Functions related to modular polynomials	367
14.3.1 Functions related to modular invariants	367
14.4 Other curves	367
Chapter 15: L-functions	369
15.1 Accessors	369
15.2 Conversions and constructors	370
15.3 Variants of GP functions	371
15.4 Inverse Mellin transforms of Gamma products	371

Chapter 16: Modular symbols	373
Chapter 17: Modular forms	375
17.1 Implementation of public data structures	375
17.1.1 Accessors for modular form spaces	375
17.1.2 Accessors for individual modular forms	376
17.1.3 Nebentypus	377
17.1.4 Miscellaneous functions	377
Chapter 18: Plots	379
18.1 Highlevel functions	379
18.2 Function	380
18.2.1 Obsolete functions	380
18.3 Dump rectwindows to a PostScript or SVG file	381
18.4 Technical functions exported for convenience	381
Appendix A: A Sample program and Makefile	383
Appendix B: PARI and threads	385
Index	388

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 describes high level PARI routines available through the calculator. The present manual focuses on the usage of the PARI library, assuming general familiarity with the contents of these chapters and with ANSI C programming. In the present chapter, we introduce concepts 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, called `pari_PRIMES`, of type `pari_prime*` (pointer to 32bit unsigned integer). 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 evolve 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 function prototype remained the same).

This file is included by `pari.h`, but a large section is commented out by default. Define `PARI_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 `-DPARI_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

```
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)`gen_1`. 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 **e**lement, 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 **GEN**s. 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 **m**ultidimensional **a**rray **e**lement). The syntax is **gmael***n*(*V*, *a*₁, ..., *a*_{*n*}), where *V* is a **GEN**, the *a*_{*i*} 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. **gmael12**).

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_MAT**s 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 **GEN**s, 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*)

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 **ulongs**), 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 **GEN**s of type **t_INT** (this is not checked).

GEN addrr(**GEN** *x*, **GEN** *y*): here *x* and *y* are **GEN**s 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. 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_INTMOD** arithmetic:

GEN Fp_add(**GEN** *x*, **GEN** *y*, **GEN** *m*): returns the sum of *x* and *y* modulo *m*. Here *x*, *y*, *m* are **t_INT**s (this is not checked). The operation is more efficient if the inputs *x*, *y* are reduced modulo *m*, 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 `addii` on a `t_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 `gadd` or `gmul`, is dropped in kernel routines of Level 1. There, it is replaced by the simpler rule: the result is a `t_INT` if and only if all arguments are integer types (`t_INT` but also C `long` and `ulong`) and a `t_REAL` otherwise. For instance, multiplying a `t_REAL` by a `t_INT` always yields a `t_REAL` if you use `mulir`, where `gmul` returns the `t_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 `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 `long` integers might matter. Here are the most important ones:

<code>DEFAULTPREC</code>	64	(≈ 19 decimal digits)
<code>MEDDEFAULTPREC</code>	128	(≈ 38 decimal digits)
<code>BIGDEFAULTPREC</code>	192	(≈ 57 decimal digits)

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

```
GEN gexp(GEN x, long prec).
```

The last argument `prec` is the default floating point precision required, given in bits. It is *only* used if `x` is an exact object, otherwise the relative precision is determined by the precision of `x`.

4.2.6 Using `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:

`void* pari_malloc(size_t size)` calls `malloc` to allocate `size` 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 `realloc` instead of `malloc`.

`void pari_realloc_ip(void** ptr, size_t size)` equivalent to `*ptr= realloc(*ptr, size)`, while blocking `SIGINT`.

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

`void pari_free(void* ptr)` 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.

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

4.3 Garbage collection.

4.3.1 Why and how.

As we have seen, `pari_init` allocates a big range of addresses, the *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 `bot` and ends just before `top`. This means that the quantity

$$(\text{top} - \text{bot}) / \text{sizeof}(\text{long})$$

is (roughly) equal to the `size` argument of `pari_init`. The PARI stack also has a “current stack pointer” called `avma`, which stands for **a**vailab**l**e **m**emory **a**ddress. These three variables are global (declared by `pari.h`). They are of type `pari_sp`, which means *pari stack pointer*.

The stack is oriented upside-down: the more recent an object, the closer to `bot`. Accordingly, initially `avma = top`, and `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 subsections.

- 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 (undocumented 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 ...
set_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. Think of the `set_avma` call as a simple `avma = av` restoring the `avma` value.

- 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`.

- If on the contrary, you want to keep the latest computed object z and destroy everything else since the reference point, use

```
pari_sp av = avma; /* record initial avma */
... garbage ...
oldz = fun();
newz = gc_upto(av, oldz); /* destroy garbage but keep z safe */
```

Of course, this can be written directly as

```
z = gc_upto(av, fun());
```

This form is actually preferred since it guarantees garbage collection occurs at the right time, just after `fun()` returns its result. Note that the `gc_upto` call moves `oldz` and in fact *destroys* it. One should never refer to it again, but use the updated `newz` instead.

- When an arbitrary GEN must be recovered from the middle of a computation, even if it was constructed incrementally, then you can use the most robust (hence slowest) garbage collection function: `gc_GEN`.

```
pari_sp av = avma; /* record initial avma */
... garbage ...
oldz = fun();
... garbage ...
newz = gc_GEN(av, oldz); /* destroy garbage but keep z safe */
```

This function even allows z to contain shallow data (pointers to other parts of the stack), which is forbidden for high-level PARI functions. Robustness comes at a price but this is at most twice slower than `gc_upto`. Again `oldz` should not be accessed anymore.

- Finally, when more than one result needs to be saved, you can use the following idiom:

```
pari_sp av = avma; /* record initial avma */
... garbage ...
z1 = fun();
... garbage ...
z2 = fun();
... garbage ...
gc_all(av, 2, &z1, &z2); /* destroy garbage, keep z1 and z2 safe */
```

As usual, `gc_all` changes `z1` and `z2`: the old values should no longer be accessed. This is a variadic function, which essentially performs as many simultaneous `gc_GEN` as needed: if n objects need to be updated, use

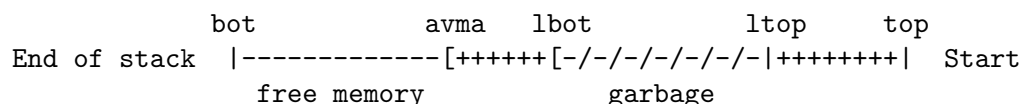
```
gc_all(av, n, &z1, ..., &zn);
```

4.3.2 Gory technical details.

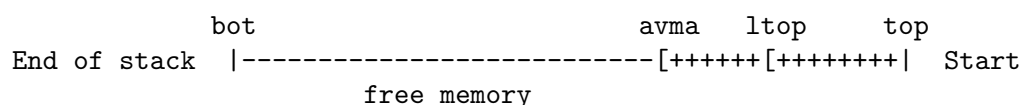
You may skip this section on first reading. We will now describe the underlying mechanisms in terms of the lowest level function

`size_t gc_stack_update(pari_sp ltop, pari_sp lbot)`. This function cleans up the stack between `ltop` and `lbot`, where `lbot ≤ ltop`, and returns the offset $o = ltop - lbot$. This means the following.

1) we translate (copy) all data in the interval `[avma, lbot[` to `[avma + o, lbot + o]`. Note that the right extremity is now exactly `ltop`. Graphically



becomes:



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 (the original) `avma` and `lbot`, i.e. the ones that we want to keep and that have been translated. Every component of such an object which is not a codeword is a pointer to an address which is either

- between `avma` and `lbot`, in which case we add to it the offset o ;
- larger than or equal to `ltop`, in which case it does not change, or
- between `lbot` and `ltop` in which case the function raises an error: “significant pointers lost in gc”.

3) `avma` is replaced by `avma + o` as in the diagram above.

4) The function returns the offset o .

As stated above, no component of the remaining objects should belong to the erased segment `[lbot, ltop[`, and the function checks this. But beware as well that the addresses of the objects in the translated zone have now changed, so you must not access any pointer which previously pointed into the zone below `ltop`.

4.3.3 The basic 3 garbage collecting functions.

Let us recap: although all is implemented in terms of the `gc_stack_update` low level function, you don’t want to use it directly. You mostly want to use one of the following three functions to collect garbage (reclaim memory) while keeping some results safe. Many others will be described in Section 5.5.3 for specialized scenarios.

`GEN gc_upto(pari_sp ltop, GEN q)`. This applies when `q` is the last object on the stack, returned by a high level PARI function. It cleans the stack between `ltop` and the object `q` and returns `q` updated. Your own functions or low level PARI functions (such as shallow functions) described in the next Chapter may not satisfy this: this will be mentioned in their documentation.

GEN `gc_GEN(pari_sp ltop, GEN q)`. Works in all cases. Functionally equivalent to, but more efficient than

```
gc_upto(ltop, gcopy(q))
```

In this case, the GEN parameter `q` need not satisfy any property before the garbage collection. Of course, this is about twice slower than `gc_upto`, because `q` has to be copied to a clean stack zone first. This function is a special case of `gc_all` below, where $n = 1$.

GEN `gc_all(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;
garbage(); x = anything();
garbage(); y = anything();
garbage(); gc_all(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. This is as robust as `gc_GEN`. For convenience, this function also returns the new value of x , the first object recovered.

4.3.4 A first detailed example.

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, the stack contains as unnecessary garbage `x2` and `y2`. More precisely it contains (in this order) `x2`, `y2`, `z`.

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 the latest computed objects and we do not want to give back `z`.

Using `gc_upto`, 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);
GEN z = gc_upto(ltop, gadd(x2, y2));
```

or even

```
ltop = avma;
z = gc_upto(ltop, gadd(gsqr(x), gsqr(y)));
```

This works because the output of `gadd` is the last object on the stack and everything else that was introduced after recording the `ltop` reference point is deleted. Another possibility is to use

```
ltop = avma;
z = gc_GEN(ltop, gadd(gsqr(x), gsqr(y)));
```

which would work in all cases, even if `gadd` were shallow. Garbage collection would be at most twice slower, and certainly negligible compared to the squaring times. Make sure you understand exactly what has happened before you go on!

4.3.5 More examples: variations on a theme. it is often necessary to do several garbage collections during a computation. However, the fewer the better. The only condition for `gc_upto` to work is that the garbage be connected and we need only recover the last result on the stack. The computation should be arranged so that there is a minimal number of connected pieces of garbage.

For example suppose we want to write a function of two GEN variables `x` and `y` which creates the vector $[x^2 + y, y^2 + x]$. Without garbage collecting, one would write:

```
x2 = gsqr(x); a = gadd(x2, y);
y2 = gsqr(y); b = gadd(y2, x);
z = mkvec2(a, b); /* shallow: not suitable for gc_upto! */
```

This leaves a dirty stack containing `x2`, `a`, `y2`, `b` and `z`. The garbage here consists of `x2` and `y2`, which are separated by `a`, which we need to keep. But if we compute `y2` *before* `a` then the garbage becomes connected, and we get the following program with garbage collecting:

```
ltop = avma; x2 = gsqr(x); y2 = gsqr(y);
z = cgetg(3, t_VEC);
gel(z, 1) = gadd(x2,y);
gel(z, 2) = gadd(y2,x); z = gc_upto(ltop, z);
```

Beware that

```
ltop = avma; a = gadd(gsqr(x), y); b = gadd(gsqr(y), x);
z = cgetg(3, t_VEC);
gel(z, 1) = a;
gel(z, 2) = b; z = gc_upto(ltop,z); /* WRONG */
```

is a disaster since `a` and `b` are created before `z`, so the call to `gc_upto` overwrites them, leaving `gel(z, 1)` and `gel(z, 2)` pointing at random data!

On the other hand

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

leaves the results of `gsqr(x)` and `gsqr(y)` on the stack (and lets `gc_upto` update them for naught). In some sense, the best version is

```
z = cgetg(3, t_VEC);
ltop = avma; gel(z, 1) = gc_upto(ltop, gadd(gsqr(x), y));
ltop = avma; gel(z, 2) = gc_upto(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 $z = x*y$, and set $x = x_r + i*x_i$ and similarly for y and z . We compute $a = x_r * y_r$, $b = x_i * y_i$, $c = (x_r + x_i) * (y_r + y_i)$, and then we have $z_r = a - b$, $z_i = c - (a + b)$. The program is as follows:

```
ltop = avma;
a = gmul(gel(x,1), gel(y,1));
b = gmul(gel(x,2), gel(y,2));
c = gmul(gadd(gel(x,1), gel(x,2)), gadd(gel(y,1), gel(y,2)));
d = gadd(a,b); z = cgetg(3, t_COMPLEX);
gel(z, 1) = gsub(a,b);
gel(z, 2) = gsub(c,d); z = gc_upto(ltop, z);
```

4.3.6 Random garbage collection.

The most important usage of `gc_all` is to make sure loops don't fill up the stack:

```
pari_sp av = avma;
GEN x, y;
while (...)
{
    garbage(); x = anything();
    garbage(); y = anything();
    garbage(); gc_all(av, 2, &x, &y);
}
```

Here we assume that only `x` and `y` are needed from one iteration to the next. As it can be costly to call `gc_all` once for each iteration, we may do this only when it is necessary:

```
/* memory is running low (half spent since entry) */
if (gc_needed(av,1)) gc_all(av, 2, &x, &y);
```

More precisely, the macro `stack_lim(av,n)` denotes an address where $2^{n-1}/(2^{n-1} + 1)$ of the remaining stack space since reference point `av` is exhausted ($1/2$ for $n = 1$, $2/3$ for $n = 2$, etc.). The test `gc_needed(av,n)` becomes true whenever `avma` drops below that address.

4.3.7 Final comments.

PARI's manual garbage collection may seem cumbersome at first. But it is flexible and very fast, and compares favorably with methods used in other systems. Our benchmarks indicate that the price paid during garbage collections, when properly used, is usually less than 1% of the total running time, which is quite acceptable!

It is of course harder on the programmer, and error-prone if you do not stick to a consistent PARI programming style. At first, just use `gc_GEN` or `gc_all` 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. In any case, all garbage collections routines run in almost linear complexity in the size of the recovered data.

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 (for instance `affii` or `affrr`).

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 5 for type `t_PADIC` and `t_QFB`. 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 create efficiently a constant object. 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 prec)`

should be used instead of `cgetg(length, t_INT)` and `cgetg(nbits2lg(prec), t_REAL)` respectively. Although `cgetg(length, t_REAL)` remains a possibility, if the `length` value is available, rather than `cgetr(bit_accuracy(length))`. 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 = cgeti(DEFAULTPREC)` and `cgetg(DEFAULTPREC, t_INT)` create a `t_INT` whose “precision” is `bit_accuracy(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. `cgeti(4)`, but this would have different meanings on different machines as `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 `bit_accuracy(MEDDEFAULTPREC) = 96` bits:

```
z = cgetg(3, t_COMPLEX);
gel(z, 1) = cgetr(MEDDEFAULTPREC);
gel(z, 2) = cgetr(MEDDEFAULTPREC);
```

or simply `z = cgetc(MEDDEFAULTPREC)`.

3) To create a matrix object for 4×3 matrices:

```

z = cgetg(4, t_MAT);
for(i=1; i<4; i++) gel(z, i) = cgetg(5, t_COL);

```

or simply `z = zeromatcopy(4, 3)`, which further initializes all entries to `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 `cgetg` creates only the “root”, and other calls to `cgetg` must be made to produce the whole tree. For matrices, a common mistake is to think that `z = cgetg(4, 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 `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 `varargs` functions:

`GEN mkpoln(long n, ...)` Returns the `t_POL` whose n coefficients (`GEN`) follow, in order of decreasing degree.

```

mkpoln(3, gen_1, gen_2, 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.

See also the technical functions `mkintn` and `mkvecsma1ln` in the next chapters, which require great care with integer constants (whose type must be explicited).

Warning. Contrary to the policy of general PARI functions, the functions `mkvecn`, `mkcoln` and `mkpoln` are shallow: they do *not* copy their arguments. For instance

```

/* stack-clean: components are universal objects */
z = mkvecn(3, gen_1, gen_0, gen_2);

/* not OK for gc_upto: 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);

```

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.

4.4.3 Assignment and copy. 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.

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. The function which does this is called `gcopy`. Its syntax is:

```
GEN gcopy(GEN x)
```

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 `GEN shallowcopy(GEN x)` for this. Note that the result is not suitable for `gc_upto` !

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

4.4.4 Clones. Sometimes, it is more efficient to create a *persistent* copy of a PARI object. This is not created on the stack but on the heap, hence unaffected by garbage collection functions. The function which does this is called `gclone`. Its syntax is:

```
GEN gclone(GEN x)
```

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

```
void guncclone(GEN x)
```

No PARI object should keep references to a clone which has been destroyed!

4.4.5 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 precision DEFAULTPREC = 64 bits, although only 53 of them are significant
(the others are set to 0).
```

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 not `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* attached 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 4.4.4). 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`.

`void lg_increase(GEN z)` increase the length of z by 1; you should not have to use this function if you use `cgetg`.

`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 nonzero. 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

$$|z| = \sum_{i=0}^n z_i B^i, \quad \text{where } z_n \neq 0 \text{ and } B = 2^{\text{BITS_IN_LONG}}.$$

With these notations, n is `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

`void int_normalize(GEN z, long known0)`

normalizes in place a nonnegative 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:

```
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).

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.

`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 a real number of type `t_INT`, `t_FRAC` or `t_REAL`, an infinity (`t_INFINITY`) or a `t_QUAD` of positive discriminant.

The real zero is characterized by having its sign equal to 0. If `z` is not equal to 0, then it is represented as $\pm 2^e M$, where e is the exponent, and $M \in [1, 2[$ is the (normalized) significand of `z`, whose bits are stored in `z[2], ..., z[lg(z) - 1]`.

More precisely, let m be the integer $(z[2], \dots, z[lg(z)-1])$ in base $2^{\text{BITS_IN_LONG}}$; here, `z[2]` is the most significant longword and its most significant bit is 1. Then M is the rational number $m/2^{\text{bit_accuracy}(\lg(z))-1-\text{expo}(z)}$.

`long realprec(GEN x)` returns the precision of a `t_REAL` in bits, which is, which is the total number of bits which can *fit* in this `t_REAL`. This is 0 for a real 0 and a positive multiple of `BITS_IN_LONG` otherwise. There is potential for confusion here: many functions take a `prec` or `bitprec` argument to indicate the bit precision required in the output. If this requested bit precision is not a multiple of `BITS_IN_LONG`, then the number of correct bits in `t_REAL` components of the output will be *lower* than what `realprec` indicates. You will know the actual bit precision because it was given in the function call but there is no way to recover it exactly from a `t_REAL` output: `realprec` returns an upper bound (and the difference is less than `BITS_IN_LONG`).

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

Thus, the real number 3.5 to accuracy $\text{realprec}(z)$ is represented as $z[0]$ (encoding $\text{type} = \text{t_REAL}$, $\lg(z)$), $z[1]$ (encoding $\text{sign} = 1$, $\text{expo} = 1$), $z[2] = 0\text{xe}0000000$, $z[3] = \dots = z[\lg(z) - 1] = 0\text{x}0$.

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 $A \pmod{T}$ in $\mathbf{F}_p[X]/T$.
- **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 $A \pmod{T}$ in $\mathbf{F}_l[X]/T$.
- **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 $A \pmod{T}$ in $\mathbf{F}_2[X]/T$.

4.5.6 Type 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 t_INT, t_REAL or t_FRAC. For historical reasons t_INTMOD and 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 t_POLMOD construction.

4.5.7 Type 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:

`long precp(GEN z)` returns the p -adic precision of z . This is 0 if $z = 0$.

`void setprecp(GEN z, long d)` sets the p -adic precision of z to d .

`long valp(GEN z)` returns the p -adic valuation of z (i.e. the exponent). This is defined even if z is equal to 0.

`void setvalp(GEN z, long e)` sets the p -adic valuation of z to e .

The following accessors are available:

- `padic_p(z)`: the prime p .
- `padic_pd(z)`: p^d , where d is the p -adic precision
- `padic_u(z)`: the p -unit u modulo p^d such that $z = p^v(u + O(p^d))$ or 0 (if and only if z is 0).

4.5.8 Type `t_QUAD` (quadratic number). `z[1]` points to the canonical polynomial P defining the quadratic field (as output by `quadpoly`), `z[2]` to the “real part” and `z[3]` to the “imaginary part”. The latter are of type `t_INT`, `t_FRAC`, `t_INTMOD`, or `t_PADIC` and are to be taken as the coefficients of `z` with respect to the canonical basis $(1, X)$ of $\mathbf{Q}[X]/(P(X))$. Exact complex numbers may be implemented as quadratics, but `t_COMPLEX` is in general more versatile (`t_REAL` components are allowed) and more efficient.

Operations involving a `t_QUAD` and `t_COMPLEX` are implemented by converting the `t_QUAD` to a `t_REAL` (or `t_COMPLEX` with `t_REAL` components) to the accuracy of the `t_COMPLEX`. As a consequence, operations between `t_QUAD` and *exact* `t_COMPLEX`s are not allowed.

4.5.9 Type `t_POLMOD` (polmod). as for `t_INTMOD`s, `z[1]` points to the modulus, and `z[2]` to a polynomial representing the class of `z`. Both must be of type `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 4.6) 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.

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, 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 nonzero sign, `degpol`, `leading_coeff`, `constant_coeff`, return its degree, and a pointer to the leading, resp. constant, coefficient with respect to the main variable. Note that this is a *shallow* construction: no copy is made on the PARI stack so the returned value is not safe for a basic `gc_upto` 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 10.7.

`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 nonrational 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 `0(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

```
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

`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 function 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 `lg(x)` is 2 or `degpol(x) < 0`, although both tests are valid when the coefficient types are under control: for instance, when they are guaranteed to be `t_INTs` or `t_FRACs`. The same remark applies to `t_SERs`.

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**, **varn**, **setvarn** as for polynomials. Beware: do *not* use **expo** and **setexpo** on power series.

`long valser(GEN z)` returns the valuation of `z`. This is defined even if `z` is equal to 0.

`void setvalser(GEN z, long e)` sets the valuation of `z` to `e`.

The coefficients `z[2]`, `z[3]`, ... `z[lg(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.) A series whose coefficients are integers equal to zero is represented as $O(x^n)$ (`zeroser(vx, n)`). A series whose coefficients are exact zeroes, but not all of them integers (e.g. an `t_INTMOD` such as `Mod(0,2)`) is represented as $z * x^{n-1} + O(x^n)$, where `z` is the 0 of the base ring, as per `Rg_get_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 priority at least as high as that of the numerator. The numerator n is not an exact 0 and $(n, d) = 1$ (see `gred_rfac2`).

4.5.13 Type `t_QFB` (binary quadratic form). `z[1]`, `z[2]`, `z[3]` point to the three coefficients of the form, and `z[4]` point to the form discriminant. All four are of type `t_INT`.

4.5.14 Type `t_VEC` and `t_COL` (vector). `z[1], z[2], ... z[lg(z)-1]` point to the components of the vector.

4.5.15 Type `t_MAT` (matrix). `z[1], z[2], ... z[lg(z)-1]` point to the column vectors of `z`, i.e. they must be of type `t_COL` and of the same length.

4.5.16 Type `t_VECSMALL` (vector of small integers). `z[1], z[2], ... z[lg(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.

4.5.17 Type `t_STR` (character string).

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

4.5.18 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 `pari_err`). The subsequent words `z[2], ... z[lg(z)-1]` are GENs containing additional data, depending on the error type.

4.5.19 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 12.1.

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

`long closure_is_variadic(GEN C)` returns 1 if the closure `C` is variadic, 0 else.

4.5.20 Type `t_INFINITY` (infinity).

This type has a single `t_INT` component, which is either 1 or -1 , corresponding to $+\infty$ and $-\infty$ respectively.

`GEN mkmoo()` returns $-\infty$

`GEN mkoo()` returns ∞

`long inf_get_sign(GEN x)` returns 1 if x is $+\infty$, and -1 if x is $-\infty$.

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 nonnegative exponent (bit-ORing the biased exponent to the codeword), obtained by adding a constant to the true exponent: either `HIGHEXPBIT` (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. 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.

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 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.

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 and 1 (the variables `x` and `y` under `gp`), and $2, \dots, 9$ (printed as `t2`, $\dots$ `t9`), with decreasing priority.

4.6.2.1 User variables. When the program starts, `x` (number 0) and `y` (number 1) are the only available variables, numbers 2 to 9 (decreasing priority) are reserved for building polynomials with predictable priorities.

To define further ones, you may use

```
GEN varhigher(const char *s)
```

```
GEN varlower(const char *s)
```

to recover a monomial of degree 1 in a new variable, which is guaranteed to have higher (resp. lower) priority than all existing ones at the time of the function call. The variable is printed as s , but is not part of GP’s interpreter: it is not a symbol bound to a value.

* 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).

On the other hand

`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.

```
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. This function is mostly useless since it returns a variable with unpredictable priority. Don't use it to create new variables.

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

```
GEN gy = gp_read_str("'y"); /* returns pol_x(v), for some v */
long v = varn(gy);
```

But please note the quote `'y` in the above. Using `gp_read_str("y")` might work, but is dangerous, especially when programming functions to be used under `gp`. The latter reads the value of `y`, as *currently* known by the `gp` interpreter, possibly creating it in the process. But if `y` has been modified by previous `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 `gp` variable (e.g. `gen_1`).

`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 `pol_x(v)` if assignments have been performed in the interpreter.

4.6.2.2 Temporary variables. You can create temporary variables using

`long fetch_var()` returns a new variable with *lower* priority than any variable currently in use.

`long fetch_var_higher()` returns a new variable with *higher* priority than any variable currently in use.

After the statement `v = fetch_var()`, you can use `pol_1(v)` and `pol_x(v)`. The variables created in this way have no identifier assigned to them though, and are printed as `tnumber`. 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 to build polynomials and power series, 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 garbage collection). 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 none remain). 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.

4.6.3 Comparing variables.

Let us go back to `varncmp`. There is an interesting corner case, when one of the compared variables (from `gvar`, say) is `NO_VARIABLE`. In this case, `varncmp` declares it has lower priority than any other variable; of course, comparing `NO_VARIABLE` with itself yields 0 (same priority);

In addition to `varncmp` we have

`long varnmax(long v, long w)` given two variable numbers (possibly `NO_VARIABLE`), returns the variable with the highest priority. This function always returns a valid variable number unless it is comparing `NO_VARIABLE` to itself.

`long varnmin(long x, long y)` given two variable numbers (possibly `NO_VARIABLE`), returns the variable with the lowest priority. Note that when comparing a true variable with `NO_VARIABLE`, this function returns `NO_VARIABLE`, which is not a valid variable number.

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 several powerful high level functions which enable you to input your objects as if you were under `gp`. In fact, it *is* essentially the GP syntactical parser.

There are two similar functions available to parse a string:

```
GEN gp_read_str(const char *s)
```

```
GEN gp_read_str_multiline(const char *s, char *last)
```

Both functions read the whole string `s`. The function `gp_read_str` ignores newlines: it assumes that the input is one expression and returns the result of this expression.

The function `gp_read_str_multiline` processes the text in the same way as the GP command `read`: newlines are significant and can be used to separate expressions. The return value is that of the last nonempty expression evaluated.

In `gp_read_str_multiline`, if `last` is not `NULL`, then `*last` receives the last character from the *filtered* input: this can be used to check if the last character was a semi-colon (to hide the output in interactive usage). If (and only if) the input contains no statements, then `*last` is set to 0.

For both functions, `gp`'s metacommands *are* recognized.

Two variants allow to specify a default precision while evaluating the string:

`GEN gp_read_str_prec(const char *s, long prec)` As `gp_read_str`, but set the bit precision to `prec` while evaluating `s`.

`GEN gp_read_str_bitprec(const char *s, long bitprec)` identical to `gp_read_str_prec`. DEPRECATED.

Note. The obsolete form

`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

`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)`. Return `NULL` when there are no more expressions to read (we reached EOF).

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

`GEN gp_read_file(char *name)`

As usual, the return value is that of the last nonempty 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_putc(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

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

```
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

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

```
pari_printf("%Ps\n", x);
pari_flush();
```

`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 `%Pd` and obtain the same (or better) effect with

```
pari_printf("%Pd\n", x);
pari_flush();
```

A variant like `%5Pd` would improve alignment by imposing 5 chars for each coefficient. Similarly if all entries are to be converted to floats, a format like `%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 `gp`. In most programs, this is not a concern and it is more flexible to write to an explicit `FILE*`, or to recover a character string:

`void pari_fprintf(FILE *file, const char *fmt, ...)` writes the remaining arguments to stream `file` according to the format specification `fmt`.

`char* pari_sprintf(const char *fmt, ...)` produces a string from the remaining arguments, according to the PARI format `fmt` (see `printf`). This is the `libpari` equivalent of `strprintf`, and returns a `malloc`'ed string, which must be freed by the caller. Note that contrary to the analogous `sprintf` in the `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 `asprintf`, although the latter has a different interface.

Simple variants of `pari_sprintf` convert a `GEN` to a `malloc`'ed ASCII string, which you must still `free` after use:

`char* GENTostr(GEN x)`, using the current default output format (`prettymat` by default).

`char* GENToTeXstr(GEN x)`, suitable for inclusion in a `TeX` file.

Note that we have `va_list` analogs of the functions of `printf` type seen so far:

```
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)
```

4.7.3 Errors.

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

```
pari_err(e_MISC, "error message");
```

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

```
pari_err(e_MISC, format, ...);
```

where `format` describes the format to use to write the remaining operands, as in the `pari_printf` function. For instance:

```
pari_err(e_MISC, "x[%d] = %Ps is not invertible!", i, gel(x,i));
```

The simple syntax seen above is just a special case with a constant format and no remaining arguments. The general syntax is

```
void pari_err(numerr, ...)
```

where `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 GEN with internal type `t_STR`, say, `pari_err(e_TYPE, "extgcd", x)` prints the message:

```
*** incorrect type in extgcd (t_STR),
```

See Section 11.4 for details. In the libpari code itself, the general-purpose `e_MISC` is used sparingly: it is so flexible that the corresponding error contexts (`t_ERROR`) become hard to use reliably. Other more rigid error types are generally more useful: for instance the error context attached to the `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

`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

```
pari_warn(warner, "Strategy 1 failed. Trying strategy 2")
```

which is the exact equivalent of `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 `gp`, whereas a simple `printf` would not.

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

```
pari_warn(warnprec, "bnfinit", newprec);
pari_warn(warnmem, "bnfinit");
pari_warn(warnfile, "close", "afile"); /* error when closing "afile" */
```

4.7.5 Debugging output.

For debugging output, you can use the standard output functions, `output` and `pari_printf` mainly. Corresponding to the `gp` metacommand `\x`, you can also output the hexadecimal tree attached to an object:

`void dbgGEN(GEN x, long nb = -1)`, displays the recursive structure of `x`. If `nb = -1`, the full structure is printed, otherwise the leaves (nonrecursive components) are truncated to `nb` words.

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

```
define oo
  call output((GEN)$arg0)
end
define xx
  call dbgGEN($arg0,-1)
end
```

Typing `i x` at a breakpoint in `gdb` then prints the value of the `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**) 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`. Note that **DEBUGLEVEL** is redefined in each code module, attaching it to a particular debug domain (see `setdebug`).

`void setalldbg(long L)` sets all **DEBUGLEVEL** incarnations (all debug domains) to `L`.

`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. Assume `T` was started by `timer_start`.

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

`void walltimer_start(pari_timer *T)` start a timer, as if it had been started at the Unix epoch (see `getwalltime`).

`long walltimer_delay(pari_timer *T)` returns the number of milliseconds elapsed since the timer was last checked. Assume `T` was started by `walltimer_start`.

`long walltimer_get(pari_timer *T)` returns the number of milliseconds elapsed since the timer was last reset. Does *not* reset the timer. Assume `T` was started by `walltimer_start`.

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

```
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:

```
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 nonreentrant 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)` attached to a private timer `T`.

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`, `forqfvec` and `for subgroup` are currently not implemented as an iterator but as a procedure with callbacks.

`void forell(void *E, long fun(void*, GEN), GEN a, GEN b, long flag)` goes through the same curves as `forell(ell,a,b,,flag)`, calling `fun(E, ell)` for each curve `ell`, stopping if `fun` returns a nonzero value.

`void forqfvec(void *E, long (*fun)(void *, GEN, GEN, double), GEN q, GEN b)`
: Evaluate `fun(E,U,v,m)` on all v such that $q(Uv) < b$, where U is a `t_MAT`, v is a `t_VECSMALL` and $m = q(v)$ is a `C double`. The function `fun` must return 0, unless `forqfvec` should stop, in which case, it should return 1.

`void forqfvec1(void *E, long (*fun)(void *, GEN), GEN q, GEN b)`: Evaluate `fun(E,v)` on all v such that $q(v) < b$, where v is a `t_COL`. The function `fun` must return 0, unless `forqfvec` should stop, in which case, it should return 1.

`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 nonzero value.

- `forprime` and `forprimestep`, iterators over primes and primes in arithmetic progressions, for which we refer you to the next subsection.

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

`int forcomposite_init(forcomposite_t *T, GEN a, GEN b)` initialize an iterator T over composite integers in $[a, b]$; over composites $\geq a$ if $b = \text{NULL}$. We must have $a \geq 0$. 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 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

`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.

`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 v are 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:

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

It is also available as an iterator:

`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.

`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.

In both cases, the partition must be used or copied before the next call since it is returned from a state array which will be modified in place. 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.

- `forperm` to loop over permutations of k . See the documentation of the `forperm` GP function for details. This function is available as an iterator:

`void forperm_init(forperm_t *T, GEN k)` initializes an iterator over the permutations of k (`t_INT`, `t_VEC` or `t_VECSMALL`).

GEN `forperm_next(forperm_t *T)` returns the next permutation as a `t_VECSMALL` or `NULL` when all permutations have been exhausted. The permutation must be used or copied before the next call since it is returned from a state array which will be modified in place.

- `forsubset` to loop over subsets. See the documentation of the `forsubset` GP function for details. This function is available as two iterators:

```
void forallsubset_init(forsubset_t *T, long n)
```

```
void forksubset_init(forsubset_t *T, long n, long k)
```

It is also available in generic form:

```
void forsubset_init(forsubset_t *T, GEN nk) where nk is either a t_INT n or a t_VEC with two integral components [n, k].
```

In all three cases, GEN `forsubset_next(forsubset_t *T)` returns the next subset as a `t_VECSMALL` or `NULL` when all subsets have been exhausted.

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 `ulongs`, the other as GEN. They are initialized as follows, where we expect to run over primes $\geq a$ and $\leq b$:

```
int u_forprime_init(forprime_t *T, ulong a, ulong b) for the ulong variant, where b = ULONG_MAX means we will run through all primes representable in a ulong type.
```

```
int forprime_init(forprime_t *T, GEN a, GEN b) for the GEN variant, where b = NULL means  $+\infty$ .
```

```
int forprimestep_init(forprime_t *T, GEN a, GEN b, GEN q) initialize an iterator T over primes in an arithmetic progression,  $p \geq a$  and  $p \leq b$  (where  $b = NULL$  means  $+\infty$ ). The argument q is either a t_INT ( $p \equiv a \pmod{q}$ ) or a t_INTMOD Mod(c, N) and we restrict to that congruence class.
```

All 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 and the next suitable prime as a t_INT otherwise.
```

```
ulong u_forprime_next(forprime_t *T), returns 0 if no more primes are available in the interval and fitting in an ulong and the next suitable prime otherwise.
```

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

```
forprime_t T;
GEN p;
pari_sp av = avma, av2;
forprime_init(&T, gen_2, stoi(1000));
av2 = avma;
```

```

while ( (p = forprime_next(&T)) )
{
    ...
    if ( prime_is_OK(p) ) break;
    set_avma(av2); /* delete garbage accumulated in this iteration */
}
set_avma(av); /* delete all */

```

Of course, the final `set_avma(av)` could be replaced by a `gc_upto` call. Beware that swapping the `av2 = avma` and `forprime_init` call would be incorrect: the first `set_avma(av2)` would delete the `forprime_t` structure!

4.8.3 Parallel iterators.

These iterators loop over the values of a `t_CLOSURE` taken at some data, where the evaluations are done in parallel.

- **parfor.** To initialize the analog of `parfor(i = a, b, ...)`, call

`void parfor_init(parfor_t *T, GEN a, GEN b, GEN code)` initialize an iterator over the evaluation of `code` on the integers between *a* and *b*.

`GEN parfor_next(parfor_t *T)` returns a `t_VEC [i,code(i)]` where *i* is one of the integers and `code(i)` is the evaluation, `NULL` when all data have been exhausted. Once it happens, `parfor_next` must not be called anymore with the same initialization.

`void parfor_stop(parfor_t *T)` needs to be called when leaving the iterator before `parfor_next` returned `NULL`.

The following returns an integer $1 \leq i \leq N$ such that `fun(i)` is not zero, or `NULL`.

```

GEN
parfirst(GEN fun, GEN N)
{
    parfor_t T;
    GEN e;
    parfor_init(&T, gen_1, N, fun);
    while ((e = parfor_next(&T)))
    {
        GEN i = gel(e,1), funi = gel(e,2);
        if (!gequal0(funi))
        { /* found: stop the iterator and return the index */
            parfor_stop(&T);
            return i;
        }
    }
    return NULL; /* not found */
}

```

- **parforeach.** To initialize the analog of `parforeach(V, X, ...)`, call

`void parforeach_init(parforeach_t *T, GEN V, GEN code)` initialize an iterator over the evaluation of `code` on the components of *V*.

GEN `parforeach_next(parforeach_t *T)` returns a `t_VEC [V[i],code(V[i])]` where $V[i]$ is one of the components of V and `code(V[i])` is the evaluation, NULL when all data have been exhausted. Once it happens, `parforprime_next` must not be called anymore with the same initialization.

`void parforeach_stop(parforeach_t *T)` needs to be called when leaving the iterator before `parforeach_next` returned NULL.

- `parforstep`. To initialize the analog of `parforstep(i = a, b, s, ...)`, call

`void parforstep_init(parforstep_t *T, GEN a, GEN b, GEN s, GEN code)`

initialize an iterator over the evaluation of `code` between a and b , with steps s (see `forstep` documentation for the possibilities for s).

GEN `parforstep_next(parforstep_t *T)`

returns a `t_VEC [x,code(x)]` where x is one of the evaluation points and `code(x)` is the evaluation, NULL when all data have been exhausted. Once it happens, `parforstep_next` must not be called anymore with the same initialization.

`void parforstep_stop(parforstep_t *T)` needs to be called when leaving the iterator before `parforstep_next` returned NULL.

- `parforprime`. To initialize the analog of `parforprime(p = a, b, ...)`, call

`void parforprime_init(parforprime_t *T, GEN a, GEN b, GEN code)` initialize an iterator over the evaluation of `code` on the prime numbers between a and b .

- `parforprimestep`. To initialize the analog of `parforprimestep(p = a, b, q, ...)`, call

`void parforprimestep_init(parforprime_t *T, GEN a, GEN b, GEN q, GEN code)` initialize an iterator over the evaluation of `code` on the prime numbers between a and b in the congruence class defined by q .

GEN `parforprime_next(parforprime_t *T)` returns a `t_VEC [p,code(p)]` where p is one of the prime numbers and `code(p)` is the evaluation, NULL when all data have been exhausted. Once it happens, `parforprime_next` must not be called anymore with the same initialization.

`void parforprime_stop(parforprime_t *T)` needs to be called when leaving the iterator before `parforprime_next` returned NULL.

- `parforvec`. To initialize the analog of `parforvec(X = V, ..., flag)`, call

`void parforvec_init(parforvec_t *T, GEN V, GEN code, long flag)` initialize an iterator over the evaluation of `code` on the vectors specified by V and `flag`, see `forvec` for detail.

GEN `parforvec_next(parforvec_t *T)` returns a `t_VEC [v,code(v)]` where v is one of the vectors and `code(v)` is the evaluation, NULL when all data have been exhausted. Once it happens, `parforvec_next` must not be called anymore with the same initialization.

`void parforvec_stop(parforvec_t *T)` needs to be called when leaving the iterator before `parforvec_next` returned NULL.

4.8.4 parallel data processing.

Below `f` and `g` are `t_CLOSURE`s of arity 1.

`GEN gen_parapply(GEN f, GEN V)` applies `f` to each components of `V` in parallel. Used to implement `parapply`.

`GEN gen_parapply_percent(GEN f, GEN V, long percent)` as `gen_parapply`, if `percent` not zero, print percentage of completion every `percent` steps. In `libpari`, `percent` should be 0 when `DEBUGLEVEL` is 0.

`GEN gen_parapply_slice(GEN g, GEN V, long m)` splits `V` in at most `m` chunks in arithmetic progressions, and calls `g` on each in parallel, and reassembles the result. `g` needs to apply `f` to each components of the chunk. The result is identical to `gen_parapply(f, V)` with a lower overhead. The usual choice of `m` is `mt_nbthreads()`.

`GEN gen_parapply_slice_zv(GEN g, GEN V, long m)` As `gen_parapply_slice`, but `g` must return `t_VECSMALL` and the output is a `t_VECSMALL` of long (`zv`).

`GEN gen_parpairwiseop(GEN h, GEN V)` `h` being a `t_CLOSURE` of arity 2, compute the vector $[h(V[1], V[2]), h(V[3], V[4]), \dots]$ by apply `h` in parallel.

`GEN gen_parpairwiseop(GEN h, GEN V, long percent)` as `gen_parpairwiseop`, if `percent` not zero, print percentage of completion every `percent` steps. In `libpari`, `percent` should be 0 when `DEBUGLEVEL` is 0.

4.8.5 Numerical analysis.

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

```
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 $f_t : x \rightarrow (x + t)^2$ via

```
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

```
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

```
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 11.4 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

`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

```
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)` error *i*-th component, as documented in Section 11.4.

For instance

```
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: noninvertible 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 **gcdext**.

Let x, y two integers and initially $\begin{pmatrix} s_x & s_y \\ t_x & t_y \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, so that

$$\begin{pmatrix} s_x & s_y \\ t_x & t_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}.$$

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

$$s_x x + s_y y = \text{gcd}(x, y).$$

In practice, there is no need to update s_y and t_y since $\text{gcd}(x, y)$ 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

```

/*
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&&", "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 );
    return gc_all(av, 3, &a, U, V);
}

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;
}

```

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;
    ...
    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 (gc_needed(av,1))
            (void)gc_all(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 `gc_upto` 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 GEN returned by a shallow function is called *not gc-safe*: one should use `gc_GEN` or `gc_all` to recover it when collecting garbage.

A function is *not stack-clean* if it leaves intermediate data on the stack besides its output, for efficiency reasons. If it is nevertheless suitable for `gc_upto`, the function description will state it. Of course `gc_GEN` and `gc_all` are always able to handle it.

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 `pari_init_primes` 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.

`INIT_noINTGMPm`: do not install PARI-specific GMP memory functions. This option is ignored when the GMP library is not in use. You may install PARI-specific GMP memory functions later by calling

```
void pari_kernel_init(void)
```

and restore the previous values using

```
void pari_kernel_close(void)
```

This option should not be used without a thorough understanding of the problem you are trying to solve. The GMP memory functions are global variables used by the GMP library. If your program is linked with two libraries that require these variables to be set to different values, conflict ensues. To avoid a conflict, the proper solution is to record their values with `mp_get_memory_functions` and to call `mp_set_memory_functions` to restore the expected values each time the code switches from using one library to the other. Here is an example:

```
void *(*pari_alloc_ptr) (size_t);
void *(*pari_realloc_ptr) (void *, size_t, size_t);
void (*pari_free_ptr) (void *, size_t);
void *(*otherlib_alloc_ptr) (size_t);
void *(*otherlib_realloc_ptr) (void *, size_t, size_t);
void (*otherlib_free_ptr) (void *, size_t);

void init(void)
{
    pari_init(8000000, 500000);
    mp_get_memory_functions(&pari_alloc_ptr,&pari_realloc_ptr,
                          &pari_free_ptr);

    otherlib_init();
    mp_get_memory_functions(&otherlib_alloc_ptr,&otherlib_realloc_ptr,
                          &otherlib_free_ptr);
}

void function_that_use_pari(void)
{
```

```

    mp_set_memory_functions(pari_alloc_ptr, pari_realloc_ptr,
                           pari_free_ptr);
    /*use PARI functions*/
}
void function_that_use_otherlib(void)
{
    mp_set_memory_functions(otherlib_alloc_ptr, otherlib_realloc_ptr,
                           otherlib_free_ptr);
    /*use OTHERLIB functions*/
}

```

`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. `INIT_noINTGMPm`: do not restore GMP memory functions.

`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_init_primes(ulong maxprime, ulong factorlimit)` Initialize the PARI primes, according to `maxprime` and `factorlimit`, either of which may be 0 (use a sensible default value). The parameters are updated then stored in `GP_DATA`: if is guaranteed on output that `maxprime` is less than or equal to `factorlimit` (it is silently increased to match it, if needed). See `initprimetable` for details.

This function is called by `pari_init(..., maxprime)`. It is provided for users calling `pari_init_opts` with the flag `INIT_noPRIMEm`.

`void pari_sighandler(int signum)` the actual signal handler that PARI uses. This can be used as argument to `pari_sig_init` or `signal(2)`.

`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)` forks a PARI daemon, detaching from the main process group. The function returns 1 in the parent, and 0 in the forked son.

`void paristack_setsize(size_t rsize, size_t vsize)` sets the default `parisize` to `rsize` and the default `parisizemax` to `vsize`, and reallocate the stack to match these value, destroying its content. Generally used just after `pari_init`.

`void paristack_resize(ulong newsize)` changes the current stack size to `newsize` (double it if `newsize` is 0). The new size is clipped to be at least the current stack size and at most `parisizemax`. The stack content is not affected by this operation.

`void parivstack_reset(void)` resets the current stack to its default size `parisize`. This is used to recover memory after a computation that enlarged the stack. This function destroys the content of the enlarged stack (between the old and the new bottom of the stack). Before calling this function, you must ensure that `avma` lies within the new smaller stack.

`void paristack_newsize(ulong newsize)` (*does not return*). Library version of
`default(parisize, "newsize")`

Set the default `parisize` to `newsize`, or double `parisize` if `newsize` is equal to 0, then call `cb_pari_err_recover(-1)`.

`void parivstack_resize(ulong newsize)` (*does not return*). Library version of
`default(parisizemax, "newsize")`

Set the default `parisizemax` to `newsize` and call `cb_pari_err_recover(-1)`.

5.1.3 Notions specific to the GP interpreter.

An **entree** is the generic object attached 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`: a small integer ≥ 1 (to which group of function help do we belong, for the `?n` help menu).

`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 four modules in PARI/GP:

- general functions (`functions_basic`, known to `libpari`),
- gp-specific functions (`functions_gp`),

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

`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

```
pari_add_module(functions_gp);
```

A GP default is likewise attached to a helper routine, that is run when the value is consulted, or changed by `default0` or `setdefault`. Such routines are grouped in the module `functions_default`.

`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.

`long (*cb_pari_display_hist)(long n)` initialized to `NULL`. If set, called by `gp_embedded` in place of `gp_display_hist`.

`void (*cb_pari_init_histfile)(void)` initialized to `NULL`. Called when the `histfile` default is changed. The intent is for that callback to read the file content, append it to history in memory, then dump the expanded history to the new `histfile`.

`int (*cb_pari_is_interactive)(void)`; initialized to `NULL`.

`void (*cb_pari_quit)(long)` initialized to a no-op. Called when `gp` must evaluate the quit command.

`void (*cb_pari_start_output)(void)` initialized to `NULL`.

`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 nonzero value, the error or signal handler returns, in effect further ignoring the error or signal, otherwise it raises a fatal error. A possible simple-minded handler, used by the `gp` interpreter, is

`int gp_handle_exception(long err)` if the `breakloop` default is enabled (set to 1) and `cb_pari_break_loop` is not `NULL`, we call this routine with `err` argument and return the result.

`int (*cb_pari_err_handle)(GEN)` If not `NULL`, this routine is called with a `t_ERROR` argument from `pari_err`. If it returns a nonzero value, the error returns, in effect further ignoring the error, otherwise it raises a fatal error.

The default behavior is to print a descriptive error message (display the error), then return 0, thereby raising a fatal error. This differs from `cb_pari_handle_exception` in that the function is not called on `SIGINT` (which do not generate a `t_ERROR`), only from `pari_err`. Use `cb_pari_sigint` if you need to handle `SIGINT` as well.

The following function can be used by `cb_pari_err_handle` to display the error message.

`const char* closure_func_err()` return a statically allocated string holding the name of the function that triggered the error. Return `NULL` if the error was not caused by a function.

`int (*cb_pari_break_loop)(int)` initialized to `NULL`.

`void (*cb_pari_sigint)(void)`. Function called when we receive `SIGINT`. By default, raises

```
pari_err(e_MISC, "user interrupt");
```

A possible simple-minded variant, used by the `gp` interpreter, is

```
void gp_sigint_fun(void)
```

`void (*cb_pari_pre_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.

`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 nonzero result otherwise. May print a help message.

`void (*cb_pari_long_help)(const char *s, long n)` It is called in place of the external help, to handle ??.

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.

5.1.6 Utility functions.

`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);
```

`char* gp_filter(const char *s)` pre-processor for the GP parser: filter out whitespace and GP comments from `s`. The returned string is allocated on the PARI stack and must not be freed.

`GEN pari_compile_str(const char *s)` low-level form of `compile_str`: assumes that `s` does not contain spaces or GP comments and returns the closure attached to the GP expression `s`. Note that GP metacommands are not recognized.

`int gp_meta(const char *s, int ismain)` low-level component of `gp_read_str`: assumes that `s` does not contain spaces or GP comments and try to interpret `s` as a GP metacommand (e.g. starting by `\` or `?`). If successful, execute the metacommand and return 1; otherwise return 0. The `ismain` parameter modifies the way `\r` commands are handled: if nonzero, act as if the file contents were entered via standard input (i.e. call `switchin` and divert `pari_infile`); otherwise, simply call `gp_read_file`.

`void pari_hit_return(void)` wait for the use to enter `\n` via standard input.

`void gp_load_gprc(void)` read and execute the user's GPRC file.

`void pari_center(const char *s)` print `s`, centered.

`void pari_print_version(void)` print verbose version information.

`long pari_community(void)` return the index of the support section in the help.

`const char* gp_format_time(long t)` format a delay of t ms suitable for `gp` output, with timer set. The string is allocated in the PARI stack via `stack_malloc`.

`const char* gp_format_prompt(const char *p)` format a prompt p suitable for `gp` prompting (includes colors and protecting ANSI escape sequences for readline).

`void pari_alarm(long s)` set an alarm after s seconds (raise an `e_ALARM` exception).

`void gp_help(const char *s, long flag)` print help for s , depending on the value of *flag*:

- `h_REGULAR`, basic help (?);
- `h_LONG`, extended help (??);
- `h_APROPOS`, a propos help (??).

`const char ** gphelp_keyword_list(void)` return a NULL-terminated array of strings, containing keywords known to `gphelp` besides GP functions (e.g. `modulus` or `operator`). Used by the online help system and the contextual completion engine.

`void gp_echo_and_log(const char *p, const char *s)` given a prompt p and attached input command s , update logfile and possibly print on standard output if `echo` is set and we are not in interactive mode. The callback `cb_pari_is_interactive` must be set to a sensible value.

`void gp_alarm_handler(int sig)` the `SIGALRM` handler set by the `gp` interpreter.

`void print_fun_list(char **list, long n)` print all elements of `list` in columns, pausing (hit return) every n lines. `list` is NULL terminated.

5.1.7 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.

5.1.8 GP history.

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

`void pari_add_hist(GEN x, long t, long r)` adds x as the last history entry; t (resp. r) is the cpu (resp. real) 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 `%`, `%'`, `%''`, etc.).

`long pari_get_histtime(long p)` as `pari_get_hist`, returning the cpu time used to compute the history entry, instead of the entry itself.

`long pari_get_histrtime(long p)` as `pari_get_hist`, returning the real time used to compute the history entry, instead of the entry itself.

`GEN pari_histtime(long p)` return the vector `[cpu, real]` where `cpu` and `real` are as above.

`ulong pari_nb_hist(void)` return the index of the last entry.

`void gp_display_hist(long n)` print the history entry n using the usual GP format `"%n = ..."`.

`void str_display_hist(pari_str *S, long n)` as `gp_display_hist`, but output to the string S .

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.

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 prec)` allocates a `t_REAL` of precision `prec`. Sets `z[0]` only.

GEN `cgetc(long prec)` allocates a `t_COMPLEX` whose real and imaginary parts are `t_REALs` of precision `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 `vec trunc_init(long l)` perform `cgetg(1,t_VEC)`, then set the length to `l` 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:

void `vec trunc_append(GEN x, GEN y)` assuming `x` was allocated using `vec trunc_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);
```

Beware that the maximal size of `x` (the `l` argument to `vec trunc_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.

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.

void `vec trunc_append_batch(GEN x, GEN y)` successively apply

```
vec trunc_append(x, gel(y, i))
```

for all elements of the vector `y`.

GEN `col trunc_init(long l)` as `vec trunc_init` but perform `cgetg(1,t_COL)`.

GEN `vec small trunc_init(long l)`

void `vec small trunc_append(GEN x, long t)` analog to the above for a `t_VECSMALL` container.

5.2.2 Length conversions.

These routines convert nonnegative lengths, often attached to floating point significands, between different units. Their behavior is undefined at negative integers. Strictly speaking the *length* of the GEN x is the output of $\lg(x)$ and counts a number of words, themselves comprising BITS_IN_LONG bits. By extension, a length could also be expressed in bits or even in decimal digits. We will not use the word “length” in this context but rather *number of bits* (**nbits**) or *bit precision* and *number of decimal digits*.

A **t_REAL** has a length, allowing to store a given number of bits, which is a multiple of BITS_IN_LONG, called *maximal bit precision*. This may be larger than the number of correct bits if a precision which is not a multiple of BITS_IN_LONG was requested in the function call producing it.

long ndec2nlong(long x) converts a number of decimal digits to a number of words. Returns $1 + \text{floor}(x \log_2 10 / \text{BITS_IN_LONG})$.

long nbits2ndec(long x) converts a number of bits to a number of decimal digits. Returns $\text{floor}(x \log_{10} 2)$.

long nbits2nlong(long x) converts a number of bits to a number of words. Returns the smallest word count containing x bits, i.e., $\text{ceil}(x / \text{BITS_IN_LONG})$.

long nbits2lg(long x) converts a number of bits to a **t_REAL** length in words. This is $\text{nbits2nlong}(x) + 2$ since a **t_REAL** has 2 code words.

long nchar2nlong(long x) converts a number of bytes to number of words. Returns the smallest word count containing x bytes, i.e. $\text{ceil}(x / \text{sizeof}(\text{long}))$.

We can convert a **t_REAL** length to a bit precision without losing information. In the other direction, a conversion from bits to length (of a significand that can hold that many bits) rounds to the next multiple of BITS_IN_LONG and loses information: the original bit count can only be recovered with an error in $[0, \text{BITS_IN_LONG} - 1[$. There is no error if the bit count is known to be a multiple of BITS_IN_LONG.

long realprec(GEN x) maximal bit precision of a **t_REAL**.

long ndec2prec(long x) converts a number of decimal digits to a maximal bit precision. This is $\text{ndec2nlong}(x)$ times BITS_IN_LONG. Avoid using this function: it is used to approximately synchronize PARI's internal precision (**realbitprecision** in bits) and the decimal “pseudo-precision” (**realprecision**) provided as a convenience by **gp**'s user interface. Use bit precision throughout.

long nbits2prec(long x) convert x to the next multiple of BITS_IN_LONG. Avoid using this function, make sure your implementation works for an arbitrary bit precision, without rounding. An exception is when caching a **t_REAL** constant x for later use: computing it at a given maximal bit precision allows to check later whether the cache is sufficiently accurate by comparing the requested accuracy with **realprec**(x) (otherwise the cache accuracy would have to be maintained separately).

long bit_accuracy(long x) converts a **t_REAL** length into a number of significant bits. This is $(x - 2)\text{BITS_IN_LONG}$.

double bit_accuracy_mul(long x, double y) returns $(x - 2)(y \times \text{BITS_IN_LONG})$. This is equal to **ybit_accuracy**(x).

long precdbl(long x) return $2 * x$. Doubles a bit precision.

The following functions are *deprecated* and kept for backward compatibility. They will be removed in a future version:

```
long bit_prec(GEN x) alias for realprec.
long lg2prec(long x) alias for bit_accuracy.
long nbits2extraprec(long x) alias for nbits2prec.
long prec2nbits(long x) return  $x$ .
double prec2nbits_mul(long x, double y) returns  $x \times y$ .
long prec2ndec(long x) alias for nbits2ndec.
long ndec2nbits(long x) alias for ndec2prec.
long prec2lg(long x) alias for nbits2lg).
```

5.2.3 Read type-dependent information.

`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).

`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` and `t_QUAD` of positive discriminants. Raise a type error if `typ(x)` is not among those.

`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 gexpo_safe(GEN x)` same as `gexpo`, but returns a value strictly less than `-HIGHEXPOBIT` when the exponent is not defined (e.g. for a `t_PADIC` or `t_INTMOD` component).

`long valp(GEN x)` returns the p -adic valuation (for a `t_PADIC`).

`long valser(GEN x)` returns the X -adic valuation (for a `t_SER`).

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

`long varn(GEN x)` returns the variable number of the `t_POL` or `t_SER` x (between 0 and `MAXVARN`).

`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 nonzero (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 .

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

```
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

- if x is not zero: the bit size of its mantissa;
- if x has sign 0: $\max(1, -\text{expo}(x))$.

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(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(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 `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] = evalsigne(-1) | 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 evallg(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 _evallg(long x) as evallg *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 valuation x to bitmask (second codeword of t_PADIC). 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 evalvalser(long x) convert X -adic valuation x to bitmask (second codeword of t_SER). Raise overflow error if x is so large that the corresponding valuation cannot be represented.

long _evalvalser(long x) same as evalvalser *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 evallgefint(long x) convert effective length x to bitmask (second codeword 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 or 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.

`void togglesign_safe(GEN *x)` sets the s 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_1` by `gen_m1`).

`void setabssign(GEN x)` sets the sign s of x to $|s|$, 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 ± 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 ± 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 valuation of x to s , if x is a `t_PADIC`.

`void setvalser(GEN x, long s)` sets the X -adic valuation of x to s , if x is a `t_SER`, respectively.

`void setprecp(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 $0 \leq s \leq \text{MAXVARN}$).

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 nonrecursive types are `t_INT`, `t_REAL`, `t_STR`, `t_VECSMALL`). Somewhat contrary to intuition, `t_LIST` is also nonrecursive, ; 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_real_t(long t)` true iff t is `t_INT` or `t_REAL` or `t_FRAC`.

`int is_qfb_t(long t)` true iff t is `t_QFB`.

`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_coeff(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_coeff(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.

The following routines return the component of `t_VECSMALL` as `ulong`.

`ulong uel(GEN x, long i)` returns the entry `x[i]` as a `ulong`, assuming `x` is a `t_VECSMALL`.

`ulong ucoeff(GEN x, long i, long j)` returns the `x[j][i]` entry of `x` as a `ulong` assuming `x[i][j]` is a `t_VECSMALL`.

`ulong umael(GEN x, long i, long j)` returns the `x[i][j]` entry of `x` as a `ulong` assuming `x[i]` is a `t_VECSMALL`.

`ulong umael2(GEN A, long x1, long x2)` is an alias for `umael`. Similar macros `umael3`, `umael4`, `umael5` 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 = 2TWOPOTBITS_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 precision (`realprec`) of a `t_REAL` with 64 bits of accuracy

`long MEDDEFAULTPREC`: the precision (`realprec`) of a `t_REAL` with 128 bits of accuracy

`long BIGDEFAULTPREC`: the precision (`realprec`) 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 nonnegative remainder).

`long remsBIL(n)` returns the (nonnegative) Euclidean remainder of n by `BITS_IN_LONG`

`long dvmdsBIL(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`:

`EXP0numBITS`, `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 `prec`, `varn`, `typ`, `signe`

`CLONEBIT`, `EXPOBITS`, `LGBITS`, `PRECPBITS`, `SIGNBITS`, `TYPBITS`, `VALPBITS`, `VARNBITS`: bitmasks used to extract `isclone`, `expo`, `lg`, `prec`, `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.

`HIGHEXPBIT`: a power of 2, one more than the largest possible exponent for a `t_REAL`.

`HIGHVALPBIT`: a power of 2, one more than 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:

M_PI: π .

M_LN2: $\log 2$.

LOG10_2: $\log 2 / \log 10$.

LOG2_10: $\log 10 / \log 2$.

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 `pari_init(size, maxprime)`, a prime table `pari PRIMES` is initialized with the successive primes up to (possibly just a little beyond) `maxprime`. The prime table occupies roughly $4\text{maxprime}/\log(\text{maxprime})$ bytes in memory, so be sensible when choosing `maxprime`; it is 2^{20} by default under `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 `maxprime`. In any case, the implementation requires that `maxprime` $< 2^{\text{BITS_IN_LONG}} - 2048$, whatever memory is available. In fact, `maxprime` is automatically replaced by

`max(maxprime, 65557)` .

In particular, PARI guarantees that the first 6547 primes, up to and including 65557, are present in the table, even if you set `maxprime` to zero in the `pari_init` call.

Some convenience functions:

`ulong maxprime(void)` the largest prime computable using our prime table.

`ulong maxprimeN(void)` the index N of the largest prime computable using the prime table. I.e., $p_N = \text{maxprime}()$.

`GEN prodprimes(void)` a vector T where $T[i]$ is the product of the primes in our prime table that are smaller than 2^{7+i} . Used to quickly detect small prime divisors of large integers, using gcd's.

`GEN prodprimeslim(void)` a vector L where $L[i]$ is the largest prime less than 2^{7+i} .

`ulong maxprimelim(void)` the argument used for the last `initprimetable` call. This is at least 65537 and the largest prime less than or equal to this number is `maxprime()`.

`long PRIMES_search(ulong x)` assumes $x \leq \text{maxprimelim}()$. Return $i > 0$ such that $x = \text{pari PRIMES}[i]$ iff x is prime. Else return $-i < 0$ such that x lies (strictly) between the $(i-1)$ -th and the i -th prime; this is understood as $i = 1$ when $x < 2$: the statement about the non-existing 0-th prime is disregarded.

`void maxprime_check(ulong B)` raise an error if `maxprime()` is $< B$.

`void initprimetable(ulong maxp, ulong factorlimit)` computes the prime table `pari PRIMES` (of all primes $p < \text{maxp}$); time complexity is softly linear in `maxp`. We don't recommend values greater than 2^{32} : prime iterators produce fast primes up to `maxp`².

This function also caches data allowing fast trial division up to `factorlimit`; time complexity is also softly linear in `factorlimit` with much larger implied constants. Value greater than 2^{32}

will not generally be useful: for such values, expect to increase startup time by 15s and memory use by 10GB.

We recommend to set `factorlimit` $\leq$ `maxp`; otherwise, the parameter `factorlimit` is silently decreased to match `maxp`. A generic value of 2^{20} is quite sensible for `factorlimit`, for a negligible cost at startup (less than 50ms). Only increase this if your own code will impose *many* trial divisions up to the given limit: PARI itself uses other algorithms to detect factors larger than 2^{20} .

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_INT` of length `n`, and initializes its first codeword. Identical to `cgetg(n,t_INT)`.

`GEN cgetr(long b)` allocates memory on the stack for a `t_REAL` with `b` bits of accuracy, and initializes its first codeword. Identical to `cgetg(nbits2lg(b),t_REAL)`.

`GEN cgetc(long b)` allocates memory on the stack for a `t_COMPLEX`, whose real and imaginary parts are `t_REAL`s allocated with `cgetr(b)`.

`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 `((GEN)avma) - n`, after checking that it is larger than `(GEN)bot`.

`void new_chunk_resize(size_t x)` this function is called by `new_chunk` when the PARI stack overflows. There is no need to call it manually. It will either extend the stack or report an `e_STACK` error.

`char* stack_malloc(size_t n)` allocates memory on the stack for `n` chars (*not* `n` `GEN`s). 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 `set_avma(oldavma)` might be enough. On the other hand, beware that this is not permanent independent storage, but part of the stack. The memory is aligned on `sizeof(long)` bytes boundaries.

`char* stack_malloc_align(size_t n, long k)` as `stack_malloc`, but the memory is aligned on `k` bytes boundaries. The number `k` must be a multiple of the `sizeof(long)`.

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

`char* stack_calloc_align(size_t n, long k)` as `stack_malloc_align`, setting the memory to zero.

Objects allocated through these last three functions cannot be handled by garbage collection routines, since they are not yet valid `GEN`s: their codewords must be filled first.

`GEN cgetalloc(size_t l, long t)`, same as `cgetg(l, t)`, 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 nonrecursive 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 $C = \text{NULL}$ 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 set_avma(ulong av)` reset `avma` to `av`. You may think of this as a simple `avma = av` statement, but PARI developpers modify this statement in special code branches to detect garbage collecting issues (by invalidating the PARI stack below `av`).

`ulong get_avma(void)` return `avma`. Useful for languages that do not provide access to TLS variables.

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

`GEN gerepile(pari_sp av, pari_sp av2, GEN x)` obsolete low level garbage collector. Don't use this. In proper code, this can be replaced by `gc_upto(av, x)`

`GEN gc_GEN(pari_sp av, GEN x)` cleans up the stack from `av` on, recopying the object x to a new address which is returned.

`GEN gc_INT(pari_sp av, GEN x)` faster variant of `gc_GEN` when x is a `t_INT`. The length and effective length of the resulting `t_INT` are equal.

GEN `gc_INT_affii(pari_sp av, GEN x, GEN *z)` This routine is technical, meant to reduce stack use when we modify big structures in place, for instance in linear algebra over the integers. Let `t_INTs` `x` and `*z` be such that `*z` was created *before* `av` was recorded. Then this function sets `*z` $\leftarrow$ `x` and collects accumulated garbage. This is done as follows:

- if `*z` is on the stack and large enough to store `x`, do so using `affii`. This destroys the *content* of `z` and leave a clean stack with `avma = av`.

- else set `*z = gc_INT(av, x)`. Compared to the previous case, we cleaned up the stack from `av` on, but the old `*z` is still taking up space that couldn't be (easily) reclaimed.

In a nutshell, we perform a `gc_INT` while taking advantage of the opportunity of overwriting `*z` without growing the stack, if possible.

GEN `gc_NULL(pari_sp av)` reset `avma` to `av` and return `NULL`.

The following 6 functions reset `avma` to `av` and return `x`:

`int gc_bool(pari_sp av, int x)`

`double gc_double(pari_sp av, double x)`

`int gc_int(pari_sp av, int x)`

`long gc_long(pari_sp av, long x)`

`ulong gc_ulong(pari_sp av, ulong x)` This allows for instance to return `gc_ulong(av, itou(z))`, whereas

```

    pari_sp av = avma;
    GEN z = ...
    set_avma(av);
    return itou(z);

```

should be frowned upon since `set_avma(av)` conceptually destroys everything from the reference point on, including `z`.

GEN `gc_const(pari_sp av, GEN x)` assumes that `x` is either not on the stack (clone, universal constant such as `gen_0`) or was defined before `av`.

GEN `gc_stoi(pari_sp av, long x)` reset `avma` to `av` and return `stoi(x)`.

GEN `gc_utoi(pari_sp av, long x)` reset `avma` to `av` and return `utoi(x)`.

GEN `gc_utoipos(pari_sp av, long x)` reset `avma` to `av` and return `utoipos(x)`.

`void retgc_const(pari_sp av, GEN x)` (macro) reset the stack to `av` before evaluating `x` and returning it.

GEN `gc_all(pari_sp av, int n, ...)`. Assumes that $1 \leq n$. This functions expects `n` further `GEN*` arguments: the stack is cleaned and the corresponding `GEN` are copied to the stack starting from `av` (in this order: the first argument comes first), and the first such `GEN` is returned. To be used in the following scenario:

```

    GEN f(..., GEN *py)
    {
        pari_sp av = avma;
        GEN x = ..., y = ...
        *py = y; return gc_all(av, 2, &x, py);
    }

```

```
}
```

This function returns x , and the user also recovers y as a side effect. Not that we can later use `cgiv( $y$ )` to recover the memory used by y while still keeping x .

`GEN gc_all_unsafe(pari_sp av, pari_sp ltop, int n, ...)` unsafe variant of `gc_all`, about twice faster. This is subject to the same restrictions as `gc_upto`, the only difference being that more than one address gets updated. 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`. To be used in the following scenario:

```
GEN f(..., GEN *py)
{
    ltop = avma;
    ... junk ...
    av = avma;
    x = fun1(); // gc clean
    y = fun2(); // gc clean
    *py = y; return gc_all_unsafe(av, ltop, 2, &x, py);
}
```

This function returns x , and we also recover y as a side effect. Not that we can later use `cgiv( $y$ )` to recover the memory used by y while still keeping x .

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

`void gc_slice_unsafe(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 `gc_all_unsafe`, of which this is a variant. For instance

```
z = cgetg(3, t_COMPLEX);
av = avma; garbage(); ltop = avma;
gel(z,1) = fun1();
gel(z,2) = fun2();
gc_slice_unsafe(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

```
av = avma; garbage();
z = cgetg(3, t_COMPLEX);
gel(z,1) = fun1();
gel(z,2) = fun2(); return gc_upto(av, z);
```

`GEN gc_upto(pari_sp av, GEN q)` analogous to (but faster than) `gc_GEN`. 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 `set_avma(av)`; in particular, sentinels which are not even proper `GENs` such as `q = NULL` are allowed.

`GEN gc_leaf(pari_sp av, GEN q)` assumes only that `q` is a leaf, i.e a nonrecursive type (`is_recursive_t(typ(q))` is nonzero). Contrary to `gc_INT` and `gc_upto`, `gc_leaf` 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 garbage collection. This avoids a garbage collection which would copy objects around, but the stack space between `av` and `ltop` is not reclaimed.

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

```
av0 = avma; z = cgetg(t_VEC, 3);
gel(z,1) = HUGE(); av = avma; garbage(); ltop = avma;
gel(z,2) = HUGE(); stackdummy(av, ltop);
```

Compared to the orthodox

```
gel(z,2) = gc_upto(av, gel(z,2));
```

or even more wasteful

```
z = gc_GEN(av0, z);
```

we temporarily lose $(av - ltop)$ words but save a costly `gc_upto`. 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

```
gc_upto(..., z);
```

And even if it were made of valid GENs, inhibiting the area makes sure `gc_stack_update` 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 garbage collection problems.

`GEN gcopy_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_gc_upto(GEN x)` returns 1 if `x` is suitable for `gc_upto`, and 0 otherwise. In the latter case, print a warning explaining the problem.

`void dbg_stack(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 `gc_GEN_unsafe(..., ltop, ...)`.

`void dbg_gc_upto(GEN q)` outputs the list of all objects on the stack that would be inspected in a call to `gc_upto(..., q)`.

`void dbg_fill_stack(void)` marks the unused portion of the stack (between its bottom and `avma`) with repeated magic values: `0xBADC0FFEE0DDF00D` on 64-bit archs, and `0xDEADBEEF` on 32-bit. This allows to quickly detect garbage collection errors, e.g., objects one of whose component would be overwritten later or uninitialized memory access. The `valgrind` framework offers more thorough possibilities but requires instrumenting the code.

5.5.6 Copies.

`GEN gc_copy(GEN x)` creates a new copy of `x` on the stack.

`GEN gc_copy_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 `gc_copy(x)`.

`int isonstack(GEN x)` true iff `x` belongs to the stack.

`void copyifstack(GEN x, GEN y)` sets `y = gc_copy(x)` if `x` belongs to the stack, and `y = x` otherwise. This macro evaluates its arguments once, contrary to

```
y = isonstack(x)? gc_copy(x): x;
```

`void icopyifstack(GEN x, GEN y)` as `copyifstack` assuming `x` is a `t_INT`.

5.5.7 Simplify.

`GEN simplify(GEN x)` you should not need that function in library mode. One rather uses:

`GEN simplify_shallow(GEN x)` shallow, faster, version of `simplify`.

5.6 The PARI heap.

5.6.1 Introduction.

It is implemented as `malloc`'ed blocks of memory, equipped with reference counts. Each block has type `GEN` but need not be a valid `GEN`: it is a chunk of data preceded by a hidden header (meaning that we allocate `x` and return `x + headersize`). A *clone*, created by `gclone`, is a block which is a valid `GEN` and whose *clone bit* is set. The collection of blocks is maintained as a doubly-linked list (in creation order) *and* as an AVL tree (allowing to check in logarithmic time whether a given object / address is a block).

5.6.2 Public interface.

`GEN newblock(size_t n)` allocates a block of n words (not bytes).

`void killblock(GEN x)` deletes the block x created by `newblock`. Fatal error if x not a block.

`GEN gclone(GEN x)` creates a new permanent copy of x on the heap (allocated using `newblock`). The *clone bit* of the result is set.

`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 .

`void guncclone(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 `killblock`, but it is cleaner to kill clones (valid GENs) using this function, and other blocks using `killblock`.

`void gunccloneNULL(GEN x)` same as `guncclone`, first checking whether x is NULL (and doing nothing in this case).

`void guncclone_deep(GEN x)` is only useful in the context of the GP interpreter which may replace arbitrary components of container types (`t_VEC`, `t_COL`, `t_MAT`, `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 `guncclone`.

`void gunccloneNULL_deep(GEN x)` same as `guncclone_deep`, first checking whether x is NULL (and doing nothing in this case).

`void traverseheap(void(*f)(GEN, void*), void *data)` this applies $f(x, data)$ to each object x on the PARI heap, most recent first. Mostly for debugging purposes.

`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. Decrement in `killblock`, incremented 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 $2^{\text{BITS_IN_LONG}}$ 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. Currently variables are implemented by a single array of values divided in 3 zones: `0-nvar` (user variables), `max_aval-MAXVARN` (temporary variables), and `nvar+1-max_aval-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 `max_aval`, notwithstanding existing variables! In effect, this even deletes `x`. Don't use it.

`void pari_var_close(void)` attached destructor, called by `pari_close`.

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

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

`long pari_var_create(entree *ep)` low-level initialization of an `EpVAR`. Return the attached (new) variable number.

`GEN vars_sort_inplace(GEN z)` given a `t_VECSMALL` `z` of variable numbers, sort `z` in place according to variable priorities (highest priority comes first).

`GEN vars_to_RgXV(GEN h)` given a `t_VECSMALL` `z` of variable numbers, return the `t_VEC` of `pol_x(z[i])`.

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.

`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*` attached to an identifier `s` (variable or function), from the interpreter hashtables. Return `NULL` if the identifier is unknown.

5.7.3 Temporary variables.

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

`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.

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`.

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 `gc_upto` (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_putc(char c):` write character `c` to the output stream.

`void pari_puts(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. A *GP prototype* is a character string describing all the GP bytecode compiler needs to know about the function prototype. It contains a sequence of the following atoms:

- Return type: **GEN** by default (must be valid for `gc_upto`), otherwise the following can appear as the *first* char of the code string:

```

i      return int
l      return long
u      return ulong
v      return void
m      return a GEN which is not gc-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:

```

G      GEN
&      *GEN
L      long (we implicitly typecast int to long)
U      ulong
V      loop variable
n      variable, expects a variable number (a long, not an *entree)
W      a GEN which is a lvalue to be modified in place (for t_LIST)
r      raw input (treated as a string without quotes). Quoted args are copied as strings
        Stops at first unquoted ')' or ', '. Special chars can be quoted using '\ '
        Example: aa"b\n)"c yields the string "aab\n)c"
s      expanded string. Example: Pi"x"2 yields "3.142x2"
        Unquoted components can be of any PARI type, converted to string following
        current output format
I      inline closure whose value is ignored, as in for loops,
        to be processed by void closure_evalvoid(GEN C)
E      inline closure whose value is used, as in sum loops,
        to be processed by void closure_evalgen(GEN C)
J      closure of arity 1, as in parsum loops,
        to be processed by void closure_callgen1(GEN C)
```

Closures are **GEN** of type `t_CLOSURE`.

A *closure* is a GP function in compiled (bytecode) form. It can be efficiently evaluated using the `closure_callxxx` functions. A C function can be converted to a closure by using `strtoclosure`.

An *inline closure* is a GP expression in compiled (bytecode) form. It can be efficiently evaluated using the `closure_evalxxx` functions. A C function can be converted to a closure by using `strtoclosure`. and then to a pseudo-inline closure suitable for **I** and **E** by using `closuretoinl`.

In both cases, the arity of the closure needs to match the number of **V** atoms appearing before the code **I**, **E** or **J** in the prototype.

- Automatic arguments:

```

f      Fake *long. C function requires a pointer but we do not use the resulting long
p      current real precision in bits
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* 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 `D0,L`, stands for "this optional argument is converted to a `long`, and is 0 by default". So 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 `&`.
- DI, DE** optional closure, send NULL if argument omitted.
- DP** optional `long`, send `prec dl` if argument omitted.
- DV** optional loop variable.
- Dn** optional variable number, `-1` if omitted.
- Dr** optional raw string, send NULL if argument omitted.
- Ds** optional `char *`, send NULL if argument 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 prototype attached 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 bytecode evaluator 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 in Section 5.8.2; in case the function returns a `GEN`, it must satisfy `gc_upto` 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 prototype 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")
```

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 (or `gprc`), 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:

```
{
  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`):

```
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 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!

```
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:

```
Function:  gpname
Section:   section
C-Name:    libname
Prototype: code
Help:      some help text
```

(If the help text does not fit on a single line, continuation lines must start by a whitespace character.) 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.

`long PARI_VERSION(long M, long m, long p)` produces the version code attached 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_compileddate` 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.

In the string `paricfg_buildinfo`, the substring `"%s"` needs to be substituted by the output of the function `pari_kernel_version`.

```
const char * pari_kernel_version(void)
```

`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`.

`paricfg_gphelp`: character string. The name of an external help command for ?? (such as the `gphelp` script)

Chapter 6:

Arithmetic kernel: Level 0 and 1

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

```
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

```
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 $(\text{hiremainder} * 2^{\text{BITS_IN_LONG}}) + x$ by y and stores the remainder into `hiremainder`. An error occurs if the quotient cannot be represented by an `ulong`, i.e. if initially $\text{hiremainder} \geq y$.

`long hammingu(ulong x)` returns the Hamming weight of x , i.e. the number of nonzero bits in its binary expansion.

`long thuemorseu(ulong x)` return the parity of the number of nonzero bits in the binary expansion of x , that is, the Hamming weight modulo 2.

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 $0 \leq y \leq \text{BITS_IN_LONG}$. The global variable `hiremainder` receives the bits that were shifted out, i.e. $x \gg (\text{BITS_IN_LONG} - y)$.

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

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 $x^{-1} \bmod 2^{\text{BITS_IN_LONG}}$, assuming x is odd.

`ulong Fl_add(ulong x, ulong y, ulong m)` returns the smallest nonnegative representative of $x + y$ modulo m .

`ulong Fl_neg(ulong x, ulong m)` returns the smallest nonnegative representative of $-x$ modulo m .

`ulong Fl_sub(ulong x, ulong y, ulong m)` returns the smallest nonnegative representative of $x - y$ modulo m .

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

`ulong Fl_mul(ulong x, ulong y, ulong m)` returns the smallest nonnegative representative of xy modulo m .

`ulong Fl_double(ulong x, ulong m)` returns $2x$ modulo m .

`ulong Fl_triple(ulong x, ulong m)` returns $3x$ modulo m .

`ulong Fl_half(ulong x, ulong m)` returns z such that $2z = x$ modulo m assuming such z exists.

`ulong Fl_sqr(ulong x, ulong m)` returns the smallest nonnegative representative of x^2 modulo m .

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

`ulong Fl_invgen(ulong x, ulong m, ulong *pg)` set `*pg` to $g = \gcd(x, m)$ and return u in $(\mathbb{Z}/m\mathbb{Z})^*$ such that $xu = g$ modulo m . We have $g = 1$ if and only if x is invertible, and in this case u is its inverse.

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

`ulong Fl_powu(ulong x, ulong n, ulong m)` returns the smallest nonnegative representative of x^n modulo m .

`GEN Fl_powers(ulong x, long n, ulong p)` returns $[x^0, \dots, x^n]$ modulo m , as a `t_VECSMALL`.

`ulong Fl_sqrt(ulong x, ulong p)` returns the square root of x modulo p (smallest nonnegative representative). Assumes p to be prime, and x to be a square modulo p .

`ulong Fl_sqrtl(ulong x, ulong l, ulong p)` returns a l -th root of x modulo p . Assumes p to be prime and $p \equiv 1 \pmod{l}$, and x to be a l -th power modulo p .

`ulong Fl_sqrtn(ulong a, ulong n, ulong p, ulong *zn)` returns `ULONG_MAX` if a is not an n -th power residue mod p . Otherwise, returns an n -th root of a ; if `zn` is not `NULL` set it to a primitive m -th root of 1, $m = \gcd(p-1, n)$ allowing to compute all m solutions in $\mathbb{F}_p$ of the equation $x^n = a$. Assume that p is prime.

`ulong Fl_log(ulong a, ulong g, ulong ord, ulong p)` Let g such that $g^{\text{ord}} \equiv 1 \pmod{p}$. Return an integer e such that $a^e \equiv g \pmod{p}$. If e does not exist, the result is undefined.

`ulong Fl_order(ulong a, ulong o, ulong p)` returns the order of the $\mathbb{F}_p$ a . It is assumed that o is a multiple of the order of a , 0 being allowed (no nontrivial information).

`ulong random_Fl(ulong p)` returns a pseudo-random integer uniformly distributed in $0, 1, \dots, p-1$.

`ulong nonsquare_Fl(ulong p)` return a quadratic nonresidue modulo p , assuming p is an odd prime. If p is 3 mod 4, return $p-1$, else return the smallest (prime) nonresidue.

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

`ulong pgener_Zl(ulong p)` returns the smallest positive integer which is a primitive root modulo p^k for all $k \geq 1$, assuming p is an odd prime.

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

`ulong factorial_Fl(long n, ulong p)` return $n! \bmod p$.

6.1.3 Modular kernel with “precomputed inverse”.

This is based on an algorithm by T. Grandlund and N. Möller in “Improved division by invariant integers” <https://gmplib.org/~tege/division-paper.pdf>.

In the following, we set $B = \text{BITS_IN_LONG}$.

`ulong get_Fl_red(ulong p)` returns a pseudoinverse pi for p . Namely an integer $0 < pi < B$ such that, given $0 \leq x < B^2$ (by two long words), we can compute the Euclidean quotient and remainder of x modulo p by performing 2 multiplications and some additions. Precisely, once we set $q = 2^k p$ for the unique k such that $B/2 \leq q < B$, the pseudoinverse pi is equal to the Euclidean quotient of $B^2 - qB + B - 1$ by q . In particular $(pi + B)/B^2$ is very close to $1/q$.

Note that this algorithm is generally less efficient than ordinary quotient and remainders (`divll` or even `/` and `%`) when $0 \leq x < B$ and $p \leq B^{1/2}$ are small. High level functions below allow setting $pi = 0$ to cater for this possibility and avoid calling `get_Fl_red` for arguments where the standard algorithm is preferable.

`ulong divll_pre(ulong x, ulong p, ulong pi)` as `divll`, where pi is the pseudoinverse of p .

`ulong remll_pre(ulong u1, ulong u0, ulong p, ulong pi)` returns the Euclidean remainder of $u_1 2^B + u_0$ modulo p , assuming pi is the pseudoinverse of p . This function is faster if $u_1 < p$.

`ulong remlll_pre(ulong u2, ulong u1, ulong u0, ulong p, ulong pi)` returns the Euclidean remainder of $u_2 2^{2B} + u_1 2^B + u_0$ modulo p , assuming pi is the pseudoinverse of p .

`ulong Fl_sqr_pre(ulong x, ulong p, ulong pi)` returns x^2 modulo p , assuming pi is the pseudoinverse of p .

`ulong Fl_mul_pre(ulong x, ulong y, ulong p, ulong pi)` returns xy modulo p , assuming pi is the pseudoinverse of p .

`ulong Fl_addmul_pre(ulong a, ulong b, ulong c, ulong p, ulong pi)` returns $a + bc$ modulo p , assuming pi is the pseudoinverse of p .

`ulong Fl_addmulmul_pre(ulong a, ulong b, ulong c, ulong d, ulong p, ulong pi)` returns $ab + cd$ modulo p , assuming pi is the pseudoinverse of p .

`ulong Fl_powu_pre(ulong x, ulong n, ulong p, ulong pi)` returns x^n modulo p , assuming pi is the pseudoinverse of p , or 0 in which case we either use ordinary divisions if $p < B^{1/2}$ is small and call `get_Fl_red` ourselves otherwise.

`GEN Fl_powers_pre(ulong x, long n, ulong p, ulong pi)` returns the vector `(t_VECSMALL)` $(x^0, \dots, x^n)$, assuming pi is the pseudoinverse of p , or 0 in which case we either use ordinary divisions if $p < B^{1/2}$ is small and call `get_Fl_red` ourselves otherwise.

`ulong Fl_log_pre(ulong a, ulong g, ulong ord, ulong p, ulong pi)` as `Fl_log`, assuming pi is the pseudoinverse of p , or 0 in which case we either use ordinary divisions if $p < B^{1/2}$ is small and call `get_Fl_red` ourselves otherwise.

`ulong Fl_sqrt_pre(ulong x, ulong p, ulong pi)` returns a square root of x modulo p , see `Fl_sqrt`. Assume p is prime. We assume pi is the pseudoinverse of p , or 0 in which case we either use ordinary divisions if $p < B^{1/2}$ is small and call `get_Fl_red` ourselves otherwise.

`ulong Fl_sqrtl_pre(ulong x, ulong l, ulong p, ulong pi)` returns a l -th root of x modulo p , assuming p prime, $p \equiv 1 \pmod{l}$, and x to be a l -th power modulo p . We assume pi is the

pseudoinverse of p , or 0 in which case we either use ordinary divisions if $p < B^{1/2}$ is small and call `get_Fl_red` ourselves otherwise.

`ulong Fl_sqrtn_pre(ulong x, ulong n, ulong p, ulong pi, ulong *zn)` See `Fl_sqrtn`, assuming pi is the pseudoinverse of p , or 0 in which case we either use ordinary divisions if $p < B^{1/2}$ is small and call `get_Fl_red` ourselves otherwise.

`ulong Fl_2gener_pre(ulong p, ulong pi)` return a generator of the 2-Sylow subgroup of $\mathbf{F}_p^*$, to be used in `Fl_sqrt_pre_i`. We assume pi is the pseudoinverse of p , or 0 in which case we either use ordinary divisions if $p < B^{1/2}$ is small and call `get_Fl_red` ourselves otherwise.

`ulong Fl_2gener_pre_i(ulong ns, ulong p, ulong pi)` as `Fl_2gener_pre` where ns is a non-square modulo p .

`ulong Fl_sqrt_pre_i(ulong x, ulong s2, ulong p, ulong pi)` as `Fl_sqrt_pre` where $s2$ is the element returned by `Fl_2gener_pre`. We assume pi is the pseudoinverse of p , or 0 in which case we either use ordinary divisions if $p < B^{1/2}$ is small and call `get_Fl_red` ourselves otherwise.

6.1.4 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 $2p^2 < 2^{\text{BITS_IN_LONG}}$. In that case, it is possible to use ordinary operators efficiently. If $p < 2^{\text{BITS_IN_LONG}}$, 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.

```

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);
}

```

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).

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 prec)` allocates memory on the PARI stack for a `t_REAL` of precision `prec`, and initializes its first codeword. Identical to `cgetg(nbits2lg(prec), t_REAL)`.

GEN `cgetc(long prec)` allocates memory on the PARI stack for a `t_COMPLEX`, whose real and imaginary parts are `t_REALs` of precision `prec`.

GEN `real_1(long prec)` create a `t_REAL` equal to 1 with precision `prec`.

GEN `real_m1(long prec)` create a `t_REAL` equal to -1 to `prec` words of accuracy.

GEN `real_0_expo(long e)` create a `t_REAL` equal to 0 with exponent `e`.

GEN `real_0(long prec)` is a shorthand for `real_0_expo(-prec)`. It allows to initialize real constants with `real_0`, `real_1`, `...mplog2`, `mppi` ... in a uniform way.

GEN `int2n(long n)` creates a `t_INT` equal to $1 < n$ (i.e 2^n if $n \geq 0$, and 0 otherwise).

GEN `int2u(ulong n)` creates a `t_INT` equal to 2^n .

GEN `int2um1(long n)` creates a `t_INT` equal to $2^n - 1$.

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 nonnegative `t_INT`. Decimal numbers, hexadecimal numbers prefixed by `0x` and binary numbers prefixed by `0b` are allowed. 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 nonnegative `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. It is in general easier to use a direct conversion, e.g. `y = stoi(s)`, than to allocate a target of correct type and sufficient size, then assign to it:

```
GEN y = cgeti(3); affsi(s, y);
```

These functions can still be useful in complicated garbage collecting scenarios. When the type and size of the PARI objects which appear in a computation are under control, one may allocate sufficiently large objects at the beginning, use assignment statements, then simply restore `avma`.

In this section, the `z` argument in the `z`-functions must be of type `t_INT` or `t_REAL`.

`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.

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

`void affir(GEN x, GEN z)` assigns the `t_INT` `x` into the `t_REAL` `z`. Assumes that `lg(z) > 2`.

`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 affsr(long s, GEN z)` assigns the `long` `s` into the `t_REAL` `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 real number (`t_INT`, `t_FRAC` or `t_REAL`) `x` into the `t_REAL` `z`.

`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 garbage collection functions 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 (fictitious) avma is just the return value (typecast to pari_sp).

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

6.2.4 Conversions.

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

GEN itor(GEN x, long prec) converts the t_INT x to a t_REAL of $prec$ and return the latter. Assumes that $prec > 0$.

long itos(GEN x) converts the t_INT x to a long if possible, otherwise raise an exception. We consider the conversion to be possible if and only if $|x| \leq \text{LONG_MAX}$, i.e. $|x| < 2^{63}$ on a 64-bit architecture. Since the range is symmetric, the output of itos can safely be negated.

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 give an error.

ulong itou(GEN x) converts the t_INT $|x|$ to an ulong if possible, otherwise raise an exception. The conversion is possible if and only if $\text{lgfint}(x) \leq 3$.

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_lg(long s, long l) converts the long s into a t_REAL of length l and return the latter. Assumes that $l > 2$.

GEN stor(long s, long prec) converts the long s into a t_REAL of precision $prec$ and return the latter. Assumes that $prec > 0$.

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

GEN utoipos(ulong s) converts the *nonzero* ulong s into a t_INT and return the latter.

GEN utoineg(ulong s) converts the *nonzero* ulong s into the t_INT $-s$ and return the latter.

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

GEN utor(ulong s, long prec) converts the ulong s into a t_REAL of precision $prec$ and return the latter. Assumes that $prec > 0$.

GEN rtor_lg(GEN x, long l) converts the t_REAL x to a t_REAL of length l and return the latter. If $l < \text{lg}(x)$, round properly. If $l > \text{lg}(x)$, pad with zeroes. Assumes that $l > 2$.

GEN rtor(GEN x, long prec) converts the t_REAL x to a t_REAL of precision $prec$ and return the latter. If $prec < \text{realprec}(x)$, round properly. If $prec > \text{realprec}(x)$, pad with zeroes. Assumes that $prec > 0$.

The following functions are variants of mkintn with unsigned long arguments (instead of unsigned int for mkintn). As mkintn, they are mostly used to create large integer constants in a platform independent way:

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 `uu32toineg(ulong a, ulong b)` returns the GEN equal to $-(2^{32}a + b)$, assuming that $a, b < 2^{32}$ and that one of a or b is positive. 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 $+\infty$ 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 `mpround(GEN x)` as `roundr` except that x may be a `t_INT`.

GEN `mptrunc(GEN x)` as `truncr` except that x may be a `t_INT`.

GEN `diviiround(GEN x, GEN y)` if x and y are `t_INT`s, 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 shifti(GEN x, long n)` shifts the `t_INT x` by `n`.

`GEN shiftr(GEN x, long n)` shifts the `t_REAL x` by `n`.

`void 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 mantissa2nr(GEN x, long n)` given a `t_REAL x`, returns the mantissa of $x2^n$ (disregards the exponent of `x`). Equivalent to

$$\text{trunc2nr}(x, n - \text{expo}(x) + \text{realprec}(x) - 1)$$

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

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 $s[m], s[m+1], \dots s[M]$ (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.

`void shift_left(GEN t, GEN s, long m, long M, ulong f, ulong n)` shifts the mantissa

$$s[m], s[m+1], \dots s[M], f$$

left by n bits.

`void shift_right(GEN t, GEN s, long m, long M, ulong f, ulong n)` shifts the mantissa

$$f, s[m], s[m+1], \dots s[M]$$

right by n bits.

6.2.7 From t_INT to bits or digits in base 2^k and back.

GEN `binary_zv`(GEN `x`) given a t_INT `x`, return a t_VEC SMALL of bits, from most significant to least significant.

GEN `binary_2k`(GEN `x`, long `k`) given a t_INT `x`, and $k > 0$, return a t_VEC of digits of `x` in base 2^k , as t_INTs, from most significant to least significant.

GEN `binary_2k_nv`(GEN `x`, long `k`) given a t_INT `x`, and $0 < k < \text{BITS_IN_LONG}$, return a t_VEC SMALL of digits of `x` in base 2^k , as ulongs, from most significant to least significant.

GEN `bits_to_int`(GEN `x`, long `l`) given a vector `x` of `l` bits (as a t_VEC SMALL or even a pointer to a part of a larger vector, so not a proper GEN), return the integer $\sum_{i=1}^l x[i]2^{l-i}$, as a t_INT.

ulong `bits_to_u`(GEN `v`, long `l`) same as `bits_to_int`, where $l < \text{BITS_IN_LONG}$, so we can return an ulong.

GEN `fromdigitsu`(GEN `x`, GEN `B`) given a t_VEC SMALL `x` of length `l` and a t_INT `B`, return the integer $\sum_{i=1}^l x[i]B^{i-1}$, as a t_INT, where the `x[i]` are seen as unsigned integers.

GEN `fromdigits_2k`(GEN `x`, long `k`) converse of `binary_2k`; given a t_VEC `x` of length `l` and a positive long `k`, where each `x[i]` is a t_INT with $0 \leq x[i] < 2^k$, return the integer $\sum_{i=1}^l x[i]2^{k(l-i)}$, as a t_INT.

GEN `nv_fromdigits_2k`(GEN `x`, long `k`) as `fromdigits_2k`, but with `x` being a t_VEC SMALL and each `x[i]` being a ulong with $0 \leq x[i] < 2^{\min\{k, \text{BITS_IN_LONG}\}}$. Here `k` may be any positive long, and the `x[i]` are regarded as `k`-bit integers by truncating or extending with zeroes.

6.2.8 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`.

long `Z_pvalrem`(GEN `x`, GEN `p`, GEN `*r`) applied to t_INTs `x` $\neq 0$ and `p`, $|p| > 1$, returns $e := v_p(x)$. The quotient `x/pe` 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`) assume $x > 0$; returns $e := v_p(x)$ and replaces `x` by `x/pe`. 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`?), and allows to decide that `n` is prime if no prime $< p$ divides `n`. Not stack-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 factorial_lval(ulong n, ulong p)` returns $v_p(n!)$, assuming p is prime.

The following convenience functions generalize `Z_pval` and its variants to “containers” (`ZV` and `ZX`):

`long ZV_pvalrem(GEN x, GEN p, GEN *r)` x being a `ZV` (a vector of `t_INTs`), 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 ZV_pval(GEN x, GEN p)` as `ZV_pvalrem` but only returns the “valuation”.

`int ZV_Z_dvd(GEN x, GEN p)` returns 1 if p divides all components of x and 0 otherwise. Faster than testing `ZV_pval(x,p) >= 1`.

`long ZV_lvalrem(GEN x, ulong p, GEN *px)` as `ZV_pvalrem`, except that `p` is an `ulong` ($p > 1$). This function is not stack-clean.

`long ZV_lval(GEN x, ulong p)` as `ZV_pval`, except that `p` is an `ulong` ($p > 1$).

`long ZX_pvalrem(GEN x, GEN p, GEN *r)` as `ZV_pvalrem`, for a `ZX` x (a `t_POL` with `t_INT` coefficients). This function is not stack-clean.

`long ZX_pval(GEN x, GEN p)` as `ZV_pval` for a `ZX` x .

`long ZX_lvalrem(GEN x, ulong p, GEN *px)` as `ZV_lvalrem`, a `ZX` x . This function is not stack-clean.

`long ZXX_pval(GEN x, GEN p)` as `ZX_pval`, for a `ZXX` x (a `t_POL` with `ZX` coefficients).

`long ZXX_pvalrem(GEN x, GEN p, GEN *r)` as `ZX_pvalrem`, for a `ZXX` x (a `t_POL` with `ZX` coefficients). This function is not stack-clean.

`long ZXV_pval(GEN x, GEN p)` as `ZV_pval`, for a `ZXV` x (a `t_VEC` with `ZX` coefficients).

`long ZXV_pvalrem(GEN x, GEN p, GEN *r)` as `ZV_pvalrem`, for a `ZXV` x (a `t_VEC` with `ZX` coefficients). This function is not stack-clean.

`long ZX_lval(GEN x, ulong p)` as `ZX_pval`, except that `p` is an `ulong` ($p > 1$).

6.2.9 Generic unary operators. Let “*op*” be a unary operation among

- **neg**: negation ($-x$).
- **abs**: absolute value ($|x|$).
- **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 *opi*(GEN *x*) creates the result of *op* applied to the *t_INT* *x*.

GEN *opr*(GEN *x*) creates the result of *op* applied to the *t_REAL* *x*.

GEN *mpop*(GEN *x*) creates the result of *op* applied to the *t_INT* or *t_REAL* *x*.

Complete list of available functions:

GEN *absi*(GEN *x*), GEN *absr*(GEN *x*), GEN *mpabs*(GEN *x*)

GEN *negi*(GEN *x*), GEN *negr*(GEN *x*), GEN *mpneg*(GEN *x*)

GEN *sqri*(GEN *x*), GEN *sqrr*(GEN *x*), GEN *mpsqr*(GEN *x*)

GEN *absi_shallow*(GEN *x*) *x* being a *t_INT*, returns a shallow copy of $|x|$, in particular returns *x* itself when $x \geq 0$, and *negi*(*x*) otherwise.

GEN *mpabs_shallow*(GEN *x*) *x* being a *t_INT* or a *t_REAL*, returns a shallow copy of $|x|$, in particular returns *x* itself when $x \geq 0$, and *mpneg*(*x*) otherwise.

Some miscellaneous routines:

GEN *sqrs*(long *x*) returns x^2 .

GEN *sqru*(ulong *x*) returns x^2 .

6.2.10 Comparison operators.

int *cmpss*(long *s*, long *t*) compares the long *s* to the *t_long* *t*.

int *cmpuu*(ulong *u*, ulong *v*) compares the ulong *u* to the *t_ulong* *v*.

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 cmpiu(GEN x, ulong s)` compares the `t_INT` `x` to the `ulong` `s`.
`int cmpsi(long s, GEN x)` compares the `long` `s` to the `t_INT` `x`.
`int cmpui(ulong s, GEN x)` compares the `ulong` `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_INT`s `x` and `y`. The result is 1 if $x = y$, 0 otherwise.
`int equalrr(GEN x, GEN y)` compares the `t_REAL`s `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 nonzero real; two nonzero 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 equalis(GEN x, long s)` compare the `t_INT` `x` and the `long` `s`. The result is 1 if $x = y$, 0 otherwise.
`int equalsi(long s, GEN x)`
`int equaliu(GEN x, ulong s)` compare the `t_INT` `x` and the `ulong` `s`. The result is 1 if $x = y$, 0 otherwise.
`int equalui(ulong s, GEN x)`

The remaining comparison operators disregard the sign of their operands

`int absequaliu(GEN x, ulong u)` compare the absolute value of the `t_INT` `x` and the `ulong` `s`. The result is 1 if $|x| = y$, 0 otherwise. This is marginally more efficient than `equalis` even when `x` is known to be nonnegative.
`int absequalui(ulong u, GEN x)`
`int absncmpiu(GEN x, ulong u)` compare the absolute value of the `t_INT` `x` and the `ulong` `u`.
`int absncmpui(ulong u, GEN x)`
`int absncmpii(GEN x, GEN y)` compares the `t_INT`s `x` and `y`. The result is the sign of $|x| - |y|$.
`int absequalii(GEN x, GEN y)` compares the `t_INT`s `x` and `y`. The result is 1 if $|x| = |y|$, 0 otherwise.
`int absncmprr(GEN x, GEN y)` compares the `t_REAL`s `x` and `y`. The result is the sign of $|x| - |y|$.
`int absrnz_equal2n(GEN x)` tests whether a nonzero `t_REAL` `x` is equal to $\pm 2^e$ for some integer `e`.
`int absrnz_equal1(GEN x)` tests whether a nonzero `t_REAL` `x` is equal to ± 1 .

6.2.11 Generic binary operators. The operators in this section have arguments of C-type `GEN`, `long`, and `ulong`, and only `t_INT` and `t_REAL` `GEN`s are allowed. We say an argument is a real type if it is a `t_REAL` `GEN`, and an integer type otherwise. The result is always a `t_REAL` unless both `x` and `y` are integer types.

Let “*op*” be a binary operation among

- **add:** addition (`x + y`).
- **sub:** subtraction (`x - y`).
- **mul:** multiplication (`x * y`).
- **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 `t_INT`:

- **rem:** remainder (`x % y`). The result is the Euclidean remainder corresponding to **div**, i.e. its sign is that of the dividend `x`.
- **mod:** true remainder (`x % y`). The result is the true Euclidean remainder, i.e. nonnegative and less than the absolute value of `y`.

Important technical note. The rules given above fixing the output type (to `t_REAL` unless both inputs are integer types) are subtly incompatible with the general rules obeyed by PARI’s generic functions, such as `gmul` or `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 `t_INT` and `y` a `t_REAL`, then `gmul(x,y)` is always the same as `mulir(x,y)`. The exception is `x = 0`, in that case we can deduce that the result is an exact 0, so `gmul` returns `gen_0`, while `mulir` returns a `t_REAL` 0. Specifically, the one resulting from the conversion of `gen_0` to a `t_REAL` of precision `precision(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 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, such as being nonzero).

The names and prototypes of the low-level functions corresponding to *op* are as follows.

`GEN mpop(GEN x, GEN y)` applies *op* to the `t_INT` or `t_REAL` `x` and `y`. The result is a `t_INT` if both inputs are integers and a `t_REAL` otherwise.

Warning about division. the `mpdiv` function is obsolete and should not be used: it is only sensible when one argument is a `t_REAL`. Indeed, if both inputs are `t_INT` : we can't return x/y as a `t_FRAC` (not an allowed type here) or as a `t_REAL` since its accuracy can't be determined from the inputs; finally Euclidean division semantics $x = qy + r$ is not the desired one. Thus `mpdiv` will error out in this case of `t_INT` arguments, to avoid unexpected behaviour. Instead we have `divrmp(x, y)` when x is a `t_REAL` and y is a `t_INT` or `t_REAL`; and `divmpr(x, y)` when y is a `t_REAL`.

`GEN opsi(long s, GEN x)` 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(long s, GEN x)` applies *op* to the `long s` and the `t_REAL x`.

`GEN opss(long s, long t)` 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(GEN x, GEN y)` applies *op* to the `t_INTs 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(GEN x, GEN y)` applies *op* to the `t_INT x` and the `t_REAL y`.

`GEN opis(GEN x, long s)` 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 opri(GEN x, GEN y)` applies *op* to the `t_REAL x` and the `t_INT y`.

`GEN oprr(GEN x, GEN y)` applies *op* to the `t_REALs x` and y .

`GEN oprs(GEN x, long s)` 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 .

`ulong umuluu_le(ulong x, ulong y, ulong n)` multiplies x by y . Return xy if $xy \leq n$ and 0 otherwise (in particular if xy does not fit in an `ulong`).

`ulong umuluu_or_0(ulong x, ulong y)` multiplies x by y . Return 0 if xy does not fit in an `ulong`.

`GEN mului(ulong x, GEN y)` multiplies x by y .

`GEN muluui(ulong x, ulong y, GEN z)` return xyz .

`GEN muliu(GEN x, ulong y)` multiplies x by y .

`void addumului(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 $yz = 0$. Shallow function.
`GEN addmuliu_inplace(GEN x, GEN y, ulong u)` returns $x + yu$, but returns x itself and not a copy if $yu = 0$. Shallow function.
`GEN submuliu_inplace(GEN x, GEN y, ulong u)` returns $x - yu$, but returns x itself and not a copy if $yu = 0$. Shallow function.
`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 $a \leq b$.
`GEN mulu_interval_step(ulong a, ulong b, ulong s)` returns the product of all integers in $[a, b]$ congruent to a modulo s . Assume $a \leq b$ and $s > 0$.
`GEN muls_interval(long a, long b)` returns $a(a+1) \cdots b$, assuming that $a \leq b$.
`GEN invr(GEN x)` returns the inverse of the nonzero `t_REAL` x .
`GEN truedivii(GEN x, GEN y)` returns the true Euclidean quotient (with nonnegative remainder less than $|y|$).
`GEN truedivis(GEN x, long y)` returns the true Euclidean quotient (with nonnegative remainder less than $|y|$).
`GEN truedivsi(long x, GEN y)` returns the true Euclidean quotient (with nonnegative 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 $x \bmod 2^n$.
`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 addrr_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 .

6.2.12 Exact division and divisibility.

GEN `diviexact`(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 nonzero.

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. All GEN are assumed to be `t_INTs`:

```
int dvdi(GEN x, GEN y), int dvdis(GEN x, long y), int dvdiu(GEN x, ulong y),
int dvdsi(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).
```

6.2.13 Division with integral operands and `t_REAL` result.

GEN `rdivii_lg`(GEN `x`, GEN `y`, long `l`), assuming `x` and `y` are both of type `t_INT`, return the quotient x/y as a `t_REAL` of length `l` > 2 .

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` > 0 .

GEN `rdiviiz`(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 as a `t_REAL` of precision `prec`.

GEN `rdivss`(long `x`, long `y`, long `prec`), return the quotient x/y as a `t_REAL` of precision `prec`.

6.2.14 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 nonnegative 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 `ONLY_REM`, 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`.

Analogous routines `dvmdis`, `dvmdsi`, `dvmdss` 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 absdiviu_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 uabsdiviu_rem(GEN n, ulong d, ulong *r)` as `absdiviu_rem`, assuming that $|n|/d$ fits into an `ulong`.

`ulong uabsdivui_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.

`ulong ceildivuu(ulong x, ulong y)` return the ceiling of x/y .

`GEN divsi_rem(long s, GEN y, long *r)` computes the Euclidean quotient and remainder of the `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 `long` `x` by the `long` `y`. As `sdivss_rem` otherwise.

`GEN truedvmdii(GEN x, GEN y, GEN *r)`, as `dvmdii` but with a nonnegative remainder.

`GEN truedvmdis(GEN x, long y, GEN *z)`, as `dvmdis` but with a nonnegative remainder.

`GEN truedvmdsi(long x, GEN y, GEN *z)`, as `dvmdsi` but with a nonnegative remainder.

6.2.15 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 nonnegative 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 umodsu(long x, ulong y)` computes the true Euclidean remainder of the `long` `x` by a `ulong` `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 `smodsi` does not exist, since it would not always be defined: for a *negative* `x`, if the quotient is ± 1 , the result `x + |y|` would in general not fit into a `long`. Use either `umodui` or `modsi`.

These functions directly access the binary data and are thus much faster than the generic modulo functions:

`int mpodd(GEN x)` which is 1 if `x` is odd, and 0 otherwise.

`ulong Mod2(GEN x)`

`ulong Mod4(GEN x)`

`ulong Mod8(GEN x)`

`ulong Mod16(GEN x)`

`ulong Mod32(GEN x)`

`ulong Mod64(GEN x)` give the residue class of x modulo the corresponding power of 2.

`ulong umodi2n(GEN x, long n)` give the residue class of x modulo 2^n , $0 \leq n < BITS_IN_LONG$.

The following functions assume that $x \neq 0$ and in fact disregard the sign of x . There are about 10% faster than the safer variants above:

`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 *nonzero* `x`. As well,

`ulong mod2BIL(GEN x)` returns the least significant word of $|x|$, still assuming that $x \neq 0$.

6.2.16 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 `gpows(x,0)` would return `gen_1`, see the technical note in Section 6.2.11.

`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 powersr(GEN e, long n)`. Given a `t_REAL` e , return the vector v of all e^i , $0 \leq i \leq n$, where $v[i] = e^{i-1}$.

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 powrfac(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 $< 2^{\text{BITS_IN_LONG}}$, and 0 otherwise (overflow).

ulong upowers(ulong x, long n), returns $[1, x, \dots, x^n]$ as a `t_VECSMALL`. Assume there is no overflow.

GEN sqrtremi(GEN N, GEN *r), returns the integer square root S of the nonnegative `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 = 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 nonnegative `t_INT` N (rounded towards 0). This is identical to `sqrtremi(N, NULL)`.

long logintall(GEN B, GEN y, GEN *ptq) returns the floor e of $\log_y B$, where $B > 0$ and $y > 1$ are integers. If `ptq` is not `NULL`, set it to y^e . (Analogous to `logint0`, without sanity checks.)

ulong ulogintall(ulong B, ulong y, ulong *ptq) as `logintall` for `ulong` arguments.

long logint(GEN B, GEN y) returns the floor e of $\log_y B$, where $B > 0$ and $y > 1$ are integers.

ulong ulogint(ulong B, ulong y) as `logint` for `ulong` arguments.

GEN vecpowuu(long N, ulong a) return the vector of n^a , $n = 1, \dots, N$. Not stack-clean.

GEN vecpowug(long N, GEN a, long prec) return the vector of n^a , $n = 1, \dots, N$, where the powers are computed at precision `prec`. Not stack-clean.

6.2.17 GCD, extended GCD and LCM.

long cgcd(long x, long y) returns the GCD of x and y .

ulong ugcd(ulong x, ulong y) returns the GCD of x and y .

ulong ugcdiu(GEN x, ulong y) returns the GCD of x and y .

ulong ugcdui(ulong x, GEN y) returns the GCD of x and y .

GEN coprimes_zv(ulong N) return a `t_VECSMALL` T with N entries such that $T[i] = 1$ iff $(i, N) = 1$ and 0 otherwise.

long clcm(long x, long y) returns the LCM of x and y , provided it fits into a `long`. Silently overflows otherwise.

ulong ulcm(ulong x, ulong y) returns the LCM of x and y , provided it fits into an `ulong`. Silently overflows otherwise.

GEN gcdii(GEN x, GEN y), returns the GCD of the $\mathbf{t_INTs}$ x and y.

GEN lcmii(GEN x, GEN y), returns the LCM of the $\mathbf{t_INTs}$ x and y.

GEN bezout(GEN a, GEN b, GEN *u, GEN *v), returns the GCD d of $\mathbf{t_INTs}$ a and b and sets u, v to the Bezout coefficients such that $au + bv = d$.

long cbezout(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 halfgcdii(GEN x, GEN y) assuming x and y are $\mathbf{t_INTs}$, returns a 2-components $\mathbf{t_VEC}$ $[M, V]$ where M is a 2×2 $\mathbf{t_MAT}$ and V a 2-component $\mathbf{t_COL}$, both with $\mathbf{t_INT}$ entries, such that $M * [x, y] == V$ and such that if $V = [a, b]$, then $a \geq \left\lceil \sqrt{\max(|x|, |y|)} \right\rceil > b$.

GEN ZV_extgcd(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 $\mathrm{GL}_n(\mathbf{Z})$ such that $AU = [0, \dots, 0, D]$.

GEN ZV_lcm(GEN v) given a vector v of integers returns the LCM of its entries.

GEN ZV_snf_gcd(GEN v, GEN N) given a vector v of integers and a positive integer N , return the vector whose entries are the gcds $(v[i], N)$. Use case: if v gives the cyclic components for some Abelian group G of finite type, then this returns the structure of the finite groupe G/G^N .

6.2.18 Continued fractions and convergents.

GEN ZV_allpnqn(GEN x) given $x = [a_0, \dots, a_n]$ a continued fraction from `gboundcf`, $n \geq 0$, return all convergents as $[P, Q]$, where $P = [p_0, \dots, p_n]$ and $Q = [q_0, \dots, q_n]$.

6.2.19 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 $\mathbf{t_VECSMALL}$), and can only be used as an argument to `setrand`.

ulong `pari_rand`(void) returns a random $0 \leq x < 2^{\mathbf{BITS_IN_LONG}}$.

long `random_bits`(long k) returns a random $0 \leq x < 2^k$. Assumes that $0 \leq k \leq \mathbf{BITS_IN_LONG}$.

ulong `random_Fl`(ulong p) returns a pseudo-random integer in $0, 1, \dots, p - 1$.

GEN `randomi`(GEN n) returns a random $\mathbf{t_INT}$ between 0 and $n - 1$.

GEN `randomr`(long prec) returns a random $\mathbf{t_REAL}$ in $[0, 1[$, with precision `prec`.

6.2.20 Modular operations. In this subsection, all GENs are `t_INT`.

`GEN Fp_red(GEN a, GEN m)` returns a modulo m (smallest nonnegative residue). (This is identical to `modii`).

`GEN Fp_neg(GEN a, GEN m)` returns $-a$ modulo m (smallest nonnegative residue).

`GEN Fp_add(GEN a, GEN b, GEN m)` returns the sum of a and b modulo m (smallest nonnegative residue).

`GEN Fp_sub(GEN a, GEN b, GEN m)` returns the difference of a and b modulo m (smallest nonnegative residue).

`GEN Fp_center(GEN a, GEN p, GEN pov2)` assuming that `pov2` is `shifti(p,-1)` and that $-p/2 < a < p$, returns the representative of a in the symmetric residue system $] -p/2, p/2]$.

`GEN Fp_center_i(GEN a, GEN p, GEN pov2)` shallow internal variant of `Fp_center`: when a is already in the proper interval, it is returned as is, without a copy.

`GEN Fp_mul(GEN a, GEN b, GEN m)` returns the product of a by b modulo m (smallest nonnegative 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 nonnegative residue).

`GEN Fp_muls(GEN a, long b, GEN m)` returns the product of a by b modulo m (smallest nonnegative residue).

`GEN Fp_half(GEN x, GEN m)` returns z such that $2z = x$ modulo m assuming such z exists. Assume that $0 \leq x < m$. Not stack-clean, but suitable for `gc_upto`.

`GEN Fp_double(GEN x, GEN m)` return $2x$ modulo m . Assume that $0 \leq x < m$. Not stack-clean, but suitable for `gc_upto`.

`GEN Fp_sqr(GEN a, GEN m)` returns a^2 modulo m (smallest nonnegative residue).

`ulong Fp_powu(GEN x, ulong n, GEN m)` raises x to the n -th power modulo m (smallest nonnegative residue). Not stack-clean, but suitable for `gc_upto`.

`ulong Fp_pows(GEN x, long n, GEN m)` raises x to the n -th power modulo m (smallest nonnegative residue). A negative n is allowed. Not stack-clean, but suitable for `gc_upto`.

`GEN Fp_pow(GEN x, GEN n, GEN m)` returns x^n modulo m (smallest nonnegative residue).

`GEN Fp_pow_init(GEN x, GEN n, long k, GEN p)` Return a table R that can be used with `Fp_pow_table` to compute the powers of x up to n . The table is of size $2^k \log_2(n)$.

`GEN Fp_pow_table(GEN R, GEN n, GEN p)` return x^n , where R is given by `Fp_pow_init(x,m,k,p)` for some integer $m \geq n$.

`GEN Fp_powers(GEN x, long n, GEN m)` returns $[x^0, \dots, x^n]$ modulo m as a `t_VEC` (smallest nonnegative residue).

`GEN Fp_inv(GEN a, GEN m)` returns an inverse of a modulo m (smallest nonnegative 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 Fp_invgen(GEN x, GEN m, GEN *pg) set *pg to $g = \gcd(x, m)$ and return u in $(\mathbf{Z}/m\mathbf{Z})^*$ such that $xu = g$ modulo m . We have $g = 1$ if and only if x is invertible, and in this case u is its inverse.

GEN FpV_prod(GEN x, GEN p) returns the product of the components of x .

GEN FpV_inv(GEN x, GEN m) x being a vector of $\mathbf{t_INTs}$, 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 nonnegative residue). Raise an error if b is not invertible.

GEN Fp_divu(GEN a, ulong 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 $g = \gcd(a, m)$.

In the following three functions the integer parameter `ord` can be given either as a positive $\mathbf{t_INT}$ N , or as its factorization matrix faN , or as a pair $[N, faN]$. The parameter may be omitted by setting it to `NULL` (the value is then $p-1$).

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 undefined.

GEN Fp_order(GEN a, GEN ord, GEN p) returns the order of the Fp a . Assume that `ord` is a multiple of the order of a .

GEN Fp_factored_order(GEN a, GEN ord, GEN p) returns $[o, F]$, where o is the multiplicative order of the Fp a in $\mathbf{F}_p^*$, and F is the factorization of o . Assume that `ord` is a multiple of the order of a .

int Fp_issquare(GEN x, GEN p) returns 1 if x is a square modulo p , and 0 otherwise. Assume that p is a prime.

int Fp_ispower(GEN x, GEN n, GEN p) returns 1 if x is an n -th power modulo p , and 0 otherwise. Assume that p is a prime.

GEN Fp_sqrt(GEN x, GEN p) returns a square root of x modulo p (the smallest nonnegative residue), where x, p are $\mathbf{t_INTs}$, and p is assumed to be prime. Return `NULL` if x is not a quadratic residue modulo p .

GEN Fp_2gener(GEN p) return a generator of the 2-Sylow subgroup of $\mathbf{F}_p^*$. To use with `Fp_sqrt_i`. Assume that p is prime; may return `NULL` if p is proven to be composite.

GEN Fp_2gener_i(GEN ns, GEN p) as `Fp_2gener`, where ns is a non-square modulo p .

GEN Fp_sqrt_i(GEN x, GEN s2, GEN p) as `Fp_sqrt` where $s2$ is the element returned by `Fp_2gener`.

GEN Fp_sqrtn(GEN a, GEN n, GEN p, GEN *zn) returns `NULL` if a is not an n -th power residue mod p . Otherwise, returns an n -th root of a ; if zn is not `NULL` set it to a primitive m -th root of 1, $m = \gcd(p-1, n)$ allowing to compute all m solutions in $\mathbf{F}_p$ of the equation $x^n = a$. Assume that p is prime.

`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 factorization matrix. Return `NULL` if no such square root exist.

`GEN Zn_sqrtall(GEN x, GEN n)` returns all the square roots of x modulo n (possibly not prime), as a `t_VEC` of representatives, where x is a `t_INT` and n is either a `t_INT` or is given by its factorization matrix. Return `NULL` if no such square root exist.

`GEN Zn_quad_roots(GEN N, GEN B, GEN C)` solves the equation $X^2 + BX + C$ modulo N . Return `NULL` if there are no solutions. Else returns $[M, v]$ where $M \mid N$ and the `FpV` v of distinct integers (reduced, implicitly modulo M) is such that x modulo N is a solution to the equation if and only if x modulo M belongs to v . If the discriminant $B^2 - 4C$ is coprime to N , we have $M = N$ but in general M can be a strict divisor of N .

`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 nonzero. 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 nonzero `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 kroui(ulong 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 factorial_Fp(long n, GEN p)` return $n! \bmod p$.

`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. Assume that p is prime.

`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 factorization 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 factorization 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 ℓ -Sylow of $\mathbf{F}_p^*$ for every ℓ in L . In other words, $x^{(p-1)/\ell} \neq 1$ for all such ℓ . In particular, returns `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. Assume that p is prime.

`GEN rootsof1_Fp(GEN n, GEN p)` returns a primitive n -th root modulo the prime p .

`GEN rootsof1u_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 .

6.2.21 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 `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

$$f_n(x_1, \dots, x_n) = f(f_{n-1}(x_1, \dots, x_{n-1}), x_n),$$

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 .

6.2.22 Miscellaneous arithmetic functions.

`long bigomegau(ulong n)` returns the number of prime divisors of $n > 0$, counted with multiplicity.

`ulong coreu(ulong n)`, unique squarefree integer d dividing n such that n/d is a square.

`ulong coreu_fact(GEN fa)` same, where `fa` is `factoru(n)`.

`ulong corediscs(long d, ulong *pt_f)`, d (possibly negative) being congruent to 0 or 1 modulo 4, return the fundamental discriminant D such that $d = D * f^2$ and set `*pt_f` to f (if `*pt_f` not `NULL`).

`GEN coredisc2_fact(GEN fa, long s, GEN *pP, GEN *pE)` let d be an integer congruent to 0 or 1 mod 4. Return $D = \text{coredisc}(d)$ assuming that `fa` is the factorization of $|d|$ and $sd > 0$ (s is the sign of d). Set `*pP` and `*pE` to the factorization of the conductor f such that $d = Df^2$, where P is a `t_VEC` of primes and E a `t_VECSMALL` of exponents.

`ulong coredisc2u_fact(GEN fa, long s, GEN *pP, GEN *pE)` let d be an integer congruent to 0 or 1 mod 4 whose absolute value fits in an `ulong`. Return the absolute value of $D = \text{corediscs}(d)$ assuming that `fa` is the factorization of $|d|$ and $sd > 0$ (s is the sign of d and D). Set `*pP` and `*pE`

to the factorization of the conductor f such that $d = Df^2$, where P is a `t_VECSMALL` of primes and E a `t_VECSMALL` of exponents.

`ulong eulerphiu(ulong n)`, Euler's totient function of n .

`ulong eulerphiu_fact(GEN fa)` same, where `fa` is `factoru(n)`.

`long moebiusu(ulong n)`, Moebius μ -function of n .

`long moebiusu_fact(GEN fa)` same, where `fa` is `factoru(n)`.

`ulong radicalu(ulong n)`, product of primes dividing n .

`GEN divisorsu(ulong n)`, returns the divisors of n in a `t_VECSMALL`, sorted by increasing order.

`GEN divisorsu_fact(GEN fa)` same, where `fa` is `factoru(n)`.

`GEN divisors_fact_degmax(GEN fa, long degmax)` return the list of divisors of degree at most `degmax` of the polynomial whose factorization is given by `fa`.

`GEN divisorsu_fact_factored(GEN fa)` where `fa` is `factoru(n)`. Return a vector $[D, F]$, where D is a `t_VECSMALL` containing the divisors of u and $F[i]$ contains `factoru(D[i])`.

`GEN divisorsu_moebius(GEN P)` returns the divisors of n of the form $\prod_{p \in S} (-p)$, $S \subset P$ in a `t_VECSMALL`. The vector is not sorted but its first element is guaranteed to be 1. If P is `factoru(n)[1]`, this returns the set of $\mu(d)d$ where d runs through the squarefree divisors of n .

`long numdivu(ulong n)`, returns the number of positive divisors of $n > 0$.

`long numdivu_fact(GEN fa)` same, where `fa` is `factoru(n)`.

`long omegau(ulong n)` returns the number of prime divisors of $n > 0$.

`long maxomegau(ulong x)` return the optimal B such that $\omega(n) \leq B$ for all $n \leq x$.

`long maxomegaoddu(ulong x)` return the optimal B such that $\omega(n) \leq B$ for all odd $n \leq x$.

`long uissquarefree(ulong n)` returns 1 if n is square-free, and 0 otherwise.

`long uissquarefree_fact(GEN fa)` same, where `fa` is `factoru(n)`.

`long uposisfundamental(ulong x)` return 1 if x is a fundamental discriminant, and 0 otherwise.

`long unegisfundamental(ulong x)` return 1 if $-x$ is a fundamental discriminant, and 0 otherwise.

`long sisfundamental(long x)` return 1 if 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 uisnilpotent(ulong n)` return 1 if every finite group of order n is nilpotent, 0 otherwise.

`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).

`int uis2psp(ulong n)` returns 1 if the odd `ulong n > 1` fails a strong Rabin-Miller test for the base 2 (it thus may be prime or composite), and 0 otherwise (proving that n is composite).

`int uispsp(ulong a, ulong n)` returns 1 if the odd `ulong n > 1` fails a strong Rabin-Miller test for the base $1 < a < n$ (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 usumdiv_fact(GEN fa)`, sum of divisors of `ulong n`, where `fa` is `factoru(n)`.

`GEN usumdivk_fact(GEN fa, ulong k)`, sum of k -th powers of divisors of `ulong n`, where `fa` is `factoru(n)`.

`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 attached 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: $A_1 \dots A_n fun$ for an operation fun with n arguments of class $A_1, \dots, A_n$. A class name is given by a base ring followed by a number of code letters. Base rings are among

F1: $\mathbf{Z}/l\mathbf{Z}$ where $l < 2^{\text{BITS_IN_LONG}}$ is not necessarily prime. Implemented using **ulongs**

Fp: $\mathbf{Z}/p\mathbf{Z}$ 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.

Fq: $\mathbf{Z}[X]/(p, T(X))$, 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.

Z: the integers $\mathbf{Z}$, implemented as **t_INTs**.

Zp: the p -adic integers $\mathbf{Z}_p$, implemented as **t_INTs**, for arbitrary p

Z1: the p -adic integers $\mathbf{Z}_p$, implemented as **t_INTs**, for $p < 2^{\text{BITS_IN_LONG}}$

z: the integers $\mathbf{Z}$, implemented using (signed) **longs**.

Q: the rational numbers $\mathbf{Q}$, implemented as **t_INTs** and **t_FRACs**.

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 $\mathbf{Z}/p\mathbf{Z}[X]$

Y: polynomial in $Y \neq X$. This is used to resolve ambiguities. E.g. **FpXY** means $((\mathbf{Z}/p\mathbf{Z}[X])[Y])$.

V: vector (**t_VEC** or **t_COL**), treated as a row vector (independently of the actual type). E.g. **ZV** means $\mathbf{Z}^k$ 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 $(\mathbf{Z}/p\mathbf{Z})[X]/(T(X))$, **FpXQV** means a vector of such elements, etc.

n: a polynomial representative (**t_POL**) for a truncated power series modulo X^n . E.g. an **FpXn** belongs to $(\mathbf{Z}/p\mathbf{Z})[X]/(X^n)$, **FpXnV** 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 Coefficient ring.

`long Rg_type(GEN x, GEN *ptp, GEN *ptpol, long *ptprec)` returns the “natural” base ring over which the object x is defined.

Raise an error if it detects consistency problems in modular objects: incompatible rings (e.g. $\mathbf{F}_p$ and $\mathbf{F}_q$ for primes $p \neq q$, $\mathbf{F}_p[X]/(T)$ and $\mathbf{F}_p[X]/(U)$ for $T \neq U$). Minor discrepancies are supported if they make general sense (e.g. $\mathbf{F}_p$ and $\mathbf{F}_{p^k}$, but not $\mathbf{F}_p$ and $\mathbf{Q}_p$); `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.

- 0: none of the others (presumably multivariate, possibly inconsistent).
- `t_INT`: defined over $\mathbf{Z}$.
- `t_FRAC`: defined over $\mathbf{Q}$.
- `t_INTMOD`: defined over $\mathbf{Z}/p\mathbf{Z}$, where `*ptp` is set to p . It is not checked whether p is prime.
- `t_COMPLEX`: defined over $\mathbf{C}$ (at least one `t_COMPLEX` with at least one inexact floating point `t_REAL` component). Set `*ptprec` to the minimal accuracy (as per `precision`) of inexact components.
- `t_REAL`: defined over $\mathbf{R}$ (at least one inexact floating point `t_REAL` component). Set `*ptprec` to the minimal accuracy (as per `precision`) of inexact components.
- `t_PADIC`: defined over $\mathbf{Q}_p$, where `*ptp` is set to p and `*ptprec` to the p -adic accuracy.
- `t_FFELT`: defined over a finite field $\mathbf{F}_{p^k}$, where `*ptp` is set to the field characteristic p and `*ptpol` is set to a `t_FFELT` belonging to the field.
- `t_POL`: defined over a polynomial ring.
- 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 `RgX_type`, `t1` and `t2` are recovered using

```
void RgX_type_decode(long t, long *t1, long *t2)
```

`t1` is one of

`t_POLMOD`: at least one `t_POLMOD` component, set `*ppol` to the modulus,

`t_QUAD`: no `t_POLMOD`, at least one `t_QUAD` component, set `*ppol` to the modulus (`-.pol`) of the `t_QUAD`,

`t_COMPLEX`: no `t_POLMOD` or `t_QUAD`, at least one `t_COMPLEX` component, set `*ppol` to $y^2 + 1$.

and the underlying base ring R is given by `t2`, which is one of `t_INT`, `t_INTMOD` (set `*ptp`) or `t_PADIC` (set `*ptp` and `*ptprec`), with the same meaning as above.

`long RgX_type_code(long t1, long t2)` is the reverse of `RgX_type_decode`, it returns the component type t built from `t1` and `t2`. This is implemented as a macro so as to be usable as a label in a switch when both types are constants, e.g.,

```
case RgX_type_code(t_POLMOD, t_INTMOD): ...
```

`int RgX_type_is_composite(long t)` t as returned by `RgX_type`, return 1 if t is a composite type, and 0 otherwise.

`GEN Rg_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 Rg_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),

`long RgX_type(GEN x, GEN *ptp, GEN *ptpol, long *ptprec)` returns the “natural” base ring over which the polynomial x is defined, otherwise as `Rg_type`.

`long RgX_Rg_type(GEN x, GEN y, GEN *ptp, GEN *ptpol, long *ptprec)` returns the “natural” base ring over which the polynomial x and the element y are defined, otherwise as `Rg_type`.

`long RgX_type2(GEN x, GEN y, GEN *ptp, GEN *ptpol, long *ptprec)` returns the “natural” base ring over which the polynomials x and y are defined, otherwise as `Rg_type`.

`long RgX_type3(GEN x, GEN y, GEN z, GEN *ptp, GEN *ptpol, long *ptprec)` returns the “natural” base ring over which the polynomials x , y and z are defined, otherwise as `Rg_type`.

`long RgM_type(GEN x, GEN *ptp, GEN *ptpol, long *ptprec)` returns the “natural” base ring over which the matrix x is defined, otherwise as `Rg_type`.

`long RgM_type2(GEN x, GEN y, GEN *ptp, GEN *ptpol, long *ptprec)` returns the “natural” base ring over which the matrices x and y are defined, otherwise as `Rg_type`.

`long RgV_type(GEN x, GEN *ptp, GEN *ptpol, long *ptprec)` returns the “natural” base ring over which the vector x is defined, otherwise as `Rg_type`.

`long RgV_type2(GEN x, GEN y, GEN *ptp, GEN *ptpol, long *ptprec)` returns the “natural” base ring over which the vectors x and y are defined, otherwise as `Rg_type`.

`long RgM_RgC_type(GEN x, GEN y, GEN *ptp, GEN *ptpol, long *ptprec)` returns the “natural” base ring over which the matrix x and the vector y are defined, otherwise as `Rg_type`.

7.3 Modular arithmetic.

These routines implement univariate polynomial arithmetic and linear algebra over finite fields, in fact over finite rings of the form $(\mathbf{Z}/p\mathbf{Z})[X]/(T)$, where p is not necessarily prime and $T \in (\mathbf{Z}/p\mathbf{Z})[X]$ 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.

7.3.1 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 $\mathbf{Z}/p\mathbf{Z}$.

7.3.1.1 Conversions.

`int Rg_is_Fp(GEN z, GEN *p)`, checks if z can be mapped to $\mathbf{Z}/p\mathbf{Z}$: 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 $\mathbf{Z}/p\mathbf{Z}$: 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 ℓ satisfying $p = \ell^n$ for some n (less than the accuracy of the input). Returns `lift(z * 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 $\mathbf{F}_p$ (as per `Rg_to_Fp`), and a prime number p . This routine returns a normal form of z : either an `F2m` ($p = 2$), an `F1m` (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 follows:

```
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 * Mod(1,p))`, which is an `FpM`.

7.3.1.2 Basic operations.

GEN random_FpC(long n, GEN p) returns a random FpC with n components.

GEN random_FpV(long n, GEN p) returns a random FpV with n components.

GEN FpC_center(GEN z, GEN p, GEN pov2) returns a t_COL whose entries are the Fp_center of the $\text{gel}(z, i)$.

GEN FpM_center(GEN z, GEN p, GEN pov2) returns a matrix whose entries are the Fp_center of the $\text{gcoeff}(z, i, j)$.

void FpC_center_inplace(GEN z, GEN p, GEN pov2) in-place version of FpC_center, using affii.

void FpM_center_inplace(GEN z, GEN p, GEN pov2) in-place version of FpM_center, using affii.

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 an FpV.

GEN FpM_add(GEN x, GEN y, GEN p) adds the two ZMs x and y (assumed to have compatible dimensions), and reduce modulo p to obtain an FpM.

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 an FpV.

GEN FpM_sub(GEN x, GEN y, GEN p) subtracts the two ZMs x and y (assumed to have compatible dimensions), and reduce modulo p to obtain an FpM.

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 FpM_Fp_mul(GEN x, GEN y, GEN p) multiplies the ZM x (seen as a column vector) by the t_INT y and reduce modulo p to obtain an FpM.

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 ZMs 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 stack-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.

GEN FpV_factorback(GEN L, GEN e, GEN p) given an FpV L and a ZV or zv e of the same length, return $\prod_i L_i^{e_i}$ modulo p .

7.3.1.3 Fp-linear algebra. The implementations are not asymptotically efficient ($O(n^3)$ standard algorithms). Unless mentioned otherwise, all these functions assume that p is prime.

GEN FpM_deplin(GEN x, GEN p) returns a nontrivial 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 a and b are FpM.

GEN FpM_FpC_gauss(GEN a, GEN b, GEN p) as `gauss`, where a is a FpM and b a FpC.

GEN FpM_image(GEN x, GEN p) as `image`

GEN FpM_intersect(GEN x, GEN y, GEN p) as `intersect`

GEN FpM_intersect_i(GEN x, GEN y, GEN p) internal variant of `FpM.intersect` but the result is only a generating set, not necessarily an $\mathbf{F}_p$ -basis. It is not stack-clean, but suitable for `gc_upto`.

GEN FpM_inv(GEN x, GEN p) returns a left inverse of x (the inverse if x is square), or NULL if x is not invertible.

GEN FpM_FpC_invimage(GEN A, GEN y, GEN p) given an FpM A and an FpC y , returns an x such that $Ax = y$, or NULL if no such vector exist.

GEN FpM_invimage(GEN A, GEN y, GEN p) given two FpM A and y , returns x such that $Ax = y$, or NULL if no such matrix exist.

GEN FpM_ker(GEN x, GEN p) as `ker`

long FpM_rank(GEN x, GEN p) as `rank`

GEN FpM_indexrank(GEN x, GEN p) as `indexrank`

GEN FpM_suppl(GEN x, GEN p) as `suppl`

GEN FpM_hess(GEN x, GEN p) upper Hessenberg form of x over $\mathbf{F}_p$.

GEN FpM_charpoly(GEN x, GEN p) characteristic polynomial of x .

GEN FpM_trace(GEN x, GEN p) trace of x . The integer p need not be prime.

7.3.1.4 FqC, FqM and Fq-linear algebra.

An FqM (resp. FqC) is a matrix (resp a `t_COL`) with Fq coefficients (with respect to given T , p), not necessarily reduced (i.e arbitrary `t_INTs` and `ZXs` in the same variable as T).

GEN RgC_to_FqC(GEN z, GEN T, GEN p)

GEN RgM_to_FqM(GEN z, GEN T, GEN p)

GEN FqC_to_FlxqC(GEN x, GEN T, ulong p) where T is a non-NULL Flx.

GEN FqM_to_FlxqM(GEN x, GEN T, ulong p) where T is a non-NULL Flx.

GEN FqC_add(GEN a, GEN b, GEN T, GEN p)

GEN FqC_sub(GEN a, GEN b, GEN T, GEN p)
 GEN FqC_Fq_mul(GEN a, GEN b, GEN T, GEN p)
 GEN FqC_FqV_mul(GEN a, GEN b, GEN T, GEN p)
 GEN FqM_add(GEN a, GEN b, GEN T, GEN p)
 GEN FqM_sub(GEN a, GEN b, GEN T, GEN p)
 GEN FqM_FqC_mul(GEN a, GEN b, GEN T, GEN p)
 GEN FqM_FqC_gauss(GEN a, GEN b, GEN T, GEN p) as gauss, where b is a FqC.
 GEN FqM_FqC_invimage(GEN a, GEN b, GEN T, GEN p)
 GEN FqM_deplin(GEN x, GEN T, GEN p) returns a nontrivial kernel vector, or NULL if none exist.
 GEN FqM_det(GEN x, GEN T, GEN p) as det
 GEN FqM_gauss(GEN a, GEN b, GEN T, GEN p) as gauss, where b is a FqM.
 GEN FqM_image(GEN x, GEN T, GEN p) as image
 GEN FqM_indexrank(GEN x, GEN T, GEN p) as indexrank
 GEN FqM_inv(GEN x, GEN T, GEN p) returns the inverse of x , or NULL if x is not invertible.
 GEN FqM_invimage(GEN a, GEN b, GEN T, GEN p) as invimage
 GEN FqM_ker(GEN x, GEN T, GEN p) as ker
 GEN FqM_mul(GEN a, GEN b, GEN T, GEN p)
 long FqM_rank(GEN x, GEN T, GEN p) as rank
 GEN FqM_suppl(GEN x, GEN T, GEN p) as suppl

7.3.2 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 random_Flv(long n, ulong p) returns a random Flv with n components.
 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 $x + y * \text{Id}$ (x must be square).
 GEN Flm_Fl_sub(GEN x, ulong y, ulong p) returns $x - y * \text{Id}$ (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_Flc_mul_pre(GEN x, GEN y, ulong p, ulong pi) multiplies x and y (assumed to have compatible dimensions), assuming pi is the pseudoinverse of p , or 0 in which case we assume $\text{SMALL_ULONG}(p)$.

GEN Flc_Flv_mul(GEN x, GEN y, ulong p) multiplies the column vector x by the row vector y . The result is a matrix.

GEN Flm_Flc_mul_pre_Flx(GEN x, GEN y, ulong p, ulong pi, long sv) return Flv_to_Flx(Flm_Flc_mul_pre(x, y, p, pi), sv).

GEN Flm_Fl_mul(GEN x, ulong y, ulong p) multiplies the Flm x by y .

GEN Flm_Fl_mul_pre(GEN x, ulong y, ulong p, ulong pi) multiplies the Flm x by y assuming pi is the pseudoinverse of p , or 0 in which case we assume $p < B^{1/2}$ is small.

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 $x * y$.

GEN Flv_Fl_mul(GEN x, ulong y, ulong p) multiplies the Flv x by y .

void Flv_Fl_mul_inplace(GEN x, ulong y, ulong p) replaces the Flc x by $x * y$.

void Flv_Fl_mul_part_inplace(GEN x, ulong y, ulong p, long l) multiplies $x[1..l]$ by y modulo p . In place.

GEN Flv_Fl_div(GEN x, ulong y, ulong p) divides the Flv x by y .

void Flv_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 $v = 0$); if the input was gc-safe, it remains so.

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_neg(GEN x, ulong p) returns $-x$.

void Flv_neg_inplace(GEN x, ulong p) replaces x by $-x$.

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_dotproduct_pre(GEN x, GEN y, ulong p, ulong pi) returns the scalar product of x and y assuming pi is the pseudoinverse of p , or 0 in which case we assume $\text{SMALL_ULONG}(p)$.

GEN Flv_factorback(GEN L, GEN e, ulong p) given an Flv L and a zv e of the same length, return $\prod_i L_i^{e_i}$ modulo p .

ulong Flv_sum(GEN x, ulong p) returns the sum of the components of x .

ulong Flv_prod(GEN x, ulong p) returns the product of the components of x .

ulong Flv_prod_pre(GEN x, ulong p, ulong pi) as Flv_prod assuming pi is the pseudoinverse of p .

GEN Flv_inv(GEN x, ulong p) returns the vector of inverses of the elements of x (as a Flv). Use Montgomery's trick.

void Flv_inv_inplace(GEN x, ulong p) in place variant of Flv_inv.

GEN Flv_inv_pre(GEN x, ulong p, ulong pi) as Flv_inv assuming pi is the pseudoinverse of p .

void Flv_inv_pre_inplace(GEN x, ulong p, ulong pi) in place variant of Flv_inv.

GEN Flc_FpV_mul(GEN x, GEN y, GEN p) multiplies x (seen as a column vector) by y (seen as a row vector, assumed to have compatible dimensions) to obtain an Flm.

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 Flm_row(GEN A, long x0) return $A[i,]$, the i -th row of the Flm A .

GEN Flm_add(GEN x, GEN y, ulong p) adds x and y (assumed to have compatible dimensions).

GEN Flm_sub(GEN x, GEN y, ulong p) subtracts x and y (assumed to have compatible dimensions).

GEN Flm_mul(GEN x, GEN y, ulong p) multiplies x and y (assumed to have compatible dimensions).

GEN Flm_mul_pre(GEN x, GEN y, ulong p, ulong pi) multiplies x and y (assumed to have compatible dimensions), assuming pi is the pseudoinverse of p , or 0 in which case we assume $\text{SMALL_ULONG}(p)$.

GEN Flm_sqr(GEN x, ulong p) squares x (assumed to be square).

GEN Flm_powers(GEN x, ulong n, ulong p) returns $[x^0, \dots, x^n]$ as a t_VEC of Flms.

GEN Flm_powu(GEN x, ulong n, ulong p) computes x^n where x is a square Flm.

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) Assume that p is prime.

ulong Flm_det(GEN x, ulong p) Assume that p is prime.

ulong Flm_det_sp(GEN x, ulong p), as Flm_det, in place (destroys x). Assume that p is prime.

ulong Flm_trace(GEN x, ulong p)

GEN Flm_gauss(GEN a, GEN b, ulong p) as gauss, where b is a Flm. Assume that p is prime.

GEN Flm_Flc_gauss(GEN a, GEN b, ulong p) as gauss, where b is a Flc. Assume that p is prime.

GEN Flm_indexrank(GEN x, ulong p) Assume that p is prime.

GEN Flm_inv(GEN x, ulong p) Assume that p is prime.

GEN Flm_adjoint(GEN x, ulong p) as matadjoint. Assume that p is prime.

GEN Flm_Flc_invimage(GEN A, GEN y, ulong p) given an Flm A and an Flc y , returns an x such that $Ax = y$, or NULL if no such vector exist. Assume that p is prime.

GEN Flm_invimage(GEN A, GEN y, ulong p) given two Flm A and y , returns x such that $Ax = y$, or NULL if no such matrix exist. Assume that p is prime.

GEN Flm_ker(GEN x, ulong p) Assume that p is prime.

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). Assume that p is prime.

long Flm_rank(GEN x, ulong p) Assume that p is prime.

long Flm_suppl(GEN x, ulong p) Assume that p is prime.

GEN Flm_image(GEN x, ulong p) Assume that p is prime.

GEN Flm_intersect(GEN x, GEN y, ulong p) Assume that p is prime.

GEN Flm_intersect_i(GEN x, GEN y, GEN p) internal variant of Flm_intersect but the result is only a generating set, not necessarily an $\mathbf{F}_p$ -basis. It *is* a basis if both x and y have independent columns. The function is not stack-clean, but suitable for gc_upto. Assume that p is prime.

GEN Flm_transpose(GEN x)

GEN Flm_hess(GEN x, ulong p) upper Hessenberg form of x over $\mathbf{F}_p$. Assume that p is prime.

GEN Flm_hess_pre(GEN x, ulong p, ulong pi) as Flm_hess assuming pi is the pseudoinverse of p .

7.3.3 F2c / F2v, F2m. An F2v v is a t_VECSMALL representing a vector over $\mathbf{F}_2$. 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.

int F2v_equal0(GEN x) returns 1 if all entries are 0, and return 0 otherwise.

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 $x[a], \dots, x[b]$. 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.

GEN F2m_copy(GEN x) returns a copy of x .

GEN F2m_transpose(GEN x) returns the transpose of x .

GEN F2m_row(GEN x, long j) returns the F2v which corresponds to the j -th row of the F2m 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 $a \leq b$.

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 $n \times n$ identity matrix.
`GEN zero_F2v(long n)` creates a `F2v` with n components set to 0.
`GEN const_F2v(long n)` creates a `F2v` with n components set to 1.
`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 $m \times n$ 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_F2m_copy(long m, long n)` creates a `F2m` with $m \times n$ components set to 0.
`GEN F2v_to_Flv(GEN x)`
`GEN F2c_to_ZC(GEN x)`
`GEN ZV_to_F2v(GEN x)`
`GEN RgV_to_F2v(GEN x)`
`GEN F2m_to_Flm(GEN x)`
`GEN F2m_to_ZM(GEN x)`
`GEN Flv_to_F2v(GEN x)`
`GEN Flm_to_F2m(GEN x)`
`GEN ZM_to_F2m(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 .
`void F2v_and_inplace(GEN x, GEN y)` replaces x by the term-by term product of x and y (which is the logical and). It is allowed for y to be shorter than x .
`void F2v_negimply_inplace(GEN x, GEN y)` replaces x by the term-by term logical **and not** of x and y . It is allowed for y to be shorter than x .

`void F2v_or_inplace(GEN x, GEN y)` replaces x by the term-by term logical or of x and y . It is allowed for y to be shorter than x .

`int F2v_subset(GEN x, GEN y)` return 1 if the set of indices of non-zero components of y is a subset of the set of indices of non-zero components of x , 0 otherwise.

`ulong F2v_hamming(GEN x)` returns the Hamming weight of x , that is the number of nonzero entries.

`ulong F2m_det(GEN x)`

`ulong F2m_det_sp(GEN x)`, as `F2m_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 `F2m_ker` (if `deplin=0`) or `F2m_deplin` (if `deplin=1`), in place (destroys x).

7.3.4 F3c / F3v, F3m. An F3v v is a `t_VECSMALL` representing a vector over $\mathbf{F}_3$. Specifically $z[0]$ is the usual codeword, $z[1]$ is the number of components of v and the coefficients are given by pair of adjacent bits of remaining words by increasing indices, with the coding $00 \mapsto 0, 01 \mapsto 1, 10 \mapsto 2$ and 11 is undefined.

`ulong F3v_coeff(GEN x, long i)` returns the coefficient $i \geq 1$ of x .

`void F3v_clear(GEN x, long i)` sets the coefficient $i \geq 1$ of x to 0.

`void F3v_set(GEN x, long i, ulong n)` sets the coefficient $i \geq 1$ of x to $n < 3$,

`ulong F3m_coeff(GEN x, long i, long j)` returns the coefficient (i, j) of x .

`void F3m_set(GEN x, long i, long j, ulong n)` sets the coefficient (i, j) of x to $n < 3$.

`GEN F3m_copy(GEN x)` returns a copy of x .

`GEN F3m_transpose(GEN x)` returns the transpose of x .

`GEN F3m_row(GEN x, long j)` returns the F3v which corresponds to the j -th row of the F3m x .

`GEN F3m_ker(GEN x)`

`GEN F3m_ker_sp(GEN x, long deplin)`, as `F3m_ker` (if `deplin=0`) or `F2m_deplin` (if `deplin=1`), in place (destroys x).

`GEN F3m_mul(GEN x, GEN y)` multiplies x and y (assumed to have compatible dimensions).

`long F3m_rank(GEN x)`

`GEN F3v_dotproduct(GEN x, GEN y)`

`GEN F3v_dotsquare(GEN x)`

`GEN zero_F3v(long n)` creates a F3v with n components set to 0.

`GEN zero_F3m_copy(long m, long n)` creates a F3m with $m \times n$ components set to 0.

```

GEN F3v_to_Flv(GEN x)
GEN ZV_to_F3v(GEN x)
GEN RgV_to_F3v(GEN x)
GEN F3c_to_ZC(GEN x)
GEN F3m_to_Flm(GEN x)
GEN F3m_to_ZM(GEN x)
GEN Flv_to_F3v(GEN x)
GEN Flm_to_F3m(GEN x)
GEN ZM_to_F3m(GEN x)
GEN RgM_to_F3m(GEN x)

```

7.3.5 FlxqV, FlxqC, FlxqM. See FqV, FqC, FqM operations.

GEN FlxqV_dotproduct(GEN x, GEN y, GEN T, ulong p) as FpV_dotproduct.

GEN FlxqV_dotproduct_pre(GEN x, GEN y, GEN T, ulong p, ulong pi) where pi is the pseudoinverse of p , or 0 in which case we assume $\text{SMALL_ULONG}(p)$.

GEN FlxM_Flx_add_shallow(GEN x, GEN y, ulong p) as RgM_Rg_add_shallow.

GEN FlxqC_Flxq_mul(GEN x, GEN y, GEN T, ulong p)

GEN FlxqM_Flxq_mul(GEN x, GEN y, GEN T, ulong p)

GEN FlxqM_FlxqC_gauss(GEN a, GEN b, GEN T, ulong p)

GEN FlxqM_FlxqC_invimage(GEN a, GEN b, GEN T, ulong p)

GEN FlxqM_FlxqC_mul(GEN a, GEN b, GEN T, ulong p)

GEN FlxqM_deplin(GEN x, GEN T, ulong p)

GEN FlxqM_det(GEN x, GEN T, ulong p)

GEN FlxqM_gauss(GEN a, GEN b, GEN T, ulong p)

GEN FlxqM_image(GEN x, GEN T, ulong p)

GEN FlxqM_indexrank(GEN x, GEN T, ulong p)

GEN FlxqM_inv(GEN x, GEN T, ulong p)

GEN FlxqM_invimage(GEN a, GEN b, GEN T, ulong p)

GEN FlxqM_ker(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 FlxqC_red_pre(GEN V, ulong p, ulong pi)

GEN FlxqM_red_pre(GEN M, ulong p, ulong pi)

GEN FlxqM_suppl(GEN x, GEN T, ulong p)

GEN matid_FlxqM(long n, GEN T, ulong p)

7.3.6 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 FpX s. All operations are understood to take place in $(\mathbf{Z}/p\mathbf{Z})[X]$.

7.3.6.1 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 FpX s).

`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 FpX s).

7.3.6.2 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_half(GEN x, GEN p)` returns z such that $2z = x$ modulo p assuming such z exists.

`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_powu(GEN x, ulong n, GEN p)` returns x^n .

`GEN FpX_conv(GEN x, GEN y, GEN p)` return the-term by-term product of x and y .

`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 $x \bmod y$.

`long FpX_valrem(GEN x, GEN t, GEN p, GEN *r)` The arguments x and e being nonzero 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 stack-clean, but nevertheless suitable for `gc_upto`.

GEN FpX_integ(GEN x, GEN p) returns the primitive of x whose constant term is 0.

GEN FpX_digits(GEN x, GEN B, GEN p) returns a vector of FpX $[c_0, \dots, c_n]$ of degree less than the degree of B and such that $x = \sum_{i=0}^n c_i B^i$.

GEN FpXV_FpX_fromdigits(GEN v, GEN B, GEN p) where $v = [c_0, \dots, c_n]$ is a vector of FpX, returns $\sum_{i=0}^n c_i B^i$.

GEN FpX_Fp_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 FpXM 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 FpX_halfgcd_all(GEN x, GEN y, GEN p, GEN *pt_a, GEN *pt_b) as FpX_halfgcd, in addition, if `pt_a` (resp. `pt_b`) is not NULL, `*pt_a` (resp. `*pt_b`) is set to a (resp. 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 NULL, `*u` 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 $] -p/2, p[$ and that `pov2` is `shifti(p, -1)`.

GEN FpX_center_i(GEN z, GEN p, GEN pov2) shallow internal variant of FpX_center.

GEN FpX_Frobenius(GEN T, GEN p) returns $X^p \pmod{T(X)}$.

GEN FpX_matFrobenius(GEN T, GEN p) returns the matrix of the Frobenius automorphism $x \mapsto x^p$ over the power basis of $\mathbf{F}_p[X]/(T)$.

7.3.6.3 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) add the Fp x to the FpX y .

GEN FpX_Fp_add_shallow(GEN y, GEN x, GEN p) add the Fp x to the FpX y . Shallow function.

GEN FpX_Fp_sub(GEN y, GEN x, GEN p) subtract the Fp x from the FpX y .

GEN FpX_Fp_sub_shallow(GEN y, GEN x, GEN p) subtract the Fp x from the FpX y . Shallow function.

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 Fp 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 $x \neq 0$).

GEN FpX_Fp_div(GEN x, GEN y, GEN p) divides the FpX x by the Fp y .

GEN FpX_divu(GEN x, ulong y, GEN p) divides the FpX x by y .

7.3.6.4 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 nonzero Fp (the routine would work with any nonzero t_INT but is not efficient in this case). Shallow function.

GEN FpX_eval(GEN x, GEN y, GEN p) evaluates the FpX x at the Fp y . The result is an Fp.

GEN FpX_FpV_multieval(GEN P, GEN v, GEN p) returns the vector $[P(v[1]), \dots, P(v[n])]$ as a FpV.

GEN FpX_dotproduct(GEN x, GEN y, GEN p) return the scalar product $\sum_{i \geq 0} x_i y_i$ of the coefficients of x and y .

GEN FpXV_FpC_mul(GEN V, GEN W, GEN p) multiplies a nonempty row vector of FpX by a column vector of Fp of compatible dimensions. The result is an FpX.

GEN FpXV_prod(GEN V, GEN p), V being a vector of FpX, returns their product.

GEN FpXV_composedsum(GEN V, GEN p), V being a vector of FpX, returns their composed sum, see FpX_composedsum.

GEN FpXV_factorback(GEN L, GEN e, GEN p, long v) returns $\prod_i L_i^{e_i}$ where L is a vector of FpXs in the variable v and e a vector of non-negative t_INT s or a $t_VECSMALL$.

GEN FpV_roots_to_pol(GEN V, GEN p, long v), V being a vector of INT s, 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 $x \pmod{Tx}$ and to $y \pmod{Ty}$. Assumes Tx and Ty are coprime, and $Tz = Tx * Ty$ or $NULL$ (in which case it is computed within).

GEN FpV_polint(GEN x, GEN y, GEN p, long v) returns the FpX interpolation polynomial with value $y[i]$ at $x[i]$. Assumes lengths are the same, components are t_INT s, and the $x[i]$ are distinct modulo p .

GEN FpV_FpM_polint(GEN x, GEN V, GEN p, long v) equivalent (but faster) to applying FpV_polint($x, \dots$) to all the elements of the vector V (thus, returns a FpXV).

GEN FpX_FpXV_multirem(GEN A, GEN P, GEN p) given a FpX A and a vector P of pairwise coprime FpX of length $n \geq 1$, return a vector B of the same length such that $B[i] = A \pmod{P[i]}$ and $B[i]$ of minimal degree for all $1 \leq i \leq n$.

GEN FpXV_chinese(GEN A, GEN P, GEN p, GEN *pM) let P be a vector of pairwise coprime FpX, let A be a vector of FpX of the same length $n \geq 1$ and let M be the product of the elements of P .

Returns a **FpX** of minimal degree congruent to $A[i] \bmod P[i]$ for all $1 \leq i \leq n$. If **pM** is not **NULL**, set ***pM** to M .

GEN FpV_invVandermonde(**GEN L**, **GEN d**, **GEN p**) L being a **FpV** of length n , return the inverse M of the Vandermonde matrix attached to the elements of L , eventually multiplied by d if it is not **NULL**. If A is a **FpV** and $B = MA$, then the polynomial $P = \sum_{i=1}^n B[i]X^{i-1}$ verifies $P(L[i]) = dA[i]$ for $1 \leq i \leq n$.

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. The 0 polynomial is not totally split.

long FpX_ispower(**GEN f**, **ulong k**, **GEN p**, **GEN *pt**) return 1 if the **FpX** f is a k -th power, 0 otherwise. If pt is not **NULL**, set it to g such that $g^k = f$.

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).

GEN FpX_factor_squarefree(**GEN f**, **GEN p**) returns the squarefree factorization of f modulo p . This is a vector $[u_1, \dots, u_k]$ of squarefree and pairwise coprime **FpX** such that $u_k \neq 1$ and $f = \prod u_i^i$. The other u_i may equal 1. Shallow function.

GEN FpX_ddf(**GEN f**, **GEN p**) assuming that f is squarefree, returns the distinct degree factorization of f modulo p . The returned value v is a **t_VEC** with two components: $F=v[1]$ is a vector of (**FpX**) factors, and $E=v[2]$ is a **t_VECSMALL**, such that f is equal to the product of the $F[i]$ and each $F[i]$ is a product of irreducible factors of degree $E[i]$.

long FpX_ddf_degree(**GEN f**, **GEN XP**, **GEN p**) assuming that f is squarefree and that all its factors have the same degree, return the common degree, where XP is **FpX_Frobenius(f, p)**.

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_nbfact_Frobenius(**GEN f**, **GEN XP**, **GEN p**), as **FpX_nbfact(f, p)** but faster, where XP is **FpX_Frobenius(f, p)**.

GEN 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 $\mathbf{Z}/p\mathbf{Z}$ of the **FpX** f . Assumes that p is prime.

GEN FpX_oneroot(**GEN f**, **GEN p**) returns one root in $\mathbf{Z}/p\mathbf{Z}$ of the **FpX** f . Return **NULL** if no root exists. Assumes that p is prime.

GEN FpX_oneroot_split(**GEN f**, **GEN p**) as **FpX_oneroot**. Faster when f is close to be totally split.

GEN FpX_roots(**GEN f**, **GEN p**) returns the roots in $\mathbf{Z}/p\mathbf{Z}$ of the **FpX** f (without multiplicity, as a vector of **Fps**). Assumes that p is prime.

GEN FpX_roots_mult(GEN f, long n, GEN p) returns the roots in $\mathbf{Z}/p\mathbf{Z}$ with multiplicity at least n of the FpX f (without multiplicity, as a vector of Fps). Assumes that p is prime.

GEN FpX_split_part(GEN f, GEN p) returns the largest totally split squarefree factor of f.

GEN FpX_factcyclo(ulong n, GEN p, ulong m) returns the factors of the n -th cyclotomic polynomial over Fp. if $m = 1$ returns a single factor.

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 $\mathbf{Z}[Y]$ or t_INTs. Returns $\text{Res}_X(a, b)$ in $\mathbf{F}_p[Y]$ as an FpY. The function assumes that X has lower priority than Y.

ulong FpX_extresultant(GEN a, GEN b, GEN p, GEN *ptU, GEN *ptV) given two FpX a and b, returns their resultant and sets Bezout coefficients (if the resultant is 0, the latter are not set). Assumes that p is prime.

GEN FpX_composedprod(GEN P, GEN Q, GEN p) if $P = a \prod_{i=1}^m (x - p_i)$ and $Q = b \prod_{j=1}^n (x - q_j)$ in some suitable algebraic extension, return $a^n b^m \prod_{i,j} (x - p_i q_j)$.

GEN FpX_composedsum(GEN P, GEN Q, GEN p) if $P = a \prod_{i=1}^m (x - p_i)$ and $Q = b \prod_{j=1}^n (x - q_j)$ in some suitable algebraic extension, return $a^n b^m \prod_{i,j} (x - (p_i + q_j))$.

GEN FpX_Newton(GEN x, long n, GEN p) return $\sum i = 0^{n-1} \pi_i X^i$ where π_i is the sum of the i th-power of the roots of x in an algebraic closure.

GEN FpX_fromNewton(GEN x, GEN p) recover a polynomial from its Newton sums given by the coefficients of x . This function assumes that p and the accuracy of x as a FpXn is larger than the degree of the solution.

GEN FpX_Laplace(GEN x, GEN p) return $\sum_{i=0}^{n-1} x_i i! X^i$.

GEN FpX_invLaplace(GEN x, GEN p) return $\sum_{i=0}^{n-1} x_i / i! X^i$.

7.3.7 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 $(\mathbf{Z}/p\mathbf{Z})[X]/(T)$, in which all operations below take place. In addition, Fq routines also allow T = 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 gc_upto). 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.)

7.3.7.1 Preconditioned reduction.

For faster reduction, the modulus T can be replaced by an extended modulus in all FpXQ - and Fq -classes functions, and in FpX_rem and FpX_divrem . An extended modulus (FpXT , which is a tree whose leaves are FpX) In current implementation, an extended modulus is either a plain modulus (an FpX) or a pair of polynomials, one being the plain modulus T and the other being $\text{FpX_invBarret}(T, p)$.

$\text{GEN FpX_get_red}(\text{GEN } T, \text{GEN } p)$ returns the extended modulus eT .

To write code that works both with plain and extended moduli, the following accessors are defined:

$\text{GEN get_FpX_mod}(\text{GEN } eT)$ returns the underlying modulus T .

$\text{GEN get_FpX_var}(\text{GEN } eT)$ returns the variable number $\text{varn}(T)$.

$\text{GEN get_FpX_degree}(\text{GEN } eT)$ returns the degree $\text{degpol}(T)$.

7.3.7.2 Conversions.

$\text{int ff_parse_Tp}(\text{GEN } Tp, \text{GEN } *T, \text{GEN } *p, \text{long red})$ Tp is either a prime number p or a t_VEC with 2 entries T (an irreducible polynomial mod p) and p (a prime number). Sets $*p$ and $*T$ to the corresponding GENs (NULL if undefined). If red is nonzero, normalize $*T$ as an FpX ; on the other hand, to initialize a p -adic function, set red to 0 and $*T$ is left as is and must be a ZX to start with. Return 1 on success, and 0 on failure. This helper routine is used by GP functions such as factormod where a single user argument defines a finite field. t_FFELT is not supported.

$\text{GEN Rg_is_FpXQ}(\text{GEN } z, \text{GEN } *T, \text{GEN } *p)$, checks if z is a GEN which can be mapped to $\mathbf{F}_p[X]/(T)$: 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 $*T$ if $*T$ is not NULL (once mapped to a FpX), or a t_FFELT z with the same definition field as $*T$ if $*T$ is not NULL and is a t_FFELT .

If an integer modulus is found it is put in $*p$, else $*p$ is left unchanged. If a polynomial modulus is found it is put in $*T$, if a t_FFELT z is found, z is put in $*T$, else $*T$ is left unchanged.

$\text{int RgX_is_FpXQX}(\text{GEN } z, \text{GEN } *T, \text{GEN } *p)$, z a t_POL , checks if it can be mapped to a FpXQX , by checking Rg_is_FpXQ coefficientwise.

$\text{GEN Rg_to_FpXQ}(\text{GEN } z, \text{GEN } T, \text{GEN } p)$, z a GEN which can be mapped to $\mathbf{F}_p[X]/(T)$: 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 (once mapped to a FpX), a suitable t_RFRAC . Returns z as an FpXQ , normalized.

$\text{GEN Rg_to_Fq}(\text{GEN } z, \text{GEN } T, \text{GEN } p)$, applies Rg_to_Fp if T is NULL and Rg_to_FpXQ otherwise.

$\text{GEN RgX_to_FpXQX}(\text{GEN } z, \text{GEN } T, \text{GEN } p)$, z a t_POL , returns the FpXQ obtained by applying Rg_to_FpXQ coefficientwise.

$\text{GEN RgX_to_FqX}(\text{GEN } z, \text{GEN } T, \text{GEN } p)$: let z be a t_POL ; returns the FqX obtained by applying Rg_to_Fq coefficientwise.

$\text{GEN Fq_to_FpXQ}(\text{GEN } z, \text{GEN } T, \text{GEN } p \text{ /*unused*/})$ if z is a t_INT , convert it to a constant polynomial in the variable of T , otherwise return z (shallow function).

$\text{GEN Fq_red}(\text{GEN } x, \text{GEN } T, \text{GEN } p)$, x a ZX or t_INT , reduce it to an Fq ($T = \text{NULL}$ is allowed iff x is a t_INT).

GEN FqX_red(GEN x, GEN T, GEN p), x a t_POL whose coefficients are ZXs or t_INTs, reduce them to Fqs. (If T = NULL, as FpXX_red(x, p).)

GEN FqC_red(GEN x, GEN T, GEN p), x a vector of ZXs or t_INTs, reduce them to a t_COL of Fqs. (If T = NULL, only reduce components mod p to FpXs or Fps.)

GEN FqV_red(GEN x, GEN T, GEN p), x a vector of ZXs or t_INTs, reduce them to a t_VEC of Fqs. (If T = NULL, only reduce components mod p to FpXs or Fps.)

GEN FqM_red(GEN x, GEN T, GEN p), x a matrix of ZXs or t_INTs, reduce them to Fqs. (If T = NULL, only reduce components mod p to FpXs or Fps.)

GEN FpXQ_red(GEN x, GEN T, GEN p) x a t_POL whose coefficients are t_INTs, reduce them to FpXQs.

7.3.8 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 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.

In the following three functions the integer parameter ord can be given either as a positive t_INT N, or as its factorization matrix faN, or as a pair [N, faN]. The parameter may be omitted by setting it to NULL (the value is then $p^d - 1$, $d = \deg T$).

GEN FpXQ_log(GEN a, GEN g, GEN ord, GEN T, GEN p) Let g be of order dividing ord in the finite field $\mathbf{F}_p[X]/(T)$, return e such that $a^e = g$. If e does not exists, the result is 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. Assume that ord is a multiple of the order of a. Assume 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) Let T be irreducible mod p and $q = p^{\deg T}$; returns NULL if a is not an n-th power residue mod p. Otherwise, returns an n-th root of a; if zn is not NULL set it to a primitive m-th root of 1 in $\mathbf{F}_q$, $m = \gcd(q - 1, n)$ allowing to compute all m solutions in $\mathbf{F}_q$ of the equation $x^n = a$.

7.3.9 Fq.

GEN random_Fq(GEN T, GEN p) returns a random 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_half(GEN x, GEN T, GEN p) returns z such that $2z = x$ assuming such z exists.

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 FqV_factorback(GEN L, GEN e, GEN T, GEN p) given an FqV L and a ZV or zv e of the same length, return $\prod_i L_i^{e_i}$ modulo p .

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 .

GEN Fq_log(GEN a, GEN g, GEN ord, GEN T, GEN p) as Fp_log or FpXQ_log.

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 and that p is prime; $T = \text{NULL}$ is forbidden unless x is an Fp.

long Fq_ispower(GEN x, GEN n, GEN T, GEN p) returns 1 if x is a n -th power and 0 otherwise. Assumes that T is irreducible mod p and that p is prime; $T = \text{NULL}$ 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 a, GEN n, GEN T, GEN p, GEN *zn) as FpXQ_sqrtn.

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 $[x, x^p, x^{p^2}, \dots, x^{p^{n-1}}]$ 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), L being a vector of primes dividing $p^{\deg T} - 1$, returns an element of $G := \mathbf{F}_p[x]/(T)$ 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 $\ell = 2$, since 2 is treated separately in any case, i.e. the generator obtained is never a square if p is odd.

GEN gener_Fq_local(GEN T, GEN p, GEN L) as pgener_Fp_local(p, L) if T is NULL, or gener_FpXQ_local (otherwise).

GEN FpXQ_powers(GEN x, long n, GEN T, GEN p) returns $[x^0, \dots, x^n]$ as a t_VEC of FpXQs.

GEN FpXQ_matrix_pow(GEN x, long m, long n, GEN T, GEN p), as FpXQ_powers($x, n-1, T, p$), but returns the powers as a $m \times n$ matrix. Usually, we have $m = n = \deg T$.

GEN FpXQ_autpow(GEN a, ulong n, GEN T, GEN p) computes $\sigma^n(X)$ assuming $a = \sigma(X)$ where σ is an automorphism of the algebra $\mathbf{F}_p[X]/T(X)$.

GEN FpXQ_autsum(GEN a, ulong n, GEN T, GEN p) a being a two-component vector, σ being the automorphism defined by $\sigma(X) = a[1] \pmod{T(X)}$, returns the vector $[\sigma^n(X), b\sigma(b) \dots \sigma^{n-1}(b)]$ where $b = a[2]$.

GEN FpXQ_auttrace(GEN a, ulong n, GEN T, GEN p) a being a two-component vector, σ being the automorphism defined by $\sigma(X) = a[1] \pmod{T(X)}$, returns the vector $[\sigma^n(X), b + \sigma(b) + \dots + \sigma^{n-1}(b)]$ where $b = a[2]$.

GEN FpXQ_outpowers(GEN S, long n, GEN T, GEN p) returns $[x, S(x), S(S(x)), \dots, S^{(n)}(x)]$ as a t_VEC of FpXQs.

GEN FpXQM_autsum(GEN a, long n, GEN T, GEN p) σ being the automorphism defined by $\sigma(X) = a[1] \pmod{T(X)}$, returns the vector $[\sigma^n(X), b\sigma(b) \dots \sigma^{n-1}(b)]$ where $b = a[2]$ is a square matrix.

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).

GEN FpXC_FpXQ_eval(GEN C, GEN x, GEN T, GEN p) applies FpX_FpXQV_eval to all elements of the vector C and returns a t_COL.

GEN FpXC_FpXQV_eval(GEN C, GEN V, GEN T, GEN p) applies FpX_FpXQV_eval to all elements of the vector C and returns a t_COL.

GEN FpXM_FpXQV_eval(GEN M, GEN V, GEN T, GEN p) applies FpX_FpXQV_eval to all elements of the matrix M .

7.3.10 FpXn. Let p a $\mathbf{t_INT}$ and T an $\mathbf{FpX}$ for p , both to be given in the function arguments; an $\mathbf{FpXn}$ object is an $\mathbf{FpX}$ whose degree is strictly less than n . They represent a class in $(\mathbf{Z}/p\mathbf{Z})[X]/(X^n)$, in which all operations below take place. They can be seen as truncated power series.

$\mathbf{GEN\ FpXn_mul}(\mathbf{GEN\ x, GEN\ y, long\ n, GEN\ p})$ return $xy \pmod{X^n}$.

$\mathbf{GEN\ FpXn_sqr}(\mathbf{GEN\ x, long\ n, GEN\ p})$ return $x^2 \pmod{X^n}$.

$\mathbf{GEN\ FpXn_div}(\mathbf{GEN\ x, GEN\ y, long\ n, GEN\ p})$ return $x/y \pmod{X^n}$.

$\mathbf{GEN\ FpXn_inv}(\mathbf{GEN\ x, long\ n, GEN\ p})$ return $1/x \pmod{X^n}$.

$\mathbf{GEN\ FpXn_exp}(\mathbf{GEN\ f, long\ n, GEN\ p})$ return $\exp(f)$ as a composition of formal power series. It is required that the valuation of f is positive and that $p > n$.

$\mathbf{GEN\ FpXn_expint}(\mathbf{GEN\ f, long\ n, GEN\ p})$ return $\exp(F)$ where F is the primitive of f that vanishes at 0. It is required that $p > n$.

7.3.11 FpXC, FpXM.

$\mathbf{GEN\ FpXC_center}(\mathbf{GEN\ C, GEN\ p, GEN\ pov2})$

$\mathbf{GEN\ FpXM_center}(\mathbf{GEN\ M, GEN\ p, GEN\ pov2})$

7.3.12 FpXX, FpXY. Contrary to what the name implies, an $\mathbf{FpXX}$ is a $\mathbf{t_POL}$ whose coefficients are either $\mathbf{t_INTs}$ or $\mathbf{FpXs}$. This reduces memory overhead at the expense of consistency. The prefix $\mathbf{FpXY}$ is an alias for $\mathbf{FpXX}$ when variables matters.

$\mathbf{GEN\ FpXX_red}(\mathbf{GEN\ z, GEN\ p})$, z a $\mathbf{t_POL}$ whose coefficients are either $\mathbf{ZXs}$ or $\mathbf{t_INTs}$. Returns the $\mathbf{t_POL}$ equal to z with all components reduced modulo p .

$\mathbf{GEN\ FpXX_renormalize}(\mathbf{GEN\ x, long\ l})$, as $\mathbf{normalizopol}$, where $l = \lg(x)$, in place.

$\mathbf{GEN\ FpXX_add}(\mathbf{GEN\ x, GEN\ y, GEN\ p})$ adds x and y .

$\mathbf{GEN\ FpXX_sub}(\mathbf{GEN\ x, GEN\ y, GEN\ p})$ returns $x - y$.

$\mathbf{GEN\ FpXX_Fp_sub}(\mathbf{GEN\ x, GEN\ y, GEN\ p})$ subtracts the $\mathbf{FpXX\ x}$ by the $\mathbf{Fp\ y}$.

$\mathbf{GEN\ FpXX_neg}(\mathbf{GEN\ x, GEN\ p})$ returns $-x$.

$\mathbf{GEN\ FpXX_Fp_mul}(\mathbf{GEN\ x, GEN\ y, GEN\ p})$ multiplies the $\mathbf{FpXX\ x}$ by the $\mathbf{Fp\ y}$.

$\mathbf{GEN\ FpXX_FpX_mul}(\mathbf{GEN\ x, GEN\ y, GEN\ p})$ multiplies the coefficients of the $\mathbf{FpXX\ x}$ by the $\mathbf{FpX\ y}$.

$\mathbf{GEN\ FpXX_mulu}(\mathbf{GEN\ x, GEN\ y, GEN\ p})$ multiplies the $\mathbf{FpXX\ x}$ by the scalar y .

$\mathbf{GEN\ FpXX_halve}(\mathbf{GEN\ x, GEN\ p})$ returns z such that $2z = x$ assuming such z exists.

$\mathbf{GEN\ FpXX_deriv}(\mathbf{GEN\ P, GEN\ p})$ differentiates P with respect to the main variable.

$\mathbf{GEN\ FpXX_integ}(\mathbf{GEN\ P, GEN\ p})$ returns the primitive of P with respect to the main variable whose constant term is 0.

$\mathbf{GEN\ FpXY_eval}(\mathbf{GEN\ Q, GEN\ y, GEN\ x, GEN\ p})$ Q being an $\mathbf{FpXY}$, i.e. a $\mathbf{t_POL}$ with $\mathbf{Fp}$ or $\mathbf{FpX}$ coefficients representing an element of $\mathbf{F}_p[X][Y]$. Returns the $\mathbf{Fp\ Q(x, y)}$.

$\mathbf{GEN\ FpXY_evalx}(\mathbf{GEN\ Q, GEN\ x, GEN\ p})$ Q being an $\mathbf{FpXY}$, returns the $\mathbf{FpX\ Q(x, Y)}$, where Y is the main variable of Q .

GEN FpXY_evaly(GEN Q, GEN y, GEN p, long vx) Q an FpXY, returns the FpX $Q(X, y)$, where X is the second variable of Q , and vx is the variable number of X .

GEN FpXY_FpXQ_evaly(GEN Q, GEN y, GEN T, GEN p, long vx) Q an FpXY and y being an FpXQ, returns the FpXQX $Q(X, y)$, where X is the second variable of Q , and vx is the variable number of X .

GEN FpXY_Fq_evaly(GEN Q, GEN y, GEN T, GEN p, long vx) Q an FpXY and y being an Fq, returns the FqX $Q(X, y)$, where X is the second variable of Q , and vx is the variable number of X .

GEN FpXY_FpXQ_evalx(GEN Q, GEN x, ulong p) Q an FpXY and x being an FpXQ, returns the FpXQX $Q(x, Y)$, where Y is the first variable of Q .

GEN FpXY_FpXQV_evalx(GEN Q, GEN V, ulong p) Q an FpXY and x being an FpXQ, returns the FpXQX $Q(x, Y)$, where Y is the first variable of Q , assuming that V was computed by FpXQ_powers(x, n, T, p).

GEN FpXYQQ_pow(GEN x, GEN n, GEN S, GEN T, GEN p), x being a FpXY, T being a FpX and S being a FpY, return $x^n \pmod{S, T, p}$.

7.3.13 FpXk. An FpXk is a multivariate polynomial with t_INT coefficients, considered over $\mathbf{Z}/p\mathbf{Z}$.

GEN FpXk_red(GEN z, GEN p), z a ZXk, returns $\text{lift}(z * \text{Mod}(1, p))$, normalized.

GEN FpXk_gcd(GEN x, GEN y, GEN p) returns a gcd of the FpXk x and y .

int Rg_is_FpXk(GEN z, GEN *p) checks if z can be mapped to a FpXk by applying Rg_is_Fp recursively.

GEN Rg_to_FpXk(GEN z, GEN p) converts z to a FpXk by applying Rg_to_Fp recursively.

7.3.14 FpXQX, FqX. Contrary to what the name implies, an FpXQX is a t_POL whose coefficients are Fqs. So the only difference between FqX and FpXQX routines is that $T = \text{NULL}$ is not allowed in the latter. (It was thought more useful to allow t_INT components than to enforce strict consistency, which would not imply any efficiency gain.)

7.3.14.1 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_Fq_sub(GEN x, GEN y, GEN T, GEN p) subtracts 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_mul(GEN x, GEN y, GEN T, GEN p)

GEN FqX_Fq_mul(GEN x, GEN y, GEN T, GEN p) multiplies the FqX x by the Fq y .

GEN FqX_mulu(GEN x, ulong y, GEN T, GEN p) multiplies the FqX x by the scalar y .

GEN FqX_halfve(GEN x, GEN T, GEN p) returns z such that $2z = x$ assuming such z exists.

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 FpXQX_normalize(GEN z, GEN T, GEN p)`
`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_powu(GEN x, ulong n, GEN T, GEN p)`
`GEN FqX_divrem(GEN x, GEN y, GEN T, GEN p, GEN *z)`
`GEN FqX_div(GEN x, GEN y, GEN T, GEN p)`
`GEN FqX_div_by_X_x(GEN a, GEN x, GEN T, GEN p, GEN *r)`
`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`. Not stack-clean but suitable for `gc_upto`.
`GEN FqX_integ(GEN x, GEN T, GEN p)` returns the primitive of `x`. whose constant term is 0.
`GEN FqX_Fq_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_halfgcd(GEN x, GEN y, GEN T, GEN p)` returns a two-by-two `FqXM` 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 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 $\mathbf{F}_q[X][Y]$. 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 random_FpXQX(long d, long v, GEN T, GEN p)` returns a random `FpXQX` in variable v , of degree less than d .
`GEN FpXQX_renormalize(GEN x, long lx)`
`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 FpXQXV_red(GEN z, GEN T, GEN p)`, z a `t_VEC` of `ZXX`. Applies `FpX_red` componentwise and returns the result (and we obtain a vector of `FpXQXs`).
`GEN FpXQXT_red(GEN z, GEN T, GEN p)`, z a tree of `ZXX`. Applies `FpX_red` to each leaf and returns the result (and we obtain a tree of `FpXQXs`).
`GEN FpXQX_mul(GEN x, GEN y, GEN T, GEN p)`
`GEN Kronecker_to_FpXQX(GEN z, GEN T, GEN p)`. Let $n = \deg T$ and let $P(X, Y) \in \mathbf{Z}[X, Y]$ lift a polynomial in $K[Y]$, where $K := \mathbf{F}_p[X]/(T)$ and $\deg_X P < 2n - 1$ — such as would result from

multiplying minimal degree lifts of two polynomials in $K[Y]$. Let $z = P(t, t^{2*n-1})$ be a Kronecker form of P (see `RgXX_to_Kronecker`), this function returns $Q \in \mathbb{Z}[X, t]$ such that Q is congruent to $P(X, t) \bmod (p, T(X))$, $\deg_X Q < n$, and all coefficients are in $[0, p[$. Not stack-clean. Note that t need not be the same variable as Y !

```

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_div_by_X_x(GEN a, GEN x, GEN T, GEN p, GEN *r)
GEN FpXQX_rem(GEN x, GEN y, GEN T, GEN p)
GEN FpXQX_powu(GEN x, ulong n, GEN T, GEN p) returns  $x^n$ .
GEN FpXQX_digits(GEN x, GEN B, GEN T, GEN p)
GEN FpXQX_dotproduct(GEN x, GEN y, GEN T, GEN p) returns the scalar product of the coefficients of  $x$  and  $y$ .
GEN FpXQXV_FpXQX_fromdigits(GEN v, GEN B, GEN T, GEN p)
GEN FpXQX_invBarrett(GEN y, GEN T, GEN p) returns the Barrett inverse of the FpXQX  $y$ , namely a lift of  $1/\text{polrecip}(y) + O(x^{\deg(y)-1})$ .
GEN FpXQXV_prod(GEN V, GEN T, GEN p),  $V$  being a vector of FpXQX, returns their product.
GEN FpXQX_gcd(GEN x, GEN y, GEN T, GEN p)
GEN FpXQX_extgcd(GEN x, GEN y, GEN T, GEN p, GEN *ptu, GEN *ptv)
GEN FpXQX_halfgcd(GEN x, GEN y, GEN T, GEN p)
GEN FpXQX_halfgcd_all(GEN x, GEN y, GEN T, GEN p, GEN *ptu, GEN *ptv)
GEN FpXQX_resultant(GEN x, GEN y, GEN T, GEN p) returns the resultant of  $x$  and  $y$ .
GEN FpXQX_disc(GEN x, GEN T, GEN p) returns the discriminant of  $x$ .
GEN FpXQX_FpXQXQ_eval(GEN f, GEN x, GEN S, GEN T, GEN p) returns  $f(x)$ .
GEN FpXQX_FpXQ_translate(GEN P, GEN c, GEN T, GEN p)

```

7.3.15 `FpXQXn`, `FqXn`.

A `FpXQXn` is a `t_FpXQX` which represents an element of the ring $(Fp[X]/T(X))[Y]/(Y^n)$, where T is a `FpX`.

```

GEN FpXQXn_sqr(GEN x, long n, GEN T, GEN p)
GEN FqXn_sqr(GEN x, long n, GEN T, GEN p)
GEN FpXQXn_mul(GEN x, GEN y, long n, GEN T, GEN p)
GEN FqXn_mul(GEN x, GEN y, long n, GEN T, GEN p)
GEN FpXQXn_div(GEN x, GEN y, long n, GEN T, GEN p)
GEN FpXQXn_inv(GEN x, long n, GEN T, GEN p)

```

GEN FqXn_inv(GEN x, long n, GEN T, GEN p)

GEN FpXQXn_exp(GEN x, long n, GEN T, GEN p) return $\exp(x)$ as a composition of formal power series. It is required that the valuation of x is positive and that $p > n$.

GEN FqXn_exp(GEN x, long n, GEN T, GEN p)

GEN FpXQXn_expint(GEN f, long n, GEN p) return $\exp(F)$ where F is the primitive of f that vanishes at 0. It is required that $p > n$.

GEN FqXn_expint(GEN x, long n, GEN T, GEN p)

7.3.16 FpXQXQ, FqXQ.

A FpXQXQ is a $\mathbf{t_FpXQX}$ which represents an element of the ring $(Fp[X]/T(X))[Y]/S(X, Y)$, where T is a FpX and S a FpXQX modulo T . A FqXQ is identical except that T is allowed to be NULL in which case S must be a FpX.

7.3.16.1 Preconditioned reduction.

For faster reduction, the modulus S can be replaced by an extended modulus, which is an FpXQXT, in all FpXQXQ- and FqXQ-classes functions, and in FpXQX_rem and FpXQX_divrem.

GEN FpXQX_get_red(GEN S, GEN T, GEN p) returns the extended modulus eS.

GEN FqX_get_red(GEN S, GEN T, GEN p) identical, but allow T to be NULL, in which case it returns FpX_get_red(S, p).

To write code that works both with plain and extended moduli, the following accessors are defined:

GEN get_FpXQX_mod(GEN eS) returns the underlying modulus S.

GEN get_FpXQX_var(GEN eS) returns the variable number of the modulus.

GEN get_FpXQX_degree(GEN eS) returns the degree of the modulus.

Furthermore, ZXXT_to_FlxXT allows to convert an extended modulus for a FpXQX to an extended modulus for the corresponding FlxqX.

7.3.16.2 basic operations.

GEN FpXQX_FpXQXQV_eval(GEN f, GEN V, GEN S, GEN T, GEN p) returns $f(x)$, assuming that V was computed by FpXQXQ_powers(x, n, S, T, p).

GEN FpXQXQ_div(GEN x, GEN y, GEN S, GEN T, GEN p), x, y and S being FpXQXs, returns $x * y^{-1}$ modulo S .

GEN FpXQXQ_inv(GEN x, GEN S, GEN T, GEN p), x and S being FpXQXs, returns x^{-1} modulo S .

GEN FpXQXQ_invsafe(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 FpXQXs, returns xy modulo S .

GEN FpXQXQ_sqr(GEN x, GEN S, GEN T, GEN p), x and S being FpXQXs, returns x^2 modulo S .

GEN FpXQXQ_pow(GEN x, GEN n, GEN S, GEN T, GEN p), x and S being FpXQXs, returns x^n modulo S .

GEN FpXQXQ_powers(GEN x, long n, GEN S, GEN T, GEN p), x and S being FpXQXs, returns $[x^0, \dots, x^n]$ as a t_VEC of FpXQXs.

GEN FpXQXQ_halfFrobenius(GEN A, GEN S, GEN T, GEN p) returns $A(X)^{(q-1)/2} \pmod{S(X)}$ over the finite field $\mathbf{F}_q$ defined by T and p , thus $q = p^n$ where n is the degree of T .

GEN FpXQXQ_minpoly(GEN x, GEN S, GEN T, GEN p), as FpXQ_minpoly

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(x, n-1, S, T, p), but as an $m \times n$ matrix.

GEN FpXQXQ_autpow(GEN a, long n, GEN S, GEN T, GEN p) σ being the automorphism defined by $\sigma(X) = a[1] \pmod{T(X)}$, $\sigma(Y) = a[2] \pmod{S(X, Y), T(X)}$, returns $[\sigma^n(X), \sigma^n(Y)]$.

GEN FpXQXQ_autsum(GEN a, long n, GEN S, GEN T, GEN p) σ being the automorphism defined by $\sigma(X) = a[1] \pmod{T(X)}$, $\sigma(Y) = a[2] \pmod{S(X, Y), T(X)}$, returns the vector $[\sigma^n(X), \sigma^n(Y), b\sigma(b) \dots \sigma^{n-1}(b)]$ where $b = a[3]$.

GEN FpXQXQ_auttrace(GEN a, long n, GEN S, GEN T, GEN p) σ being the automorphism defined by $\sigma(X) = X \pmod{T(X)}$, $\sigma(Y) = a[1] \pmod{S(X, Y), T(X)}$, returns the vector $[\sigma^n(X), \sigma^n(Y), b + \sigma(b) + \dots + \sigma^{n-1}(b)]$ where $b = a[2]$.

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 $[x^0, \dots, x^n]$ as a t_VEC of FqXs.

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 FqV_roots_to_pol(GEN V, GEN T, GEN p, long v), V being a vector of Fqs, returns the monic FqX $\prod_i (\text{pol_x}[v] - V[i])$.

7.3.16.3 Miscellaneous operations.

GEN `init_Fq`(GEN `p`, long `n`, long `v`) returns an irreducible polynomial of degree $n > 0$ over $\mathbf{F}_p$, in variable `v`.

int `FpXQX_is_squarefree`(GEN `P`, GEN `T`, GEN `p`) returns 1 if P is squarefree, 0 otherwise.

int `FqX_is_squarefree`(GEN `P`, GEN `T`, GEN `p`)

GEN `FpXQX_roots`(GEN `f`, GEN `T`, GEN `p`) return the roots of f in $\mathbf{F}_p[X]/(T)$. Assumes p is prime and T irreducible in $\mathbf{F}_p[X]$.

GEN `FpXQX_roots_mult`(GEN `f`, long `n`, GEN `T`, GEN `p`) returns the roots in $\mathbf{Z}/p\mathbf{Z}$ with multiplicity at least n of the `FpXQX` `f` (without multiplicity, as a vector of `FpXQs`). Assumes p is prime and T irreducible in $\mathbf{F}_p[X]$.

GEN `FqX_roots`(GEN `f`, GEN `T`, GEN `p`) same but allow $T = \text{NULL}$.

GEN `FpXQX_factor`(GEN `f`, GEN `T`, GEN `p`) same output convention as `FpX_factor`. Assumes p is prime and T irreducible in $\mathbf{F}_p[X]$.

GEN `FqX_factor`(GEN `f`, GEN `T`, GEN `p`) same but allow $T = \text{NULL}$.

GEN `FpXQX_factor_squarefree`(GEN `f`, GEN `T`, GEN `p`) squarefree factorization of f modulo (T, p) ; same output convention as `FpX_factor_squarefree`. Assumes p is prime and T irreducible in $\mathbf{F}_p[X]$.

GEN `FqX_factor_squarefree`(GEN `f`, GEN `T`, GEN `p`) same but allow $T = \text{NULL}$.

GEN `FpXQX_ddf`(GEN `f`, GEN `T`, GEN `p`) as `FpX_ddf`.

GEN `FqX_ddf`(GEN `f`, GEN `T`, GEN `p`) same but allow $T = \text{NULL}$.

long `FpXQX_ddf_degree`(GEN `f`, GEN `XP`, GEN `T`, GEN `p`), as `FpX_ddf_degree`.

GEN `FpXQX_degfact`(GEN `f`, GEN `T`, GEN `p`), as `FpX_degfact`.

GEN `FqX_degfact`(GEN `f`, GEN `T`, GEN `p`) same but allow $T = \text{NULL}$.

GEN `FpXQX_split_part`(GEN `f`, GEN `T`, GEN `p`) returns the largest totally split squarefree factor of f .

long `FpXQX_ispower`(GEN `f`, ulong `k`, GEN `T`, GEN `p`, GEN `*pt`) return 1 if the `FpXQX` f is a k -th power, 0 otherwise. If `pt` is not `NULL`, set it to g such that $g^k = f$.

long `FqX_ispower`(GEN `f`, ulong `k`, GEN `T`, GEN `p`, GEN `*pt`) same but allow $T = \text{NULL}$.

GEN `FpX_factorff`(GEN `P`, GEN `T`, GEN `p`). Assumes p prime and T irreducible in $\mathbf{F}_p[X]$. Factor the `FpX` P over the finite field $\mathbf{F}_p[Y]/(T(Y))$. See `FpX_factorff_irred` if P is known to be irreducible of $\mathbf{F}_p$.

GEN `FpX_rootsff`(GEN `P`, GEN `T`, GEN `p`). Assumes p prime and T irreducible in $\mathbf{F}_p[X]$. Returns the roots of the `FpX` P belonging to the finite field $\mathbf{F}_p[Y]/(T(Y))$.

GEN `FpX_factorff_irred`(GEN `P`, GEN `T`, GEN `p`). Assumes p prime and T irreducible in $\mathbf{F}_p[X]$. Factors the *irreducible* `FpX` P over the finite field $\mathbf{F}_p[Y]/(T(Y))$ and returns the vector of irreducible `FqXs` factors (the exponents, being all equal to 1, are not included).

GEN `FpX_ffisom`(GEN `P`, GEN `Q`, GEN `p`). Assumes p prime, P, Q are `ZXs`, both irreducible mod p , and $\deg(P) \mid \deg(Q)$. Outputs a monomorphism between $\mathbf{F}_p[X]/(P)$ and $\mathbf{F}_p[X]/(Q)$, as a polynomial R such that $Q \mid P(R)$ in $\mathbf{F}_p[X]$. If P and Q have the same degree, it is of course an isomorphism.

`void FpX_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 $\mathbf{F}_p[X]/(P)$ generated by SP and the subfield of $\mathbf{F}_p[X]/(Q)$ 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 $\mathbf{F}_p[X]/(P)$ (resp. $\mathbf{F}_p[X]/(Q)$).

`GEN FpXQ_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 $\mathbf{F}_q := \mathbf{F}_p[X]/(T)$. ($S(X)$ is the image of X by the automorphism.) Returns the inverse automorphism of S , in the same format, i.e. an $\mathbf{FpX}$ H such that $H(S) \equiv X$ modulo (T, p) .

`long FpXQX_nbfact(GEN S, GEN T, GEN p)` returns the number of irreducible factors of the polynomial S over the finite field $\mathbf{F}_q$ defined by T and p .

`long FpXQX_nbfact_Frobenius(GEN S, GEN Xq, GEN T, GEN p)` as `FpXQX_nbfact` where Xq is `FpXQX_Frobenius(S, T, p)`.

`long FqX_nbfact(GEN S, GEN T, GEN p)` as above but accept $T=NULL$.

`long FpXQX_nbroots(GEN S, GEN T, GEN p)` returns the number of roots of the polynomial S over the finite field $\mathbf{F}_q$ defined by T and p .

`long FqX_nbroots(GEN S, GEN T, GEN p)` as above but accept $T=NULL$.

`GEN FpXQX_Frobenius(GEN S, GEN T, GEN p)` returns $X^q \pmod{S(X)}$ over the finite field $\mathbf{F}_q$ defined by T and p , thus $q = p^n$ where n is the degree of T .

7.3.17 Flx. Let p be an `ulong`, not assumed to be prime unless mentioned otherwise (e.g., all functions involving Euclidean divisions and factorizations), to be given the function arguments; an `Fl` is an `ulong` belonging to $[0, p-1]$, an `Flx` z is a `t_VECSMALL` representing a polynomial with small integer coefficients. Specifically $z[0]$ is the usual codeword, $z[1] = \text{evalvarn}(v)$ for some variable v , then the coefficients by increasing degree. An `FlxX` is a `t_POL` whose coefficients are `Flxs`.

In the following, an argument called `sv` is of the form `evalvarn(v)` for some variable number v .

7.3.17.1 Preconditioned reduction.

For faster reduction, the modulus T can be replaced by an extended modulus (`FlxT`) in all `Flxq`-classes functions, and in `Flx_divrem`.

`GEN Flx_get_red(GEN T, ulong p)` returns the extended modulus `eT`.

`GEN Flx_get_red_pre(GEN T, ulong p, ulong pi)` as `Flx_get_red`. We assume pi is the pseudoinverse of p , or 0 in which case we assume `SMALL_ULONG(p)`.

To write code that works both with plain and extended moduli, the following accessors are defined:

`GEN get_Flx_mod(GEN eT)` returns the underlying modulus T .

`GEN get_Flx_var(GEN eT)` returns the variable number of the modulus.

`GEN get_Flx_degree(GEN eT)` returns the degree of the modulus.

Furthermore, `ZXT.to_FlxT` allows to convert an extended modulus for a `FpX` to an extended modulus for the corresponding `Flx`.

7.3.17.2 Basic operations.

In this section, pi is the pseudoinverse of p , or 0 in which case we assume $SMALL_ULONG(p)$.

`ulong Flx_lead(GEN x)` returns the leading coefficient of x as a `ulong` (return 0 for the zero polynomial).

`ulong Flx_constant(GEN x)` returns the constant coefficient of x as a `ulong` (return 0 for the zero polynomial).

`GEN Flx_red(GEN z, ulong p)` converts from `zx` with nonnegative coefficients to `Flx` (by reducing them mod p).

`int Flx_equal1(GEN x)` returns 1 (true) if the `Flx` x is equal to 1, 0 (false) otherwise.

`int Flx_equal(GEN x, GEN y)` returns 1 (true) if the `Flx` x and y are equal, and 0 (false) otherwise.

`GEN Flx_copy(GEN x)` returns a copy of x .

`GEN Flx_add(GEN x, GEN y, ulong p)`

`GEN Flx_Fl_add(GEN y, ulong x, ulong p)`

`GEN Flx_neg(GEN x, ulong p)`

`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_Fl_sub(GEN y, ulong x, ulong p)`

`GEN Flx_half(GEN x, ulong p)` returns z such that $2z = x$ modulo p assuming such z exists.

`GEN Flx_mul(GEN x, GEN y, ulong p)`

`GEN Flx_mul_pre(GEN x, GEN y, ulong p, ulong pi)`

`GEN Flx_Fl_mul(GEN y, ulong x, ulong p)`

`GEN Flx_Fl_mul_pre(GEN y, ulong x, ulong p, ulong pi)`

`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 $x \neq 0$).

`GEN Flx_sqr(GEN x, ulong p)`

`GEN Flx_sqr_pre(GEN x, ulong p, ulong pi)`

`GEN Flx_powu(GEN x, ulong n, ulong p)` return x^n .

`GEN Flx_powu_pre(GEN x, ulong n, ulong p, ulong pi)`

`GEN Flx_conv(GEN x, GEN y, ulong p)` return the-term by-term product of x and y .

`GEN Flx_divrem(GEN x, GEN y, ulong p, GEN *pr)`, here p must be prime.

`GEN Flx_divrem_pre(GEN x, GEN y, ulong p, ulong pi, GEN *pr)`

`GEN Flx_div(GEN x, GEN y, ulong p)`, here p must be prime.
`GEN Flx_div_pre(GEN x, GEN y, ulong p, ulong pi)`
`GEN Flx_rem(GEN x, GEN y, ulong p)`, here p must be prime.
`GEN Flx_rem_pre(GEN x, GEN y, ulong p, ulong pi)`
`GEN Flx_deriv(GEN z, ulong p)`
`GEN Flx_integ(GEN z, ulong p)`, here p must be prime.
`GEN Flx_Fl_translate(GEN P, ulong c, ulong p)` return $P(x + c)$, p must be prime. Asymptotically fast (quasi-linear in the degree of P).
`GEN Flx_translate1(GEN P, ulong p)` return $P(x + 1)$, p must be prime. Asymptotically fast (quasi-linear in the degree of P).
`GEN Flx_translate1_basecase(GEN P, ulong p)` return $P(x + 1)$, p need not be prime. Not asymptotically fast (quadratic in the degree of P).
`GEN zlx_translate1(GEN P, ulong p, long e)` return $P(x + 1)$ modulo p^e for prime p . Asymptotically fast (quasi-linear in the degree of P).
`GEN Flx_diff1(GEN P, ulong p)` return $P(x + 1) - P(x)$; p must be prime.
`GEN Flx_digits(GEN x, GEN B, ulong p)` returns a vector of `Flx` $[c_0, \dots, c_n]$ of degree less than the degree of B and such that $x = \sum_{i=0}^n c_i B^i$.
`GEN FlxV_Flx_fromdigits(GEN v, GEN B, ulong p)` where $v = [c_0, \dots, c_n]$ is a vector of `Flx`, returns $\sum_{i=0}^n c_i B^i$.
`GEN Flx_Frobenius(GEN T, ulong p)` here p must be prime.
`GEN Flx_Frobenius_pre(GEN T, ulong p, ulong pi)`
`GEN Flx_matFrobenius(GEN T, ulong p)` here p must be prime.
`GEN Flx_matFrobenius_pre(GEN T, ulong p, ulong pi)`
`GEN Flx_gcd(GEN a, GEN b, ulong p)` returns a (not necessarily monic) greatest common divisor of x and y . Here p must be prime.
`GEN Flx_gcd_pre(GEN a, GEN b, ulong p)`
`GEN Flx_halfgcd(GEN x, GEN y, ulong 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 $\deg a \geq \frac{\deg x}{2} > \deg b$. Assumes that p is prime.
`GEN Flx_halfgcd_pre(GEN a, GEN b, ulong p)`
`GEN Flx_halfgcd_all(GEN x, GEN y, ulong p, GEN *pt_a, GEN *pt_b)`
`GEN Flx_halfgcd_all_pre(GEN x, GEN y, ulong p, GEN *pt_a, GEN *pt_b)`
`GEN Flx_extgcd(GEN a, GEN b, ulong p, GEN *ptu, GEN *ptv)`, here p must be prime.
`GEN Flx_extgcd_pre(GEN a, GEN b, ulong p, ulong pi, GEN *ptu, GEN *ptv)`
`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.

`GEN Flx_roots_pre(GEN f, ulong p, ulong pi)`
`ulong Flx_oneroot(GEN f, ulong p)` returns one root $0 \leq r < p$ of the Flx f in $\mathbf{Z}/p\mathbf{Z}$. Return p if no root exists. Assumes that p is prime.
`GEN Flx_oneroot_pre(GEN f, ulong p)`, as `Flx_oneroot`
`ulong Flx_oneroot_split(GEN f, ulong p)` as `Flx_oneroot` but assume f is totally split. Assumes that p is prime.
`ulong Flx_oneroot_split_pre(GEN f, ulong p, ulong pi)`
`long Flx_ispower(GEN f, ulong k, ulong p, GEN *pt)` return 1 if the Flx f is a k -th power, 0 otherwise. If `pt` is not NULL, set it to g such that $g^k = f$.
`GEN Flx_factor(GEN f, ulong p)` Assumes that p is prime.
`GEN Flx_ddf(GEN f, ulong p)` Assumes that p is prime.
`GEN Flx_ddf_pre(GEN f, ulong p, ulong pi)`
`GEN Flx_factor_squarefree(GEN f, ulong p)` returns the squarefree factorization of f modulo p . This is a vector $[u_1, \dots, u_k]$ of pairwise coprime Flx such that $u_k \neq 1$ and $f = \prod u_i^i$. Shallow function. Assumes that p is prime.
`GEN Flx_factor_squarefree_pre(GEN f, ulong p, ulong pi)`
`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`. Assumes that p is prime.
`GEN Flx_factorff_irred(GEN P, GEN Q, ulong p)` as `FpX_factorff_irred`. Assumes that p is prime.
`GEN Flx_rootsff(GEN P, GEN T, ulong p)` as `FpX_rootsff`. Assumes that p is prime.
`GEN Flx_factcyclo(ulong n, ulong p, ulong m)` returns the factors of the n -th cyclotomic polynomial over $\mathbf{F}_p$. if $m = 1$ returns a single factor.
`GEN Flx_ffisom(GEN P, GEN Q, ulong l)` as `FpX_ffisom`. Assumes that p is prime.

7.3.17.3 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 polxn_Flx(long n, long sv)` Returns the monomial of degree n as a Flx in variable v ; assume that $n \geq 0$.
`GEN monomial_Flx(ulong a, long d, long sv)` returns the Flx aX^d in variable v .
`GEN init_Flxq(ulong p, long n, long sv)` returns an irreducible polynomial of degree $n > 0$ over $\mathbf{F}_p$, in variable v .
`GEN Flx_normalize(GEN z, ulong p)`, as `FpX_normalize`.

GEN Flx_rescale(GEN P, ulong h, ulong p) returns $h^{\deg(P)}P(x/h)$, P is a Flx and h is a nonzero integer.

GEN Flx_unscale(GEN P, ulong h, ulong p) returns $P(hx)$, P is a Flx and h is a nonzero integer.

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. Assumes that p is prime.

ulong Flx_resultant_pre(GEN a, GEN b, ulong p, ulong pi)

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). Assumes that p is prime.

ulong Flx_extresultant_pre(GEN a, GEN b, ulong p, ulong pi, GEN *ptU, GEN *ptV)

GEN Flx_composedprod(GEN P, GEN Q, ulong p) if $P = a \prod_{i=1}^m (x - p_i)$ and $Q = b \prod_{j=1}^n (x - q_j)$ in some suitable algebraic extension, return $a^n b^m \prod_{i,j} (x - p_i q_j)$.

GEN Flx_composedsum(GEN P, GEN Q, ulong p) if $P = a \prod_{i=1}^m (x - p_i)$ and $Q = b \prod_{j=1}^n (x - q_j)$ in some suitable algebraic extension, return $a^n b^m \prod_{i,j} (x - (p_i + q_j))$.

GEN Flx_invBarrett(GEN T, ulong p), returns the Barrett inverse M of T defined by $M(x) \times x^n T(1/x) \equiv 1 \pmod{x^{n-1}}$ where n is the degree of T. Assumes that p is prime.

GEN Flx_renormalize(GEN x, long l), as FpX_renormalize, where $l = \lg(x)$, in place.

GEN Flx_shift(GEN T, long n) returns $T * x^n$ if $n \geq 0$, and $T \setminus x^{-n}$ 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.

ulong Flx_eval_pre(GEN x, ulong y, ulong p, ulong pi)

ulong Flx_eval_powers_pre(GEN P, GEN y, ulong p, ulong pi). Let y be the t_VEC SMALL $(1, a, \dots, a^n)$, where n is the degree of the Flx P, return $P(a)$.

GEN Flx_Flv_multieval(GEN P, GEN v, ulong p) returns the vector $[P(v[1]), \dots, P(v[n])]$ as a Flv.

ulong Flx_dotproduct(GEN x, GEN y, ulong p) returns the scalar product of the coefficients of x and y.

ulong Flx_dotproduct_pre(GEN x, GEN y, ulong p, ulong pi).

GEN Flx_deflate(GEN P, long d) assuming P is a polynomial of the form $Q(X^d)$, return Q.

GEN Flx_inflate(GEN P, long d) returns $P(X^d)$.

`GEN Flx_splitting(GEN P, long k), as RgX_splitting.`
`GEN Flx_blocks(GEN P, long n, long m), as RgX_blocks.`
`int Flx_is_squarefree(GEN z, ulong p).` Assumes that p is prime.
`int Flx_is_irred(GEN f, ulong p), as FpX_is_irred.` Assumes that p is prime.
`int Flx_is_totally_split(GEN f, ulong p)` returns 1 if the `Flx` f splits into a product of distinct linear factors, 0 otherwise. Assumes that p is prime.
`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. Assumes that p is prime.
`int Flx_is_smooth_pre(GEN f, long r, ulong p, ulong pi)`
`long Flx_nbroots(GEN f, ulong p), as FpX_nbroots.` Assumes that p is prime.
`long Flx_nbfact(GEN z, ulong p), as FpX_nbfact.` Assumes that p is prime.
`long Flx_nbfact_pre(GEN z, ulong p, ulong pi)`
`long Flx_nbfact_Frobenius(GEN f, GEN XP, ulong p), as FpX_nbfact_Frobenius.` Assumes that p is prime.
`long Flx_nbfact_Frobenius_pre(GEN f, GEN XP, ulong p, ulong pi)`
`GEN Flx_degfact(GEN f, ulong p), as FpX_degfact.` Assumes that p is prime.
`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_VEC SMALL` 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 $D[i]$). Assumes that p is prime.
`void Flx_ffintersect(GEN P, GEN Q, long n, ulong p, GEN*SP, GEN*SQ, GEN MA, GEN MB)`
`,`
`as FpX_ffintersect.` Assumes that p is prime.
`GEN Flx_Laplace(GEN x, ulong p)`
`GEN Flx_invLaplace(GEN x, ulong p)`
`GEN Flx_Newton(GEN x, long n, ulong p)`
`GEN Flx_fromNewton(GEN x, ulong p)`
`GEN Flx_Teichmuller(GEN P, ulong p, long n)` Return a `ZX` Q such that $P \equiv Q \pmod{p}$ and $Q(X^p) = 0 \pmod{Q, p^n}$. Assumes that p is prime.
`GEN Flv_polint(GEN x, GEN y, ulong p, long sv)` as `FpV_polint`, returning an `Flx` in variable v . Assumes that p is prime.
`GEN Flv_Flm_polint(GEN x, GEN V, ulong p, long sv)` equivalent (but faster) to applying `Flv_polint(x,...)` to all the elements of the vector V (thus, returns a `FlxV`). Assumes that p is prime.
`GEN Flv_invVandermonde(GEN L, ulong d, ulong p)` L being a `Flv` of length n , return the inverse M of the Vandermonde matrix attached to the elements of L , multiplied by d . If A is a `Flv` and $B = MA$, then the polynomial $P = \sum_{i=1}^n B[i]X^{i-1}$ verifies $P(L[i]) = dA[i]$ for $1 \leq i \leq n$. Assumes that p is prime.
`GEN Flv_roots_to_pol(GEN a, ulong p, long sv)` as `FpV_roots_to_pol` returning an `Flx` in variable v .

7.3.18 FlxV. See FpXV operations.

GEN FlxV_Flc_mul(GEN V, GEN W, ulong p), as FpXV_FpC_mul.

GEN FlxV_red(GEN V, ulong p) reduces each components with Flx_red.

GEN FlxV_prod(GEN V, ulong p), V being a vector of Flx, returns their product.

GEN FlxV_factorback(GEN L, GEN e, ulong p, long v) returns $\prod_i L_i^{e_i}$ where L is a vector of Flxs in the shifted variable v and e a vector of non-negative t_INTs or a t_VECSMALL.

GEN FlxV_composedsum(GEN V, ulong p), V being a vector of Flx, returns their composed sum, see Flx_composedsum.

ulong FlxC_eval_powers_pre(GEN x, GEN y, ulong p, ulong pi) apply Flx_eval_powers_pre to all elements of x.

GEN FlxV_Flv_multieval(GEN F, GEN v, ulong p) assuming F is a vector of Flx with m entries and v is a Flv with m entries, returns the n -components vector (FlvV) whose j -th entry is $[F_j(v[1]), \dots, F_j(v[n])]$, with $F_j = F[j]$.

GEN FlxC_neg(GEN x, ulong p)

GEN FlxC_sub(GEN x, GEN y, ulong p)

GEN FlxC_Fl_translate(GEN x, ulong c, ulong p)

GEN zero_FlxC(long n, long sv)

7.3.19 FlxM. See FpXM operations.

ulong FlxM_eval_powers_pre(GEN M, GEN y, ulong p, ulong pi) this function applies FlxC_eval_powers_pre to all entries of M.

GEN FlxM_neg(GEN x, ulong p)

GEN FlxM_sub(GEN x, GEN y, ulong p)

GEN FlxM_mul(GEN x, GEN y, ulong p)

GEN FlxM_Fl_translate(GEN x, ulong c, ulong p)

GEN zero_FlxM(long r, long c, long sv)

GEN zero_FlxM_copy(long r, long c, long sv)

7.3.20 FlxT. See FpXT operations.

GEN FlxT_red(GEN V, ulong p) reduces each leaf with Flx_red.

7.3.21 Flxn. See FpXn operations. In this section, pi is the pseudoinverse of p , or 0 in which case we assume $\text{SMALL_ULONG}(p)$.

`GEN Flxn_mul(GEN a, GEN b, long n, ulong p)` returns ab modulo X^n .

`GEN Flxn_mul_pre(GEN a, GEN b, long n, ulong p, ulong pi)`

`GEN Flxn_sqr(GEN a, long n, ulong p)` returns a^2 modulo X^n .

`GEN Flxn_sqr_pre(GEN a, long n, ulong p, ulong pi)`

`GEN Flxn_inv(GEN a, long n, ulong p)` returns $1/a$ modulo X^n .

`GEN Flxn_div(GEN a, GEN b, long n, ulong p)` returns a/b modulo X^n .

`GEN Flxn_div_pre(GEN a, GEN b, long n, ulong p, ulong pi)`

`GEN Flxn_red(GEN a, long n)` returns a modulo X^n .

`GEN Flxn_exp(GEN x, long n, ulong p)` return $\exp(x)$ as a composition of formal power series. It is required that the valuation of x is positive and that $p > n$.

`GEN Flxn_expint(GEN f, long n, ulong p)` return $\exp(F)$ where F is the primitive of f that vanishes at 0. It is required that $p > n$.

7.3.22 Flxq. See FpXQ operations. In this section, pi is the pseudoinverse of p , or 0 in which case we assume $\text{SMALL_ULONG}(p)$.

`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_mul_pre(GEN x, GEN y, GEN T, ulong p, ulong pi)`

`GEN Flxq_sqr(GEN y, GEN T, ulong p)`

`GEN Flxq_sqr_pre(GEN y, GEN T, ulong p)`

`GEN Flxq_inv(GEN x, GEN T, ulong p)`

`GEN Flxq_inv_pre(GEN x, GEN T, ulong p, ulong pi)`

`GEN Flxq_invsafe(GEN x, GEN T, ulong p)`

`GEN Flxq_invsafe_pre(GEN x, GEN T, ulong p, ulong pi)`

`GEN Flxq_div(GEN x, GEN y, GEN T, ulong p)`

`GEN Flxq_div_pre(GEN x, GEN y, GEN T, ulong p, ulong pi)`

`GEN Flxq_pow(GEN x, GEN n, GEN T, ulong p)`

`GEN Flxq_pow_pre(GEN x, GEN n, GEN T, ulong p, ulong pi)`

`GEN Flxq_powu(GEN x, ulong n, GEN T, ulong p)`

`GEN Flxq_powu_pre(GEN x, ulong n, GEN T, ulong p)`

`GEN FlxqV_factorback(GEN L, GEN e, GEN Tp, ulong p)`

`GEN Flxq_pow_init(GEN x, GEN n, long k, GEN T, ulong p)`

GEN Flxq_pow_init_pre(GEN x, GEN n, long k, GEN T, ulong p, ulong pi)
 GEN Flxq_pow_table(GEN R, GEN n, GEN T, ulong p)
 GEN Flxq_pow_table_pre(GEN R, GEN n, GEN T, ulong p, ulong pi)
 GEN Flxq_powers(GEN x, long n, GEN T, ulong p)
 GEN Flxq_powers_pre(GEN x, long n, GEN T, ulong p, ulong pi)
 GEN Flxq_matrix_pow(GEN x, long m, long n, GEN T, ulong p), see FpXQ_matrix_pow.
 GEN Flxq_matrix_pow_pre(GEN x, long m, long n, GEN T, ulong p, ulong pi)
 GEN Flxq_autpow(GEN a, long n, GEN T, ulong p) see FpXQ_autpow.
 GEN Flxq_autpow_pre(GEN a, long n, GEN T, ulong p, ulong pi)
 GEN Flxq_autpowers(GEN a, long n, GEN T, ulong p) return $[X, \sigma(X), \dots, \sigma^n(X)]$, assuming $a = \sigma(X)$ where σ is an automorphism of the algebra $\mathbf{F}_p[X]/T(X)$.
 GEN Flxq_autsum(GEN a, long n, GEN T, ulong p) see FpXQ_autsum.
 GEN Flxq_auttrace(GEN a, ulong n, GEN T, ulong p) see FpXQ_auttrace.
 GEN Flxq_auttrace_pre(GEN a, ulong n, GEN T, ulong p, ulong pi)
 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_Flxq_eval_pre(GEN f, GEN x, GEN T, ulong p, ulong pi)
 GEN Flx_FlxqV_eval(GEN f, GEN x, GEN T, ulong p), see FpX_FpXQV_eval.
 GEN Flx_FlxqV_eval_pre(GEN f, GEN x, GEN T, ulong p, ulong pi)
 GEN FlxC_Flxq_eval(GEN C, GEN x, GEN T, ulong p), see FpXC_FpXQ_eval.
 GEN FlxC_Flxq_eval_pre(GEN C, GEN x, GEN T, ulong p, ulong pi)
 GEN FlxC_FlxqV_eval(GEN C, GEN V, GEN T, ulong p) see FpXC_FpXQV_eval.
 GEN FlxC_FlxqV_eval_pre(GEN C, GEN V, GEN T, ulong p, ulong pi)
 GEN FlxqV_roots_to_pol(GEN V, GEN T, ulong p, long v) as FqV_roots_to_pol returning an FlxqX in variable v .
 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_order(GEN a, GEN ord, GEN T, ulong p) as FpXQ_order.
 GEN Flxq_log(GEN a, GEN g, GEN ord, GEN T, ulong p) as FpXQ_log
 GEN Flxq_sqrtn(GEN x, GEN n, GEN T, ulong p, GEN *zn) as FpXQ_sqrtn.
 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_sqrt_pre(GEN x, GEN T, ulong p, ulong pi)

`GEN Flxq_root(GEN a, GEN T, ulong p)` returns x such that $x^p = a$.
`GEN Flxq_root_pre(GEN a, GEN T, ulong p, ulong pi)`
`GEN Flxq_root_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_root_fast_pre(GEN a, GEN V, GEN T, ulong p, ulong pi)`
`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
`GEN Flxq_minpoly_pre(GEN x, GEN T, ulong p, ulong pi)`
`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 $[x, x^p, x^{p^2}, \dots, x^{p^{n-1}}]$ 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.

7.3.23 FlxX. See `FpXX` operations. In this section, we assume pi is the pseudoinverse of p , or 0 in which case we assume `SMALL_ULONG(p)`.

`GEN pol1_FlxX(long vX, long sx)` returns the unit `FlxX` as a `t_POL` in variable vX which only coefficient is `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 .
`long FlxY_degreeex(GEN P)` return the degree of P with respect to the secondary variable.
`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 FlxY_Flx_add(GEN x, GEN y, ulong p)`
`GEN FlxY_Flx_sub(GEN x, GEN y, ulong p)`
`GEN FlxY_Flx_mul(GEN x, GEN y, ulong p)`
`GEN FlxY_Flx_mul_pre(GEN x, GEN y, ulong p, ulong pi)`
`GEN FlxY_Flx_div(GEN x, GEN y, ulong p)` divides the coefficients of x by y using `Flx_div`.
`GEN FlxY_Flx_div_pre(GEN x, GEN y, ulong p, ulong pi)`

`GEN FlxX_deriv(GEN P, ulong p)` returns the derivative of P with respect to the main variable.
`GEN FlxX_Laplace(GEN x, ulong p)`
`GEN FlxX_invLaplace(GEN x, ulong p)`
`GEN FlxY_evalx(GEN P, ulong z, ulong p)` P being an FlxY , returns the Flx $P(z, Y)$, where Y is the main variable of P .
`GEN FlxY_evalx_pre(GEN P, ulong z, ulong p, ulong pi)`
`GEN FlxX_translate1(GEN P, ulong p, long n)` P being an FlxX with all coefficients of degree at most n , return $(P(x, Y + 1))$, where Y is the main variable of P .
`GEN zlxX_translate1(GEN P, ulong p, long e, long n)` P being an zlxX with all coefficients of degree at most n , return $(P(x, Y + 1))$ modulo p^e for prime p , where Y is the main variable of P .
`GEN FlxY_Flx_translate(GEN P, GEN f, ulong p)` P being an FlxY and f being an Flx , return $(P(x, Y + f(x)))$, where Y is the main variable of P .
`ulong FlxY_evalx_powers_pre(GEN P, GEN xp, ulong p, ulong pi)`, xp being the vector $[1, x, \dots, x^n]$, where n is larger or equal to the degree of P in X , return $P(x, Y)$, where Y is the main variable of Q .
`ulong FlxY_eval_powers_pre(GEN P, GEN xp, GEN yp, ulong p, ulong pi)`, xp being the vector $[1, x, \dots, x^n]$, where n is larger or equal to the degree of P in X and yp being the vector $[1, y, \dots, y^m]$, where m is larger or equal to the degree of P in Y return $P(x, y)$.
`GEN FlxY_Flxq_evalx(GEN x, GEN y, GEN T, ulong p)` as FpXY_FpXQ_evalx .
`GEN FlxY_Flxq_evalx_pre(GEN x, GEN y, GEN T, ulong p, ulong pi)`
`GEN FlxY_FlxqV_evalx(GEN x, GEN V, GEN T, ulong p)` as FpXY_FpXQV_evalx .
`GEN FlxY_FlxqV_evalx_pre(GEN x, GEN V, GEN T, ulong p, ulong pi)`
`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 $\text{Res}_X(u, v)$, which is an Flx . The coefficients of u and v are assumed to be in the variable v .
`GEN FlxX_resultant_pre(GEN u, GEN v, ulong p, ulong pi, long sv)`
`GEN Flx_FlxY_resultant(GEN a, GEN b, ulong p)` Returns $\text{Res}_x(a, b)$, which is an Flx in the main variable of b .
`GEN FlxX_blocks(GEN P, long n, long m, long sv)`, as RgX_blocks , where v is the secondary variable.
`GEN FlxX_shift(GEN a, long n, long sv)`, as RgX_shift_shallow , where v is the secondary variable.
`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 .
7.3.24 FlxXV , FlxXC , FlxXM . See FpXX operations.
`GEN FlxXC_sub(GEN x, GEN y, ulong p)`

7.3.25 FlxqX. See FpXQX operations.

7.3.25.1 Preconditioned reduction.

For faster reduction, the modulus S can be replaced by an extended modulus, which is an FlxqXT , in all FlxqXQ -classes functions, and in FlxqX_rem and FlxqX_divrem .

$\text{GEN FlxqX_get_red}(\text{GEN } S, \text{GEN } T, \text{ulong } p)$ returns the extended modulus eS .

$\text{GEN FlxqX_get_red_pre}(\text{GEN } S, \text{GEN } T, \text{ulong } p, \text{ulong } pi)$, where pi is a pseudoinverse of p , or 0 in which case we assume $\text{SMALL_ULONG}(p)$.

To write code that works both with plain and extended moduli, the following accessors are defined:

$\text{GEN get_FlxqX_mod}(\text{GEN } eS)$ returns the underlying modulus S .

$\text{GEN get_FlxqX_var}(\text{GEN } eS)$ returns the variable number of the modulus.

$\text{GEN get_FlxqX_degree}(\text{GEN } eS)$ returns the degree of the modulus.

7.3.25.2 basic functions.

In this section, pi is a pseudoinverse of p , or 0 in which case we assume $\text{SMALL_ULONG}(p)$.

$\text{GEN random_FlxqX}(\text{long } d, \text{long } v, \text{GEN } T, \text{ulong } p)$ returns a random FlxqX in variable v , of degree less than d .

$\text{GEN zxX_to_Kronecker}(\text{GEN } P, \text{GEN } Q)$ 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 .

$\text{GEN Kronecker_to_FlxqX}(\text{GEN } z, \text{GEN } T, \text{ulong } p)$. Let $n = \deg T$ and let $P(X, Y) \in \mathbf{Z}[X, Y]$ lift a polynomial in $K[Y]$, where $K := \mathbf{F}_p[X]/(T)$ and $\deg_X P < 2n-1$ — such as would result from multiplying minimal degree lifts of two polynomials in $K[Y]$. Let $z = P(t, t^{2*n-1})$ be a Kronecker form of P , this function returns $Q \in \mathbf{Z}[X, t]$ such that Q is congruent to $P(X, t) \pmod{(p, T(X))}$, $\deg_X Q < n$, and all coefficients are in $[0, p[$. Not stack-clean. Note that t need not be the same variable as Y !

$\text{GEN Kronecker_to_FlxqX_pre}(\text{GEN } z, \text{GEN } T, \text{ulong } p, \text{ulong } pi)$

$\text{GEN FlxqX_red}(\text{GEN } z, \text{GEN } T, \text{ulong } p)$

$\text{GEN FlxqX_red_pre}(\text{GEN } z, \text{GEN } T, \text{ulong } p, \text{ulong } pi)$

$\text{GEN FlxqX_normalize}(\text{GEN } z, \text{GEN } T, \text{ulong } p)$

$\text{GEN FlxqX_normalize_pre}(\text{GEN } z, \text{GEN } T, \text{ulong } p, \text{ulong } pi)$

$\text{GEN FlxqX_mul}(\text{GEN } x, \text{GEN } y, \text{GEN } T, \text{ulong } p)$

$\text{GEN FlxqX_mul_pre}(\text{GEN } x, \text{GEN } y, \text{GEN } T, \text{ulong } p, \text{ulong } pi)$

$\text{GEN FlxqX_Flxq_mul}(\text{GEN } P, \text{GEN } U, \text{GEN } T, \text{ulong } p)$

$\text{GEN FlxqX_Flxq_mul_pre}(\text{GEN } P, \text{GEN } U, \text{GEN } T, \text{ulong } p, \text{ulong } pi)$

$\text{GEN FlxqX_Flxq_mul_to_monic}(\text{GEN } P, \text{GEN } U, \text{GEN } T, \text{ulong } p)$ returns $P * U$ assuming the result is monic of the same degree as P (in particular $U \neq 0$).

$\text{GEN FlxqX_Flxq_mul_to_monic_pre}(\text{GEN } P, \text{GEN } U, \text{GEN } T, \text{ulong } p, \text{ulong } pi)$

$\text{GEN FlxqX_sqr}(\text{GEN } x, \text{GEN } T, \text{ulong } p)$

GEN FlxqX_sqr_pre(GEN x, GEN T, ulong p, ulong pi)
 GEN FlxqX_powu(GEN x, ulong n, GEN T, ulong p)
 GEN FlxqX_powu_pre(GEN x, ulong n, GEN T, ulong p, ulong pi)
 GEN FlxqX_divrem(GEN x, GEN y, GEN T, ulong p, GEN *pr)
 GEN FlxqX_divrem_pre(GEN x, GEN y, GEN T, ulong p, ulong pi, GEN *pr)
 GEN FlxqX_div(GEN x, GEN y, GEN T, ulong p)
 GEN FlxqX_div_pre(GEN x, GEN y, GEN T, ulong p, ulong pi)
 GEN FlxqX_div_by_X_x(GEN a, GEN x, GEN T, ulong p, GEN *r)
 GEN FlxqX_div_by_X_x_pre(GEN a, GEN x, GEN T, ulong p, ulong pi, GEN *r)
 GEN FlxqX_rem(GEN x, GEN y, GEN T, ulong p)
 GEN FlxqX_rem_pre(GEN x, GEN y, GEN T, ulong p, ulong pi)
 GEN FlxqX_invBarrett(GEN T, GEN Q, ulong p)
 GEN FlxqX_invBarrett_pre(GEN T, GEN Q, ulong p, ulong pi)
 GEN FlxqX_gcd(GEN x, GEN y, ulong p) returns a (not necessarily monic) greatest common divisor of x and y .
 GEN FlxqX_gcd_pre(GEN x, GEN y, ulong p, ulong pi)
 GEN FlxqX_extgcd(GEN x, GEN y, GEN T, ulong p, GEN *ptu, GEN *ptv)
 GEN FlxqX_extgcd_pre(GEN x, GEN y, GEN T, ulong p, ulong pi, GEN *ptu, GEN *ptv)
 GEN FlxqX_halfgcd(GEN x, GEN y, GEN T, ulong p), see FpX_halfgcd.
 GEN FlxqX_halfgcd_pre(GEN x, GEN y, GEN T, ulong p, ulong pi)
 GEN FlxqX_halfgcd_all(GEN x, GEN y, GEN T, ulong p, GEN *a, GEN *b), see FpX_halfgcd_all.
 GEN FlxqX_halfgcd_all_pre(GEN x, GEN y, GEN T, ulong p, ulong pi, GEN *a, GEN *b), see FpX_halfgcd_all_pre.
 GEN FlxqX_resultant(GEN x, GEN y, GEN T, ulong p), see FpX_resultant.
 GEN FlxqX_resultant_pre(GEN x, GEN y, GEN T, ulong p, ulong pi)
 GEN FlxqX_saferes resultant(GEN P, GEN Q, GEN T, ulong p) Returns the resultant 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 $\mathbf{F}_p[X]$, the routine may still be used (but will fail if noninvertible leading terms occur).
 GEN FlxqX_composedsum(GEN P, GEN Q, GEN T, ulong p)
 GEN FlxqX_disc(GEN x, GEN T, ulong p)
 GEN FlxqX_eval(GEN x, GEN y, GEN T, ulong p) evaluates the FlxqX x at the Flxq y . The result is an Flxq.
 GEN FlxqXV_prod(GEN V, GEN T, ulong p)

`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 $\mathbf{F}_p[X]$, the routine may still be used (but will fail if noninvertible leading terms occur).

`GEN FlxqX_dotproduct(GEN x, GEN y, GEN T, ulong p)` returns the scalar product of the coefficients of x and y .

`GEN FlxqX_Newton(GEN x, long n, GEN T, ulong p)`

`GEN FlxqX_Newton_pre(GEN x, long n, GEN T, ulong p, ulong pi)`

`GEN FlxqX_fromNewton(GEN x, GEN T, ulong p)`

`GEN FlxqX_fromNewton_pre(GEN x, GEN T, ulong p, ulong pi)` We assume pi is a pseudoinverse of p , or 0 in which case we assume `SMALL_ULONG(p)`.

`long FlxqX_is_squarefree(GEN S, GEN T, ulong p)`, as `FpX_is_squarefree`.

`long FlxqX_ispower(GEN f, ulong k, GEN T, ulong p, GEN *pt)` return 1 if the `FlxqX` f is a k -th power, 0 otherwise. If pt is not `NULL`, set it to g such that $g^k = f$.

`GEN FlxqX_Frobenius(GEN S, GEN T, ulong p)`, as `FpXQX_Frobenius`

`GEN FlxqX_Frobenius_pre(GEN S, GEN T, ulong p, ulong pi)`

`GEN FlxqX_roots(GEN f, GEN T, ulong p)` return the roots of f in $\mathbf{F}_p[X]/(T)$. Assumes p is prime and T irreducible in $\mathbf{F}_p[X]$.

`GEN FlxqX_factor(GEN f, GEN T, ulong p)` return the factorization of f over $\mathbf{F}_p[X]/(T)$. Assumes p is prime and T irreducible in $\mathbf{F}_p[X]$.

`GEN FlxqX_factor_squarefree(GEN f, GEN T, ulong p)` returns the squarefree factorization of f , see `FpX_factor_squarefree`.

`GEN FlxqX_factor_squarefree_pre(GEN f, GEN T, ulong p, ulong pi)`

`GEN FlxqX_ddf(GEN f, GEN T, ulong p)` as `FpX_ddf`.

`long FlxqX_ddf_degree(GEN f, GEN XP, GEN T, GEN p)`, as `FpX_ddf_degree`.

`GEN FlxqX_degfact(GEN f, GEN T, ulong p)`, as `FpX_degfact`.

`long FlxqX_nbroots(GEN S, GEN T, ulong p)`, as `FpX_nbroots`.

`long FlxqX_nbfact(GEN S, GEN T, ulong p)`, as `FpX_nbfact`.

`long FlxqX_nbfact_Frobenius(GEN S, GEN Xq, GEN T, ulong p)`, as `FpX_nbfact_Frobenius`.

`GEN FlxqX_nbfact_by_degree(GEN z, long *nb, GEN T, ulong p)` Assume that the `FlxqX` 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 $D[i]$).

`GEN FlxqX_FlxqXQ_eval(GEN Q, GEN x, GEN S, GEN T, ulong p)` as `FpX_FpXQ_eval`.

`GEN FlxqX_FlxqXQ_eval_pre(GEN Q, GEN x, GEN S, GEN T, ulong p, ulong pi)`

`GEN FlxqX_FlxqXQV_eval(GEN P, GEN V, GEN S, GEN T, ulong p)` as `FpX_FpXQV_eval`.

`GEN FlxqX_FlxqXQV_eval_pre(GEN P, GEN V, GEN S, GEN T, ulong p, ulong pi)`


```

GEN FlxqXC_FlxqXQ_eval(GEN Q, GEN x, GEN S, GEN T, ulong p) as FpXC_FpXQ_eval.
GEN FlxqXC_FlxqXQ_eval_pre(GEN Q, GEN x, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXC_FlxqXQV_eval(GEN P, GEN V, GEN S, GEN T, ulong p) as FpXC_FpXQV_eval.
GEN FlxqXC_FlxqXQV_eval_pre(GEN P, GEN V, GEN S, GEN T, ulong p, ulong pi)

```

7.3.26 FlxqXQ. See FpXQXQ operations. In this section, pi is a pseudoinverse of p , or 0 in which case we assume $\text{SMALL_ULONG}(p)$.

```

GEN FlxqXQ_mul(GEN x, GEN y, GEN S, GEN T, ulong p)
GEN FlxqXQ_mul_pre(GEN x, GEN y, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_sqr(GEN x, GEN S, GEN T, ulong p)
GEN FlxqXQ_sqr_pre(GEN x, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_inv(GEN x, GEN S, GEN T, ulong p)
GEN FlxqXQ_inv_pre(GEN x, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_invsafe(GEN x, GEN S, GEN T, ulong p)
GEN FlxqXQ_invsafe_pre(GEN x, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_div(GEN x, GEN y, GEN S, GEN T, ulong p)
GEN FlxqXQ_div_pre(GEN x, GEN y, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_pow(GEN x, GEN n, GEN S, GEN T, ulong p)
GEN FlxqXQ_pow_pre(GEN x, GEN n, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_powu(GEN x, ulong n, GEN S, GEN T, ulong p)
GEN FlxqXQ_powu_pre(GEN x, ulong n, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_powers(GEN x, long n, GEN S, GEN T, ulong p)
GEN FlxqXQ_powers_pre(GEN x, long n, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_matrix_pow(GEN x, long n, long m, GEN S, GEN T, ulong p)
GEN FlxqXQ_autpow(GEN a, long n, GEN S, GEN T, ulong p) as FpXQXQ_autpow
GEN FlxqXQ_autpow_pre(GEN a, long n, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_autsum(GEN a, long n, GEN S, GEN T, ulong p) as FpXQXQ_autsum
GEN FlxqXQ_autsum_pre(GEN a, long n, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_auttrace(GEN a, long n, GEN S, GEN T, ulong p) as FpXQXQ_auttrace
GEN FlxqXQ_auttrace_pre(GEN a, long n, GEN S, GEN T, ulong p, ulong pi)
GEN FlxqXQ_halfFrobenius(GEN A, GEN S, GEN T, ulong p), as FpXQXQ_halfFrobenius
GEN FlxqXQ_minpoly(GEN x, GEN S, GEN T, ulong p), as FpXQ_minpoly
GEN FlxqXQ_minpoly_pre(GEN x, GEN S, GEN T, ulong p, ulong pi)

```

7.3.27 FlxqXn. See FpXn operations. In this section, we assume pi is the pseudoinverse of p , or 0 in which case we assume `SMALL_ULONG( $p$ )`.

`GEN FlxXn_red(GEN a, long n)` returns a modulo X^n .

`GEN FlxqXn_mul(GEN a, GEN b, long n, GEN T, ulong p)`

`GEN FlxqXn_mul_pre(GEN a, GEN b, long n, GEN T, ulong p, ulong pi)`

`GEN FlxqXn_sqr(GEN a, long n, GEN T, ulong p)`

`GEN FlxqXn_sqr_pre(GEN a, long n, GEN T, ulong p, ulong pi)`

`GEN FlxqXn_inv(GEN a, long n, GEN T, ulong p)`

`GEN FlxqXn_inv_pre(GEN a, long n, GEN T, ulong p, ulong pi)`

`GEN FlxqXn_div(GEN a, GEN b, long n, GEN T, ulong p)`

`GEN FlxqXn_div_pre(GEN a, GEN b, long n, GEN T, ulong p, ulong pi)`

`GEN FlxqXn_expint(GEN a, long n, GEN T, ulong p)`

`GEN FlxqXn_expint_pre(GEN a, long n, GEN T, ulong p, ulong pi)`

7.3.28 F2x. An F2x z is a `t_VECSMALL` representing a polynomial over $\mathbf{F}_2[X]$. 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.

7.3.28.1 Preconditioned reduction.

For faster reduction, the modulus T can be replaced by an extended modulus (`FlxT`) in all `Flxq`-classes functions, and in `Flx_divrem`.

`GEN F2x_get_red(GEN T)` returns the extended modulus `eT`.

To write code that works both with plain and extended moduli, the following accessors are defined:

`GEN get_F2x_mod(GEN eT)` returns the underlying modulus T .

`GEN get_F2x_var(GEN eT)` returns the variable number of the modulus.

`GEN get_F2x_degree(GEN eT)` returns the degree of the modulus.

7.3.28.2 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 F2x_copy(GEN x)`

`GEN Flx_to_F2x(GEN x)`

`GEN Z_to_F2x(GEN x, long sv)`

`GEN ZX_to_F2x(GEN x)`

GEN F2v_to_F2x(GEN x, long sv)
 GEN F2x_to_Flx(GEN x)
 GEN F2x_to_F2xX(GEN x, long sv)
 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 monomial_F2x(long d, long sv) returns the F2x X^d in variable v .
 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 .
 GEN F2x_recip(GEN x)
 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)
 ulong F2x_eval(GEN P, ulong u) returns $P(u)$.
 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_degfact(GEN f) as FpX_degfact.
 GEN F2x_sqrt(GEN x) returns the squareroot of x , assuming x is a square of a F2x.
 GEN F2x_Frobenius(GEN T)
 GEN F2x_matFrobenius(GEN T)
 GEN F2x_factor(GEN f)
 GEN F2x_factor_squarefree(GEN f)
 GEN F2x_ddf(GEN f)
 GEN F2x_Teichmuller(GEN P, long n) Return a ZX Q such that $P \equiv Q \pmod{2}$ and $Q(X^p) = 0 \pmod{Q, 2^n}$.
7.3.29 F2xq. See FpXQ operations.
 GEN F2xq_mul(GEN x, GEN y, GEN T)
 GEN F2xq_sqr(GEN x, GEN T)
 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 T)
 GEN F2xq_powu(GEN x, ulong n, GEN T)
 GEN F2xq_pow_init(GEN x, GEN n, long k, GEN T)
 GEN F2xq_pow_table(GEN R, GEN n, GEN T)
 ulong F2xq_trace(GEN x, GEN T)
 GEN F2xq_conjvec(GEN x, GEN T) returns the vector of conjugates $[x, x^2, x^{2^2}, \dots, x^{2^{n-1}}]$ 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_sqrtn(GEN a, GEN n, GEN T, GEN *zeta)
 GEN gener_F2xq(GEN T, GEN *po)
 GEN F2xq_powers(GEN x, long n, GEN T)
 GEN F2xq_matrix_pow(GEN x, long m, 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_FpXQV_eval.

GEN F2xq_autpow(GEN a, long n, GEN T) computes $\sigma^n(X)$ assuming $a = \sigma(X)$ where σ is an automorphism of the algebra $\mathbf{F}_2[X]/T(X)$.

GEN F2xqV_roots_to_pol(GEN V, GEN T, long v) as FqV_roots_to_pol returning an F2xqX in variable v .

7.3.30 F2xn. See FpXn operations.

GEN F2xn_red(GEN a, long n)

GEN F2xn_div(GEN x, GEN y, long e)

GEN F2xn_inv(GEN x, long e)

7.3.31 F2xqV, F2xqM.. See FqV, FqM operations.

GEN F2xqM_F2xqC_gauss(GEN a, GEN b, GEN T)

GEN F2xqM_F2xqC_invimage(GEN a, GEN b, GEN T)

GEN F2xqM_F2xqC_mul(GEN a, GEN b, GEN T)

GEN F2xqM_deplin(GEN x, GEN T)

GEN F2xqM_det(GEN a, GEN T)

GEN F2xqM_gauss(GEN a, GEN b, GEN T)

GEN F2xqM_image(GEN x, GEN T)

GEN F2xqM_indexrank(GEN x, GEN T)

GEN F2xqM_inv(GEN a, GEN T)

GEN F2xqM_invimage(GEN a, GEN b, GEN T)

GEN F2xqM_ker(GEN x, GEN T)

GEN F2xqM_mul(GEN a, GEN b, GEN T)

long F2xqM_rank(GEN x, GEN T)

GEN F2xqM_suppl(GEN x, GEN T)

GEN matid_F2xqM(long n, GEN T)

7.3.32 F2xX.. See FpXX operations.

GEN ZXX_to_F2xX(GEN x, long v)

GEN FlxX_to_F2xX(GEN x)

GEN F2xX_to_FlxX(GEN B)

GEN F2xX_to_F2xC(GEN B, long N, long sv)

GEN F2xXV_to_F2xM(GEN B, long N, long sv)

GEN F2xX_to_ZXX(GEN B)

GEN F2xX_renormalize(GEN x, long lx)

GEN F2xX_shift(GEN a, long n, long sv), as RgX_shift_shallow, where v is the secondary variable.

long F2xY_degreeex(GEN P) return the degree of P with respect to the secondary variable.

GEN pol1_F2xX(long v, long sv)

GEN polx_F2xX(long v, long sv)

GEN F2xX_add(GEN x, GEN y)

GEN F2xX_F2x_add(GEN x, GEN y)

GEN F2xX_F2x_mul(GEN x, GEN y)

GEN F2xX_deriv(GEN P) returns the derivative of P with respect to the main variable.

GEN Kronecker_to_F2xqX(GEN z, GEN T)

GEN F2xX_to_Kronecker(GEN z, GEN T)

GEN F2xY_F2xq_evalx(GEN x, GEN y, GEN T) as FpXY_FpXQ_evalx.

GEN F2xY_F2xqV_evalx(GEN x, GEN V, GEN T) as FpXY_FpXQV_evalx.

7.3.33 F2xXV/F2xXC.. See FpXXV operations.

GEN FlxXC_to_F2xXC(GEN B)

GEN F2xXC_to_ZXXC(GEN B)

7.3.34 F2xqX.. See FlxqX operations.

7.3.34.1 Preconditioned reduction.

For faster reduction, the modulus S can be replaced by an extended modulus, which is an $F2xqXT$, in all $F2xqXQ$ -classes functions, and in $F2xqX_{\text{rem}}$ and $F2xqX_{\text{divrem}}$.

GEN F2xqX_get_red(GEN S, GEN T) returns the extended modulus eS .

To write code that works both with plain and extended moduli, the following accessors are defined:

GEN get_F2xqX_mod(GEN eS) returns the underlying modulus S .

GEN get_F2xqX_var(GEN eS) returns the variable number of the modulus.

GEN get_F2xqX_degree(GEN eS) returns the degree of the modulus.

7.3.34.2 basic functions.

GEN random_F2xqX(long d, long v, GEN T, ulong p) returns a random F2xqX in variable v, of degree less than d.

GEN F2xqX_red(GEN z, GEN T)

GEN F2xqX_normalize(GEN z, GEN T)

GEN F2xqX_F2xq_mul(GEN P, GEN U, GEN T)

GEN F2xqX_F2xq_mul_to_monic(GEN P, GEN U, GEN T)

GEN F2xqX_mul(GEN x, GEN y, GEN T)

GEN F2xqX_sqr(GEN x, GEN T)

GEN F2xqX_powu(GEN x, ulong n, GEN T)

GEN F2xqX_rem(GEN x, GEN y, GEN T)

GEN F2xqX_div(GEN x, GEN y, GEN T)

GEN F2xqX_divrem(GEN x, GEN y, GEN T, GEN *pr)

GEN F2xqXQ_inv(GEN x, GEN S, GEN T)

GEN F2xqXQ_invsafe(GEN x, GEN S, GEN T)

GEN F2xqX_invBarrett(GEN T, GEN Q)

GEN F2xqX_extgcd(GEN x, GEN y, GEN T, GEN *ptu, GEN *ptv)

GEN F2xqX_gcd(GEN x, GEN y, GEN T)

GEN F2xqX_halfgcd(GEN x, GEN y, GEN T)

GEN F2xqX_halfgcd_all(GEN x, GEN y, GEN T, GEN *a, GEN *b)

GEN F2xqX_resultant(GEN x, GEN y, GEN T)

GEN F2xqX_disc(GEN x, GEN T)

GEN F2xqXV_prod(GEN V, GEN T)

long F2xqX_ispower(GEN f, ulong k, GEN T, GEN *pt)

GEN F2xqX_F2xqXQ_eval(GEN Q, GEN x, GEN S, GEN T) as FpX_FpXQ_eval.

GEN F2xqX_F2xqXQV_eval(GEN P, GEN V, GEN S, GEN T) as FpX_FpXQV_eval.

GEN F2xqX_Frobenius(GEN S, GEN T), as FpXQX_Frobenius

GEN F2xqX_roots(GEN f, GEN T) return the roots of f in $\mathbf{F}_2[X]/(T)$. Assumes T irreducible in $\mathbf{F}_2[X]$.

GEN F2xqX_factor(GEN f, GEN T) return the factorization of f over $\mathbf{F}_2[X]/(T)$. Assumes T irreducible in $\mathbf{F}_2[X]$.

GEN F2xqX_factor_squarefree(GEN f, GEN T) as FlxqX_factor_squarefree.

GEN F2xqX_ddf(GEN f, GEN T) as FpX_ddf.

GEN F2xqX_degfact(GEN f, GEN T) as FpX_degfact.

long F2xqX_nbroots(GEN f, GEN T) as FpX_nbroots.

7.3.35 F2xqXQ.. See FlxqXQ operations.

GEN FlxqXQ_inv(GEN x, GEN S, GEN T)

GEN FlxqXQ_invsafe(GEN x, GEN S, GEN T)

GEN F2xqXQ_mul(GEN x, GEN y, GEN S, GEN T)

GEN F2xqXQ_sqr(GEN x, GEN S, GEN T)

GEN F2xqXQ_pow(GEN x, GEN n, GEN S, GEN T)

GEN F2xqXQ_powers(GEN x, long n, GEN S, GEN T)

GEN F2xqXQ_autpow(GEN a, long n, GEN S, GEN T) as FpXQXQ_autpow

GEN F2xqXQ_auttrace(GEN a, long n, GEN S, GEN T). Let σ be the automorphism defined by $\sigma(X) = a[1] \pmod{T(X)}$ and $\sigma(Y) = a[2] \pmod{S(X,Y), T(X)}$; returns the vector $[\sigma^n(X), \sigma^n(Y), b + \sigma(b) + \dots + \sigma^{n-1}(b)]$ where $b = a[3]$.

GEN F2xqXQV_red(GEN x, GEN S, GEN T)

7.3.36 Functions returning objects with t_INTMOD coefficients.

Those functions are mostly needed for interface reasons: 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:

GEN Fp_to_mod(GEN z, GEN p), z a t_INT. Returns $z * \text{Mod}(1,p)$, normalized. Hence the returned value is a t_INTMOD.

GEN FpX_to_mod(GEN z, GEN p), z a ZX. Returns $z * \text{Mod}(1,p)$, normalized. Hence the returned value has t_INTMOD coefficients.

GEN FpC_to_mod(GEN z, GEN p), z a ZC. Returns $\text{Col}(z) * \text{Mod}(1,p)$, a t_COL with t_INTMOD coefficients.

GEN FpV_to_mod(GEN z, GEN p), z a ZV. Returns $\text{Vec}(z) * \text{Mod}(1,p)$, a t_VEC with t_INTMOD coefficients.

GEN FpVV_to_mod(GEN z, GEN p), z a ZVV. Returns $\text{Vec}(z) * \text{Mod}(1,p)$, a t_VEC of t_VEC with t_INTMOD coefficients.

GEN FpM_to_mod(GEN z, GEN p), z a ZM. Returns $z * \text{Mod}(1,p)$, with t_INTMOD coefficients.

GEN F2c_to_mod(GEN x)

GEN F3c_to_mod(GEN x)

GEN F2m_to_mod(GEN x)

GEN F3m_to_mod(GEN x)

GEN Flc_to_mod(GEN z)

GEN Flm_to_mod(GEN z)

GEN FqC_to_mod(GEN z, GEN T, GEN p)

GEN FqM_to_mod(GEN z, GEN T, GEN p)

GEN FpXC_to_mod(GEN V, GEN p)

GEN FpXM_to_mod(GEN V, GEN p)

GEN FpXQC_to_mod(GEN V, GEN T, GEN p) V being a vector of FpXQ, converts each entry to a t_POLMOD with t_INTMOD coefficients, and return a t_COL .

GEN FpXQX_to_mod(GEN P, GEN T, GEN p) P being a FpXQX, converts each coefficient to a t_POLMOD with t_INTMOD coefficients.

GEN FqX_to_mod(GEN P, GEN T, GEN p) same but allow $T = \text{NULL}$.

GEN FqXC_to_mod(GEN P, GEN T, GEN p)

GEN FqXM_to_mod(GEN P, GEN T, GEN p)

GEN QXQ_to_mod_shallow(GEN x, GEN T) x a QXQ, which is a lifted representative of elements of $\mathbf{Q}[X]/(T)$ (number field elements in most applications) and T is in $\mathbf{Z}[X]$. Convert it to a t_POLMOD modulo T ; no reduction mod T is attempted: the representatives should be already reduced. Shallow function.

GEN QXQV_to_mod(GEN V, GEN T) V a vector of QXQ, which are lifted representatives of elements of $\mathbf{Q}[X]/(T)$ (number field elements in most applications) and T is in $\mathbf{Z}[X]$. Return a vector where all nonrational entries are converted to t_POLMOD modulo T ; no reduction mod T is attempted: the representatives should be already reduced. Used to normalize the output of `nfroots`.

GEN QXQX_to_mod_shallow(GEN P, GEN T) P a polynomial with QXQ coefficients; replace them by `mkpolmod(.,T)`. Shallow function.

GEN QXQC_to_mod_shallow(GEN V, GEN T) V a vector with QXQ coefficients; replace them by `mkpolmod(.,T)`. Shallow function.

GEN QXQM_to_mod_shallow(GEN M, GEN T) M a matrix with QXQ coefficients; replace them by `mkpolmod(.,T)`. Shallow function.

GEN QXQXV_to_mod(GEN V, GEN T) V a vector of polynomials whose coefficients are QXQ. Analogous to `QXQV_to_mod`. Used to normalize the output of `nfactor`.

The following functions are obsolete and should not be used: they receive a polynomial with arbitrary coefficients, apply a conversion function to map them to a finite field, a function from the modular kernel, then `*_to_mod`:

GEN rootmod(GEN f, GEN p), applies `FpX_roots`.

GEN rootmod2(GEN f, GEN p), (now) identical to `rootmod`.

GEN rootmod0(GEN f, GEN p, long flag), (now) identical to `rootmod`; ignores *flag*.

GEN factmod(GEN f, GEN p) applies `*_factor`.

GEN simplefactmod(GEN f, GEN p) applies `*_degfact`.

7.3.37 Slow Chinese remainder theorem over $\mathbf{Z}$. The routines in this section have quadratic time complexity with respect to the input size; see the routines in the next two sections for quasi-linear time variants.

`GEN Z_chinese(GEN a, GEN b, GEN A, GEN B)` returns the integer in $[0, \text{lcm}(A, B)[$ congruent to $a \bmod A$ and $b \bmod B$, assuming it exists; in other words, that a and b are congruent mod $\text{gcd}(A, B)$.

`GEN Z_chinese_all(GEN a, GEN b, GEN A, GEN B, GEN *pC)` as `Z_chinese`, setting `*pC` to the lcm of A and B .

`GEN Z_chinese_coprime(GEN a, GEN b, GEN A, GEN B, GEN C)`, as `Z_chinese`, assuming that $\text{gcd}(A, B) = 1$ and that $C = \text{lcm}(A, B) = AB$.

`ulong u_chinese_coprime(ulong a, ulong b, ulong A, ulong B, ulong C)`, as `Z_chinese_coprime` for `ulong` inputs and output.

`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 $\text{gcd}(A, B)$, `*pU` to an integer congruent to 0 mod (A/d) and 1 mod (B/d) . It is allowed to set `pd = 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 $a \bmod A$ and $b \bmod B$, where C, U, d were set in `Z_chinese_pre`. If d is `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 $H_p \bmod p$ 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 $H = H$, 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 H_p 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 (H_p, p) with p coprime to q , this function updates `*H` so that it also becomes congruent to (H_p, 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 $\mathbf{Z}$, moduli not assumed coprime).

As above, for H a `ZM`: we assume that H and all H_p 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)

As above, for H a matrix whose coefficient are ZX. The original $*H$ is destroyed. The entries of H are not normalized, use ZX_renormalize for this.

GEN ZXM_init_CRT(GEN Hp, long deg, ulong p) where deg is the maximal degree of all the Hp

int ZXM_incremental_CRT(GEN *H, GEN Hp, GEN *q, ulong p)

7.3.38 Fast remainders.

The routines in these section are asymptotically fast (quasi-linear time in the input size).

GEN Z_ZV_mod(GEN A, GEN P) given a $\mathbf{t_INT}$ A and a vector P of positive pairwise coprime integers of length $n \geq 1$, return a vector B of the same length such that $B[i] = A \pmod{P[i]}$ and $0 \leq B[i] < P[i]$ for all $1 \leq i \leq n$. The vector P may be a $\mathbf{t_VEC}$ or a $\mathbf{t_VECSMALL}$ (treated as $\mathbf{ulongs}$) and B has the same type as P .

GEN Z_nv_mod(GEN A, GEN P) given a $\mathbf{t_INT}$ A and a $\mathbf{t_VECSMALL}$ P of positive pairwise coprime integers of length $n \geq 1$, return a $\mathbf{t_VECSMALL}$ B of the same length such that $B[i] = A \pmod{P[i]}$ and $0 \leq B[i] < P[i]$ for all $1 \leq i \leq n$. The entries of P and B are treated as $\mathbf{ulongs}$.

The following low level functions allow precomputations:

GEN ZV_producttree(GEN P) where P is a vector of integers (or $\mathbf{t_VECSMALL}$) of length $n \geq 1$, return the vector of $\mathbf{t_VECS}$ $[f(P), f^2(P), \dots, f^k(P)]$ where f is the transformation $[p_1, p_2, \dots, p_m] \mapsto [p_1 p_2, p_3 p_4, \dots, p_{m-1} p_m]$ if m is even and $[p_1 p_2, p_3 p_4, \dots, p_{m-2} p_{m-1}, p_m]$ if m is odd, and $k = O(\log m)$ is minimal so that $f^k(P)$ has length 1; in other words, $f^k(P) = [p_1 p_2 \dots p_m]$.

GEN ZMV_producttree(GEN P) where P is a vector of ZM of length $n \geq 1$, return the vector of $\mathbf{t_VECS}$ $[f(P), f^2(P), \dots, f^k(P)]$ where f is the transformation $[p_1, p_2, \dots, p_m] \mapsto [p_2 p_1, p_4 p_3, \dots, p_m p_{m-1}]$ if m is even and $[p_2 p_1, p_4 p_3, \dots, p_{m-1} p_{m-2}, p_m]$ if m is odd, and $k = O(\log m)$ is minimal so that $f^k(P)$ has length 1; in other words, $f^k(P) = [p_m \dots p_2 p_1]$.

GEN Z_ZV_mod_tree(GEN A, GEN P, GEN T) as Z_ZV_mod where T is the tree ZV_producttree(P).

GEN ZV_nv_mod_tree(GEN A, GEN P, GEN T) A being a ZV and P a $\mathbf{t_VECSMALL}$ of length $n \geq 1$, the elements of P being pairwise coprime, return the vector of $\mathbf{Flv}$ $[A \pmod{P[1]}, \dots, A \pmod{P[n]}]$, where T is the tree ZV_producttree(P).

GEN ZM_nv_mod_tree(GEN A, GEN P, GEN T) A being a ZM and P a $\mathbf{t_VECSMALL}$ of length $n \geq 1$, the elements of P being pairwise coprime, return the vector of $\mathbf{Flm}$ $[A \pmod{P[1]}, \dots, A \pmod{P[n]}]$, where T is the tree ZV_producttree(P).

GEN ZVV_nv_mod_tree(GEN A, GEN P, GEN T) A being a $\mathbf{t_VEC}$ of $\mathbf{t_ZV}$ (all of length $n \geq 1$) and P a $\mathbf{t_VECSMALL}$ of length n , the elements of P being pairwise coprime, return the vector of $\mathbf{FlvV}$ $[A \pmod{P[1]}, \dots, A \pmod{P[n]}]$, where T is the tree ZV_producttree(P).

GEN ZX_nv_mod_tree(GEN A, GEN P, GEN T) A being a ZX and P a `t_VECSMALL` of length $n \geq 1$, the elements of P being pairwise coprime, return the vector of `Flx` polynomials $[A \pmod{P[1]}, \dots, A \pmod{P[n]}]$, where T is the tree `ZV_producttree(P)`.

GEN ZXC_nv_mod_tree(GEN A, GEN P, GEN T) A being a ZXC and P a `t_VECSMALL` of length $n \geq 1$, the elements of P being pairwise coprime, return the vector of `FlxC` $[A \pmod{P[1]}, \dots, A \pmod{P[n]}]$, where T is the tree `ZV_producttree(P)`.

GEN ZXM_nv_mod_tree(GEN A, GEN P, GEN T) A being a ZXM and P a `t_VECSMALL` of length $n \geq 1$, the elements of P being pairwise coprime, return the vector of `FlxM` $[A \pmod{P[1]}, \dots, A \pmod{P[n]}]$, where T is the tree `ZV_producttree(P)`.

GEN ZXX_nv_mod_tree(GEN A, GEN P, GEN T, long v) A being a ZXX, and P a `t_VECSMALL` of length $n \geq 1$, the elements of P being pairwise coprime, return the vector of `FlxX` $[A \pmod{P[1]}, \dots, A \pmod{P[n]}]$, where T is assumed to be the tree created by `ZV_producttree(P)`.

7.3.39 Fast Chinese remainder theorem over $\mathbf{Z}$. The routines in these section are asymptotically fast (quasi-linear time in the input size) and should be used whenever the moduli are known from the start.

The simplest function is

GEN ZV_chinese(GEN A, GEN P, GEN *pM) let P be a vector of positive pairwise coprime integers, let A be a vector of integers of the same length $n \geq 1$ such that $0 \leq A[i] < P[i]$ for all i , and let M be the product of the elements of P . Returns the integer in $[0, M[$ congruent to $A[i] \pmod{P[i]}$ for all $1 \leq i \leq n$. If `pM` is not NULL, set `*pM` to M . We also allow `t_VECSMALL`s for A and P (seen as vectors of unsigned integers).

GEN ZV_chinese_center(GEN A, GEN P, GEN *pM) As `ZV_chinese` but return integers in $[-M/2, M/2[$ instead.

The following functions allow to solve many Chinese remainder problems simultaneously, for a given set of moduli:

GEN nxV_chinese_center(GEN A, GEN P, GEN *pt_mod) where A is a vector of `nx` and P a `t_VECSMALL` of the same length $n \geq 1$, the elements of P being pairwise coprime, and M being the product of the elements of P , returns the `t_POL` whose entries are integers in $[-M/2, M/2[$ congruent to $A[i] \pmod{P[i]}$ for all $1 \leq i \leq n$. If `pt_mod` is not NULL, set `*pt_mod` to M .

GEN ncV_chinese_center(GEN A, GEN P, GEN *pM) where A is a vector of `VECSMALL`s (seen as vectors of unsigned integers) and P a `t_VECSMALL` of the same length $n \geq 1$, the elements of P being pairwise coprime, and M being the product of the elements of P , returns the `t_COL` whose entries are integers in $[-M/2, M/2[$ congruent to $A[i] \pmod{P[i]}$ for all $1 \leq i \leq n$. If `pM` is not NULL, set `*pM` to M .

GEN nmV_chinese_center(GEN A, GEN P, GEN *pM) where A is a vector of `MATSMALL`s (seen as matrices of unsigned integers) and P a `t_VECSMALL` of the same length $n \geq 1$, the elements of P being pairwise coprime, and M being the product of the elements of P , returns the matrix whose entries are integers in $[-M/2, M/2[$ congruent to $A[i] \pmod{P[i]}$ for all $1 \leq i \leq n$. If `pM` is not NULL, set `*pM` to M . N.B.: this function uses the parallel GP interface.

GEN nxCV_chinese_center(GEN A, GEN P, GEN *pM) where A is a vector of `nxCs` and P a `t_VECSMALL` of the same length $n \geq 1$, the elements of P being pairwise coprime, and M being

the product of the elements of P , returns the `t_COL` whose entries are integers in $[-M/2, M/2[$ congruent to $A[i] \bmod P[i]$ for all $1 \leq i \leq n$. If `pM` is not NULL, set `*pt_mod` to M .

`GEN nxMV_chinese_center(GEN A, GEN P, GEN *pM)` where A is a vector of `nxMs` and P a `t_VECSMALL` of the same length $n \geq 1$, the elements of P being pairwise coprime, and M being the product of the elements of P , returns the matrix whose entries are integers in $[-M/2, M/2[$ congruent to $A[i] \bmod P[i]$ for all $1 \leq i \leq n$. If `pM` is not NULL, set `*pM` to M . N.B.: this function uses the parallel GP interface.

The other routines allow for various precomputations :

`GEN ZV_chinesetree(GEN P, GEN T)` given P a vector of integers (or `t_VECSMALL`) and a product tree T from `ZV_producttree(P)` for the same P , return a “chinese remainder tree” R , preconditionning the solution of Chinese remainder problems modulo the $P[i]$.

`GEN ZV_chinese_tree(GEN A, GEN P, GEN T, GEN R)` return `ZV_chinese(A, P, NULL)`, where T is created by `ZV_producttree(P)` and R by `ZV_chinesetree(P, T)`.

`GEN ncV_chinese_center_tree(GEN A, GEN P, GEN T, GEN R)` as `ncV_chinese_center` where T is assumed to be the tree created by `ZV_producttree(P)` and R by `ZV_chinesetree(P, T)`.

`GEN nmV_chinese_center_tree(GEN A, GEN P, GEN T, GEN R)` as `nmV_chinese_center` where T is assumed to be the tree created by `ZV_producttree(P)` and R by `ZV_chinesetree(P, T)`.

`GEN nxV_chinese_center_tree(GEN A, GEN P, GEN T, GEN R)` as `nxV_chinese_center` where T is assumed to be the tree created by `ZV_producttree(P)` and R by `ZV_chinesetree(P, T)`.

`GEN nxCV_chinese_center_tree(GEN A, GEN P, GEN T, GEN R)` as `nxCV_chinese_center` where T is assumed to be the tree created by `ZV_producttree(P)` and R by `ZV_chinesetree(P, T)`.

7.3.40 Rational reconstruction.

`int Fp_ratlift(GEN x, GEN m, GEN amax, GEN bmax, GEN *a, GEN *b)`. Assuming that $0 \leq x < m$, $amax \geq 0$, 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

- $a \equiv bx \bmod m$,
- $|a| \leq amax$, $0 < b \leq bmax$,
- $\gcd(m, b) = \gcd(a, b)$.

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 nonzero multiple B of b , and $m = p^\ell$ 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`.

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.

7.3.41 Zp.

GEN Zp_invlift(GEN b, GEN a, GEN p, long e) let p be a prime t_INT, a be a t_INT and b a t_INT such that $ab \equiv 1 \pmod{p}$. Returns an t_INT A such that $A \equiv a^{-1} \pmod{p}$ and $Ab \equiv 1 \pmod{p^e}$.

GEN Zp_inv(GEN b, GEN p, long e) let p be a prime t_INT and b be a t_INT Returns an t_INT A such that $Ab \equiv 1 \pmod{p^e}$.

GEN Zp_div(GEN a, GEN b, GEN p, long e) let p be a prime t_INT and a and b be a t_INT Returns an t_INT c such that $cb \equiv a \pmod{p^e}$.

GEN Zp_sqrt(GEN b, GEN p, long e) b and p being t_INTs, with p a prime (possibly 2), returns a t_INT a such that $a^2 \equiv b \pmod{p^e}$.

GEN Z2_sqrt(GEN b, long e) b being a t_INTs returns a t_INT a such that $a^2 \equiv b \pmod{2^e}$.

GEN Zp_sqrtlift(GEN b, GEN a, GEN p, long e) let a, b, p be t_INTs, with $p > 2$, such that $a^2 \equiv b \pmod{p}$. Returns a t_INT A such that $A^2 \equiv b \pmod{p^e}$. 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 $a^n \equiv b \pmod{p}$. Returns a t_INT A such that $A^n \equiv b \pmod{p^e}$. Special case of ZpX_liftroot.

GEN Zp_teichmuller(GEN x, GEN p, long e, GEN pe) for p an odd prime, x a t_INT coprime to p , and $pe = p^e$, returns the $(p-1)$ -th root of 1 congruent to x modulo p , modulo p^e . For convenience, $p = 2$ is also allowed and we return 1 (x is 1 mod 4) or $2^e - 1$ (x is 3 mod 4).

GEN teichmullerinit(long p, long n) returns the values of Zp_teichmuller at all $x = 1, \dots, p-1$.

GEN Zp_exp(GEN z, GEN p, ulong e) given a t_INT z (preferably reduced mod p^e), return $\exp_p(a) \pmod{p^e}$ (t_INT).

GEN Zp_log(GEN z, GEN p, ulong e) given a t_INT z (preferably reduced mod p^e), such that $a \equiv 1 \pmod{p}$, return $\log_p(a) \pmod{p^e}$ (t_INT).

7.3.42 ZpM.

GEN ZpM_invlift(GEN M, GEN Np, GEN p, long e) let p be a prime t_INT, Np be a FpM (modulo p) and M a ZpM such that $MNp \equiv 1 \pmod{p}$. Returns an ZpM N such that $N \equiv Np^{-1} \pmod{p}$ and $MN \equiv 1 \pmod{p^e}$.

GEN ZpM_ZpC_gauss(GEN M, GEN V, GEN q, GEN p, long e) M be a ZpM (modulo $q = p^e$) such M is invertible modulo p , solve $M * X = V$ modulo p^e .

7.3.43 ZpX.

GEN ZpX_roots(GEN *f*, GEN *p*, long *e*) *f* a 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 .

GEN ZpX_liftroot(GEN *f*, GEN *a*, GEN *p*, long *e*) *f* a ZX with leading term prime to *p*, and *a* a root mod *p* such that $v_p(f'(a)) = 0$. Return a **t_INT** which is the root of *f* mod p^e congruent to *a* mod *p*.

GEN ZX_Zp_root(GEN *f*, GEN *a*, GEN *p*, long *e*) same as **ZpX_liftroot** without the assumption $v_p(f'(a)) = 0$. Return a **t_VEC** of **t_INTs**, which are the *p*-adic roots of *f* congruent to *a* mod *p* (given modulo p^e). Assume that $0 \leq a < p$.

GEN ZpX_liftroots(GEN *f*, GEN *S*, GEN *p*, long *e*) *f* a 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 ZpX_liftfact(GEN *A*, GEN *B*, GEN *pe*, GEN *p*, long *e*) is the routine underlying **pol-hensellift**. Here, *p* is prime defines a finite field $\mathbf{F}_p$. *A* is a polynomial in $\mathbf{Z}[X]$, whose leading coefficient is nonzero in $\mathbf{F}_q$. *B* is a vector of monic **FpX**, pairwise coprime in $\mathbf{F}_p[X]$, whose product is congruent to $A/\text{lc}(A)$ in $\mathbf{F}_p[X]$. Lifts the elements of *B* mod $\text{pe} = p^e$.

GEN ZpX_Frobenius(GEN *T*, GEN *p*, ulong *e*) returns the *p*-adic lift of the Frobenius automorphism of $\mathbf{F}_p[X]/(T)$ to precision *e*.

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* ZX 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 ZX, 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 ZX, *g* a ZX, $\text{pm} = p^m$ a prime power. There is a unique integer $r \geq 0$ and a monic $h \in \mathbf{Q}_p[X]$ such that

$$p^r h \mathbf{Z}_p[X] + p^m \mathbf{Z}_p[X] = f \mathbf{Z}_p[X] + g \mathbf{Z}_p[X] + p^m \mathbf{Z}_p[X].$$

Return the 0 polynomial if $r \geq m$ and a monic $h \in \mathbf{Z}[1/p][X]$ otherwise (whose valuation at *p* is $> -m$).

GEN ZpX_reduced_resultant(GEN *f*, GEN *g*, GEN *p*, GEN *pm*) *f* a monic ZX, *g* a ZX, $\text{pm} = p^m$ a prime power. The *p*-adic *reduced resultant* of *f* and *g* is 0 if *f, g* not coprime in $\mathbf{Z}_p[X]$, and otherwise the generator of the form p^d of

$$(f \mathbf{Z}_p[X] + g \mathbf{Z}_p[X]) \cap \mathbf{Z}_p.$$

Return the reduced resultant modulo p^m .

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 *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 nonzero result. It is very inefficient when the resultant is 0, but otherwise usually more efficient than computations using a priori bounds.

GEN ZpX_monict_factor(GEN f, GEN p, long M) f a monic ZX, p a prime, return the p -adic factorization of f , modulo p^M . This is the underlying low-level recursive function behind **factorpadic** (using a combination of Round 4 factorization and Hensel lifting); the factors are not sorted and the function is shallow.

GEN ZpX_primedec(GEN T, GEN p) T a monic separable ZX, p a prime, return as a factorization matrix the shape of the prime ideal decomposition of (p) in $\mathbf{Q}[X]/(T)$: the first column contains inertia degrees, the second columns contains ramification degrees.

7.3.44 ZpXQ.

GEN ZpXQ_invlift(GEN b, GEN a, GEN T, GEN p, long e) let p be a prime $\mathbf{t_INT}$, a be a FpXQ (modulo (p, T)) and b a ZpXQ such that $ab \equiv 1 \pmod{(p, T)}$. Returns an ZpXQ A such that $A \equiv a \pmod{p}$ and $Ab \equiv 1 \pmod{(p^e, T)}$.

GEN ZpXQ_inv(GEN b, GEN T, GEN p, long e) let p be a prime $\mathbf{t_INT}$ and b be a FpXQ (modulo T, p^e). Returns an FpXQ A such that $Ab \equiv 1 \pmod{(p^e, T)}$.

GEN ZpXQ_div(GEN a, GEN b, GEN T, GEN q, GEN p, long e) let p be a prime $\mathbf{t_INT}$ and a and b be a FpXQ (modulo T, p^e). Returns an FpXQ c such that $cb \equiv a \pmod{(p^e, T)}$. The parameter q must be equal to p^e .

GEN ZpXQ_sqrtnlift(GEN b, GEN n, GEN a, GEN T, GEN p, long e) let n, p be $\mathbf{t_INT}$ s, with $n, p > 1$ and p coprime to n , and a, b be FpXQs (modulo T) such that $a^n \equiv b \pmod{(p, T)}$. Returns an Fq A such that $A^n \equiv b \pmod{(p^e, T)}$.

GEN ZpXQ_sqrt(GEN b, GEN T, GEN p, long e) let p being a odd prime and b be a FpXQ (modulo T, p^e), returns a such that $a^2 \equiv b \pmod{(p^e, T)}$.

GEN ZpX_ZpXQ_liftroot(GEN f, GEN a, GEN T, GEN p, long e) as **ZpXQX_liftroot**, but f is a polynomial in $\mathbf{Z}[X]$.

GEN ZpX_ZpXQ_liftroot_ea(GEN f, GEN a, GEN T, GEN p, long e, void *E, GEN early(void *E, GEN x, GEN q)) as **ZpX_ZpXQ_liftroot** with early abort: the function **early**(E,x,q) will be called with x is a root of f modulo $q = p^n$ for some n . If **early** returns a non-NULL value z , the function returns z immediately.

GEN ZpXQ_log(GEN a, GEN T, GEN p, long e) T being a ZpX irreducible modulo p , return the logarithm of a in $\mathbf{Z}_p[X]/(T)$ to precision e , assuming that $a \equiv 1 \pmod{p\mathbf{Z}_p[X]}$ if p odd or $a \equiv 1 \pmod{4\mathbf{Z}_2[X]}$ if $p = 2$.

7.3.45 Zq.

GEN Zq_sqrtnlift(GEN b, GEN n, GEN a, GEN T, GEN p, long e)

7.3.46 ZpXQM.

GEN ZpXQM_prodFrobenius(GEN M, GEN T, GEN p, long e) returns the product of matrices $M\sigma(M)\sigma^2(M)\dots\sigma^{n-1}(M)$ to precision e where σ is the lift of the Frobenius automorphism over $\mathbf{Z}_p[X]/(T)$ and n is the degree of T .

7.3.47 ZpXQX.

GEN ZpXQX_liftfact(GEN A, GEN B, GEN T, GEN pe, GEN p, long e) is the routine underlying polhensellift. Here, p is prime, $T(Y)$ defines a finite field $\mathbf{F}_q$. A is a polynomial in $\mathbf{Z}[X, Y]$, whose leading coefficient is nonzero in $\mathbf{F}_q$. B is a vector of monic or $\mathbf{F}_q[X]$, pairwise coprime in $\mathbf{F}_q[X]$, whose product is congruent to $A/\text{lc}(A)$ in $\mathbf{F}_q[X]$. Lifts the elements of $B \bmod pe = p^e$, such that the congruence now holds mod (T, p^e) .

GEN ZpXQX_liftroot(GEN f, GEN a, GEN T, GEN p, long e) as ZpX_liftroot, but f is now a polynomial in $\mathbf{Z}[X, Y]$ and lift the root a in the unramified extension of $\mathbf{Q}_p$ with residue field $\mathbf{F}_p[Y]/(T)$, assuming $v_p(f(a)) > 0$ and $v_p(f'(a)) = 0$.

GEN ZpXQX_liftroot_vald(GEN f, GEN a, long v, GEN T, GEN p, long e) returns the foots of f as 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 ZpXQX_roots(GEN F, GEN T, GEN p, long e)

GEN ZpXQX_liftroots(GEN F, GEN S, GEN T, GEN p, long e)

GEN ZpXQX_divrem(GEN x, GEN Sp, GEN T, GEN q, GEN p, long e, GEN *pr) as FpXQX_divrem. The parameter q must be equal to p^e .

GEN ZpXQX_digits(GEN x, GEN B, GEN T, GEN q, GEN p, long e) As FpXQX_digits. The parameter q must be equal to p^e .

GEN ZpXQX_ZpXQXQ_liftroot(GEN f, GEN a, GEN S, GEN T, GEN p, long e) as ZpXQX_liftroot, except that a is an element of $\mathbf{Z}_p[X, Y]/(S(X, Y), T(X))$.

7.3.48 ZqX. ZqX are either ZpX or ZpXQX depending whether T is NULL or not.

GEN ZqX_roots(GEN F, GEN T, GEN p, long e)

GEN ZqX_liftfact(GEN A, GEN B, GEN T, GEN pe, GEN p, long e)

GEN ZqX_liftroot(GEN f, GEN a, GEN T, GEN p, long e)

GEN ZqX_ZqXQX_liftroot(GEN f, GEN a, GEN P, GEN T, GEN p, long e)

7.3.49 ZpXQXQ.

GEN ZpXQXQ_invlift(GEN b, GEN a, GEN S, GEN T, GEN p, long e) let p be a prime t_INT, a be a FpXQXQ (modulo (S, T, p)) and b a ZpXQXQ such that $ab \equiv 1 \bmod (S, T, p)$. Returns an ZpXQXQ A such that $A \equiv a \pmod{p}$ and $Ab \equiv 1 \bmod (p^e, T)$.

GEN ZpXQXQ_inv(GEN b, GEN S, GEN T, GEN p, long e) let p be a prime t_INT and b be a ZpXQXQ (modulo S, T, p^e). Returns a ZpXQXQ A such that $Ab \equiv 1 \bmod (S, T, p^e)$.

GEN ZpXQXQ_div(GEN a, GEN b, GEN S, GEN T, GEN q, GEN p, long e) let p be a prime t_INT and a and b be a ZpXQXQ (modulo S, T, p^e). Returns a ZpXQXQ c such that $cb \equiv a \bmod (S, T, p^e)$. The parameter q must be equal to p^e .

7.3.50 Other p -adic functions.

`GEN ZpM_echelon(GEN M, long early_abort, GEN p, GEN pm)` given a ZM 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 $E = MU$ modulo p^m . If `early_abort` is nonzero, 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 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.

`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 nonzero `t_PADIC`, this returns `valp(x) + 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 `precp(x)` if $x \neq 0$, and 0 for $x = 0$.

7.3.50.1 low-level.

The following technical function returns an optimal sequence of p -adic accuracies, for a given target accuracy:

`ulong quadratic_prec_mask(long 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 $a_0 = 1, a_1 = 2, \dots, a_k = n$, be the desired sequence of accuracies. To obtain it, we work backwards and set

$$a_k = n, \quad a_{i-1} = (a_i + 1) \setminus 2.$$

This is in essence what the function returns. But we do not want to store the a_i explicitly, even as a `t_VECSMALL`, since this would leave an object on the stack. Instead, we store a_i implicitly in a bitmask `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 `MASK + 2k+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:

```
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

```
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.

7.3.51 Conversions involving single precision objects.

7.3.51.1 To single precision.

`ulong Rg_to_Fl(GEN z, ulong p)`, `z` which can be mapped to $\mathbf{Z}/p\mathbf{Z}$: 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 ℓ satisfying $p = \ell^n$ for some n (less than the accuracy of the input). Returns `lift(z * Mod(1,p))`, normalized, as an `F1`.

`ulong Rg_to_F2(GEN z)`, as `Rg_to_F1` for $p = 2$.

`ulong padic_to_F1(GEN x, ulong p)` special case of `Rg_to_F1`, for a `x` a `t_PADIC`.

`GEN RgX_to_F2x(GEN x)`, `x` a `t_POL`, returns the `F2x` obtained by applying `Rg_to_F1` coefficientwise.

`GEN RgX_to_Flx(GEN x, ulong p)`, `x` a `t_POL`, returns the `Flx` obtained by applying `Rg_to_F1` coefficientwise.

`GEN RgXV_to_FlxV(GEN x, ulong p)`, `x` a vector, returns the `FlxV` obtained by applying `RgX_to_Flx` coefficientwise.

`GEN Rg_to_F2xq(GEN z, GEN T)`, `z` a `GEN` which can be mapped to $\mathbf{F}_2[X]/(T)$: anything `Rg_to_F1` 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 $\mathbf{F}_p[X]/(T)$: anything `Rg_to_F1` 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 RgX_to_FlxqX(GEN z, GEN T, ulong p)`, `z` a `GEN` which can be mapped to $\mathbf{F}_p[x]/(T)[X]$: anything `Rg_to_Flxq` 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 `FlxqX`, 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 ZXT_to_FlxT(GEN v, ulong p), as ZX_to_Flx, repeatedly called on the tree leaves.

GEN ZXX_to_FlxX(GEN B, ulong p, long v), as ZX_to_Flx, repeatedly called on the polynomial's coefficients.

GEN zxX_to_FlxX(GEN z, ulong p) 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 ZXXT_to_FlxXT(GEN V, ulong p, long v), as ZXX_to_FlxX, repeatedly called on the tree leaves.

GEN RgV_to_Flv(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 ZVV_to_FlvV(GEN x, ulong p) reduce ZVV x modulo p (yielding an FlvV).

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`

7.3.51.2 From single precision.

GEN Flx_to_ZX(GEN z), converts to ZX (t_POL of nonnegative t_INT s 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 FlxXC_to_ZXXC(GEN B), converts an FlxXC to a t_COL with ZXX coefficients (repeated calls to FlxX_to_ZXX).

GEN FlxXM_to_ZXXM(GEN B), converts an FlxXM to a t_MAT with ZXX coefficients (repeated calls to FlxX_to_ZXX).

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 the coefficients to be nonnegative.
 GEN `zx_to_Flx`(GEN `z`, `ulong p`) as `Flx_red` without assuming the coefficients to be nonnegative.
 GEN `Flc_to_ZC`(GEN `z`), converts to `ZC` (`t_COL` of nonnegative `t_INTs` in this case)
 GEN `Flc_to_ZC_inplace`(GEN `z`), same as `Flc_to_ZC`, in place (`z` is destroyed).
 GEN `Flv_to_ZV`(GEN `z`), converts to `ZV` (`t_VEC` of nonnegative `t_INTs` in this case)
 GEN `Flm_to_ZM`(GEN `z`), converts to `ZM` (`t_MAT` with nonnegative `t_INTs` coefficients in this case)
 GEN `Flm_to_ZM_inplace`(GEN `z`), same as `Flm_to_ZM`, in place (`z` is destroyed).
 GEN `zc_to_ZC`(GEN `z`) as `Flc_to_ZC`, without assuming coefficients are nonnegative.
 GEN `zv_to_ZV`(GEN `z`) as `Flv_to_ZV`, without assuming coefficients are nonnegative.
 GEN `zm_to_ZM`(GEN `z`) as `Flm_to_ZM`, without assuming coefficients are nonnegative.
 GEN `zv_to_Flv`(GEN `z`, `ulong p`)
 GEN `zm_to_Flm`(GEN `z`, `ulong p`)

7.3.51.3 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 `RgMrow_zc_mul`(GEN `x`, GEN `y`, `long i`)
 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 `zv_ZM_mul`(GEN `x`, GEN `y`)
 GEN `ZV_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`.

7.3.51.4 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 sv) converts a t_INT to a scalar Flx polynomial. Assume that $sv = evalvarn(v)$ for some variable number 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 FlxM_to_FlxXV(GEN V, long v) see RgM_to_RgXV

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_Flx(GEN P) Let $P(x, X)$ be a FlxX, return $P(0, X)$ as a Flx.

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_FlxX(GEN x, long v) see RgV_to_RgX.

GEN FlxXV_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.

7.3.51.5 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 F2xC_to_FlxC(GEN x)

GEN FlxC_to_F2xC(GEN x)

GEN F2xV_to_F2m(GEN v, long n) F2x_to_F2v to each polynomial to get an F2m with n rows.

7.4 Higher arithmetic over Z: primes, factorization.

7.4.1 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 $64 \times 63 \times 65 \times 11$ 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 $k \geq 2$ 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 `*pmask` 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 nonzero, 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, `cutoff` > 0 and T be an iterator over primes ≥ 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 $x = \text{*ptn}$: it will start at the next prime.

7.4.2 Factorization.

GEN `Z_factor`(GEN `n`) factors the `t_INT` `n`. The “primes” in the factorization are actually strong pseudoprimes.

GEN `absZ_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 `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). Shallow function.

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 `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 the `addprimes` prime table. Return the corresponding factorization matrix. The first column of the factorization matrix may contain a single composite, which may or may not be the last entry in presence of a prime table.

If `lim = 0`, the effect is the same as setting `lim = maxprime() + 1`: use all precomputed primes.

int `Z_has_prime_3mod4`(GEN `n`) return 1 if and only if the non-zero integer `n` has a prime divisor congruent to 3 modulo 4.

GEN `absZ_factor_limit`(GEN `n`, ulong `all`) returns `Z_factor_limit(absi(n))`.

GEN `absZ_factor_limit_strict`(GEN `n`, ulong `lim`, GEN `*pU`). This function is analogous to `absZ_factor_limit`, with a better interface: trial divide `n` by all primes $p < \text{lim}$ (supposedly in the precomputed list of prime numbers, but `lim` may be larger) and then the full `addprimes` prime table (whatever their size). Return the corresponding factorization matrix. In this case, a composite cofactor is *not* included.

If `pU` is not NULL, set it to the cofactor, which is either NULL (no cofactor) or $[q, k]$, where $q > \text{lim}$ and $k > 0$. The prime divisors of q are greater than `lim` and not in the `addprimes` table, q is not a pure power, and q^k is the largest power of q dividing `n`. It may happen that q is prime.

GEN `boundfact`(GEN `x`, ulong `lim`) as `Z_factor_limit`, applying to `t_INT` or `t_FRAC` inputs.

GEN `Z_smooththen`(GEN `n`, GEN `L`, GEN `*pP`, GEN `*pE`) given a `t_VEC` `L` containing a list of primes and a `t_INT` `n`, trial divide `n` by the elements of `L` and return the cofactor. Return NULL if the cofactor is ± 1 . `*P` and `*E` contain the list of prime divisors found and their exponents, as `t_VECSMALLs`. Shallow function.

GEN `Z_lsmooththen`(GEN `n`, GEN `L`, GEN `*pP`, GEN `*pE`) as `Z_smooththen` where `L` is a `t_VECSMALL` of small primes and both `*P` and `*E` are given as `t_VECSMALL`.

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)`. Shallow function.

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 ($p\zeta_n$ is a square in $\mathbf{Q}(\zeta_n)$).

GEN factor_Aurifeuille(GEN *a*, long *n*) an Aurifeuillian factor of $\phi_n(a)$, assuming *a* is a nonzero 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 $p^{v_p(n)}$.

GEN vecfactoru(ulong *a*, ulong *b*), returns a **t_VEC** *v* containing the factorizations (**factoru** format) of $a, \dots, b$; assume that $b \geq a > 0$. Uses a sieve with primes up to $\sqrt{b}$. For all *c*, $a \leq c \leq b$, the factorization of *c* is given in $v[c - a + 1]$.

GEN vecfactoroddu(ulong *a*, ulong *b*), returns a **t_VEC** *v* containing the factorizations (**factoru** format) of odd integers in $a, \dots, b$; assume that $b \geq a > 0$ are odd. Uses a sieve with primes up to $\sqrt{b}$. For all odd *c*, $a \leq c \leq b$, the factorization of *c* is given in $v[(c - a)/2 + 1]$.

GEN vecfactoru_i(ulong *a*, ulong *b*), private version of **vecfactoru**, not stack-clean.

GEN vecfactoroddu_i(ulong *a*, ulong *b*), private version of **vecfactoroddu**, not stack-clean.

GEN vecfactorsquarefreeu(ulong *a*, ulong *b*) return a **t_VEC** *v* containing the prime divisors of squarefree integers in $a, \dots, b$; assume that $a \leq b$. Uses a sieve with primes up to $\sqrt{b}$. For all squarefree *c*, $a \leq c \leq b$, the prime divisors of *c* (as a **t_VECSMALL**) are given in $v[c - a + 1]$, and the other entries are NULL. Note that because of these NULL markers, *v* is not a valid **GEN**, it is not stack-clean.

GEN vecfactorsquarefreeu_coprime(ulong *a*, ulong *b*, GEN *P*) given a *sorted* **t_VECSMALL** of primes *P*, return a **t_VEC** *v* containing the prime divisors of squarefree integers in $a, \dots, b$ coprime to the elements of *P*; assume that $a \leq b$. Uses a sieve with primes up to $\sqrt{b}$. For all squarefree *c*, $a \leq c \leq b$, the prime divisors of *c* (as a **t_VECSMALL**) are given in $v[c - a + 1]$, and the other entries are NULL. Note that because of these NULL markers, *v* is not a valid **GEN**, it is not stack-clean.

GEN vecsquarefreeu(ulong *a*, ulong *b*) return a **t_VECSMALL** *v* containing the squarefree integers in $a, \dots, b$. Assume that $a \leq b$. Uses a sieve with primes up to $\sqrt{b}$.

ulong tridiv_bound(GEN *n*) returns the trial division bound used by **Z_factor**(*n*).

GEN tridiv_boundu(ulong *n*) returns the trial division bound used by **factorun**.

GEN Z_pollardbrent(GEN *N*, long *n*, long *seed*) try to factor **t_INT** *N* using $n \geq 1$ rounds of Pollard iterations; *seed* is an integer whose value (mod 8) selects the quadratic polynomial use to generate Pollard's (pseudo)random walk. Returns NULL on failure, else a vector of 2 (possibly 3) integers whose product is *N*.

GEN Z_ECM(GEN *N*, long *n*, long *seed*, ulong *B1*) try to factor **t_INT** *N* using $n \geq 1$ rounds of ECM iterations (on 8 to 64 curves simultaneously, depending on the size of *N*); *seed* is an integer whose value selects the curves to be used: increase it by $64n$ to make sure that a subsequent call

with a factor of N uses a disjoint set of curves. Finally $B_1 > 7$ determines the computations performed on the curves: we compute $[k]P$ for some point in $E(\mathbf{Z}/N\mathbf{Z})$ and $k = q \prod p^{e_p}$ where $p^{e_p} \leq B_1$ and $q \leq B_2 := 110B_1$; a higher value of B_1 means higher chances of hitting a factor and more time spent. The computation is deterministic for a given set of parameters. Returns NULL on failure, else a nontrivial factor or N .

GEN `Q_factor`(GEN `x`) as `Z_factor`, where x is a `t_INT` or a `t_FRAC`.

GEN `Q_factor_limit`(GEN `x`, `ulong lim`) as `Z_factor_limit`, where x is a `t_INT` or a `t_FRAC`.

7.4.3 Coprime factorization.

Given a and b two nonzero integers, let **ppi**(a, b), **ppo**(a, b), **ppg**(a, b), **pple**(a, b) (powers in a of primes inside b , outside b , greater than those in b , less than or equal to those in b) be the integers defined by

- $v_p(\text{ppi}) = v_p(a)[v_p(b) > 0]$,
- $v_p(\text{ppo}) = v_p(a)[v_p(b) = 0]$,
- $v_p(\text{ppg}) = v_p(a)[v_p(a) > v_p(b)]$,
- $v_p(\text{pple}) = v_p(a)[v_p(a) \leq v_p(b)]$.

GEN `Z_ppo`(GEN `a`, GEN `b`) returns `ppo(a, b)`; shallow function.

`ulong u_ppo`(`ulong a`, `ulong b`) returns `ppo(a, b)`.

GEN `Z_ppgle`(GEN `a`, GEN `b`) returns `[ppg(a, b), pple(a, b)]`; shallow function.

GEN `Z_ppio`(GEN `a`, GEN `b`) returns `[gcd(a, b), ppi(a, b), ppo(a, b)]`; shallow function.

GEN `Z_cba`(GEN `a`, GEN `b`) fast natural coprime base algorithm. Returns a vector of coprime divisors of a and b such that both a and b can be multiplicatively generated from this set. Perfect powers are not removed, use `Z_isanypower` if needed; shallow function.

GEN `ZV_cba_extend`(GEN `P`, GEN `b`) extend a coprime basis P by the integer b , the result being a coprime basis for $P \cup \{b\}$. Perfect powers are not removed; shallow function.

GEN `ZV_cba`(GEN `v`) given a vector of nonzero integers v , return a coprime basis for v . Perfect powers are not removed; shallow function.

7.4.4 Checks attached 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 nonzero 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 nonzero 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 nonzero 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 nonzero.

`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.

`GEN fuse_Z_factor(GEN f, GEN B)` assuming f is the factorization of an integer n , return `boundfact(n, B)`, i.e. return a factorization where all primary factors for $|p| \leq B$ are preserved, and all others are “fused” into a single composite integer; if that remainder is trivial, i.e. equal to 1, it is of course not included. Shallow function.

In the following three 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 attached 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 attached to the factorization of a nonzero 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 attached to the factorization of an integer, and return NULL (plain `t_INT`) or a factorization extracted from n otherwise.

7.4.5 Incremental integer factorization.

Routines attached 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 nonzero, 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 garbage collection 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.

7.4.6 Integer core, squarefree factorization.

`long Z_issquarefree(GEN n)` returns 1 if the `t_INT` n is square-free, and 0 otherwise.

`long Z_issquarefree_fact(GEN fa)` same, where fa is `factor(n)`.

`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 $p^2 \mid n$ 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 $p^2 \mid n$ implies $p > \text{lim}$.

7.4.7 Primes, primality and compositeness tests.

7.4.7.1 Chebyshev's π function, bounds.

`ulong uprimepi(ulong n)`, returns the number of primes $p \leq n$ (Chebyshev's π 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`.

7.4.7.2 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`.

GEN `mpprimal(long n)` return $n\#$, the product of primes less than or equal to n . The integer n must be non-negative.

7.4.7.3 Tests.

int `uisprime(ulong p)`, returns 1 if p is a prime number and 0 otherwise.

int `uisprime_101(ulong p)`, assuming that p has no divisor ≤ 101 , returns 1 if p is a prime number and 0 otherwise.

int `uisprime_661(ulong p)`, assuming that p has no divisor ≤ 661 , 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 nonprimes are included!

long `isprimeECPP(GEN n)`, returns 1 if the `t_INT` n is a prime number and 0 otherwise, using only the ECPP test. The workhorse `isprime` should be faster on average.

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 $k \geq 4$ and we expect it to be less reliable.

GEN `ecpp(GEN N)` returns an ECPP certificate for `t_INT` N ; underlies `primecert`.

GEN `ecpp0(GEN N, long t)` returns a (potentially) partial ECPP certificate for `t_INT` N where strong pseudo-primes $< 2^t$ are included as primes in the certificate. Underlies `primecert` with t set to the `partial` argument.

GEN `ecppexport(GEN cert, long flag)` export a PARI ECPP certificate to MAGMA or Primo format; underlies `primecertexport`.

long `ecppisvalid(GEN cert)` checks whether a PARI ECPP certificate is valid; underlies `primecertisvalid`.

long `check_ecppcert(GEN cert)` checks whether `cert` looks like a PARI ECPP certificate, (valid or invalid) without doing any computation.

7.4.8 Iterators over primes.

`int forprime_init(forprime_t *T, GEN a, GEN b)` initialize an iterator T over primes in $[a, b]$; over primes $\geq a$ if $b = \text{NULL}$. Return 0 if the range is known to be empty from the start (as if $b < a$ or $b < 0$), and return 1 otherwise. Use `forprime_next` to iterate over the prime collection.

`int forprimestep_init(forprime_t *T, GEN a, GEN b, GEN q)` initialize an iterator T over primes in an arithmetic progression in $[a, b]$; over primes $\geq a$ if $b = \text{NULL}$. The argument q is either a `t_INT` ($p \equiv a \pmod{q}$) or a `t_INTMOD` $\text{Mod}(c, N)$ and we restrict to that congruence class. Return 0 if the range is known to be empty from the start (as if $b < a$ or $b < 0$), and return 1 otherwise. Use `forprime_next` to iterate over the prime collection.

`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(&T, a, b)`, possibly followed by a number of calls to `u_forprime_next`, and $a \leq c \leq b$. 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 . Subsequent calls to `u_forprime_next` will only return primes congruent to c modulo q . Note that unless $(c, q) = 1$ there will be at most one such prime.

7.5 Integral, rational and generic linear algebra.

7.5.1 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.

7.5.1.1 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_ZVpos(GEN x)` Assuming x is a `t_VEC` or `t_COL` return 1 if it is a ZV with positive entries, and 0 otherwise.

`int RgV_is_ZVnon0(GEN x)` Assuming x is a `t_VEC` or `t_COL` return 1 if it is a ZV with nonzero entries, 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 RgV_is_arithprog(GEN v, GEN *a, GEN *b)` assuming x is a `t_VEC` or `t_COL` return 1 if its entries follow an arithmetic progression of the form $a + b * n$, $n = 0, 1, \dots$ and set a and b . Else return 0.

`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).

`int ZV_abscmp(GEN x, GEN y)` compare two ZV, which we assume have the same length (lexicographic order, comparing absolute values).

`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 identity_ZV(long n)` return the `t_VEC` $[1, 2, \dots, n]$.

`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 Z_ZC_sub(GEN a, GEN x)` returns the vector $[a - x_1, -x_2, \dots, -x_n]$.

`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 ZC_divexactu(GEN x, ulong y)` returns x/y assuming all divisions are exact.

`GEN ZC_Z_div(GEN x, GEN y)` returns x/y , where the resulting vector has rational entries.

`GEN ZV_ZV_mod(GEN a, GEN b)`. Assuming a and b are two ZV of the same length, returns the vector whose i -th component is `modii(a[i], b[i])`.

`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 $X \leftarrow X + vY$, 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 shallow.

`void ZC_lincomb1_inplace_i(GEN X, GEN Y, GEN v, long n)` variant of `ZC_lincomb1_inplace`: only update $X[1], \dots, X[n]$, assuming that $n < \lg(X)$.

`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_extgcd(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 $GL_n(\mathbf{Z})$ such that $AU = [D, 0, \dots, 0]$.

`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 .

`long ZV_max_expi(GEN x)` returns the number of bits of the largest 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_shallow(GEN L)` shallow version of `ZV_sort`.

`void ZV_sort_inplace(GEN L)` sort the ZV L , in place.

`GEN ZV_sort_uniq(GEN L)` sort the ZV L , removing duplicate entries. Returns a vector with the same type as L .

`GEN ZV_sort_uniq_shallow(GEN L)` shallow version of `ZV_sort_uniq`.

`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 .

`GEN ZC_union_shallow(GEN x, GEN y)` as `ZV_union_shallow` but return a `t_COL`.

7.5.1.2 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 RgM_rescale_to_int(GEN x)` given a matrix x with real entries (`t_INT`, `t_FRAC` or `t_REAL`), return a ZM which is very close to Dx for some well-chosen integer D . More precisely, if the input is exact, D is the denominator of x ; else it is a power of 2 chosen so that all inexact entries are correctly rounded to 1 ulp.

`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.

`int ZM_equal0(GEN A)` returns 1 if the ZM A is identically equal to 0.

`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 `gen_1`, `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 ZM2_mul(GEN x, GEN y)` multiplies the two-by-two ZM x and y .

`GEN ZM_sqr(GEN x)` returns x^2 , where x is a square ZM.

`GEN ZM2_sqr(GEN x)` returns x^2 , where x is a two-by-two ZM.

`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_ZX_mul(GEN x, GEN T)` returns $x \times y$, where y is `RgX_to_RgC(T, lg(x) - 1)`.

`GEN ZM_diag_mul(GEN d, GEN m)` given a vector d with integer entries and a ZM m of compatible dimensions, return `diagonal(d) * m`.

`GEN ZM_mul_diag(GEN m, GEN d)` given a vector d with integer entries and a ZM m of compatible dimensions, return `m * diagonal(d)`.

`GEN ZM_multosym(GEN x, GEN y)`

`GEN ZM_transmultosym(GEN x, GEN y)`

`GEN ZM_transmul(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 nonempty and $0 < i < \text{lg}(x[1])$.

`int ZMrow_equal0(GEN V, long i)` returns 1 if the i -th row of the ZM V is zero, and 0 otherwise.

`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_divexactu(GEN x, ulong y)` returns x/y assuming all divisions are exact.

`GEN ZM_Z_div(GEN x, GEN y)` returns x/y , where the resulting matrix has rational entries.

`GEN ZM_ZV_mod(GEN a, GEN b)`. Assuming a is a ZM whose columns have the same length as the ZV b , apply `ZV_ZV_mod(a[i], b)` to all columns.

`GEN ZC_Q_mul(GEN x, GEN y)` returns $x*y$, where y is a rational number and the resulting `t_COL` has rational entries.

`GEN ZM_Q_mul(GEN x, GEN y)` returns $x*y$, where y is a rational number and the resulting matrix has rational entries.

`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 ZMV_prod(GEN v)` returns the product of the elements of v .

`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_permanent`(GEN `M`) if `M` is a ZM, returns its permanent. This is the function underlying `mat-permanent` whenever `M` is a ZM. It assumes that the matrix is square of dimension $< \text{BITS_IN_LONG}$.

GEN `ZM_detmult`(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_trace`(GEN `M`) returns the trace of the ZM `M`.

GEN `ZM_imagecompl`(GEN `x`) returns `matimagecompl(x)`.

long `ZM_rank`(GEN `x`) returns `matrank(x)`.

GEN `ZM_ker`(GEN `x`) returns the primitive part of `matker(x)`; in other words the $\mathbf{Q}$ -basis vectors are made integral and primitive.

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 the effective length of the longest entry in `x`.

long `ZM_max_expi`(GEN `x`) returns the number of bits of the largest entry in `x`.

GEN `ZM_inv`(GEN `M`, GEN `*pd`) if `M` is a ZM, return a primitive matrix H such that MH is d times the identity and set `*pd` to d . Uses a multimodular algorithm up to Hadamard's bound. If you suspect that the denominator is much smaller than $\det M$, you may use `ZM_inv_ratlift`.

GEN `ZM_inv_ratlift`(GEN `M`, GEN `*pd`) if `M` is a ZM, return a primitive matrix H such that MH is d times the identity and set `*pd` to d . Uses a multimodular algorithm, attempting rational reconstruction along the way. To be used when you expect that the denominator of M^{-1} is much smaller than $\det M$ else use `ZM_inv`.

GEN `SL2_inv_shallow`(GEN `M`) return the inverse of $M \in \text{SL}_2(\mathbf{Z})$. Shallow function.

GEN `ZM_pseudoinv`(GEN `M`, GEN `*pv`, GEN `*pd`) if `M` is a nonempty ZM, let $v = [y, z]$ returned by `indexrank` and let M_1 be the corresponding square invertible matrix. Return a primitive left-inverse H such that HM_1 is d times the identity and set `*pd` to d . If `pv` is not NULL, set `*pv` to v . Shallow function.

GEN `ZM_gauss`(GEN `a`, GEN `b`) as `gauss`, where a and b coefficients are `t_INTs`.

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 if the ZM `x` is the identity matrix, and 0 otherwise.

int `ZM_isdiagonal`(GEN `x`) return 1 if the ZM `x` is diagonal, and 0 otherwise.

int `ZM_isscalar`(GEN `x`, GEN `s`) given a ZM `x` and a `t_INT` `s`, return 1 if `x` is equal to `s` times the identity, and 0 otherwise. If `s` is NULL, test whether `x` is an arbitrary scalar matrix.

long `ZC_is_ei`(GEN `x`) return i if the ZC `x` has 0 entries, but for a 1 at position i .

int `ZM_ishnf`(GEN `x`) return 1 if `x` is in HNF form, i.e. is upper triangular with positive diagonal coefficients, and for $j > i$, $x_{i,i} > x_{i,j} \geq 0$.

7.5.2 QM.

GEN `QM_charpoly_ZX`(GEN `M`) returns the characteristic polynomial (in variable 0) of the QM `M`, assuming that the result has integer coefficients.

GEN `QM_charpoly_ZX_bound`(GEN `M`, long `b`) as `QM_charpoly_ZX` assuming that the sup norm of the (integral) result is $\leq 2^b$.

GEN `QM_gauss`(GEN `a`, GEN `b`) as `gauss`, where `a` and `b` coefficients are `t_FRACs`.

GEN `QM_gauss_i`(GEN `a`, GEN `b`, long `flag`) as `QM_gauss` if `flag` is 0. Else, no longer assume that `a` is left-invertible and return a solution of $Pax = Pb$ where `P` is a row-selection matrix such that $A = PaQ$ is square invertible of maximal rank, for some column-selection matrix `Q`; in particular, `x` is a solution of the original equation $ax = b$ if and only if a solution exists.

GEN `QM_indexrank`(GEN `x`) returns `matindexrank(x)`.

GEN `QM_inv`(GEN `M`) return the inverse of the QM `M`.

long `QM_rank`(GEN `x`) returns `matrank(x)`.

GEN `QM_image`(GEN `x`) returns an integral matrix with primitive columns generating the image of `x`.

GEN `QM_image_shallow`(GEN `A`) shallow version of the previous function.

int `QM_ishnf`(GEN `x`) return 1 if `dx` is in HNF form for some positive integer `x`.

7.5.3 Qevproj.

GEN `Qevproj_init`(GEN `M`) let `M` be a $n \times d$ ZM of maximal rank $d \leq n$, representing the basis of a $\mathbf{Q}$ -subspace `V` of $\mathbf{Q}^n$. Return a projector on `V`, to be used by `Qevproj_apply`. The interface details may change in the future, but this function currently returns $[M, B, D, p]$, where `p` is a `t_VECSMALL` with `d` entries such that the submatrix $A = \text{rowpermute}(M, p)$ is invertible, `B` is a ZM and `d` a `t_INT` such that $AB = D\text{Id}_d$.

GEN `Qevproj_apply`(GEN `T`, GEN `pro`) let `T` be an $n \times n$ QM, stabilizing a $\mathbf{Q}$ -subspace $V \subset \mathbf{Q}^n$ of dimension `d`, and let `pro` be a projector on that subspace initialized by `Qevproj_init(M)`. Return the $d \times d$ matrix representing $T|_V$ on the basis given by the columns of `M`.

GEN `Qevproj_apply_vecei`(GEN `T`, GEN `pro`, long `k`) as `Qevproj_apply`, return only the image of the `k`-th basis vector `M[k]` (still on the basis given by the columns of `M`).

GEN `Qevproj_down`(GEN `T`, GEN `pro`) given a ZC (resp. a ZM) `T` representing an element (resp. a vector of elements) in the subspace `V` return a QC (resp. a QM) `U` such that $T = MU$.

7.5.4 zv, zm.

GEN identity_zv(long n) return the t_VEC SMALL $[1, 2, \dots, n]$.

GEN random_zv(long n) returns a random zv with n components.

GEN zv_abs(GEN x) return $[|x[1]|, \dots, |x[n]|]$ as a zv.

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 $2^{\text{BITS_IN_LONG}-1}$.

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 $2^{\text{BITS_IN_LONG}-1}$.

int zv_canon_inplace(GEN x) return 1 if the first non-zero entry of x is positive (or x is 0); else return -1 and replace x by $-x$ in place by negating its entries. No check for overflow is done, which occurs in the fringe case where an entry is equal to $2^{\text{BITS_IN_LONG}-1}$.

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).

long zv_sumpart(GEN v, long n) returns the sum $v[1] + \dots + v[n]$ (assumes no overflow occurs and $\lg(v) > n$).

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 zm_row(GEN A, long x0) as Flm_row.

GEN zv_diagonal(GEN v) return the square zm whose diagonal is given by the entries of v .

GEN zm_permanent(GEN M) return the permanent of M . The function assumes that the matrix is square of dimension $< \text{BITS_IN_LONG}$.

int zvV_equal(GEN x, GEN y) returns 1 if the two zvV (vectors of zv) are equal and 0 otherwise.

7.5.5 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)

GEN ZMV_to_FlmV(GEN z, ulong m)

7.5.6 QC / QV, QM.

GEN QM_mul(GEN x, GEN y) multiplies x and y (assumed to have compatible dimensions).

GEN QM_sqr(GEN x) returns the square of x (assumed to be square).

GEN QM_QC_mul(GEN x, GEN y) multiplies x and y (assumed to have compatible dimensions).

GEN QM_det(GEN M) returns the determinant of M .

GEN QM_ker(GEN x) returns `matker(x)`.

7.5.7 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.

7.5.7.1 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 $-x$.

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 $x + y * \text{Id}$.

GEN RgM_Rg_add_shallow(GEN x, GEN y) as RgM_Rg_add with much fewer copies. Shallow function.

GEN RgM_Rg_sub(GEN x, GEN y) assuming x is a square matrix and y a scalar, returns the square matrix $x - y * \text{Id}$.

GEN RgM_Rg_sub_shallow(GEN x, GEN y) as RgM_Rg_sub with much fewer copies. Shallow function.

GEN RgC_Rg_add(GEN x, GEN y) assuming x is a nonempty column vector and y a scalar, returns the vector $[x_1 + y, x_2, \dots, x_n]$.

GEN RgC_Rg_sub(GEN x, GEN y) assuming x is a nonempty column vector and y a scalar, returns the vector $[x_1 - y, x_2, \dots, x_n]$.

GEN Rg_RgC_sub(GEN a, GEN x) assuming x is a nonempty column vector and a a scalar, returns the vector $[a - x_1, -x_2, \dots, -x_n]$.

GEN RgC_Rg_div(GEN x, GEN y)

GEN RgM_Rg_div(GEN x, GEN y) returns x/y (y treated as a scalar).

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 RgV_RgC_mul(GEN x, GEN y) returns $x \times y$.

GEN RgV_RgM_mul(GEN x, GEN y) returns $x \times y$.

GEN RgM_RgC_mul(GEN x, GEN y) returns $x \times y$.

GEN RgM_RgX_mul(GEN x, GEN T) returns $x \times y$, where y is `RgX_to_RgC(T, lg(x) - 1)`.

GEN RgM_mul(GEN x, GEN y) returns $x \times y$.

GEN RgM_div(GEN a, GEN b) returns ab^{-1} or NULL if this turns out to be impossible.

GEN RgM_ZM_mul(GEN x, GEN y) returns $x \times y$ assuming that y is a ZM.

GEN RgM_transmul(GEN x, GEN y) returns $x^{\sim} \times y$.

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 $x^{\sim} \times y$, 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 nonempty and $0 < i < \text{lg}(x[1])$.

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 RgM_sqr(GEN x) returns x^2 .

GEN RgC_RgV_mul(GEN x, GEN y) returns $x \times y$ (the matrix $(x_i y_j)$).

GEN RgC_RgV_mulrealsym(GEN x, GEN y) returns the real part of $x \times y$ (whose entries are `t_INT`, `t_FRAC`, `t_REAL` or `t_COMPLEX`), assuming the result is symmetric.

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 $x \times y[1,]$ if y is a row matrix $1 \times n$, error otherwise.

GEN RgM_RgV_mul(GEN x, GEN y) returns $x \times y[, 1]$ if y is a column matrix $n \times 1$, error otherwise.

GEN RgM_powers(GEN x, long n) returns $[x^0, \dots, x^n]$ as a t_VEC of RgMs.

GEN RgV_sum(GEN v) sum of the entries of v

GEN RgV_prod(GEN v) product of the entries of v , using a divide and conquer strategy

GEN RgV_sumpart(GEN v, long n) returns the sum $v[1] + \dots + v[n]$ (assumes that $\lg(v) > n$).

GEN RgV_sumpart2(GEN v, long m, long n) returns the sum $v[m] + \dots + v[n]$ (assumes that $\lg(v) > n$ and $m > 0$). Returns gen_0 when $m > n$.

GEN RgM_sumcol(GEN v) returns a t_COL, sum of the columns of the t_MAT v .

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 RgV_kill0(GEN v) returns a shallow copy of v where entries matched by `gequal0` are replaced by NULL. The return value is not a valid GEN and must be handled specially. The idea is to pre-treat a vector of coefficients to speed up later linear combinations or scalar products.

GEN gram_matrix(GEN v) returns the Gram matrix $(v_i \cdot v_j)$ attached 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 $X = [1, 2, \dots, n]$, where n is the length of Y . This is the function underlying `polint` for formal interpolation.

GEN polintspec(GEN X, GEN Y, GEN t, long n, long *pe) return $P(t)$ where P is the Lagrange interpolation polynomial attached to the n points $(X[0], Y[0]), \dots, (X[n-1], Y[n-1])$. If `pe` is not NULL and t is a complex numeric value, `*pe` contains an error estimate for the returned value (Neville's algorithm, see `polinterpolate`). In extrapolation algorithms, e.g., Romberg integration, this function is usually called on actual GEN vectors with offsets: $x+k$ and $y+k$ so as to interpolate on $x[k..k+n-1]$ without having to use `vecslice`. This is the function underlying `polint` for numerical interpolation.

GEN polint_i(GEN X, GEN Y, GEN t, long *pe) as `polintspec`, where X and Y are actual GEN vectors.

GEN vandermondeinverse(GEN r, GEN T, GEN d, GEN V) Given a vector r of n scalars and the t_POL $T = \prod_{i=1}^n (X - r_i)$, return dM^{-1} , where $M = (r_i^{j-1})_{1 \leq i, j \leq n}$ is the van der Monde matrix; V is NULL or a vector containing the $T'(r_i)$, as returned by `vandermondeinverseinit`. The demonimator d may be set to NULL (handled as 1). If c is the k -th column of the result, it is essentially d times the k -th Lagrange interpolation polynomial: we have $\sum_j c_j r_i^{j-1} = d\delta_{i=k}$.

This function underlies `RgV_polint` when the base field is not $\mathbf{Z}/p\mathbf{Z}$: it uses $O(n^2)$ scalar operations and is asymptotically slower than variants using multi-evaluation such as `FpV_polint`; it is also accurate over inexact fields.

A technical variant allows V from `vandermondeinverseinit_real` (complex roots up to conjugation). Then the output need no longer be a square matrix: only the columns corresponding to the given roots representative are included. (The others are implied, being their conjugates.) This allows to reconstruct an algebraic number in $\mathbf{Q}[X]/(T)$ given $r_1 + r_2$ real and (non-conjugate) complex embeddings only, instead of $r_1 + 2r_2 = \deg T$.

GEN vandermondeinverseinit(GEN r) Given a vector r of n scalars, let T be the $\mathbf{t_POL}$ $T = \prod_{j=1}^n (X - r_j)$. This function returns the $T'(r_i)$ computed stably via products of difference: the i -th entry is $T'(r_i) = \prod_{j \neq i} (r_i - r_j)$. It is asymptotically slow (uses $O(n^2)$ scalar operations, where multi-evaluation achieves quasi-linear running time) but allows accurate computation at low accuracies when T has large complex coefficients.

GEN vandermondeinverseinit_real(GEN R, long r1) This variant improves on **vandermondeinverse** in the important special case where the roots are complex and T has real coefficients, i.e., the set of roots is stable under complex conjugation. The argument r is replaced by a set R of representatives modulo conjugation: the first r_1 elements of R are real and the remaining r_2 are representatives of pairs of complex conjugates. This returns the $T'(R_i)$, from which the $T'(r_i)$ can be reconstructed by adding appropriate conjugates.

7.5.7.2 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 $1, T, \dots, T^{n-1}$, or on **nf.zk** (whose first element is 1).

int QV_isscalar(GEN x) as **RgV_isscalar**, assuming x is a QV ($\mathbf{t_INT}$ and $\mathbf{t_FRAC}$ entries only).

int ZV_isscalar(GEN x) as **RgV_isscalar**, assuming x is a ZV ($\mathbf{t_INT}$ 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 if the $\mathbf{t_MAT}$ x is the identity matrix, and 0 otherwise.

int RgM_isdiagonal(GEN x) return 1 if the $\mathbf{t_MAT}$ x is a diagonal matrix, and 0 otherwise.

long RgC_is_ei(GEN x) return i if the $\mathbf{t_COL}$ x has 0 entries, but for a 1 at position i .

int RgM_is_ZM(GEN x) return 1 if the $\mathbf{t_MAT}$ x has only $\mathbf{t_INT}$ coefficients, and 0 otherwise.

long qfiseven(GEN M) return 1 if the square symmetric $\mathbf{typZM}$ x is an even quadratic form (all diagonal coefficients are even), and 0 otherwise.

int RgM_is_QM(GEN x) return 1 if the $\mathbf{t_MAT}$ x has only $\mathbf{t_INT}$ or $\mathbf{t_FRAC}$ 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.

long RgV_isin_i(GEN v, GEN x, long n) return the first index $i \leq n$ such that $v[i] = x$ if it exists, and 0 otherwise. Naive search in linear time, does not assume that v is sorted. Assume that $n < \lg(v)$.

GEN RgM_diagonal(GEN m) returns the diagonal of m as a $\mathbf{t_VEC}$.

GEN RgM_diagonal_shallow(GEN m) shallow version of **RgM_diagonal**

7.5.7.3 Conversion to floating point entries.

GEN RgC_gtofp(GEN x, GEN prec) returns the `t_COL` obtained by applying `gtofp(gel(x,i), prec)` to all coefficients of x .

GEN RgV_gtofp(GEN x, GEN prec) returns the `t_VEC` obtained by applying `gtofp(gel(x,i), prec)` to all coefficients of x .

GEN RgC_gtomp(GEN x, long prec) returns the `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 `t_MAT` obtained by applying `gtofp(gel(x,i), prec)` to all coefficients of x .

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) )`

7.5.7.4 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 nonempty invertible upper triangular matrix, hence a little faster. The diagonal coefficients of the inverse matrix are computed using `ginv(gcoeff(a,i,i))`; in particular, they remain rational when the original diagonal coefficients were rational.

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 an upper bound for the absolute value of $\det(a)$. The bound is a `t_INT`.

GEN RgM_solve(GEN a, GEN b) returns $a^{-1}b$ 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 $b = \text{NULL}$, 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 $r_1 + r_2$ rows and $r_1 + 2r_2$ 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 r_1 rows of M and y are real (the r_2 others are complex), and x is real. This is stabler and faster than calling `RgM_solve(M,b)` over $\mathbf{C}$. 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 $r_1 + r_2$ rows, whose first r_1 rows have real entries (the r_2 others are complex). Return an object of the same type as `x` and $r_1 + 2r_2$ rows, such that the first $r_1 + r_2$ rows contain the real part of `x`, and the r_2 following ones contain the imaginary part of the last r_2 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 $M - x\text{Id}$, where M is an $n \times n$ square matrix. Returns the Frobenius form of M .

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 (Gram-Schmidt) orthogonalized vectors $x[i]^*$, `*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`: Q is orthogonal and R is upper triangular with positive diagonal, both matrices having `t_INT` or `t_REAL` coefficients. Abort and returns 0 is `x` if not invertible or we hit a precision problem (none of B , Q , L should be accessed in that case), 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 accomodate `qfgaussred` conventions. Not stack-clean.

GEN `RgM_gram_schmidt`(GEN `e`, GEN `*ptB`) naive (unstable) Gram-Schmidt orthogonalization of the basis (e_i) given by the columns of `t_MAT` `e`. Return the e_i^* (as columns of a `t_MAT`) and set `*ptB` to the vector of squared lengths $|e_i^*|^2$.

GEN `RgM_Babai`(GEN `M`, GEN `y`) given a `t_MAT` M of maximal rank n and a `t_COL` y of the same dimension, apply Babai's nearest plane algorithm to return an *integral* x such that $y - Mx$ has small L_2 norm. This yields an approximate solution to the closest vector problem: if M is LLL-reduced, then

$$\|y - Mx\|_2 \leq 2(2/\sqrt{3})^n \|y - MX\|_2$$

for all $X \in \mathbf{Z}^n$.

GEN `RgM_Cholesky`(GEN `M`) given a square symmetric `t_MAT` M , return R such that ${}^tRR = M$, or NULL is no solution is found (M is not positive or was given with insufficient accuracy).

7.5.8 ZG.

Let G be a multiplicative group with neutral element 1_G whose multiplication is supported by `gmul` and where equality test is performed using `gidentical`, e.g. a matrix group. The following routines implement basic computations in the group algebra $\mathbf{Z}[G]$. All of them are shallow for efficiency reasons. A **ZG** is either

- a `t_INT` n , representing $n[1_G]$
- or a “factorization matrix” with two columns $[g, e]$: the first one contains group elements, sorted according to `cmp_universal`, and the second one contains integer “exponents”, representing $\sum e_i[g_i]$.

Note that `to_famat` and `to_famat_shallow(g, e)` allow to build the **ZG** $e[g]$ from $e \in \mathbf{Z}$ and $g \in G$.

`GEN ZG_normalize(GEN x)` given a `t_INT` x or a factorization matrix *without* assuming that the first column is properly sorted. Return a valid (sorted) **ZG**. Shallow function.

`GEN ZG_add(GEN x, GEN y)` return $x + y$; shallow function.

`GEN ZG_neg(GEN x)` return $-x$; shallow function.

`GEN ZG_sub(GEN x, GEN y)` return $x - y$; shallow function.

`GEN ZG_mul(GEN x, GEN y)` return xy ; shallow function.

`GEN ZG_G_mul(GEN x, GEN y)` given a **ZG** x and $y \in G$, return xy ; shallow function.

`GEN G_ZG_mul(GEN x, GEN y)` given a **ZG** y and $x \in G$, return xy ; shallow function.

`GEN ZG_Z_mul(GEN x, GEN n)` given a **ZG** x and $y \in \mathbf{Z}$, return xy ; shallow function.

`GEN ZGC_G_mul(GEN v, GEN x)` given v a vector of **ZG** and $x \in G$ return the vector (with the same type as v with entries $v[i] \cdot x$. Shallow function.

`void ZGC_G_mul_inplace(GEN v, GEN x)` as `ZGC_G_mul`, modifying v in place.

`GEN ZGC_Z_mul(GEN v, GEN n)` given v a vector of **ZG** and $n \in \mathbf{Z}$ return the vector (with the same type as v with entries $n \cdot v[i]$. Shallow function.

`GEN G_ZGC_mul(GEN x, GEN v)` given v a vector of **ZG** and $x \in G$ return the vector of $x \cdot v[i]$. Shallow function.

`GEN ZGCs_add(GEN x, GEN y)` add two sparse vectors of **ZG** elements (see Sparse linear algebra below).

7.5.9 Sparse and 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 $\mathbf{F}_p$. 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.

F2Ms are **t_VEC** of **F2Cs**. **F2Cs** are **t_VECSMALL** whoses entries are the nonzero coefficients (1).

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 **F2Ms_to_F2m**(GEN M , long **nbrow**) convert a **F2m** to a **F2Ms**.

GEN **F2m_to_F2Ms**(GEN M) convert a **F2m** to a **F2Ms**.

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 **zMs_compress**(GEN M , long **nbrow**) Renumber the rows to use consecutive integers starting at 1, skipping unused ones. Return $[L, p]$ where L is the new **zMs** and p is a **t_VECSMALL** giving the correspondance from the new column numbers to the original ones. the new number of rows is the length of p .

GEN **FpMs_FpC_mul**(GEN M , GEN B , GEN p) multiply the sparse matrix M (over $\mathbf{F}_p$) by the **FpC** 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 **ZC** B (over $\mathbf{Z}$). The result is an **ZC**, i.e. a dense vector. (Not stack clean).

GEN **FpV_FpMs_mul**(GEN B , GEN M , GEN p) multiply the **FpV** B by the sparse matrix M (over $\mathbf{F}_p$). The result is an **FpV**, i.e. a dense vector.

GEN **ZV_zMs_mul**(GEN B , GEN M , GEN p) multiply the **FpV** B (over $\mathbf{Z}$) 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 **F2Ms_colelim**(GEN M , long **nbrow**) returns some subset of the columns of M as a **t_VECSMALL** of indices, selected such that the dimension of the kernel of the matrix is preserved. The subset is not guaranteed to be minimal.

GEN F2Ms_ker(GEN M, long nbrow) returns some kernel vectors of M using block Lanczos algorithm.

GEN FpMs_leftkernel_elt(GEN M, long nbrow, GEN p) M being a sparse matrix over $\mathbf{F}_p$, return a nonzero FpV 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 $\mathbf{F}_p$. Return either a solution as a t_COL (dense vector), the index of a column which is linearly dependent from the others as a t_VECSMALL with a single component, or NULL (can happen if B is not in the image of M).

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 NULL.

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 $\mathbf{Z}/p^e\mathbf{Z}$. Return either a solution as a t_COL (dense vector), or the index of a column which is linearly dependent from the others as a t_VECSMALL with a single component, or NULL (can happen if B is not in the image of M).

GEN gen_FpM_Wiedemann(void *E, GEN (*f)(void*, GEN), GEN B, GEN p) solve the equation $f(X) = B$ over $\mathbf{F}_p$, where B is a FpV, 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 t_COL, or a kernel vector as a t_VEC, or NULL (can happen if B is not in the image of M).

GEN gen_ZpM_Dixon_Wiedemann(void *E, GEN (*f)(void*, GEN), GEN B, GEN p, long e) solve equation $f(X) = B$ over $\mathbf{Z}/p^e\mathbf{Z}$, where B is a ZV, 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 t_COL, or a kernel vector as a t_VEC, or NULL (can happen if B is not in the image of M).

GEN gen_ZpM_Dixon(GEN F, GEN B, GEN q, GEN p, long N, void *E, GEN lin(void *, GEN F, GEN d, GEN q), GEN inv(void *E, GEN d)) Let F be a square invertible ZpM of dimension k representing an automorphism f of $(\mathbf{Z}_p)^k$, let B be an element of $(\mathbf{Z}_p)^k$ and let $q = p^N$. Return $y \in (\mathbf{Z}_p)^k$ such that $f(y) = B \pmod{p^N}$ assuming the following holds for $n \leq N$:

- $\text{lin}(E, \text{FpM_red}(F, p^n), z, p^n) \equiv f(z) \pmod{p^n}$
- $f(\text{inv}(E, z)) \equiv z \pmod{p}$

The argument F is redundant; including it allows the function to reduce it to the required p -adic precision when evaluating f modulo p^n .

7.5.10 Obsolete functions.

The functions in this section are kept for backward compatibility only and will eventually disappear.

GEN image2(GEN x) compute the image of x using a very slow algorithm. Use `image` instead.

7.6 Integral, rational and generic polynomial arithmetic.

7.6.1 ZX.

`void RgX_check_ZX(GEN x, const char *s)` Assuming `x` is a `t_POL` raise an error if it is not a `ZX` (`s` should point to the name of the caller).

`GEN ZX_copy(GEN x, GEN p)` returns a copy of `x`.

`long ZX_max_lg(GEN x)` returns the effective length of the longest component in `x`.

`GEN scalar_ZX(GEN x, long v)` returns the constant `ZX` in variable `v` equal to the `t_INT` `x`.

`GEN scalar_ZX_shallow(GEN x, long v)` returns the constant `ZX` in variable `v` equal to the `t_INT` `x`. Shallow function.

`GEN ZX_renormalize(GEN x, long l)`, as `normalizpol`, where `l = lg(x)`, in place.

`int ZX_equal(GEN x, GEN y)` returns 1 if the two `ZX` have the same `degpol` and their coefficients are equal. Variable numbers are not checked.

`int ZX_equal1(GEN x)` returns 1 if the `ZX` `x` is equal to 1 and 0 otherwise.

`int ZX_is_monic(GEN x)` returns 1 if the `ZX` `x` is monic and 0 otherwise. The zero polynomial considered not monic.

`GEN ZX_add(GEN x, GEN y)` adds `x` and `y`.

`GEN ZX_sub(GEN x, GEN y)` subtracts `x` and `y`.

`GEN ZX_neg(GEN x)` returns $-x$.

`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_divuexact(GEN x, ulong 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`) in variable 0.

`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`) in variable 0.

GEN ZX_rem(GEN x, GEN y) returns the remainder of the Euclidean division of $x \bmod y$. 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_div_by_X_1(GEN T, GEN *r) return the quotient of T by $X - 1$. If r is not NULL set it to $T(1)$.

GEN ZX_digits(GEN x, GEN B) returns a vector of ZX $[c_0, \dots, c_n]$ of degree less than the degree of B and such that $x = \sum_{i=0}^n c_i B^i$. Assume that B is monic.

GEN ZXV_ZX_fromdigits(GEN v, GEN B) where $v = [c_0, \dots, c_n]$ is a vector of ZX, returns $\sum_{i=0}^n c_i B^i$.

GEN ZX_gcd(GEN x, GEN y) returns a gcd of the ZX x and y .

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 the primitive part of x/d . If x and y have coprime leading terms, then d is monic and $*pX$ is $Q_primpart(x)$ divided by d . Not stack-clean.

GEN ZX_radical(GEN x) returns the largest squarefree primitive divisor of the ZX x . Not stack-clean.

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 nonzero 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. Shallow function.

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. Shallow function.

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 $Q(X) := L^{-\deg q} q(Lx)$ 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 . Shallow function.

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. Shallow function.

GEN ZX_Q_mul(GEN x, GEN y) returns $x * y$, where y is a rational number and the resulting t_POL has rational entries.

long ZX_deflate_order(GEN P) given a nonconstant ZX P , returns the largest exponent d such that P is of the form $P(x^d)$.

long ZX_deflate_max(GEN P, long *d). Given a nonconstant polynomial with integer coefficients P , sets d to ZX_deflate_order(P) and returns RgX_deflate(P, d). Shallow function.

GEN ZX_rescale(GEN P, GEN h) returns $h^{\deg(P)} P(x/h)$. P is a ZX and h is a nonzero integer. Shallow function.

GEN ZX_rescale2n(GEN P, long n) returns $2^{n \deg(P)} P(x \gg n)$ where P is a ZX.

GEN ZX_rescale_lt(GEN P) returns the monic integral polynomial $h^{\deg(P)-1}P(x/h)$, where P is a nonzero ZX and h is its leading coefficient. Shallow function.

GEN ZX_Z_translate(GEN P, GEN c) assume P is a ZX and c an integer. Returns $P(X + c)$ (optimized for $c = \pm 1$).

GEN ZX_affine(GEN P, GEN a, GEN b) P is a ZX, a and b are t_INT. Return $P(aX + b)$ (optimized for $b = \pm 1$). Not memory clean.

GEN ZX_Z_eval(GEN P, GEN x) evaluate the ZX P at the integer x.

GEN ZX_unscale(GEN P, GEN h) given a ZX P and a t_INT h, returns $P(hx)$. Assume $h \neq 0$. Not stack-clean.

GEN ZX_z_unscale(GEN P, long h) given a ZX P, returns $P(hx)$. Assume $h \neq 0$. Not stack-clean.

GEN ZX_unscale2n(GEN P, long n) given a ZX P, returns $P(x \ll n)$. Not stack-clean.

GEN ZX_unscale_div(GEN P, GEN h) given a ZX P and a t_INT h such that $h \mid P(0)$, returns $P(hx)/h$. Not stack-clean.

GEN ZX_unscale_divpow(GEN P, GEN h, long k) given a ZX P, a t_INT h and $k > 0$, returns $P(hx)/h^k$ assuming the result has integral coefficients. Not stack-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 ZX 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 ZX A and B.

GEN ZX_disc(GEN T) returns the discriminant of the ZX T.

GEN ZX_factor(GEN T) returns the factorization of the primitive part of T over $\mathbf{Q}[X]$ (the content is lost).

int ZX_is_squarefree(GEN T) returns 1 if the ZX T is squarefree, 0 otherwise.

long ZX_is_irred(GEN T) returns 1 if T is irreducible, and 0 otherwise.

GEN ZX_squff(GEN T, GEN *E) write the primitive part of the nonzero ZX T as a product $\prod T_i^{e_i}$ with the $e_1 < e_2 < \dots$ all distinct and the T_i primitive and pairwise coprime. Return the vector of the T_i , and set *E to the vector of the e_i , as a t_VECSMALL. For efficiency, powers of x should have been removed from T using ZX_valrem, but the result is also correct if not. Not stack-clean.

GEN ZX_Uspensky(GEN P, GEN ab, long flag, long bitprec) let P be a ZX polynomial whose real roots are simple and bitprec is the relative precision in bits. For efficiency reasons, P should not only have simple real roots but actually be primitive and squarefree, but the routine neither checks nor enforces this, and it returns correct results in this case as well.

- If flag is 0 returns a list of intervals that isolate the real roots of P. The return value is a column of elements which are either vectors [a,b] of rational numbers meaning that there is a single nonrational root in the open interval (a,b) or elements x0 such that x0 is a rational root of P. Beware that the limits of the open intervals can be roots of the polynomial.

- If flag is 1 returns an approximation of the real roots of P.

- If flag is 2 returns the number of roots.

The argument **ab** specify the interval in which the roots are searched. The default interval is $(-\infty, \infty)$. If **ab** is an integer or fraction a then the interval is $[a, \infty)$. If **ab** is a vector $[a, b]$, where **t_INT**, **t_FRAC** or **t_INFINITY** are allowed for a and b , the interval is $[a, b]$.

long ZX_sturm(GEN *P*) number of real roots of the nonconstant squarefree **ZX** P . For efficiency, it is advised to make P primitive first. If P is already known to be irreducible, use **ZX_sturm_irred** for a small speed up.

long ZX_sturmpart(GEN *P*, GEN *ab*) number of real roots of the nonconstant squarefree **ZX** P in the interval specified by *ab*: either **NULL** (no restriction) or a **t_VEC** $[a, b]$ with two real components (of type **t_INT**, **t_FRAC** or **t_INFINITY**). For efficiency, it is advised to make P primitive first.

long ZX_sturm_irred(GEN *P*) number of real roots of the irreducible **ZX** P . For efficiency, it is advised to make P primitive first.

long ZX_realroots(GEN *P*, **long prec**) real roots of the nonconstant squarefree **ZX** P , to precision *prec*. For efficiency, it is advised to make P primitive first. If P is known to be irreducible, **ZX_realroots_irred** will be faster.

long ZX_realroots_irred(GEN *P*, **long prec**) real roots of the irreducible **ZX** P , to precision *prec*. For efficiency, it is advised to make P primitive first.

7.6.2 Resultants.

GEN ZX_ZXY_resultant(GEN *A*, GEN *B*) under the assumption that A in $\mathbf{Z}[Y]$, B in $\mathbf{Q}[Y][X]$, and $R = \text{Res}_Y(A, B) \in \mathbf{Z}[X]$, returns the resultant R .

GEN ZXX_resultant(GEN *A*, GEN *B*) returns the resultant with respect to the main variable.

GEN ZX_composedsum(GEN *P*, GEN *Q*) if $P = a \prod_i (x - p_i)$ and $Q = b \prod_j (x - q_j)$ in some suitable algebraic extension, return $ab \prod_{i,j} (x - (p_i + q_j))$.

GEN ZX_compositum(GEN *A*, GEN *B*, **long *lambda**) given two irreducible **ZX**, returns an irreducible **ZX** C defining their compositum and set *lambda* to a small integer k such that if α is a root of A and β is a root of B , then $k\alpha + \beta$ is a root of C .

GEN ZX_ZXY_rnfequation(GEN *A*, GEN *B*, **long *lambda**), assume A in $\mathbf{Z}[Y]$, B in $\mathbf{Q}[Y][X]$, and $R = \text{Res}_Y(A, B) \in \mathbf{Z}[X]$. If *lambda* = **NULL**, returns R as in **ZY_ZXY_resultant**. Otherwise, *lambda* must point to some integer, e.g. 0 which is used as a seed. The function then finds a small $\lambda \in \mathbf{Z}$ (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_λ .

7.6.3 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.

7.6.4 ZXT.

GEN ZXT_remi2n(GEN *x*, **long n**) applies **ZX_remi2n** to all leaves of the tree *x*.

7.6.5 ZXQ.

GEN ZXQ_mul(GEN x, GEN y, GEN T) returns $x * y \bmod T$, assuming that all inputs are ZXs and that T is monic.

GEN ZXQ_sqr(GEN x, GEN T) returns $x^2 \bmod T$, assuming that all inputs are ZXs and that T is monic.

GEN ZXQ_powu(GEN x, ulong n, GEN T) returns $x^n \bmod T$, assuming that all inputs are ZXs and that T is monic.

GEN ZXQ_powers(GEN x, long n, GEN T) returns $[x^0, \dots, x^n] \bmod T$ as a t_VEC, assuming that all inputs are ZXs and that T is monic.

GEN ZXQ_charpoly(GEN A, GEN T, long v): let T and A be ZXs, returns the characteristic polynomial of $\text{Mod}(A, T)$. More generally, A is allowed to be a QX, hence possibly has rational coefficients, *assuming* the result is a ZX, i.e. the algebraic number $\text{Mod}(A, T)$ is integral over $\mathbb{Z}$.

GEN ZXQ_minpoly(GEN A, GEN T, long d, ulong bound) let T and A be ZXs, returns the minimal polynomial of $\text{Mod}(A, T)$ assuming it has degree d and its coefficients are less than 2^{bound} . More generally, A is allowed to be a QX, hence possibly has rational coefficients, *assuming* the result is a ZX, i.e. the algebraic number $\text{Mod}(A, T)$ is integral over $\mathbb{Z}$.

7.6.6 ZXn.

GEN ZXn_mul(GEN x, GEN y, long n) return $xy \pmod{X^n}$.

GEN ZXn_sqr(GEN x, long n) return $x^2 \pmod{X^n}$.

GEN eta_ZXn(long r, long n) return $\eta(X^r) = \prod_{i>0} (1 - X^{ri}) \pmod{X^n}$, $r > 0$.

GEN eta_product_ZXn(GEN DR, long n): $DR = [D, R]$ being a vector with two t_VECSMALL components, return $\prod_i \eta(X^{d_i})^{r_i}$. Shallow function.

7.6.7 ZXQM.

ZXQM are matrices of ZXQ. All entries must be integers or polynomials of degree strictly less than the degree of T .

GEN ZXQM_mul(GEN x, GEN y, GEN T) returns $x * y \bmod T$, assuming that all inputs are ZXs and that T is monic.

GEN ZXQM_sqr(GEN x, GEN T) returns $x^2 \bmod T$, assuming that all inputs are ZXs and that T is monic.

7.6.8 ZXQX.

GEN ZXQX_mul(GEN x, GEN y, GEN T) returns $x * y$, assuming that all inputs are ZXQXs and that T is monic.

GEN ZXQX_ZXQ_mul(GEN x, GEN y, GEN T) returns $x * y$, assuming that x is a ZXQX, y is a ZXQ and T is monic.

GEN ZXQX_sqr(GEN x, GEN T) returns x^2 , assuming that all inputs are ZXQXs and that T is monic.

GEN ZXQX_gcd(GEN x, GEN y, GEN T) returns the gcd of x and y , assuming that all inputs are ZXQXs and that T is monic.

GEN ZXQXQ_minpoly(GEN A, GEN B, GEN T, long d, ulong bound) let B and A be ZXQXs mod T , returns the minimal polynomial of $\text{Mod}(A, B)$ assuming it has degree d and its coefficients are less than 2^{bound} . More generally, A is allowed to be a QXQX, hence possibly has rational coefficients, *assuming* the result is a ZXQX, i.e. the algebraic number $\text{Mod}(A, B)$ is integral mod T .

7.6.9 ZXX.

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_evalx0(GEN P) returns $P(X, 0)$.

GEN ZXX_Z_mul(GEN x, GEN y) returns xy .

GEN ZXX_shifti(GEN x, long n) shifts all coefficients of x by n bits, which can be negative.

GEN ZXX_Q_mul(GEN x, GEN y) returns $x*y$, where y is a rational number and the resulting t_POL has rational entries.

GEN ZXX_Z_add_shallow(GEN x, GEN y) returns $x + y$. Shallow function.

GEN ZXX_Z_divexact(GEN x, GEN y) returns x/y assuming all integer divisions are exact.

GEN Kronecker_to_ZXX(GEN z, long n, long v) recover $P(X, Y)$ from its Kronecker form $P(X, X^{2n-1})$ (see `RgXX_to_Kronecker`), v is the variable number corresponding to Y. Shallow function.

GEN Kronecker_to_ZXQX(GEN z, GEN T). Let $n = \deg T$ and let $P(X, Y) \in \mathbf{Z}[X, Y]$ lift a polynomial in $K[Y]$, where $K := \mathbf{Z}[X]/(T)$ and $\deg_X P < 2n - 1$ — such as would result from multiplying minimal degree lifts of two polynomials in $K[Y]$. Let $z = P(t, t^{2n-1})$ be a Kronecker form of P (see `RgXX_to_Kronecker`), this function returns $Q \in \mathbf{Z}[X, t]$ such that Q is congruent to $P(X, t)$ mod $(T(X))$, $\deg_X Q < n$. Not stack-clean. Note that t need not be the same variable as Y !

GEN ZXX_mul_Kronecker(GEN P, GEN Q, long n) return ZX_mul applied to the Kronecker forms $P(X, X^{2n-1})$ and $Q(X, X^{2n-1})$ of P and Q . Not stack-clean.

GEN ZXX_sqr_Kronecker(GEN P, long n) return ZX_sqr applied to the Kronecker forms $P(X, X^{2n-1})$ of P . Not stack-clean.

7.6.10 ZXk/QXk. ZXk are multivariate polynomials with t_INT coefficients. QXk are multivariate polynomials with t_FRAC or t_INT coefficients.

GEN ZXk_divexact(GEN x, GEN y) returns x/y assuming the quotient is a ZXk.

GEN ZXk_divides(GEN x, GEN y) returns x/y if the quotient is a ZXk, NULL otherwise.

GEN ZXk_gcd(GEN x, GEN y) returns a gcd of the ZXk x and y .

int Rg_is_QXk(GEN z) return 1 if z is a QXk, 0 otherwise.

GEN QXk_gcd(GEN x, GEN y) returns a gcd of the QXk x and y .

7.6.11 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_mul(GEN x, GEN y)`

`GEN QX_sqr(GEN x)`

`GEN QX_ZX_rem(GEN x, GEN y)` y is assumed to be monic.

`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 T, long prec)` returns the complex roots of the QX T at accuracy `prec`, where real roots are returned as `t_REALs`. More efficient when T is irreducible and primitive. Special case of `cleanroots`.

7.6.12 QXQ.

`GEN QXQ_norm(GEN A, GEN B)` A being a QX and B being a ZX, returns the norm of the algebraic number $A \bmod B$, using a modular algorithm. To ensure that B is a ZX, one may replace it by `Q_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_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 $A \bmod B$, *assuming* that the result is an integer, which is for instance the case is $A \bmod B$ is an algebraic integer, in particular if A is a ZX. To ensure that B is a ZX, one may replace it by `Q_primpart(B)` (which of course does not change the norm).

If the result is not known to be an integer, you must use `QXQ_norm` instead, which is slower.

`GEN QXQ_mul(GEN A, GEN B, GEN T)` returns the product of A and B modulo T where both A and B are a QX and T is a monic ZX.

`GEN QXQ_sqr(GEN A, GEN T)` returns the square of A modulo T where A is a QX and T is a monic ZX.

`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_inv(A, Q_primpart(B));`

But in all cases where modular arithmetic modulo B is desired, it is much more efficient to replace B by `Q_primpart(B)` once and for all.

`GEN QXQ_div(GEN A, GEN B, GEN T)` returns A/B modulo T where A and B are QX and T is a ZX. Use this function when the result is expected to be of the same size as $B^{-1} \bmod T$ or smaller. Otherwise, it will be faster to use `QXQ_mul(A, QXQ_inv(B, T), T)`.

`GEN QXQ_charpoly(GEN A, GEN T, long v)` where A is a QX and T is a ZX, returns the characteristic polynomial of $\text{Mod}(A, T)$. If the result is known to be a ZX, then calling `ZXQ_charpoly` will be faster.

GEN QXQ_powers(GEN x, long n, GEN T) returns $[x^0, \dots, x^n]$ as RgXQ_powers would, but in a more efficient way when x has a huge integer denominator (we start by removing that denominator). Assume that x is a QX and T is a ZX. Meant to precompute powers of algebraic integers in $\mathbf{Q}[t]/(T)$.

GEN QXQ_reverse(GEN f, GEN T) as RgXQ_reverse, assuming f is a QX and T is a ZX.

GEN QX_ZXQV_eval(GEN f, GEN nV, GEN dV) as 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 stack-clean, but suitable for gc_upto.

GEN QXV_QXQ_eval(GEN v, GEN a, GEN T) v is a vector of QXs (possibly scalars, i.e. rational numbers, for convenience), a and T both QX. Return the vector of evaluations at a modulo T . Not stack-clean.

GEN QXY_QXQ_evalx(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 $P(a \bmod T, Y)$. Not stack-clean.

7.6.13 QXQX.

GEN QXQX_mul(GEN x, GEN y, GEN T) where T is a monic ZX.

GEN QXQX_QXQ_mul(GEN x, GEN y, GEN T) where T is a monic ZX.

GEN QXQX_sqr(GEN x, GEN T) where T is a monic ZX

GEN QXQX_powers(GEN x, long n, GEN T) where T is a monic ZX

GEN nfgcd(GEN P, GEN Q, GEN T, GEN den) given P and Q in $\mathbf{Z}[X, Y]$, T monic irreducible in $\mathbf{Z}[Y]$, returns the primitive d in $\mathbf{Z}[X, Y]$ which is a gcd of P, Q in $K[X]$, where K is the number field $\mathbf{Q}[Y]/(T)$. If not NULL, 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 Pnew is not NULL, set *Pnew to a nonzero integer multiple of P/d . If P and Q are both monic, then d is monic and *Pnew is exactly P/d . Not stack-clean if the gcd is 1 (in that case *Pnew is set to P).

GEN QXQX_gcd(GEN x, GEN y, GEN T) returns the gcd of x and y , assuming that x and y are QXQXs and that T is a monic ZX.

GEN QXQX_homogenous_evalpow(GEN P, GEN a, GEN B, GEN T) Evaluate the homogenous polynomial associated to the univariate polynomial P on (a, b) where B is the vector of powers of b with exponents 0 to the degree of P (QXQ_powers(b, degpol(P), T)).

7.6.14 QXQM.

QXQM are matrices of QXQ. All entries must be t_INT, t_FRAC or polynomials of degree strictly less than the degree of T , which must be a monic ZX.

GEN QXQM_mul(GEN x, GEN y, GEN T) returns $x * y \bmod T$.

GEN QXQM_sqr(GEN x, GEN T) returns $x^2 \bmod T$.

7.6.15 zx.

GEN zero_zx(long sv) returns a zero **zx** in variable v .

GEN polx_zx(long sv) returns the variable v as degree 1 **Flx**.

GEN zx_renormalize(GEN x, long l), as **Flx_renormalize**, where $l = \lg(x)$, in place.

GEN zx_shift(GEN T, long n) return T multiplied by x^n , assuming $n \geq 0$.

long zx_lval(GEN f, long p) return the valuation of f at p .

GEN zx_z_divexact(GEN x, long y) return x/y assuming all divisions are exact.

7.6.16 RgX.

7.6.16.1 Tests.

long RgX_degree(GEN x, long v) x being a **t_POL** and $v \geq 0$, returns the degree in v of x . Error if x is not a polynomial in v .

int RgX_isscalar(GEN x) return 1 if all the coefficients of x of degree > 0 are 0 (as per **gequal0**).

int RgX_is_rational(GEN P) return 1 if the **RgX** P has only rational coefficients (**t_INT** and **t_FRAC**), and 0 otherwise.

int RgX_is_QX(GEN P) return 1 if the **RgX** P has only **t_INT** and **t_FRAC** coefficients, and 0 otherwise.

int RgX_is_ZX(GEN P) return 1 if the **RgX** P has only **t_INT** coefficients, and 0 otherwise.

int RgX_is_ZXX(GEN P, long *v) return 1 if the **RgX** P is a bivariate polynomial with **t_INT** coefficients. If $*v \geq 0$, also check that the secondary variable is equal to v , otherwise set $*v$ to the secondary variable, if any.

int RgX_is_monomial(GEN x) returns 1 (true) if x is a nonzero 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.

7.6.16.2 Coefficients, blocks.

GEN RgX_coeff(GEN P, long n) return the coefficient of x^n in P , defined as **gen_0** if $n < 0$ or $n > \text{degpol}(P)$. Shallow function.

int RgX_blocks(GEN P, long n, long m) writes $P(X) = a_0(X) + X^n * a_1(X) * X^n + \dots + X^{n*(m-1)} a_{m-1}(X)$, where the a_i are polynomial of degree at most $n - 1$ (except possibly for the last one) and returns $[a_0(X), a_1(X), \dots, a_{m-1}(X)]$. Shallow function.

void RgX_even_odd(GEN p, GEN *pe, GEN *po) write $p(X) = E(X^2) + XO(X^2)$ and set $*pe = E$, $*po = O$. Shallow function.

GEN RgX_splitting(GEN P, long k) write $P(X) = a_0(X^k) + X a_1(X^k) + \dots + X^{k-1} a_{k-1}(X^k)$ and return $[a_0(X), a_1(X), \dots, a_{k-1}(X)]$. Shallow function.

GEN `RgX_copy`(GEN `x`) returns (a deep copy of) `x`.

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 $lx \leq \lg(x)$.

GEN `RgX_recip`(GEN `P`) returns the reverse of the polynomial P , i.e. $X^{\deg P} P(1/X)$.

GEN `RgX_recip_shallow`(GEN `P`) shallow function of `RgX_recip`.

GEN `RgX_recip_i`(GEN `P`) shallow function of `RgX_recip`, where we further assume that $P(0) \neq 0$, so that the degree of the output is the degree of P .

long `rfracrecip`(GEN `*a`, GEN `*b`) let `*a` and `*b` be such that their quotient F is a `t_RFRAC` in variable X . Write $F(1/X) = X^v A/B$ where A and B are coprime to X and v in $\mathbf{Z}$. Set `*a` to A , `*b` to B and return v .

GEN `RgX_deflate`(GEN `P`, long `d`) assuming P is a polynomial of the form $Q(X^d)$, return Q . Shallow function, may return P itself.

long `RgX_deflate_order`(GEN `P`) given a nonconstant polynomial P , returns the largest exponent d such that P is of the form $P(x^d)$ (use `gequal0` to check whether coefficients are 0).

long `RgX_deflate_max`(GEN `P`, long `*d`) given a nonconstant polynomial P , sets `d` to `RgX_deflate_order(P)` and returns `RgX_deflate(P,d)`. Shallow function, may return P itself.

long `rfrac_deflate_order`(GEN `F`) as `RgX_deflate_order` where F is a nonconstant `t_RFRAC`.

long `rfrac_deflate_max`(GEN `F`, long `*d`) as `RgX_deflate_max` where F is a nonconstant `t_RFRAC`.

GEN `rfrac_deflate`(GEN `F`, long `m`) as `RgX_deflate` where F is a `t_RFRAC`.

GEN `RgX_inflate`(GEN `P`, long `d`) return $P(X^d)$. Shallow function.

GEN `RgX_rescale_to_int`(GEN `x`) given a polynomial x with real entries (`t_INT`, `t_FRAC` or `t_REAL`), return a `ZX` which is very close to Dx for some well-chosen integer D . More precisely, if the input is exact, D is the denominator of x ; else it is a power of 2 chosen so that all inexact entries are correctly rounded to 1 ulp.

GEN `RgX_Graeffe`(GEN `P`, long `d`) as the GP function `polgraeffe`.

GEN `RgX_homogenize`(GEN `P`, long `v`) Return the homogenous polynomial associated to P in the secondary variable v , that is $y^d * P(x/y)$ where d is the degree of P , x is the variable of P , and y is the variable with number v .

GEN `RgX_homogenize_deg`(GEN `P`, long `d`, long `v`) Return the homogenous polynomial of degree d associated to P in the secondary variable v , that is $y^d * P(x/y)$ x is the variable of P , and y is the variable with number v .

GEN `RgXX_to_Kronecker`(GEN `P`, long `n`) Assuming $P(X, Y)$ is a bivariate polynomial of degree in X strictly less than n , returns $P(X, X^{2*n-1})$, the Kronecker form of P . Shallow function. The polynomial P is allowed to be constant in X , whether it has been simplified or not (whether X^0 appears or not in terms), but one should prefer `RgX_inflate` in this case.

On the other hand, use `RgXX_to_Kronecker_var` instead if P involves 3 variables or more: in this case, there is an ambiguity as to what X denotes and the function does not inspect all

coefficients to determine it. (Note that an inspection would only work if we could guarantee that the intended X variable appears in at least one of the terms, which is brittle and breaks if `simplify` makes it disappear. Don't take the risk.)

`GEN RgXX_to_Kronecker_var(GEN P, long n, long vx)` Safe variant of `RgXX_to_Kronecker` where `vx` is the variable number attached to X . It is assumed that its priority is less than the priority of Y , and that all other variables involved in P have lower priority than these two. Shallow function.

`GEN RgXX_to_Kronecker_spec(GEN Q, long lQ, long n)` return `RgXX_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.

7.6.16.3 Shifts, valuations.

`GEN RgX_shift(GEN x, long n)` returns $x * t^n$ if $n \geq 0$, and $x \backslash t^{-n}$ 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 $P * X^k \pmod{X^p - 1}$, assuming the degree of P is strictly less than p , and $k \geq 0$.

`void RgX_shift_inplace_init(long v)` $v \geq 0$, prepare for a later call to `RgX_shift_inplace`. Reserves v words on the stack.

`GEN RgX_shift_inplace(GEN x, long v)` $v \geq 0$, 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 = gc_upto(av, ...); /* a t_POL */
return RgX_shift_inplace(x, v);
```

`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 `isexactzero`. 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 `isexactzero`.

`long RgX_valrem_inexact(GEN P, GEN *z)` as `RgX_valrem`, using `gequal0` instead of `isexactzero`.

`long RgXV_maxdegree(GEN V)` returns the maximum of the degrees of the components of the vector of `t_POLs` V .

7.6.16.4 Basic arithmetic.

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 Rg_RgX_sub(GEN x, GEN y)

GEN RgX_Rg_sub(GEN y, GEN x) returns $x - y$

GEN RgX_Rg_mul(GEN y, GEN x) multiplies the RgX y by the scalar x .

GEN RgX_muls(GEN y, long s) multiplies the RgX y by the long s .

GEN RgX_mul2n(GEN y, long n) multiplies the RgX y by 2^n .

GEN RgX_Rg_div(GEN y, GEN x) divides the RgX y by the scalar x .

GEN RgX_divs(GEN y, long s) divides the RgX y by the long s .

GEN RgX_Rg_divexact(GEN x, GEN y) exact division of the RgX y by the scalar x .

GEN RgX_Rg_eval_bk(GEN f, GEN x) returns $f(x)$ using Brent and Kung algorithm. (Use `poleval` for Horner algorithm.)

GEN RgX_RgV_eval(GEN f, GEN V) as `RgX_Rg_eval_bk(f, x)`, assuming V was output by `gpowers(x, n)` for some $n \geq 1$.

GEN RgXV_RgV_eval(GEN f, GEN V) apply `RgX_RgV_eval_bk(, V)` to all the components of the vector f .

GEN RgX_normalize(GEN x) divides x by its leading coefficient. If the latter is 1, x itself is returned, not a copy. Leading coefficients equal to 0 are stripped, e.g.

$$0.*t^3 + \text{Mod}(0,3)*t^2 + 2*t$$

is normalized to t .

GEN RgX_mul(GEN x, GEN y) multiplies the two `t_POL` (in the same variable) x and y . Detect the coefficient ring and use an appropriate algorithm.

GEN RgX_mul_i(GEN x, GEN y) multiplies the two `t_POL` (in the same variable) x and y . Do not detect the coefficient ring. Use a generic 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_sqr(GEN x) squares the `t_POL` x . Detect the coefficient ring and use an appropriate algorithm.

GEN RgX_sqr_i(GEN x) squares the `t_POL` x . Do not detect the coefficient ring. Use a generic Karatsuba algorithm.

GEN RgXV_prod(GEN V), V being a vector of RgX, returns their product.

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.

In the generic case, the remainder is created after the quotient and can be disposed of individually with a `cgiv(r)`.

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 $\text{lc}(y)^{\deg(x)-\deg(y)+1}x = qy + r$. 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 .

GEN RgXQX_pseudorem(GEN x, GEN y, GEN T) return the remainder in the pseudo-division of x by y over $R[X]/(T)$.

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 $\text{lc}(y)^{\deg(x)-\deg(y)+1}x = qy + r$ in $R[X]/(T)$. Return q and set *ptr to r .

GEN RgX_mulXn(GEN a, long n) returns $a * X^n$. This may be a `t_FRAC` if $n < 0$ and the valuation of a is not large enough.

GEN RgX_addmulXn(GEN a, GEN b, long n) returns $a + b * X^n$, assuming that $n > 0$.

GEN RgX_addmulXn_shallow(GEN a, GEN b, long n) shallow variant of RgX_addmulXn.

GEN RgX_digits(GEN x, GEN B) returns a vector of RgX $[c_0, \dots, c_n]$ of degree less than the degree of B and such that $x = \sum_{i=0}^n c_i B^i$.

7.6.16.5 Internal routines working on coefficient arrays.

These routines operate on coefficient blocks which are invalid GENs A GEN argument a or b in routines below is actually a coefficient arrays representing the polynomials $\sum_{i=0}^{na-1} a[i]X^i$ and $\sum_{i=0}^{nb-1} b[i]X^i$. Note that $a[0]$ and $b[0]$ contain coefficients and not the mandatory GEN codeword. This allows to implement divide-and-conquer methods directly, without needing to allocate wrappers around coefficient blocks.

GEN RgX_mulspec(GEN a, GEN b, long na, long nb). Internal routine: given two coefficient arrays representing polynomials, return their product (as a true GEN) in variable 0.

GEN RgX_sqrspec(GEN a, long na). Internal routine: given a coefficient array representing a polynomial r return its square (as a true GEN) in variable 0.

GEN RgX_addspec(GEN x, GEN y, long nx, long ny) given two coefficient arrays representing polynomials, return their sum (as a true GEN) in variable 0.

GEN RgX_addspec_shallow(GEN x, GEN y, long nx, long ny) shallow variant of RgX_addspec.

7.6.16.6 GCD, Resultant.

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_halfgcd(GEN x, GEN y) assuming x and y are t_POLs in the same variable, returns a 2×2 t_MAT M with t_POL entries, such that $M * [x, y] == [a, b]$ such that $\deg a \geq \lceil \max(\deg x, \deg y) / 2 \rceil > \deg b$.

GEN RgX_halfgcd_all(GEN x, GEN y, GEN *pt_a, GEN *pt_b) as RgX_halfgcd, in addition, if pt_a (resp. pt_b) is not NULL, *pt_a (resp. *pt_b) is set to a (resp. b).

GEN RgX_chinese_coprime(GEN x, GEN y, GEN Tx, GEN Ty, GEN Tz) returns an RgX, congruent to x mod Tx and to y mod Ty. Assumes Tx and Ty are coprime, and Tz = Tx * Ty or NULL (in which case it is computed within).

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 resultant(x,y). If sol is not NULL, sets it to the last nonconstant remainder in the polynomial remainder sequence if it exists and to gen_0 otherwise (e.g. one polynomial has degree 0).

7.6.16.7 Other operations.

GEN RgX_gtofp(GEN x, GEN prec) returns the polynomial obtained by applying

gtofp(gel(x,i), prec)

to all coefficients of x.

GEN RgX_fpnorml2(GEN x, long prec) returns (a stack-clean variant of)

gnorml2(RgX_gtofp(x, prec))

GEN RgX_deriv(GEN x) returns the derivative of x with respect to its main variable.

GEN RgX_derivn(GEN x, long n) returns the nth 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_rescale(GEN P, GEN h) returns $h^{\deg(P)} P(x/h)$. P is an RgX and h is nonzero. (Leaves small objects on the stack. Suitable but inefficient for gc_upto.)

GEN RgXV_rescale(GEN v, GEN h) apply RgX_unscale to a vector of RgX.

GEN RgX_unscale(GEN P, GEN h) returns $P(hx)$. (Leaves small objects on the stack. Suitable but inefficient for gc_upto.)

GEN RgXV_unscale(GEN v, GEN h) apply RgX_unscale to a vector of RgX.

GEN RgX_Rg_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 $c = \pm 1$).

GEN RgX_affine(GEN P, GEN a, GEN b) Return $P(aX + b)$ (optimized for $b = \pm 1$). Not stack-clean.

7.6.16.8 Implicit bivariate homogeneous polynomials.

A bivariate homogeneous polynomial $P(X, Y)$ of (homogeneous) degree d is represented by giving $P(X, 1)$ and d .

`GEN RgX_homogenous_derivn(GEN P, long n, long d)` returns the bivariate homogeneous polynomial of homogeneous degree $d-n$ corresponding to the n -th derivative with respect to the implicit variables of the homogeneous polynomial P of degree d .

`GEN RgX_homogenous_eval(GEN P, GEN a, GEN b, long d)` Evaluate the homogeneous polynomial of degree d associated to the univariate polynomial P on (a, b) .

`GEN RgX_homogenous_evalpow(GEN P, GEN a, GEN B, long d)` Evaluate the homogeneous polynomial of degree d associated to the univariate polynomial P on (a, b) where B is the vector of powers of b with exponents 0 to d (`gpowers(b, d)`).

7.6.16.9 Function related to modular forms.

`GEN RgX_act_Gl2Q(GEN g, long k)` let R be a commutative ring and $g = [a, b; c, d]$ be in $\text{GL}_2(\mathbf{Q})$, g acts (on the left) on homogeneous polynomials of degree $k-2$ in $V := R[X, Y]_{k-2}$ via

$$g \cdot P := P(dX - cY, -bX + aY) = (\det g)^{k-2} P((X, Y) \cdot g^{-1}).$$

This function returns the matrix in $M_{k-1}(R)$ of $P \mapsto g \cdot P$ in the basis $(X^{k-2}, \dots, Y^{k-2})$ of V .

`GEN RgX_act_ZGl2Q(GEN z, long k)` let $G := \text{GL}_2(\mathbf{Q})$, acting on $R[X, Y]_{k-2}$ and $z \in \mathbf{Z}[G]$. Return the matrix giving $P \mapsto z \cdot P$ in the basis $(X^{k-2}, \dots, Y^{k-2})$.

7.6.17 RgXn.

`GEN RgXn_red_shallow(GEN x, long n)` return $x \% t^n$, where $n \geq 0$. Shallow function.

`GEN RgXn_recip_shallow(GEN P)` returns $X^n P(1/X)$. Shallow function.

`GEN RgXn_mul(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 RgXn_sqr(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_mulhigh_i(GEN f, GEN g, long n)` return the Euclidean quotient of $f(x) * g(x)$ by x^n (high product). Uses `RgXn_mul` applied to the reciprocal polynomials of f and g . Shallow function.

`GEN RgX_sqrhigh_i(GEN f, long n)` return the Euclidean quotient of $f(x)^2$ by x^n (high product). Uses `RgXn_sqr` applied to the reciprocal polynomial of f . Shallow function.

`GEN RgXn_inv(GEN a, long n)` returns a^{-1} modulo X^n , where a is a `t_POL` in the variable X and $n \geq 0$. Uses Newton-Raphson algorithm.

`GEN RgXn_inv_i(GEN a, long n)` as `RgXn_inv` without final garbage collection (suitable for `gc_upto`).

`GEN RgXn_div(GEN a, GEN b, long n)` returns a/b modulo X^n , where a and b are `t_POLs` in the variable X and $n \geq 0$. Uses Newton-Raphson/Karp-Markstein algorithm.

`GEN RgXn_div_i(GEN a, GEN b, long n)` as `RgXn_div` without final garbage collection (suitable for `gc_upto`).

GEN RgXn_powers(GEN x, long m, long n) returns $[x^0, \dots, x^m]$ modulo X^n as a t_VEC of RgXns.

GEN RgXn_powu(GEN x, ulong m, long n) returns x^m modulo X^n .

GEN RgXn_powu_i(GEN x, ulong m, long n) as RgXn_powu, not stack-clean.

GEN RgXn_sqrt(GEN a, long n) returns $a^{1/2}$ modulo X^n , where a is a t_POL in the variable X and $n \geq 0$. Assume that $a = 1 \bmod X$. Uses Newton algorithm.

GEN RgXn_exp(GEN a, long n) returns $\exp(a)$ modulo X^n , assuming $a = 0 \bmod X$.

GEN RgXn_expint(GEN f, long n) return $\exp(F)$ where F is the primitive of f that vanishes at 0.

GEN RgXn_eval(GEN Q, GEN x, long n) special case of RgX_RgXQ_eval, when the modulus is a monomial: returns $Q(x)$ modulo t^n , where $x \in R[t]$.

GEN RgX_RgXn_eval(GEN f, GEN x, long n) returns $f(x)$ modulo X^n .

GEN RgX_RgXnV_eval(GEN f, GEN V, long n) as RgX_RgXn_eval(f, x, n), assuming V was output by RgXn_powers(x, m, n) for some $m \geq 1$.

GEN RgXn_reverse(GEN f, long n) assuming that $f = ax \bmod x^2$ with a invertible, returns a t_POL g of degree $< n$ such that $(g \circ f)(x) = x \bmod x^n$.

7.6.18 RgXnV.

GEN RgXnV_red_shallow(GEN x, long n) apply RgXn_red_shallow to all the components of the vector x .

7.6.19 RgXQ.

GEN RgXQ_mul(GEN y, GEN x, GEN T) computes $xy \bmod T$

GEN RgXQ_sqr(GEN x, GEN T) computes $x^2 \bmod T$

GEN RgXQ_inv(GEN x, GEN T) return the inverse of $x \bmod T$.

GEN RgXQ_pow(GEN x, GEN n, GEN T) computes $x^n \bmod T$

GEN RgXQ_powu(GEN x, ulong n, GEN T) computes $x^n \bmod T$, n being an ulong.

GEN RgXQ_powers(GEN x, long n, GEN T) returns $[x^0, \dots, x^n]$ as a t_VEC of RgXQs.

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 $\text{Mod}(x, T)$.

GEN RgXQ_trace(GEN x, GEN T) returns the trace of $\text{Mod}(x, T)$.

GEN RgXQ_charpoly(GEN x, GEN T, long v) returns the characteristic polynomial of $\text{Mod}(x, T)$, in variable v .

GEN RgXQ_minpoly(GEN x, GEN T, long v) returns the minimal polynomial of $\text{Mod}(x, T)$, 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 $n \geq 1$.

`int RgXQ_ratlift(GEN x, GEN T, long amax, long bmax, GEN *P, GEN *Q)` Assuming that $\text{amax} + \text{bmax} < \deg T$, attempts to recognize x as a rational function a/b , i.e. to find $\mathbf{t_POLs}$ P and Q such that

- $P \equiv Qx$ modulo 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 $\mathbf{t_POL}$ g of degree $< n = \deg T$ 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 `[modreverse(f,T)]`; faster than the high-level function since it needs not compute the characteristic polynomial of $f \bmod T$ (often already known in applications). In the trivial case where $n \leq 1$, returns a scalar, not a constant $\mathbf{t_POL}$.

7.6.20 RgXQV, RgXQC.

`GEN RgXQC_red(GEN z, GEN T)` z a vector whose coefficients are $\mathbf{RgXs}$ (arbitrary $\mathbf{GENs}$ in fact), reduce them to $\mathbf{RgXQs}$ (applying `gre`m coefficientwise) in a $\mathbf{t_COL}$.

`GEN RgXQV_red(GEN z, GEN T)` z a vector whose coefficients are $\mathbf{RgXs}$ (arbitrary $\mathbf{GENs}$ in fact), reduce them to $\mathbf{RgXQs}$ (applying `gre`m coefficientwise) in a $\mathbf{t_VEC}$.

`GEN RgXQV_RgXQ_mul(GEN z, GEN x, GEN T)` z multiplies the $\mathbf{RgXQV}$ z by the scalar ($\mathbf{RgXQ}$) x .

`GEN RgXQV_factorback(GEN L, GEN e, GEN T)` returns $\prod_i L_i^{e_i} \bmod T$ where L is a vector of $\mathbf{RgXQs}$ and e a vector of $\mathbf{t_INTs}$.

7.6.21 RgXQM.

`GEN RgXQM_red(GEN z, GEN T)` z a matrix whose coefficients are $\mathbf{RgXs}$ (arbitrary $\mathbf{GENs}$ in fact), reduce them to $\mathbf{RgXQs}$ (applying `gre`m coefficientwise).

`GEN RgXQM_mul(GEN x, GEN y, GEN T)`

7.6.22 RgXQX.

`GEN RgXQX_red(GEN z, GEN T)` z a $\mathbf{t_POL}$ whose coefficients are $\mathbf{RgXs}$ (arbitrary $\mathbf{GENs}$ in fact), reduce them to $\mathbf{RgXQs}$ (applying `gre`m coefficientwise).

`GEN RgXQX_mul(GEN x, GEN y, GEN T)`

`GEN RgXQX_RgXQ_mul(GEN x, GEN y, GEN T)` multiplies the $\mathbf{RgXQX}$ y by the scalar ($\mathbf{RgXQ}$) x .

`GEN RgXQX_sqr(GEN x, GEN T)`

`GEN RgXQX_powers(GEN x, long n, GEN T)`

`GEN RgXQX_divrem(GEN x, GEN y, GEN T, GEN *pr)`

`GEN RgXQX_div(GEN x, GEN y, GEN T)`

`GEN RgXQX_rem(GEN x, GEN y, GEN T)`

`GEN RgXQX_RgXQ_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 $P(X + \text{Mod}(c(Y), T(Y)))$.

GEN Kronecker_to_mod(GEN *z*, GEN *T*) $z \in R[X]$ represents an element $P(X, Y)$ in $R[X, Y] \bmod T(Y)$ in Kronecker form, i.e. $z = P(X, X^{2*n-1})$

Let R be some commutative ring, $n = \deg T$ and let $P(X, Y) \in R[X, Y]$ 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 $z = P(t, t^{2*n-1})$ 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:

Black box algebraic structures

The generic routines like `gmul` or `gadd` allow handling objects belonging to a fixed list of basic types, with some natural polymorphism (you can mix rational numbers and polynomials, etc.), at the expense of efficiency and sometimes of clarity when the recursive structure becomes complicated, e.g. a few levels of `t_POLMODs` attached to different polynomials and variable numbers for quotient structures. This is the only possibility in GP.

On the other hand, the Level 2 Kernel allows dedicated routines to handle efficiently objects of a very specific type, e.g. polynomials with coefficients in the same finite field. This is more efficient, but involves a lot of code duplication since polymorphism is no longer possible.

A third and final option, still restricted to library programming, is to define an arbitrary algebraic structure (currently groups, fields, rings, algebras and $\mathbf{Z}_p$ -modules) by providing suitable methods, then using generic algorithms. For instance naive Gaussian pivoting applies over all base fields and need only be implemented once. The difference with the first solution is that we no longer depend on the way functions like `gmul` or `gadd` will guess what the user is trying to do. We can then implement independently various groups / fields / algebras in a clean way.

8.1 Black box groups.

A black box group is defined by a `bb_group` struct, describing methods available to handle group elements:

```
struct bb_group
{
    GEN (*mul)(void*, GEN, GEN);
    GEN (*pow)(void*, GEN, GEN);
    GEN (*rand)(void*);
    ulong (*hash)(GEN);
    int (*equal)(GEN, GEN);
    int (*equal1)(GEN);
    GEN (*easylog)(void *E, GEN, GEN, GEN);
};
```

`mul(E,x,y)` returns the product xy .

`pow(E,x,n)` returns x^n (n integer, possibly negative or zero).

`rand(E)` returns a random element in the group.

`hash(x)` returns a hash value for x (`hash_GEN` is suitable for this field).

`equal(x,y)` returns one if $x = y$ and zero otherwise.

`equal1(x)` returns one if x is the neutral element in the group, and zero otherwise.

`easylog(E,a,g,o)` (optional) returns either NULL or the discrete logarithm n such that $g^n = a$, the element g being of order o . This provides a short-cut in situation where a better algorithm than the generic one is known.

A group is thus described by a `struct bb_group` as above and auxiliary data typecast to `void*`. The following functions operate on black box groups:

`GEN gen_Shanks_log(GEN x, GEN g, GEN N, void *E, const struct bb_group *grp)`
Generic baby-step/giant-step algorithm (Shanks's method). Assuming that g has order N , compute an integer k such that $g^k = x$. Return `cgetg(1, t_VEC)` if there are no solutions. This requires $O(\sqrt{N})$ group operations and uses an auxiliary table containing $O(\sqrt{N})$ group elements.

The above is useful for a one-shot computation. If many discrete logs are desired: `GEN gen_Shanks_init(GEN g, long n, void *E, const struct bb_group *grp)` return an auxiliary data structure T required to compute a discrete log in base g . Compute and store all powers g^i , $i < n$.

`GEN gen_Shanks(GEN T, GEN x, ulong N, void *E, const struct bb_group *grp)` Let T be computed by `gen_Shanks_init(g,n,...)`. Return $k < nN$ such that $g^k = x$ or NULL if no such index exist. It uses $O(N)$ operation in the group and fast table lookups (in time $O(\log n)$). The interface is such that the function may be used when the order of the base g is unknown, and hence compute it given only an upper bound B for it: e.g. choose n, N such that $nN \geq B$ and compute the discrete log l of g^{-1} in base g , then use `gen_order` with multiple $N = l + 1$.

`GEN gen_Pollard_log(GEN x, GEN g, GEN N, void *E, const struct bb_group *grp)`
Generic Pollard rho algorithm. Assuming that g has order N , compute an integer k such that $g^k = x$. This requires $O(\sqrt{N})$ group operations in average and $O(1)$ storage. Will enter an infinite loop if there are no solutions.

`GEN gen_plog(GEN x, GEN g, GEN N, void *E, const struct bb_group)` Assuming that g has prime order N , compute an integer k such that $g^k = x$, using either `gen_Shanks_log` or `gen_Pollard_log`. Return `cgetg(1, t_VEC)` if there are no solutions.

`GEN gen_Shanks_sqrtn(GEN a, GEN n, GEN N, GEN *zetan, void *E, const struct bb_group *grp)` returns one solution of $x^n = a$ in a black box cyclic group of order N . Return NULL if no solution exists. If `zetan` is not NULL it is set to an element of exact order n . This function uses `gen_plog` for all prime divisors of $\gcd(n, N)$.

`GEN gen_PH_log(GEN a, GEN g, GEN N, void *E, const struct bb_group *grp)` returns an integer k such that $g^k = x$, assuming that the order of g divides N , using Pohlig-Hellman algorithm. Return `cgetg(1, t_VEC)` if there are no solutions. This calls `gen_plog` repeatedly for all prime divisors p of N .

In the following functions the integer parameter `ord` can be given in all the formats recognized for the argument of arithmetic functions, i.e. either as a positive `t_INT` N , or as its factorization matrix faN , or (preferred) as a pair $[N, faN]$.

`GEN gen_order(GEN x, GEN ord, void *E, const struct bb_group *grp)` computes the order of x ; `ord` is a multiple of the order, for instance the group order.

GEN `gen_factored_order`(GEN `x`, GEN `ord`, void `*E`, const struct `bb_group *grp`) returns a pair $[o, F]$, where o is the order of x and F is the factorization of o ; `ord` is as in `gen_order`.

GEN `gen_gener`(GEN `ord`, void `*E`, const struct `bb_group *grp`) returns a random generator of the group, assuming it is of order exactly `ord`.

GEN `get_arith_Z`(GEN `ord`) given `ord` as above in one of the formats recognized for arithmetic functions, i.e. a positive t_INT N , its factorization faN , or the pair $[N, faN]$, return N .

GEN `get_arith_ZZM`(GEN `ord`) given `ord` as above, return the pair $[N, faN]$. This may require factoring N .

GEN `gen_select_order`(GEN `v`, void `*E`, const struct `bb_group *grp`) Let v be a vector of possible orders for the group; try to find the true order by checking orders of random points. This will not terminate if there is an ambiguity.

8.1.1 Black box groups with pairing.

These functions handle groups of rank at most 2 equipped with a family of bilinear pairings which behave like the Weil pairing on elliptic curves over finite field. In the descriptions below, the function `pairorder`(`E`, `P`, `Q`, `m`, `F`) must return the order of the m -pairing of P and Q , both of order dividing m , where F is the factorization matrix of a multiple M of m . The format $[M, \text{factor}(M)]$ is also accepted for F , and in fact preferred.

GEN `gen_ellgroup`(GEN `o`, GEN `d`, GEN `*pt_m`, void `*E`, const struct `bb_group *grp`, GEN `pairorder`(void `*E`, GEN `P`, GEN `Q`, GEN `m`, GEN `F`)) returns the elementary divisors $[d_1, d_2]$ of the group, assuming it is of order exactly $o > 1$, and that d_2 divides d . If $d_2 = 1$ then $[o]$ is returned, otherwise `m=*pt_m` is set to the order of the pairing required to verify a generating set which is to be used with `gen_ellgens`. For the parameter o , all formats recognized by arithmetic functions are allowed, preferably a factorization matrix or a pair $[n, \text{factor}(n)]$.

GEN `gen_ellgens`(GEN `d1`, GEN `d2`, GEN `m`, void `*E`, const struct `bb_group *grp`, GEN `pairorder`(void `*E`, GEN `P`, GEN `Q`, GEN `m`, GEN `F`)) the parameters d_1, d_2, m being as returned by `gen_ellgroup`, returns a pair of generators $[P, Q]$ such that P is of order d_1 and the m -pairing of P and Q is of order m . (Note: Q needs not be of order d_2). For the parameter d_1 , all formats recognized by arithmetic functions are allowed, preferably a factorization matrix or a pair $[n, \text{factor}(n)]$.

8.1.2 Functions returning black box groups.

const struct `bb_group *` `get_Flxq_star`(void `**E`, GEN `T`, ulong `p`)

const struct `bb_group *` `get_FpXQ_star`(void `**E`, GEN `T`, GEN `p`) returns a pointer to the black box group $(\mathbf{F}_p[x]/(T))^*$.

const struct `bb_group *` `get_FpE_group`(void `**pE`, GEN `a4`, GEN `a6`, GEN `p`) returns a pointer to a black box group and set `*pE` to the necessary data for computing in the group $E(\mathbf{F}_p)$ where E is the elliptic curve $E : y^2 = x^3 + a_4x + a_6$, with a_4 and a_6 in $\mathbf{F}_p$.

const struct `bb_group *` `get_FpXQE_group`(void `**pE`, GEN `a4`, GEN `a6`, GEN `T`, GEN `p`) returns a pointer to a black box group and set `*pE` to the necessary data for computing in the group $E(\mathbf{F}_p[X]/(T))$ where E is the elliptic curve $E : y^2 = x^3 + a_4x + a_6$, with a_4 and a_6 in $\mathbf{F}_p[X]/(T)$.

const struct `bb_group *` `get_FlxqE_group`(void `**pE`, GEN `a4`, GEN `a6`, GEN `T`, ulong `p`) idem for small p .

`const struct bb_group * get_F2xqE_group(void **pE, GEN a2, GEN a6, GEN T)` idem for $p = 2$.

8.2 Black box fields.

A black box field is defined by a `bb_field` struct, describing methods available to handle field elements:

```
struct bb_field
{
    GEN (*red)(void *E ,GEN);
    GEN (*add)(void *E ,GEN, GEN);
    GEN (*mul)(void *E ,GEN, GEN);
    GEN (*neg)(void *E ,GEN);
    GEN (*inv)(void *E ,GEN);
    int (*equal0)(GEN);
    GEN (*s)(void *E, long);
};
```

In contrast of black box group, elements can have non canonical forms, and only `red` is required to return a canonical form. For instance a black box implementation of finite fields, all methods except `red` may return arbitrary representatives in $\mathbf{Z}[X]$ of the correct congruence class modulo $(p, T(X))$.

`red(E,x)` returns the canonical form of x .

`add(E,x,y)` returns the sum $x + y$.

`mul(E,x,y)` returns the product xy .

`neg(E,x)` returns $-x$.

`inv(E,x)` returns the inverse of x .

`equal0(x)` x being in canonical form, returns one if $x = 0$ and zero otherwise.

`s(n)` n being a small signed integer, returns n times the unit element.

A field is thus described by a `struct bb_field` as above and auxiliary data typecast to `void*`. The following functions operate on black box fields:

```
GEN gen_Gauss(GEN a, GEN b, void *E, const struct bb_field *ff)
GEN gen_Gauss_pivot(GEN x, long *rr, void *E, const struct bb_field *ff)
GEN gen_det(GEN a, void *E, const struct bb_field *ff)
GEN gen_ker(GEN x, long deplin, void *E, const struct bb_field *ff)
GEN gen_matcolinvimage(GEN a, GEN b, void *E, const struct bb_field *ff)
GEN gen_matcolmul(GEN a, GEN b, void *E, const struct bb_field *ff)
GEN gen_matid(long n, void *E, const struct bb_field *ff)
GEN gen_matinvimage(GEN a, GEN b, void *E, const struct bb_field *ff)
GEN gen_matmul(GEN a, GEN b, void *E, const struct bb_field *ff)
```

8.2.1 Functions returning black box fields.

```
const struct bb_field * get_Fp_field(void **pE, GEN p)
const struct bb_field * get_Fq_field(void **pE, GEN T, GEN p)
const struct bb_field * get_Flxq_field(void **pE, GEN T, ulong p)
const struct bb_field * get_F2xq_field(void **pE, GEN T)
const struct bb_field * get_nf_field(void **pE, GEN nf)
```

8.3 Black box algebra.

A black box algebra is defined by a `bb_algebra` struct, describing methods available to handle algebra elements:

```
struct bb_algebra
{
    GEN (*red)(void *E, GEN x);
    GEN (*add)(void *E, GEN x, GEN y);
    GEN (*sub)(void *E, GEN x, GEN y);
    GEN (*mul)(void *E, GEN x, GEN y);
    GEN (*sqr)(void *E, GEN x);
    GEN (*one)(void *E);
    GEN (*zero)(void *E);
};
```

In contrast with black box groups, elements can have non canonical forms, but only `add` is allowed to return a non canonical form.

`red(E,x)` returns the canonical form of x .

`add(E,x,y)` returns the sum $x + y$.

`sub(E,x,y)` returns the difference $x - y$.

`mul(E,x,y)` returns the product xy .

`sqr(E,x)` returns the square x^2 .

`one(E)` returns the unit element.

`zero(E)` returns the zero element.

An algebra is thus described by a `struct bb_algebra` as above and auxiliary data typecast to `void*`. The following functions operate on black box algebra:

`GEN gen_bkeval(GEN P, long d, GEN x, int use_sqr, void *E, const struct bb_algebra *ff, GEN cmul(void *E, GEN P, long a, GEN x))` x being an element of the black box algebra, and P some black box polynomial of degree d over the base field, returns $P(x)$. The function `cmul(E,P,a,y)` must return the coefficient of degree a of P multiplied by y . `cmul` is allowed to return a non canonical form; it is also allowed to return `NULL` instead of an exact 0.

The flag `use_sqr` has the same meaning as for `gen_powers`. This implements an algorithm of Brent and Kung (1978).

GEN gen_bkeval_powers(GEN P, long d, GEN V, void *E, const struct bb_algebra *ff, GEN cmul(void *E, GEN P, long a, GEN x)) as gen_RgX_bkeval assuming V was output by gen_powers(x, l, E, ff) for some $l \geq 1$. For optimal performance, l should be computed by brent_kung_optpow.

long brent_kung_optpow(long d, long n, long m) returns the optimal parameter l for the evaluation of n/m polynomials of degree d . Fractional values can be used if the evaluations are done with different accuracies, and thus have different weights.

8.3.1 Functions returning black box algebras.

const struct bb_algebra * get_FpX_algebra(void **E, GEN p, long v) return the algebra of polynomials over $\mathbf{F}_p$ in variable v .

const struct bb_algebra * get_Flx_algebra(void **E, ulong p, long v) return the algebra of polynomials over $\mathbf{F}_p$ in variable v (for ulong p).

const struct bb_algebra * get_FpXQ_algebra(void **E, GEN T, GEN p) return the algebra $\mathbf{F}_p[X]/(T(X))$.

const struct bb_algebra * get_Flxq_algebra(void **E, GEN T, ulong p) return the algebra $\mathbf{F}_p[X]/(T(X))$ (for ulong p).

const struct bb_algebra * get_FpXQX_algebra(void **E, GEN T, GEN p, long v) return the algebra of polynomials over $\mathbf{F}_p[X]/(T(X))$ in variable v .

const struct bb_algebra * get_FlxqX_algebra(void **E, GEN T, ulong p, long v) return the algebra of polynomials over $\mathbf{F}_p[X]/(T(X))$ in variable v (for ulong p).

const struct bb_algebra * get_FpXQXQ_algebra(void **E, GEN S, GEN T, GEN p) return the algebra $\mathbf{F}_p[X, Y]/(S(X, Y), T(X))$.

const struct bb_algebra * get_FlxqXQ_algebra(void **E, GEN S, GEN T, ulong p) return the algebra $\mathbf{F}_p[X, Y]/(S(X, Y), T(X))$ (for ulong p).

const struct bb_algebra * get_Rg_algebra(void) return the generic algebra (E is not used and can be set to NULL).

8.4 Black box ring.

A black box ring is defined by a `bb_ring` struct, describing methods available to handle ring elements:

```
struct bb_ring
{
    GEN (*add)(void *E, GEN x, GEN y);
    GEN (*mul)(void *E, GEN x, GEN y);
    GEN (*sqr)(void *E, GEN x);
};
```

add(E,x,y) returns the sum $x + y$.

mul(E,x,y) returns the product xy .

sqr(E,x) returns the square x^2 .

GEN `gen_fromdigits`(GEN `v`, GEN `B`, void `*E`, struct `bb_ring` `*r`) where B is a ring element and $v = [c_0, \dots, c_{n-1}]$ a vector of ring elements, return $\sum_{i=0}^n c_i B^i$ using binary splitting.

GEN `gen_digits`(GEN `x`, GEN `B`, long `n`, void `*E`, struct `bb_ring` `*r`, GEN `(*div)`(void `*E`, GEN `x`, GEN `y`, GEN `*r`))

(Require the ring to be Euclidean)

`div(E,x,y,&r)` performs the Euclidean division of x by y in the ring R , returning the quotient q and setting r to the residue so that $x = qy + r$ holds. The residue must belong to a fixed set of representatives of $R/(y)$.

The argument x being a ring element, `gen_digits` returns a vector of ring elements $[c_0, \dots, c_{n-1}]$ such that $x = \sum_{i=0}^n c_i B^i$. Furthermore for all $i \neq n-1$, the elements c_i belong to the fixed set of representatives of $R/(B)$.

8.5 Black box free $\mathbf{Z}_p$ -modules.

(Very experimental)

GEN `gen_ZpX_Dixon`(GEN `F`, GEN `V`, GEN `q`, GEN `p`, long `N`, void `*E`, GEN `lin`(void `*E`, GEN `F`, GEN `z`, GEN `q`), GEN `inv`(void `*E`, GEN `z`))

Let F be a `ZpXT` representing the coefficients of some abstract linear mapping f over $\mathbf{Z}_p[X]$ seen as a free $\mathbf{Z}_p$ -module, let V be an element of $\mathbf{Z}_p[X]$ and let $q = p^N$. Return $y \in \mathbf{Z}_p[X]$ such that $f(y) = V \pmod{p^N}$ assuming the following holds for $n \leq N$:

- `lin(E,FpX_red(F,p^n),z,p^n)  $\equiv f(z) \pmod{p^n}$`
- `f(inv(E,z))  $\equiv z \pmod{p}$`

The argument F is redundant; including it allows the function to reduce it to the required p -adic precision when evaluating f modulo p^n .

GEN `gen_ZpX_Newton`(GEN `x`, GEN `p`, long `n`, void `*E`, GEN `eval`(void `*E`, GEN `a`, GEN `q`), GEN `invd`(void `*E`, GEN `b`, GEN `v`, GEN `q`, long `N`))

Let x be an element of $\mathbf{Z}_p[X]$ seen as a free $\mathbf{Z}_p$ -module, and f some differentiable function over $\mathbf{Z}_p[X]$ such that $f(x) \equiv 0 \pmod{p}$. Return y such that $f(y) \equiv 0 \pmod{p^n}$, assuming the following holds for all $a, b \in \mathbf{Z}_p[X]$ and $M \leq N$:

- `v = eval(E,a,p^N)` is a vector of elements of $\mathbf{Z}_p[X]$,
- `w = invd(E,b,v,p^M,M)` is an element in $\mathbf{Z}_p[X]$,
- `v[1]  $\equiv f(a) \pmod{p^N \mathbf{Z}_p[X]}$` ,
- `df_a(w)  $\equiv b \pmod{p^M \mathbf{Z}_p[X]}$`

and df_a denotes the differential of f at a . Motivation: `eval` allows to evaluate f and `invd` allows to invert its differential. Frequently, data useful to compute the differential appear as a subproduct of computing the function. The vector v allows `eval` to provide these to `invd`. The implementation of `invd` will generally involves the use of the function `gen_ZpX_Dixon`.

GEN `gen_ZpM_Newton`(GEN `x`, GEN `p`, long `n`, void `*E`, GEN `eval`(void `*E`, GEN `a`, GEN `q`), GEN `invd`(void `*E`, GEN `b`, GEN `v`, GEN `q`, long `N`)) as above, with polynomials replaced by matrices.

GEN gen_ZpXX_Newton(GEN x, GEN p, long n, void *E, GEN eval(void *E, GEN a, GEN q),
GEN invd(void *E, GEN b, GEN v, GEN q, long N)) as above, with ZpX replaced by ZpXX.

Chapter 9:

Operations on general PARI objects

9.1 Conversions.

9.1.1 Scalars.

`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 length `prec`.

`double rtodbl(GEN x)` applied to a `t_REAL` `x`, converts `x` into a `double` if possible.

`GEN dbltor(double x)` converts the `double` `x` into a `t_REAL`.

`long dblexpo(double x)` returns `expo(dbltor(x))`, but faster and without cluttering the stack.

`ulong dblmantissa(double x)` returns the most significant word in the mantissa of `dbltor(x)`.

`int gisdouble(GEN x, double *dx)` if `x` is a real number (not necessarily a `t_REAL`) and if `x` can be converted to a `double`, set `*dx` to the converted value and return 1. Otherwise return 0.

`double gtodouble(GEN x)` if `x` is a real number (not necessarily a `t_REAL`), converts `x` into a `double` if possible.

`long gtos(GEN x)` converts the `t_INT` `x` to a small integer if possible, otherwise raise an exception. This function is similar to `itos`, slightly slower since it checks the type of `x`.

`ulong gtou(GEN x)` converts the non-negative `t_INT` `x` to an unsigned small integer if possible, otherwise raise an exception. This function is similar to `itou`, slightly slower since it checks the type of `x`.

`double dblmodulus(GEN x)` return an approximation to $|x|$.

`long gtolong(GEN x)` if `x` is an integer (not necessarily a `t_INT`), converts `x` into a `long` if possible.

`GEN fractor_lg(GEN x, long l)` applied to a `t_FRAC` `x`, converts `x` into a `t_REAL` of length `l`.

`GEN fractor(GEN x, long prec)` applied to a `t_FRAC` `x`, converts `x` into a `t_REAL` of precision `prec`.

`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`.

`GEN upper_to_cx(GEN x, long *prec)` valid for a `t_COMPLEX` or `t_QUAD` belonging to the upper half-plane. If a `t_QUAD`, convert it to `t_COMPLEX` using accuracy `*prec`. If `x` is inexact, sets `*prec` to the precision of `x`.

`GEN cxtofp(GEN x, long prec)` converts the `t_COMPLEX` `x` to a complex whose real and imaginary parts are `t_REAL` of length `prec` (special case of `gtofp`).

`GEN cxcompotor(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 `cxtoreal`(GEN `x`) converts the complex (`t_INT`, `t_REAL`, `t_FRAC` or `t_COMPLEX`) x to a real number if its imaginary part is 0. Shallow function.

converts the `t_COMPLEX` x to a complex whose real and imaginary parts are `t_REAL` of length `prec` (special case of `gtofp`).

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 $1.E-100 + 0.E-38 = 0.E-38$.

GEN `gtomp`(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 stack-clean if x is a `t_INT`: we return x itself and not a copy.

GEN `gcvtop`(GEN `x`, GEN `p`, long `l`) converts x into a `t_PADIC` of precision l . Works componentwise on recursive objects, e.g. `t_POL` or `t_VEC`. Converting 0 yields $O(p^l)$; converting a nonzero number yield a result well defined modulo $p^{v_p(x)+l}$.

GEN `cvtop`(GEN `x`, GEN `p`, long `l`) as `gcvtop`, assuming that x is a scalar.

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 `prec`) returns a shallow copy of x whose `t_REAL` components have their precision changed to `prec` bits. This is often more useful than `gprec`.

GEN `gprec_wtrunc`(GEN `x`, long `prec`) returns a shallow copy of x whose `t_REAL` components have their precision *truncated* to `prec` bits. Contrary to `gprec_w`, this function may never increase the precision of x .

GEN `gprec_wensure`(GEN `x`, long `prec`) returns a shallow copy of x whose `t_REAL` components have their precision *increased* to at least `prec` bits. Contrary to `gprec_w`, this function may never decrease the precision of x .

The following functions are obsolete and kept for backward compatibility only:

GEN `precision0`(GEN `x`, long `n`)

GEN `bitprecision0`(GEN `x`, long `n`)

9.1.2 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 `lift_shallow`(GEN `x`) shallow version of `lift`

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)`.

9.1.3 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 $(1, v, v^2, v^3, \dots)$.

GEN `normalizpol`(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 nonrational zero, and return the constant polynomial `c`. (We do this so that information about the base ring is not lost.)

GEN `normalizpol_lg`(GEN `x`, long `l`) applies `normalizpol` 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 `normalizpol(x)`.

GEN `normalizpol_approx`(GEN `x`, long `lx`) as `normalizpol_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 are all shallow: they do *not* copy coefficients on the stack and only move pointers around, hence are very fast but not suitable for `gc_GEN_unsafe` or `gc_upto` 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_RgC(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_RgC(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 RgM_to_RgXV_reverse(GEN x, long v) converts the RgM x to a t_VEC of RgX, by repeated calls to RgV_to_RgX_reverse.

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_POL s 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 $\text{varncmp}(v, w) \geq 0$.

GEN RgXX_to_RgM(GEN x, long N) converts the t_POL x with RgX (or constant) coefficients to a matrix with N rows.

long RgXY_degrees(GEN P) return the degree of P with respect to the secondary variable.

GEN RgXY_derivx(GEN P) return the derivative of P with respect to the secondary variable.

GEN RgXY_swap(GEN P, long n, long w) 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 $\deg_v(P)$.

GEN RgXY_swapspec(GEN C, long n, long w, long lP) as RgXY_swap where the coefficients of P are given by $\text{gel}(C, 0), \dots, \text{gel}(C, lP - 1)$.

GEN RgX_to_ser(GEN x, long l) convert the t_POL x to a *shallow* t_SER of length $l \geq 2$. 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 $O(T^{d+l-2})$.

GEN RgX_to_ser_inexact(GEN x, long l) convert the t_POL x to a *shallow* t_SER of length $l \geq 2$. Unless the polynomial is zero, the coefficient of lowest degree T^d of the result is not zero (as per `gequal0`). The remainder is $O(T^{d+l-2})$.

GEN RgV_to_ser(GEN x, long v, long l) convert the t_VEC x, to a *shallow* t_SER of length $l \geq 2$.

GEN rfrac_to_ser(GEN F, long l) applied to a t_RFRAC F, creates a t_SER of length $l \geq 2$ congruent to F. Not stack-clean but suitable for `gc_upto`.

GEN rfrac_to_ser_i(GEN F, long l) internal variant of `rfrac_to_ser`. Shallow function.

GEN rfracrecip_to_ser_absolute(GEN F, long d) applied to a t_RFRAC F, creates the t_SER $F(1/t) + O(t^d)$. Note that we use absolute and not relative precision here.

GEN `gtoser`(GEN `s`, long `v`, long `d`). This function is deprecated, kept for backward compatibility: it follows the semantic of `Ser(s,v)`, with $d = \text{seriesprecision}$ implied and is hard to use as a general conversion function. Use `gtoser_prec` instead.

It converts the object `s` into a `t_SER` with main variable number `v` and $d > 0$ significant terms, but the argument `d` is sometimes ignored. More precisely

- if `s` is a scalar (with respect to variable `v`), we return a constant power series with d significant terms;
- if `s` is a `t_POL` in variable `v`, it is truncated to d terms if needed;
- 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*;
- if `s` is already a power series in `v`, we return a copy, and the precision d is again *ignored*.

GEN `gtoser_prec`(GEN `s`, long `v`, long `d`) this function is a variant of `gtoser` following the semantic of `Ser(s,v,d)`: the precision d is always taken into account.

GEN `gtocol`(GEN `x`) converts the object `x` into a `t_COL`

GEN `gtomat`(GEN `x`) converts the object `x` into a `t_MAT`.

GEN `gtovec`(GEN `x`) converts the object `x` into a `t_VEC`.

GEN `gtovecsmall`(GEN `x`) converts the object `x` into a `t_VECSMALL`.

GEN `normalizeser`(GEN `x`) 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.

GEN `serchop0`(GEN `s`) given a `t_SER` of the form $x^v s(x)$, with $s(0) \neq 0$, return $x^v (s - s(0))$. Shallow function.

GEN `serchop_i`(GEN `x`, long `n`) returns a shallow choppy of `t_SER` `x` with all terms of degree strictly less than n removed. Shallow version of `serchop`.

9.2 Constructors.

9.2.1 Clean constructors.

GEN `zeropadic`(GEN `p`, long `n`) creates a 0 `t_PADIC` equal to $O(p^n)$.

GEN `zeroser`(long `v`, long `n`) creates a 0 `t_SER` in variable `v` equal to $O(X^n)$.

GEN `scalarser`(GEN `x`, long `v`, long `prec`) creates a constant `t_SER` in variable `v` and precision `prec`, whose constant coefficient is (a copy of) `x`, in other words $x + O(v^{\text{prec}})$. Assumes that `prec` ≥ 0 .

GEN `pol_0`(long `v`) Returns the constant polynomial 0 in variable `v`.

GEN `pol_1`(long `v`) Returns the constant polynomial 1 in variable `v`.

GEN `pol_x`(long `v`) Returns the monomial of degree 1 in variable `v`.

GEN `pol_xn`(long `n`, long `v`) Returns the monomial of degree n in variable `v`; assume that $n \geq 0$.

GEN `pol_xnall(long n, long v)` Returns the Laurent monomial of degree n in variable v ; $n < 0$ is allowed.

GEN `pol_x_powers(long N, long v)` returns the powers of `pol_x(v)`, of degree 0 to $N - 1$, in a vector with N components.

GEN `scalarpol(GEN x, long v)` creates a constant `t_POL` in variable v , whose constant coefficient is (a copy of) x .

GEN `deg1pol(GEN a, GEN b, long v)` creates the degree 1 `t_POL` $a\text{pol}_x(v) + b$

GEN `zeropol(long v)` is identical `pol_0`.

GEN `zerocol(long n)` creates a `t_COL` with n components set to `gen_0`.

GEN `zerovec(long n)` creates a `t_VEC` with n components set to `gen_0`.

GEN `zerovec_block(long n)` as `zerovec` but return a clone.

GEN `col_ei(long n, long i)` creates a `t_COL` with n components set to `gen_0`, but for the i -th one which is set to `gen_1` (i -th vector in the canonical basis).

GEN `vec_ei(long n, long i)` creates a `t_VEC` with n components set to `gen_0`, but for the i -th one which is set to `gen_1` (i -th vector in the canonical basis).

GEN `trivial_fact(void)` returns the trivial (empty) factorization `Mat([ ]~, [ ]~)`

GEN `prime_fact(GEN x)` returns the factorization `Mat([x]~, [1]~)`

GEN `Rg_col_ei(GEN x, long n, long i)` creates a `t_COL` with n components set to `gen_0`, but for the i -th one which is set to x .

GEN `vecsmall_ei(long n, long i)` creates a `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).

GEN `scalarcoll(GEN x, long n)` creates a `t_COL` with n components set to `gen_0`, but the first one which is set to a copy of x . (The name comes from `RgV_isscalar`.)

GEN `mkintmodu(ulong x, ulong y)` creates the `t_INTMOD` `Mod(x, y)`. The inputs must satisfy $x < y$.

GEN `zeromat(long m, long n)` creates a `t_MAT` with $m \times n$ components set to `gen_0`. Note that the result allocates a *single* column, so modifying an entry in one column modifies it in all columns. To fully allocate a matrix initialized with zero entries, use `zeromatcopy`.

GEN `zeromatcopy(long m, long n)` creates a `t_MAT` with $m \times n$ components set to `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 `mkfracss(long x, long y)` creates the `t_FRAC` x/y . Assumes that $y > 1$ and $(x, y) = 1$.

GEN sstoQ(long x, long y) returns the `t_INT` or `t_FRAC` x/y ; no assumptions.

GEN uutoQ(ulong x, ulong y) returns the `t_INT` or `t_FRAC` x/y ; no assumptions.

void Qtoss(GEN q, long *n, long *d) given a `t_INT` or `t_FRAC` q , set n and d such that $q = n/d$ with $d \geq 1$ and $(n, d) = 1$. Overflow error if numerator or denominator do not fit into a long integer.

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 mkcols(long x) creates a 1-dimensional `t_COL` containing `stoi(x)`.

GEN mkcol2s(long x, long y) creates a 2-dimensional `t_COL` containing $[\text{stoi}(x), \text{stoi}(y)]$.

GEN mkcol3s(long x, long y, long z) creates a 3-dimensional `t_COL` containing $[\text{stoi}(x), \text{stoi}(y), \text{stoi}(z)]$.

GEN mkcol4s(long x, long y, long z, long t) creates a 4-dimensional `t_COL` containing $[\text{stoi}(x), \text{stoi}(y), \text{stoi}(z), \text{stoi}(t)]$.

GEN mkvecs(long x) creates a 1-dimensional `t_VEC` containing `stoi(x)`.

GEN mkvec2s(long x, long y) creates a 2-dimensional `t_VEC` containing $[\text{stoi}(x), \text{stoi}(y)]$.

GEN mkmat22s(long a, long b, long c, long d) creates the 2 by 2 `t_MAT` with successive rows $[\text{stoi}(a), \text{stoi}(b)]$ and $[\text{stoi}(c), \text{stoi}(d)]$.

GEN mkvec3s(long x, long y, long z) creates a 3-dimensional `t_VEC` containing $[\text{stoi}(x), \text{stoi}(y), \text{stoi}(z)]$.

GEN mkvec4s(long x, long y, long z, long t) creates a 4-dimensional `t_VEC` containing $[\text{stoi}(x), \text{stoi}(y), \text{stoi}(z), \text{stoi}(t)]$.

GEN mkvec5s(long x, long y, long z, long t, long u) creates a 5-dimensional `t_VEC` containing $[\text{stoi}(x), \text{stoi}(y), \text{stoi}(z), \text{stoi}(t), \text{stoi}(u)]$.

GEN mkvec6s(long x, long y, long z, long t, long u, long v) creates a 6-dimensional `t_VEC` containing $[\text{stoi}(x), \text{stoi}(y), \text{stoi}(z), \text{stoi}(t), \text{stoi}(u), \text{stoi}(v)]$.

GEN mkvecsmall(long x) creates a 1-dimensional `t_VECSMALL` containing x .

GEN mkvecsmall2(long x, long y) creates a 2-dimensional `t_VECSMALL` containing $[x, y]$.

GEN mkvecsmall3(long x, long y, long z) creates a 3-dimensional `t_VECSMALL` containing $[x, y, z]$.

GEN mkvecsmall4(long x, long y, long z, long t) creates a 4-dimensional `t_VECSMALL` containing $[x, y, z, t]$.

GEN mkvecsmall5(long x, long y, long z, long t, long u) creates a 5-dimensional `t_VECSMALL` containing $[x, y, z, t, u]$.

GEN mkvecsmalln(long n, ...) returns the `t_VECSMALL` whose n coefficients (long) follow.

Warning.. Since this is a variadic function, the compiler does not perform C type promotion on the arguments. You have to ensure that all the arguments following n are of type `long`, in particular integer constants need to be written with the L suffix: `mkvecsmalln(2, 1L, 2L)` is correct, but `mkvecsmalln(2, 1, 2)` is not.

9.2.2 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 `gc_upto`. 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 $-ix$.

`GEN mulcxpowIs(GEN x, long k)`, as `mulcxI`, but returns $x \cdot i^k$.

`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 quadpoly_i(GEN D)` creates the canonical quadratic polynomial of discriminant D . Assume that the `t_INT` D is congruent to 0,1 mod 4 and not a square.

`GEN mkfrac(GEN x, GEN y)` creates the `t_FRAC` x/y . Assumes that $y > 1$ and $(x, y) = 1$.

`GEN mkrfrac(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 mkcol6(GEN x, GEN y, GEN z, GEN t, GEN u, GEN v)` creates the 6-dimensional column vector $[x, y, z, t, u, v]$.

`GEN mkintmod(GEN x, GEN y)` creates the `t_INTMOD` $\text{Mod}(x, y)$. The inputs must be `t_INT`s 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 mkpadic(GEN u, GEN p, GEN pd, long v, long d)` creates the `t_PADIC` $p^v(u + O(p^d))$ of valuation v and precision d . The arguments u, p, pd must be `t_INT` with $0 \leq u < p^d$, p prime and $pd = p^d$. The value $u = 0$ is allowed if and only if $d = 0$, otherwise u must be a p -unit.

`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 `mkmat22`(GEN `a`, GEN `b`, GEN `c`, GEN `d`) creates the 2 by 2 `t_MAT` with successive rows `[a, b]` and `[c, d]`.

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 `t_VEC` equal to `[a1, a2, a3, a4, a5]`.

GEN `mkvec6`(GEN `a1`, GEN `a2`, GEN `a3`, GEN `a4`, GEN `a5`, GEN `a6`) creates the 6-dimensional `t_VEC` equal to `[a1, a2, a3, a4, a5, a6]`.

GEN `mkqfb`(GEN `a`, GEN `b`, GEN `c`, GEN `D`) creates `t_QFB` equal to `Qfb(a,b,c)`, assuming that $D = b^2 - 4ac$.

GEN `mkerr`(long `n`) returns a `t_ERROR` with error code `n` (enum `err_list`).

It is sometimes useful to return such a container whose entries are not universal objects, but nonetheless suitable for `gc_upto`. 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

```
{
  GEN res = cgetg(2, t_VEC);
  gel(res, 1) = x; /* or rather, the expansion of x */
  return res;
}
```

For instance, the `retmkvec(gcopy(x))` returns a clean object, just like `return mkveccopy(x)` would.

GEN `retmkvec2`(GEN `x`, GEN `y`) returns the 2-dimensional `t_VEC` `[x,y]`.

GEN `retmkvec3`(GEN `x`, GEN `y`, GEN `z`) returns the 3-dimensional `t_VEC` `[x,y,z]`.

GEN `retmkvec4`(GEN `x`, GEN `y`, GEN `z`, GEN `t`) returns the 4-dimensional `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 retmkvec6(GEN x, GEN y, GEN z, GEN t, GEN u, GEN v) returns the 5-dimensional row vector $[x,y,z,t,u,v]$.

GEN retconst_vec(long n, GEN x) returns the n -dimensional t_VEC whose entries are constant and all equal to x .

GEN retmkcol(GEN x) returns the 1-dimensional t_COL $[x]$.

GEN retmkcol2(GEN x, GEN y) returns the 2-dimensional t_COL $[x,y]$.

GEN retmkcol3(GEN x, GEN y, GEN z) returns the 3-dimensional t_COL $[x,y,z]$.

GEN retmkcol4(GEN x, GEN y, GEN z, GEN t) returns the 4-dimensional 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 retmkcol6(GEN x, GEN y, GEN z, GEN t, GEN u, GEN v) returns the 6-dimensional column vector $[x,y,z,t,u,v]$.

GEN retconst_col(long n, GEN x) returns the n -dimensional t_COL whose entries are constant and all equal to x .

GEN retmkmat(GEN x) returns the 1-column t_MAT with column x .

GEN retmkmat2(GEN x, GEN y) returns the 2-column t_MAT with columns x, y .

GEN retmkmat3(GEN x, GEN y, GEN z) returns the 3-dimensional t_MAT with columns x, y, z .

GEN retmkmat4(GEN x, GEN y, GEN z, GEN t) returns the 4-dimensional 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 retmkmat22(GEN a, GEN b, GEN c, GEN d) return the 2 by 2 t_MAT with successive rows $[a,b]$ and $[c,d]$.

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 retmkrffrac(GEN x, GEN y) returns the t_RFRAC x / y . Assume x and y are coprime and more generally that the rational function cannot be simplified.

GEN retmkintmod(GEN x, GEN y) returns the t_INTMOD $\text{Mod}(x, y)$.

GEN retmkpadic(GEN u, GEN p, GEN pd, long v, long d) returns the t_PADIC $p^v(u + O(p^d))$ of valuation v and precision d . See `mkpadic` for constraints on arguments.

GEN retmkpadic_i(GEN u, GEN p, GEN pd, long v, long d) this is a technical internal variant of `retmkpadic`, where the `pd` argument is an expression computing $_pd := p^d$ and the u argument is an expression allowed to use that private `\_pd` variable. Both expressions must be stack-clean. This is used as in

```
retmkpadic_i(modii(x, \_pd), p, powiu(p,d), 0, d);
```

where the `\_pd` trick allows to defer evaluation until the macro has created the t_PADIC root of the object. The result is then stack-clean.

GEN retmkquad(GEN n, GEN a, GEN b).

GEN retmkpolmod(GEN x, GEN y) returns the t_POLMOD Mod(x , y).

GEN retmkqfb(GEN a, GEN b, GEN c, GEN D) returns the t_QFB equal to $Qfb(a,b,c)$, assuming that $D = b^2 - 4ac$.

GEN mkintn(long n, ...) returns the nonnegative t_INT whose expansion in base 2^{32} is given by the following n digits. Digits are `unsigned int` less than 2^{32} .

```
mkintn(3, 0x1U, 0x2U, 0x3U);
```

returns $2^{64} + 2 \cdot 2^{32} + 3$.

Warning.. Since this is a variadic function, the compiler does not perform C type promotion on the arguments. You have to ensure that all the arguments following n are of type `unsigned int`, in particular integer constants need to be written with the U suffix as above: `mkvecsmalln(3, 1, 2, 3)` would not be correct.

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

```
mkpoln(3, gen_1, gen_2, 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 scalarcol_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_isscalar`.)

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 RgX_sylvestermatrix(GEN f, GEN g) return the Sylvester matrix attached to the two t_POL in the same variable f and g .

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 deg1pol_shallow(GEN a, GEN b, long v) returns the degree 1 t_POL $apol_x(v) + b$

GEN deg2pol_shallow(GEN a, GEN b, GEN c, long v) returns the degree 2 t_POL $ax^2 + bx + c$ where $x = pol_x(v)$.

GEN zeropadic_shallow(GEN p, long n) returns a (shallow) 0 t_PADIC equal to $O(p^n)$.

9.2.3 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*]. Leaves some garbage on stack, but suitable for **gc_upto**.

GEN roots_to_pol_r1(GEN *L*, long *v*, long *r1*) as **roots_to_pol** assuming the first *r1* 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 $X^2 - \text{trace}(z)X + \text{norm}(z)$. Leaves some garbage on stack, but suitable for **gc_upto**.

9.3 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, then significant bits are lost in the rounded result. This kind of situation raises an error message in **ground** but not in **grndtoi**. The parameter *e* can be set to NULL if an error estimate is not needed, for a minor speed up.

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*).

9.4 Valuation and shift.

GEN gshift(GEN *x*, long *n*) yields the result of shifting (the components of) *x* left by *n* (if *n* is nonnegative) 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(GEN *x*, long *n*) 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*.

9.5 Comparison operators.

9.5.1 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` cannot be compared to any non-string type. If neither x nor y is a `t_STR`, their allowed types are `t_INT`, `t_REAL`, `t_FRAC`, `t_QUAD` with positive discriminant (use the canonical embedding $w \rightarrow \sqrt{D}/2$ or $w \rightarrow (1 + \sqrt{D})/2$) or `t_INFINITY`. Use `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 componentwise. 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.

`GEN gmax(GEN x, GEN y)` returns a copy of the maximum of x and y , compared using `gcmp`.

`GEN gmin(GEN x, GEN y)` returns a copy of the minimum of x and y , compared using `gcmp`.

`GEN gmax_shallow(GEN x, GEN y)` shallow version of `gmax`.

`GEN gmin_shallow(GEN x, GEN y)` shallow version of `gmin`.

9.5.2 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.

`GEN gisexactzero(GEN x)` returns `NULL` unless x is exactly equal to 0 (as per `isexactzero`). When x is an exact zero return the attached scalar zero as a `t_INT` (`gen_0`), a `t_INTMOD` (`Mod(0,N)` for the largest possible N) or a `t_FFELT`.

`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 ismpzzero(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 *nonzero* `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 gminsg(long s, GEN x)`

`GEN gmings(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.

9.6 Miscellaneous Boolean functions.

`int isrationalzeroscalar(GEN x)` equivalent to, but faster than,

`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 `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 `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 `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 `t_REAL`), or raises an error if x is not embeddable into the complex numbers.

9.6.1 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)
```

9.7 Sorting.

9.7.1 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.

9.7.2 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 minimal 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`).

9.7.3 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_shallow(GEN x, void *data, int (*cmp)(void *, GEN, GEN))`, shallow variant of `gen_sort`.

`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 not `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 .

`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. Shallow function.

`GEN setunion_i(GEN x, GEN y)` shallow version of `setunion`, a simple alias for

`merge_sort_uniq(x,y, (void*)cmp_universal, cmp_nodata)`

`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 $X * Y$.

`GEN ZM_merge_factor(GEN fx, GEN fy)` as `merge_factor`, where `fx` and `fy` are factorization matrices of integers.

`long gen_search(GEN v, GEN y, 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 . If y is found, return i (not necessarily the first occurrence in case of multisets), else return $-i$ where i is the index where y should be inserted.

`long tablesearch(GEN T, GEN x, int (*cmp)(GEN, GEN))` is a faster implementation for the common case `gen_search(T,x,cmp,cmp_nodata)` when we have no need to insert missing elements; return 0 in case x is not found.

9.7.4 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_Flx(GEN x, GEN y)` compare two Flx, which we assume have the same main variable (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`.

`int cmp_padic(GEN x, GEN y)` compare two `t_PADIC` (for the same prime 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;
```

9.8 Division.

`GEN gdivgu(GEN x, ulong u)` return x/u .

`GEN gdivgunextu(GEN x, ulong u)` return $x/(u(u+1))$. If $u(u+1)$ does not fit into an `ulong`, it is created and left on the stack for efficiency.

`GEN divrunextu(GEN x, ulong i)` as `gdivgunextu` for a `t_REAL` x .

9.9 Divisibility, Euclidean division.

`GEN gdivexact(GEN x, GEN y)` returns the quotient x/y , assuming y divides x . Not stack clean if $y = 1$ (we return x , not a copy).

`int gdvd(GEN x, GEN y)` returns 1 (true) if y divides x , 0 otherwise.

`GEN gdiventres(GEN x, GEN y)` creates a 2-component vertical vector whose components are the true Euclidean quotient and remainder of x and y .

`GEN gdivent(GEN x, GEN y)` yields the true Euclidean quotient of x and the `t_INT` or `t_POL` y , as per the `\ GP` operator.

`GEN gdiventsg(long s, GEN y)`, as `gdivent` except that x is a `long`.

`GEN gdiventgs(GEN x, long s)`, as `gdivent` except that y is a `long`.

`GEN gmod(GEN x, GEN y)` yields the remainder of x modulo the `t_INT` or `t_POL` y , as per the `% GP` operator. A `t_REAL` or `t_FRAC` y is also allowed, in which case the remainder is the unique real r such that $0 \leq r < |y|$ and $y = qx + r$ for some (in fact unique) integer q .

`GEN gmodsg(long s, GEN y)` as `gmod`, except x is a `long`.

`GEN gmodgs(GEN x, long s)` as `gmod`, except y is a `long`.

`GEN gdivmod(GEN x, GEN y, GEN *r)` If r is not equal to `NULL` or `ONLY_REM`, creates the (false) Euclidean quotient of x and y , and puts (the address of) the remainder into $*r$. If r is equal to `NULL`, do not create the remainder, and if r 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)`.

`GEN poldivrem(GEN x, GEN y, GEN *r)` same as `gdivmod` but specifically for `t_POLs` x and y , not necessarily in the same variable. Either of x and y may also be scalars, treated as polynomials of degree 0.

`GEN gdeuc(GEN x, GEN y)` creates the Euclidean quotient of the `t_POLs` x and y . Either of x and y may also be scalars, treated as polynomials of degree 0.

`GEN grem(GEN x, GEN y)` creates the Euclidean remainder of the `t_POL` x divided by the `t_POL` y . Either of x and y may also be scalars, treated as polynomials of degree 0.

`GEN gdivround(GEN x, GEN y)` if x and y are real (`t_INT`, `t_REAL`, `t_FRAC`), return the rounded Euclidean quotient of x and y as per the `\ / GP` operator. Operate componentwise if x is a `t_COL`, `t_VEC` or `t_MAT`. Otherwise as `gdivent`.

`GEN centermod_i(GEN x, GEN y, GEN y2)`, as `centermodii`, componentwise.

`GEN centermod(GEN x, GEN y)`, as `centermod_i`, except that $y2$ is computed (and left on the stack for efficiency).

`GEN ginvmod(GEN x, GEN y)` creates the inverse of x modulo y when it exists. y must be of type `t_INT` (in which case x is of type `t_INT`) or `t_POL` (in which case x is either a scalar type or a `t_POL`).

9.10 GCD, content and primitive part.

9.10.1 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 $\mathbf{Q}[X]$) 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 glcm(GEN x, GEN y)` creates the LCM of `x` and `y`.

`GEN gbezout(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 subrext(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.)

`GEN denom_i(GEN x)` shallow version of `denom`.

`GEN numer_i(GEN x)` shallow version of `numer`.

9.10.2 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_lval(GEN x, ulong p)` same for `ulong p`.

`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`.

`long Q_lvalrem(GEN x, ulong p, GEN *r)` same for `ulong p`.

`GEN Q_abs(GEN x)` absolute value of the `t_INT` or `t_FRAC` `x`.

`GEN Qdivii(GEN x, GEN y)`, assuming x and y are both of type `t_INT`, return the quotient x/y as a `t_INT` or `t_FRAC`; marginally faster than `gdiv`.

`GEN Qdivis(GEN x, long y)`, assuming x is an `t_INT`, return the quotient x/y as a `t_INT` or `t_FRAC`; marginally faster than `gdiv`.

`GEN Qdiviu(GEN x, ulong y)`, assuming x is an `t_INT`, return the quotient x/y as a `t_INT` or `t_FRAC`; marginally faster than `gdiv`.

`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 $M \otimes_{\mathbf{Z}} \mathbf{Q}$ for some natural $\mathbf{Z}$ -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 *integral*. We are interested in contents, denominators, etc. with respect to this canonical integral structure; in particular, contents belong to $\mathbf{Q}$, denominators to $\mathbf{Z}$. For instance the $\mathbf{Q}$ -content of $(1/2)xy$ is $(1/2)$, and its $\mathbf{Q}$ -denominator is 2, whereas `content` would return $y/2$ and `denom` 1.

`GEN Q_content(GEN x)` the $\mathbf{Q}$ -content of x .

`GEN Z_content(GEN x)` as `Q_content` but assume that all rationals are in fact `t_INTs` and return `NULL` when the content is 1. This function returns as soon as the content is found to equal 1.

`GEN Q_content_safe(GEN x)` as `Q_content`, returning `NULL` when the $\mathbf{Q}$ -content is not defined (e.g. for a `t_REAL` or `t_INTMOD` component).

`GEN Q_denom(GEN x)` the $\mathbf{Q}$ -denominator of x . Shallow function. Raises an `e_TYPE` error out when the notion is meaningless, e.g. for a `t_REAL` or `t_INTMOD` component.

`GEN Q_denom_safe(GEN x)` the $\mathbf{Q}$ -denominator of x . Shallow function. Return `NULL` when the notion is meaningless.

`GEN Q_primitive_part(GEN x, GEN *c)` sets c to the $\mathbf{Q}$ -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 vec_Q_primpart(GEN x)` as above component-wise. Applied to a `t_MAT`, the result has primitive columns.

`GEN row_Q_primpart(GEN x)` as above, applied to the rows of a `t_MAT`, so that the result has primitive rows. Shallow function.

`GEN Q_remove_denom(GEN x, GEN *ptd)` sets d to the $\mathbf{Q}$ -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 div_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 quotient (either a `GEN` or `NULL`).

`GEN inv_content(GEN c)` c is as set by `primitive_part`: either a `GEN` or `NULL` representing the trivial content 1. Returns its inverse (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`).

9.11 Generic arithmetic operators.

9.11.1 Unary operators.

GEN `gneg`(GEN `x`) yields $-x$.

GEN `gneg_i`(GEN `x`) shallow function yielding $-x$.

GEN `gabs`(GEN `x`) yields $|x|$.

GEN `gsqr`(GEN `x`) creates the square of `x`.

GEN `ginv`(GEN `x`) creates the inverse of `x`.

9.11.2 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 `gopgu`(GEN `x`, ulong `u`)

GEN `gopsg`(long `s`, GEN `y`)

GEN `gopug`(ulong `u`, GEN `y`)

Explicitly

GEN `gadd`(GEN `x`, GEN `y`), GEN `gaddgs`(GEN `x`, long `s`), GEN `gaddsg`(long `s`, GEN `x`)

GEN `gmul`(GEN `x`, GEN `y`), GEN `gmulgs`(GEN `x`, long `s`), GEN `gmulsg`(long `s`, GEN `x`), GEN `gmulgu`(GEN `x`, ulong `u`), GEN `gmulug`(GEN `x`, ulong `u`),

GEN `gsub`(GEN `x`, GEN `y`), GEN `gsubgs`(GEN `x`, long `s`), GEN `gsubsg`(long `s`, GEN `x`)

GEN `gdiv`(GEN `x`, GEN `y`), GEN `gdivgs`(GEN `x`, long `s`), GEN `gdivsg`(long `s`, GEN `x`), GEN `gdivgu`(GEN `x`, ulong `u`),

GEN `gpow`(GEN `x`, GEN `y`, long `l`) creates x^y . If `y` is a `t_INT`, return `powgi(x,y)` (the precision `l` is not taken into account). Otherwise, the result is $\exp(y * \log(x))$ where exact arguments are converted to floats of precision `l` 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`.
- `qfb_1(x)` for `t_QFB`.
- the identity permutation for `t_VECSMALL`.
- `Rg_get_1(x)` otherwise

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 `gpows(x, 0)`.

`GEN gpowers(GEN x, long n)` returns the vector $[1, x, \dots, x^n]$.

`GEN grootsof1(long n, long prec)` returns the vector $[1, x, \dots, x^{n-1}]$, where x is the n -th root of unity $\exp(2i\pi/n)$.

`GEN gsqrpowers(GEN x, long n)` returns the vector $[x, x^4, \dots, x^{n^2}]$.

9.12 Generic operators: product, powering, factorback.

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 gen_product(GEN v, void *D, F2 op)` Given two objects x, y , assume that `op(D, x, y)` implements an associative binary operator. If v has k entries, return

$$v[1] \text{ op } v[2] \text{ op } \dots \text{ op } v[k];$$

returns `gen_1` if $k = 0$ and a copy of $v[1]$ if $k = 1$. Use divide and conquer strategy. Leave some garbage on stack, but suitable for `gc_upto` if `mul` is.

`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 stack-clean.

GEN `gen_powu_i`(GEN `x`, `ulong n`, `void *E`, `F1 sqr`, `F2 mul`) internal variant of `gen_powu`, not stack-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 stack-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_pow_init`(GEN `x`, GEN `n`, `long k`, `void *E`, GEN `(*sqr)(void*, GEN)`, GEN `(*mul)(void*, GEN, GEN)`) Return a table `R` that can be used with `gen_pow_table` to compute the powers of `x` up to `n`. The table is of size $2^k \log_2(n)$.

GEN `gen_pow_table`(GEN `R`, GEN `n`, `void *E`, GEN `(*one)(void*)`, GEN `(*mul)(void*, GEN, GEN)`)

Return x^n , where `R` is as given by `gen_pow_init(x,m,k,E,sqr,mul)` for some integer $m \geq n$.

GEN `gen_powers`(GEN `x`, `long n`, `long usesqr`, `void *D`, `F1 sqr`, `F2 mul`, `F0 one`) returns $[x^0, \dots, x^n]$ 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`, `void *D`, `F2 mul`, `F2 pow`, GEN `(*one)(void *)D`) generic form of `factorback`. The pair $[L, e]$ is of the form

- `[fa, NULL]`, `fa` a two-column factorization matrix: expand it.
- `[v, NULL]`, `v` a vector of objects: return their product.
- or `[v, e]`, `v` a vector of objects, `e` a vector of integral exponents (a `ZV` or `zv`): return the product of the $v[i]^{e[i]}$.

`mul(D, x, y)` and `pow(D, x, n)` return xy and x^n respectively.

`one(D)` returns the neutral element. If `one` is `NULL`, `gen_1` is used instead.

9.13 Matrix and polynomial norms.

This section concerns only standard norms of **R** and **C** vector spaces, not algebraic norms given by the determinant of some multiplication operator. We have already seen type-specific functions like `ZM_supnorm` or `RgM_fpnorml2` and limit ourselves to generic functions assuming nothing about their `GEN` argument; these functions allow the following scalar types: `t_INT`, `t_FRAC`, `t_REAL`, `t_COMPLEX`, `t_QUAD` and are defined recursively (in terms of norms of their components) for the following “container” types: `t_POL`, `t_VEC`, `t_COL` and `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 *prec* in case the inputs include **t_COMPLEX** or **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 **t_QUAD** $x + wy$ or **t_COMPLEX** $x + Iy$ is defined as $|x| + |y|$, where we use the ordinary real absolute value. This is still a norm of **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 *prec* must be included for the same reason as in **gnorml1**.

void gsupnorm_aux(GEN *x*, GEN **m*, GEN **m2*, long *prec*) is the low-level function underlying **gsupnorm**, used as follows:

```
GEN m = NULL, m2 = NULL;
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 **t_QUAD** or **t_COMPLEX** z : *m2* is the max of those $|z|^2$, and *m* is the max of the other $|z|$.

9.14 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 **t_POL** or **t_RFRAC** q at x . For convenience, a **t_VEC** or **t_COL** is also recognized as the **t_POL** **gtovecrev**(*q*).

GEN RgX_cxeval(GEN *T*, GEN *x*, GEN *xi*) evaluate the **t_POL** T at x via Horner's scheme. If xi is not NULL it must be equal to $1/x$ and we evaluate $x^{\deg T} T(1/x)$ instead. This is useful when $|x| > 1$ is a **t_REAL** or an inexact **t_COMPLEX** and T has "balanced" coefficients, since the evaluation becomes numerically stable.

GEN RgXY_cxevalx(GEN *T*, GEN *x*, GEN *xi*) Apply **RgX_cxeval** to all the polynomials coefficients of T .

GEN RgX_RgM_eval(GEN *q*, GEN *x*) evaluates the **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 **FpXQ_powers**(*x*, *n*) for some $n > 1$.

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 attached 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_RgM_apply(GEN q, GEN M) q is a symmetric $n \times n$ matrix, M an $n \times k$ matrix, return $M'qM$.

GEN qf_ZM_apply(GEN q, GEN M) as above assuming that both q and M have integer entries.

Chapter 10: Miscellaneous mathematical functions

10.1 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 `gneg(x)` otherwise.

GEN `sqrfrac`(GEN `x`) returns the square of the `t_FRAC` `x`.

10.2 Binomials.

GEN `binomial`(GEN `x`, long `k`)

GEN `binomialuu`(ulong `n`, ulong `k`)

GEN `vecbinomial`(long `n`), which returns a vector `v` with $n + 1$ `t_INT` components such that $v[k + 1] = \text{binomial}(n, k)$ for k from 0 up to n .

10.3 Real numbers.

GEN `R_abs`(GEN `x`) `x` being a `t_INT`, a `t_REAL` or a `t_FRAC`, returns $|x|$.

GEN `R_abs_shallow`(GEN `x`) `x` being a `t_INT`, a `t_REAL` or a `t_FRAC`, returns a shallow copy of $|x|$, in particular returns `x` itself when $x \geq 0$, and `gneg(x)` otherwise.

GEN `modRr_safe`(GEN `x`, GEN `y`) let `x` be a `t_INT`, a `t_REAL` or `t_FRAC` and let `y` be a `t_REAL`. Return $x \% y$ unless the input accuracy is insufficient to compute the floor or x/y in which case we return `NULL`.

GEN `modRr_i`(GEN `x`, GEN `y`, GEN `iy`) let `x` be a `t_INT`, a `t_REAL` or `t_FRAC` and let `y` be a `t_REAL` and `iy = invr(y)`. Return $x \% y$ unless the input accuracy is insufficient to compute the floor or x/y in which case we return `NULL`.

10.4 Complex numbers.

GEN `gimag`(GEN `x`) returns a copy of the imaginary part of `x`.

GEN `greal`(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 $(1, \omega)$.

GEN `gconj`(GEN `x`) returns `greal(x) - 2gimag(x)`, which is the ordinary complex conjugate except for a real `t_QUAD`.

GEN `imag_i`(GEN `x`), shallow variant of `gimag`.

GEN `real_i`(GEN `x`), shallow variant of `greal`.

GEN `conj_i`(GEN `x`), shallow variant of `gconj`.

GEN `mulreal`(GEN `x`, GEN `y`) returns the real part of xy ; x, y have type `t_INT`, `t_FRAC`, `t_REAL` or `t_COMPLEX`. See also `RgM_mulreal`.

GEN `mulimag`(GEN `x`, GEN `y`) returns the imaginary part of xy ; x, y have type `t_INT`, `t_FRAC`, `t_REAL` or `t_COMPLEX`.

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 .

int `cx_approx_equal`(GEN `a`, GEN `b`) test whether (`t_INT`, `t_FRAC`, `t_REAL`, or `t_COMPLEX` of those) a and b are approximately equal. This returns 1 if and only if the division by $a - b$ would produce a division by 0 (which is a less stringent test than testing whether $a - b$ evaluates to 0).

int `cx_approx0`(GEN `a`, GEN `b`) test whether (`t_INT`, `t_FRAC`, `t_REAL`, or `t_COMPLEX` of those) a is approximately 0, where b is a reference point. A non-0 `t_REAL` component x is approximately 0 if

$$\text{exponent}(b) - \text{exponent}(x) > \text{bit_prec}(x) .$$

10.5 Low accuracy input or output.

double `dbllog2r`(GEN `x`) assuming that `x` is a nonzero `t_REAL`, returns an approximation to $\log_2(|x|)$.

double `dbllog2`(GEN `x`) assuming that `x` is a complex number of type `t_INT`, `t_FRAC`, `t_REAL` or a `t_COMPLEX` (with real components) returns an approximation to $\log_2(|x|)$. Returns 1./0. if x is 0.

double `dbllambertW0`(double `y`) given $y > e^{-1}$ return a rough approximation x to $W_0(y)$, where W_0 is the principal branch of the Lambert function (relative error less than 1%). We thus have $xe^x \approx y$.

double `dbllambertW_1`(double `y`) given $0 > y > e^{-1}$ return a rough approximation x to $W_{-1}(y)$ (relative error less than 1%). We thus have $xe^x \approx y$.

Finally the following 4 functions provide low-accuracy evaluations at the complex number $z = s + it$:

double `dblcnorm`(double `s`, double `t`) returns $|z|^2 = s^2 + t^2$.

double `dblcabs`(double `s`, double `t`) returns $|z| = \sqrt{s^2 + t^2}$.

`double dblcarg(double s, double t)` returns $\arg z$, with values in $] - \pi, \pi]$. Assumes $z \neq 0$.

`void dblclog(double s, double t, double *a, double *b)` sets $a + ib = \text{Log}(z)$, where Log is the principal branch of the natural logarithm, with imaginary part in $] - \pi, \pi]$. Assumes $z \neq 0$.

10.6 Quadratic numbers and binary quadratic forms.

`GEN quad_disc(GEN x)` returns the discriminant of the `t_QUAD` x . Not stack-clean but suitable for `gc_upto`.

`GEN quadnorm(GEN x)` norm of the `t_QUAD` x .

`GEN qfb_disc(GEN x)` returns the discriminant of the `t_QFB` x .

`GEN qfb_disc3(GEN x, GEN y, GEN z)` returns $y^2 - 4xz$ assuming all inputs are `t_INTs`. Not stack-clean.

`GEN qfb_ZM_apply(GEN q, GEN g)` returns $q \circ g$.

`GEN qfbforms(GEN D)` given a discriminant $D < 0$, return the list of reduced forms of discriminant D as `t_VECSMALL` with 3 components. The primitive forms in the list enumerate the class group of the quadratic order of discriminant D ; if D is fundamental, all returned forms are automatically primitive.

10.7 Polynomials.

`GEN truecoef(GEN x, long n)` returns `polcoef(x,n,-1)`, i.e. the coefficient of the term of degree n in the main variable. This is a safe but expensive function that *copies* its return value to ease garbage collection. Use `polcoef_i` for a fast shallow variant.

`GEN polcoef_i(GEN x, long n, long v)` 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 nonzero. 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 nonzero (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` and `t_PADIC` components. 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_INTMODs` occurring

in the structure, as well as primes p arising from `t_PADICs` or `t_FFELTs`. The function raises an exception if incompatible primes arise from `t_FFELT` and `t_PADIC` components. 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 $\mathbf{Q}[X]$) 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 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_REALs` of precision `prec` (guaranteeing a nonzero imaginary part). See `QX_complex_roots`.

`double fujiwara_bound(GEN x)` return a quick upper bound for the logarithm in base 2 of the modulus of the largest complex roots of the polynomial x (complex coefficients).

`double fujiwara_bound_real(GEN x, long sign)` return a quick upper bound for the logarithm in base 2 of the absolute value of the largest real root of sign $sign$ (1 or -1), for the polynomial x (real coefficients).

`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`.

`GEN pollegendre_reduced(long n, long v)` let $P_n(t) \in \mathbf{Q}[t]$ be the n -th Legendre polynomial in variable v . Return $p \in \mathbf{Z}[t]$ such that $2^n P_n(t) = p(t^2)$ (n even) or $tp(t^2)$ (n odd).

10.7.1 Obsolete functions.

`long sturmpart(GEN x, GEN a, GEN b)`. Please use `RgX_sturmpart` instead, and preferably `ZX_sturmpart` with exact rational endpoints.

10.8 Power series.

`GEN sertoser(GEN x, long prec)` return the `t_SER` x truncated or extended (with zeros) to `prec` terms. Shallow function, assume that `prec` ≥ 0 .

`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 truecoef(GEN x, long n)` returns `polcoef(x,n,-1)`, i.e. the coefficient of the term of degree n in the main variable. This is a safe but expensive function that *copies* its return value to simplify garbage collection. Use `polcoef_i` for a fast shallow variant.

`GEN ser_unscale(GEN P, GEN h)` return $P(hx)$, not stack-clean.

`GEN ser_normalize(GEN x)` divide x by its “leading term” so that the series is either 0 or equal to $t^v(1 + O(t))$. Shallow function if the “leading term” is 1.

`int ser_isexactzero(GEN x)` return 1 if x is a zero series, all of whose known coefficients are exact zeroes; this implies that $\text{sign}(x) = 0$ and $\text{lg}(x) \leq 3$.

`GEN ser_inv(GEN x)` return the inverse of the `t_SER` x using Newton iteration.

`GEN psi1series(long n, long v, long prec)` creates the `t_SER` $\psi(1 + x + O(x^n))$ in variable v .

10.9 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 cardinality 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`.

`long FF_var(GEN a)` returns the variable used to display `a`.

`GEN FF_gen(GEN a)` returns the standard generator of the definition field of the `t_FFELT` element `a`, see `ffgen`, that is $x \pmod{T}$ where T is the polynomial over the prime field that define the finite field.

`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 $\mathbf{Z}/p\mathbf{Z}$. Useful for generic code that wants to handle (inefficiently) $\mathbf{Z}/p\mathbf{Z}$ as if it were not a prime field.

`GEN Tp_to_FF(GEN T, GEN p)` returns a `t_FFELT` equal to 1 in the finite field $\mathbf{F}_p/(T)$, where T is a `ZX`, assumed to be irreducible modulo p , or `NULL` in which case the routine acts as `p_to_FF(p,0)`. No checks.

`GEN Fq_to_FF(GEN x, GEN ff)` returns a `t_FFELT` equal to x in the finite field defined by the `t_FFELT` `ff`, where x is an `Fq` (either a `t_INT` or a `ZX`: a `t_POL` with `t_INT` coefficients). No checks.

`GEN FqC_to_FFC(GEN x, GEN ff)` given an `FqC` x , return the `t_COL` with `t_FFELT` coefficients obtained by applying `Fq_to_FF` coefficientwise. No checks.

`GEN FqV_to_FFV(GEN x, GEN ff)` given an `FqV` x , return the `t_VEC` with `t_FFELT` coefficients obtained by applying `Fq_to_FF` coefficientwise. No checks.

`GEN FqM_to_FFM(GEN x, GEN ff)` given an `FqM` x , return the `t_MAT` with `t_FFELT` coefficients obtained by applying `Fq_to_FF` coefficientwise. No checks.

GEN FqX_to_FFX(GEN x, GEN ff) given an FqX x , return the polynomial with t_FFELT coefficients obtained by applying Fq_to_FF coefficientwise. No checks, and no normalization if the leading coefficient maps to 0.

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) returns 1 if the t_FFELT a is equal to 0 else returns 0.

int FF_equal1(GEN a) returns 1 if the t_FFELT a is equal to 1 else returns 0.

int FF_equalm1(GEN a) returns -1 if the t_FFELT a is equal to 1 else returns 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 $*ff = \text{NULL}$ (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_mul2n(GEN a, long n) returns a^{2^n} where a is a t_FFELT .

GEN FF_pow(GEN a, GEN n) returns a^n where a is a t_FFELT and n is a t_INT.

GEN FF_Frobenius(GEN a, GEN n) returns x^{p^n} where x is the standard generator of the definition field of the t_FFELT element a, t_FFELT, n is a t_INT, and p is the characteristic of the definition field of a.

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) return 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 $[a, a^p, a^{p^2}, \dots, a^{p^{n-1}}]$ 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 FF_map(GEN m, GEN a) returns $A(m)$ where A=a.pol assuming a and m belongs to fields having the same characteristic.

10.9.1 FFX.

The functions in this sections take polynomial arguments and a `t_FFELT` a . The coefficients of the polynomials must be of type `t_INT`, `t_INTMOD` or `t_FFELT` and compatible with a .

`GEN FFX_add(GEN P, GEN Q, GEN a)` returns the sum of the polynomials P and Q defined over the definition field of the `t_FFELT` a .

`GEN FFX_mul(GEN P, GEN Q, GEN a)` returns the product of the polynomials P and Q defined over the definition field of the `t_FFELT` a .

`GEN FFX_sqr(GEN P, GEN a)` returns the square of the polynomial P defined over the definition field of the `t_FFELT` a .

`GEN FFX_rem(GEN P, GEN Q, GEN a)` returns the remainder of the polynomial P modulo the polynomial Q , where P and Q are defined over the definition field of the `t_FFELT` a .

`GEN FFX_gcd(GEN P, GEN Q, GEN a)` returns the GCD of the polynomials P and Q defined over the definition field of the `t_FFELT` a .

`GEN FFX_extgcd(GEN P, GEN Q, GEN a, GEN *U, GEN *V)` returns the GCD of the polynomials P and Q defined over the definition field of the `t_FFELT` a and sets $*U$, $*V$ to the Bezout coefficients such that $*UP + *VQ = d$. If $*U$ is set to `NULL`, it is not computed which is a bit faster.

`GEN FFX_halfgcd(GEN x, GEN y, GEN a)` returns a two-by-two matrix 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 FFX_halfgcd_all(GEN x, GEN y, GEN a, GEN *ptA, GEN *ptB)` as `FFX_halfgcd`, in addition, if `ptA` (resp. `ptB`) is not `NULL`, `*ptA` (resp. `*ptB`) is set to A (resp. B).

`GEN FFX_resultant(GEN P, GEN Q, GEN a)` returns the resultant of the polynomials P and Q where P and Q are defined over the definition field of the `t_FFELT` a .

`GEN FFX_disc(GEN P, GEN a)` returns the discriminant of the polynomial P where P is defined over the definition field of the `t_FFELT` a .

`GEN FFX_isplayer(GEN P, ulong k, GEN a, GEN *py)` return 1 if the FFX P is a k -th power, 0 otherwise, where P is defined over the definition field of the `t_FFELT` a . If `py` is not `NULL`, set it to g such that $g^k = f$.

`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_factor_squarefree(GEN f, GEN a)` returns the squarefree factorization of the univariate polynomial f over the definition field of the `t_FFELT` a . This is a vector $[u_1, \dots, u_k]$ of pairwise coprime FFX such that $u_k \neq 1$ and $f = \prod u_i^i$.

`GEN FFX_ddf(GEN f, GEN a)` assuming that f is squarefree, returns the distinct degree factorization of f modulo p . The returned value `v` is a `t_VEC` with two components: `F=v[1]` is a vector of (FFX) factors, and `E=v[2]` is a `t_VECSMALL`, such that f is equal to the product of the `F[i]` and each `F[i]` is a product of irreducible factors of degree `E[i]`.

`GEN FFX_degfact(GEN f, GEN a)`, as `FFX_factor`, but the degrees of the irreducible factors are returned instead of the factors themselves (as a `t_VECSMALL`).

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 .

long FFX_nbroots(GEN f, GEN a) returns the number of roots 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_preimagerel(GEN F, GEN x, GEN a) returns $P\%F$ where $P=x.pol$ assuming a and x belongs to fields having the same characteristic, and that the coefficients of F belong to the definition field of a .

GEN FFX_preimage(GEN F, GEN x, GEN a) as FFX_preimagerel but return NULL if the remainder is of degree greater or equal to 1, the constant coefficient otherwise.

GEN FFV_roots_to_pol(GEN V, GEN ff, long v) V being a vector over the finite field given by ff (t_FFELT), returns the monic t_POL $\prod_i (pol_x[v] - V[i])$.

10.9.2 FFM.

GEN FFM_FFC_gauss(GEN M, GEN C, GEN ff) given a matrix M (t_MAT) and a column vector C (t_COL) over the finite field given by ff (t_FFELT) such that M is invertible, return the unique column vector X such that $MX = C$.

GEN FFM_FFC_invimage(GEN M, GEN C, GEN ff) given a matrix M (t_MAT) and a column vector C (t_COL) over the finite field given by ff (t_FFELT), return a column vector X such that $MX = C$, or NULL if no such vector exists.

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_deplin(GEN M, GEN ff) returns a nonzero vector (t_COL) in the kernel of the matrix M over the finite field given by ff , or NULL if no such vector exists.

GEN FFM_det(GEN M, GEN ff) returns the determinant of the matrix M over the finite field given by ff .

GEN FFM_gauss(GEN M, GEN N, GEN ff) given two matrices M and N (t_MAT) over the finite field given by ff (t_FFELT) such that M is invertible, return the unique matrix X such that $MX = N$.

GEN FFM_image(GEN M, GEN ff) returns a matrix whose columns span the image of the matrix M over the finite field given by ff .

GEN FFM_indexrank(GEN M, GEN ff) given a matrix M of rank r over the finite field given by ff , returns a vector with two $t_VECSMALL$ components y and z containing r row and column indices, respectively, such that the $r \times r$ -matrix formed by the $M[i,j]$ for i in y and j in z is invertible.

GEN FFM_inv(GEN M, GEN ff) returns the inverse of the square matrix M over the finite field given by ff , or NULL if M is not invertible.

GEN FFM_invimage(GEN M, GEN N, GEN ff) given two matrices M and N (t_MAT) over the finite field given by ff (t_FFELT), return a matrix X such that $MX = N$, or NULL if no such matrix exists.

GEN FFM_ker(GEN M, GEN ff) returns the kernel of the t_MAT M over the finite field given by the t_FFELT 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) returns the rank of the matrix M over the finite field given by ff.

GEN FFM_suppl(GEN M, GEN ff) given a matrix M over the finite field given by ff whose columns are linearly independent, returns a square invertible matrix whose first columns are those of M.

10.9.3 FFXQ.

GEN FFXQ_mul(GEN P, GEN Q, GEN T, GEN a) returns the product of the polynomials P and Q modulo the polynomial T, where P, Q and T are defined over the definition field of the `t_FFELT` a.

GEN FFXQ_sqr(GEN P, GEN T, GEN a) returns the square of the polynomial P modulo the polynomial T, where P and T are defined over the definition field of the `t_FFELT` a.

GEN FFXQ_inv(GEN P, GEN Q, GEN a) returns the inverse of the polynomial P modulo the polynomial Q, where P and Q are defined over the definition field of the `t_FFELT` a.

GEN FFXQ_minpoly(GEN Pf, GEN Qf, GEN a) returns the minimal polynomial of the polynomial P modulo the polynomial Q, where P and Q are defined over the definition field of the `t_FFELT` a.

10.10 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 bits 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 bit precision `ndec2prec(n)`.

10.10.1 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 cbrtr(GEN x) returns the real cube root of *x*.

GEN sqrtnr(GEN x, long n) returns the *n*-th root of *x*, assuming $n \geq 1$ and $x \geq 0$.

GEN sqrtnr_abs(GEN x, long n) returns the *n*-th root of $|x|$, assuming $n \geq 1$ and $x \neq 0$.

GEN mpcos(GEN x) returns $\cos(x)$.

GEN mpsin(GEN x) returns $\sin(x)$.

GEN mpsinc(GEN x) returns $\sin(x)/x$.

GEN mptan(GEN x) returns $\tan(x)$, assumes $\cos(x) \neq 0$.

GEN mpcotan(GEN x) returns $\cotan(x)$, assumes $\sin(x) \neq 0$.

GEN mpacos(GEN x) returns $\operatorname{acos}(x)$, assumes $|x| \leq 1$.

GEN `mpasin`(GEN `x`) returns $\operatorname{asin}(x)$, assumes $|x| \leq 1$.

GEN `mpatan`(GEN `x`) returns $\operatorname{atan}(x)$.

GEN `mpcosh`(GEN `x`) returns $\cosh(x)$.

GEN `mpsinh`(GEN `x`) returns $\sinh(x)$.

GEN `mptanh`(GEN `x`) returns $\tanh(x)$.

GEN `mpcotanh`(GEN `x`) returns $\operatorname{cotanh}(x)$, $x \neq 0$.

GEN `mpacosh`(GEN `x`) returns $\operatorname{acosh}(x)$, assumes $|x| \geq 1$.

GEN `mpasinh`(GEN `x`) returns $\operatorname{asinh}(x)$.

GEN `mpatanh`(GEN `x`) returns $\operatorname{atanh}(x)$, assumes $|x| < 1$.

GEN `mplog`(GEN `x`) 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 `mplog1p`(GEN `x`) returns $\log(1+x)$, computed in a way that is also accurate near $x = 0$.

GEN `mpexp`(GEN `x`) 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 $\operatorname{eint1}(x)$, for a `t_REAL` $x \neq 0$, assuming that `expx` is `mpexp(x)`.

A few variants on the Lambert function: they actually work when `gtofp` can map all GEN arguments to a `t_REAL`.

GEN `mplambertW`(GEN `y`) solution $x = W_0(y)$ of the implicit equation $x \exp(x) = y$, for $y > -1/e$ a `t_REAL`.

GEN `mplambertx_logx`(GEN `a`, GEN `b`, long `bit`) solve $x - a \log(x) = b$ with $a > 0$ and $b \geq a(1 - \log(a))$.

GEN `mplambertX`(GEN `y`, long `bit`) as `mplambertx_logx` in the special case $a = 1$, $b = \log(y)$. In other words, solve $e^x/x = y$ with $y \geq e$.

GEN `mplambertxlogx_x`(GEN `a`, GEN `b`, long `bit`) solve $x \log(x) - ax = b$; if $b < 0$, assume $a \geq 1 + \log|b|$.

Useful low-level functions which *disregard* the sign of x :

GEN `sqrtr_abs`(GEN `x`) returns $\sqrt{|x|}$ assuming $x \neq 0$.

GEN `cbrtr_abs`(GEN `x`) returns $|x|^{1/3}$ 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 $x \neq 0$.

10.10.2 Other complex transcendental functions.

GEN `atanhuu(ulong u, ulong v, long prec)` computes $\operatorname{atanh}(u/v)$ using binary splitting, assuming $0 < u < v$. Not stack-clean but suitable for `gc_upto`. The complexity is $O(\text{prec}/\log(v/u))$ so becomes impractical when v/u is too close to 1.

GEN `atanhui(ulong u, GEN v, long prec)` computes $\operatorname{atanh}(u/v)$ using binary splitting, assuming $0 < u < v$. Not stack-clean but suitable for `gc_upto`.

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 `veczeta(GEN a, GEN b, long N, long prec)` returns in a vector all the $\zeta(aj + b)$, where $j = 0, 1, \dots, N - 1$, where a and b are real numbers (of arbitrary type, although `t_INT` is treated more efficiently). Assumes that $N \geq 1$, $b > 1$ and $a(N - 1) + b > 1$ (in case $a < 0$); the silly case $a = 0$ is wasteful but allowed.

GEN `ggamma1m1(GEN x, long prec)` return $\Gamma(1 + x) - 1$ assuming $|x| < 1$. Guard against cancellation when x is small.

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 `t_REAL`

`void mpsinhcosh(GEN x, GEN *s, GEN *c)` sets s and c to $\sinh(x)$ and $\cosh(x)$ respectively, where x is a `t_REAL`

GEN `expIr(GEN x)` returns $\exp(ix)$, where x is a `t_REAL`. The return type is `t_COMPLEX` unless the imaginary part is equal to 0 to the current accuracy (its sign is 0).

GEN `expIPiQ(GEN x, long prec)` returns $\exp(i\pi x)$, where x is a rational number (`t_INT` or `t_FRAC`). Contrary to general transcendental functions, `expIPiQ` returns a result with exact components if possible. E.g, it returns `gen_1` or `gen_m1` when x is integral.

GEN `expIPiR(GEN x, long prec)` returns $\exp(i\pi x)$, where x is a real number (`t_INT`, `t_FRAC` or `t_REAL`). Calls `expIPiQ` for rational arguments and returns a result with exact components if possible.

GEN `expIPiC(GEN z, long prec)` return $\exp(i\pi x)$, where x is a complex number (`t_INT`, `t_FRAC`, `t_REAL` or `t_COMPLEX`). Calls `expIPiR` for real arguments and returns a result with exact components if possible.

GEN `expIxy(GEN x, GEN y, long prec)` returns $\exp(ixy)$. Efficient when x is real and y pure imaginary.

GEN `pow2Pis(GEN s, long prec)` returns $(2\pi)^s$. The intent of this function and the next ones is to be accurate even if s has a huge imaginary part: π is computed at an accuracy taking into account the cancellation induced by argument reduction when computing the sine or cosine of $\Im s \log 2\pi$.

GEN `powPis(GEN s, long prec)` returns π^s , as `pow2Pis`.

`long powcx_prec(long e, GEN s, long prec)` if $e \approx \log_2 |x|$ return the precision at which $\log(x)$ must be computed to evaluate x^s reliably (taking into account argument reduction).

GEN `powcx(GEN x, GEN logx, GEN s, long prec)` assuming s is a `t_COMPLEX` and `logx` is $\log(x)$ computed to accuracy `powcx_prec`, return x^s .

`void gsincos(GEN x, GEN *s, GEN *c, long prec)` general case.

`GEN rootsof1_cx(GEN d, long prec)` return $e(1/d)$ at precision `prec`, $e(x) = \exp(2i\pi x)$.

`GEN rootsof1u_cx(ulong d, long prec)` return $e(1/d)$ at precision `prec`.

`GEN rootsof1q_cx(long a, long b, long prec)` return $e(a/b)$ at precision `prec`.

`GEN rootsof1powinit(long a, long b, long prec)` precompute b -th roots of 1 for `rootsof1pow`, i.e. to later compute $e(ac/b)$ for varying c .

`GEN rootsof1pow(GEN T, long c)` given $T = \text{rootsof1powinit}(a, b, \text{prec})$, return $e(ac/b)$.

`void affgc(GEN x, GEN res)` assume `res` was allocated using `cgetc`, and that x is a complex number (`t_INT`, `t_FRAC`, `t_REAL` or `t_COMPLEX` with real components). Assign x to `res`. Generalizes `affgr`.

`GEN affc_fixlg(GEN x, GEN res)` assume `res` was allocated using `cgetc`, and that x is either a `t_REAL` or a `t_COMPLEX` with `t_REAL` components. Assign x to `res`, first shortening the components of `res` if needed (in a stack-safe way). Further convert `res` to a `t_REAL` if x is a `t_REAL`. Generalizes `affrr_fixlg`.

`GEN trans_eval(const char *fun, GEN (*f)(GEN, long), GEN x, long prec)` evaluate the transcendental function f (named "`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 C function handling only `t_REAL` and `t_COMPLEX` arguments. This routine first converts x to a suitable type:

- `t_INT/t_FRAC` to `t_REAL` of precision `prec`, `t_QUAD` to `t_REAL` or `t_COMPLEX` of precision `prec`.

- `t_POLMOD` to a `t_COL` of complex embeddings (as in `conjvec`)

Then evaluates the function at `t_VEC`, `t_COL`, `t_MAT` arguments coefficientwise.

`GEN trans_evalgen(const char *fun, void *E, GEN (*f)(void*, GEN, long), GEN x, long prec)`, general variant evaluating $f(E, x, \text{prec})$, where the function prototype allows to wrap an arbitrary context given by the argument E .

10.10.3 Modular functions.

`GEN cxredsl2(GEN z, GEN *g)` given t a `t_COMPLEX` belonging to the upper half-plane, find $\gamma \in \text{SL}_2(\mathbf{Z})$ such that $\gamma \cdot z$ belongs to the standard fundamental domain and set `*g` to γ .

`GEN cxredsl2_i(GEN z, GEN *g, GEN *czd)` as `cxredsl2`; also sets `*czd` to $cz+d$, if $\gamma = [a, b; c, d]$.

`GEN cxEk(GEN tau, long k, long prec)` returns $E_k(\tau)$ by direct evaluation of $1 + 2/\zeta(1-k) \sum_n n^{k-1} q^n / (1 - q^n)$, $q = e(\tau)$. Assume that $\Im \tau > 0$ and k even. Very slow unless τ is already reduced modulo $\text{SL}_2(\mathbf{Z})$. Not stack-clean but suitable for `gc_upto`.

10.10.4 Transcendental functions with `t_PADIC` arguments.

The argument x is assumed to be a `t_PADIC`.

GEN `Qp_exp`(GEN x) shortcut for `gexp(x, /*ignored*/prec)`

`long Qp_exp_prec`(GEN x) number of terms to sum in the $\exp(x)$ series to reach the same p -adic accuracy as $x \neq 0$. If $n = p - 1$, $e = v_p(x)$ and $b = \text{prec}_p(x)$, this is the ceiling of $nb/(ne - 1)$. Return -1 if the series does not converge ($ne \leq 1$).

GEN `Qp_gamma`(GEN x) shortcut for `ggamma(x, /*ignored*/prec)`

GEN `Qp_zeta`(GEN x) shortcut for `gzeta(x, /*ignored*/prec)`; assume that $x \neq 1$.

GEN `Qp_zetahurwitz`(GEN x , GEN y , long k) shortcut for `zetahurwitz(x, y, -k, /*ignored*/prec)`.

GEN `Qp_psi`(GEN x , long der) shortcut for `gpsi_der(x, der, /*ignored*/prec)`.

GEN `Qp_log`(GEN x) shortcut for `glog(x, /*ignored*/prec)`.

GEN `Qp_sqrt`(GEN x) shortcut for `gsqrt(x, /*ignored*/prec)` Return NULL if x is not a square.

GEN `Qp_sqrtn`(GEN x , GEN n , GEN $*z$) shortcut for `gsqrtn(x, n, z, /*ignored*/prec)`. Return NULL if x is not an n -th power.

GEN `Qp_agm2_sequence`(GEN a_1 , GEN b_1) assume $a_1/b_1 = 1 \pmod p$ if p odd and $\pmod{2^4}$ if $p = 2$. Let $A_1 = a_1/p^v$ and $B_1 = b_1/p^v$ with $v = v_p(a_1) = v_p(b_1)$; let further $A_{n+1} = (A_n + B_n + 2B_{n+1})/4$, $B_{n+1} = B_n \sqrt{A_n/B_n}$ (the square root of $A_n B_n$ congruent to $B_n \pmod p$) and $R_n = p^v(A_n - B_n)$. We stop when R_n is 0 at the given p -adic accuracy. This function returns in a triplet `t_VEC` the three sequences (A_n) , (B_n) and (R_n) , corresponding to a sequence of 2-isogenies on the Tate curve $y^2 = x(x - a_1)(x + a_1 - b_1)$. The common limit of A_n and B_n is the $M_2(a_1, b_1)$, the square of the p -adic AGM of $\sqrt{a_1}$ and $\sqrt{b_1}$. This is given by `ellQp_Ei` and is used by corresponding ascending and descending p -adic Landen transforms:

`void Qp_ascending_Landen`(GEN ABR , GEN $*ptx$, GEN $*pty$)

`void Qp_descending_Landen`(GEN ABR , GEN $*ptx$, GEN $*pty$)

10.10.5 Cached constants.

The cached constant is returned at its current precision, which may be larger than `prec`. One should always use the `mpxxx` variant: `mppi`, `mpeuler`, or `mplog2`.

GEN `consteuler`(long $prec$) precomputes Euler-Mascheroni's constant at precision $prec$.

GEN `constcatalan`(long $prec$) precomputes Catalan's constant at precision $prec$.

GEN `constpi`(long $prec$) precomputes π at precision $prec$.

GEN `constlog2`(long $prec$) precomputes $\log(2)$ at precision $prec$.

`void constbern`(long n) precomputes the n even Bernoulli numbers $B_2, \dots, B_{2n}$ as `t_FRAC`. No more than n Bernoulli numbers will ever be stored (by `bernfrac` or `bernreal`), unless a subsequent call to `constbern` increases the cache.

GEN `constzeta`(long n , long $prec$) ensures that the n values $\gamma, \zeta(2), \dots, \zeta(n)$ are cached at accuracy bigger than or equal to $prec$ and return a vector containing at least those value. Note

that $\gamma = \lim_1 \zeta(s) - 1/(s-1)$. If the accuracy of cached data is too low or n is greater than the cache length, the cache is recomputed at the given parameters.

The following functions use cached data if `prec` is smaller than the precision of the cached value; otherwise the newly computed data replaces the old cache.

GEN `mppi(long prec)` returns π at precision `prec`.

GEN `Pi2n(long n, long prec)` returns $2^n\pi$ at precision `prec`.

GEN `PiI2(long n, long prec)` returns the complex number $2\pi i$ at precision `prec`.

GEN `PiI2n(long n, long prec)` returns the complex number $2^n\pi i$ at precision `prec`.

GEN `mpeuler(long prec)` returns Euler-Mascheroni's constant at precision `prec`.

GEN `mpeuler(long prec)` returns Catalan's number at precision `prec`.

GEN `mplog2(long prec)` returns $\log 2$ at precision `prec`.

The following functions use the Bernoulli numbers cache initialized by `constbern`:

GEN `bernreal(long i, long prec)` returns the Bernoulli number B_i as a `t_REAL` at precision `prec`. If `constbern(n)` was called previously with $n \geq i$, then the cached value is (converted to a `t_REAL` of accuracy `prec` then) returned. Otherwise, the missing value is computed; the cache is not updated.

GEN `bernfrac(long i)` returns the Bernoulli number B_i as a rational number (`t_FRAC` or `t_INT`). If the `constbern` cache includes B_i , the latter is returned. Otherwise, the missing value is computed; the cache is not updated.

10.10.6 Obsolete functions.

void `mpbern(long n, long prec)`

10.11 Permutations .

Permutations are represented in two different ways

- (`perm`) a `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,

- (`cyc`) a `t_VEC` of `t_VECSMALL`s representing a product of disjoint cycles.

GEN `identity_perm(long n)` return the identity permutation on n symbols.

GEN `cyclic_perm(long n, long d)` return the cyclic permutation mapping i to $i + d \pmod{n}$ in S_n . Assume that $d \leq n$.

GEN `perm_mul(GEN s, GEN t)` multiply s and t (composition $s \circ t$)

GEN `perm_sqr(GEN s)` multiply s by itself (composition $s \circ s$)

GEN `perm_conj(GEN s, GEN t)` return sts^{-1} .

int `perm_commute(GEN p, GEN q)` return 1 if p and q commute, 0 otherwise.

GEN `perm_inv(GEN p)` returns the inverse of p .

GEN `perm_pow(GEN p, GEN n)` returns p^n

GEN perm_powu(GEN p, ulong n) returns p^n

GEN cyc_pow_perm(GEN p, long n) the permutation p is given as a product of disjoint cycles (cyc); return p^n (as a perm).

GEN cyc_pow(GEN p, long n) the permutation p is given as a product of disjoint cycles (cyc); return p^n (as a cyc).

GEN perm_cycles(GEN p) return the cyclic decomposition of p .

GEN perm_order(GEN p) returns the order of the permutation p (as the lcm of its cycle lengths).

ulong perm_orderu(GEN p) returns the order of the permutation p (as the lcm of its cycle lengths) assuming it fits in a ulong.

long perm_sign(GEN p) returns the sign of the permutation p .

GEN vecperm_orbits(GEN gen, long n) return the orbits of $\{1, 2, \dots, n\}$ under the action of the subgroup of S_n generated by gen .

GEN Z_to_perm(long n, GEN x) as numtoperm, returning a t_VECSMALL.

GEN perm_to_Z(GEN v) as permtonum for a t_VECSMALL input.

GEN perm_to_GAP(GEN p) return a t_STR which is a representation of p compatible with the GAP computer algebra system.

10.12 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 an 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 $[s_1, \dots, s_g]$ of grp expressed as a vector of permutations of length n .

$grp[2]$ contains the relative orders $[o_1, \dots, o_g]$ of the generators $grp[1]$.

or alternatively, if it is not WSS, by the sorted list of its elements.

See `galoisinit` for the technical details.

GEN checkgroup(GEN gal) check whether gal is a small group or a Galois group. Returns the underlying small group.

GEN checkgroupelts(GEN gal) check whether gal is a small group or a Galois group, or a vector of permutations listing the group elements. Returns the list of group elements as permutations.

GEN galois_group(GEN gal) return the underlying small group of the Galois group gal .

GEN cyclicgroup(GEN g, long s) return the cyclic group with generator g of order s .

GEN trivialgroup(void) return 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 *nontrivial* 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 groupeelts_gen(GEN elts), where $elts$ is the list of elements of a group, returns a small generating set.

GEN groupeelts_to_group(GEN elts), where $elts$ is the list of elements of a group, returns the corresponding small group.

GEN group_set(GEN grp, long n) returns a $F2v$ 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 groupeelts_set(GEN elts, long n), where $elts$ is the list of elements of a small group of domain n , returns a $F2v$ b such that $b[i]$ is set if and only if the small group contains a permutation sending 1 to i .

GEN groupeelts_conj_set(GEN elts, GEN p), where $elts$ is the list of elements of a small group of domain n , returns a $F2v$ b such that $b[i]$ is set if and only if the small group contains a permutation sending $p^{-1}[1]$ to $p^{-1}[i]$.

int group_subgroup_is_faithful(GEN G, GEN H) return 1 if the action of G on G/H by translation is faithful, 0 otherwise.

GEN groupeelts_conjclasses(GEN elts, long *pn), where $elts$ is the list of elements of a small group (sorted with respect to `vecsmall_lexcmp`), return a `t_VECSMALL` `conj` of the same length such that `conj[i]` is the index in $\{1, \dots, n\}$ of the conjugacy class of `elts[i]` for some unspecified but deterministic ordering of the classes, where n is the number of conjugacy classes. If `pn` is non NULL, `*pn` is set to n .

GEN conjclasses_repr(GEN conj, long nb), where `conj` and `nb` are as returned by the call `groupeelts_conjclasses(elts)`, return `t_VECSMALL` of length `nb` which gives the indices in `elts` of a representative of each conjugacy class.

GEN group_to_cc(GEN G), where G is a small group or a Galois group, returns a `cc` (conjugacy classes) structure `[elts,conj,rep,flag]`, as obtained by `alggroupcenter`, where `conj` is `groupeelts_conjclasses(elts)` and `rep` is the attached `conjclasses_repr`. `flag` is 1 if the permutation representation is transitive (in which case an element g of G is characterized by $g[1]$), and 0 otherwise. Shallow function.

long group_order(GEN grp) returns the order of the small group grp (which is the product of the relative orders).

long group_istrivial(GEN grp) returns 1 if the small group grp is of order 1, else 0.

long group_isabelian(GEN grp) returns 1 if the small group grp is Abelian, else 0.

long group_iswss(GEN grp) return 1 if the small group grp is WSS, 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 A_4 , 2 if it is isomorphic to S_4 , 3 if it is isomorphic to $(3 \times 3) : 4$ 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 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 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 $gGg^{-1} = G$, 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 $G \rightarrow G/H$ as an abstract data structure.

`GEN groupeelts_quotient(GEN elts, GEN H)`, where *elts* is the list of elements of a small group *G*, *H* is a subgroup of *G*, returns the quotient map $G \rightarrow G/H$ as an abstract data structure.

`GEN quotient_perm(GEN C, GEN g)` where *C* is the quotient map $G \rightarrow G/H$ 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 $G \rightarrow G/H$ for some *normal* subgroup *H* of *G*, return the quotient group G/H as a small group.

`GEN quotient_groupeelts(GEN C)` where *C* is the quotient map $G \rightarrow G/H$ for some group *G* and some *normal* subgroup *H* of *G*, return the list of elements of the quotient group G/H (as permutations over corresponding to the regular representation).

`GEN quotient_subgroup_lift(GEN C, GEN H, GEN S)` where *C* is the quotient map $G \rightarrow G/H$ 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 groupeelts_solvablesubgroups(GEN elts)` where *elts* is the list of elements of a finite group, returns the list of its solvable subgroups, each as a list of its elements.

`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 tableset_find_index(GEN tbl, GEN set)` searches the set *set* in the table *tbl* and returns its attached index, or 0 if not found.

`GEN groupeelts_abelian_group(GEN elts)` where *elts* is the list of elements of an *Abelian* small group, returns the corresponding small group.

`long groupeelts_exponent(GEN elts)` where *elts* is the list of elements of a small group, returns the exponent the group (the LCM of the order of the elements of the 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)` convert a small group to another format, as a `t_STR` describing the group for the given syntax, see `galoisexport`.

`GEN groupgen_export(GEN gen, long format)` convert a small group given by its generators to another format, as a `t_STR` describing the group for the given syntax, see `galoisexport`.

`long group_ident(GEN grp)` returns the index of the small group *grp* in the GAP4 Small Group library, see `galoisidentify`.

`long group_ident_trans(GEN grp)` returns the index of the regular representation of the small group *grp* in the GAP4 Transitive Group library, see `polgalois`.

Chapter 11:

Standard data structures

11.1 Character strings.

11.1.1 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* uordinal(ulong x)` return the ordinal number attached to *x* (i.e. 1st, 2nd, etc.) as a `stack_malloc`'ed string.

`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_raw(GEN x)` as `GENTostr` with the following differences: 1) the output format is `f_RAW`; 2) the result is allocated on the stack and *must not* be freed.

`char* GENTostr_unquoted(GEN x)` as `GENTostr_raw` with the following additional difference: a `t_STR` *x* is printed without enclosing quotes (to be used by `print`).

`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`, `f_PRETTYMAT` or `f_TEX`.

11.1.2 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 strexpend(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`

`GEN pari_base64(const char *s)` convert the string to base64 (RFC4648 "+" "/" with padding).

11.1.3 Dynamic strings.

A `pari_str` is a dynamic string which grows dynamically as needed. This structure contains private data and two public members `char *string`, which is the string itself and `use_stack` which tells whether the string lives

- on the PARI stack (value 1), meaning that it will be destroyed by any manipulation of the stack, e.g. a garbage collection or resetting `avma`;
- in malloc'ed memory (value 0), in which case it is impervious to stack manipulation but will need to be explicitly freed by the user after use, via `pari_free(s.string)`.

`void str_init(pari_str *S, int use_stack)` initializes a dynamic string; if `use_stack` is 0, then the string is malloc'ed, else it lives on the PARI stack.

`void str_printf(pari_str *S, const char *fmt, ...)` write to the end of `S` the remaining arguments according to PARI format `fmt`.

`void str_putc(pari_str *S, char c)` write the character `c` to the end of `S`.

`void str_puts(pari_str *S, const char *s)` write the string `s` to the end of `S`.

11.2 Output.

11.2.1 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()`

`int pari_last_was_newline(void)` returns a nonzero value if the last character printed via `out_putc` or `out_puts` was `\n`, and 0 otherwise.

`void pari_set_last_newline(int last)` sets the boolean value to be returned by the function `pari_last_was_newline` to *last*.

11.2.2 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 nontrivial 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

`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)`.

11.2.3 PARI colors. In this section we describe the low-level functions used to implement GP's color scheme, attached to the `colors` default. The following symbolic names are attached 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 nonempty 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.

`void term_color(long c)` prints (as if using `pari_puts`) the ANSI color escape sequence attached to output object `c`. If `c` is `c_NONE`, revert to default printing style.

`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 attached 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`).

11.2.4 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.

`void brute(GEN x, char format, long dec)`

`void matbrute(GEN x, char format, long dec)`

`void texe(GEN x, char format, long dec)`

11.3 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 the debug level for "files" is high enough.

`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 file, 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`.

11.3.1 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 the debug level for "files" 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_safeopen(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.

11.3.2 Temporary files.

PARI has its own idea of the system temp directory derived 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 and creates an empty file with that name. The name returned is malloc’ed.

`char* pari_unique_filename_suffix(const char *s, const char *suf)` analogous to above `pari_unique_filename`, creating a (previously nonexistent) tempfile whose name ends with suffix *suf*.

11.4 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.

11.4.1 Internal errors, “system” errors.

11.4.1.1 e_ARCH. A requested feature *s* is not available on this architecture or operating system.

```
pari_err(e_ARCH)
```

prints the error message: `sorry, 's' not available on this system.`

11.4.1.2 e_BUG. A bug in the PARI library, in function *s*.

```
pari_err(e_BUG, const char *s)
pari_err_BUG(const char *s)
```

prints the error message: `Bug in s, please report.`

11.4.1.3 e_FILE. Error while trying to open a file.

```
pari_err(e_FILE, const char *what, const char *name)
pari_err_FILE(const char *what, const char *name)
```

prints the error message: error opening *what*: '*name*'.

11.4.1.4 e_FILEDESC. Error while handling a file descriptor.

```
pari_err(e_FILEDESC, const char *where, long n)
pari_err_FILEDESC(const char *where, long n)
```

prints the error message: invalid file descriptor in *where*: '*name*'.

11.4.1.5 e_IMPL. A requested feature *s* is not implemented.

```
pari_err(e_IMPL, const char *s)
pari_err_IMPL(const char *s)
```

prints the error message: sorry, *s* is not yet implemented.

11.4.1.6 e_PACKAGE. Missing optional package *s*.

```
pari_err(e_PACKAGE, const char *s)
pari_err_PACKAGE(const char *s)
```

prints the error message: package *s* is required, please install it

11.4.2 Syntax errors, type errors.

11.4.2.1 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 nonsquare matrix.

```
pari_err(e_DIM, const char *s)
pari_err_DIM(const char *s)
```

prints the error message: inconsistent dimensions in *s*.

11.4.2.2 e_FLAG. A flag argument is out of bounds in function *s*.

```
pari_err(e_FLAG, const char *s)
pari_err_FLAG(const char *s)
```

prints the error message: invalid flag in *s*.

11.4.2.3 e_NOTFUNC. Generated by the PARI evaluator; tried to use a GEN which is not a `t_CLOSURE` in a function call syntax (as in `f = 1; f(2);`).

```
pari_err(e_NOTFUNC, GEN fun)
```

prints the error message: not a function in a function call.

11.4.2.4 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 $0(2) + 0(3)$: it is valid to add two `t_PADICs`, 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.

11.4.2.5 e_PRIORITY. object o in function s contains variables whose priority is incompatible with the expected operation. E.g. `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  $v$ , where  $v_o$  is gvar(o).`

11.4.2.6 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.

11.4.2.7 e_TYPE. An argument x of function s had an unexpected type. (As in `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 .

11.4.2.8 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 `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  $t_x$ ,  $t_y$` .

11.4.2.9 e_VAR. polynomials x and y submitted to function s have inconsistent variables. E.g., considering the algebraic number `Mod(t,t^2+1)` in `nfinite(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.

11.4.3 Overflows.

11.4.3.1 e_COMPONENT. Trying to access an inexistent component of a vector/matrix/list: the index is less than 1 or greater than the allowed length.

```
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: `nonexistent 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: `nonexistent component: index op lim`.

11.4.3.2 e_DOMAIN. An argument x is not in the function's domain (as in `moebius(0)` or `zeta(1)`).

```
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`.

11.4.3.3 e_MAXPRIME. A function using the precomputed list of prime numbers ran out of primes.

```
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 nonzero`. And simply `not enough precomputed primes` otherwise.

11.4.3.4 e_MEM. A call to `pari_malloc` or `pari_realloc` failed.

```
pari_err(e_MEM)
```

prints the error message: `not enough memory`.

11.4.3.5 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`.)

```
pari_err(e_OVERFLOW, const char *s)
pari_err_OVERFLOW(const char *s)
```

prints the error message: `overflow in s`.

11.4.3.6 e_PREC. Function s fails because input accuracy is too low. (As in `floor(1e100)` at default accuracy.)

```
pari_err(e_PREC, const char *s)
pari_err_PREC(const char *s)
```

prints the error message: `precision too low in s`.

11.4.3.7 e_STACK. The PARI stack overflows.

```
pari_err(e_STACK)
```

prints the error message: `the PARI stack overflows !` as well as some statistics concerning stack usage.

11.4.4 Errors triggered intentionally.

11.4.4.1 e_ALARM. A timeout, generated by the `alarm` function.

```
pari_err(e_ALARM, const char *fmt, ...)
```

prints the error message: `s`.

11.4.4.2 e_USER. A user error, as triggered by `error(g1, ..., gn)` in GP.

```
pari_err(e_USER, GEN g)
```

prints the error message: `user error:`, then the entries of the vector g .

11.4.5 Mathematical errors.

11.4.5.1 e_CONSTPOL. An argument of function s is a constant polynomial, which does not make sense. (As in `galoisinit(Pol(1))`.)

```
pari_err(e_CONSTPOL, const char *s)
pari_err_CONSTPOL(const char *s)
```

prints the error message: `constant polynomial in s`.

11.4.5.2 e_COPRIME. Function s expected two coprime arguments, and did receive x, y which were not.

```
pari_err(e_COPRIME, const char *s, GEN x, GEN y)
pari_err_COPRIME(const char *s, GEN x, GEN y)
```

prints the error message: `elements not coprime in s: x,y`.

11.4.5.3 e_INV. Tried to invert a noninvertible object x .

```
pari_err(e_INV, const char *s, GEN x)
pari_err_INV(const char *s, GEN x)
```

prints the error message: `impossible inverse in s: x`. If $x = \text{Mod}(a, b)$ is a `t_INTMOD` and a is not 0 mod b , this allows to factor the modulus, as $\text{gcd}(a, b)$ is a nontrivial divisor of b .

11.4.5.4 e_IRREDPOL. Function s expected an irreducible polynomial, and did not receive one. (As in `nfinit(x^2-1)`.)

```
pari_err(e_IRREDPOL, const char *s, GEN x)
pari_err_IRREDPOL(const char *s, GEN x)
```

prints the error message: `not an irreducible polynomial in s: x`.

11.4.5.5 e_MISC. Generic uncategorized error.

```
pari_err(e_MISC, const char *fmt, ...)
```

prints the error message: `s`.

11.4.5.6 e_MODULUS. moduli x and y submitted to function s are inconsistent. E.g., considering the algebraic number $\text{Mod}(t, t^2+1)$ in `nfinit(t^3-2)`.

```
pari_err(e_MODULUS, const char *s, GEN x, GEN y)
pari_err_MODULUS(const char *s, GEN x, GEN y)
```

prints the error message: `inconsistent moduli in s, then the moduli`.

11.4.5.7 e_PRIME. Function s expected a prime number, and did receive p , which was not. (As in `idealprimedec(nf, 4)`.)

```
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`.

11.4.5.8 e_ROOTS0. An argument of function *s* is a zero polynomial, and we need to consider its roots. (As in `polroots(0)`.)

```
pari_err(e_ROOTS0, const char *s)
pari_err_ROOTS0(const char *s)
```

prints the error message: zero polynomial in *s*.

11.4.5.9 e_SQRTN. Tried to compute an *n*-th root of *x*, which does not exist, in function *s*. (As in `sqrt(Mod(-1,3))`.)

```
pari_err(e_SQRTN, GEN x)
pari_err_SQRTN(GEN x)
```

prints the error message: not an *n*-th power residue in *s*: *x*.

11.4.6 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.

`int pari_err_display(GEN err)` displays the error message corresponding to `err`. Always return 0. Default value of the callback `cb_pari_err_handle`.

11.5 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**.

hashtable* hash_create_GEN(ulong size, long stack) special case when the keys are GEN with **==** (gidentical) equality test.

hashtable* hash_create_INT(ulong size, long stack) special case when the keys are **t_INT** with ordinary equality test.

hashtable* hash_create_ulong(ulong size, long stack) special case when the keys are **ulongs** with ordinary equality test.

hashtable* hash_create_str(ulong size, long stack) special case when the keys are character strings with string equality test (and **hash_str** hash function).

void hash_init(hashtable *h, ulong size, ulong (*hash)(void*), int (*eq)(void*, void*), use_stack) Initialize **h** for an hashtable with enough room to contain **size** entries of type **void***. The functions **eq** test keys for equality. If **use_stack** is non zero, the resulting table will use the PARI stack; otherwise, we use **malloc**.

void hash_init_GEN(hashtable *h, ulong size, int (*eq)(GEN, GEN), use_stack) Initialize **h** for an hashtable with enough room to contain **size** entries of type **GEN**. The functions **eq** test keys for equality. If **use_stack** is non zero, the resulting table will use the PARI stack; otherwise, we use **malloc**. The hash used is **hash_GEN**.

void hash_init_ulong(hashtable *h, ulong size, use_stack) Initialize **h** for an hashtable with enough room to contain **size** entries of type **ulong**. 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.

void hash_insert2(hashtable *h, void *k, void *v, ulong hash) as **hash_insert**, assuming **h->hash(k)** is **hash**.

`void hash_insert_long(hashtable *h, void *k, long v)` as `hash_insert` but `v` is a long.

`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_search2(hashtable *h, void *k, ulong hash)` as `hash_search` assuming $h \rightarrow \text{hash}(k)$ is `hash`.

`GEN hash_haskey_GEN(hashtable *h, void *k)` returns the associate value if the key k belongs to the hash, otherwise returns NULL.

`int hash_haskey_long(hashtable *h, void *k, long *v)` returns 1 if the key k belongs to the hash and set v to its value, otherwise returns 0.

`hashentry * hash_select(hashtable *h, void *k, void *E, int (*select)(void *, hashentry *))` variant of `hash_search`, useful when entries with identical keys are inserted: among the entries attached to key k , return one satisfying the selection criterion (such that `select(E,e)` is nonzero), or NULL if none exist.

`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 attached to k and v is not reclaimed.

`hashentry* hash_remove_select(hashtable *h, void *k, void *E, int(*select)(void*, hashentry *))` a variant of `hash_remove`, useful when entries with identical keys are inserted: among the entries attached to key k , return one satisfying the selection criterion (such that `select(E,e)` is nonzero) and delete it, or NULL if none exist. Only the linking structures are freed, memory attached to k and v is not reclaimed.

`GEN hash_keys(hashtable *h)` return in a `t_VECSMALL` the keys stored in hashtable h .

`GEN hash_keys_GEN(hashtable *h)` return in a `t_VEC` the keys stored in hashtable h (which are assumed to be GENs).

`GEN hash_values(hashtable *h)` return in a `t_VECSMALL` the values stored in hashtable h .

`GEN hash_values_GEN(hashtable *h)` return in a `t_VEC` the values stored in hashtable h (which are assumed to be GENs).

`void hash_destroy(hashtable *h)` deletes the hashtable, by removing all entries.

`void hash_dbg(hashtable *h)` print statistics for hashtable h , allows to evaluate the attached hash function performance on actual data.

Some interesting hash functions are available:

`ulong hash_str(const char *s)`

`ulong hash_str_len(const char *s, long len)` hash the prefix string containing the first `len` characters (assume `strlen(s) ≥ len`).

`ulong hash_GEN(GEN x)` generic hash function. Not suitable for cryptographic use. The value returned is different on 32 and 64 bit architectures but it does not depend on the multiprecision kernel (whether GMP is used or not).

`ulong hash_zv(GEN x)` hash a `t_VECSMALL`.

11.6 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.

11.6.1 Initialization.

To create a stack of objects of type `foo`, we proceed as follows:

```
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.

11.6.2 Adding elements. The following function pushes an element on the stack.

```
/* access globals t_foo and s_foo */
void push_foo(foo x)
{
    long n = pari_stack_new(&s_foo);
    t_foo[n] = x;
}
```

11.6.3 Accessing elements.

Elements are accessed naturally through the `t_foo` pointer. For example this function swaps two elements:

```
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;
}
```

11.6.4 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.

11.6.5 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 $0 \leq s.n \leq s.alloc$ 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 $s.alloc \geq s.n + nb$, 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 **`.

11.7 Vectors and Matrices.

11.7.1 Access and extract. See Section 9.2.1 and Section 9.2.2 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, all of them are shallow functions. 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 nonrecursive 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`, `t_COL` or `t_LIST`, concatenate its elements from left to right. Shallow version of `gconcat1`.

GEN `shallowmatconcat`(GEN v) shallow version of `matconcat`.

GEN `RgM_flatten_RgC`(GEN x) concatenates the columns of x in order. Shallow.

GEN `RgM_flatten_RgV`(GEN x) concatenates the rows of x in order. Shallow.

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 `RgV_F2v_extract_shallow`(GEN V , GEN x) extract components of the vector V whose indices corresponds to non-zero components of x . Shallow function.

GEN `shallowmatextract`(GEN M , GEN $l1$, GEN $l2$) extract components of the matrix M according to the `t_VECSMALL` $l1$ (list of lines indices) and $l2$ (list of columns indices). This is the shallow analog of `extract0`(x , $l1$, $l2$), 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 `matslice`(GEN A , long a , long b , long c , long d) returns the submatrix $A[a..b, c..d]$. Assume $a \leq b$ and $c \leq d$.

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 : $A[i, j1]$, $A[i, j1 + 1] \dots, A[i, j2]$. Assume $j1 \leq j2$.

GEN `rowcopy`(GEN A , long i) return the row $A[i,]$ of the `t_MAT` A . This function is stack-clean. See `row` for the shallow equivalent.

GEN `rowslice`(GEN A , long $i1$, long $i2$) return the `t_MAT` formed by the $i1$ -th through $i2$ -th rows of `t_MAT` A . Assume $i1 \leq i2$.

GEN `rowsplice`(GEN A , long i) return the `t_MAT` formed from the coefficients of `t_MAT` A with j -th row removed.

GEN `rowpermute`(GEN A , GEN p), p being a `t_VECSMALL` representing a list $[p_1, \dots, p_n]$ of rows of `t_MAT` A , returns the matrix whose rows are $A[p_1,], \dots, A[p_n,]$.

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 $A[j1], \dots, A[j2]$. If A is a `t_MAT`, these correspond to *columns* of A . The object returned has the same type as A (`t_VECSMALL`, `t_VEC`, `t_COL` or `t_MAT`). Assume $j1 \leq j2$ or $j2 = j1 - 1$ (return empty vector/matrix).

GEN `vecsplce`(GEN `A`, long `j`) return `A` with j -th entry removed (`t_VEC`, `t_COL`) or j -th column removed (`t_MAT`).

GEN `veclast`(GEN `A`) return the last entry of `A` (`t_VEC`, `t_COL`) or last column of `A` (`t_MAT`). Shallow, undefined if `lg(A)` is 1.

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 $A[n], \dots, A[1]$. If `A` is a `t_MAT`, these are the *columns* of `A`.

void `vecreverse_inplace`(GEN `A`) as `vecreverse`, but reverse `A` in place.

GEN `vecpermute`(GEN `A`, GEN `p`) `p` is a `t_VECSMALL` representing a list $[p_1, \dots, p_n]$ of indices. Returns a GEN which has the same type as `A` (`t_VEC`, `t_COL` or `t_MAT`), and whose components are $A[p_1], \dots, A[p_n]$. If `A` is a `t_MAT`, these are the *columns* of `A`.

GEN `vecsmallpermute`(GEN `A`, GEN `p`) as `vecpermute` when `A` is a `t_VECSMALL`.

GEN `vecslicpermute`(GEN `A`, GEN `p`, long `y1`, long `y2`) short for

`vecslice(vecpermute(A,p), y1, y2)`

(more efficient).

11.7.2 Componentwise operations.

The following convenience routines automate trivial loops of the form

for (`i` = 1; `i` < `lg(a)`; `i`++) `gel(v,i)` = `f(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 `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 `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 `gdiv(a[i], b[i])`.

GEN `vecsqr`(GEN `a`) returns the vector whose i -th component is `gsqr(a[i])`.

GEN `vecpow`(GEN `a`, GEN `n`). Given `n` a `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 ZV of the same length, returns the vector whose i -th component is `modii(a[i], b[i])`.

GEN `vecmoduu`(GEN `a`, GEN `b`). Assuming `a` and `b` are two `t_VECSMALL` of the same length, returns the vector whose i -th component is $a[i] \% b[i]$.

Note that `vecadd` or `vecsub` do not exist since `gadd` and `gsub` have the expected behavior. On the other hand, `ginv` does not accept vector types, hence `vecinv`.

11.7.3 Low-level vectors and columns functions.

These functions handle `t_VEC` as an abstract container type of GENs. No specific meaning is attached to the content. They accept both `t_VEC` and `t_COL` as input, but `col` functions always return `t_COL` and `vec` functions always return `t_VEC`.

Note. All the functions below are shallow.

GEN `const_col(long n, GEN x)` returns a `t_COL` of `n` components equal to `x`.

GEN `const_vec(long n, GEN x)` returns a `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_append(GEN V, GEN s)` append `s` to the vector `V`.

GEN `vec_prepend(GEN V, GEN s)` prepend `s` to the vector `V`.

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`.

GEN `vec_equiv(GEN O)` given a vector of objects `O`, return a vector with `n` components where `n` is the number of distinct objects in `O`. The i -th component is a `t_VECSMALL` containing the indices of the elements in `O` having the same value. Applied to the image of a function evaluated on some finite set, it computes the fibers of the function.

GEN `vec_reduce(GEN O, GEN *pE)` given a vector of objects `O`, return the vector `v` (of the same type as `O`) of *distinct* elements of `O` and set a `t_VECSMALL` `E` with the same length as `v`, such that `E[i]` is the multiplicity of object `v[i]` in the original `O`. Shallow function.

11.8 Vectors of small integers.

11.8.1 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_to_vec_inplace(GEN z)` apply `stoi` to all entries of `z` and set its type to `t_VEC`.

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` (merge sort).

`void vecsmall_sort(GEN v)` sorts the `t_VECSMALL v` in place (merge sort).

`GEN vecsmall_reverse(GEN v)` as `vecreverse` for a `t_VECSMALL v`.

`long vecsmall_max(GEN v)` returns the maximum of the elements of `t_VECSMALL v`, assumed nonempty.

`long vecsmall_indexmax(GEN v)` returns the index of the largest element of `t_VECSMALL v`, assumed nonempty.

`long vecsmall_min(GEN v)` returns the minimum of the elements of `t_VECSMALL v`, assumed nonempty.

`long vecsmall_indexmin(GEN v)` returns the index of the smallest element of `t_VECSMALL v`, assumed nonempty.

`int vecsmall_isconst(GEN v)` Returns 1 if all the components of `v` are equal, else returns 0.

`int vecsmall_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.

`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.

`GEN vecsmall_prod(GEN v)` given a `t_VECSMALL v`, return the product of its entries.

The following sorting functions assume all entries of the `t_VECSMALL v` satisfy $0 \leq v[i] \leq M$ and use a counting sort, in linear time $O(\#v + M)$:

`void vecsmall_counting_sort(GEN v, long M)` sorts the `t_VECSMALL v` in place.

`GEN vecsmall_counting_indexsort(GEN v, long M)` as `vecsmall_indexsort` using a counting sort.

`GEN vecsmall_counting_uniq(GEN v, long M)` as `vecsmall_uniq` using a counting sort.

11.8.2 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_shallow(GEN x)`, shallow variant of `vecvecsmall_sort`.

`void vecvecsmall_sort_inplace(GEN x, GEN *perm)` sort lexicographically `x` in place, without copying its components. If `perm` is not `NULL`, it is set to the permutation that would sort the original `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)` `x` being a sorted `vecvecsmall` and `y` a `t_VECSMALL`, search `y` inside `x`.

`GEN vecvecsmall_max(GEN x)` returns the largest entry in all $x[i]$, assumed nonempty. Shallow function.

Chapter 12:

Functions related to the GP interpreter

12.1 Handling closures.

12.1.1 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 $\geq n$.

`GEN closure_callgenvec(GEN C, GEN args)` evaluate the `t_CLOSURE` `C` with the arguments supplied in the vector `args`. Assumes `C` has arity $\geq \lg(\text{args}) - 1$.

`GEN closure_callgenvecprec(GEN C, GEN args, long prec)` as `closure_callgenvec` but set the precision locally to `prec`.

`GEN closure_callgenvecdef(GEN C, GEN args, GEN def)` evaluate the `t_CLOSURE` `C` with the arguments supplied in the vector `args`, where the `t_VECSMALL` `def` indicates which arguments are actually present. Assumes `C` has arity $\geq \lg(\text{args}) - 1$.

`GEN closure_callgenvecdefprec(GEN C, GEN args, GEN def, long prec)` as `closure_callgenvecdef` but set the precision locally to `prec`.

`GEN closure_callgen0(GEN C, long prec)` evaluate the `t_CLOSURE` `C` without arguments.

`GEN closure_callgen0prec(GEN C, long prec)` evaluate the `t_CLOSURE` `C` without arguments, but set the precision locally to `prec`.

`GEN closure_callgen1(GEN C, GEN x)` evaluate the `t_CLOSURE` `C` with argument `x`. Assumes `C` has arity ≥ 1 .

`GEN closure_callgen1prec(GEN C, GEN x, long prec)` as `closure_callgen1`, but set the precision locally to `prec`.

`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 an inline closure and returns the result, or `NULL` if a flow change occurred.

`GEN closure_evalnoprk(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 an inline 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 an inline 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 nonzero value if a flow change occurred. This variant is not stack clean. Used by the break loop.

12.1.2 Control flow functions.

`GEN derivnum0(GEN a, GEN code, GEN ind, long prec)`

`GEN direuler0(GEN a, GEN b, GEN code, GEN c)`

`GEN dirpowerssum0(GEN N, GEN s, GEN code, long both, long prec)`

`void fordiv(GEN a, GEN code)`

`void fordivfactored(GEN a, GEN code)`

`void foreachpari(GEN a, GEN code)`

`void forell0(long a, long b, GEN code, long flag)`

`void forfactored(GEN a, GEN b, GEN code)`

`void forpari(GEN a, GEN b, GEN code)`

`void forpart0(GEN k, GEN code, GEN nbound, GEN abound)`

`void forperm0(GEN k, GEN code)`

`void forsquarfree(GEN a, GEN b, GEN code)`

`void forstep(GEN a, GEN b, GEN s, GEN code)`

`void forsubgroup0(GEN cyc, GEN bound, GEN code)`

`void forsubset0(GEN nk, GEN code)`

`GEN iferrpari(GEN a, GEN b, GEN c)`

`GEN ifpari(GEN g, GEN a, GEN b)`

`GEN intcirc0(GEN a, GEN R, GEN code, GEN tab, long prec)`

`GEN intfuncinit0(GEN a, GEN b, GEN code, long m, long prec)`

`GEN intnum0(GEN a, GEN b, GEN code, GEN tab, long prec)`

```

GEN laurentseries0(GEN code, long M, long v, long prec)
void pari_breakpoint(void)
void pariplot0(GEN a, GEN b, GEN code, GEN ysmlu, GEN ybigu, long prec)
GEN ploth0(GEN a, GEN b, GEN code, long flag, long n, long prec)
GEN plothexport0(GEN fmt, GEN a, GEN b, GEN code, long flags, long n, long prec)
GEN prodeuler0(GEN a, GEN b, GEN code, long prec)
GEN prodinfo0(GEN a, GEN code, long flag, long prec)
GEN solvestep0(GEN a, GEN b, GEN step, GEN code, long flag, long prec)
GEN sumalt0(GEN a, GEN code, long flag, long prec)
GEN sumdivmultexpr0(GEN num, GEN code)
GEN suminfo0(GEN a, GEN code, long bit)
GEN sumnum0(GEN a, GEN code, GEN tab, long prec)
GEN sumnumap0(GEN a, GEN code, GEN tab, long prec)
GEN sumnumlagrange0(GEN a, GEN code, GEN tab, long prec)
GEN sumpos0(GEN a, GEN code, long flag, long prec)
void untilpari(GEN a, GEN code)
void whilepari(GEN a, GEN code)
GEN zbrent0(GEN a, GEN b, GEN code, long prec)

```

12.1.3 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.

12.1.4 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 $-1, -2, \dots$

`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 .

12.1.5 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 `closure_derivn(GEN code, long n)` returns a closure corresponding to the numerical derivative of order $n > 0$ of the closure `code`.

GEN `closuretoinl(GEN code)` convert the closure `code` to a pseudo-inline closure suitable for use with prototype codes I and E.

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 $g(x_1, \dots, x_n) = f(x_1, \dots, x_n, gel(data, 1), \dots, gel(data, m))$. If `data` is `NULL`, then $m = 0$ is assumed. Shallow function.

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`. This is analogous to `snm_closure(is_entry(str), mkvecn(...))` but the latter has lower overhead since it does not copy arguments, nor does it validate inputs.

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);
```

12.1.6 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(long level)` prints a backtrace of the last 20 stack frames, starting at frame `level`, the numbering starting at 0.

12.1.7 Standard wrappers for iterators. Two families of standard wrappers are provided to interface iterators like `intnum` or `sumnum` with GP.

12.1.7.1 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'.

GEN `gp_evalprec(void *E, GEN x, long prec)` as `gp_eval`, but set the precision locally to `prec`.

`long gp_evalvoid(void *E, GEN x)` is used for the prototype code 'I'. The resulting value is discarded. Return a nonzero 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 iff the value is nonzero).

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).

12.1.7.2 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 .

`GEN gp_callprec(void *E, GEN x, long prec)` as `gp_call`, but set the precision locally to `prec`.

`GEN gp_call2(void *E, GEN x, GEN y)` evaluates the closure (GEN)E on (x, y) .

`long gp_callbool(void *E, GEN x)` evaluates the closure (GEN)E on x , returns 1 if its result is nonzero, and 0 otherwise.

`long gp_callvoid(void *E, GEN x)` evaluates the closure (GEN)E on x , discarding the result. Return a nonzero value if a control-flow instruction request the iterator to terminate immediately.

12.2 Defaults.

`entree* pari_is_default(const char *s)` return the `entree` structure attached to s if it is the name of a default, NULL 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_breakloop(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_doctest(const char *v, long flag)`

`GEN sd_echo(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_factorlimit(const char *v, long flag)
GEN sd_format(const char *v, long flag)
GEN sd_graphcolormap(const char *v, long flag)
GEN sd_graphcolors(const char *v, long flag)
GEN sd_help(const char *v, long flag)
GEN sd_histfile(const char *v, long flag)
GEN sd_histsize(const char *v, long flag)
GEN sd_lines(const char *v, long flag)
GEN sd_linewrap(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_parisizemax(const char *v, long flag)
GEN sd_path(const char *v, long flag)
GEN sd_plothsizes(const char *v, long flag)
GEN sd_prettyprinter(const char *v, long flag)
GEN sd_primelimit(const char *v, long flag)
GEN sd_prompt(const char *v, long flag)
GEN sd_prompt_cont(const char *v, long flag)
GEN sd_psfile(const char *v, long flag) The psfile default is obsolete, don't use this func-
tion.
GEN sd_readline(const char *v, long flag)
GEN sd_realbitprecision(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(const char *v, int flag)
GEN sd_strictargs(const char *v, long flag)
GEN sd_strictmatch(const char *v, long flag)
GEN sd_timer(const char *v, long flag)
GEN sd_threadsize(const char *v, long flag)
GEN sd_threadsizemax(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 acknowledgements.

```

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 nonempty).

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 nonempty).

- **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** ($\times 10^3$), **m/M** ($\times 10^6$), or **g/G** ($\times 10^9$) 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 attached to the value *i* of the default. The last entry in the **msg** array is used as a message attached to all subsequent ones.

The exact behavior and return value depends on **flag**:

- **d_RETURN**: returns the new value, as a **GEN**.

- `d_ACKNOWLEDGE`: prints a message indicating the new value, possibly a message attached to it via the `msg` argument, and return `gnil`.

- other cases: print nothing and return `gnil`.

GEN `sd_intarray(const char *v, long flag, const char *s, GEN *pz)`

- records a `t_VECSMALL` array of nonnegative integers.
- `pz` points to the current `t_VECSMALL` value, and is set to the new value (when `v` is nonempty).

The exact return value depends on `flag`:

- `d_RETURN`: returns the new value, as a `t_VEC` (converted via `zv_to_ZV`)
- `d_ACKNOWLEDGE`: prints a message indicating the new value, (as a `t_VEC`) 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 nonempty).

12.3 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 nonnegative.

After `S = obj_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 garbage collections. 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 `gcopy` or `gc_upto`), rather than having to deal with individual components of actual C `structs`.

`GEN obj_reinit(GEN S)` make a shallow copy of S , re-initializing all dynamic components. This allows “forking” a lazy vector while avoiding both a memory leak, and storing pointers to the same data in different objects (with risks of a double free later).

`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_realprec(GEN S, long tag, GEN (*build)(GEN, long), long prec)` if the *tag*-component of S is non empty *and* satisfies `gprecision` $\geq$ `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), GEN (*gpr)(GEN), long prec)` if the *tag*-component of S is non empty *and* has precision `gpr(x)` $\geq$ `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 13:

Algebraic Number Theory

13.1 General Number Fields.

13.1.1 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 $\mathbf{Z}$ -module generated by its columns may not be an $\mathbf{Z}_K$ -module (i.e. the expensive `nfisideal` routine will *not* be called).

`long nftyp(GEN x)`. Returns the type of number field structure stored in `x`, `typ_NF`, `typ_BNF`, or `typ_BNR`. Other answers are possible, meaning `x` is not a number field structure.

`GEN get_nf(GEN x, long *t)`. Extract an *nf* structure from `x` if possible and return it, otherwise return `NULL`. Sets `t` to the `nftyp` of `x` in any case.

`GEN get_bnf(GEN x, long *t)`. Extract a *bnf* structure from `x` if possible and return it, otherwise return `NULL`. Sets `t` to the `nftyp` of `x` in any case.

`GEN get_nfpol(GEN x, GEN *nf)` try to extract an *nf* structure from `x`, and sets `*nf` to `NULL` (failure) or to the *nf*. Returns the (monic, integral) polynomial defining the field.

`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 `NULL` (failure) or to the corresponding structure. Returns the (monic, integral) polynomial defining the field.

`GEN checknf(GEN x)` if an *nf* structure can be extracted from `x`, return it; otherwise raise an exception. The more general `get_nf` is often more flexible.

`GEN checkbnf(GEN x)` if an *bnf* structure can be extracted from `x`, return it; otherwise raise an exception. The more general `get_bnf` is often more flexible.

`GEN checkbnf_i(GEN bnf)` same as `checkbnf` but return `NULL` instead of raising an exception.

`void checkbnr(GEN bnr)` Raise an exception if the argument is not a *bnr* structure.

`GEN checkbnr_i(GEN bnr)` same as `checkbnr` but returns the *bnr* or `NULL` instead of raising an exception.

`GEN checknf_i(GEN nf)` same as `checknf` but return `NULL` instead of raising an exception.

`void checkrnf(GEN rnf)` Raise an exception if the argument is not an *rnf* structure.

`int checkrnf_i(GEN rnf)` same as `checkrnf` but return 0 on failure and 1 on success.

`void checkbid(GEN bid)` Raise an exception if the argument is not a *bid* structure.

`GEN checkbid_i(GEN bid)` same as `checkbid` but return `NULL` instead of raising an exception and return `bid` on success.

`GEN checkznstar_i(GEN G)` return G if it is a *znstar*; else return NULL on failure.

`GEN checkgal(GEN x)` if a *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 checkprid(GEN pr)` Raise an exception if the argument is not a prime ideal structure.

`int checkprid_i(GEN pr)` same as `checkprid` but return 0 instead of raising an exception and return 1 on success.

`int is_nf_factor(GEN F)` return 1 if F is an ideal factorization and 0 otherwise.

`int is_nf_extfactor(GEN F)` return 1 if F is an extended ideal factorization (allowing 0 or negative exponents) and 0 otherwise.

`int RgV_is_prV(GEN v)` returns 1 if the vector v contains only prime ideals and 0 otherwise.

`GEN get_prid(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.

`GEN cyc_get_expo(GEN cyc)` return the exponent of the group with structure *cyc*; 0 for an infinite group.

`void checkmodpr(GEN modpr)` Raise an exception if the argument is not a `modpr` structure (from `nfmodprinit`).

`GEN get_modpr(GEN x)` return x if it is a `modpr` structure and NULL otherwise.

`GEN checknfelt_mod(GEN nf, GEN x, const char *s)` given an *nf* structure *nf* and a `t_POLMOD` x , return the attached 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.

`int check_ZKmodule_i(GEN x)` return 1 if x looks like a projective $\mathbf{Z}_K$ -module, i.e., a pair $[A, I]$ where A is a matrix and I is a list of ideals and A has as many columns as I has elements. Or possibly a longer list $[A, I, \dots]$ such as the output of `rnfpsudobasis`. Otherwise return 0.

`void check_ZKmodule(GEN x, const char *s)` raise an exception unless x is recognized as a projective $\mathbf{Z}_K$ -module. Set *s* to the name of the caller for a meaningful error message.

`long idealtyp(GEN *ideal, GEN *fa)` The input is *ideal*, a pointer to an ideal or extended ideal; returns the type of the underlying ideal among `id_PRINCIPAL` (a number field element), `id_PRIME` (a prime ideal) `id_MAT` (an ideal in matrix form).

As a first side effect, **ideal* is set to the underlying ideal, possibly simplified (for instance the zero ideal represented by an empty matrix is replaced by `gen_0`).

If *fa* is not NULL, then **fa* is set to the extended part in the input: either NULL (regular ideal) or the extended part of an extended ideal.

13.1.2 Extracting info from a `nf` structure.

These functions expect a true `nf` argument attached to a number field $K = \mathbf{Q}[x]/(T)$, e.g. a `bnf` will not work. Let $n = [K : \mathbf{Q}]$ be the field degree.

`GEN nf_get_pol(GEN nf)` returns the polynomial T (monic, in $\mathbf{Z}[x]$).

`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 $(1, x, \dots, x^{n-1})$ in the maximal order of K .

`GEN nf_get_zk(GEN nf)` returns a basis $(w_1, w_2, \dots, w_n)$ for the maximal order of K . Those are polynomials in $\mathbf{Q}[x]$ of degree $< n$; it is guaranteed that $w_1 = 1$.

`GEN nf_get_zkden(GEN nf)` returns the denominator of `nf_get_zk`, as a positive `t_INT`.

`GEN nf_get_zkprimpart(GEN nf)` returns `nf_get_zk` times its denominator.

`GEN nf_get_invzk(GEN nf)` returns the matrix $(m_{i,j}) \in M_n(\mathbf{Z})$ 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 `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 `t_REALs`), then the r_2 representatives of the pairs of complex conjugates.

`GEN ZX_nf_roots(GEN T, long r1, long prec)` computes the same $r_1 + r_2$ roots as `nf_get_roots` from scratch, starting with a squarefree `ZX T`, assuming it has exactly r_1 real roots. Note that `QX_complex_roots` returns all $r_1 + 2r_2$ complex roots.

`GEN nf_get_allroots(GEN nf)` returns all the complex roots of T : first the r_1 real roots (as `t_REALs`), 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 α_i is the i -th element of `nf_get_roots(nf)`. In particular, if v is an n -th dimensional `t_COL` representing the element $\sum_{i=1}^n v[i] w_i$ of K , then `RgM_RgC_mul(M, v)` represents the embeddings of v .

`GEN nf_get_G(GEN nf)` returns a $n \times n$ real matrix G such that $Gv \cdot Gv = T_2(v)$, where v is an n -th dimensional `t_COL` representing the element $\sum_{i=1}^n v[i] w_i$ of K and T_2 is the standard Euclidean form on $K \otimes \mathbf{R}$, i.e. $T_2(v) = \sum_{\sigma} |\sigma(v)|^2$, where σ 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 `RM_round_maxrank(G)`.

`GEN nf_get_ramified_primes(GEN nf)` returns the vector of ramified primes.

`GEN nf_get_Tr(GEN nf)` returns the matrix of the Trace quadratic form on the basis $(w_1, \dots, w_n)$: its (i, j) entry is $\text{Tr} w_i w_j$.

`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.

13.1.3 Extracting info from a *bnf* structure.

These functions expect a true *bnf* argument, e.g. a *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) $[d_1, \dots, d_k]$, where $d_k \mid \dots \mid d_1$.

`GEN bnf_get_gen(GEN bnf)` returns the generators $[g_1, \dots, g_k]$ 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 $d_i g_i = 0$.

`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 *bnf* does not contain units in algebraic form.

`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 $\mathbf{Z}_K^*$.

`long bnf_get_tuN(GEN bnf)` returns the order of the torsion part of $\mathbf{Z}_K^*$, i.e. the number of roots of unity in K .

`GEN bnf_get_sunits(GEN bnf)` allows access to the algebraic data stored by `bnfinit(,1)`. The function returns NULL unless the *bnf* was initialized by `bnfinit(,1)`, else a vector $[X, U, E, \text{lim}]$ where

- X is a vector of rational primes and algebraic integers all of whose prime divisors have norm less than `lim`,

- U is a matrix of exponents whose columns yield the fundamental units `bnf.fu`. More precisely,

$$\text{bnf.fu}[j] = \prod_i X[i]^{U[i,j]}.$$

- E is a matrix of exponents whose columns yield the generators of principal ideals attached to the HNF of the *bnf* relation matrix between the maximal ideals of norm less `lim` (that generate the class group under GRH). More precisely, `bnf[5]` contains the prime factor base P (its first r elements being independant class group generators), `bnf[1]` contains a matrix W in HNF in $M_r(\mathbf{Z})$ and `bnf[2]`, contains a matrix B in $M_{r \times c}(\mathbf{Z})$. We define algebraic numbers e_j for $j \leq r + c$ such that

$$\prod_{i \leq r} P_i^{W[i,j]} = (e_j), \quad j \leq r$$

$$P_j \prod_{i \leq r} P_i^{B[i,j]} = (e_j), \quad j > r$$

Then $e_j = \prod_i X[i]^{E[i,j]}$.

`GEN bnf_has_fu(GEN bnf)` return fundamental units in expanded form if `bnf` contains them. Else return `NULL`.

`GEN bnf_compactfu(GEN bnf)` return fundamental units as a vector of algebraic numbers in compact form if `bnf` contains them. Else return `NULL`.

`GEN bnf_compactfu_mat(GEN bnf)` as a pair (X, U) , where X is a vector of S -units and U is a matrix with integer entries (without 0 rows), see `bnf_get_sunits`, if `bnf` contains them. Else return `NULL`.

13.1.4 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) $[d_1, \dots, d_k]$, where $d_k \mid \dots \mid d_1$.

`GEN bnr_get_gen(GEN bnr)` returns the generators $[g_1, \dots, g_k]$ 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 $d_i g_i = 0$. 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* attached to the *bnr* modulus.

`GEN bnr_get_mod(GEN bnr)` returns the modulus attached to the *bnr*.

13.1.5 Extracting info from an rnf structure.

These functions expect a true *rnf* argument, attached to an extension L/K , $K = \mathbf{Q}[y]/(T)$, $L = K[x]/(P)$.

`long rnf_get_degree(GEN rnf)` returns the *relative* degree $[L : K]$.

`long rnf_get_absdegree(GEN rnf)` returns the absolute degree $[L : \mathbf{Q}]$.

`long rnf_get_nfdegree(GEN rnf)` returns the degree of the base field $[K : \mathbf{Q}]$.

`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 attached to the base field K .

`GEN rnf_get_nfzk(GEN rnf)` 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 attached to L .

`GEN rnf_get_zk(GEN nf)` returns the relative integer basis generating $\mathbf{Z}_L$ as a $\mathbf{Z}_K$ -module, as a pseudo-matrix (A, I) in HNF.

`GEN rnf_get_disc(GEN rnf)` is the output $[\mathfrak{d}, s]$ of `rnfdisc`.

`GEN rnf_get_ramified_primes(GEN rnf)` returns the vector of rational primes below ramified primes in the relative extension, i.e. all prime numbers appearing in the factorization of

`ideallnorm(rnf_get_nf(rnf), rnf_get_disc(rnf));`

`GEN rnf_get_idealdisc(GEN rnf)` is the ideal discriminant $\mathfrak{d}$ from `rnfdisc`.

`GEN rnf_get_index(GEN rnf)` is the index ideal $\mathfrak{f}$

`GEN rnf_get_polabs(GEN rnf)` returns an absolute polynomial defining $L/\mathbf{Q}$.

`GEN rnf_get_alpha(GEN rnf)` a root α of the polynomial defining the base field, modulo `polabs` (cf. `rnfequation`)

`GEN rnf_get_k(GEN rnf)` a small integer k such that $\theta = \beta + k\alpha$ is a root of `polabs`, where β is a root of `pol` and α a root of the polynomial defining the base field, as in `rnf_get_alpha` (cf. also `rnfequation`).

`GEN rnf_get_invzk(GEN rnf)` contains A^{-1} , where (A, I) is the chosen pseudo-basis for $\mathbf{Z}_L$ over $\mathbf{Z}_K$.

`GEN rnf_get_map(GEN rnf)` returns technical data attached to the map $K \rightarrow L$. Currently, this contains data from `rnfequation`, as well as the polynomials T and P .

13.1.6 Extracting info from a bid structure.

These functions expect a true *bid* argument, attached to a modulus $I = I_0 I_\infty$ in a number field K . The underlying abelian group is $G = (\mathbf{Z}_K/I)^*$. Not that if the *bid* was initialized by `Idealstarmod` with a non-NULL `cycmod` argument, computations take place in G/G^{cycmod} instead.

`GEN bid_get_mod(GEN bid)` returns the modulus attached to the *bid*.

`GEN bid_get_MOD(GEN bid)` returns the integer `cycmod` given as argument to `Idealstarmod`, or NULL if we used `Idealstar` to initialize *bid*.

`GEN bid_get_grp(GEN bid)` returns the abelian group attached to $(\mathbf{Z}_K/I)^*$.

`GEN bid_get_ideal(GEN bid)` return the finite part I_0 of the *bid* modulus (an integer ideal).

`GEN bid_get_arch(GEN bid)` return the Archimedean part I_∞ of the *bid* modulus as a vector of real places in `vec01` format, see Section 13.1.20.

`GEN bid_get_archp(GEN bid)` return the Archimedean part I_∞ of the *bid* modulus, as a vector of real places in indices format see Section 13.1.20.

`GEN bid_get_fact(GEN bid)` returns the ideal factorization $I_0 = \prod_i \mathfrak{p}_i^{e_i}$.

`GEN bid_get_fact2(GEN bid)` as `bid_get_fact` with all factors $\mathfrak{p}^1$ with $\mathfrak{p}$ of norm 2 removed from the factorization. (They play no role in the structure of $(\mathbf{Z}_K/I)^*$, except that the generators must be made coprime to them.)

`bid_get_ideal(bid)`, via `idealfactor`.

`GEN bid_get_no(GEN bid)` returns the cardinality of the group $(\mathbf{Z}_K/I)^*$.

`GEN bid_get_cyc(GEN bid)` returns the elementary divisors of the group $(\mathbf{Z}_K/I)^*$ (cyclic components) $[d_1, \dots, d_k]$, where $d_k \mid \dots \mid d_1$.

`GEN bid_get_gen(GEN bid)` returns the generators of $(\mathbf{Z}_K/I)^*$ 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!

`GEN bid_get_sprk(GEN bid)` return a list of structures attached to the $(\mathbf{Z}_K/\mathfrak{p}^e)^*$ where $\mathfrak{p}^e$ divides I_0 exactly.

`GEN bid_get_sarch(GEN bid)` return the structure attached to $(\mathbf{Z}_K/I_\infty)^*$, by `nfarchstar`.

`GEN bid_get_U(GEN bid)` return the matrix with integral coefficients relating the local generators (from chinese remainders) to the global SNF generators (*bid.gen*).

13.1.7 Extracting info from a znstar structure.

These functions expect an argument G as returned by `znstar0(N, 1)`, attached to a positive N and the abelian group $(\mathbf{Z}/N\mathbf{Z})^*$. Let (g_i) be the SNF generators, where g_i has order d_i ; we call (g'_i) the (canonical) Conrey generators, where g'_i has order d'_i . Both sets of generators have the same cardinality.

`GEN znstar_get_N(GEN bid)` return N .

`GEN znstar_get_faN(GEN G)` return the factorization `factor(N)`, $N = \prod_j p_j^{e_j}$.

`GEN znstar_get_pe(GEN G)` return the vector of primary factors $(p_j^{e_j})$.

`GEN znstar_get_no(GEN G)` the cardinality $\phi(N)$ of G .

`GEN znstar_get_cyc(GEN G)` elementary divisors (d_i) of $(\mathbf{Z}/N\mathbf{Z})^*$.

`GEN znstar_get_gen(GEN G)` SNF generators divisors (g_i) of $(\mathbf{Z}/N\mathbf{Z})^*$.

`GEN znstar_get_conreycyc(GEN G)` orders (d'_i) of Conrey generators.

`GEN znstar_get_conreygen(GEN G)` Conrey generators (g'_i) .

`GEN znstar_get_U(GEN G)` a square matrix U such that $(g_i) = U(g'_i)$.

`GEN znstar_get_Ui(GEN G)` a square matrix U' such that $U'(g_i) = (g'_i)$. In general, UU' will not be the identity.

13.1.8 Inserting info in a number field structure.

If the required data is not part of the structure, it is computed then inserted, and the new value is returned.

These functions expect a `bnf` argument:

`GEN bnf_build_cycgen(GEN bnf)` the *bnf* contains generators $[g_1, \dots, g_k]$ of the class group, each with order d_i . Then $g_i^{d_i} = (x_i)$ is a principal ideal. This function returns the x_i as a factorization matrix (`famat`) giving the element in factored form as a product of S -units.

`GEN bnf_build_matalpha(GEN bnf)` the class group was computed using a factorbase S of prime ideals $\mathfrak{p}_i$, $i \leq r$. They satisfy relations of the form $\prod_j \mathfrak{p}_i^{e_{i,j}} = (\alpha_j)$, where the $e_{i,j}$ are given by the matrices *bnf*[1] (W , singling out a minimal set of generators in S) and *bnf*[2] (B , expressing the rest of S in terms of the singled out generators). This function returns the α_j in factored form as a product of S -units.

`GEN bnf_build_units(GEN bnf)` returns a minimal set of generators for the unit group in expanded form. The first element is a torsion unit, the others have infinite order. This expands units in compact form contained in a `bnf` from `bnfinit(, 1)` and may be *very* expensive if the units are huge.

`GEN bnf_build_cheapfu(GEN bnf)` as `bnf_build_units` but only expand units in compact form if the computation is inexpensive (a few seconds). Return `NULL` otherwise.

These functions expect a `rnf` argument:

`GEN rnf_build_nfabs(GEN rnf, long prec)` given a *rnf* structure attached to L/K , (compute and) return an *nf* structure attached to L at precision `prec`.

`void rnfcomplete(GEN rnf)` as `rnf_build_nfabs` using the precision of K for `prec`.

`GEN rnf_zkabs(GEN rnf)` returns a $\mathbf{Z}$ -basis in HNF for $\mathbf{Z}_L$ as a pair $[T, v]$, where T is `rnf_get_polabs(rnf)` and v a vector of elements lifted from $\mathbf{Q}[X]/(T)$. Note that the function `rnf_build_nfabs` essentially applies `nfinit` to the output of this function.

13.1.9 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 from x before increasing its accuracy, and returns only the latter.

`GEN bnrnewprec(GEN x, long prec)`. As `nfnewprec`, but extracts a *bnr* structure from x before increasing its accuracy, and returns only the latter.

`GEN rnfnewprec(GEN x, long prec)`. As `nfnewprec`, but extracts a *rnf* structure from x before increasing its accuracy, and returns only the latter.

`GEN nfnewprec_shallow(GEN nf, long prec)`

`GEN bnfnewprec_shallow(GEN bnf, long prec)`

`GEN rnfnewprec_shallow(GEN rnf, 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*.

13.1.10 Number field arithmetic. The number field $K = \mathbf{Q}[X]/(T)$ 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 $N = [K : \mathbf{Q}]$, representing the element in terms of the computed integral basis (e_i) , as

$$\text{sum}(i = 1, N, v[i] * nf.zk[i])$$

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 nfsub(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 k)` returns x^k , k is in $\mathbf{Z}$.

`GEN nfpow_u(GEN nf, GEN x, ulong k)` returns x^k , $k \geq 0$; the argument `nf` is a true `nf` structure.

`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 $\mathfrak{p}$ attached to the `prid` `pr`. Returns `LONG_MAX` if x is 0.

`GEN nfnorm(GEN nf, GEN x)` absolute norm of x .

`GEN nftrace(GEN nf, GEN x)` absolute trace of x .

`GEN nfpoleval(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 nfdiveuc(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 = nfdiveuc(nf, a, b);
r = nfsub(nf, a, nfmul(nf,q,b));    \\ or r = nfmod(nf,a,b);
```

`GEN nfmod(GEN nf, GEN a, GEN b)` returns r such that

```
q = nfdiveuc(nf, a, b);
r = nfsub(nf, a, nfmul(nf,q,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 nf_to_scalar_or_polmod(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 `t_POLMOD` representation (`nfbasistoalg`). Shallow function.

GEN `nfV_to_scalar_or_alg`(GEN `nf`, GEN `v`) apply `nf_to_scalar_or_alg` to all components of vector `v`.

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.

GEN `nfX_to_monico`(GEN `nf`, GEN `T`, GEN `*pL`) given a nonzero `t_POL` T with coefficients in nf , return a monic polynomial f with integral coefficients such that $f(x) = CT(x/L)$ for some integral L and some C in nf . The function allows coefficients in basis form; if $L \neq 1$, it will return them in algebraic form. If `pL` is not NULL, `*pL` is set to L . 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 `nfX_disc`(GEN `nf`, GEN `A`) given an nf structure attached to a number field K with main variable Y (`nf_get_varn(nf)`), a `t_POL` $A \in K[X]$ given as a lift in $\mathbf{Q}[X, Y]$ (implicitly modulo `nf_get_pol(nf)`), return the discriminant of A as a `t_POL` in $\mathbf{Q}[Y]$ (representing an element of K).

GEN `nfX_resultant`(GEN `nf`, GEN `A`, GEN `B`) analogous to `nfX_disc`, $A, B \in \mathbf{Q}[X, Y]$; return the resultant of A and B with respect to X as a `t_POL` in $\mathbf{Q}[Y]$ (representing an element of K).

GEN `nfinvmodideal`(GEN `nf`, GEN `x`, GEN `A`) given an algebraic integer x and a nonzero integral ideal A in HNF, returns a y such that $xy \equiv 1$ modulo A .

GEN `nfpowmodideal`(GEN `nf`, GEN `x`, GEN `n`, GEN `ideal`) given an algebraic integer x , an integer n , and a nonzero integral ideal A in HNF, returns an algebraic integer congruent to x^n modulo A .

GEN `nfmuli`(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. The argument `nf` is a true nf structure.

GEN `nfsqri`(GEN `nf`, GEN `x`) returns x^2 assuming that x is a `t_INT` or a ZV of the correct dimension. The argument `nf` is a true nf structure.

GEN `nfC_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 `nfC_multable_mul`(GEN `v`, GEN `mx`) same as `nfC_nf_mul`, where the argument x is replaced by its multiplication table `mx`.

GEN `zkC_multable_mul`(GEN `v`, GEN `x`) same as `nfC_nf_mul`, where v is a vector of algebraic integers, x is an algebraic integer, and x is replaced by `zk_multable(x)`.

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_inv`(GEN `nf`, GEN `x`) given a ZC x (implicitly representing an algebraic integer), returns the QC giving the inverse x^{-1} . Return NULL if x is 0. Not stack-clean but safe for `gc_upto`.

GEN `zkmultable_inv`(GEN `mx`) as `zk_inv`, where the argument given is `zk_multable(x)`.

GEN `zkmultable_capZ`(GEN `mx`) given a nonzero *zkmultable* mx attached to $x \in \mathbf{Z}_K$, return the positive generator of $(x) \cap \mathbf{Z}$.

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.

13.1.11 Number field arithmetic for linear algebra.

The following routines implement multiplication in a commutative R -algebra, generated by $(e_1 = 1, \dots, e_n)$, 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 $x e_1$.

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 $x \cdot v[i]$.

The following routines implement naive linear algebra using the *black box field* mechanism:

GEN `nfM_det`(GEN `nf`, GEN `M`)

GEN `nfM_inv`(GEN `nf`, GEN `M`)

GEN `nfM_ker`(GEN `nf`, GEN `M`)

GEN `nfM_mul`(GEN `nf`, GEN `A`, GEN `B`)

GEN `nfM_nfC_mul`(GEN `nf`, GEN `A`, GEN `B`)

13.1.12 Cyclotomic field arithmetic for linear algebra.

The following routines implement modular algorithms in cyclotomic fields. In the prototypes, P is the n -th cyclotomic polynomial Φ_n and M is a `t_MAT` with `t_INT` or `ZX` coefficients, understood modulo P .

`GEN ZabM_ker(GEN M, GEN P, long n)` returns an integral (primitive) basis of the kernel of M .

`GEN ZabM_indexrank(GEN M, GEN P, long n)` return a vector with two `t_VEC` components given the rank profile of M . Inefficient (but correct) when M does not have almost full column rank.

`GEN ZabM_inv(GEN M, GEN P, long n, GEN *pden)` assume that M is invertible; return N and sets the algebraic integer `*pden` (an integer or a `ZX`, implicitly modulo P) such that $MN = \text{den} \cdot \text{Id}$.

`GEN ZabM_pseudoinv(GEN M, GEN P, long n, GEN *pv, GEN *pden)` analog of `ZM_pseudoinv`. Not stack-clean.

`GEN ZabM_inv_ratlift(GEN M, GEN P, long n, GEN *pden)` return a primitive matrix H such that MH is d times the identity and set `*pden` to d . Uses a multimodular algorithm, attempting rational reconstruction along the way. To be used when you expect that the denominator of M^{-1} is much smaller than $\det M$ else use `ZabM_inv`.

13.1.13 Cyclotomic trace.

Given two positive integers m and n such that $K_m = \mathbf{Q}(\zeta_m) \subset K_n = \mathbf{Q}(\zeta_n)$, these functions implement relative trace computation from K_n to K_m . This is in particular useful for character values.

`GEN Qab_trace_init(long n, long m, GEN Pn, GEN Pm)` assume that Pn is `polcyclo(n)`, Pm is `polcyclo(m)` (both in the same variable), initialize a structure T used in the following routines. Shallow function.

`GEN Qab_tracerel(GEN T, long t, GEN z)` assume T was created by `Qab_trace_init`, t is an integer such that $0 \leq t < [K_n : K_m]$ and z belongs to the cyclotomic field $\mathbf{Q}(\zeta_n) = \mathbf{Q}[X]/(Pn)$. Return the normalized relative trace $[K_n : K_m]^{-1} \text{Tr}_{K_n/K_m}(\zeta_n^t z)$. Shallow function.

`GEN QabV_tracerel(GEN T, long t, GEN v)` v being a vector of entries belonging to K_n , apply `Qab_tracerel` to all entries. Shallow function.

`GEN QabM_tracerel(GEN T, long t, GEN m)` m being a matrix of entries belonging to K_n , apply `Qab_tracerel` to all entries. Shallow function.

13.1.14 Elements in factored form.

Computational algebraic theory performs extensively linear algebra on $\mathbf{Z}$ -modules with a natural multiplicative structure (K^* , fractional ideals in K , $\mathbf{Z}_K^*$, ideal class group), thereby raising elements to horrendously large powers. A seemingly innocuous elementary linear algebra operation like $C_i \leftarrow C_i - 10000C_1$ 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_INT`s, 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) shallow version of `famat_mul`.

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 .

GEN `famat_pow_shallow`(GEN f , GEN n) shallow version of `famat_pow`.

GEN `famat_pows_shallow`(GEN f , long n) shallow version of `famat_pow` where n is a small integer.

GEN `famat_mulpow_shallow`(GEN f , GEN g , GEN e) *famat* corresponding to $f \cdot g^e$. Shallow function.

GEN `famat_mulpows_shallow`(GEN f , GEN g , long e) *famat* shallow version of `famat_mulpow` where e is a small integer.

GEN `famat_sqr`(GEN f) returns f^2 .

GEN `famat_inv`(GEN f) returns f^{-1} .

GEN `famat_div`(GEN f , GEN g) return f/g .

GEN `famat_inv_shallow`(GEN f) shallow version of `famat_inv`.

GEN `famat_div_shallow`(GEN f , GEN g) return f/g ; shallow.

GEN `famat_Z_gcd`(GEN M , GEN n) restrict the *famat* M to the prime powers dividing n .

GEN `to_famat`(GEN x , GEN k) given an element x and an exponent k , returns the *famat* x^k .

GEN `to_famat_shallow`(GEN x , GEN k) same, as a shallow function.

GEN `Z_to_famat`(GEN x) converts the `t_INT` x to a *famat*. This does not factor x but will replace it by y^k with y integral and k maximal. Note that 0 gets converted to 0^1 . Shallow function.

GEN `Q_to_famat`(GEN x) converts the `t_INT` or `t_FRAC` x to a *famat*. If x is a `t_INT`, as `Z_to_famat`. Else, this does not factor x but will replace it by $y^a z^{-b}$ with integral y and z and maximal (positive) a and b . Shallow function.

GEN `famatV_factorback`(GEN v , GEN e) given a vector of *famats* v and a ZV e return the *famat* $\prod_i v[i]^{e[i]}$. Shallow function.

GEN `famatV_zv_factorback`(GEN v , GEN e) given a vector of *famats* v and a zv e return the *famat* $\prod_i v[i]^{e[i]}$. Shallow function.

GEN `ZM_famat_limit`(GEN f , GEN $limit$) given a *famat* f with `t_INT` entries, returns a *famat* g with all factors larger than $limit$ multiplied out as the last entry (with exponent 1). Shallow function.

Note that it is trivial to break up a *famat* into its two constituent columns: `gel(f,1)` and `gel(f,2)` are the elements and exponents respectively. Conversely, `mkmat2` builds a (shallow) *famat* from two `t_COLS` of the same length.

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 `famat_remove_trivial`(GEN *f*) given a *famat* *f*, returns a *famat* *g* without 0 exponents. Shallow function.

GEN `famat_small_reduce`(GEN *f*) as `famat_reduce`, but for exponents given by a `t_VECSMALL`.

GEN `famat_to_nf`(GEN *nf*, GEN *f*) You normally never want to do this! This is a simplified form of `nfactorback`, where we do not check the user input for consistency. The elements must be regular algebraic numbers (not *famats*) over the given number field.

Why should you *not* want to use this function? You should not need to: most of the functions useful in this context accept *famats* as inputs, for instance `nfsign`, `nfsign_arch`, `ideallog` and `bnfisunit`. Otherwise, we can hopefully make good use of a quotient operation (modulo a fixed conductor, modulo ℓ -th powers); see the end of Section 13.1.26. If nothing else works, this function is available but is expected to be slow or even overflow the possibilities of the implementation.

GEN `famat_ideal_factor`(GEN *nf*, GEN *x*) This is a good alternative for `famat_to_nf`, returning the factorization of the ideal generated by *x*. Since the answer is still given in factorized form, there is no risk of coefficient explosion when the exponents are large. Of course, all components of *x* must be factored individually.

GEN `famat_nfvalrem`(GEN *nf*, GEN *x*, GEN *pr*, GEN **py*) return the valuation *v* at *pr* of `famat_to_nf(x)`, without performing the expansion of course. Notice that the output is a GEN since it cannot be assumed to fit into a long. If *py* is not NULL it contains the *famat* obtained by applying `nfvalrem` to each entry of the first column and copying the second column, with 0 exponents removed. The expanded algebraic number is coprime to *pr* (in fact, all its components are coprime to *pr*) and equal to $x\tau^v$ where τ is the fixed anti-uniformizer for *pr* (`pr_get_tau`).

Caveat. Receiving a *famat* input, `bnfisunit` assumes that it is an actual unit, since this is expensive to check, and normally easy to ensure from the user's side.

13.1.15 Ideal arithmetic.

Conversion to HNF.

GEN `idealhnf`(GEN *nf*, GEN *x*) where the argument *nf* is a true *nf* structure. 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 `idealprimedec` or `ideal_factor`), 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 $\mathbf{Z}_K$ -module they generate,
- if *N* or more are given, it is assumed that they form a $\mathbf{Z}$ -basis (that the matrix has maximal rank *N*). This acts as `mathnf` since the $\mathbf{Z}_K$ -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:

GEN `idealhnf_principal`(GEN `nf`, GEN `x`) returns the ideal generated by the algebraic number x .

GEN `idealhnf_shallow`(GEN `nf`, GEN `x`) a shallow `idealhnf`: if x is already in HNF, we return x , not a copy!

GEN `idealhnf_two`(GEN `nf`, GEN `v`) assuming $a = v[1]$ is a nonzero `t_INT` and $b = v[2]$ is an algebraic integer, possibly given in regular representation by a `t_MAT` (the multiplication table by b , see `zk_multable`), returns the HNF of $a\mathbf{Z}_K + b\mathbf{Z}_K$.

GEN `nfV_idealhnf`(GEN `nf`, GEN `v`, GEN `*pden`) v being a vector of elements of *nf* with common denominator d over the integral basis, returns the HNF of the ideal generated by the elements of dv and set `*pden` = d if $d \neq 1$, otherwise set `*pden` to NULL.

Operations.

The basic ideal routines accept all *nfs* (*nf*, *bnf*, *bnr*) and ideals in any form, including extended ideals, and return ideals in HNF, or an extended ideal when that makes sense:

GEN `idealadd`(GEN `nf`, GEN `x`, GEN `y`) returns $x + y$.

GEN `idealdiv`(GEN `nf`, GEN `x`, GEN `y`) returns x/y . Returns an extended ideal if x or y is an extended ideal.

GEN `idealmul`(GEN `nf`, GEN `x`, GEN `y`) returns xy . Returns an extended ideal if x or y is an extended ideal.

GEN `idealsqr`(GEN `nf`, GEN `x`) returns x^2 . Returns an extended ideal if x is an extended ideal.

GEN `idealinv`(GEN `nf`, GEN `x`) returns x^{-1} . Returns an extended ideal if x is an extended ideal.

GEN `idealpow`(GEN `nf`, GEN `x`, GEN `n`) returns x^n . Returns an extended ideal if x is an extended ideal.

GEN `idealpows`(GEN `nf`, GEN `ideal`, long `n`) returns x^n . Returns an extended ideal if x is an extended ideal.

GEN `idealmulred`(GEN `nf`, GEN `x`, GEN `y`) returns an extended ideal equal to xy .

GEN `idealpowred`(GEN `nf`, GEN `x`, GEN `n`) returns an extended ideal equal to x^n .

More specialized routines suffer from various restrictions:

GEN `idealdivexact`(GEN `nf`, GEN `x`, GEN `y`) returns x/y , assuming that the quotient is an integral ideal. Much faster than `idealdiv` when the norm of the quotient is small compared to Nx . Strips the principal parts if either x or y is an extended ideal.

GEN `idealdivpowprime`(GEN `nf`, GEN `x`, GEN `pr`, GEN `n`) returns $x\mathfrak{p}^{-n}$, assuming x is an ideal in HNF or a rational number, and `pr` a *prid* attached to `p`. Shallow function: it returns x when $n = 0$. The `nf` argument must be a true *nf* structure.

GEN `idealmulpowprime`(GEN `nf`, GEN `x`, GEN `pr`, GEN `n`) returns $x\mathfrak{p}^n$, assuming x is an ideal in HNF or a rational number, and `pr` a *prid* attached to `p`. Shallow function: it returns x when $n = 0$. The `nf` argument must be a true *nf* structure.

GEN idealprodprime(GEN *nf*, GEN *v*) given a list *v* of prime ideals in *prid* form, return their product. Assume that *nf* is a true *nf* structure.

GEN idealprod(GEN *nf*, GEN *v*) given a list *v* of ideals, return their product.

GEN idealprodval(GEN *nf*, GEN *v*, GEN *pr*) given a list *v* of ideals return the valuation of their product at the prime ideal *pr*.

GEN idealHNF_mul(GEN *nf*, GEN *x*, GEN *y*) returns xy , assuming that *nf* is a true *nf*, *x* is an integral ideal in HNF and *y* is an integral ideal in HNF or precompiled form (see below). For maximal speed, the second ideal *y* may be given in precompiled form $y = [a, b]$, where *a* is a nonzero **t_INT** and *b* is an algebraic integer in regular representation (a **t_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 $xy \cap \mathbf{Z}$).

GEN idealHNF_inv(GEN *nf*, GEN *I*) returns I^{-1} , assuming that *nf* is a true *nf* and *x* is a fractional ideal in HNF.

GEN idealHNF_inv_Z(GEN *nf*, GEN *I*) returns $(I \cap \mathbf{Z}) \cdot I^{-1}$, assuming that *nf* is a true *nf* and *x* is an integral fractional ideal in HNF. The result is an integral ideal in HNF.

GEN ideals_by_norm(GEN *nf*, GEN *N*) given a true *nf* structure and a integer *N*, which can also be given by a factorization matrix or (preferably) by a pair $[N, \mathbf{factor}(N)]$, return all ideals of norm *N* in factored form. Not stack-clean.

Approximation.

GEN idealaddtoone(GEN *nf*, GEN *A*, GEN *B*) given to coprime integer ideals *A*, *B*, returns $[a, b]$ with $a \in A$, $b \in B$, such that $a + b = 1$. The result is reduced mod AB , so *a*, *b* will be small.

GEN idealaddtoone_i(GEN *nf*, GEN *A*, GEN *B*) as **idealaddtoone** except that *nf* must be a true *nf*, and only *a* is returned.

GEN idealaddtoone_raw(GEN *nf*, GEN *A*, GEN *B*) as **idealaddtoone_i** except that the reduction mod AB is only performed modulo the lcm of $A \cap \mathbf{Z}$ and $B \cap \mathbf{Z}$, which will increase the size of *a*.

GEN zkchineseinit(GEN *nf*, GEN *A*, GEN *B*, GEN *AB*) given two coprime integral ideals *A* and *B* (in any form, preferably HNF) and their product *AB* (in HNF form), initialize a solution to the Chinese remainder problem modulo AB . The *nf* argument must be a true *nf* structure.

GEN zkchinese(GEN *zkc*, GEN *x*, GEN *y*) given *zkc* from **zkchineseinit**, and *x*, *y* two integral elements given as **t_INT** or **ZC**, return a *z* modulo AB such that $z = x \bmod A$ and $z = y \bmod B$.

GEN zkchinese1(GEN *zkc*, GEN *x*) as **zkchinese** for $y = 1$; useful to lift elements in a nice way from $(\mathbf{Z}_K/A_i)^*$ to $(\mathbf{Z}_K/\prod_i A_i)^*$.

GEN hnfmerge_get_1(GEN *A*, GEN *B*) given two square upper HNF integral matrices *A*, *B* of the same dimension $n > 0$, return *a* in the image of *A* such that $1 - a$ is in the image of *B*. (By abuse of notation we denote 1 the column vector $[1, 0, \dots, 0]$.) If such an *a* does not exist, return **NULL**. This is the function underlying **idealaddtoone**.

GEN idealadddmultoone(GEN *nf*, GEN *v*) given a list of *n* (globally) coprime integer ideals ($v[i]$) returns an *n*-dimensional vector *a* such that $a[i] \in v[i]$ and $\sum a[i] = 1$. If $[K : \mathbf{Q}] = N$, this routine computes the HNF reduction (with $Gl_{nN}(\mathbf{Z})$ base change) of an $N \times nN$ matrix; so it is well worth pruning "useless" ideals from the list (as long as the ideals remain globally coprime).

GEN `idealapprfact`(GEN `nf`, GEN `fx`) as `idealappr`, except that x *must* be given in factored form. (This is unchecked.)

GEN `idealcoprime`(GEN `nf`, GEN `x`, GEN `y`). Given 2 integral ideals x and y , returns an algebraic number α such that αx is an integral ideal coprime to y .

GEN `idealcoprimefact`(GEN `nf`, GEN `x`, GEN `fy`) same as `idealcoprime`, except that y is given in factored form, as from `idealfactor`.

GEN `idealchinese`(GEN `nf`, GEN `x`, GEN `y`)

GEN `idealchineseinit`(GEN `nf`, GEN `x`)

13.1.16 Maximal ideals.

The PARI structure attached to maximal ideals is a *prid* (for *prime ideal*), usually produced by `idealprimedec` and `idealfactor`. In this section, we describe the format; other sections will deal with their daily use.

A *prid* attached to a maximal ideal $\mathfrak{p}$ 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 π with valuation 1 at $\mathfrak{p}$ and valuation 0 at all other primes dividing p and a rescaled “anti-uniformizer” τ used to compute valuations. This τ is an algebraic integer such that τ/p has valuation -1 at $\mathfrak{p}$ and is integral at all other primes; in particular, the valuation of $x \in \mathbf{Z}_K$ is positive if and only if the algebraic integer $x\tau$ is divisible by p (easy to check for elements in `t_COL` form).

GEN `pr_get_p`(GEN `pr`) returns p . Shallow function.

GEN `pr_get_gen`(GEN `pr`) returns π . 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 `zk_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.

ulong `upr_norm`(GEN `pr`) returns the norm p^f of the maximal ideal, as an `ulong`. Assume that the result does not overflow.

GEN `pr_hnf`(GEN `pr`) return the HNF of $\mathfrak{p}$.

GEN `pr_inv`(GEN `pr`) return the fractional ideal $\mathfrak{p}^{-1}$, in HNF.

GEN `pr_inv_p`(GEN `pr`) return the integral ideal $p\mathfrak{p}^{-1}$, in HNF.

GEN `idealprimedec`(GEN `nf`, GEN `p`) list of maximal ideals dividing the prime p .

GEN `idealprimedec_limit_f`(GEN `nf`, GEN `p`, long `f`) as `idealprimedec`, limiting the list to primes of residual degree $\leq f$ if f is nonzero.

GEN `idealprimedec_limit_norm`(GEN `nf`, GEN `p`, GEN `B`) as `idealprimedec`, limiting the list to primes of norm $\leq B$, which must be a positive `t_INT`.

int `prime_over_p_full`(GEN `P`) given a vector of distinct primes above the same rational prime p , return 1 if it contains all primes above p .

GEN `idealprimedec_galois`(GEN `nf`, GEN `p`) return a single prime ideal above p . The `nf` argument is a true *nf* structure.

GEN `idealprimedec_degrees`(GEN `nf`, GEN `p`) return a (sorted) `t_VECSMALL` containing the residue degrees $f(\mathfrak{p}/p)$. The `nf` argument is a true *nf* structure.

GEN `idealprimedec_kummer`(GEN `nf`, GEN `Ti`, long `ei`, GEN `p`) let *nf* (true *nf*) correspond to $K = \mathbf{Q}[X]/(T)$ (T monic $\mathbf{Z}[X]$). Let $T \equiv \prod_i T_i^{e_i} \pmod{p}$ be the factorization of T and let (f, g, h) be as in Dedekind criterion for prime p : $f \equiv \prod T_i$, $g \equiv \prod T_i^{e_i-1}$, $h = (T - fg)/p$, and let D be the gcd of (f, g, h) in $\mathbf{F}_p[X]$. Let `Ti` (`FpX`) be one irreducible factor T_i not dividing D , with `ei` = e_i . This function returns the prime ideal attached to T_i by Kummer / Dedekind criterion, namely $p\mathbf{Z}_K + T_i(\bar{X})\mathbf{Z}_K$, which has ramification index e_i over p . The `nf` argument is a true *nf* structure. Shallow function.

GEN `idealfactor`(GEN `nf`, GEN `x`) factors the fractional (hence nonzero) ideal x into prime ideal powers; return the factorization matrix.

GEN `idealfactor_limit`(GEN `nf`, GEN `x`, ulong `lim`) as `idealfactor`, including only prime ideals above rational primes $< \text{lim}$.

GEN `idealfactor_partial`(GEN `nf`, GEN `x`, GEN `L`) return partial factorization of fractional ideal x as limited by argument L :

- $L = \text{NULL}$: as `idealfactor`;
- L a `t_INT`: as `idealfactor_limit`;

• L a vector of prime ideals of *nf* and/or rational primes (standing for “all prime ideal divisors of given rational prime”) limit factorization to trial division by elements of L ; do not include the cofactor. For efficiency, the list should not contain the same element twice, nor both a rational prime and one of its prime ideal divisors, but the function will work in that case as well.

GEN `idealHNF_Z_factor`(GEN `x`, GEN `*pvN`, GEN `*pvZ`) given an integral (nonzero) ideal x in HNF, compute both the factorization of Nx and of $x \cap \mathbf{Z}$. This returns the vector of prime divisors of both and sets `*pvN` and `*pvZ` to the corresponding `t_VECSMALL` vector of exponents for the factorization for the Norm and intersection with $\mathbf{Z}$ respectively.

GEN `idealHNF_Z_factor_i`(GEN `x`, GEN `fa`, GEN `*pvN`, GEN `*pvZ`) internal variant of `idealHNF_Z_factor` where `fa` is either a partial factorization of $x \cap \mathbf{Z}$ ($= x[1,1]$) or `NULL`. Returns the prime divisors of x above the rational primes in `fa` and attached `vn` and `vZ`. If `fa` is `NULL`, use the full factorization, i.e. identical to `idealHNF_Z_factor`.

GEN `nf_pV_to_prV`(GEN `nf`, GEN `P`) given a vector of rational primes P , return the vector of all prime ideals above the $P[i]$.

GEN `nf_deg1_prime`(GEN `nf`) let `nf` be a true *nf*. This function returns a degree 1 (unramified) prime ideal not dividing `nf.index`. In fact it returns an ideal above the smallest prime $p \geq [K : \mathbf{Q}]$ satisfying those conditions.

GEN `prV_lcm_capZ`(GEN `L`) given a vector L of *prid* (maximal ideals) return the squarefree positive integer generating their lcm intersected with $\mathbf{Z}$. Not stack-clean.

GEN `prV_primes`(GEN `L`) given a vector of *prid*, return the (sorted) list of rational primes P they divide. Not stack-clean but suitable for `gc_upto`.

GEN `pr_uniformizer`(GEN `pr`, GEN `F`) given a *prid* attached to $\mathfrak{p}/p$ and F in $\mathbf{Z}$ divisible exactly by p , return an F -uniformizer for `pr`, i.e. a t in $\mathbf{Z}_K$ such that $v_{\mathfrak{p}}(t) = 1$ and $(t, F/\mathfrak{p}) = 1$. Not stack-clean.

GEN `pr_anti_uniformizer`(GEN `nf`, GEN `pr`) given a number field `nf` and a *prid* attached to a prime ideal $\mathfrak{p}/p$ of ramification index e , return an anti-uniformizer for `pr`, i.e. a t in $\mathbf{Z}_K$ such that $t \equiv 1 \pmod{\mathfrak{p}}$ and $t \equiv 0 \pmod{p\mathfrak{p}^{-e}}$. Not stack-clean.

13.1.17 Decomposition groups.

GEN `idealramfrobienius`(GEN `nf`, GEN `gal`, GEN `pr`, GEN `ram`) Let K be the number field defined by `nf` and assume $K/\mathbf{Q}$ be a Galois extension with Galois group given `gal=galoisinit(nf)`, and that `pr` is the prime ideal $\mathfrak{P}$ in *prid* format, and that $\mathfrak{P}$ is ramified, and `ram` is its list of ramification groups as output by `idealramgroups`. This function returns a permutation of `gal.group` which defines an automorphism σ in the decomposition group of $\mathfrak{P}$ such that if p is the unique prime number in $\mathfrak{P}$, then $\sigma(x) \equiv x^p \pmod{\mathbf{P}}$ for all $x \in \mathbf{Z}_K$.

GEN `idealramfrobienius_aut`(GEN `nf`, GEN `gal`, GEN `pr`, GEN `ram`, GEN `aut`) as `idealramfrobienius`(`nf`, `gal`, `pr`, `ram`).

GEN `idealramgroups_aut`(GEN `nf`, GEN `gal`, GEN `pr`, GEN `aut`) as `idealramgroups`(`nf`, `gal`, `pr`).

GEN `idealfrobienius_aut`(GEN `nf`, GEN `gal`, GEN `pr`, GEN `aut`) faster version of `idealfrobienius`(`nf`, `gal`, `pr` where `aut` must be equal to `nfgaloispermtobasis`(`nf`, `gal`).

13.1.18 Reducing modulo maximal ideals.

GEN `nfmodprinit`(GEN `nf`, GEN `pr`) returns an abstract `modpr` structure, attached 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.

GEN `modpr_get_pr`(GEN `x`) return the `pr` component from a `modpr` structure.

GEN `modpr_get_p`(GEN `x`) return the p component from a `modpr` structure (underlying rational prime).

GEN `modpr_get_T`(GEN `x`) return the T component from a `modpr` structure: either NULL (prime of degree 1) or an irreducible $\mathbf{F}_p[X]$ defining the residue field over $\mathbf{F}_p$.

In library mode, it is often easier to use directly

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 $\mathbf{F}_p[X]/(T)$ for some monic $T \in \mathbf{F}_p[X]$, 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 $\mathbf{F}_p$.

In short, this receives (or initializes) a `modpr` structure, and extracts from it T , p and $\mathfrak{p}$.

GEN `nf_to_Fq`(GEN `nf`, GEN `x`, GEN `modpr`) returns an $\mathbf{F}_q$ congruent to x modulo the maximal ideal attached to `modpr`. The output is canonical: all elements in a given residue class are represented by the same $\mathbf{F}_q$.

`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 nonnegative 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`.

The following functions are technical and avoid computing a true `nfmodpr`:

`GEN pr_basis_perm(GEN nf, GEN pr)` given a true `nf` structure and a prime ideal `pr` above p , return as a `t_VEC` the $f(\mathfrak{p}/p)$ indices i such that the `nf.zk[i]` mod p form an $\mathbf{F}_p$ -basis of the residue field.

`GEN QXQV_to_FpM(GEN v, GEN T, GEN p)` let p be a positive integer, v be a vector of n polynomials with rational coefficients whose denominators are coprime to p , and T be a `ZX` (preferably monic) of degree d whose leading coefficient is coprime to p . Return the $d \times n$ `FpM` whose columns are the $v[i] \pmod{T, p}$ in the canonical basis $1, X, \dots, X^{d-1}$, see `RgX_to_RgC`. This is for instance useful when v contains a $\mathbf{Z}$ -basis of the maximal order of a number field $\mathbf{Q}[X]/(P)$, p is a prime not dividing the index of P and T is an irreducible factor of $P \pmod{p}$, attached to a maximal ideal $\mathfrak{p}$: left-multiplication by the matrix maps number field elements (in basis form) to the residue field of $\mathfrak{p}$.

`GEN hnf_Znproj(GEN H)` let H be a $n \times n$ HNF matrix which is an identity apart from the first row and assume that $q = H[1,1]$ is positive. Then $\mathbf{Z}^n / \text{im}_{\mathbf{Z}} H \simeq \mathbf{Z}/q\mathbf{Z}$ and this function returns a vector v allowing to map any $x \in \mathbf{Z}^n$ to the quotient: the image is `FpV.dotproduct(x, v, q)`.

The conditions are for instance satisfied when H is the HNF matrix attached to a prime ideal of degree 1 (because we assume that the implied fixed $\mathbf{Z}$ -basis for the maximal order starts with 1). This allows to map algebraic integers to the (prime) residue field. This function is a special case of `zkmodprinit`.

13.1.19 Valuations.

`long nfval(GEN nf, GEN x, GEN P)` return $v_P(x)$

Unsafe functions. assume that P, Q are `prid`.

`long ZC_nfval(GEN x, GEN P)` returns $v_P(x)$, assuming x is a `ZC`, representing a nonzero algebraic integer.

`long ZC_nfvalrem(GEN x, GEN P, GEN *newx)` returns $v = v_P(x)$, assuming x is a `ZC`, representing a nonzero algebraic integer, and sets `*newx` to $x\tau^v$ which is an algebraic integer coprime to p .

`int ZC_prdvd(GEN x, GEN P)` returns 1 if 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 P, GEN Q)` returns 1 if 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 τ element may differ. Returns 0 otherwise.

13.1.20 Signatures.

“Signs” of the real embeddings of number field element are represented in additive notation, using the standard identification $(\mathbf{Z}/2\mathbf{Z}, +) \rightarrow (\{-1, 1\}, \times)$, $s \mapsto (-1)^s$.

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 return as a `t_VECSMALL` the list of indices i such that $v[i] \neq 0$. (Typically used with 0, 1-vectors but not necessarily so.) If v is already a `t_VECSMALL`, return it: not stack-clean in this case.

`GEN vecsmall01_to_indices(GEN v)` as

`vec01_to_indices(zv_to_ZV(v));`

`GEN indices_to_vec01(GEN p, long n)` return the 0/1 vector of length n with ones exactly at the positions $p[1], p[2], \dots$

`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. The argument `nf` is a true `nf` structure.

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`. The argument `nf` is a true `nf` structure.

`int nfchecksigns(GEN nf, GEN x, GEN pl)` pl is a `t_VECSMALL` with r_1 components, all of which are in $\{-1, 0, 1\}$. Return 1 if $\sigma_i(x)pl[i] \geq 0$ for all i , and 0 otherwise.

`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

bnf.tu and of system of fundamental units for the field **bnf.fu**, in that order if **add_tu** is set; and in the same order as **bnf.fu** otherwise.

GEN nfsign_fu(GEN bnf, GEN archp) returns the signs at **archp** of the fundamental units **bnf.fu**. This is an alias for **nfsign_units** with **add_tu** unset.

GEN nfsign_tu(GEN bnf, GEN archp) returns the signs at **archp** of the torsion unit generator **bnf.tu**.

GEN nfsign_from_logarch(GEN L, GEN invpi, GEN archp) given L the vector of the $\log \sigma(x)$, where σ runs through the (real or complex) embeddings of some number field, **invpi** being a floating point approximation to $1/\pi$, 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 $\log \sigma(x)$ for a system of fundamental units of the field.

GEN set_sign_mod_divisor(GEN nf, GEN x, GEN y, GEN sarch) let $f = f_0 f_\infty$ be a divisor, let **sarch** be the output of **nfarchstar**(**nf**, **f0**, **finf**), let x encode a vector of signs at the places of f_∞ (see below), and let y be a nonzero number field element. Returns z congruent to $y \bmod f_0$ (integral if y is) such that z and x have the same signs at f_∞ . The argument **nf** is a true *nf* structure.

The following formats are supported for x : a $\{0,1\}$ -vector of signs as a **t_VECSMALL** (0 for positive, 1 for negative); **NULL** for a totally positive element (only 0s); a number field element which is replaced by its signature at f_∞ .

GEN nfarchstar(GEN nf, GEN f0, GEN finf) for a divisor $f = f_0 f_\infty$ represented by the integral ideal **f0** in HNF and the **finf** in **indices** form, returns $(\mathbf{Z}_K/f_\infty)^*$ in a form suitable for computations mod f . See **set_sign_mod_divisor**.

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.

13.1.21 Complex embeddings.

GEN nfembed(GEN nf, GEN x, long k) returns a floating point approximation of the k -th embedding of x (attached to the k -th complex root in **nf.roots**). Note that the semantic is different from **nfeltembed** (which increases the precision of *nf* until the embeddings have the requested precision): **nfembed** provides the embedding with the precision that is achievable given *nf*.

GEN nfeltembed_i(GEN *pnf, GEN x, GEN ind, long prec) as **nfeltembed**, except that no garbage collecting is performed and ***pnf** must be initially set to a (true) *nf* structure and, if the routine needs to increase the accuracy of *nf* to achieve the requested accuracy **prec**, then the functions sets it to the new more precise *nf*.

GEN nf_cxlog(GEN nf, GEN x, long prec) return the vector of complex logarithmic embeddings $(e_i \log(\sigma_i X))$ where $e_i = 1$ if $i \leq r_1$ and $e_i = 2$ if $r_1 < i \leq r_2$ of $X = \mathbf{Q_primpart}(x)$. Returns **NULL** if loss of accuracy. Not stack-clean but suitable for **gc_upto**. Allows x in compact representation, in which case **Q_primpart** is taken componentwise.

GEN nf_cxlog_normalize(GEN nf, GEN x, long prec) an *nf* structure attached to a number field K and x from **nf_cxlog**(*nf*, X) (a column vector of complex logarithmic embeddings with

$r_1 + r_2$ components) and let $e = (e_1, \dots, e_{r_1+r_2})$. Return

$$x - \frac{\log(N_{K/\mathbf{Q}}X)}{[K:\mathbf{Q}]}e$$

where the imaginary parts are further normalized modulo $2\pi i \cdot e$.

The composition `nf_cxlog` followed by `nf_cxlog_normalize` is a morphism from $(K^*/\mathbf{Q}_+^*, \times)$ to $((\mathbf{C}/2\pi i\mathbf{Z})^{r_1} \times (\mathbf{C}/4\pi i\mathbf{Z})^{r_2}, +)$. Its real part maps the units $\mathbf{Z}_K^*$ to a lattice in the hyperplane $\sum_i x_i = 0$ in $\mathbf{R}^{r_1+r_2}$.

`GEN nfV_cxlog(GEN nf, GEN x, long prec)` applies `nf_cxlog` to each component of the vector x . Returns NULL if loss of accuracy for even one component. Not stack-clean.

`GEN nflogembed(GEN nf, GEN x, GEN *emb, long prec)` return the vector of real logarithmic embeddings $(e_i \text{Log}|\sigma_i x|)$ where $e_i = 1$ if $i \leq r_1$ and $e_i = 2$ if $r_1 < i \leq r_2$. Returns NULL if loss of accuracy. Not stack-clean. If `emb` is non-NULL set it to $(e_i \sigma_i x)$. Allows x in compact representation, in which case `emb` is returned in compact representation as well, as a factorization matrix (expanding the factorization may overflow exponents).

13.1.22 Maximal order and discriminant, conversion to `nf` structure.

A number field $K = \mathbf{Q}[X]/(T)$ is defined by a monic $T \in \mathbf{Z}[X]$. 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 $\mathbf{Q}[X]/(T_0)$ and let $T = \text{ZX_Q_normalize}(T_0)$, i.e. $T = CT_0(X/L)$ is monic and integral for some $C, Q \in \mathbf{Q}$.

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 $\mathbf{Q}[X]/(T_0)$:

```
RgXV_unscale(S.basis, S.unscale)
```

Instead of passing T (monic `ZX`), one can use the format $[T, \text{list}P]$ as in `nfbasis` or `nfinit`, which computes an order which is maximal at a set of primes, but need not be the maximal order.

The `flag` is an or-ed combination of the binary flags, both of them deprecated:

`nf_PARTIALFACT`: do not try to fully factor `dT` and only look for primes less than `factorlimit`. In that case, the elements in `dTP` and `dKP` need not all be primes. But the resulting `dK`, `index` and `basis` are correct provided there exists no prime $p > \text{factorlimit}$ such that p^2 divides the field discriminant `dK`. This flag is *deprecated*: the $[T, \text{list}P]$ format is safer and more flexible.

`nf_ROUND2`: this flag is *deprecated* and now ignored.

`void nfinit_basic(nfmaxord_t *S, GEN T0)` a wrapper around `nfmaxord` (without the deprecated flag) that also accepts number field structures (`nf`, `bnf`, ...) for `T0`.

`GEN nfmaxord_to_nf(nfmaxord_t *S, GEN ro, long prec)` convert an `nfmaxord_t` to an `nf` structure at precision `prec`, where `ro` is `NULL`. The argument `ro` may also be set to a vector with $r_1 + r_2$ components containing the roots of $S \rightarrow T$ suitably ordered, i.e. first r_1 `t_REAL` roots, then r_2 `t_COMPLEX` representing the conjugate pairs, but this is *strongly discouraged*: the format is error-prone, and it is hard to compute the roots to the right accuracy in order to achieve `prec` accuracy for the `nf`. This function uses the integer basis $S \rightarrow \text{basis}$ as is, *without* performing LLL-reduction. Unless the basis is already known to be reduced, use rather the following higher-level function:

`GEN nfinit_complete(nfmaxord_t *S, long flag, long prec)` convert an `nfmaxord_t` to an `nf` structure at precision `prec`. The `flag` has the same meaning as in `nfinit0`. If $S \rightarrow \text{basis}$ is known to be reduced, it will be faster to use `nfmaxord_to_nf`.

`GEN indexpartial(GEN T, GEN dT)` T a monic separable $\mathbf{Z}[X]$, dT is either `NULL` (no information) or a multiple of the discriminant of T . Let $K = \mathbf{Q}[X]/(T)$ and $\mathbf{Z}_K$ its maximal order. Returns a multiple of the exponent of the quotient group $\mathbf{Z}_K/(\mathbf{Z}[X]/(T))$. In other word, a *denominator* d such that $dx \in \mathbf{Z}[X]/(T)$ for all $x \in \mathbf{Z}_K$.

`GEN FpX_gcd_check(GEN x, GEN y, GEN D)` let x and y be two coprime polynomials with integer coefficients and let D be a factor of the resultant of x and y ; try to factor D by running the Euclidean algorithm on x and y modulo D . This returns `NULL` or a non trivial factor of D . This is the low-level function underlying `poldiscfactors` (applied to x , `ZX_deriv(x)` and the discriminant of x). It succeeds when D has at least two prime divisors p and q such that one sub-resultant of x and y is divisible by p but not by q .

13.1.23 Computing in the class group.

We compute with arbitrary ideal representatives (in any of the various formats seen above), and call

`GEN bnfisprincipal0(GEN bnf, GEN x, long flag)`. The `bnf` structure already contains information about the class group in the form $\oplus_{i=1}^n (\mathbf{Z}/d_i\mathbf{Z})g_i$ for canonical integers d_i (with $d_n \mid \dots \mid d_1$ 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 (nonzero) 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 `flag`, which is an or-ed combination of the following symbolic flags:

- `nf_GEN` tries to compute t . Returns $[e, t]$, with t an empty vector if the computation failed. This flag is normally useless in nontrivial situations since the next two serve analogous purposes in more efficient ways.

- `nf_GENMAT` tries to compute t in factored form, which is much more efficient than `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 `idealmulred`, where the ideal reduction extracts small principal ideals along the way, eventually raised to large powers because of the binary exponentiation technique; the point is to keep this principal part in

factored *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 **bnf** was computed was ridiculously low (and then **bnfinit** should not have succeeded either). Setting/unsetting **nf_GEN** has no effect when this flag is set.

- **nf_GEN_IF_PRINCIPAL** tries to compute t *only* if the ideal is principal ($e = 0$). Returns **gen_0** if the ideal is not principal. Setting/unsetting **nf_GEN** has no effect when this flag is set, but setting/unsetting **nf_GENMAT** is possible.

- **nf_FORCE** in the above, insist on computing t , even if it requires recomputing a **bnf** from scratch. This is a last resort, and normally the accuracy of a **bnf** can be increased without trouble, but it may be that some algebraic information simply cannot be recovered from what we have: see **bnfnewprec**. It should be very rare, though.

In simple cases where you do not care about t , you may use

GEN isprincipal(GEN bnf, GEN x), which is a shortcut for **bnfisprincipal0(bnf, x, 0)**.

The following low-level functions are often more useful:

GEN isprincipalfact(GEN bnf, GEN C, GEN L, GEN f, long flag) is about the same as **bnfisprincipal0** applied to $C \prod L[i]^{f[i]}$, where the $L[i]$ are ideals, the $f[i]$ integers and C is either an ideal or NULL (omitted). Make sure to include **nf_GENMAT** in **flag**!

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). It performs

```
isprincipalfact(bnf,C, L, f, nf_GENMAT);
```

but if it fails to compute t , it just returns a **t_INT**, 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.

void bnftestprimes(GEN bnf, GEN B) is an ingredient to certify unconditionnally a **bnf** computed assuming GRH, cf. **bnfcertify**. Running this function successfully proves that the classes of all prime ideals of norm $\leq B$ belong to the subgroup of the class group generated by the factorbase used to compute the **bnf** (equal to the class group under GRH). If the condition is not true, then (GRH is false and) the function will run forever.

If it is known that primes of norm less than B generate the class group (through variants of Minkowski's convex body or Zimmert's twin classes theorems), then the true class group is proven to be a quotient of **bnf.clgp**.

13.1.24 Floating point embeddings, the T_2 quadratic form.

We assume the nf is a true **nf** structure, attached to a number field K of degree n and signature (r_1, r_2) . We saw that

`GEN nf_get_M(GEN nf)` returns the $(r_1 + r_2) \times n$ matrix M giving the embeddings of K , so that if v is an n -th dimensional **t_COL** representing the element $\sum_{i=1}^n v[i]w_i$ of K , then `RgM_RgC_mul(M, v)` represents the embeddings of v . Its first r_1 components are real numbers (**t_INT**, **t_FRAC** or **t_REAL**, usually the latter), and the last r_2 are complex numbers (usually of **t_COMPLEX**, but not necessarily for embeddings of rational numbers).

`GEN embed_T2(GEN x, long r1)` assuming x is the vector of floating point embeddings of some algebraic number v , i.e.

```
x = RgM_RgC_mul(nf_get_M(nf), 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

```
gnorml2( RgM_RgC_mul(nf_get_G(nf), algtobasis(nf, v)));
```

`GEN embednorm_T2(GEN x, long r1)` analogous to `embed_T2`, applied to the **gnorm** of the floating point embeddings. Assuming that

```
x = 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 **t_REAL** of precision **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 .

13.1.25 Ideal reduction, low level.

In the following routines nf is a true **nf**, attached to a number field K of degree n :

`GEN nf_get_Gtwist(GEN nf, GEN v)` assuming v is a **t_VECSMALL** with $r_1 + r_2$ entries, let

$$||x||_v^2 = \sum_{i=1}^{r_1+r_2} 2^{v_i \varepsilon_i} |\sigma_i(x)|^2,$$

where as usual the σ_i are the (real and) complex embeddings and $\varepsilon_i = 1$, resp. 2, for a real, resp. complex place. This is a twisted variant of the T_2 quadratic form, the standard Euclidean form on $K \otimes \mathbf{R}$. In applications, only the relative size of the v_i will matter.

Let $G_v \in M_n(\mathbf{R})$ be a square matrix such that if $x \in K$ is represented by the column vector X in terms of the fixed **Z**-basis of $\mathbf{Z}_K$ in nf , then

$$||x||_v^2 = {}^t(G_v X) \cdot G_v X.$$

(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 `gc_upto`, but does not collect garbage. For convenience, also allow $v = \text{NULL}$ (`nf_get_roundG`) and v a `t_MAT` as output from the function itself: in both these cases, shallow function.

`GEN nf_get_Gtwist1(GEN nf, long i)`. Simple special case. Returns the twisted G matrix attached 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 ZMs, such that the product Gx is well-defined. This returns a “small” integral linear combinations of the columns of x , given by the LLL-algorithm applied to the lattice Gx . Suitable for `gc_upto`, 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 `nf_get_Gtwist`, and we return a small element a in the lattice (x, T_2) . This is used to implement `idealred`.

`GEN idealpseudomin_nonscalar(GEN x, GEN G)`. As `idealpseudomin`, but we insist of returning a nonscalar a (`ZV_isscalar` is false), if the dimension of x is > 1 .

In the interpretation where x defines an integral ideal on a fixed $\mathbf{Z}_K$ basis whose first element is 1, this means that a is not rational.

`GEN idealpseudominvec(GEN x, GEN G)`. As `idealpseudomin_nonscalar`, but we return about $n^2/2$ nonscalar elements in x with small T_2 -norm, where the dimension of x is n .

`GEN idealpseudored(GEN x, GEN G)`. As `idealpseudomin` but we return the full reduced $\mathbf{Z}$ -basis of x as a `t_MAT` instead of a single vector.

`GEN idealred_elt(GEN nf, GEN x)` shortcut for

`idealpseudomin(x, nf_get_roundG(nf))`

13.1.26 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

`GEN idealred0(GEN nf, GEN x, GEN v)` finds an $a \in K^*$ such that $(a)x$ is integral of small norm and returns it, as an ideal in HNF. What “small” means depends on the parameter v , see the GP description. More precisely, a is returned by `idealpseudomin((x $\mathbf{Z}$ ) $x^v - 1$ ),  $G$ )` divided by $x\mathbf{Z}$, where $x\mathbf{Z} = (x \cap \mathbf{Z})$ and where G is `nf_get_Gtwist(nf, v)` for $v \neq \text{NULL}$ and `nf_get_roundG(nf)` otherwise.

Usually one sets $v = \text{NULL}$ to obtain an element of small T_2 norm in x :

`GEN idealred(GEN nf, GEN x)` is a shortcut for `idealred0(nf, x, NULL)`.

The function `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, `bnfisprincipal0` will recover it).

`GEN idealmoddivisor(GEN bnr, GEN x)` A proof-of-concept implementation, useless in practice. If bnr is attached 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 $(\mathbf{Z}_K/f)^*$.

`GEN famat_to_nf_moddivisor(GEN nf, GEN g, GEN e, GEN bid)` given a true nf attached to a number field K , a bid structure attached 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 $\mathbf{Z}_K$ 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:

`GEN famat_to_nf_modideal_coprime(GEN nf, GEN g, GEN e, GEN f, 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 $(\mathbf{Z}_K/f)^*$ 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`).

`GEN nf_to_Fp_coprime(GEN nf, GEN x, GEN modpr)` this low-level function is variant of `famat_to_nf_modideal_coprime`: nf is a true nf structure, `modpr` is from `zkmodprinit` attached to a prime of degree 1 above the prime number p , and x is either a number field element or a `famat` factorization matrix. We finally assume that no component of x has a denominator p .

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 be the valuation attached to a maximal ideal `pr`:

`GEN famat_makecoprime(GEN nf, GEN g, GEN e, GEN pr, GEN prk, GEN expo)` returns an element in $(\mathbf{Z}_K/\mathfrak{pr}^k)^*$ congruent to the product $\prod g[i]^{e[i]}$, assumed to be globally coprime to `pr`. As above, `expo` is any positive multiple of the exponent of $(\mathbf{Z}_K/\mathfrak{pr}^k)^*$, 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 `famat_to_nf_modideal_coprime`).

`GEN sunits_makecoprime(GEN g, GEN pr, GEN prk)` is a specialized variant that allows to precondition a vector of $g[i]$ assumed to be integral primes or algebraic integers so that it becomes suitable for `famat_to_nf_modideal_coprime` modulo `pr`. This is in particular useful for the output of `bnf_get_sunits`.

`GEN Idealstarprk(GEN nf, GEN pr, long k, long flag)` same as `Idealstar` for $I = \mathfrak{pr}^k$. The `nf` argument is a true nf structure.

13.1.27 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

`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, the single most useful value of `flag` is `nf_INIT` to initialize a proper bnr : omitting `nf_GEN` saves a lot of time and will not adversely affect any class field theoretic function; adding `nf_GEN` makes debugging easier. The flag

0 allows to compute only the ray class group structure but will gain little time; if we only need the *order* of the ray class group, then `bnrclassno` is fastest.

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 `NULL`. It is also allowed to replace H by a character χ of the ray class group modulo *mod*: it represents the subgroup $\text{Ker}\chi$.

`GEN bnr_subgroup_check(GEN bnr, GEN H, GEN *pdeg)` given a *bnr* attached to a modulus *mod*, check whether H represents a congruence subgroup (of the ray class group modulo *mod*) and returns a normalized representation: `NULL` for the trivial subgroup, or in HNF, reduced modulo the elementary divisors of the ray class group. In particular, if H is a character of the ray class group, the returned value is the character kernel. If *pdeg* is not `NULL`, **pdeg* is set to the degree of the attached class field: the index of H in the ray class group.

`void bnr_subgroup_sanitize(GEN *pbnr, GEN *pH)` given a *bnr* and a congruence subgroup H (a subgroup of $G = \text{Cl}_f(K)$), make sanity checks and compute the subgroup conductor. Then replace the pair to match the conductor: the *bnr* has the right conductor as modulus, and the subgroup is normalized. Instead of a *bnr*, this function also accepts a *bnf* (gets replaced by the *bnr* with trivial conductor). Instead of a subgroup, the function also accepts an integer N (replaced by $\text{Cl}_f(K)^N$) or a character (replaced by its kernel). The *bnr* (or rather the underlying ray class group) is computed modulo the exponent of the Galois group G/H . Not stack-clean.

`void bnr_char_sanitize(GEN *pbnr, GEN *pchi)` same idea as `bnr_subgroup_sanitize`: we are given a *bnr* and a ray class character, make sanity checks and update the data to use the conductor as modulus. The *bnr* (or rather the underlying ray class group) is computed modulo the order of the character. Not stack-clean.

`void bnr_vecchar_sanitize(GEN *pbnr, GEN *pCHI)` same idea as `bnr_char_sanitize`: we are given a *bnr* and a non empty vector of ray class characters of the *same* conductor, make sanity checks and update the data to use the conductor as modulus. to use the conductor as modulus. The *bnr* (or rather the underlying ray class group) is computed modulo the lcm of all character orders. Not stack-clean.

`void bnr_sanitize(GEN *pbnr)` sanitize its argument to make sure we get a *bnr* on output (a *bnf* gets replaced by the *bnr* with trivial conductor). This is called as a preliminary step in all the above `sanitize` functions.

`GEN bnrconductor(GEN bnr, GEN H, long flag)` see the documentation of the `GP` function.

`GEN bnrconductor_factored(GEN bnr, GEN H)` return a pair $[F, fa]$ where F is the conductor of H and fa is the factorization of the finite part of the conductor. Shallow function. The argument H goes through `bnr_subgroup_check`, so can be a congruence subgroup, a ray class character, etc.

`GEN bnrconductor_factored_arch(GEN bnr, GEN H, GEN *parch)` H must a congruence subgroup with respect to *bnr* in normalized form, as returned by `bnr_subgroup_check`. This function returns the factorization fa of the finite part of the conductor or `NULL` if fa coincides with the finite part of the *bnr* modulus. If *parch* is not `NULL`, then **parch* is set to the conductor at infinity, or `NULL` if it coincides with the infinite places in the *bnr* modulus. In particular, the modulus of the *bnr* is the conductor of H if and only if **parch* and fa are `NULL`. Shallow function.

`GEN bnrconductor_raw(GEN bnr, GEN H)` return the conductor of H . Shallow function. The argument H goes through `bnr_subgroup_check`, so can be a congruence subgroup, a ray class character, etc.

`GEN bnrtoprimitive(GEN bnr, GEN H, GEN mod)` H must a congruence subgroup with respect to bnr in normalized form, as returned by `bnr_subgroup_check`. This function computes the conductor F of H and returns `NULL` if it coincides with the bnr modulus. If not, it returns a new bnr structure attached to the modulus F , without generators. If `mod` is not `NULL`, the latter ray class group is computed modulo `mod`-th powers. Not stack-clean.

This is meant to be used in conjunction with `bnrmap` (or `bnrsurjection`), which can then be used to map subgroups or characters from the original bnr to the new one using the `abmap_XXX` functions. Beware that `mod` must be chosen accordingly, with respect to all objects which must be mapped in this way; for instance the lcm of all character orders.

`long bnrconductor(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.

`GEN ideallog_units(GEN bnf, GEN bid)` return the images of the units generators `bnf.tu` and `bnf.tu` in the finite abelian group $(\mathbf{Z}_K/f)^*$ attached to `bid`.

`GEN ideallog_units0(GEN bnf, GEN bid, GEN N)` let $G = (\mathbf{Z}_K/f)^*$ be the finite abelian group attached to `bid`. Return the images of the units generators `bnf.tu` and `bnf.tu` in G/G^N . If N is `NULL`, same as `ideallog_units`.

`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).

`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 `NULL`), return the attached bnr , and set H to the attached subgroup. If `addgen` is 1, make sure that if the bnr needed to be computed, then it contains generators.

13.1.28 Abelian maps. A map $f : A \rightarrow B$ between two abelian groups of finite type is given by a triple: $[M, cyc_A, cyc_B]$, where $cyc_A = [a_1, \dots, a_m]$ and $cyc_B = [b_1, \dots, b_n]$ are the elementary divisors for A and B (see `ZM_snf`) so that $A = \oplus_{i \leq m} (\mathbf{Z}/a_i \mathbf{Z}) g_i$ and $B = \oplus_{j \leq n} (\mathbf{Z}/b_j \mathbf{Z}) G_j$. The matrix M gives the image of the generators g_i in terms of the G_j : $(f(g_i))_{i \leq m} = (G_j)_{j \leq n} \cdot M$. The function `bnrmap` returns such a structure.

`GEN bnrsurjection(GEN BNR, GEN bnr)` BNR and bnr defined over the same field K , for moduli F and f with $f \mid F$, returns the canonical surjection $\text{Cl}_K(F) \rightarrow \text{Cl}_K(f)$ as an abelian map. I.e., a triple $[M, cyc_F, cyc_f]$. M gives the image of the fixed ray class group generators of BNR in terms of the ones in bnr , cyc_F and cyc_f are the cyclic structures of BNR and bnr respectively (as per `bnr_get_cyc`). Shallow function.

`GEN abmap_kernel(GEN f)` returns the kernel of the abelian map f , ans a matrix H in HNF: the subgroup is $(g_i) \cdot H$.

`GEN abmap_subgroup_image(GEN f, GEN H)` given a subgroup H of A (its generators are the $(g_i)H$); for efficiency, H should be given in canonical form, i.e., as an HNF left divisor of $diag(a_1, \dots, a_m)$. Returns the subgroup $f(H)$ of B , as an HNF left divisor of $diag(b_1, \dots, b_n)$.

GEN `abmap_char_image`(GEN `f`, GEN `chi`) given a regular (unnormalized) character `chi`, return its image by f , as a regular (unnormalized) character with respect to cyc_B .

GEN `abmap_nchar_image`(GEN `f`, GEN `nchi`) given a normalized character `nchi`, return its image by f , still as a normalized character with respect to cyc_A .

13.1.29 Grunwald–Wang theorem.

GEN `nfgwkummer`(GEN `nf`, GEN `Lpr`, GEN `Ld`, GEN `pl`, long `var`) low-level version of `nfgrunwaldwang`, assuming that `nf` contains suitable roots of unity, and directly using Kummer theory to construct the extension.

GEN `bnfgwgeneric`(GEN `bnf`, GEN `Lpr`, GEN `Ld`, GEN `pl`, long `var`) low-level version of `nfgrunwaldwang`, assuming that `bnf` is a `bnfinit` structure, and calling `rnfkummer` to construct the extension.

13.1.30 Relative equations, Galois conjugates.

GEN `nfissquarefree`(GEN `nf`, GEN `P`) given P a polynomial with coefficients in nf , return 1 if P is squarefree, and 0 otherwise. If allowed (though less efficient) to replace nf by a monic `ZX` defining the field.

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 $\mathbf{Q}$) 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 `NULL` set it to $[h_0, h_1]$, $h_i \in \mathbf{Q}[X]$, where $h_0 + h_1Y$ is the last nonconstant polynomial in the pseudo-Euclidean remainder sequence attached to $A(Y)$ and $B(X - kY)$, leading to $C = \text{Res}_Y(A(Y), B(X - kY))$. In particular $a := -h_0/h_1$ is a root of A in $\mathbf{Q}[X]/(C)$, and $X - ka$ is a root of B .

GEN `nf_rnfeq`(GEN `A`, GEN `B`) wrapper around `rnfequationall` to allow mapping $K \rightarrow L$ (`eltup`) and converting elements of L between absolute and relative form (`reltoabs`, `abstorel`), *without* computing a full rnf structure, which is useful if the relative integral basis is not required. In fact, since A may be a `t_POL` or an nf , the integral basis of the base field is not needed either. The return value is the same as `rnf_get_map`. Shallow function.

GEN `nf_rnfeqsimple`(GEN `A`, GEN `B`) as `nf_rnfeq` except some fields are omitted, so that only the `abstorel` operation is supported. Shallow function.

GEN `eltabstorel`(GEN `rnfeq`, GEN `x`) `rnfeq` is as given by `rnf_get_map` (but in this case `rnfeltabstorel` is more robust), `nf_rnfeq` or `nf_rnfeqsimple`, return x as an element of L/K , i.e. as a `t_POLMOD` with `t_POLMOD` coefficients. Shallow function.

GEN `eltabstorel_lift`(GEN `rnfeq`, GEN `x`) same as `eltabstorel`, except that x is returned in partially lifted form, i.e. as a `t_POL` with `t_POLMOD` coefficients.

GEN `eltreltoabs`(GEN `rnfeq`, GEN `x`) `rnfeq` is as given by `rnf_get_map` (but in this case `rnfeltreltoabs` is more robust) or `nf_rnfeq`, return x in absolute form.

GEN `nf_nfzk`(GEN `nf`, GEN `rnfeq`) `rnfeq` as given by `nf_rnfeq`, `nf` a true nf structure, return a suitable representation of `nf.zk` allowing quick computation of the map $K \rightarrow L$ by the function

`nfeltup`, *without* computing a full *rnf* structure, which is useful if the relative integral basis is not required. The computed value is the same as in `rnf_get_nfzk`. Shallow function.

`GEN nfeltup(GEN nf, GEN x, GEN zknf) zknf` and is initialized by `nf_nfzk` or `rnf_get_nfzk` (but in this case `nfeltup` is more robust); `nf` is a true *nf* structure for K , returns $x \in K$ as a (lifted) element of L , in absolute form.

`GEN rnfdisc_factored(GEN nf, GEN pol, GEN *pd)` variant of `rnfdisc` returning the relative discriminant ideal *factorization*, and setting `*pd` to the discriminant as an element in $K^*/(K^*)^2$. Shallow function. The argument `nf` is a true *nf* structure.

`GEN Rg_nffix(const char *f, GEN T, GEN c, int lift)` given a ZX T and a “coefficient” c supposedly belonging to $\mathbf{Q}[y]/(T)$, 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 `t_INT`, `t_FRAC`, `t_POL` in the variable y with rational coefficients, or `t_POLMOD` modulo T which lift to a rational `t_POL` as above. The cleanup consists in the following improvements:

- `t_POL` coefficients are reduced modulo T .
- `t_POL` and `t_POLMOD` belonging to $\mathbf{Q}$ are converted to rationals, `t_INT` or `t_FRAC`.
- if `lift` is nonzero, convert `t_POLMOD` to `t_POL`, and otherwise convert `t_POL` to `t_POLMODs` modulo T .

`GEN RgX_nffix(const char *f, GEN T, GEN P, 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 ZX and returns a cleaned up version of P . This checks whether P is indeed a `t_POL` with variable compatible with coefficients in $\mathbf{Q}[y]/(T)$, i.e.

$$\text{varncmp}(\text{varn}(P), \text{varn}(T)) < 0$$

and applies `Rg_nffix` to each coefficient.

`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 in `polmod_nffix`, where the relative extension is explicitly defined as $L = (\mathbf{Q}[y]/(T))[x]/(P)$, instead of by an *rnf* structure.

`long numberofconjugates(GEN T, long pinit)` returns a quick multiple for the number of $\mathbf{Q}$ -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.

`GEN nfroots_if_split(GEN *pt, GEN T)` let `*pt` point either to a number field structure or an irreducible ZX, defining a number field K . Given T a monic squarefree polynomial with coefficients in $\mathbf{Z}_K$, return the list of roots of `pol` in K if the polynomial splits completely, and `NULL` otherwise. In other words, this checks whether $K[X]/(T)$ is normal over K (hence Galois since T is separable by assumption).

In the case where `*pT` is a `ZX`, the function has to compute internally a conditional `nf` attached to K , whose `nf.zk` may not define the maximal order $\mathbf{Z}_K$ (see `nfroots`); `*pT` is then replaced by the conditional `nf` to avoid losing that information.

`GEN rnfabelianconngen(GEN nf, GEN P)` *nf* being a number field structure attached to K and P being an irreducible polynomial in $K[X]$. This function returns `gen_0` if $L = K[X]/(P)$ is not abelian over K , else it returns a pair (g, o) where g is a vector of K -automorphisms of L generating the abelian group $G = \text{Gal}(L/K)$ and o is a `t_VECSMALL` of the same length giving the relative orders of the g_i : $o[1]$ is the order of g_1 and for $i \geq 2$, $o[i]$ is the order of g_i in $G/(g_1, \dots, g_{i-1})$. The length need not be minimal: the $o[i]$ need not be the elementary divisors of G .

13.1.31 Units.

`GEN nfrootsof1(GEN nf)` returns a two-component vector $[w, z]$ where w is the number of roots of unity in the number field *nf*, and z is a primitive w -th root of unity.

`GEN nfcyclotomicunits(GEN nf, GEN zu)` where `zu` is as output by `nfrootsof1(nf)`, return the vector of the cyclotomic units in *nf* expressed over the integral basis. If $\zeta = \zeta_n$ belongs to the base field (n maximal), this function returns

- (when n is not a prime power) the $\zeta^a - 1$, for all $1 \leq a < n/2$ such that $n/(a, n)$ is not a prime power and a is a strict divisor of n .
- (all n) for p prime, $v_p(n) = k > 0$, the $(z^a - 1)/(z - 1)$, where $z = \zeta^{n/p^k}$, for all $1 < a \leq (p^k - 1)/2$, $(p, a) = 1$.

These are independent modulo torsion if n is a prime power, but not necessarily so otherwise.

`GEN sunits_mod_units(GEN bnf, GEN S)` return independent generators for $U_S(K)/U(K)$.

13.1.32 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 idealstar0(GEN nf, GEN x, long flag)` short for `idealstarmod(nf, ideal, flag, NULL)`. Use `Idealstarmod` or `Idealstar`.

`GEN bnrinit0(GEN bnf, GEN ideal, long flag)` short for `bnrinitmod(bnf,ideal,flag,NULL)`. Use `Buchray` or `Buchraymod`.

`GEN buchimag(GEN D, GEN c1, GEN c2, GEN gCO)` short for

`Buchquad(D,gtodouble(c1),gtodouble(c2), /*ignored*/0)`

`GEN buchreal(GEN D, GEN gsens, GEN c1, GEN c2, GEN RELSUP, long prec)` short for

`Buchquad(D,gtodouble(c1),gtodouble(c2), 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 bnrprincipal.
 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 polred0(GEN x, long flag, GEN p)
 GEN polredabs(GEN x)
 GEN polredabs2(GEN x)
 GEN polredabsall(GEN x, long flun)
 Superseded by bnrdisclist0:
 GEN discrayabslist(GEN bnf, GEN L)
 GEN discrayabslistarch(GEN bnf, GEN arch, long bound)
 Superseded by idealappr (*flag* ignored)
 GEN idealappr0(GEN nf, GEN x, long flag)
 Superseded by bnrconductor_raw or bnrconductor_factored, bnrconductor_factored_arch:
 GEN bnrconductorofchar(GEN bnr, GEN chi)

13.2 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.

13.2.1 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.pe`.
 GEN gal_get_mod(GEN gal) returns `gal.mod`.
 GEN gal_get_roots(GEN gal) returns `gal.roots`.
 GEN gal_get_invvdm(GEN gal) `gal[4]`.
 GEN gal_get_den(GEN gal) return `gal[5]`.
 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`.

13.2.2 Miscellaneous functions.

`GEN nfgaloispermtobasis(GEN nf, GEN gal)` return the images of the field generator by the automorphisms `gal.orders` expressed on the integral basis `nf.zk`.

`GEN nfgaloismatrix(GEN nf, GEN s)` returns the ZM attached to the automorphism s , seen as a linear operator expressed on the number field integer basis. This allows to use

```
M = nfgaloismatrix(nf, s);  
sx = ZM_ZC_mul(M, x); /* or RgM_RgC_mul(M, x) if x is not integral */
```

instead of

```
sx = nfgaloisapply(nf, s, x);
```

for an algebraic integer x .

`GEN nfgaloismatrixapply(GEN nf, GEN M, GEN x)` given an automorphism M in `nfgaloismatrix` form, return the image of x under the automorphism. Variant of `galoisapply`.

13.3 Quadratic number fields and quadratic forms.

13.3.1 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), unless `mod4` is NULL.

`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.

13.3.2 Class number.

Given a D congruent to 0 or 1 modulo 4, let $h(D)$ denote the class number of the order of discriminant D . The function `quadclassunit` uses index calculus and computes $h(D)$ in subexponential time in $\log |D|$ but it assumes the truth of the GRH. For imaginary quadratic orders, it is also comparatively slow for *small* values, say $|D| \leq 10^{18}$. Here are some alternatives:

`GEN classno(GEN D)` corresponds to `qfbclassno(D,0)` and is only useful for $D < 0$, uses a baby-step giant-step technique and runs in time $O(D^{1/4})$. The result is guaranteed correct for $|D| < 2 \cdot 10^{10}$ and fastest in that range. For larger values of $|D|$, the algorithm is no longer rigorous and may give incorrect results (we know no concrete example); it also becomes relatively less interesting compared to `quadclassunit`.

`GEN classno2(GEN D)` corresponds to `qfbclassno(D,1)` and runs in time $O(D^{1/2})$; the function is provided for testing purposes only since it is never competitive.

`GEN quadclassnoF(GEN D, GEN *pd)` returns the positive integer $h(D)/h(d)$ where d is the fundamental discriminant attached to D . If `pd` is not NULL, set $*pd$ to d .

`GEN quadclassno(GEN D)` returns $h(D)$ using Buchmann's algorithm on the order of discriminant D . If D is not fundamental, it will usually be faster to call `coredisc2_fact` and `quadclassnoF_fact` to reduce to this case first.

`long quadclassno(long D)` returns $h(D)$ using Buchmann's algorithm on the order of discriminant D .

`ulong unegquadclassnoF(ulong x, long *pd)` returns $h(-x)/h(d)$. Set `*pd` to d .

`ulong uposquadclassnoF(ulong x, long *pd)` returns $h(x)/h(d)$. Set `*pd` to d .

`GEN quadclassnoF_fact(GEN D, GEN P, GEN E)` let D be a fundamental discriminant, and $f = \prod_i P[i]^{E[i]}$ be a positive conductor for the order of discriminant Df^2 (P is a `ZV` and E is a `ZV` or `zv`). Returns the positive integer

$$h(Df^2)/h(D) = \frac{f}{[O_D^\times : O_{Df^2}^\times]} \prod_{p|f} (1 - (D/p)p^{-1}).$$

`ulong uquadclassnoF_fact(ulong d, long s, GEN P, GEN E)` let $s = 1$ or -1 be a sign, $D = sd$ be a fundamental discriminant, and $f = \prod_i P[i]^{E[i]}$ be a positive conductor for the order of discriminant Df^2 (P and E are `t_VECSMALL`). Returns the integer

$$h(Df^2)/h(d) = \frac{f}{[O_D^\times : O_{Df^2}^\times]} \prod_{p|f} (1 - (D/p)p^{-1}).$$

`GEN hclassno(GEN d)` returns the Hurwitz-Kronecker class number $H(d)$. These play a central role in trace formulas and are usually needed for many consecutive values of d . Thus, the function uses a cache so that later calls for *small* consecutive values of d are instantaneous, see `getcache`. Large values of d ($d > 500000$) call `quadclassunit` individually and are not memoized.

`GEN hclassnoF_fact(GEN P, GEN E, GEN D)` return $H(Df^2)/H(D)$ assuming D is a negative fundamental discriminant, where the conductor f is given in factored form: P (`ZV`) is the list of prime divisors of f and E (`t_VECSMALL`) their multiplicities.

`long uhclassnoF_fact(GEN f, long D)` return $H(Df^2)/H(D)$ assuming D is a negative fundamental discriminant and $d = Df^2$ is an `ulong` and `f` is `factoru(d)`.

`GEN hclassno6(GEN d)` assuming $d > 0$, returns the integer $6H(d)$. This is a low-level function behind `hclassno`.

`ulong hclassno6u(ulong d)` assuming $d > 0$, returns the integer $6H(d)$. Using this function creates (or extends) caches of Hurwitz class numbers and `Corediscs` of negative integers to speed up consecutive or repeated calls (see `getcache`). If this is a problem, use:

`ulong hclassno6u_no_cache(ulong d)` as `hclassno6u` without creating caches. Existing caches will be used.

13.3.3 t_QFB.

The functions in this section operate on binary quadratic forms of type `t_QFB`. When specified, a `t_QFB` argument q attached to an indefinite form can be replaced by the pair $[q, d]$ where the `t_REAL` d is Shanks's distance.

`GEN qfb_1(GEN q)` given a `t_QFB` q , return the unit form q^0 .

`int qfb_equal1(GEN q)` returns 1 if the `t_QFB` q is the unit form.

13.3.3.1 Reduction.

GEN `qfbred`(GEN `x`) reduction of a `t_QFB` x . Also allow extended indefinite forms.

GEN `qfbred_i`(GEN `x`) internal version of `qfbred`: assume x is a `t_QFB`.

GEN `qfi_red`(GEN `x`) internal version of `qfbred`: assume x is an imaginary `t_QFB`.

GEN `qfr_red`(GEN `x`) internal version of `qfbred`: assume x is a real `t_QFB` and returns a `t_QFB`.

GEN `qfr_cf`(GEN `x`) internal version of `contfrac(x)`: assume x is a real `t_QFB`.

GEN `qfr_boundcf`(GEN `x`, long `n`) internal version of `contfrac(x, n)`: assume x is a real `t_QFB`.

13.3.3.2 Composition.

GEN `qfbcomp`(GEN `x`, GEN `y`) compose the two `t_QFB` x and y (with same discriminant), then reduce the result. This is the same as `gmul(x, y)`. Also allow extended indefinite forms.

GEN `qfbcomp_i`(GEN `x`, GEN `y`) internal version of `qfbcomp`: assume x and y are `t_QFB` of the same discriminant.

GEN `qfbsqr`(GEN `x`) as `qfbcomp(x, x)`.

GEN `qfbsqr_i`(GEN `x`) as `qfbcomp_i(x, y)`.

Same as above, *without* reducing the result:

GEN `qfbcomprow`(GEN `x`, GEN `y`) compose two `t_QFB`s, without reducing the result. Also allow extended indefinite forms.

GEN `qfbcomprow_i`(GEN `x`, GEN `y`) internal version of `qfbcomprow`: assume x and y are `t_QFB` of the same discriminant.

13.3.3.3 Powering.

GEN `qfbpow`(GEN `x`, GEN `n`) computes x^n and reduce the result. Also allow extended indefinite forms.

GEN `qfbpows`(GEN `x`, long `n`) computes x^n and reduce the result. Also allow extended indefinite forms.

GEN `qfbpow_i`(GEN `x`, GEN `n`) internal version of `qfbcomp`. Assume x is a `QFB`.

GEN `qfbpowrow`(GEN `x`, long `n`) compute x^n (pure composition, no reduction), for a `t_QFB` x . Also allow indefinite forms.

13.3.3.4 Order and discrete log for imaginary qfb.

GEN `qfi_order`(GEN `q`, GEN `o`) assuming that the imaginary `t_QFB` q has order dividing o , compute its order in the class group. The order can be given in all formats allowed by generic discrete log functions, the preferred format being [`ord`, `fa`] (`t_INT` and its factorization).

GEN `qfi_log`(GEN `a`, GEN `g`, GEN `o`) given an imaginary `t_QFB` a and assuming that the `t_QFB` g has order o , compute an integer k such that $a^k = g$. Return `cgetg(1, t_VEC)` if there are no solutions. Uses a generic Pollig-Hellman algorithm, then either Shanks (small o) or Pollard rho (large o) method. The order can be given in all formats allowed by generic discrete log functions, the preferred format being [`ord`, `fa`] (`t_INT` and its factorization).

GEN `qfi_Shanks`(GEN `a`, GEN `g`, long `n`) given an imaginary `t_QFB` a and assuming that the `t_QFB` g has (small) order n , compute an integer k such that $a^k = g$. Return `cgetg(1, t_VEC)` if there are no solutions. Directly uses Shanks algorithm, which is inefficient when n is composite.

13.3.3.5 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 an imaginary `t_QFB` 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 real `t_QFB` 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 $d > 0$ is congruent to 0 or 3 modulo 4. 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 $x^2 + dy^2 = 4p$.

`long cornacchia2_sqrt(GEN d, GEN p, GEN b, GEN *px, GEN *py)` as `cornacchia2`, where $p > 2$ and b is the smallest squareroot of d modulo p .

13.3.3.6 Prime forms.

`GEN primeform_u(GEN D, ulong p) t_QFB` of discriminant D whose first coefficient is the prime p , assuming $(D/p) \geq 0$.

`GEN primeform(GEN D, GEN p)`

13.3.4 Efficient real quadratic forms. Unfortunately, real `t_QFB`s are very inefficient, and are only provided for backward compatibility.

- they do not contain needed quantities, which are thus constantly recomputed (the discriminant square root $\sqrt{D}$ 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_QFB` will do whenever a `qfr3` is expected. Routines using these objects 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 `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_compraw`(GEN `x`, GEN `y`) as `qfr3_comp`, without 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; `qfr3_red` just performs reduction steps until we hit a reduced form.

GEN `qfr3_to_qfr`(GEN `x`, GEN `d`) recover an ordinary `t_QFB` from the `qfr3` x , adding discriminant component d .

Before we explain `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 $\rho(a, b, c) = (c, -b \bmod 2c, *)$; 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_compraw`(GEN `x`, GEN `y`) as `qfr5_comp`, without reducing the result.

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 (true Shanks's distance).

GEN `qfr_to_qfr5`(GEN `x`, long `prec`) convert a real `t_QFB` to a `qfr5` with initial trivial distance component ($= 1$).

GEN `qfr5_to_qfr`(GEN `x`, GEN `d`), assume x is a `qfr5` and d is NULL or the original distance component of some real `t_QFB`. Convert x to a `t_QFB`, with the correct (logarithmic) distance component if d is not NULL.

13.4 Linear algebra over $\mathbf{Z}$.

13.4.1 Hermite and Smith Normal Forms.

GEN `ZM_hnf`(GEN `x`) returns the upper triangular Hermite Normal Form of the ZM `x` (removing 0 columns), using the `ZM_hnfall` algorithm. If you want the true HNF, use `ZM_hnfall(x, NULL, 0)`.

GEN `ZM_hnfmod`(GEN `x`, GEN `d`) returns the HNF of the ZM `x` (removing 0 columns), assuming the `t_INT` `d` is a multiple of the determinant of `x`. This is usually faster than `ZM_hnf` (and uses less memory) if the dimension is large, > 50 say.

GEN `ZM_hnfintersectmod`(GEN `L1`, GEN `L2`, GEN `d`) returns the HNF of the intersection of the $\mathbf{Z}$ -modules with bases `L1` and `L2`, assuming that this intersection contains $d\mathbf{Z}^n$.

GEN `ZM_hnfmodid`(GEN `x`, GEN `d`) returns the HNF of the ZM `x` concatenated with the diagonal matrix with diagonal `d`, where `d` is a vector of integers of compatible dimension. Variant: if `d` is a `t_INT`, then concatenate `dId`.

GEN `ZM_hnfmodprime`(GEN `x`, GEN `p`) returns the HNF of the matrix $(x \mid p\text{Id})$ (removing 0 columns), for a ZM `x` and a prime number `p`. The algorithm involves only $\mathbf{F}_p$ -linear algebra and is faster than `ZM_hnfmodid` (which will call it when `d` is prime).

GEN `ZM_hnfmodall`(GEN `x`, GEN `d`, long `flag`) low-level function underlying the `ZM_hnfmod` variants. If `flag` is 0, calls `ZM_hnfmod(x,d)`; `flag` is an or-ed combination of:

- `hnf_MODID` call `ZM_hnfmodid` instead of `ZM_hnfmod`,
- `hnf_PART` return as soon as we obtain an upper triangular matrix, saving time. The pivots are nonnegative 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.
- `hnf_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.

GEN `ZM_hnfmodall_i`(GEN `x`, GEN `d`, long `flag`) as `ZM_hnfmodall` without final garbage collection. Not stack-clean.

GEN `ZM_hnfall`(GEN `x`, GEN `*U`, long `remove`) returns the upper triangular HNF H of the ZM `x`; if `U` is not NULL, set it to the matrix U such that $xU = H$. If `remove = 0`, H is the true HNF, including 0 columns; if `remove = 1`, delete the 0 columns from H but do not update U accordingly (so that the integer kernel may still be recovered): we no longer have $xU = H$; if `remove = 2`, remove 0 columns from H and update U so that $xU = H$. The matrix U 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.

GEN `ZM_hnfall_i`(GEN `x`, GEN `*U`, long `remove`) as `ZM_hnfall` without final garbage collection. Not stack-clean.

GEN `ZM_hnfperm`(GEN `A`, GEN `*ptU`, GEN `*ptperm`) returns the hnf $H = PAU$ of the matrix PA , where P is a suitable permutation matrix, and $U \in \text{Gl}_n(\mathbf{Z})$. P is chosen so as to (heuristically) minimize the size of U ; in this respect it is less efficient than `ZM_hnfall` but usually faster. Set `*ptU` to U and `*ptperm` to a `t_VECSMALL` representing the row permutation attached to $P = (\delta_{i,\text{perm}[i]}$. If

`ptU` is set to `NULL`, U is not computed, saving some time; although useless, setting `ptperm` to `NULL` is also allowed.

`GEN ZM_hnf_knapsack(GEN x)` given a `ZM` x , compute its HNF h . Return h if it has the knapsack property: every column contains only zeroes and ones and each row contains a single 1; return `NULL` otherwise. Not stack-clean.

`GEN ZM_hnfall1(GEN x, GEN *U, int remove)` returns the HNF H of the `ZM` x ; if U is not `NULL`, set it to the matrix U such that $xU = H$. 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.

`GEN ZM_hnfcenter(GEN M)`. Given a `ZM` in HNF M , update it in place so that nondiagonal entries belong to a system of *centered* residues. Not stack-clean.

Some direct applications: the following routines apply to upper triangular integral matrices; in practice, these come from HNF algorithms.

`GEN hnf_divscale(GEN A, GEN B, GEN t)` A an upper triangular `ZM`, B a `ZM`, t an integer, such that $C := tA^{-1}B$ is integral. Return C .

`GEN hnf_invscale(GEN A, GEN t)` A an upper triangular `ZM`, t an integer such that $C := tA^{-1}$ is integral. Return C . Special case of `hnf_divscale` when B is the identity matrix.

`GEN hnf_solve(GEN A, GEN B)` A a `ZM` in upper HNF (not necessarily square), B a `ZM` or `ZC`. Return $A^{-1}B$ if it is integral, and `NULL` if it is not.

`GEN hnf_invimage(GEN A, GEN b)` A a `ZM` in upper HNF (not necessarily square), b a `ZC`. Return $A^{-1}B$ if it is integral, and `NULL` if it is not.

`int hnfddivide(GEN A, GEN B)` A and B are two upper triangular `ZM`. Return 1 if $A^{-1}B$ is integral, and 0 otherwise.

`GEN ZC_modhnf(GEN A, GEN H, GEN *pQ)` reduce the integral column vector A modulo the HNF matrix H (of compatible dimensions). The reduction R is a `ZC` satisfying $0 \leq R_i < H_{i,i}$ for all i . If `pQ` is not `NULL`, `*pQ` is set to the `ZC` Q such that $A = HQ + R$. Both Q and R are unique.

The function does not in fact assume that H is in HNF, it needs only be upper triangular with positive diagonal entries.

`GEN ZM_modhnf(GEN A, GEN H, GEN *Q)` reduces the integral matrix A modulo the HNF matrix H (of compatible dimensions). This is the same as applying `ZC_modhnf` to successive columns of A . We still have $A = HQ + R$ as a matrix equality, with unique Q and R , both of them `ZM`.

Smith Normal Form.

GEN ZM_snf(GEN x) returns the Smith Normal Form (vector of elementary divisors) of the ZM x .

GEN ZM_snfall(GEN x, GEN *U, GEN *V) returns ZM_snf(x) and sets U and V to unimodular matrices such that $UxV = D$ (diagonal matrix of elementary divisors). Either (or both) U or V may be NULL in which case the corresponding matrix is not computed.

GEN ZV_snfall(GEN d, GEN *U, GEN *V) here d is a ZV; same as ZM_snfall applied to diagonal(d), but faster.

GEN ZM_snfall_i(GEN x, GEN *U, GEN *V, long flag) low level version of ZM_snfall:

- if the first bit of *flag* is 0, return a diagonal matrix (as in ZM_snfall), else a vector of elementary divisors (as in ZM_snf).

- if the second bit of *flag* is 1, assume that x is invertible and allow U and V to have determinant congruent to 1 modulo d , where d is the largest elementary divisor of x . Rationale: the finite group $G = \mathbf{Z}^n/\mathfrak{S}x$ has exponent d and we are only interested in the action of U, V as they act on G not in genuine unimodular matrices. (See ZM_snf_group.)

void ZM_snfclean(GEN d, GEN U, GEN V) assuming d, U, V come from $d = \text{ZM_snfall}(x, \&U, \&V)$, where U or V may be NULL, cleans up the output in place. This means that elementary divisors equal to 1 are deleted and U, V are updated. This also works when d is a t_VEC of elementary divisors. Shallow function.

void ZV_snfclean(GEN d) assuming d is a t_VEC of elementary divisors, return a shortened version where divisors equal to 1 are deleted. Shallow function; we return d itself if no divisor is 1.

void ZV_snf_trunc(GEN D) given a vector D of elementary divisors (i.e. a ZV such that $d_i \mid d_{i+1}$), truncate it *in place* to leave out the trivial divisors (equal to 1).

GEN ZM_snf_group(GEN H, GEN *U, GEN *Uinv) 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 $g = (g_1, \dots, g_n)$ and a vector $X = (x_i) \in \mathbf{Z}^n$, 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 ZM (not necessarily square) such that $gX = 0$ for some $X \in \mathbf{Z}^n$ if and only if X is in the integer image of H , so that the abelian group is isomorphic to $\mathbf{Z}^n/\text{Im}H$. 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 ($\dots \mid d_2 \mid d_1$), 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_{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 $D \mid (UU_{\text{inv}} - \text{Id})$ and $H \mid (U_{\text{inv}}U - \text{Id})$.

The function returns the vector of elementary divisors (d_i) ; if U is not NULL, it is set to U ; if U_{inv} is not NULL it is set to U_{inv} . The function is not stack-clean.

GEN ZV_snf_group(GEN d, GEN *newU, GEN *newUi), here d is a ZV; same as ZM_snf_group applied to diagonal(d), but faster.

GEN ZV_snf_gcd(GEN v, GEN N) given a vector v of integers and a positive integer N , return the vector whose entries are the gcds $(v[i], N)$. Use case: if v gives the cyclic components for some abelian group G of finite type, then this returns the structure of the finite groupe G/G^N .

The following functions compute the p^n -rank of abelian groups given a vector of elementary divisors and underly snfrank:

long ZV_snf_rank(GEN D, GEN p) assume D is a ZV and p is a t_INT.

long ZV_snf_rank_u(GEN D, ulong p) assume D is a ZV.

long zv_snf_rank(GEN D, ulong p) assume D is a zv.

The following routines underly the various matrixqz variants. In all case the $m \times n$ t_MAT x is assumed to have rational (t_INT and t_FRAC) coefficients

GEN QM_ImQ(GEN x) returns a basis for $\text{Im}_{\mathbf{Q}}x \cap \mathbf{Z}^n$.

GEN QM_ImZ(GEN x) returns a basis for $\text{Im}_{\mathbf{Z}}x \cap \mathbf{Z}^n$.

GEN QM_ImQ_hnf(GEN x) returns an HNF basis for $\text{Im}_{\mathbf{Q}}x \cap \mathbf{Z}^n$.

GEN QM_ImZ_hnf(GEN x) returns an HNF basis for $\text{Im}_{\mathbf{Z}}x \cap \mathbf{Z}^n$.

GEN QM_ImQ_hnfall(GEN A, GEN *pB, long remove) as QM_ImQ_hnf, further returning the transformation matrix as in ZM_hnfall.

GEN QM_ImZ_hnfall(GEN A, GEN *pB, long remove) as QM_ImZ_hnf, further returning the transformation matrix as in ZM_hnfall.

GEN QM_ImQ_all(GEN A, GEN *pB, long remove, long hnf) as QM_ImQ, further returning the transformation matrix as in ZM_hnfall, and returning an HNF basis if hnf is nonzero.

GEN QM_ImZ_all(GEN A, GEN *pB, long remove, long hnf) as QM_ImZ, further returning the transformation matrix as in ZM_hnfall, and returning an HNF basis if hnf is nonzero.

GEN QM_minors_coprime(GEN x, GEN D), assumes $m \geq n$, and returns a matrix in $M_{m,n}(\mathbf{Z})$ with the same $\mathbf{Q}$ -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 ZM, then calls ZM_hnf. Normally useless in library mode.

GEN hnfmmod(GEN x, GEN d) checks whether x is a ZM, then calls ZM_hnfmmod. Normally useless in library mode.

GEN hnfmmodid(GEN x, GEN d) checks whether x is a ZM, then calls ZM_hnfmmodid. Normally useless in library mode.

GEN hnfall(GEN x) calls ZM_hnfall(x, &U, 1) and returns $[H, U]$. Normally useless in library mode.

GEN hnfl1l(GEN x) calls ZM_hnfl1l(x, &U, 1) and returns $[H, U]$. Normally useless in library mode.

GEN `hnfperm`(GEN `x`) calls `ZM_hnfperm(x, &U, &P)` and returns $[H, U, P]$. Normally useless in library mode.

GEN `smith`(GEN `x`) checks whether x is a ZM, then calls `ZM_snf`. Normally useless in library mode.

GEN `smithall`(GEN `x`) checks whether x is a ZM, then calls `ZM_snfall(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 `ZM_hnfall`.

GEN `smithclean`(GEN `z`) cleanup the output of `smithall` or `gsmithall` (delete elementary divisors equal to 1, updating base change matrices).

13.4.2 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 `qflll` and `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

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 is not stack-clean. 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 ${}^t v v$ of some lattice vectors v .
- **LLL_INPLACE.** Incompatible with **LLL_GRAM**. If unset, we return the base change matrix U , otherwise the transformed matrix xU . Implies **LLL_IM** (see below).
- **LLL_KEEP_FIRST.** The first vector b_1 in the output basis is the same one as was originally input. In particular, if b_1 cannot be completed into a basis, the behavior is undefined. Provided b_1 is a shortest nonzero vector of the lattice, the output basis is still LLL-reduced. This flag 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).
- **LLL_COMPATIBLE.** DEPRECATED. This is now a no-op.

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 attached 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^*) attached 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.

`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`.

`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 `gc_upto`, but does not collect garbage.

`GEN Hermite_bound(long n, long prec)` return a majoration of γ_n^n where γ_n is the Hermite constant for lattices of dimension n . The bound is sharp in dimension $n \leq 8$ and $n = 24$.

13.4.3 Linear dependencies.

The following functions underly the `lindep` GP function:

GEN `lindep`(GEN `v`) real/complex entries, guess that about only the 80% leading bits of the input are correct.

GEN `lindep_bit`(GEN `v`, long `b`) real/complex entries, explicit form of the above: multiply the input by 2^b and round to nearest integer before looking for a linear dependency. Truncating dubious bits allows to find better relations.

GEN `lindepfull_bit`(GEN `v`, long `b`) as `lindep_bit` but return a matrix M with $n = \#v$ columns and r rows, with $r = n + 1$ (if v is real) or $n + 2$ (general case) which is an LLL-reduced basis of the lattice formed by concatenating vertically an identity matrix and the floor of $2^b \text{real}(v)$ and $2^b \text{imag}(v)$ if $r = n + 2$. The first n rows of M potentially correspond to relations: whenever the last $r - n$ entries of a column are small. The function `lindep_bit` essentially returns the first column of M truncated to n components.

GEN `lindep_padic`(GEN `v`) p -adic entries.

GEN `lindep_Xadic`(GEN `v`) polynomial entries.

GEN `deplin`(GEN `v`) returns a nonzero kernel vector for a `t_MAT` input.

Deprecated routine:

GEN `lindep2`(GEN `x`, long `dig`) analogous to `lindep_bit`, with `dig` counting decimal digits.

13.4.4 Reduction modulo matrices.

GEN `ZC_hnfdivrem`(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 $x \bmod y$ (whose i -th entry belongs to $[-y_{i,i}/2, y_{i,i}/2[)$. 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_hnfdivrem`. 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_hnfdivrem(x,y,NULL)`.

GEN `ZM_hnfrem`(GEN `x`, GEN `y`) alias for `ZM_hnfdivrem(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 $\mathbf{Z}$ -module Y spanned by y : if the columns of y are denoted by $(y_1, \dots, y_{n-1})$, we return $y_n \equiv v$ modulo 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 $\mathbf{Z}$ -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 $x - y[y^{-1}x]$, which has small entries and differs from x by an integral linear combination of the columns of y . Suitable for `gc_upto`, 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 nonsingular ZM y and call `reducemodinvertible` to find a small representative of $x \bmod y\mathbf{Z}^n$. Suitable for `gc_upto`, but does not collect garbage.

13.5 Finite abelian groups and characters.

13.5.1 Abstract groups.

A finite abelian group G in GP format is given by its Smith Normal Form as a pair $[h, d]$ or triple $[h, d, g]$. Here h is the cardinality of G , (d_i) is the vector of elementary divisors, and (g_i) is a vector of generators. In short, $G = \oplus_{i \leq n} (\mathbf{Z}/d_i\mathbf{Z})g_i$, with $d_n \mid \dots \mid d_2 \mid d_1$ and $\prod d_i = h$.

Let $e(x) := \exp(2i\pi x)$. For ease of exposition, we restrict to complex-valued characters, but everything applies to more general fields K where e denotes a morphism $(\mathbf{Q}, +) \rightarrow (K^*, \times)$ such that $e(a/b)$ denotes a b -th root of unity.

A *character* on the abelian group $\oplus (\mathbf{Z}/d_j\mathbf{Z})g_j$ is given by a row vector $\chi = [a_1, \dots, a_n]$ such that $\chi(\prod g_j^{n_j}) = e(\sum a_j n_j / d_j)$.

A *quasi-normalized character* is of the form $[d, (n_j)]$, such that $\chi(\prod g_j^{x_j}) = \zeta_d^{\sum_j n_j x_j}$, where $\zeta_d = e(1/d)$.

A *normalized character* is a quasi-normalized $[d, (n_j)]$, such that d is *minimal*, i.e., d is the order of `chi`.

`GEN cyc_normalize(GEN d)` shallow function. Given a vector $(d_i)_{i \leq n}$ of elementary divisors for a finite group (no d_i vanish), returns the vector $D = [1]$ if $n = 0$ (trivial group) and $[d_1, d_1/d_2, \dots, d_1/d_n]$ otherwise. This will allow to define characters as $\chi(\prod g_j^{x_j}) = e(\sum_j x_j a_j D_j / D_1)$, with a common denominator D_1 see `char_normalize`.

`GEN char_normalize(GEN chi, GEN ncyc)` shallow function. Given a character `chi` and `ncyc` from `cyc_normalize` above, returns the normalized representation $[d, (n_j)]$, such that $\chi(\prod g_j^{x_j}) = \zeta_d^{\sum_j n_j x_j}$, where $\zeta_d = e(1/d)$ and d is *minimal*. Shallow function.

`GEN char_simplify(GEN D, GEN N)` given a quasi-normalized character $[D, (N_j)]$ such that $\chi(\prod g_j^{x_j}) = \zeta_D^{\sum_j N_j x_j}$, return a normalized character $[d, (n_j)]$ with d *minimal*. Shallow function.

`GEN char_renormalize(GEN nchi, GEN o)` shallow function. Given a quasi-normalized character `nchi` and a multiple D of d , returns the quasi-normalized representation $[D, (N_j)]$, such that $\chi(\prod g_j^{x_j}) = \zeta_D^{\sum_j N_j x_j}$. Shallow function.

`GEN char_denormalize(GEN cyc, GEN d, GEN n)` given a normalized representation $[d, n]$ (where d need not be minimal) of a character on the abelian group with abelian divisors `cyc`, return the

attached character (where the image of each generator g_i is given in terms of roots of unity of different orders `cyc[i]`).

`GEN charconj(GEN cyc, GEN chi)` return the complex conjugate of `chi`.

`GEN charmul(GEN cyc, GEN a, GEN b)` return the product character $a \times b$.

`GEN chardiv(GEN cyc, GEN a, GEN b)` returns the character $a/b = a \times \bar{b}$.

`int char_check(GEN cyc, GEN chi)` return 1 if `chi` is a character compatible with cyclic factors `cyc`, and 0 otherwise.

`GEN cyc2elts(GEN d)` given a `t_VEC` $d = (d_1, \dots, d_n)$ of nonnegative integers, return the vector of all `t_VECSMALLs` of length n whose i -th entry lies in $[0, d_i[$. Assumes that the product of the d_i fits in a `long`.

`long zv_cyc_minimize(GEN d, GEN c, GEN coprime)` given $d = (d_1, \dots, d_n)$, $d_n \mid \dots \mid d_1 \neq 0$ a list of elementary divisors for a finite abelian group as a `t_VECSMALL`, given $c = [g_1, \dots, g_n]$ representing an element in the group, and given a mask `coprime` (as from `coprimes_zv(o)`) representing a list of forbidden congruence classes modulo o , return an integer k such that `coprime[k%o]` is nonzero and $k \cdot c$ is lexicographically minimal. For instance, if c is attached to a Dirichlet character χ of order o via the usual identification $\chi(g_i) = \zeta_{g_i}^{c_i}$, then χ^k is a “canonical” representative in the Galois orbit of χ .

`long zv_cyc_minimal(GEN d, GEN c, GEN coprime)` return 1 if `zv_cyc_minimize` would return $k = 1$, i.e. c is already the canonical representative for the attached character orbit.

13.5.2 Dirichlet characters.

The functions in this section are specific to characters on $(\mathbf{Z}/N\mathbf{Z})^*$. The argument G is a special `bid` structure as returned by `znstar0(N, nf_INIT)`. In this case, there are additional ways to input character via Conrey’s representation. The character `chi` is either a `t_INT` (Conrey label), a `t_COL` (a Conrey logarithm) or a `t_VEC` (generic character on `bid.gen` as explained in the previous subsection). The following low-level functions are called by GP’s generic character functions.

`int zncharcheck(GEN G, GEN chi)` return 1 if `chi` is a valid character and 0 otherwise.

`GEN zncharconj(GEN G, GEN chi)` as `charconj`.

`GEN znchardiv(GEN G, GEN a, GEN b)` as `chardiv`.

`GEN zncharker(GEN G, GEN chi)` as `charker`.

`GEN znchareval(GEN G, GEN chi, GEN n, GEN z)` as `chareval`.

`GEN zncharmul(GEN G, GEN a, GEN b)` as `charmul`.

`GEN zncharpow(GEN G, GEN a, GEN n)` as `charpow`.

`GEN zncharorder(GEN G, GEN chi)` as `charorder`.

The following functions handle characters in Conrey notation (attached to Conrey generators, not `G.gen`):

`int znconrey_check(GEN cyc, GEN chi)` return 1 if `chi` is a valid Conrey logarithm and 0 otherwise.

`GEN znconrey_normalized(GEN G, GEN chi)` return normalized character attached to `chi`, as in `char_normalize` but on Conrey generators.

GEN `znconreyfromchar`(GEN `G`, GEN `chi`) return Conrey logarithm attached to the generic (`t_VEC`, on `G.gen`)

GEN `znconreyfromchar_normalized`(GEN `G`, GEN `chi`) return normalized Conrey character attached to the generic (`t_VEC`, on `G.gen`) character `chi`.

GEN `znconreylog_normalize`(GEN `G`, GEN `m`) given a Conrey logarithm m (`t_COL`), return the attached normalized Conrey character, as in `char_normalize` but on Conrey generators.

GEN `znchar_quad`(GEN `G`, GEN `D`) given a nonzero `t_INT` D congruent to 0, 1 mod 4, return ($D/.$) as a character modulo N , given by a Conrey logarithm (`t_COL`). Assume that $|D|$ divides N .

GEN `Zideallog`(GEN `G`, GEN `x`) return the `znconreylog` of x expressed on `G.gen`, i.e. the ordinary discrete logarithm from `ideallog`.

GEN `ncharvecexpo`(GEN `G`, GEN `nchi`) given $nchi = [d, n]$ a quasi-normalized character (d may be a multiple of the character order), i.e. $\chi(g_i) = e(n[i]/d)$ for all Conrey or SNF generators g_i (as usual, we use SNF generators if n is a `t_VEC` and the Conrey generators otherwise). Return a `t_VECSMALL` v such that $v[i] = -1$ if $(i, N) > 1$ else $\chi(i) = e(v[i]/d)$, $1 \leq i \leq N$.

13.6 Hecke characters.

The functions in this section are specific to Hecke characters. The argument `gc` is a `gchar` structure as returned by `gcharinit(bnf, mod)`, and the character `chi` is a `t_COL` of components on the SNF generators of `gc`.

GEN `eulerf_gchar`(GEN `an`, GEN `p`, long `prec`) `an` being the first component of a Hecke L-function `Ldata` (as output by `lfungchar`) and p a prime number, return the Euler factor at p .

GEN `gchari_lfun`(GEN `gc`, GEN `chi`, GEN `w`) `chi` being a `t_VEC` describing a Hecke character encoded on the internal basis `gc[1]`, return the `Ldata` structure corresponding to the Hecke L-function associated to `chi`.

int `is_gchar_group`(GEN `gc`) return 1 if `gc` is a valid `gchar` structure and 0 otherwise.

GEN `lfungchar`(GEN `gc`, GEN `chi`) return the `Ldata` structure corresponding to the Hecke L-function associated to `chi`.

GEN `vecan_gchar`(GEN `an`, long `n`, long `prec`) `an` being the first component of a Hecke L-function `Ldata` (as output by `lfungchar`), return a `t_VEC` of length n containing the first n Dirichlet coefficients of this L-function, computed to absolute precision `prec`.

13.7 Central simple algebras.

13.7.1 Initialization.

Low-level routines underlying `alginit`; argument `rnf` (resp. `nf`) must be true *rnf* (resp. *nf*) structure. The meaning of *flag* is the same as in `alginit`.

`GEN alg_csa_table(GEN nf, GEN mt, long v, long flag)` central simple algebra defined by a multiplication table over `nf`.

`GEN alg_cyclic(GEN rnf, GEN aut, GEN b, long flag)` cyclic algebra $(L/K, \sigma, b)$.

`GEN alg_hasse(GEN nf, long d, GEN hi, GEN hf, long v, long flag)` algebra defined by local Hasse invariants.

`GEN alg_hilbert(GEN nf, GEN a, GEN b, long v, long flag)` quaternion algebra.

`GEN alg_matrix(GEN nf, long n, long v, long flag)` matrix algebra of degree `n` over `nf`.

`GEN alg_complete(GEN rnf, GEN aut, GEN hf, GEN hi, long flag)` cyclic algebra $(L/K, \sigma, b)$ with b computed from the Hasse invariants.

13.7.2 Type checks.

`void checkalg(GEN a)` raise an exception if a was not initialized by `alginit`.

`void checklat(GEN al, GEN lat)` raise an exception if `lat` is not a valid full lattice in the algebra `al`.

`void checkhasse(GEN nf, GEN hi, GEN hf, long n)` raise an exception if (hi, hf) do not describe valid Hasse invariants of a central simple algebra of degree `n` over `nf`.

`long alg_type(GEN al)` internal function called by `algtype`: assume `al` was created by `alginit` (thereby saving a call to `checkalg`). Return values are symbolic rather than numeric:

- `al_NULL`: not a valid algebra.
- `al_TABLE`: table algebra output by `alhtableinit`.
- `al_CSA`: central simple algebra output by `alginit` and represented by a multiplication table over its center.
- `al_CYCLIC`: central simple algebra output by `alginit` and represented by a cyclic algebra.

`long alg_model(GEN al, GEN x)` given an element x in algebra `al`, check for inconsistencies (raise a type error) and return the representation model used for x :

- `al_ALGEBRAIC`: `basistoalg` form, algebraic representation.
- `al_BASIS`: `algtobasis` form, column vector on the integral basis.
- `al_MATRIX`: matrix with coefficients in an algebra.
- `al_TRIVIAL`: trivial algebra of degree 1; can be understood as both basis or algebraic form (since $e_1 = 1$).

13.7.3 Shallow accessors.

All these routines assume their argument was initialized by `alginitt` and provide minor speedups compared to the GP equivalent. The routines returning a `GEN` are shallow.

`long alg_get_absdim(GEN al)` low-level version of `algabsdim`.

`long alg_get_dim(GEN al)` low-level version of `algdim`.

`long alg_get_degree(GEN al)` low-level version of `algdegree`.

`GEN alg_get_aut(GEN al)` low-level version of `algaut`.

`GEN alg_get_auts(GEN al)`, given a cyclic algebra $al = (L/K, \sigma, b)$ of degree n , returns the vector of σ^i , $1 \leq i < n$.

`GEN alg_get_b(GEN al)` low-level version of `algb`.

`GEN alg_get_basis(GEN al)` low-level version of `algbasis`.

`GEN alg_get_center(GEN al)` low-level version of `algcenter`.

`GEN alg_get_char(GEN al)` low-level version of `algchar`.

`GEN alg_get_hasse_f(GEN al)` low-level version of `alghassef`.

`GEN alg_get_hasse_i(GEN al)` low-level version of `alghassei`.

`GEN alg_get_invbasis(GEN al)` low-level version of `alginvbasis`.

`GEN alg_get_invol(GEN al, GEN x)` low-level version of `alginvol`.

`GEN alg_get_multable(GEN al)` low-level version of `algmultable`.

`GEN alg_get_relmultable(GEN al)` low-level version of `algrelmultable`.

`GEN alg_get_splittingfield(GEN al)` low-level version of `algsplittingfield`.

`GEN alg_get_abssplitting(GEN al)` returns the absolute nf structure attached to the mf returned by `algsplittingfield`.

`GEN alg_get_splitpol(GEN al)` returns the relative polynomial defining the mf returned by `algsplittingfield`.

`GEN alg_get_splittingdata(GEN al)` low-level version of `algsplittingdata`.

`GEN alg_get_splittingbasis(GEN al)` the matrix $Lbas$ from `algsplittingdata`

`GEN alg_get_splittingbasisinv(GEN al)` the matrix $Lbasinv$ from `algsplittingdata`.

`GEN alg_get_tracebasis(GEN al)` returns the traces of the basis elements; used by `algtrace`.

`GEN alglat_get_primbasis(GEN lat)` from the description of lat as λL with $L \subset \mathcal{O}_0$ and $\lambda \in \mathbf{Q}$, returns a basis of L .

`GEN alglat_get_scalar(GEN lat)` from the description of lat as λL with $L \subset \mathcal{O}_0$ and $\lambda \in \mathbf{Q}$, returns λ .

Accessors for `algmodpr` data (reduction map from algebra to $M_k(\mathbf{F}_{p^m})$, see `algmodprinit`).

`GEN algmodpr_get_pr(GEN data)` corresponding prime ideal.

`long algmodpr_get_k(GEN data)` matrix degree k .

`long almodpr_get_m(GEN data)` field degree m .
`GEN almodpr_get_ff(GEN data)` `t_FFELT` 1 in the image finite field.
`GEN almodpr_get_proj(GEN data)` `t_MAT` representing the projection map.
`GEN almodpr_get_lift(GEN data)` `t_MAT` representing the lifting map.
`GEN almodpr_get_tau(GEN data)` multiplication matrix of an anti-uniformizer.
`GEN almodpr_get_p(GEN data)` prime p .
`GEN almodpr_get_T(GEN data)` polynomial T defining the image finite field.

13.7.4 Other low-level functions.

`GEN conjclasses_algcenter(GEN cc, GEN p)` low-level function underlying `alggroupcenter`, where `cc` is the output of `groupelts_to_conjclasses`, and p is either `NULL` or a prime number. Not stack-clean.

`GEN algsimpledec_ss(GEN al, long maps)` assuming that `al` is semisimple, returns the second component of `algsimpledec(al, maps)`.

`GEN alnewprec_shallow(GEN al, long prec)` shallow version of `alnewprec`.

Chapter 14:

Elliptic curves and arithmetic geometry

This chapter is quite short, but is added as a placeholder, since we expect the library to expand in that direction.

14.1 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 `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 (`gadd`, `gmul`, `gdiv` involving them should not give errors).

14.1.1 Types of elliptic curves.

We call a 5-tuple as above an `ell5`; most functions require an `ell` structure, as returned by `ellinit`, which contains additional data (usually dynamically computed as needed), depending on the base field.

`GEN ellinit(GEN E, GEN D, long prec)`, returns an `ell` structure, attached to the elliptic curve E : either an `ell5`, a pair $[a_4, a_6]$ or a `t_STR` in Cremona's notation, e.g. "11a1". The optional D (NULL to omit) describes the domain over which the curve is defined.

14.1.2 Type checking.

`void checkell(GEN e)` raise an error unless e is an `ell`.

`int checkell_i(GEN e)` return 1 if e is an `ell` and 0 otherwise.

`void checkell5(GEN e)` raise an error unless e is an `ell` or an `ell5`.

`void checkellpt(GEN z)` raise an error unless z is a point (either finite or at infinity).

`int checkellpt_i(GEN z)` return 1 if z is a point (either finite or at infinity) and 0 otherwise.

`long ell_get_type(GEN e)` returns the domain type over which the curve is defined, one of

`t_ELL_Q` the field of rational numbers;

`t_ELL_NF` a number field;

`t_ELL_Qp` the field of p -adic numbers, for some prime p ;

`t_ELL_Fp` a prime finite field, base field elements are represented as $\mathbb{F}_p$, i.e. a `t_INT` reduced modulo p ;

`t_ELL_Fq` a nonprime 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 nonprime). Otherwise the function raises a `pari_err_TYPE` exception.

`void checkell_Q(GEN e)` checks whether e is an `ell`, defined over $\mathbb{Q}$. Otherwise the function raises a `pari_err_TYPE` exception.

`void checkell_Qp(GEN e)` checks whether e is an `ell`, defined over some $\mathbb{Q}_p$. Otherwise the function raises a `pari_err_TYPE` exception.

`void checkellisog(GEN v)` raise an error unless v is an isogeny, from `ellisogeny`.

14.1.3 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.

14.1.3.1 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)`

14.1.3.2 Curves over $\mathbf{Q}$.

`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

`int ell_is_integral(GEN e)` return 1 if e is given by an integral model, and 0 otherwise.

`long ellQ_get_CM(GEN e)` if e has CM by a principal imaginary quadratic order, return its discriminant. Else return 0.

`long ellap_CM_fast(GEN e, ulong p, long CM)` assuming that p does not divide the discriminant of E (in particular, E has good reduction at p), and that CM is as given by `ellQ_get_CM`, return the trace of Frobenius for $E/\mathbf{F}_p$. This is meant to quickly compute lots of a_p , esp. when e has CM by a principal quadratic order.

`long ellrootno_global(GEN e)` returns the global root number $c \in \{-1, 1\}$.

`GEN ellheightoo(GEN E, GEN P, long prec)` given $P = [x, y]$ an affine point on E , return

$$\lambda_\infty(P) + \frac{1}{12} \log |\text{disc} E| = \frac{1}{2} \text{real}(z\eta(z)) - \log |\sigma(E, z)| \in \mathbf{R},$$

where $\lambda_\infty(P)$ is the canonical local height at infinity and z is `ellpointtoz`(E, P). This is computed using Mestre's (quadratically convergent) AGM algorithm.

`long ellorder_Q(GEN E, GEN P)` return the order of $P \in E(\mathbf{Q})$, using the impossible value 0 for a point of infinite order. Ultimately called by the generic `ellorder` function.

`GEN point_to_a4a6(GEN E, GEN P, GEN p, GEN *a4)` given $E/\mathbf{Q}$, $p \neq 2, 3$ not dividing the discriminant of E and $P \in E(\mathbf{Q})$ outside the kernel of reduction, return the image of P on the short Weierstrass model $y^2 = x^3 + a_4x + a_6$ isomorphic to the reduction E_p of E at p . Also set `a4` to the a_4 coefficient in the above model. This function allows quick computations modulo varying primes p , avoiding the overhead of `ellinit`(E, p), followed by a change of coordinates. It produces data suitable for `FpE` routines.

`GEN point_to_a4a6_Fl(GEN E, GEN P, ulong p, ulong *pa4)` as `point_to_a4a6`, returning a `Fle`.

`GEN elldatagenerators(GEN E)` returns generators for $E(\mathbf{Q})$ extracted from Cremona's table.

`GEN ellanal_globalred(GEN e, GEN *v)` takes an *ell* over $\mathbf{Q}$ and returns a global minimal model E (in `ellinit` form, over $\mathbf{Q}$) 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.

`GEN ellanal_globalred_all(GEN e, GEN *v, GEN *N, GEN *tam)` as `ellanal_globalred`; further set `*N` to the curve conductor and `*tam` to the product of the local Tamagawa numbers, including the factor at infinity (multiply by the number of connected components of $e(\mathbf{R})$).

`GEN ellintegralmodel(GEN e, GEN *pv)` return an integral model for e (in `ellinit` form, over $\mathbf{Q}$). Set $v = \text{NULL}$ (already integral, we returned e itself), else to the variable change $[u, 0, 0, 0]$ making e integral. We have $u = 1/t$, $t > 1$.

GEN `ellintegralmodel_i`(GEN `e`, GEN `*pv`) shallow version of `ellintegralmodel`.

GEN `ellQtwist_bsdperiod`(GEN `E`, long `s`) let E be a rational elliptic curve given by a minimal model, Λ_E its period lattice, and $s \in \{-1, 1\}$. Let $\Omega_E^\pm$ be the canonical periods in $\sqrt{\pm 1}\mathbf{R}^+$ generating $\Lambda_E \cap \sqrt{\pm 1}\mathbf{R}$. Return Ω_E^+ if $s = 1$ and Ω_E^- if $s = -1$.

GEN `elltors_psylo`(GEN `e`, ulong `p`) as `elltors`, but return the p -Sylow subgroup of the torsion group.

GEN `elleulerf`(GEN `E`, GEN `p`) returns the Euler factor at p of the L -function associated to the curve E defined over a number field.

Deprecated routines.

GEN `elltors0`(GEN `e`, long `flag`) this function is deprecated; use `elltors`

14.1.3.3 Curves over a number field nf .

Let K be the number field over which E is defined, given by a *nf* or *bnf* structure.

GEN `ellnf_get_nf`(GEN `E`) returns the underlying *nf*.

GEN `ellnf_get_bnf`(GEN `x`) returns NULL if K does not contain a *bnf* structure, else return the *bnf*.

GEN `ellnf_vecarea`(GEN `E`) returns the vector of the period lattices areas of all the complex embeddings of E in the same order as `E.nf.roots`.

GEN `ellnf_veceta`(GEN `E`) returns the vector of the quasi-periods of all the complex embeddings of E in the same order as `E.nf.roots`.

GEN `ellnf_vecomega`(GEN `E`) returns the vector of the periods of all the complex embeddings of E in the same order as `E.nf.roots`.

14.1.3.4 Curves over $\mathbf{Q}_p$.

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 attached 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):

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_AGM`(GEN `E`, long `prec`) returns $[a, b, R, v]$, where v is an integer, a, b, R are vectors describing the sequence of 2-isogenous curves $E_i : y^2 = x(x + A_i)(x + A_i - B_i)$, $i \geq 1$ converging to the singular curve $E_\infty : y^2 = x^2(x + M)$. We have $a[i] = A[i]p^v$, $b[i] = B[i]p^v$, $R[i] = A_i - B_i$. These are used in `ellpointtoz` and `ellztopoint`.

GEN `ellQp_L`(GEN `E`, long `prec`) returns the $\mathcal{L}$ -invariant L .

GEN `ellQp_root`(GEN `E`, long `prec`) returns e_1 .

14.1.3.5 Curves over a finite field $\mathbf{F}_q$.

`GEN ellff_get_p(GEN E)` returns the characteristic

`GEN ellff_get_field(GEN E)` returns p if $\mathbf{F}_q$ is a prime field, and a `t_FFELT` belonging to $\mathbf{F}_q$ otherwise.

`GEN ellff_get_card(GEN E)` returns $\#E(\mathbf{F}_q)$

`GEN ellff_get_gens(GEN E)` returns a minimal set of generators for $E(\mathbf{F}_q)$.

`GEN ellff_get_group(GEN E)` returns `ellgroup(E)`.

`GEN ellff_get_m(GEN E)` returns the `t_INT` m as needed by the `gen_ellgroup` function (the order of the pairing required to verify a generating set).

`GEN ellff_get_o(GEN E)` returns $[d, \text{factord}]$, where d is the exponent of $E(\mathbf{F}_q)$.

`GEN ellff_get_D(GEN E)` returns the elementary divisors for $E(\mathbf{F}_q)$ in a form suitable for `gen_ellgens`: either $[d_1]$ or $[d_1, d_2]$, where d_1 is in `ellff_get_o` format.

$[d, \text{factord}]$, where d is the exponent of $E(\mathbf{F}_q)$.

`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 $y^2 = x^3 + A_4x + A_6$, 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

$$y^2 = x^3 + b_2x^2 + A_6,$$

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

$$y^2 = x^3 + 2b_4x + b_6.$$

- If $p = 2$ and the curve is ordinary ($a_1 \neq 0$), return $[A_2, A_6, [a_1^{-1}, da_1^{-2}, 0, (a_4 + d^2)a_1^{-1}]]$, corresponding to

$$y^2 + xy = x^3 + A_2x^2 + A_6,$$

where $d = a_3/a_1$, $a_1^2A_2 = (a_2 + d)$ and

$$a_1^6A_6 = d^3 + a_2d^2 + a_4d + a_6 + (a_4^2 + d^4)a_1^{-2}.$$

- 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

$$y^2 + a_3y = x^3 + A_4x + A_6,$$

where $A_4 = a_2^2 + a_4$, $A_6 = a_2a_4 + a_6$. The value $1/a_3$ is included in the vector since it is frequently needed in computations.

14.1.3.6 Curves over $\mathbf{C}$. (This includes curves over $\mathbf{Q}$!)

`long ellR_get_prec(GEN E)` return the default accuracy to which we must compute approximate results attached to E .

`GEN ellR_ab(GEN E, long prec)` return $[a, b]$

`GEN ellR_omega(GEN x, long prec)` return periods $[\omega_1, \omega_2]$.

`GEN ellR_eta(GEN E, long prec)` return quasi-periods $[\eta_1, \eta_2]$.

`GEN ellR_area(GEN x, long prec)` return the area $(\Im(\omega_1 \overline{\omega_2}))$.

`GEN ellR_roots(GEN E, long prec)` return $[e_1, e_2, e_3]$. If E is defined over $\mathbf{R}$, then e_1 is real. If furthermore $\text{disc}E > 0$, then $e_1 > e_2 > e_3$.

`long ellR_get_sign(GEN E)` if E is defined over $\mathbf{R}$ returns the signe of its discriminant, otherwise return 0.

14.1.4 Points.

`int ell_is_inf(GEN z)` tests whether the point z is the point at infinity.

`GEN ellinf()` returns the point at infinity $[0]$.

14.1.5 Change of variables.

`GEN ellchangeinvert(GEN w)` given a change of variables $w = [u, r, s, t]$, returns the inverse change of variables w' , such that if $E' = \text{ellchangecurve}(E, w)$, then $E = \text{ellchangecurve}(E', w')$.

14.1.6 Generic helper functions.

The naming scheme assumes an affine equation $F(x, y) = f(x) - (y^2 + h(x)y) = 0$ in standard Weierstrass form: $f = x^3 + a_2x^2 + a_4x + a_6$, $h = a_1x + a_3$. Unless mentionned otherwise, these routine assume that all arguments are compatible with generic functions of `gadd` or `gmul` type. In particular they do not handle elements in number field in `nfalgtobasis` format.

`GEN ellbasechar(GEN E)` returns the characteristic of the base ring over which E is defined.

`GEN ec_bmodel(GEN E, long v)` returns the polynomial $4x^3 + b_2x^2 + 2b_4x + b_6$ in the variable v .

`GEN ec_phi2(GEN E, long v)` returns the polynomial $x^4 - b_4x^2 - 2b_6x - b_8$ in the variable v .

`GEN ec_f_evalx(GEN E, GEN x)` returns $f(x)$.

`GEN ec_h_evalx(GEN E, GEN x)` returns $h(x)$.

`GEN ec_dFdx_evalQ(GEN E, GEN Q)` returns $3x^2 + 2a_2x + a_4 - a_1y$, where $Q = [x, y]$.

`GEN ec_dFdy_evalQ(GEN E, GEN Q)` returns $-(2y + a_1x + a_3)$, where $Q = [x, y]$.

`GEN ec_dmFdy_evalQ(GEN e, GEN Q)` returns $2y + a_1x + a_3$, where $Q = [x, y]$.

`GEN ec_2divpol_evalx(GEN E, GEN x)` returns $4x^3 + b_2x^2 + 2b_4x + b_6$. This function supports inputs in `nfalgtobasis` format.

`GEN ec_half_deriv_2divpol(GEN E, long v)` returns $6x^2 + b_2x + b_4$ as a `t_POL` in the variable v .

`GEN ec_half_deriv_2divpol_evalx(GEN E, GEN x)` returns $6x^2 + b_2x + b_4$.

`GEN ec_3divpol_evalx(GEN E, GEN x)` returns $3x^4 + b_2x^2 + 3b_4x^2 + 3b_6x + b_8$.

14.1.7 Functions to handle elliptic curves over finite fields.

14.1.7.1 Tolerant routines.

`GEN ellap(GEN E, GEN p)` given a prime number p and an elliptic curve defined over $\mathbf{Q}$ or $\mathbf{Q}_p$ (assumed integral and minimal at p), computes the trace of Frobenius $a_p = p + 1 - \#E(\mathbf{F}_p)$. If E is defined over a nonprime finite field $\mathbf{F}_q$, ignore p and return $q + 1 - \#E(\mathbf{F}_q)$. When p is implied (E defined over $\mathbf{Q}_p$ or a finite field), p can be omitted (set to `NULL`).

14.1.7.2 Curves defined a nonprime finite field. In this subsection, we assume that `ell_get_type( $E$ )` is `t_ELL_Fq`. (As noted above, a curve defined over $\mathbf{Z}/p\mathbf{Z}$ can be represented as a `t_ELL_Fq`.)

`GEN FF_elltwist(GEN E)` returns the coefficients $[a_1, a_2, a_3, a_4, a_6]$ of the quadratic twist of E .

`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 $E(\mathbf{F}_q)$. 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 $\#E(\mathbf{F}_q)$.

`GEN FF_ellcard_SEA(GEN E, long s)` This function returns $\#E(\mathbf{F}_q)$, using the Schoof-Elkies-Atkin algorithm. Assume $p \neq 2, 3$. The parameter s has the same meaning as in `Fp_ellcard_SEA`.

`GEN FF_ellgens(GEN E)` returns the generators of the group $E(\mathbf{F}_q)$.

`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, GEN *pm)` returns the structure of the Abelian group $E(\mathbf{F}_q)$ and set `*pm` to m (see `gen_ellgens`).

`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 $[m]P = 0$.

14.2 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.

14.2.1 Helper functions.

`GEN elltrace_extension(GEN t, long n, GEN q)` Let E some elliptic curve over $\mathbf{F}_q$ such that the trace of the Frobenius is t , returns the trace of the Frobenius over $\mathbf{F}_q^n$.

14.2.2 Elliptic curves over $\mathbf{F}_p$, $p > 3$.

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 $\mathbf{F}_p$. A **FpE** is a point of $E(\mathbf{F}_p)$. Since an affine point and a_4 determine a 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 .

void Fp_ellj_to_a4a6(**GEN j**, **GEN p**, **GEN *pa4**, **GEN *pa6**) sets ***pa4** to a_4 and ***pa6** to a_6 where a_4 and a_6 define a fixed elliptic curve with j -invariant j .

int Fp_elljissupersingular(**GEN j**, **GEN p**) returns 1 if j is the j -invariant of a supersingular curve over $\mathbf{F}_p$, 0 otherwise.

GEN Fp_ellcard(**GEN a4**, **GEN a6**, **GEN p**) returns the cardinality of the group $E(\mathbf{F}_p)$.

GEN Fp_ellcard_SEA(**GEN a4**, **GEN a6**, **GEN p**, **long s**) This function returns $\#E(\mathbf{F}_p)$, using the Schoof-Elkies-Atkin algorithm. If the **seadata** package is installed, the function will be faster.

The extra flag **s**, if set to a nonzero value, causes the computation to return **gen_0** (an impossible cardinality) if one of the small primes ℓ divides the curve order but does not divide s . For cryptographic applications, where one is usually interested in curves of prime order, setting $s = 1$ efficiently weeds out most uninteresting curves; if curves of order a power of 2 times a prime are acceptable, set $s = 2$. If moreover **s** is negative, similar checks are performed for the twist of the curve.

GEN Fp_ffellcard(**GEN a4**, **GEN a6**, **GEN q**, **long n**, **GEN p**) returns the cardinality of the group $E(\mathbf{F}_q)$ where $q = p^n$.

GEN Fp_ellgroup(**GEN a4**, **GEN a6**, **GEN N**, **GEN p**, **GEN *pm**) returns the group structure D of the group $E(\mathbf{F}_p)$, which is assumed to be of order N and set ***pm** to m .

GEN Fp_ellgens(**GEN a4**, **GEN a6**, **GEN ch**, **GEN D**, **GEN m**, **GEN p**) returns generators of the group $E(\mathbf{F}_p)$ 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 .

void Fp_elltwist(**GEN a4**, **GEN a6**, **GEN p**, **GEN *pa4**, **GEN *pa6**) sets ***pa4** and ***pa6** to the corresponding parameters for the quadratic twist of E . Assume p is an odd prime.

14.2.3 **FpE**.

GEN FpE_add(**GEN P**, **GEN Q**, **GEN a4**, **GEN p**) returns the sum $P + Q$ in the group $E(\mathbf{F}_p)$, 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**) return $n.P$.

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 $E(\mathbf{F}_p)$, 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 $E(\mathbf{F}_p)$, 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 exist, 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 $\mathbf{t_INTMODs}$.

GEN RgE_to_FpE(GEN P, GEN p) returns the FpE obtained by applying RgE_to_Fp coefficientwise.

int RgE_is_FpE(GEN P, GEN *p) check whether P can be mapped to a FpE using RgE_to_FpE by applying RgE_is_Fp coefficient-wise.

14.2.4 Fle. Let p be a prime ulong, and E the elliptic curve given by the equation $E : y^2 = x^3 + a_4x + a_6$, where a_4 and a_6 are ulong. A Fle is either the point at infinity (ellinf()), or a Flv with two components $[x, y]$.

long Fl_elltrace(ulong a4, ulong a6, ulong p) returns the trace t of the Frobenius of $E(\mathbf{F}_p)$. The cardinality of $E(\mathbf{F}_p)$ is thus $p + 1 - t$, which might not fit in an ulong.

long Fl_elltrace_CM(long CM, ulong a4, ulong a6, ulong p) as Fl_elltrace. If CM is 0, use the standard algorithm; otherwise assume the curve has CM by a principal imaginary quadratic order of discriminant CM and use a faster algorithm. Useful when the curve is the reduction of $E/\mathbf{Q}$, which has CM by a principal order, and we need the trace of Frobenius for many distinct p , see ellQ_get_CM.

ulong Fl_elldisc(ulong a4, ulong a6, ulong p) returns the discriminant of the curve E .

ulong Fl_elldisc_pre(ulong a4, ulong a6, ulong p, ulong pi) returns the discriminant of the curve E , assuming pi is the pseudoinverse of p .

ulong Fl_ellj(ulong a4, ulong a6, ulong p) returns the j -invariant of the curve E .

ulong Fl_ellj_pre(ulong a4, ulong a6, ulong p, ulong pi) returns the j -invariant of the curve E , assuming pi is the pseudoinverse of p .

void Fl_ellj_to_a4a6(ulong j, ulong p, ulong *pa4, ulong *pa6) sets *pa4 to a_4 and *pa6 to a_6 where a_4 and a_6 define a fixed elliptic curve with j -invariant j .

void Fl_elltwist(ulong a4, ulong a6, ulong p, ulong *pA4, ulong *pA6) set *pA4 to A_4 and *pA6 to A_6 where A_4 and A_6 define the twist of E .

void Fl_elltwist_disc(ulong a4, ulong a6, ulong D, ulong p, ulong *pA4, ulong *pA6) sets *pA4 to A_4 and *pA6 to A_6 where A_4 and A_6 define the twist of E by the discriminant D .

GEN Fl_ellptors(ulong l, ulong N, ulong a4, ulong a6, ulong p) return a basis of the l -torsion subgroup of E .

```

GEN Fle_add(GEN P, GEN Q, ulong a4, ulong p)
GEN Fle_dbl(GEN P, ulong a4, ulong p)
GEN Fle_sub(GEN P, GEN Q, ulong a4, ulong p)
GEN Fle_mul(GEN P, GEN n, ulong a4, ulong p)
GEN Fle_mulu(GEN P, ulong n, ulong a4, ulong p)
GEN Fle_order(GEN P, GEN o, ulong a4, ulong p)
GEN Fle_log(GEN P, GEN G, GEN o, ulong a4, ulong p)
GEN Fle_tatepairing(GEN P, GEN Q, ulong m, ulong a4, ulong p)
GEN Fle_weilpairing(GEN P, GEN Q, ulong m, ulong a4, ulong p)
GEN random_Fle(ulong a4, ulong a6, ulong p)
GEN random_Fle_pre(ulong a4, ulong a6, ulong p, ulong pi)
GEN Fle_changepoint(GEN x, GEN ch, ulong p), ch is assumed to give the change of coordinates
[ $u, r, s, t$ ] as a t_VECSMALL.
GEN Fle_changepointinv(GEN x, GEN ch, ulong p), as Fle_changepoint

```

14.2.5 FpJ.

Let $p > 3$ be a prime `t_INT`, and E the elliptic curve given by the equation $E : y^2 = x^3 + a_4 \times x + a_6$, where a_4 and a_6 are `t_INT`. A `FpJ` is a `FpV` with three components $[x, y, z]$, representing the affine point $[x/z^2, y/z^3]$ in Jacobian coordinates, the point at infinity being represented by $[1, 1, 0]$. The following must holds: $y^2 = x^3 + a_4xz^4 + a_6z^6$. For all nonzero u , the points $[u^2x, u^3y, uz]$ and $[x, y, z]$ are representing the same affine point.

```

GEN FpJ_add(GEN P, GEN Q, GEN a4, GEN p)
GEN FpJ_dbl(GEN P, GEN a4, GEN p)
GEN FpJ_mul(GEN P, GEN n, GEN a4, GEN p);
GEN FpJ_neg(GEN P, GEN p) return  $-P$ .
GEN FpJ_to_FpE(GEN P, GEN p) return the corresponding FpE.
GEN FpE_to_FpJ(GEN P) return the corresponding FpJ.

```

14.2.6 Flj.

Let $p > 3$ be a prime. Below, pi is assumed to be the pseudoinverse of p (see `get_Fl_red`).

`GEN Fle_to_Flj(GEN P)` convert a `Fle` to an equivalent `Flj`.

`GEN Flj_to_Fle(GEN P, ulong p)` convert a `Flj` to the equivalent `Fle`.

`GEN Flj_to_Fle_pre(GEN P, ulong p, ulong pi)` convert a `Flj` to the equivalent `Fle`.

`GEN Flj_add_pre(GEN P, GEN Q, ulong a4, ulong p, ulong pi)`

`GEN Flj_dbl_pre(GEN P, ulong a4, ulong p, ulong pi)`

`GEN Flj_neg(GEN P, ulong p)` return $-P$.

`GEN Flj_mulu_pre(GEN P, ulong n, ulong a4, ulong p, ulong pi)`

`GEN random_Flj_pre(ulong a4, ulong a6, ulong p, ulong pi)`

`GEN Flj_changepointinv_pre(GEN P, GEN ch, ulong p, ulong pi)` where ch is the `Flv` $[u, r, s, t]$.

`GEN FljV_factorback_pre(GEN P, GEN L, ulong p, ulong pi)`

14.2.7 Elliptic curves over $\mathbf{F}_{2^n}$. Let T be an irreducible `F2x` and E the elliptic curve given by either the equation $E : y^2 + xy = x^3 + a_2x^2 + a_6$, where a_2, a_6 are `F2x` in $\mathbf{F}_2[X]/(T)$ (ordinary case) or $E : y^2 + a_3y = x^3 + a_4x + a_6$, where a_3, a_4, a_6 are `F2x` in $\mathbf{F}_2[X]/(T)$ (supersingular case).

A `F2xqE` is a point of $E(\mathbf{F}_2[X]/(T))$. In the supersingular case, the parameter `a2` is actually the `t_VEC` $[a_3, a_4, a_3^{-1}]$.

`GEN F2xq_ellcard(GEN a2, GEN a6, GEN T)` Return the order of the group $E(\mathbf{F}_2[X]/(T))$.

`GEN F2xq_ellgroup(GEN a2, GEN a6, GEN N, GEN T, GEN *pm)` Return the group structure D of the group $E(\mathbf{F}_2[X]/(T))$, which is assumed to be of order N and set `*pm` to m .

`GEN F2xq_ellgens(GEN a2, GEN a6, GEN ch, GEN D, GEN m, GEN T)` Returns generators of the group $E(\mathbf{F}_2[X]/(T))$ with the base change ch (see `F2xqE.changepoint`), where D and m are as returned by `F2xq_ellgroup`.

`void F2xq_elltwist(GEN a4, GEN a6, GEN T, GEN *a4t, GEN *a6t)` sets `*a4t` and `*a6t` to the parameters of the quadratic twist of E .

14.2.8 F2xqE.

`GEN F2xqE_changepoint(GEN P, GEN m, GEN a2, GEN T)` returns the image Q of the point P on the curve $E : y^2 + x * y = x^3 + a_2x^2 + a_6$ by the coordinate change m (which is a `F2xqV`).

`GEN F2xqE_changepointinv(GEN P, GEN m, GEN a2, GEN T)` 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 `F2xqV`).

`GEN F2xqE_add(GEN P, GEN Q, GEN a2, GEN T)`

`GEN F2xqE_sub(GEN P, GEN Q, GEN a2, GEN T)`

`GEN F2xqE_dbl(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 $E(\mathbf{F}_2[X]/(T))$, where o is a multiple of the order of P , or its factorization.

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 exist, 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 F2xqE obtained by applying Rg_to_F2xq coefficient-wise.

14.2.9 Elliptic curves over $\mathbf{F}_q$, small characteristic $p > 2$. Let $p > 2$ be a prime, T an irreducible Flx mod p , and E the elliptic curve given by the equation $E : y^2 = x^3 + a_4x + a_6$, where a_4 and a_6 are Flx in $\mathbf{F}_p[X]/(T)$. A FlxqE is a point of $E(\mathbf{F}_p[X]/(T))$.

In the special case $p = 3$, ordinary elliptic curves ($j(E) \neq 0$) cannot be represented as above, but admit a model $E : y^2 = x^3 + a_2x^2 + a_6$ with a_2 and a_6 being Flx in $\mathbf{F}_3[X]/(T)$. In that case, the parameter a2 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 .

void Flxq_ellj_to_a4a6(GEN j, GEN T, ulong p, GEN *pa4, GEN *pa6) sets *pa4 to a_4 and *pa6 to a_6 where a_4 and a_6 define a fixed elliptic curve with j -invariant j .

GEN Flxq_ellcard(GEN a4, GEN a6, GEN T, ulong p) returns the order of $E(\mathbf{F}_p[X]/(T))$.

GEN Flxq_ellgroup(GEN a4, GEN a6, GEN N, GEN T, ulong p, GEN *pm) returns the group structure D of the group $E(\mathbf{F}_p[X]/(T))$, which is assumed to be of order N and sets *pm to m .

GEN Flxq_ellgens(GEN a4, GEN a6, GEN ch, GEN D, GEN m, GEN T, ulong p) returns generators of the group $E(\mathbf{F}_p[X]/(T))$ with the base change ch (see FlxqE_changepoint), where D and m are as returned by Flxq_ellgroup.

void Flxq_elltwist(GEN a4, GEN a6, GEN T, ulong p, GEN *pA4, GEN *pA6) sets *pA4 and *pA6 to the corresponding parameters for the quadratic twist of E .

14.2.10 FlxqE.

Let $p > 2$ be a prime number.

GEN FlxqE_changepoint(GEN P, GEN m, GEN a4, GEN T, ulong p) returns the image Q of the point P on the curve $E : y^2 = x^3 + a_4x + a_6$ 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 $E : y^2 = x^3 + a_4x + a_6$ 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 $E(\mathbf{F}_p[X]/(T))$, 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 exist, 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 FlxqE_weilpairing_pre(GEN P, GEN Q, GEN m, GEN a4, GEN T, ulong p, ulong pi)`, where pi is a pseudoinverse of p , or 0 in which case we assume `SMALL_ULONG(p)`.
`GEN RgE_to_FlxqE(GEN P, GEN T, ulong p)` returns the `FlxqE` obtained by applying `Rg_to_Flxq` coefficientwise.
`GEN Flxq_elldivpolmod(GEN a4, GEN a6, long n, GEN h, GEN T, ulong p)` returns the n -division polynomial of the elliptic curve E modulo the polynomial h .

14.2.11 Elliptic curves over $\mathbf{F}_q$, large characteristic .

Let $p > 3$ be a prime number, T an irreducible polynomial mod p , and E the elliptic curve given by the equation $E: y^2 = x^3 + a_4x + a_6$ with a_4 and a_6 in $\mathbf{F}_p[X]/(T)$. A `FpXQ` is a point of $E(\mathbf{F}_p[X]/(T))$.

`GEN FpXQ_ellj(GEN a4, GEN a6, GEN T, GEN p)` returns the j -invariant of the curve E .

`int FpXQ_elljissupersingular(GEN j, GEN T, GEN p)` returns 1 if j is the j -invariant of a supersingular curve over $\mathbf{F}_p[X]/(T)$, 0 otherwise.

`int Fq_elljissupersingular(GEN j, GEN T, GEN p)` as `FpXQ_elljissupersingular` but j can be a `t_INT`.

`GEN ellsupersingularj_FpXQ(GEN T, GEN p)` T being a `ZX` of degree 2, return a random supersingular j -invariant in $\mathbf{F}_p[X]/T$.

`GEN FpXQ_ellcard(GEN a4, GEN a6, GEN T, GEN p)` returns the order of $E(\mathbf{F}_p[X]/(T))$.

`GEN FpXQ_ellcard_supersingular(GEN a4, GEN a6, GEN T, GEN p)` This function returns $\#E(\mathbf{F}_p[X]/(T))$, assuming the curve is supersingular, see `FpXQ_elljissupersingular`.

`GEN Fq_ellcard_supersingular(GEN a4, GEN a6, GEN T, GEN p)` This function is identical to `FpXQ_ellcard_supersingular`, except that it allows T to be `NULL`.

`GEN Fq_ellcard_SEA(GEN a4, GEN a6, GEN q, GEN T, GEN p, long s)` This function returns $\#E(\mathbf{F}_p[X]/(T))$, using the Schoof-Elkies-Atkin algorithm. Assume $p \neq 2, 3$, and q is the cardinality of $\mathbf{F}_p[X]/(T)$. The parameter s has the same meaning as in `Fp_ellcard_SEA`. If the `seadata` package is installed, the function will be faster.

`GEN FpXQV_ellcharpoly(GEN E, GEN T, GEN p)` Let E be a vector $[a_1, a_2, a_3, a_4, a_6]$ defining the plane curve $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ over $\mathbf{F}_p[X]/(T)$, returns the characteristic

polynomial of the Frobenius acting on its first l -adic cohomology group (which can be of degree 0, 1 or 2.)

`GEN FpXQ_ellgroup(GEN a4, GEN a6, GEN N, GEN T, GEN p, GEN *pm)` Return the group structure D of the group $E(\mathbf{F}_p[X]/(T))$, which is assumed to be of order N and set $*pm$ to m .

`GEN FpXQ_ellgens(GEN a4, GEN a6, GEN ch, GEN D, GEN m, GEN T, GEN p)` Returns generators of the group $E(\mathbf{F}_p[X]/(T))$ 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 Fq_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 .

`void FpXQ_elltwist(GEN a4, GEN a6, GEN T, GEN p, GEN *pA4, GEN *pA6)` sets $*pA4$ and $*pA6$ to the corresponding parameters for the quadratic twist of E .

14.2.12 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 $E: y^2 = x^3 + a_4x + a_6$ by the coordinate change m (which is a `FpXQV`).

`GEN FpXQE_changepointinv(GEN P, GEN m, GEN a4, GEN T, GEN p)` returns the image Q on the curve $E: y^2 = x^3 + a_4x + a_6$ of the point P by the inverse of the coordinate change m (which is a `FpXQV`).

`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 exist, 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 $E(\mathbf{F}_p[X]/(T))$, where o is a multiple of the order of P , or its factorization.

`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 `RgE_to_FpXQ` coefficientwise.

`int RgE_is_FpXQE(GEN P, GEN *T, GEN *p)` check whether P can be mapped to a `FpXQE` using `RgE_to_FpXQE` by applying `RgE_is_FpXQ` coefficient-wise.

14.3 Functions related to modular polynomials.

Variants of `polmodular`, returning the modular polynomial of prime level L for the invariant coded by `inv` (0: j , 1: Weber- f , see `polclass` for the full list).

`GEN polmodular_ZXX(long L, long inv, long vx, long vy)` returns a bivariate polynomial in variables vx and vy .

`GEN polmodular_ZM(long L, long inv)` returns a matrix of (integral) coefficients.

`GEN Fp_polmodular_evalx(long L, long inv, GEN J, GEN p, long v, int derivs)` returns the modular polynomial evaluated at J modulo the prime p in the variable v (if `derivs` is nonzero, returns a vector containing the modular polynomial and its first and second derivatives, all evaluated at J modulo p).

14.3.1 Functions related to modular invariants.

`void check_modinv(long inv)` report an error if `inv` is not a valid code for a modular invariant.

`int modinv_good_disc(long inv, long D)` test whether the invariant `inv` is defined for the discriminant D .

`int modinv_good_prime(long inv, long D)` test whether the invariant `inv` is defined for the prime p .

`long modinv_height_factor(long inv)` return the height factor of the modular invariant `inv` with respect to the j -invariant. This is an integer n such that the j -invariant is asymptotically of the order of the n -th power of the invariant `inv`.

`long modinv_is_Weber(long inv)` test whether the invariant `inv` is a power of Weber f .

`long modinv_is_double_eta(long inv)` test whether the invariant `inv` is a double η quotient.

`long disc_best_modinv(long D)` the integer D being a negative discriminant, return the modular invariant compatible with D with the highest height factor.

`GEN Fp_modinv_to_j(GEN x, long inv, GEN p)` Let Φ the modular equation between j and the modular invariant `inv`, return y such that $\Phi(y, x) = 0 \pmod{p}$.

14.4 Other curves.

The following functions deal with hyperelliptic curves in weighted projective space $\mathbf{P}_{(1,d,1)}$, 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 $T \in \mathbf{Z}[X]$ is integral. Returns 1 if the curve is locally soluble over $\mathbf{Q}_p$, 0 otherwise.

`long nf_hyperell_locally_soluble(GEN nf, GEN T, GEN pr)` let K be a number field, attached to `nf`, `pr` a *prid* attached to some maximal ideal $\mathfrak{p}$; assumes that $T \in \mathbf{Z}_K[X]$ is integral. Returns 1 if the curve is locally soluble over $K_{\mathfrak{p}}$. The argument `nf` is a true *nf* structure.

Chapter 15:

L-functions

15.1 Accessors.

```
long is_linit(GEN data)
GEN ldata_get_an(GEN ldata)
GEN ldata_get_dual(GEN ldata)
long ldata_isreal(GEN ldata)
GEN ldata_get_gammavec(GEN ldata)
long ldata_get_degree(GEN ldata)
GEN ldata_get_k(GEN ldata)
GEN ldata_get_k1(GEN ldata)
GEN ldata_get_conductor(GEN ldata)
GEN ldata_get_rootno(GEN ldata)
GEN ldata_get_residue(GEN ldata)
long ldata_get_type(GEN ldata)
long linit_get_type(GEN linit)
GEN linit_get_ldata(GEN linit)
GEN linit_get_tech(GEN linit)
GEN lfun_get_domain(GEN tech)
GEN lfun_get_dom(GEN tech)
long lfun_get_bitprec(GEN tech)
GEN lfun_get_factgammavec(GEN tech)
GEN lfun_get_step(GEN tech)
GEN lfun_get_pol(GEN tech)
GEN lfun_get_Residue(GEN tech)
GEN lfun_get_k2(GEN tech)
GEN lfun_get_w2(GEN tech)
GEN lfun_get_expot(GEN tech)
long lfun_get_bitprec(GEN tech)
```

```

GEN lfunprod_get_fact(GEN tech)
GEN theta_get_an(GEN tdata)
GEN theta_get_K(GEN tdata)
GEN theta_get_R(GEN tdata)
long theta_get_bitprec(GEN tdata)
long theta_get_m(GEN tdata)
GEN theta_get_tdom(GEN tdata)
GEN theta_get_isqrtN(GEN tdata)

```

15.2 Conversions and constructors.

GEN lfunmisc_to_ldata(GEN obj) converts obj to Ldata format. Exception if obj cannot be converted.

GEN lfunmisc_to_ldata_shallow(GEN obj) as lfunmisc_to_ldata, shallow result. Exception if obj cannot be converted.

GEN lfunmisc_to_ldata_shallow_i(GEN obj) as lfunmisc_to_ldata_shallow, returning NULL on failure.

GEN lfunrtopoles(GEN r) given a vector r of “polar parts” $[\beta_i, P_i]$ where β_i is a complex number and P_i is a power series return the (β_i) , sorted with respect to `cmp_universal`.

```
int sdomain_isincl(double k, GEN dom, GEN dom0)
```

GEN ldata_vecan(GEN ldata, long N, long prec) return the vector of coefficients of indices 1 to N to precision `prec`. The output is allowed to be a `t_VECSMALL` when the coefficients are known to be all integral and fit into a `long`; for instance the Dirichlet L function of a real character or the L -function of a rational elliptic curve.

GEN ldata_newprec(GEN ldata, long prec) return a shallow copy of `ldata` with fields accurate to precision `prec`.

long etaquotype(GEN *peta, GEN *pN, GEN *pk, GEN *pCHI, long *pv, long *psd, long *pcusp) Let `eta` be the integer matrix factorization supposedly attached to an η -quotient $f(z) = \prod_i \eta(n_i z)^{e_i}$. Assuming `*peta` is initially set to `eta`, this function returns 0 if there is a type error or this does not define a function on some $X_0(N)$. Else it returns 1 and sets

- `*peta` to a normalized factorization (as would be returned by `factor`),
- `*pN` to the level N of f ,
- `*pk` to the modular weight k of f ,
- `*pCHI` to the Nebentypus of f (quadratic character) as an integer,
- `*pv` to the valuation at infinity $v_q(f)$,
- `*psd` to 1 if and only if f is self-dual,
- `*pcusp` to 1 if f is cuspidal, else to 0 if f holomorphic at all cusps, else to -1 .

The last three arguments `pCHI`, `pv` and `pcusp` can be set to `NULL`, in which case the relevant information is not computed, which saves time.

15.3 Variants of GP functions.

`GEN lfun(GEN ldata, GEN s, long bitprec)`
`GEN lfuninit(GEN ldata, GEN dom, long der, long bitprec)`
`GEN lfuninit_make(long t, GEN ldata, GEN tech, GEN domain)`
`GEN lfunlambda(GEN ldata, GEN s, long bitprec)`
`GEN lfunquadneg(long D, long k)` for $L(\chi_D, k)$, D fundamental discriminant and $k \geq 0$.
`long lfunthetacost(GEN ldata, GEN tdom, long m, long bit, long *extrabit):` `lfunthetacost0` when the first argument is known to be an `Ldata`. The `*extrabit` value is set to an upper estimate for the maximum over n of the exponent of the m -th derivative of $a_n K(nt/\sqrt{N})$, for t in the `tdom` domain. For an absolute error $2^{-\text{bit}}$, the values summed in θ should be computed at accuracy `bit + *extrabit`.
`GEN lfunthetacheckinit(GEN data, GEN tinf, long m, long bitprec)`
`GEN lfunrootno(GEN data, long bitprec)`
`GEN lfunabelianrelnit(GEN bnr, GEN subg, GEN dom, long der, long bitprec)` where `bnr` is a true *bnr* structure and `subg` is a congruence subgroup. Not GC-clean.
`GEN lfunzetakininit(GEN nf, GEN dom, long der, long bitprec)` where `nf` is a true *nf* structure. Not GC-clean.
`GEN lfunellmfpeters(GEN E, long bitprec)`

15.4 Inverse Mellin transforms of Gamma products.

`GEN gammamellinininv(GEN Vga, GEN s, long m, long bitprec)`
`GEN gammamellininvinit(GEN Vga, long m, long bitprec)`
`GEN gammamellininvrt(GEN K, GEN s, long bitprec)` not stack-clean, but suitable for `gc_upto`.
`int Vgaeasytheta(GEN Vga)` return 1 if the inverse Mellin transform is an exponential and 0 otherwise.
`double dbllemma526(double a, double b, double c, long B)`
`double dblcoro526(double a, double c, long B)`

Chapter 16: Modular symbols

`void checkms(GEN W)` raise an exception if W is not an *ms* structure from `msinit`.

`void checkmspadic(GEN W)` raise an exception if W is not an *mspadic* structure from `mspadicinit`.

`GEN mseval2_ooQ(GEN W, GEN phi, GEN c)` let W be a `msinit` structure for $k = 2$, ϕ be a modular symbol with integral values and c be a rational number. Return the integer $\phi(p)$, where p is the path $\{\infty, c\}$.

`void mspadic_parse_chi(GEN s, GEN *s1, GEN *s2)` see `mspadicL`; let χ be the cyclotomic character from $\text{Gal}(\mathbf{Q}_p(\mu_{p^\infty})/\mathbf{Q}_p)$ to $\mathbf{Z}_p^*$ and τ be the Teichmüller character for $p > 2$ and the character of order 2 on $(\mathbf{Z}/4\mathbf{Z})^*$ if $p = 2$. Let s encode the p -adic character $\chi^s := \langle \chi \rangle^{s_1} \tau^{s_2}$; set `*s1` and `*s2` to the integers s_1 and s_2 .

`GEN mspadic_unit_eigenvalue(GEN ap, long k, GEN p, long n)` let p be a prime not dividing the trace of Frobenius `ap`, return the unit root of $x^2 - ap * x + p^{(k-1)}$ to p -adic accuracy p^n .

Variants of `mfnumcusps` :

`ulong mfnumcuspsu(ulong n)`

`GEN mfnumcusps_fact(GEN fa)` where `fa` is `factor(n)`.

`ulong mfnumcuspsu_fact(GEN fa)` where `fa` is `factoru(n)`.

Chapter 17: Modular forms

17.1 Implementation of public data structures.

`void checkMF(GEN mf)` raise an exception if the argument is not a modular form space.

`GEN checkMF_i(GEN mf)` return the underlying modular form space if `mf` is either directly a modular form space from `mfinit` or a symbol from `mfsymbol`. Return `NULL` otherwise.

`int checkmf_i(GEN mf)` return 1 if the argument is a modular form and 0 otherwise.

`int checkfarey_i(GEN F)` return 1 if the argument is a Farey symbol (from `mspolygon` or `msfarey`) and 0 otherwise.

17.1.1 Accessors for modular form spaces.

Shallow functions; assume that their argument is a modular form space is created by `mfinit` and checked using `checkMF`.

`GEN MF_get_gN(GEN mf)` return the level N as a `t_INT`.

`long MF_get_N(GEN mf)` return the level N as a `long`.

`GEN MF_get_gk(GEN mf)` return the level k as a `t_INT`.

`long MF_get_k(GEN mf)` return the level k as a `long`.

`long MF_get_r(GEN mf)` assuming the level is a half-integer, return the integer $r = k - (1/2)$.

`GEN MF_get_CHI(GEN mf)` return the nebentypus χ , which is a special form of character structure attached to Dirichlet characters (see next section). Its values are given as algebraic numbers: either ± 1 or `t_POLMOD` in t .

`long MF_get_space(GEN mf)` returns the space type, corresponding to `mfinit`'s `space` flag. The current list is

`mf_NEW, mf_CUSP, mf_OLD, mf_EISEN, mf_FULL`

`GEN MF_get_basis(GEN mf)` return the $\mathbf{Q}$ -basis of the space, concatenation of `MF_get_E` and `MF_get_S`, in this order; the forms have coefficients in $\mathbf{Q}(\chi)$. Low-level version of `mfbasis`.

`long MF_get_dim(GEN mf)` returns the dimension d of the space. It is the cardinality of `MF_get_basis`.

`GEN MF_get_E(GEN mf)` returns a $\mathbf{Q}$ -basis for the subspace spanned by Eisenstein series in the space; the forms have coefficients in $\mathbf{Q}(\chi)$.

`GEN MF_get_S(GEN mf)` returns a $\mathbf{Q}$ -basis for the cuspidal subspace in the space; the forms have coefficients in $\mathbf{Q}(\chi)$.

GEN `MF_get_fields`(GEN `mf`) returns the vector of polynomials defining each Galois orbit of newforms over $\mathbf{Q}(\chi)$. Uses memoization: a first call splits the space and may be costly; subsequent calls return the cached result.

GEN `MF_get_newforms`(GEN `mf`) returns a vector `vF` containing the coordinates of the eigenforms on `MF_get_basis` (`mftobasis` form). Low-level version of `mfeigenbasis`, whose elements are recovered as `mflinear`(`mf`, `gel(vF,i)`). Uses memoization, sharing the same data as `MF_get_fields`. Note that it is much more efficient to use `mfcoefs`(`mf`,) then multiply by this vector than to compute the coefficients of eigenforms from `mfeigenbasis` individually.

The following accessors are technical,

GEN `MF_get_M`(GEN `mf`) the $(1+m) \times d$ matrix whose j -th column contain the coefficients of the j -th entry in `MF_get_basis`, m is the optimal “Sturm bound” for the space: the maximum of the $v_\infty(f)$ over nonzero forms. It has entries in $\mathbf{Q}(\chi)$.

GEN `MF_get_Mindex`(GEN `mf`) is a `t_VECSMALL` containing d row indices, the corresponding rows of M form an invertible matrix M_0 .

GEN `MF_get_Minv`(GEN `mf`) the inverse of M_0 in a form suitable for fast multiplication.

GEN `MFcusp_get_vMjd`(GEN `mf`) valid only for a full *cuspidal* space. Then the functions in `MF_get_S` are of the form $B_d T_j Tr_M^{new}$. This returns the vector of triples (`t_VECSMALL`) $[M, j, d]$, in the same order.

GEN `MFnew_get_vj`(GEN `mf`) valid only for a *new* space. Then the functions in `MF_get_S` are of the form $T_j Tr_N^{new}$. This returns a `t_VECSMALL` of the Hecke indices j , in the same order.

17.1.2 Accessors for individual modular forms.

GEN `mf_get_gN`(GEN `F`) return the level of F , which may be a multiple of the conductor, as a `t_INT`

`long mf_get_N`(GEN `F`) return the level as a `long`.

GEN `mf_get_gk`(GEN `F`) return the weight of F as a `t_INT` or a `t_FRAC` with denominator 2 (half-integral weight).

`long mf_get_k`(GEN `F`) return the weight as a `long`; if the weight is not integral, this raises an exception.

`long mf_get_r`(GEN `F`) assuming F is a modular form of half-integral weight $k = (2r+1)/2$, return $r = k - (1/2)$.

GEN `mf_get_CHI`(GEN `F`) return the nebentypus, which is a special form of character structure attached to Dirichlet characters (see next section). Its values are given as algebraic numbers: either ± 1 or `t_POLMOD` in t .

GEN `mf_get_field`(GEN `F`) return the polynomial (in variable y) defining $\mathbf{Q}(f)$ over $\mathbf{Q}(\chi)$.

GEN `mf_get_NK`(GEN `F`) return the tag attached to F : a vector containing `gN`, `gk`, `CHI`, `field`. Never use its component directly, use individual accessors as above.

`long mf_get_type`(GEN `F`) returns a symbolic name for the constructor used to create the form, e.g. `t_MF_EISEN` for a general Eisenstein series. A form has a recursive structure represented by a tree: its definition may involve other forms, e.g. the tree attached to $T_n f$ contains f as a subtree. Such trees have *leaves*, forms which do not contain a strict subtree, e.g. `t_MF_DELTA` is a leaf, attached to Ramanujan’s Δ .

Here is the current list of types; since the names are liable to change, they are not documented at this point. Use `mfdescribe` to visualize their mathematical structure.

```
/*leaves*/
  t_MF_CONST, t_MF_EISEN, t_MF_Ek, t_MF_DELTA, t_MF_ETAQUO, t_MF_ELL,
  t_MF_DIHEDRAL, t_MF_THETA, t_MF_TRACE, t_MF_NEWTRACE,
/*recursive*/
  t_MF_MUL, t_MF_POW, t_MF_DIV, t_MF_BRACKET, t_MF_LINEAR, t_MF_LINEAR_BHN,
  t_MF_SHIFT, t_MF_DERIV, t_MF_DERIVE2, t_MF_TWIST, t_MF_HECKE,
  t_MF_BD,
```

17.1.3 Nebentypus. The characters stored in modular forms and modular form spaces have a special structure. One can recover the parameters of an ordinary Dirichlet character by `G = gel(CHI,1)` (the underlying `znstar`) and `chi = gel(CHI,2)` (the underlying character in `znconreylog` form).

`long mfcharmodulus(GEN CHI)` the modulus of χ .

`long mfcharorder(GEN CHI)` the order of χ .

`GEN mfcharpol(GEN CHI)` the cyclotomic polynomial Φ_n defining $\mathbf{Q}(\chi)$, always normalized so that n is not 2 mod 4.

17.1.4 Miscellaneous functions.

`long mfnewdim(long N, long k, GEN CHI)` dimension of the new part of the cuspidal space.

`long mfcuspdim(long N, long k, GEN CHI)` dimension of the cuspidal space.

`long mfoldddim(long N, long k, GEN CHI)` dimension of the old part of the cuspidal space.

`long mfeisensteindim(long N, long k, GEN CHI)` dimension of the Eisenstein subspace.

`long mffulldim(long N, long k, GEN CHI)` dimension of the full space.

`GEN mfeisensteinspaceinit(GEN NK)`

`GEN mfdiv_val(GEN F, GEN G, long vG)`

`GEN mfembed(GEN E, GEN v)`

`GEN mfmatembed(GEN E, GEN v)`

`GEN mfvecembed(GEN E, GEN v)`

`long mfsturmNgk(long N, GEN k)`

`long mfsturmNk(long N, long k)`

`long mfsturm_mf(GEN mf)`

`long mfiscuspidal(GEN mf, GEN F)`

`GEN mftobasisES(GEN mf, GEN F)`

`GEN mftocol(GEN F, long lim, long d)`

`GEN mfvectomat(GEN vF, long lim, long d)`

Chapter 18:

Plots

A `PARI_plot` canvas is a record of dimensions, with the following fields:

```
long width; /* window width */
long height; /* window height */
long hunit; /* length of horizontal 'ticks' */
long vunit; /* length of vertical 'ticks' */
long fwidth; /* font width */
long fheight; /* font height */
void (*draw)(PARI_plot *T, GEN w, GEN x, GEN y);
```

The `draw` method performs the actual drawing of a `t_VECSMALL` `w` (rectwindow indices); x and y are `t_VECSMALL`s of the same length and rectwindow $w[i]$ is drawn with its upper left corner at offset $(x[i], y[i])$. No plot engine is available in `libpari` by default, since this would introduce a dependency on extra graphical libraries. See the files `src/graph/plot*` for basic implementations of various plot engines: `plotsvg` is particularly simple (`draw` is a 1-liner).

`void pari_set_plot_engine(void (*T)(PARI_plot *))` installs the graphical engine T and initializes the graphical subsystem. No routine in this chapter will work without this initialization.

`void pari_kill_plot_engine(void)` closes the graphical subsystem and frees the resources it occupies.

18.1 Highlevel functions.

Those functions plot $f(E, x)$ for $x \in [a, b]$, using n regularly spaced points (by default).

`GEN ploth(void *E, GEN(*f)(void*, GEN), GEN a, GEN b, long flags, long n, long prec)`
draw physically.

`GEN plotrecth(void *E, GEN(*f)(void*, GEN), long w, GEN a, GEN b, long flags, long n, long prec)` draw in rectwindow w .

18.2 Function.

```
void plotbox(long ne, GEN gx2, GEN gy2)
void plotclip(long rect)
void plotcolor(long ne, long color)
void plotcopy(long source, long dest, GEN xoff, GEN yoff, long flag)
GEN plotcursor(long ne)
void plotdraw(GEN list, long flag)
GEN plothrow(GEN listx, GEN listy, long flag)
GEN plotsizes(long flag)
void plotinit(long ne, GEN x, GEN y, long flag)
void plotkill(long ne)
void plotline(long ne, GEN x2, GEN y2)
void plotlines(long ne, GEN listx, GEN listy, long flag)
void plotlinetype(long ne, long t)
void plotmove(long ne, GEN x, GEN y)
void plotpoints(long ne, GEN listx, GEN listy)
void plotpointsize(long ne, GEN size)
void plotpointtype(long ne, long t)
void plotrbox(long ne, GEN x2, GEN y2)
GEN plotrecthrow(long ne, GEN data, long flags)
void plotrline(long ne, GEN x2, GEN y2)
void plotrmove(long ne, GEN x, GEN y)
void plotrpoint(long ne, GEN x, GEN y)
void plotscale(long ne, GEN x1, GEN x2, GEN y1, GEN y2)
void plotstring(long ne, char*x, long dir)
```

18.2.1 Obsolete functions. These draw directly to a PostScript file specified by a global variable and should no longer be used. Use `plotexport` and friends instead.

```
void psdraw(GEN list, long flag)
GEN psplothrow(GEN listx, GEN listy, long flag)
GEN psplotth(void *E, GEN(*f)(void*, GEN), GEN a, GEN b, long flags, long n, long
prec) draw to a PostScript file.
```

18.3 Dump rectwindows to a PostScript or SVG file.

w, x, y are three `t_VECSMALLs` indicating the rectwindows to dump, at which offsets. If T is `NULL`, rescale with respect to the installed graphic engine dimensions; else with respect to T .

```
char* rect2ps(GEN w, GEN x, GEN y, PARI_plot *T)
```

`char* rect2ps_i(GEN w, GEN x, GEN y, PARI_plot *T, int plotps)` if `plotps` is 0, as above; else private version used to implement the `plotps` graphic engine (do not rescale, rotate to portrait orientation).

```
char* rect2svg(GEN w, GEN x, GEN y, PARI_plot *T)
```

18.4 Technical functions exported for convenience.

`void pari_plot_by_file(const char *env, const char *suf, const char *img)` backend used by the `plotps` and `plotsvg` graphic engines.

`void colorname_to_rgb(const char *s, int *r, int *g, int *b)` convert an X11 colorname to RGB values.

`void color_to_rgb(GEN c, int *r, int *g, int *b)` convert a pari color (`t_VECSMALL` RGB triple or `t_STR` name) to RGB values.

`void long_to_rgb(long c, int *r, int *g, int *b)` split a standard hexadecimal color value `0xfdf5e6` to its rgb components (`0xfd`, `0xf5`, `0xe6`).

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.

```
CC = cc
INCDIR = /usr/include/x86_64-linux-gnu
LIBDIR = /usr/lib/x86_64-linux-gnu
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&&", "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 = diviixact( subii(a, mulii(A,ux)), B );
    return gc_all(av, 3, &a, U, V);
}

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:

```
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:

```
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_valloc(struct pari_thread *pth, size_t s, size_t v, GEN arg)` Allocate a PARI stack of size `s` which can grow to at most `v` (as with `parisize` and `parisizemax`) and associate it, together with the argument `arg`, with the PARI thread data `pth`.

`void pari_thread_alloc(struct pari_thread *pth, size_t s, GEN arg)` As above but the stack cannot grow beyond `s`.

`void pari_thread_free(struct pari_thread *pth)` Free the PARI stack attached to 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`.

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 reenale it);
- cached real constants (returned by `mppi`, `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.)

```
#include <pari/pari.h> /* Include PARI headers */
#include <pthread.h>    /* Include POSIX threads headers */
#ifdef ENABLE_TLS
#error "this example requires Configure --enable-tls or --mt=ptread"
#endif
void *
mydet(void *arg)
{
    GEN F, M;
    /* Set up thread stack and get thread parameter */
    M = pari_thread_start((struct pari_thread*) arg);
    F = QM_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)
{
```

```

long prec = DEFAULTPREC;
GEN M1,M2, N1,N2, F1,F2, D1,D2;
pthread_t th1, th2, th3, th4; /* POSIX-thread variables */
struct pari_thread pth1, pth2, pth3, pth4; /* pari thread variables */

/* Initialise the main PARI stack and global objects (gen_0, etc.) */
pari_init(32000000,500000);
/* Compute in the main PARI stack */
N1 = addis(int2n(256), 1); /*  $2^{256} + 1$  */
N2 = subis(int2n(193), 1); /*  $2^{193} - 1$  */
M1 = mathilbert(149);
M2 = mathilbert(150);
/* Allocate pari thread structures */
pari_thread_alloc(&pth1,8000000,N1);
pari_thread_alloc(&pth2,8000000,N2);
pari_thread_alloc(&pth3,32000000,M1);
pari_thread_alloc(&pth4,32000000,M2);
/* 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);
pthread_create(&th4,NULL, &mydet, (void*)&pth4); /* Start 4 threads */
pthread_join(th1,(void*)&F1);
pthread_join(th2,(void*)&F2);
pthread_join(th3,(void*)&D1);
pthread_join(th4,(void*)&D2); /* Wait for termination, get the results */
pari_printf("F1=%Ps\nF2=%Ps\nlog(D1)=%Ps\nlog(D2)=%Ps\n",
           F1,F2, glog(D1,prec),glog(D2,prec));
pari_thread_free(&pth1);
pari_thread_free(&pth2);
pari_thread_free(&pth3);
pari_thread_free(&pth4); /* clean up */
return 0;
}

```

Index

SomeWord refers to PARI-GP concepts.
SomeWord is a PARI-GP keyword.
SomeWord is a generic index entry.

A

ABC_to_bnr	329	addr	15
abelian_group	264	addr_sign	97
abgrp_get_cyc	300	addsi_sign	97
abgrp_get_gen	300	addui	96
abgrp_get_no	300	addui_sign	97
abmap_char_image	329	addumului	96
abmap_kernel	329	adduu	96
abmap_nchar_image	329	affc_fixlg	261
abmap_subgroup_image	329	affects_sign	63
abscmpii	94	affects_sign_safe	63
abscmpiu	94	affgc	261
abscmpr	94	affgr	87
abscmpui	94	affii	87
absdiviu_rem	99	affir	87
absequalii	94	affiz	87
absequaliu	94	affrr	87
absequalui	94	affrr_fixlg	87
absfrac	249	affsi	87
absfrac_shallow	249	affsr	87
absi	92	affsz	87
absi_shallow	93	affui	87
absr	92	affur	87
absrnz_equal1	94	alarm	277
absrnz_equal2n	94	algin	349
abstorel	330	alglat_get_primbasis	350
absZ_factor	175	alglat_get_scalar	350
absZ_factor_limit	176	algmodpr	350
absZ_factor_limit_strict	176	algmodprinit	350
addhelp	78	algmodpr_get_ff	350
addii	15	algmodpr_get_k	350
addii_sign	97	algmodpr_get_lift	350
addir	15	algmodpr_get_m	350
addir_sign	97	algmodpr_get_p	351
addis	15	algmodpr_get_pr	350
addiu	96	algmodpr_get_proj	350
addll	81	algmodpr_get_T	351
addllx	81	algmodpr_get_tau	351
addmul	81	algnewprec	351
addmulii	96	algnewprec_shallow	351
addmulii_inplace	96	algsimpledec_ss	351
addmuliu	96	algtype	349
addmuliu_inplace	97	alg_complete	348
addri	15	alg_csa_table	348
		alg_cyclic	348
		alg_get_absdim	349
		alg_get_abssplitting	350
		alg_get_aut	349
		alg_get_auts	350
		alg_get_b	350

alg_get_basis	350	bid_get_sarch	305
alg_get_center	350	bid_get_sprk	305
alg_get_char	350	bid_get_U	305
alg_get_degree	349	BIGDEFAULTPREC	16, 64
alg_get_dim	349	bigomegau	106
alg_get_hasse_f	350	BIL	51
alg_get_hasse_i	350	binary quadratic form	31
alg_get_invbasis	350	binary_2k	90, 91
alg_get_invol	350	binary_2k_nv	90
alg_get_multable	350	binary_zv	90
alg_get_relmultable	350	bincopy_relink	68
alg_get_splitpol	350	binomial	249
alg_get_splittingbasis	350	binomialuu	249
alg_get_splittingbasisinv	350	bin_copy	68
alg_get_splittingdata	350	bitprecision0	224
alg_get_splittingfield	350	BITS_IN_HALFULONG	64
alg_get_tracebasis	350	BITS_IN_LONG	51, 64, 90, 91
alg_hasse	348	bits_to_int	90
alg_hilbert	348	bits_to_u	90
alg_matrix	348	bit_accuracy	59
alg_model	349	bit_accuracy_mul	59
alg_type	349	bit_prec	59
atanhui	260	bl_base	73
atanhuu	259	bl_next	73
avma	17	bl_num	73
		bl_prev	73
		bl_refc	73

B

bb_algebra	219	bnfgwgeneric	330
bb_field	218	bnfisprincipal0	307, 323, 326
bb_group	215	bnfisunit	313
bb_ring	220	bnfnewprec	307, 324
bernfrac	262, 263	bnfnewprec_shallow	307
Bernoulli	262, 263	bnfttestprimes	324
bernreal	262, 263	bnf_build_cheapfu	306
bezout	101	bnf_build_cycgen	306
bfffo	81	bnf_build_matalpha	306
bid_get_arch	304	bnf_build_units	306
bid_get_archp	305	bnf_compactfu	303
bid_get_cyc	305	bnf_compactfu_mat	303
bid_get_fact	305	bnf_get_clgp	302
bid_get_fact2	305	bnf_get_cyc	302
bid_get_gen	305	bnf_get_fu	302
bid_get_gen_nocheck	305	bnf_get_fu_nocheck	302
bid_get_grp	304	bnf_get_gen	302
bid_get_ideal	304, 305	bnf_get_logfu	302
bid_get_mod	304	bnf_get_nf	302
bid_get_MOD	304	bnf_get_no	302
bid_get_no	305	bnf_get_reg	302
		bnf_get_sunits	302

bnf_get_tuN	302	cb_pari_break_loop	55
bnf_get_tuU	302	cb_pari_display_hist	55
bnf_has_fu	303	cb_pari_err_handle	55, 279
bnrclassno	327	cb_pari_err_recover	56
bnrconductor	328	cb_pari_handle_exception	55
bnrconductorofchar	333	cb_pari_init_histfile	55
bnrconductor_factored	328	cb_pari_is_interactive	55, 57
bnrconductor_factored_arch	328	cb_pari_long_help	56
bnrconductor_raw	328	cb_pari_pre_recover	55
bnrdisc	329	cb_pari_quit	55
bnrdisclist0	333	cb_pari_sigint	55
bnrinit0	332	cb_pari_start_output	55
bnrisconductor	329	cb_pari_whatnow	56
bnrisprincipal	333	ceildivuu	99
bnrnewprec	307	ceilr	89
bnrnewprec_shallow	307	ceil_safe	89
bnrsurjection	329	centerlift	225
bnrtoprimitive	328	centerlift0	225
bnr_char_sanitize	328	centermod	240
bnr_get_bid	303	centermodii	97
bnr_get_bnf	303	centermod_i	240
bnr_get_clgp	303	cgcd	101
bnr_get_cyc	303, 329	cgetalloc	67
bnr_get_gen	303	cgetc	23, 58, 67, 86, 261
bnr_get_gen_nocheck	303	cgetg	22, 24, 58, 67
bnr_get_mod	303	cgetg_block	73
bnr_get_nf	303	cgetg_copy	58
bnr_get_no	303	cgeti	23, 58, 67, 86
bnr_sanitize	328	cgetineg	86
bnr_subgroup_check	328	cgetipos	86
bnr_subgroup_sanitize	328	cgetr	23, 58, 67, 86
bnr_vecchar_sanitize	328	cgetr_block	73
both_odd	82	cgiv	17, 68
boundfact	176	character string	32
BPSW_isprime	181	<i>character</i>	346
BPSW_psp	181	characteristic	251
brent_kung_optpow	219	charconj	346
brute	272	chardiv	347
buchimag	332	charmulo	346
Buchray	327	chartoGENstr	269
buchreal	332	char_check	347
C		char_denormalize	346
CATCH_ALL	46	char_normalize	346
cbezout	102	char_renormalize	346
cbrtr	258	char_simplify	346
cbrtr_abs	259	checkabgrp	300
cb_pari_ask_confirm	55, 56	checkalg	349
		checkbid	299
		checkbid_i	299

checkbnf	299	clcm	101
checkbnf_i	299	cleanroots	204, 252
checkbnr	299	clean_Z_factor	179
checkbnr_i	299	clone	72
checkell	353	clone	14, 25
checkell5	353	CLONEBIT	65
checkellisog	354	closemodinvertible	346
checkellpt	353	closure	76
checkellpt_i	353	closure	32
checkell_Fq	354	closuretoinl	292
checkell_i	353	closure_arity	32
checkell_Q	354	closure_callgen0	289
checkell_Qp	354	closure_callgen0prec	289
checkfarey_i	375	closure_callgen1	76, 289
checkgal	299	closure_callgen1prec	289
checkgroup	264	closure_callgen2	289
checkgroupelts	264	closure_callgenall	289
checkhasse	349	closure_callgenvec	289
checklat	349	closure_callgenvecdef	289
checkMF	375	closure_callgenvecdefprec	289
checkMF_i	375	closure_callgenvecprec	289
checkmf_i	375	closure_callvoid1	289
checkmodpr	300	closure_context	292
checkms	373	closure_deriv	291
checkmspadic	373	closure_derivn	291
checknf	299	closure_disassemble	289
checknfelt_mod	300	closure_err	292
checknf_i	299	closure_evalbrk	290
checkprid	300	closure_evalgen	76, 289
checkprid_i	300	closure_evalnobrk	289
checkrnf	299	closure_evalres	290
checkrnf_i	299	closure_evalvoid	76, 290
checksqmat	299	closure_func_err	55
checkznstar_i	299	closure_is_variadic	32
check_arith_all	179	cmpii	93
check_arith_non0	179	cmpir	93
check_arith_pos	179	cmpis	93
check_ecppcert	181	cmpiu	93
check_modinv	367	cmpri	94
check_quaddisc	334	cmprr	94
check_quaddisc_imag	334	cmprrs	94
check_quaddisc_real	334	cmpsi	93
check_ZKmodule	300	cmpsr	93
check_ZKmodule_i	300	cmpss	93
chinese1	162	cmpui	93
chinese1_coprime_Z	162	cmpuu	93
chk_gc_upto	72	cmp_Flx	239
classno	334	cmp_nodata	238
classno2	334	cmp_padic	239

elldatagridgenerators	355	elltors	356
elleulerf	356	elltors0	356
ellff_get_a4a6	357	elltors_psylo	356
ellff_get_card	357	elltrace_extension	359
ellff_get_D	357	ell_get_a1	354
ellff_get_field	357	ell_get_a2	354
ellff_get_gens	357	ell_get_a3	354
ellff_get_group	357	ell_get_a4	354
ellff_get_m	357	ell_get_a6	354
ellff_get_o	357	ell_get_b2	354
ellff_get_p	356	ell_get_b4	354
ellheightoo	355	ell_get_b6	354
ellinf	358	ell_get_b8	354
ellinit	353	ell_get_c4	354
ellintegralmodel	355	ell_get_c6	354
ellintegralmodel_i	355	ell_get_disc	354
ellisogeny	354	ell_get_j	354
ellnf_get_bnf	356	ell_get_type	353, 359
ellnf_get_nf	356	ell_is_inf	358
ellnf_vecarea	356	ell_is_integral	354
ellnf_veceta	356	eltabstorel	330
ellnf_vecomega	356	eltabstorel_lift	330
ellorder	355	eltreltoabs	330
ellorder_Q	355	embednorm_T2	325
ellQp_ab	356	embed_disc	325
ellQp_AGM	356	embed_norm	325
ellQp_Ei	262	embed_roots	325
ellQp_get_p	356	embed_T2	325
ellQp_get_prec	356	equali1	235
ellQp_get_zero	356	equali1	94
ellQp_L	356	equalim1	236
ellQp_q	356	equalis	94
ellQp_root	356	equaliu	94
ellQp_Tate_uniformization	356	equalrr	94
ellQp_u	356	equalsi	94
ellQp_u2	356	equalui	94
ellQtwist_bsdperiod	355	error context	32
ellQ_get_CM	354, 361	error	37
ellQ_get_N	354	error	277
ellQ_get_Nfa	354	err_flush	75
ellrootno_global	355	err_get_compo	46
ellR_ab	357	err_get_num	46
ellR_area	357	err_printf	75
ellR_eta	357	etaquote	370
ellR_get_prec	357	eta_product_ZXn	202
ellR_get_sign	358	eta_ZXn	202
ellR_omega	357	eulerf_gchar	348
ellR_roots	357	eulerphiu	106
ellsupersingularj_FpXQ	365	eulerphiu_fact	106

F2xqM_F2xqC_invimage	157	F2xqX_sqr	159
F2xqM_F2xqC_mul	157	F2xq_Artin_Schreier	156
F2xqM_gauss	157	F2xq_autpow	156
F2xqM_image	157	F2xq_conjvec	156
F2xqM_indexrank	157	F2xq_div	156
F2xqM_inv	157	F2xq_ellcard	363
F2xqM_invimage	157	F2xq_ellgens	363
F2xqM_ker	157	F2xq_ellgroup	363
F2xqM_mul	157	F2xq_elltwist	363
F2xqM_rank	157	F2xq_inv	156
F2xqM_suppl	157	F2xq_invsafe	156
F2xqV_roots_to_pol	156	F2xq_log	156
F2xqXQV_red	160	F2xq_matrix_pow	156
F2xqXQ_autpow	160	F2xq_mul	156
F2xqXQ_auttrace	160	F2xq_order	156
F2xqXQ_inv	159	F2xq_pow	156
F2xqXQ_invsafe	159	F2xq_powers	156
F2xqXQ_mul	160	F2xq_powu	156
F2xqXQ_pow	160	F2xq_pow_init	156
F2xqXQ_powers	160	F2xq_pow_table	156
F2xqXQ_sqr	160	F2xq_sqr	156
F2xqXV_prod	159	F2xq_sqrt	156
F2xqX_ddf	159	F2xq_sqrtn	156
F2xqX_degfact	159	F2xq_sqrt_fast	156
F2xqX_disc	159	F2xq_trace	156
F2xqX_div	159	F2xV_to_F2m	174
F2xqX_divrem	159	F2xV_to_FlxV_inplace	172
F2xqX_extgcd	159	F2xV_to_ZXV_inplace	172
F2xqX_F2xqXQV_eval	159	F2xXC_to_ZXXC	158
F2xqX_F2xqXQ_eval	159	F2xXV_to_F2xM	157
F2xqX_F2xq_mul	159	F2xX_add	157
F2xqX_F2xq_mul_to_monic	159	F2xX_deriv	158
F2xqX_factor	159	F2xX_F2x_add	158
F2xqX_factor_squarefree	159	F2xX_F2x_mul	158
F2xqX_Frobenius	159	F2xX_renormalize	157
F2xqX_gcd	159	F2xX_shift	157
F2xqX_get_red	158	F2xX_to_F2xC	157
F2xqX_halfgcd	159	F2xX_to_FlxX	157
F2xqX_halfgcd_all	159	F2xX_to_Kronecker	158
F2xqX_invBarrett	159	F2xX_to_ZXX	157
F2xqX_isplayer	159	F2xY_degrees	157
F2xqX_mul	159	F2xY_F2xqV_evalx	158
F2xqX_nbroots	159	F2xY_F2xq_evalx	158
F2xqX_normalize	159	F2x_1_add	155
F2xqX_powu	159	F2x_add	155
F2xqX_red	158	F2x_clear	154
F2xqX_rem	159	F2x_coeff	154
F2xqX_resultant	159	F2x_copy	154
F2xqX_roots	159	F2x_ddf	156

F2x_deflate	155	F3m_to_ZM	122
F2x_degfact	155	F3m_transpose	121
F2x_degree	155	F3v_clear	121
F2x_deriv	155	F3v_coeff	121
F2x_div	155	F3v_dotproduct	121
F2x_divrem	155	F3v_dotsquare	121
F2x_equal	155	F3v_set	121
F2x_equal1	155	F3v_to_Flv	121
F2x_eval	155	factmod	161
F2x_even_odd	155	factor	353
F2x_extgcd	155	factorback	245
F2x_F2xqV_eval	156	factoredpolred	333
F2x_F2xq_eval	156	factoredpolred2	333
F2x_factor	156	factorial_Fl	83
F2x_factor_squarefree	156	factorial_Fp	105
F2x_flip	154	factorial_lval	92
F2x_Frobenius	155	factorint	178
F2x_gcd	155	factoru	177
F2x_get_red	154	factoru_pow	177
F2x_halfgcd	155	factor_Aurifeuille	177
F2x_issquare	155	factor_Aurifeuille_prime	176
F2x_is_irred	155	factor_pn_1	176
F2x_matFrobenius	155	factor_pn_1_limit	176
F2x_mul	155	factor_proven	180
F2x_recip	155	famat	311
F2x_rem	155	famat_small_reduce	313
F2x_renormalize	155	famatV_factorback	312
F2x_set	154	famatV_zv_factorback	312
F2x_shift	155	famat_div	312
F2x_sqr	155	famat_div_shallow	312
F2x_sqrt	155	famat_idealfactor	313
F2x_Teichmuller	156	famat_inv	312
F2x_to_F2v	174	famat_inv_shallow	312
F2x_to_F2xX	154	famat_makecoprime	327
F2x_to_Flx	154	famat_mul	312
F2x_to_ZX	154	famat_mulpow	312
F2x_valrem	155	famat_mulpows_shallow	312
F3c_to_mod	160	famat_mulpow_shallow	312
F3c_to_ZC	121	famat_mul_shallow	312
F3m_coeff	121	famat_nfvalrem	313
F3m_copy	121	famat_pow	312
F3m_ker	121	famat_pows_shallow	312
F3m_ker_sp	121	famat_pow_shallow	312
F3m_mul	121	famat_reduce	311, 313
F3m_rank	121	famat_remove_trivial	313
F3m_row	121	famat_sqr	312
F3m_set	121	famat_to_nf	313
F3m_to_Flm	122	famat_to_nf_moddivisor	326
F3m_to_mod	160	famat_to_nf_modideal_coprime	327

famat_Z_gcd	312	FF_ellcard_SEA	359
fetch_user_var	33, 74	FF_ellgens	359
fetch_var	34, 74	FF_ellgroup	359
fetch_var_higher	34	FF_elllog	359
fetch_var_value	34, 74	FF_ellmul	359
FFM_deplin	257	FF_ellorder	359
FFM_det	257	FF_ellrandom	359
FFM_FFC_gauss	257	FF_elltatepairing	359
FFM_FFC_invimage	257	FF_elltwist	359
FFM_FFC_mul	257	FF_ellweilpairing	359
FFM_gauss	257	FF_equal	254
FFM_image	257	FF_equal0	254
FFM_indexrank	257	FF_equal1	254
FFM_inv	257	FF_equalm1	254
FFM_invimage	257	FF_f	253
FFM_ker	257	FF_Frobenius	254
FFM_mul	257	FF_gen	253
FFM_rank	257	FF_inv	254
FFM_suppl	257	FF_isplayer	255
FFV_roots_to_pol	257	FF_issquare	255
FFXQ_inv	258	FF_issquareall	255
FFXQ_minpoly	258	FF_log	255
FFXQ_mul	258	FF_map	255
FFXQ_sqr	258	FF_minpoly	255
FFX_add	255	FF_mod	253
FFX_ddf	256	FF_mul	254
FFX_degfact	256	FF_mul2n	254
FFX_disc	256	FF_neg	254
FFX_extgcd	256	FF_neg_i	254
FFX_factor	256	FF_norm	255
FFX_factor_squarefree	256	FF_order	255
FFX_gcd	256	FF_p	253
FFX_halfgcd	256	ff_parse_Tp	128
FFX_halfgcd_all	256	FF_pow	254
FFX_isplayer	256	FF_primroot	255
FFX_mul	255	FF_p_i	253
FFX_nbroots	256	FF_q	253
FFX_preimage	257	FF_Q_add	254
FFX_preimagerel	257	FF_samefield	254
FFX_rem	256	FF_sqr	254
FFX_resultant	256	FF_sqrt	255
FFX_roots	256	FF_sqrtn	255
FFX_sqr	255	FF_sub	254
FF_1	253	FF_to_F2xq	253
FF_add	254	FF_to_F2xq_i	253
FF_charpoly	255	FF_to_Flxq	253
FF_conjvec	255	FF_to_Flxq_i	253
FF_div	254	FF_to_FpXQ	253
FF_ellcard	359	FF_to_FpXQ_i	253

FF_trace	255	Flm_Fl_mul	117
FF_var	253	Flm_Fl_mul_inplace	117
FF_zero	254	Flm_Fl_mul_pre	117
FF_Z_add	254	Flm_Fl_sub	116
FF_Z_mul	254	Flm_gauss	118
FF_Z_Z_muldiv	255	Flm_hess	119
file_is_binary	272	Flm_hess_pre	119
finite field element	29	Flm_image	119
fixlg	71, 87	Flm_indexrank	118
Flc_Flv_mul	116	Flm_intersect	119
Flc_FpV_mul	117	Flm_intersect_i	119
Flc_lincomb1_inplace	117	Flm_inv	118
Flc_to_mod	160	Flm_invimage	118
Flc_to_ZC	172	Flm_ker	118
Flc_to_ZC_inplace	172	Flm_ker_sp	118
Fle_add	361	Flm_mul	118
Fle_changepoint	362	Flm_mul_pre	118
Fle_changepointinv	362	Flm_neg	117
Fle_dbl	361	Flm_powers	118
Fle_log	362	Flm_powu	118
Fle_mul	362	Flm_rank	118
Fle_mulu	362	Flm_row	118
Fle_order	362	Flm_sqr	118
Fle_sub	361	Flm_sub	118
Fle_tatepairing	362	Flm_suppl	118
Fle_to_Flj	362	Flm_to_F2m	120
Fle_weilpairing	362	Flm_to_F3m	122
FljV_factorback_pre	363	Flm_to_FlxV	173
Flj_add_pre	362	Flm_to_FlxX	174
Flj_changepointinv_pre	362	Flm_to_mod	160
Flj_dbl_pre	362	Flm_to_ZM	173
Flj_mulu_pre	362	Flm_to_ZM_inplace	173
Flj_neg	362	Flm_trace	118
Flj_to_Fle	362	Flm_transpose	119
Flj_to_Fle_pre	362	floorr	89
Flm_add	118	floor_safe	89
Flm_adjoint	118	flush	271
Flm_center	116	Flv_add	117
Flm_charpoly	118	Flv_add_inplace	117, 320
Flm_copy	116	Flv_center	116
Flm_deplin	118	Flv_copy	116
Flm_det	118	Flv_dotproduct	117
Flm_det_sp	118	Flv_dotproduct_pre	117
Flm_Flc_gauss	118	Flv_factorback	117
Flm_Flc_invimage	118	Flv_Flm_polint	144
Flm_Flc_mul	116	Flv_Fl_div	117
Flm_Flc_mul_pre	116	Flv_Fl_div_inplace	117
Flm_Flc_mul_pre_Flx	116	Flv_Fl_mul	117
Flm_Fl_add	116	Flv_Fl_mul_inplace	117

Flv_Fl_mul_part_inplace	117	FlxqE_add	364
Flv_inv	117	FlxqE_changepoint	364
Flv_invVandermonde	144	FlxqE_changepointinv	364
Flv_inv_inplace	117	FlxqE_dbl	364
Flv_inv_pre	117	FlxqE_log	365
Flv_inv_pre_inplace	117	FlxqE_mul	364
Flv_neg	117	FlxqE_neg	364
Flv_neg_inplace	117	FlxqE_order	364
Flv_polint	144	FlxqE_sub	364
Flv_prod	117	FlxqE_tatepairing	365
Flv_prod_pre	117	FlxqE_weilpairing	365
Flv_roots_to_pol	144	FlxqE_weilpairing_pre	365
Flv_sub	117	FlxqM_deplin	122
Flv_sub_inplace	117	FlxqM_det	122
Flv_sum	117	FlxqM_FlxqC_gauss	122
Flv_to_F2v	120	FlxqM_FlxqC_invimage	122
Flv_to_F3v	122	FlxqM_FlxqC_mul	122
Flv_to_Flx	173	FlxqM_Flxq_mul	122
Flv_to_ZV	173	FlxqM_gauss	122
FlxC_eval_powers_pre	145	FlxqM_image	122
FlxC_FlxqV_eval	147	FlxqM_indexrank	122
FlxC_FlxqV_eval_pre	147	FlxqM_inv	122
FlxC_Flxq_eval	147	FlxqM_invimage	122
FlxC_Flxq_eval_pre	147	FlxqM_ker	122
FlxC_Fl_translate	145	FlxqM_mul	122
FlxC_neg	145	FlxqM_rank	122
FlxC_sub	145	FlxqM_red_pre	122
FlxC_to_F2xC	174	FlxqM_suppl	122
FlxC_to_ZXC	172	FlxqV_dotproduct	122
FlxM_eval_powers_pre	145	FlxqV_dotproduct_pre	122
FlxM_Flx_add_shallow	122	FlxqV_factorback	146
FlxM_Fl_translate	145	FlxqV_roots_to_pol	147
FlxM_mul	145	FlxqXC_FlxqXQV_eval	152
FlxM_neg	145	FlxqXC_FlxqXQV_eval_pre	152
FlxM_sub	145	FlxqXC_FlxqXQ_eval	152
FlxM_to_FlxXV	174	FlxqXC_FlxqXQ_eval_pre	152
FlxM_to_ZXM	172	FlxqXn_div	154
Flxn_div	145	FlxqXn_div_pre	154
Flxn_div_pre	145	FlxqXn_expint	154
Flxn_exp	145	FlxqXn_expint_pre	154
Flxn_expint	146	FlxqXn_inv	154
Flxn_inv	145	FlxqXn_inv_pre	154
Flxn_mul	145	FlxqXn_mul	153
Flxn_mul_pre	145	FlxqXn_mul_pre	153
Flxn_red	145	FlxqXn_sqr	153
Flxn_sqr	145	FlxqXn_sqr_pre	154
Flxn_sqr_pre	145	FlxqXQ_autpow	153
FlxqC_Flxq_mul	122	FlxqXQ_autpow_pre	153
FlxqC_red_pre	122	FlxqXQ_autsum	153

FlxqXQ_autsum_pre	153	FlxqX_Flxq_mul_to_monic_pre	150
FlxqXQ_auttrace	153	FlxqX_Frobenius	152
FlxqXQ_auttrace_pre	153	FlxqX_Frobenius_pre	152
FlxqXQ_div	153	FlxqX_fromNewton	152
FlxqXQ_div_pre	153	FlxqX_fromNewton_pre	152
FlxqXQ_halfFrobenius	153	FlxqX_gcd	151
FlxqXQ_inv	153, 159	FlxqX_gcd_pre	151
FlxqXQ_invsafe	153, 159	FlxqX_get_red	150
FlxqXQ_invsafe_pre	153	FlxqX_get_red_pre	150
FlxqXQ_inv_pre	153	FlxqX_halfgcd	151
FlxqXQ_matrix_pow	153	FlxqX_halfgcd_all	151
FlxqXQ_minpoly	153	FlxqX_halfgcd_all_pre	151
FlxqXQ_minpoly_pre	153	FlxqX_halfgcd_pre	151
FlxqXQ_mul	153	FlxqX_invBarrett	151
FlxqXQ_mul_pre	153	FlxqX_invBarrett_pre	151
FlxqXQ_pow	153	FlxqX_ispower	152
FlxqXQ_powers	153	FlxqX_is_squarefree	152
FlxqXQ_powers_pre	153	FlxqX_mul	150
FlxqXQ_powu	153	FlxqX_mul_pre	150
FlxqXQ_powu_pre	153	FlxqX_nbfact	152
FlxqXQ_pow_pre	153	FlxqX_nbfact_by_degree	152
FlxqXQ_sqr	153	FlxqX_nbfact_Frobenius	152
FlxqXQ_sqr_pre	153	FlxqX_nbroots	152
FlxqXV_prod	151	FlxqX_Newton	152
FlxqX_composedsum	151	FlxqX_Newton_pre	152
FlxqX_ddf	152	FlxqX_normalize	150
FlxqX_ddf_degree	152	FlxqX_normalize_pre	150
FlxqX_degfact	152	FlxqX_powu	150
FlxqX_disc	151	FlxqX_powu_pre	150
FlxqX_div	151	FlxqX_red	150
FlxqX_divrem	150	FlxqX_red_pre	150
FlxqX_divrem_pre	151	FlxqX_rem	151
FlxqX_div_by_X_x	151	FlxqX_rem_pre	151
FlxqX_div_by_X_x_pre	151	FlxqX_resultant	151
FlxqX_div_pre	151	FlxqX_resultant_pre	151
FlxqX_dotproduct	151	FlxqX_roots	152
FlxqX_eval	151	FlxqX_safegcd	151
FlxqX_extgcd	151	FlxqX_saferes resultant	151
FlxqX_extgcd_pre	151	FlxqX_sqr	150
FlxqX_factor	152	FlxqX_sqr_pre	150
FlxqX_factor_squarefree	152	Flxq_add	146
FlxqX_factor_squarefree_pre	152	Flxq_autpow	147
FlxqX_FlxqXQV_eval	152	Flxq_autpowers	147
FlxqX_FlxqXQV_eval_pre	152	Flxq_autpow_pre	147
FlxqX_FlxqXQ_eval	152	Flxq_autsum	147
FlxqX_FlxqXQ_eval_pre	152	Flxq_auttrace	147
FlxqX_Flxq_mul	150	Flxq_auttrace_pre	147
FlxqX_Flxq_mul_pre	150	Flxq_charpoly	148
FlxqX_Flxq_mul_to_monic	150	Flxq_conjvec	148

Flxq_div	146	FlxV_Flc_mul	144
Flxq_div_pre	146	FlxV_Flv_multieval	145
Flxq_ellcard	364	FlxV_Flx_fromdigits	141
Flxq_elldivpolmod	365	FlxV_prod	145
Flxq_ellgens	364	FlxV_red	145
Flxq_ellgroup	364	FlxV_to_Flm	174
Flxq_ellj	364	FlxV_to_FlxX	174
Flxq_ellj_to_a4a6	364	FlxV_to_ZXV	172
Flxq_elltwist	364	FlxV_to_ZXV_inplace	172
Flxq_ffisom_inv	147	FlxXC_sub	149
Flxq_inv	146	FlxXC_to_F2xXC	158
Flxq_invsafe	146	FlxXC_to_ZXXC	172
Flxq_invsafe_pre	146	FlxXM_to_ZXXM	172
Flxq_inv_pre	146	FlxXn_red	153
Flxq_is2npower	147	FlxXV_to_FlxM	174
Flxq_issquare	147	FlxX_add	148
Flxq_log	147	FlxX_blocks	149
Flxq_lroot	147	FlxX_deriv	148
Flxq_lroot_fast	147	FlxX_double	148
Flxq_lroot_fast_pre	148	FlxX_Fl_mul	148
Flxq_lroot_pre	147	FlxX_invLaplace	148
Flxq_matrix_pow	147	FlxX_Laplace	148
Flxq_matrix_pow_pre	147	FlxX_neg	148
Flxq_minpoly	148	FlxX_renormalize	149
Flxq_minpoly_pre	148	FlxX_resultant	149
Flxq_mul	146	FlxX_resultant_pre	149
Flxq_mul_pre	146	FlxX_shift	149
Flxq_norm	148	FlxX_sub	148
Flxq_order	147	FlxX_swap	149
Flxq_pow	146	FlxX_to_F2xX	157
Flxq_powers	146	FlxX_to_Flm	174
Flxq_powers_pre	147	FlxX_to_Flx	174
Flxq_powu	146	FlxX_to_FlxC	174
Flxq_powu_pre	146	FlxX_to_ZXX	172
Flxq_pow_init	146	FlxX_translate1	149
Flxq_pow_init_pre	146	FlxX_triple	148
Flxq_pow_pre	146	FlxYqq_pow	149
Flxq_pow_table	146	FlxY_degreex	148
Flxq_pow_table_pre	146	FlxY_evalx	148
Flxq_sqr	146	FlxY_evalx_powers_pre	149
Flxq_sqrt	147	FlxY_evalx_pre	149
Flxq_sqrtn	147	FlxY_eval_powers_pre	149
Flxq_sqrt_pre	147	FlxY_FlxqV_evalx	149
Flxq_sqr_pre	146	FlxY_FlxqV_evalx_pre	149
Flxq_sub	146	FlxY_Flxq_evalx	149
Flxq_trace	148	FlxY_Flxq_evalx_pre	149
FlxT_red	145	FlxY_Flx_add	148
FlxV_composedsum	145	FlxY_Flx_div	148
FlxV_factorback	145	FlxY_Flx_div_pre	148

FlxY_Flx_mul	148	Flx_Fl_mul	140
FlxY_Flx_mul_pre	148	Flx_Fl_mul_pre	140
FlxY_Flx_sub	148	Flx_Fl_mul_to_monic	140
FlxY_Flx_translate	149	Flx_Fl_sub	140
Flx_add	140	Flx_Fl_translate	141
Flx_blocks	143	Flx_Frobenius	141
Flx_composedprod	143	Flx_Frobenius_pre	141
Flx_composedsum	143	Flx_fromNewton	144
Flx_constant	140	Flx_gcd	141
Flx_conv	140	Flx_gcd_pre	141
Flx_copy	140	Flx_get_red	139
Flx_ddf	142	Flx_get_red_pre	139
Flx_ddf_pre	142	Flx_halfgcd	141
Flx_deflate	143	Flx_halfgcd_all	141
Flx_degfact	142, 144	Flx_halfgcd_all_pre	141
Flx_deriv	140	Flx_halfgcd_pre	141
Flx_diff1	141	Flx_halve	140
Flx_digits	141	Flx_inflate	143
Flx_div	140	Flx_integ	141
Flx_divrem	140	Flx_invBarrett	143
Flx_divrem_pre	140	Flx_invLaplace	144
Flx_div_by_X_x	143	Flx_ispower	142
Flx_div_pre	140	Flx_is_irred	144
Flx_dotproduct	143	Flx_is_smooth	144
Flx_dotproduct_pre	143	Flx_is_smooth_pre	144
Flx_double	140	Flx_is_squarefree	143
Flx_equal	140	Flx_is_totally_split	144
Flx_equal1	140	Flx_Laplace	144
Flx_eval	143	Flx_lead	139
Flx_eval_powers_pre	143	Flx_matFrobenius	141
Flx_eval_pre	143	Flx_matFrobenius_pre	141
Flx_extgcd	141	Flx_mod_Xn1	142
Flx_extgcd_pre	141	Flx_mod_Xnm1	142
Flx_extresultant	143	Flx_mul	140
Flx_extresultant_pre	143	Flx_mulu	140
Flx_factcyclo	142	Flx_mul_pre	140
Flx_factor	142	Flx_nbfact	144
Flx_factorff_irred	142	Flx_nbfact_by_degree	144
Flx_factor_squarefree	142	Flx_nbfact_Frobenius	144
Flx_factor_squarefree_pre	142	Flx_nbfact_Frobenius_pre	144
Flx_ffintersect	144	Flx_nbfact_pre	144
Flx_ffisom	142	Flx_nbroots	144
Flx_Flv_multieval	143	Flx_neg	140
Flx_FlxqV_eval	147	Flx_neg_inplace	140
Flx_FlxqV_eval_pre	147	Flx_Newton	144
Flx_Flxq_eval	147	Flx_normalize	142
Flx_Flxq_eval_pre	147	Flx_oneroot	141
Flx_FlxY_resultant	149	Flx_oneroot_pre	141
Flx_Fl_add	140	Flx_oneroot_split	142

Flx_oneroot_split_pre	142	Fl_elltwist_disc	361
Flx_powu	140	Fl_half	82
Flx_powu_pre	140	Fl_inv	82
Flx_recip	143	Fl_invgen	83
Flx_red	140	Fl_invsafe	82
Flx_rem	140	Fl_log	83
Flx_rem_pre	140	Fl_log_pre	84
Flx_renormalize	143	Fl_mul	82
Flx_rescale	142	Fl_mul_pre	84
Flx_resultant	143	Fl_neg	82
Flx_resultant_pre	143	Fl_order	83
Flx_roots	141	Fl_powers	83
Flx_rootsff	142	Fl_powers_pre	84
Flx_roots_pre	141	Fl_powu	83
Flx_shift	143	Fl_powu_pre	84
Flx_splitting	143	Fl_sqr	82
Flx_sqr	140	Fl_sqrt	83
Flx_sqr_pre	140	Fl_sqrtl	83
Flx_sub	140	Fl_sqrtl_pre	84
Flx_Teichmuller	144	Fl_sqrtn	83
Flx_to_F2x	154	Fl_sqrtn_pre	84
Flx_to_Flv	173	Fl_sqrt_pre	84
Flx_to_FlxX	172	Fl_sqrt_pre_i	85
Flx_to_ZX	172	Fl_sqr_pre	84
Flx_to_ZX_inplace	172	Fl_sub	82
Flx_translate1	141	Fl_to_Flx	173
Flx_translate1_basecase	141	Fl_triple	82
Flx_triple	140	forallsubset_init	42
Flx_unscale	142	forcomposite	41
Flx_val	143	forcomposite_init	41
Flx_valrem	143	forcomposite_next	41
Fly_to_FlxY	174	fordiv	40, 290
Fl_2gener_pre	85	fordivfactored	290
Fl_2gener_pre_i	85	foreachpari	290
Fl_add	82	forell	40
Fl_addmulmul_pre	84	forell(ell,a,b,,flag)	40
Fl_addmul_pre	84	forell0	290
Fl_center	82	forfactored	290
Fl_div	83	forksubset_init	42
Fl_double	82	format	38
Fl_elldisc	361	forpari	290
Fl_elldisc_pre	361	forpart	41
Fl_ellj	361	forpart0	290
Fl_ellj_pre	361	forpart_init	41
Fl_ellj_to_a4a6	361	forpart_next	41
Fl_ellptors	361	forpart_prev	41
Fl_elltrace	361	forpart_t	41
Fl_elltrace_CM	361	forperm	41
Fl_elltwist	361	forperm0	290

forperm_init	41	FpMs_FpCs_solve	197
forperm_next	41	FpMs_FpCs_solve_safe	197
forprime	41	FpMs_FpC_mul	196
forprimestep	41	FpMs_leftkernel_elt	196
forprimestep_init	42, 182	FpM_add	114
forprime_init	42, 43, 181, 182	FpM_center	113
forprime_next	42, 182	FpM_center_inplace	114
forprime_t	42, 43	FpM_charpoly	115
forqfvec	40	FpM_deplin	115
forqfvec1	40	FpM_det	115
forsquarefree	290	FpM_FpC_gauss	115
forstep	290	FpM_FpC_invimage	115
forsubgroup	40	FpM_FpC_mul	114
forsubgroup(H = G, B,)	40	FpM_FpC_mul_FpX	114
forsubgroup0	290	FpM_Fp_mul	114
forsubset	42	FpM_gauss	115
forsubset0	290	FpM_hess	115
forsubset_init	42	FpM_image	115
forsubset_next	42	FpM_indexrank	115
forvec	41	FpM_intersect	115
forvec_init	41	FpM_intersect_i	115
forvec_next	41	FpM_inv	115
FpC_add	114	FpM_invimage	115
FpC_center	113	FpM_ker	115
FpC_center_inplace	114	FpM_mul	114
FpC_FpV_mul	114	FpM_powu	114
FpC_Fp_mul	114	FpM_rank	115
FpC_ratlift	165	FpM_ratlift	165
FpC_red	113	FpM_red	113
FpC_sub	114	FpM_sub	114
FpC_to_mod	160	FpM_suppl	115
FpE_add	360	FpM_to_mod	160
FpE_changepoint	360	FpM_trace	115
FpE_changepointinv	360	FpVV_to_mod	160
FpE_dbl	360	FpV_add	114
FpE_log	361	FpV_dotproduct	114
FpE_mul	360	FpV_dotsquare	114
FpE_neg	360	FpV_factorback	114
FpE_order	361	FpV_FpC_mul	114
FpE_sub	360	FpV_FpMs_mul	196
FpE_tatepairing	361	FpV_FpM_polint	125
FpE_to_FpJ	362	FpV_inv	103
FpE_to_mod	361	FpV_invVandermonde	125
FpE_weilpairing	361	FpV_polint	125
FpJ_add	362	FpV_prod	103
FpJ_dbl	362	FpV_red	113
FpJ_mul	362	FpV_roots_to_pol	125
FpJ_neg	362	FpV_sub	114
FpJ_to_FpE	362	FpV_to_mod	160

FpXC_center	132	FpXQXV_FpQXQ_fromdigits	135
FpXC_FpXQV_eval	131	FpXQXV_prod	135
FpXC_FpXQ_eval	131	FpXQXV_red	134
FpXC_to_mod	160	FpXQX_ddf	138
FpXk_gcd	133	FpXQX_ddf_degree	138
FpXk_red	133	FpXQX_degfact	138
FpXM_center	132	FpXQX_digits	135
FpXM_FpXQV_eval	131	FpXQX_disc	135
FpXM_to_mod	160	FpXQX_div	135
FpXn_div	131	FpXQX_divrem	135
FpXn_exp	131	FpXQX_div_by_X_x	135
FpXn_expint	132	FpXQX_dotproduct	135
FpXn_inv	131	FpXQX_extgcd	135
FpXn_mul	131	FpXQX_factor	138
FpXn_sqr	131	FpXQX_factor_squarefree	138
FpXQC_to_mod	160	FpXQX_FpXQXQV_eval	136
FpXQE_add	366	FpXQX_FpXQXQ_eval	135
FpXQE_changepoint	366	FpXQX_FpXQ_mul	135
FpXQE_changepointinv	366	FpXQX_FpXQ_translate	135
FpXQE_dbl	366	FpXQX_Frobenius	139
FpXQE_log	366	FpXQX_gcd	135
FpXQE_mul	366	FpXQX_get_red	136
FpXQE_neg	366	FpXQX_halfgcd	135
FpXQE_order	366	FpXQX_halfgcd_all	135
FpXQE_sub	366	FpXQX_invBarrett	135
FpXQE_tatepairing	366	FpXQX_isplayer	138
FpXQE_weilpairing	366	FpXQX_is_squarefree	137
FpXQM_autsum	131	FpXQX_mul	134
FpXQV_ellcharpoly	365	FpXQX_nbfact	139
FpXQXn_div	135	FpXQX_nbfact_Frobenius	139
FpXQXn_exp	135	FpXQX_nbroots	139
FpXQXn_expint	136	FpXQX_normalize	133
FpXQXn_inv	135	FpXQX_powu	135
FpXQXn_mul	135	FpXQX_red	134
FpXQXn_sqr	135	FpXQX_rem	135
FpXQXQ_autpow	137	FpXQX_renormalize	134
FpXQXQ_autsum	137	FpXQX_resultant	135
FpXQXQ_auttrace	137	FpXQX_roots	137
FpXQXQ_div	136	FpXQX_roots_mult	137
FpXQXQ_halfFrobenius	136	FpXQX_split_part	138
FpXQXQ_inv	136	FpXQX_sqr	135
FpXQXQ_invsafe	136	FpXQX_to_mod	160
FpXQXQ_matrix_pow	137	FpXQ_add	129
FpXQXQ_minpoly	137	FpXQ_autpow	131
FpXQXQ_mul	136	FpXQ_autpowers	131
FpXQXQ_pow	136	FpXQ_autsum	131
FpXQXQ_powers	136	FpXQ_auttrace	131
FpXQXQ_sqr	136	FpXQ_charpoly	130
FpXQXT_red	134	FpXQ_conjvec	130

FpXQ_div	129	FpXY_eval	132
FpXQ_ellcard	365	FpXY_evalx	132
FpXQ_ellcard_supersingular	365	FpXY_evaly	132
FpXQ_elldivpol	366	FpXY_FpXQV_evalx	133
FpXQ_ellgens	366	FpXY_FpXQ_evalx	133
FpXQ_ellgroup	365	FpXY_FpXQ_evaly	132
FpXQ_ellj	365	FpXY_Fq_evaly	133
FpXQ_elljissupersingular	365	FpX_add	123
FpXQ_elltwist	366	FpX_center	124
FpXQ_ffisom_inv	139	FpX_center_i	124
FpXQ_inv	129	FpX_chinese_coprime	125
FpXQ_invsafe	129	FpX_composedprod	127
FpXQ_issquare	129	FpX_composedsum	127
FpXQ_log	129, 130, 147	FpX_convolve	123
FpXQ_matrix_pow	131	FpX_ddf	126
FpXQ_minpoly	130	FpX_ddf_degree	126
FpXQ_mul	129	FpX_degfact	126, 142, 155
FpXQ_norm	130	FpX_deriv	123
FpXQ_order	129, 147	FpX_digits	123
FpXQ_pow	129	FpX_disc	127
FpXQ_powers	131	FpX_div	123
FpXQ_powu	129	FpX_divrem	123
FpXQ_red	129	FpX_divu	124
FpXQ_sqr	129	FpX_div_by_X_x	123
FpXQ_sqrt	129	FpX_dotproduct	125
FpXQ_sqrtn	129, 130, 147	FpX_eval	125
FpXQ_sub	129	FpX_extgcd	124
FpXQ_trace	130	FpX_extresultant	127
FpXT_red	123	FpX_factcyclo	126
FpXV_chinese	125	FpX_factor	126
FpXV_composedsum	125	FpX_factorff	138
FpXV_factorback	125	FpX_factorff_irred	138, 142
FpXV_FpC_mul	125	FpX_factor_squarefree	126
FpXV_FpX_fromdigits	124	FpX_ffintersect	138
FpXV_prod	125	FpX_ffisom	138, 142
FpXV_red	123	FpX_FpC_nfpoleval	308
FpXX_add	132	FpX_FpV_multieval	125
FpXX_deriv	132	FpX_FpXQV_eval	131
FpXX_FpX_mul	132	FpX_FpXQ_eval	131
FpXX_Fp_mul	132	FpX_FpXV_multirem	125
FpXX_Fp_sub	132	FpX_FpXY_resultant	127
FpXX_halve	132	FpX_Fp_add	124
FpXX_integ	132	FpX_Fp_add_shallow	124
FpXX_mulu	132	FpX_Fp_div	124
FpXX_neg	132	FpX_Fp_mul	124
FpXX_red	132	FpX_Fp_mulspec	124
FpXX_renormalize	132	FpX_Fp_mul_to_monoid	124
FpXX_sub	132	FpX_Fp_sub	124
FpXYQQ_pow	133	FpX_Fp_sub_shallow	124

FpX_Fp_translate	124	Fp_center	103
FpX_Frobenius	124	Fp_center_i	103
FpX_fromNewton	127	Fp_div	104
FpX_gcd	124	Fp_divu	104
FpX_gcd_check	323	Fp_double	103
FpX_get_red	127	Fp_ellcard	360
FpX_halfgcd	124	Fp_ellcard_SEA	360
FpX_halfgcd_all	124	Fp_elldivpol	360
FpX_halfve	123	Fp_ellgens	360
FpX_integ	123	Fp_ellgroup	360
FpX_invBarret	127	Fp_ellj	359
FpX_invBarrett	125	Fp_elljissupersingular	360
FpX_invLaplace	127	Fp_ellj_to_a4a6	359
FpX_ispower	126	Fp_elltwist	360
FpX_is_irred	126, 155	Fp_factored_order	104
FpX_is_squarefree	126	Fp_ffellcard	360
FpX_is_totally_split	126	Fp_FpXQ_log	129
FpX_Laplace	127	Fp_FpX_sub	124
FpX_matFrobenius	124	Fp_halfve	103
FpX_mul	123	Fp_inv	103
FpX_mulspec	123	Fp_invgen	103
FpX_mulu	124	Fp_invsafe	103
FpX_nbfact	126	Fp_ispower	104
FpX_nbfact_Frobenius	126	Fp_issquare	104
FpX_nbroots	126	Fp_log	104, 130
FpX_neg	123	Fp_modinv_to_j	367
FpX_Newton	127	Fp_mul	103
FpX_normalize	125	Fp_muls	103
FpX_oneroot	126	Fp_mulu	103
FpX_oneroot_split	126	Fp_neg	102
FpX_powu	123	Fp_order	104
FpX_ratlift	165	Fp_polmodular_evalx	367
FpX_red	123	Fp_pow	103
FpX_rem	123	Fp_powers	103
FpX_renormalize	123	Fp_pows	103
FpX_rescale	125	Fp_powu	103
FpX_resultant	127	Fp_pow_init	103
FpX_roots	126	Fp_pow_table	103
FpX_rootsff	138, 142	Fp_ratlift	165
FpX_roots_mult	126	Fp_red	102
FpX_split_part	126	Fp_sqr	103
FpX_sqr	123	Fp_sqrt	104
FpX_sub	123	Fp_sqrttn	104
FpX_to_mod	160	Fp_sqrt_i	104
FpX_valrem	123	Fp_sub	102
Fp_2gener	104	Fp_to_mod	160
Fp_2gener_i	104	FqC_add	115
Fp_add	15, 102	FqC_FqV_mul	115
Fp_addmul	103	FqC_Fq_mul	115

FqC_red	128	FqXY_eval	134
FqC_sub	115	FqXY_evalx	134
FqC_to_FFC	253	FqX_add	133
FqC_to_FlxqC	115	FqX_ddf	138
FqC_to_mod	160	FqX_degfact	138
FqM_add	115	FqX_deriv	134
FqM_deplin	116	FqX_div	134
FqM_det	116	FqX_divrem	134
FqM_FqC_gauss	116	FqX_div_by_X_x	134
FqM_FqC_invimage	116	FqX_eval	134
FqM_FqC_mul	116	FqX_extgcd	134
FqM_gauss	116	FqX_factor	138
FqM_image	116	FqX_factor_squarefree	138
FqM_indexrank	116	FqX_Fp_mul	133
FqM_inv	116	FqX_Fq_add	133
FqM_invimage	116	FqX_Fq_mul	133
FqM_ker	116	FqX_Fq_mul_to_monic	133
FqM_mul	116	FqX_Fq_sub	133
FqM_rank	116	FqX_Fq_translate	134
FqM_red	129	FqX_gcd	134
FqM_sub	116	FqX_get_red	136
FqM_suppl	116	FqX_halfgcd	134
FqM_to_FFM	253	FqX_halve	133
FqM_to_FlxqM	115	FqX_integ	134
FqM_to_mod	160	FqX_isplayer	138
FqM_to_nfM	319	FqX_is_squarefree	137
FqV_factorback	130	FqX_mul	133
FqV_inv	130	FqX_mulu	133
FqV_red	129	FqX_nbfact	139
FqV_roots_to_pol	137	FqX_nbroots	139
FqV_to_FFV	253	FqX_neg	133
FqV_to_nfV	319	FqX_normalize	133
FqXC_to_mod	161	FqX_powu	134
FqXM_to_mod	161	FqX_red	128
FqXn_exp	135	FqX_rem	134
FqXn_expint	136	FqX_roots	138
FqXn_inv	135	FqX_sqr	133
FqXn_mul	135	FqX_sub	133
FqXn_sqr	135	FqX_to_FFX	253
FqXQ_add	137	FqX_to_mod	161
FqXQ_div	137	FqX_to_nfX	319
FqXQ_inv	137	Fq_add	129
FqXQ_invsafe	137	Fq_div	130
FqXQ_matrix_pow	137	Fq_ellcard_SEA	365
FqXQ_mul	137	Fq_ellcard_supersingular	365
FqXQ_pow	137	Fq_elldivpolmod	366
FqXQ_powers	137	Fq_elljissupersingular	365
FqXQ_sqr	137	Fq_Fp_mul	130
FqXQ_sub	137	Fq_halve	130

Fq_inv	130	gal_get_group	333
Fq_invsafe	130	gal_get_invvdm	333
Fq_ispower	130	gal_get_mod	333
Fq_issquare	130	gal_get_orders	333
Fq_log	130	gal_get_p	333
Fq_mul	130	gal_get_pol	333
Fq_mulu	130	gal_get_roots	333
Fq_neg	130	gammamellininv	371
Fq_neg_inv	130	gammamellinininit	371
Fq_pow	130	gammamellininvrt	371
Fq_powu	130	gand	237
Fq_red	128	garbage collecting	16
Fq_sqr	130	gassoc_proto	106
Fq_sqrt	130	gaussred_from_QR	194
Fq_sqrtn	130	gbezout	241
Fq_sub	129	gboundcf	102
Fq_to_FF	253	gcdext	47
Fq_to_FpXQ	128	gcdii	101
Fq_to_nf	318	gceil	234
fractor	223	gchari_lfun	348
fractor_lg	223	gclone	25, 71, 72
Frobeniusform	194	gcloneref	72
fromdigitsu	90	gclone_refc	73
fromdigits_2k	91	gcmp	235
fujiwara_bound	252	gcmpgs	236
fujiwara_bound_real	252	gcmpsg	236
fun(E, ell)	40	gcoeff	15, 64, 283
fun(E, H)	40	gconj	249
functions_basic	54	gcopy	24, 25, 72
functions_default	54	gcopy_avma	71
functions_gp	54	gcopy_lg	72
fuse_Z_factor	179	gcvtoi	234
f_PRETTYMAT	269	gcvtop	224
f_RAW	269, 270	gc_all	20, 69
f_TEX	269, 270	gc_all_unsafe	70
G		gc_bool	69
		gc_const	69
gabs	242	gc_double	69
gadd	243	gc_GEN	18, 19, 68
gaddgs	15, 243	gc_INT	68
gaddsg	15, 243	gc_int	69
galoisexport	267	gc_INT_affii	68
galoisidentify	267	gc_leaf	70
galoisinit	264, 333	gc_long	69
galois_group	264	gc_needed	22
gal_get_den	333	gc_NULL	69
gal_get_e	333	gc_slice	70
gal_get_gen	333	gc_slice_unsafe	70
		gc_stack_update	18, 71

gc_stoi	69	gen_gener	216
gc_ulong	69	gen_indexsort	238
gc_upto	19, 25, 70	gen_indexsort_uniq	238
gc_utoi	69	gen_ker	218
gc_utoipos	69	gen_m1	13
gdeuc	240	gen_m2	13
gdiv	243	gen_matcolinvimage	218
gdivent	239	gen_matcolmul	218
gdiventgs	239	gen_matid	218
gdiventres	239	gen_matinvimage	218
gdiventsg	239	gen_matmul	218
gdivexact	239	gen_order	216
gdivgs	243	gen_parapply	44
gdivgu	239, 243	gen_parapply_percent	44
gdivgunextu	239	gen_parapply_slice	45
gdivmod	240	gen_parapply_slice_zv	45
gdivround	240	gen_parpairwiseop	45
gdivsg	243	gen_PH_log	216
gdvd	239	gen_plog	216
gel	14, 15, 64, 283	gen_Pollard_log	216
GEN	13	gen_pow	244
GENbinbase	68	gen_powers	219, 245
gener_F2xq	156	gen_powu	244
gener_Flxq	148	gen_powu_fold	245
gener_FpXQ	130	gen_powu_fold_i	245
gener_FpXQ_local	130	gen_powu_i	244
gener_Fq_local	131	gen_pow_fold	244, 245
GENtoGENstr	269	gen_pow_fold_i	244, 245
GENtoGENstr_nospace	269	gen_pow_i	244
GENtostr	37, 269	gen_pow_init	245
GENtostr_raw	269	gen_pow_table	245
GENtostr_unquoted	269	gen_product	244
GENtoTeXstr	37, 269	gen_RgX_bkeval	219
gen_0	13, 30	gen_search	238
gen_1	13	gen_select_order	217
gen_2	13	gen_setminus	238
gen_bkeval	219	gen_Shanks	216
gen_bkeval_powers	219	gen_Shanks_init	216
gen_cmp_RgX	239	gen_Shanks_log	216
gen_det	218	gen_Shanks_sqrtn	216
gen_digits	220	gen_sort	237
gen_ellgens	217, 357	gen_sort_inplace	238
gen_ellgroup	217	gen_sort_shallow	238
gen_factorback	245	gen_sort_uniq	238
gen_factored_order	216	gen_ZpM_Dixon	197
gen_FpM_Wiedemann	197	gen_ZpM_Dixon_Wiedemann	197
gen_fromdigits	220	gen_ZpM_Newton	221
gen_Gauss	218	gen_ZpXX_Newton	221
gen_Gauss_pivot	218	gen_ZpX_Dixon	221

gen_ZpX_Newton	221	get_FpXQ_star	217
geq	237	get_FpX_algebra	220
gequal	206, 235	get_FpX_degree	128
gequal0	236	get_FpX_mod	128
gequal1	236	get_FpX_var	128
gequalgs	236	get_Fp_field	218
gequalm1	236	get_Fq_field	218
gequalsg	236	get_lex	291
gequalX	235	get_modpr	300
gerepile	68	get_nf	299
getheap	73	get_nfpol	299
getrand	102	get_nf_field	219
getrealprecision	258	get_prid	300
gettime	40	get_Rg_algebra	220
get_arith_Z	217	gexpo	28, 60
get_arith_ZZM	217	gexpo_safe	60
get_avma	68	gfloor	234
get_bnf	299	gfrac	234
get_bnfpol	299	ggamma1m1	260
get_F2xqE_group	217	ggcd	241
get_F2xqX_degree	158	gge	237
get_F2xqX_mod	158	ggt	237
get_F2xqX_var	158	ghalf	13
get_F2xq_field	219	gidentical	194, 235
get_F2x_degree	154	gimag	249
get_F2x_mod	154	ginv	243
get_F2x_var	154	ginvmod	240
get_FlxqE_group	217	gisdouble	223
get_FlxqXQ_algebra	220	gisexactzero	235
get_FlxqX_algebra	220	glcm	241
get_FlxqX_degree	150	gle	236
get_FlxqX_mod	150	glt	236
get_FlxqX_var	150	gmael	15, 64
get_Flxq_algebra	220	gmael1	15
get_Flxq_field	218	gmael2	64
get_Flxq_star	217	gmael3	64
get_Flx_algebra	220	gmael4	64
get_Flx_degree	139	gmael5	64
get_Flx_mod	139	gmax	235
get_Flx_var	139	gmaxgs	236
get_Fl_red	83	gmaxsg	236
get_FpE_group	217	gmax_shallow	235
get_FpXQE_group	217	gmin	235
get_FpXQXQ_algebra	220	gmings	236
get_FpXQX_algebra	220	gminsg	236
get_FpXQX_degree	136	gmin_shallow	235
get_FpXQX_mod	136	gmod	240
get_FpXQX_var	136	gmodgs	240
get_FpXQ_algebra	220	gmodsg	240

gmodulgs	224	gp_handle_exception	55
gmodulo	224	gp_help	57
gmodulsg	224	gp_load_gprc	56
gmodulss	224	gp_meta	56
gmul	243	gp_read_file	36, 56
gmul2n	234	gp_read_str	34, 35, 56, 78
gmulgs	243	gp_read_stream	36
gmulgu	243	gp_read_str_bitprec	36
gmulsg	243	gp_read_str_multiline	35
gmulug	243	gp_read_str_prec	35
gne	237	gp_sigint_fun	55
gneg	242	Gram matrix	191
gneg_i	242	gram_matrix	191
gnorml1	245, 246	greal	249
gnorml1_fake	246	gred_rfac2	31
gnorml2	245	grem	240
gnot	237	grndtoi	234
gor	237	grootsof1	244
<i>GP prototype</i>	75	ground	234
gphelp_keyword_list	57	groupelts_abelian_group	266
gpinstall	56	groupelts_center	266
gpow	243	groupelts_conjclasses	265
gpowers	244	groupelts_conj_set	265
gpowgs	243	groupelts_exponent	266
gprec	224	groupelts_gen	265
gprecision	61	groupelts_quotient	266
gprec_w	224	groupelts_set	265
gprec_wensure	224	groupelts_solvablesubgroups	266
gprec_wtrunc	224	groupelts_to_group	265
gprimepi_lower_bound	180	groupgen_export	267
gprimepi_upper_bound	180	group_abelianHNF	265
gp_alarm_handler	57	group_abelianSNF	266
gp_call	292	group_domain	265
gp_call2	293	group_elts	265
gp_callbool	293	group_export	266
gp_callprec	293	group_ident	267
gp_callvoid	293	group_ident_trans	267
gp_context_restore	57	group_isA4S4	266
gp_context_save	57	group_isabelian	265
gp_display_hist	57	group_istrivial	265
gp_echo_and_log	57	group_iswss	265
gp_eval	292	group_leftcoset	266
gp_evalbool	292	group_order	265
gp_evalprec	292	group_perm_normalize	266
gp_evalupto	292	group_quotient	266
gp_evalvoid	292	group_rightcoset	266
gp_filter	56	group_set	265
gp_format_prompt	57	group_subgroups	266
gp_format_time	56	group_subgroup_isnormal	266

group_subgroup_is_faithful	265	halfgcdii	102
group_to_cc	265	hammingu	81
gshift	234	hashentry	279
gsigne	28, 60	hashtable	279
gsincos	260	hash_create	280
gsizebyte	24	hash_create_GEN	280
gsizeword	24	hash_create_INT	280
gsmith	343	hash_create_str	280
gsmithall	343	hash_create_ulong	280
gsprintf	270	hash_dbg	281
gsqr	243	hash_destroy	281
gsqrpowers	244	hash_GEN	281
GSTR	32	hash_haskey_GEN	281
gsub	243	hash_haskey_long	281
gsubgs	243	hash_init	280
gsubsg	243	hash_init_GEN	280
gsubst	246	hash_init_ulong	280
gsupnorm	246	hash_insert	280
gsupnorm_aux	246	hash_insert2	280
gtocol	227	hash_insert_long	280
gtodouble	25, 223	hash_keys	281
gtofp	223, 224	hash_keys_GEN	281
gtolong	25, 223	hash_remove	281
gtomat	227	hash_remove_select	281
gtomp	224	hash_search	280, 281
gtopoly	225	hash_search2	281
gtopolyrev	225	hash_select	281
gtos	223	hash_str	280, 281
gtoser	226	hash_str_len	281
gtoser_prec	226, 227	hash_values	281
gtou	223	hash_values_GEN	281
gtovec	227	hash_zv	281
gtovecsmall	227	hclassno	335
gtrans	283	hclassno6	335
gtrunc	234	hclassno6u	335
gtrunc2n	90, 234	hclassno6u_no_cache	335
gunclone	25, 73	hclassnoF_fact	335
guncloneNULL	73	heap	14
guncloneNULL_deep	73	Hermite_bound	344
gunclone_deep	73	hexadecimal tree	38
gval	234	HIGHBIT	64
gvaluation	234	HIGHEXPBIT	65
gvar	30, 33, 60	HIGHMASK	65
gvar2	61	HIGHVALPBIT	65
gvsprintf	270	HIGHWORD	65
G_ZGC_mul	195	hilbertii	108
G_ZG_mul	195	hnf	342
		hnfall	342
		hnfddivide	340

H

hnflll	342	idealhnf_principal	314
hnfmerge_get_1	315	idealhnf_shallow	314
hnfmod	342	idealhnf_two	314
hnfmodid	342	idealHNF_Z_factor	317
hnfperm	342	idealHNF_Z_factor_i	317
hnf_CENTER	339	idealinv	314
hnf_divscale	340	ideallog	313
hnf_invimage	340	ideallog_units	329
hnf_invscale	340	ideallog_units0	329
hnf_MODID	339	idealmoddivisor	326
hnf_PART	339	idealmul	314
hnf_solve	340	idealmulpowprime	314
hnf_Znproj	319	idealmulred	314, 323
hqfeval	246	idealpow	314
hyperell_locally_soluble	367	idealpowred	314
h_APROPOS	57	idealpows	314
h_LONG	57	idealprimedec	313, 316
h_REGULAR	57	idealprimedec_degrees	316
		idealprimedec_galois	316
		idealprimedec_kummer	317
		idealprimedec_limit_f	316
		idealprimedec_limit_norm	316
		idealprincipalunits	321
		idealprod	314
		idealprodprime	314
		idealprodval	315
		idealpseudomin	326
		idealpseudominvec	326
		idealpseudomin_nonscalar	326
		idealpseudored	326
		idealramfrobenius	318
		idealramfrobenius_aut	318
		idealramgroups_aut	318
		idealred	326
		idealred0	326
		idealred_elt	326
		idealsqr	314
		idealstar	321
		idealstar0	332
		Idealstarmod	304
		Idealstarprk	327
		ideals_by_norm	315
		idealtyp	300
		identity_perm	263
		identity_ZV	183
		identity_zv	187
		id_MAT	300
		id_PRIME	300
		id_PRINCIPAL	300
I			
icopy	87		
icopyifstack	72		
icopyspec	88		
icopy_avma	71		
idealadd	314		
idealaddmultoone	315		
idealaddtoone	315		
idealaddtoone_i	315		
idealaddtoone_raw	315		
idealappr	315, 333		
idealappr0	333		
idealapprfact	315		
idealchinese	316		
idealchineseinit	316		
idealcoprime	315, 316		
idealcoprimefact	316		
idealdiv	314		
idealdivexact	314		
idealdivpowprime	314		
idealfactor	313, 316, 317		
idealfactor_limit	317		
idealfactor_partial	317		
idealfrobenius_aut	318		
idealhnf	313, 314		
idealhnf0	313		
idealHNF_inv	315		
idealHNF_inv_Z	315		
idealHNF_mul	315		

ifac_isprime	180	isintm1	235
ifac_next	179	isintzero	235
ifac_read	179	ismpzzero	235
ifac_skip	179	isonstack	72
ifac_start	179	isprime	181
iferrpari	290	isprimeAPRCL	181
ifpari	290	isprimeECPP	181
image	197	isprimepower	107
image2	197	isprincipal	324
imag_i	249	isprincipalfact	324
indexlexsort	237	isprincipalfact_or_fail	324
indexpartial	323	isprincipalforce	332
indexsort	237	isprincipalgen	332
indexvecsort	237	isprincipalgenforce	332
indices_to_vec01	320	isprincipalraygen	332
infinity	32	isrationalzero	235
inf_get_sign	32	isrationalzeroscalar	236
initprimetable	66	isrealappr	236
init_Flxq	142	issmall	236
init_Fq	137	is_357_power	107, 175
<i>inline closure</i>	76	is_bigint	88
input	35	is_const_t	63
install	34, 39, 78, 79	is_entry	74
int2n	86	is_extscalar_t	63
int2u	86	is_gchar_group	348
int2um1	86	is_intreal_t	63
intcirc0	290	is_linit	369
integer	26	is_matvec_t	63
integser	252	is_nf_extfactor	300
intfuncinit0	290	is_nf_factor	300
intnum0	290	is_noncalc_t	63
int_LSW	27	is_pm1	236
int_MSW	27	is_pth_power	175
int_nextW	27	is_qfb_t	63
int_normalize	27	is_rational_t	63
int_precW	27	is_real_t	63
int_W	27	is_recursive_t	63
int_W_lg	27	is_scalar_t	63
invmod	104	is_universal_constant	87
invmod2BIL	82	is_vec_t	63
invr	97	is_Z_factor	178
inv_content	242	is_Z_factornon0	178
isclone	26	is_Z_factorpos	178
iscomplex	236	itor	88
isexactzero	235	itor_lg	88
isinexact	236	itos	25, 88, 223
isinexactreal	236	itostr	269
isint	236	itos_or_0	88
isint1	235	itou	88, 223

itou_or_0	88	lfunlambda	371
K		lfunmisc_to_ldata	370
killblock	72	lfunmisc_to_ldata_shallow	370
krois	105	lfunmisc_to_ldata_shallow_i	370
kroiu	105	lfunprod_get_fact	369
Kronecker symbol	105	lfunquadneg	371
kronecker	105	lfunrootno	371
Kronecker_to_F2xqX	158	lfunrttopoles	370
Kronecker_to_FlxqX	150	lfunthetacheckinit	371
Kronecker_to_FlxqX_pre	150	lfunthetacost	371
Kronecker_to_FpXqX	134	lfunzetakinit	371
Kronecker_to_mod	214	lfun_get_bitprec	369
Kronecker_to_ZXqX	203	lfun_get_dom	369
Kronecker_to_ZXX	203	lfun_get_domain	369
krosi	105	lfun_get_expot	369
kross	105	lfun_get_factgammavec	369
kroui	105	lfun_get_k2	369
krouu	105	lfun_get_pol	369
L		lfun_get_Residue	369
laurentseries0	290	lfun_get_step	369
lcmii	101	lfun_get_w2	369
ldata_get_an	369	lg	26, 60
ldata_get_conductor	369	lg2prec	59
ldata_get_degree	369	LGBITS	65
ldata_get_dual	369	lgcols	61
ldata_get_gammavec	369	lgefint	27, 60
ldata_get_k	369	LGnumBITS	65
ldata_get_k1	369	lgpol	61
ldata_get_residue	369	lg_increase	26
ldata_get_rootno	369	library mode	13
ldata_get_type	369	lift	225
ldata_isreal	369	lift0	225
ldata_newprec	370	liftall	225
ldata_vecan	370	liftall_shallow	225
leading_coeff	30, 64	liftint	225
leafcopy	87, 283	liftint_shallow	225
leafcopy_avma	87	liftpol	225
Legendre symbol	105	liftpol_shallow	225
lexcmp	235	lift_shallow	224
lexsort	237	lincombii	97
lfun	370	lindep	344
lfunabelianreinit	371	lindep2	345
lfunellmfpeters	371	lindepfull_bit	345
lfungchar	348	lindep_bit	344
lfuninit	371	lindep_padic	345
lfuninit_make	371	lindep_Xadic	345
		linit_get_ldata	369
		linit_get_tech	369
		linit_get_type	369

MF_get_fields	375	mkmat5	231
MF_get_gk	375	mkmatcopy	229
mf_get_gk	376	mk moo	32
MF_get_gN	375	mkoo	32
mf_get_gN	376	mkpadic	230
MF_get_k	375	mkpolmod	230
mf_get_k	376	mkpoln	24, 233
MF_get_M	376	mkqfb	231
MF_get_Mindex	376	mkquad	230
MF_get_Minv	376	mkfrac	230
MF_get_N	375	mkfraccopy	229
mf_get_N	376	mkvec	231
MF_get_newforms	375	mkvec2	231
mf_get_NK	376	mkvec2copy	229
MF_get_r	375	mkvec2s	229
mf_get_r	376	mkvec3	231
MF_get_S	375, 376	mkvec3s	229
MF_get_space	375	mkvec4	231
mf_get_type	376	mkvec4s	229
millerrabin	181	mkvec5	231
mindd	93	mkvec5s	229
minss	93	mkvec6	231
minuu	93	mkvec6s	229
mkcol	230	mkveccopy	229
mkcol2	230	mkvecn	24, 233
mkcol2s	229	mkvecs	229
mkcol3	230	mkvecs small	229
mkcol3s	229	mkvecs small2	229
mkcol4	230	mkvecs small3	229
mkcol4s	229	mkvecs small4	229
mkcol5	230	mkvecs small5	229
mkcol6	230	mkvecs smalln	229
mkcolcopy	229	Mod16	100
mkcoln	24, 233	mod16	100
mkcols	229	Mod2	100
mkcomplex	230	mod2	100
mkerr	231	mod2BIL	100
mkfrac	230	Mod32	100
mkfraccopy	229	mod32	100
mkfracss	228	Mod4	100
mkintmod	230	mod4	100
mkintmodu	228	Mod64	100
mkintn	88, 233	mod64	100
mkmat	230	Mod8	100
mkmat2	230	mod8	100
mkmat22	230	modinv_good_disc	367
mkmat22s	229	modinv_good_prime	367
mkmat3	231	modinv_height_factor	367
mkmat4	231	modinv_is_double_eta	367

ndec2nlong	59	nfnewprec_shallow	307
ndec2prec	59	nfnorm	308
negi	92	nfpoleval	308
negr	93	nfpow	308
newblock	72	nfpowmodideal	309
new_chunk	67	nfpow_u	308
new_chunk_resize	67	nfrootsof1	332
nfadd	307	nfroots_if_split	331
nfalgtobasis	308	nfsign	313, 320
nfarchstar	321	nfsign_arch	313, 320
nfbasistoalg	308	nfsign_from_logarch	321
nfchecksigns	320	nfsign_fu	320
nfcyclotomicunits	332	nfsign_tu	321
nfC_multable_mul	309	nfsign_units	320
nfC_nf_mul	309	nfsqr	308
nfdiv	308	nfsqri	309
nfdivauc	308	nfsu	308
nfdivrem	308	nftrace	308
nfembed_i	321	nftyp	299
nfeltup	330	nfval	308, 319
nfembed	321	nfV_cxlog	322
nffactorback	313	nfV_idealhnf	314
nfgaloisconj	331	nfV_to_FqV	319
nfgaloismatrix	334	nfV_to_scalar_or_alg	308
nfgaloismatrixapply	334	nfX_disc	309
nfgaloispermtobasis	333	nfX_resultant	309
nfgcd	205	nfX_to_FqX	319
nfgcd_all	205	nfX_to_monic	309
nfgwkummer	330	nf_cxlog	321
nfinit_basic	322	nf_cxlog_normalize	321
nfinit_complete	323	nf_deg1_prime	317
nfinv	308	nf_FORCE	324
nfinvmodideal	309	nf_GEN	323
nfissquarefree	330	nf_GENMAT	323, 324
nflogembed	322	nf_GEN_IF_PRINCIPAL	324
nfmaxord	322	nf_get_allroots	301
nfmaxord_t	322, 323	nf_get_degree	301
nfmaxord_to_nf	322, 323	nf_get_diff	301
nfmod	308	nf_get_disc	301
nfmodprinit	318, 319	nf_get_G	301
nfmul	308	nf_get_Gtwist	325, 326, 344
nfmuli	309	nf_get_Gtwist1	325
nfM_det	310	nf_get_index	301
nfM_inv	310	nf_get_invzk	301
nfM_ker	310	nf_get_M	301, 324
nfM_mul	310	nf_get_pol	301
nfM_nfC_mul	310	nf_get_prec	301
nfM_to_FqM	319	nf_get_r1	301
nfnewprec	307	nf_get_r2	301

parforstep_next	44	pari_err_TYPE	354
parforstep_stop	44	pari_fclose	273
parforvec	44	pari_flush	36, 271
parforvec_init	44	pari_fopen	273
parforvec_next	44	pari_fopengz	273
parforvec_stop	44	pari_fopen_or_fail	273
parfor_init	43	pari_fprintf	37
parfor_next	43	pari_fread_chars	272
parfor_stop	43	pari_free	16, 67
paricfg_buildinfo	80	pari_get_hist	57
paricfg_compileddate	80	pari_get_histrtime	57
paricfg_datadir	80	pari_get_histtime	57
paricfg_gphelp	80	pari_get_homedir	273
paricfg_mt_engine	80	pari_histtime	57
paricfg_vcsversion	80	pari_hit_return	56
paricfg_version	80	pari_infile	56
paricfg_version_code	80	pari_init	13, 14, 51
pariErr	271	pari_init_opts	51
PariOUT	270	pari_init_primes	52, 53
pariOut	271	pari_is_default	293
pariplot0	290	pari_is_dir	272
paristack_newsize	54	pari_is_file	272
paristack_resize	53	pari_kernel_close	52
paristack_setsize	53	pari_kernel_init	52
parivstack_reset	54	pari_kernel_version	80
parivstack_resize	54	pari_kill_plot_engine	379
pari_add_defaults_module	54	pari_last_was_newline	271
pari_add_function	54	pari_library_path	56
pari_add_hist	57	pari_malloc	16, 67, 277
pari_add_module	54	pari_mt_close	53
pari_alarm	57	pari_mt_init	52
pari_ask_confirm	56	pari_nb_hist	57
pari_base64	270	PARI_OLD_NAMES	14
pari_breakpoint	290	pari_outfile	36, 271
pari_calloc	16	PARI_plot	379
pari_CATCH	45	pari_plot_by_file	381
pari_CATCH_reset	46	pari PRIMES	14
pari_center	56	pari_printf	36, 38, 75, 271, 272
pari_close	51	pari_print_version	56
pari_close_opts	53	pari_putc	36, 75, 271
pari_community	56	pari_puts	36, 75, 271, 272
pari_compile_str	56	pari_rand	102
pari_daemon	53	pari_realloc	16, 277
pari_ENDCATCH	46	pari_realloc_ip	16
pari_err	32, 37, 38, 46, 274, 295	pari_RETRY	46
pari_err2str	279	pari_safefopen	273
pari_errfile	271	pari_set_last_newline	271
pari_err_display	279	pari_set_plot_engine	379
pari_err_last	46	pari_sighandler	53

pari_sig_init	53	perm_sqr	263
pari_sp	17	perm_to_GAP	264
pari_sprintf	37, 269	perm_to_Z	264
pari_stackcheck_init	53	pgener_Fl	83
pari_stack_alloc	282	pgener_Fl_local	83
pari_stack_base	283	pgener_Fp	105
pari_stack_delete	283	pgener_Fp_local	105
pari_stack_init	282	pgener_Zl	83
pari_stack_new	282	pgener_Zp	105
pari_stack_pushp	283	Pi2n	263
pari_stdin_isatty	273	PiI2	263
pari_str	270	PiI2n	263
pari_strdup	269	plotbox	379
pari_strndup	269	plotclip	379
pari_thread_alloc	385	plotcolor	379
pari_thread_close	385	plotcopy	379
pari_thread_free	385	plotcursor	379
pari_thread_init	385	plotdraw	379
pari_thread_start	385	ploth	379
pari_thread_valloc	385	ploth0	290
pari_timer	39	plotlexport0	291
pari_TRY	46	plotdraw	379
pari_unique_dir	274	plotsizes	379
pari_unique_filename	274	plotinit	379
pari_unique_filename_suffix	274	plotkill	379
pari_unlink	272	plotline	379
pari_var_close	74	plotlines	379
pari_var_create	74	plotlinetype	380
pari_var_init	74	plotmove	380
pari_var_next	74	plotpoints	380
pari_var_next_temp	74	plotpointsize	380
PARI_VERSION	80	plotpointtype	380
pari_version	80	plotrbox	380
PARI_VERSION_SHIFT	80	plotrecth	379
pari_vfprintf	37	plotrecthraw	380
pari_vprintf	37	plotrline	380
pari_vsprintf	37	plotrmove	380
pari_warn	38	plotrpoint	380
path_expand	273	plotscale	380
perm_commute	263	plotstring	380
perm_conj	263	point_to_a4a6	355
perm_cycles	264	point_to_a4a6_Fl	355
perm_inv	263	pol0_F2x	154
perm_mul	263	pol0_Flx	142
perm_order	264	pol1_F2x	155
perm_orderu	264	pol1_F2xX	157
perm_pow	263	pol1_Flx	142
perm_powu	263	pol1_FlxX	148
perm_sign	264	polclass	366

polcoef_i	251	powru	100
poldivrem	240	powruhalf	100
poleval	209, 246	powuu	100
polgalois	267	ppg	178
polhensellift	167, 168	ppi	178
polintspec	191	pple	178
polint_i	191	ppo	178
pollegendre_reduced	252	prec2lg	60
polmod	30	prec2nbits	60
polmodular	366	prec2nbits_mul	60
polmodular_ZM	367	prec2ndec	60
polmodular_ZXX	366	precdbl	59
polmod_nffix	331	precision	61
polmod_nffix2	331	precision0	224
polmod_to_embed	252	precp	29, 60
Polred	333	PRECPBITS	65
polred0	333	PRECPSHIFT	65
polredabs	333	preferences file	78
polredabs2	333	<i>prid</i>	316
polredabsall	333	prime	180
Polrev	227	primeform	337
polxn_Flx	142	primeform_u	337
polx_F2x	155	primepi_lower_bound	180
polx_F2xX	157	primepi_upper_bound	180
polx_Flx	142	primes	180
polx_FlxX	148	primes_interval	180
polx_zx	205	primes_interval_zv	180
polynomial	30	PRIMES_search	66
pol_0	227	primes_upto_zv	181
pol_1	227	primes_zv	180
pol_x	227	prime_fact	228
pol_xn	227	prime_over_p_full	316
pol_xnall	227	primitive root	83
pol_x_powers	227	primitive_part	241
pop_lex	77, 291	primpart	241
pow2Pis	260	printf	36, 37, 75
powcx	260	print_fun_list	57
powcx_prec	260	prodeuler0	291
power series	31	prodinf0	291
powersr	100	prodprimes	66
powgi	244	prodprimeslim	66
powii	100	prototype code	78
powis	100	prV_lcm_capZ	317
powIs	101	prV_primes	317
powiu	100	pr_anti_uniformizer	317
powPis	260	pr_basis_perm	319
powrfrac	100	pr_equal	320
powrs	100	pr_get_e	316
powrshalf	100	pr_get_f	316

pr_get_gen	316
pr_get_p	316
pr_get_tau	316
pr_hnf	316
pr_inv	316
pr_inv_p	316
pr_is_inert	316
pr_norm	316
pr_uniformizer	317
psdraw	380
psilseries	252
psploth	380
psplothraw	380
pthread_join	385
push_lex	77, 291
putch	271
puts	271
p_to_FF	253
p_to_FF(p,0)	253

Q

QabM_tracerel	311
QabV_tracerel	311
Qab_tracerel	311
Qab_trace_init	311
Qdivii	241
Qdivis	241
Qdiviu	241
Qevproj_apply	187
Qevproj_apply_vecei	187
Qevproj_down	187
Qevproj_init	187
qfbcomp	336
qfbcompraw	336
qfbcompraw_i	336
qfbcomp_i	336
qfbforms	251
qfbpow	336
qfbpowraw	336
qfbpows	336
qfbpow_i	336
qfbred	335
qfbred_i	335
qfbsolve	336
qfbsqr	336
qfbsqr_i	336
qfb_1	335
qfb_disc	251

qfb_disc3	251
qfb_equal1	335
qfb_ZM_apply	251
qfeval	246
qfevalb	246
qfiseven	192
qfisolvep	336
qfi_log	336
qfi_order	336
qfi_red	335
qfi_Shanks	336
qflll10	343
qflllgram0	343
qfr3	337
qfr3_comp	337
qfr3_compraw	337
qfr3_pow	338
qfr3_red	338
qfr3_rho	338
qfr3_to_qfr	338
qfr5	337
qfr5_comp	338
qfr5_compraw	338
qfr5_dist	338
qfr5_pow	338
qfr5_red	338
qfr5_rho	338
qfr5_to_qfr	338
qfrsolvep	337
qfr_boundcf	336
qfr_cf	335
qfr_data_init	337
qfr_red	335
qfr_to_qfr5	338
qf_RgM_apply	246
qf_ZM_apply	246
QM_charpoly_ZX	186, 187
QM_charpoly_ZX_bound	186
QM_det	189
QM_gauss	187
QM_gauss_i	187
QM_image	187
QM_image_shallow	187
QM_ImQ	342
QM_ImQ_all	342
QM_ImQ_hnf	342
QM_ImQ_hnfall	342
QM_ImZ	342
QM_ImZ_all	342

QM_ImZ_hnf	342	QXQV_to_mod	161
QM_ImZ_hnfall	342	QXQXV_to_mod	161
QM_indexrank	187	QXQX_gcd	205
QM_inv	187	QXQX_homogenous_evalpow	205
QM_ishnf	187	QXQX_mul	205
QM_ker	189	QXQX_powers	205
QM_minors_coprime	342	QXQX_QXQ_mul	205
QM_mul	189	QXQX_sqr	205
QM_QC_mul	189	QXQX_to_mod_shallow	161
QM_rank	187	QXQ_charpoly	204
QM_sqr	189	QXQ_div	204
QpV_to_QV	170	QXQ_intnorm	204
Qp_agm2_sequence	262	QXQ_inv	204
Qp_ascending_Landen	262	QXQ_mul	204
Qp_descending_Landen	262	QXQ_mul(A,QXQ_inv(B,T),T)	204
Qp_exp	261	QXQ_norm	204
Qp_exp_prec	261	QXQ_powers	204
Qp_gamma	261	QXQ_reverse	204
Qp_log	262	QXQ_sqr	204
Qp_psi	262	QXQ_to_mod_shallow	161
Qp_sqrt	262	QXV_QXQ_eval	205
Qp_sqrtn	262	QXY_QXQ_evalx	205
Qp_zeta	261	QX_complex_roots	204, 252
Qp_zetahurwitz	261	QX_disc	204
QR_init	194	QX_factor	204
Qtoss	228	QX_gcd	204
quadclassno	334	QX_mul	203
quadclassnoF	334	QX_resultant	204
quadclassnoF_fact	335	QX_sqr	203
quadclassnos	334	QX_ZXQV_eval	205
quadnorm	251	QX_ZX_rem	204
quadpoly	29	Q_abs	241
quadpoly_i	230	Q_abs_shallow	241
quadratic number	29	Q_content	242
quadratic_prec_mask	170	Q_content_safe	242
quadtofp	223	Q_denom	242
quad_disc	251	Q_denom_safe	242
<i>quasi-normalized character</i>	346	Q_div_to_int	242
quotient_group	266	Q_factor	178
quotient_groupelts	266	Q_factor_limit	178
quotient_perm	266	Q_gcd	241
quotient_subgroup_lift	266	Q_lval	241
QV_isscalar	192	Q_lvalrem	241
QXk_gcd	203	Q_muli_to_int	242
QXQC_to_mod_shallow	161	Q_mul_to_int	242
QXQM_mul	205	Q_primitive_part	242
QXQM_sqr	205	Q_primpart	242
QXQM_to_mod_shallow	161	Q_pval	241
QXQV_to_FpM	319	Q_pvalrem	241

Q_remove_denom	242, 307
Q_to_famat	312

R

radicalu	106
random	102
randomi	102
randomr	102
random_bits	102
random_F2x	155
random_F2xqE	363
random_F2xqX	158
random_Fl	83, 102
random_Fle	362
random_Fle_pre	362
random_Flj_pre	362
random_Flv	116
random_Flx	143
random_FlxqE	364
random_FlxqX	150
random_FpC	113
random_FpE	360
random_FpV	113
random_FpX	127
random_FpXQE	366
random_FpXQX	134
random_Fq	129
random_zv	187
rational function	31
rational number	29
raw	269
rcopy	87
rdivii	98
rdiviiz	98
rdivii_lg	98
rdivis	98
rdivsi	98
rdivss	98
read	35, 36
readseq	36
real number	28
real2n	86
realprec	28, 59
real_0	86
real_0_expo	86
real_1	86
real_i	249
real_m1	86

real_m2n	86
rect2ps	380
rect2ps_i	380
rect2svg	381
reducemodinvertible	345
reducemodlll	346
remi2n	97, 198
remlll_pre	84
remll_pre	84
remsBIL	65
residual_characteristic	251
<i>resultant (reduced)</i>	167
resultant	240, 251
resultant2	252
retconst_col	232
retconst_vec	231
retgc_const	69
retmkcol	232
retmkcol2	232
retmkcol3	232
retmkcol4	232
retmkcol5	232
retmkcol6	232
retmkcomplex	232
retmkfrac	232
retmkintmod	232
retmkmat	232
retmkmat2	232
retmkmat22	232
retmkmat3	232
retmkmat4	232
retmkmat5	232
retmkpadic	232
retmkpadic_i	232
retmkpolmod	232
retmkqfb	232
retmkquad	232
retmkfrac	232
retmkvec	231
retmkvec2	231
retmkvec3	231
retmkvec4	231
retmkvec5	231
retmkvec6	231
rfracrecip	207
rfracrecip_to_ser_absolute	226
rfrac_deflate	207
rfrac_deflate_max	207
rfrac_deflate_order	207

rfrac_to_ser	226	RgM_inv_upper	193
rfrac_to_ser_i	226	RgM_isdiagonal	192
RgC_add	189	RgM_isidentity	192
RgC_fpnorml2	193	RgM_isscalar	192
RgC_gtofp	192	RgM_is_FFM	254
RgC_gtomp	192	RgM_is_FpM	112
RgC_is_ei	192	RgM_is_QM	192
RgC_is_FFC	254	RgM_is_ZM	192
RgC_neg	189	RgM_minor	284
RgC_RgM_mul	190	RgM_mul	190
RgC_RgV_mul	190	RgM_mulreal	190
RgC_RgV_mulrealsym	190	RgM_multosym	190
RgC_Rg_add	189	RgM_neg	189
RgC_Rg_div	190	RgM_powers	190
RgC_Rg_mul	190	RgM_QR_init	194
RgC_Rg_sub	189	RgM_rescale_to_int	184
RgC_sub	189	RgM_RgC_invmage	193
RgC_to_FpC	113	RgM_RgC_mul	190
RgC_to_FqC	115	RgM_RgC_type	112
RgC_to_nfC	309	RgM_RgV_mul	190
RgE_is_FpE	361	RgM_RgX_mul	190
RgE_is_FpXQE	366	RgM_Rg_add	189
RgE_to_F2xqE	364	RgM_Rg_add_shallow	189
RgE_to_FlxqE	365	RgM_Rg_div	190
RgE_to_FpE	361	RgM_Rg_mul	190
RgE_to_FpXQE	366	RgM_Rg_sub	189
RgMrow_RgC_mul	190	RgM_Rg_sub_shallow	189
RgMrow_zc_mul	173	RgM_shallowcopy	283
RgMs_structelim	196	RgM_solve	193
RgM_add	189	RgM_solve_realimag	193
RgM_Babai	194	RgM_sqr	190
RgM_check_ZM	184	RgM_sub	189
RgM_Cholesky	194	RgM_sumcol	191
RgM_det_triangular	193	RgM_to_F2m	120
RgM_diagonal	192	RgM_to_F3m	122
RgM_diagonal_shallow	192	RgM_to_Flm	172
RgM_dimensions	189	RgM_to_FpM	113
RgM_div	190	RgM_to_FqM	115
RgM_flatten_RgC	284	RgM_to_nfM	309
RgM_flatten_RgV	284	RgM_to_RgXV	226
RgM_fpnorml2	193, 245	RgM_to_RgXV_reverse	226
RgM_Fp_init	113	RgM_to_RgXX	226
RgM_gram_schmidt	194	RgM_transmul	190
RgM_gtofp	193	RgM_transmultosym	190
RgM_gtomp	193, 194	RgM_type	112
RgM_Hadamard	193	RgM_type2	112
RgM_hnfall	343	RgM_zc_mul	173
RgM_inv	193	RgM_zm_mul	173
RgM_invmage	193	RgM_ZM_mul	190

RgV_add	189	RgXn_inv_i	212
RgV_check_ZV	182	RgXn_mul	212
RgV_dotproduct	191	RgXn_powers	212
RgV_dotsquare	191	RgXn_powu	212
RgV_F2v_extract_shallow	284	RgXn_powu_i	212
RgV_gtofp	192	RgXn_recip_shallow	212
RgV_isin	192	RgXn_red_shallow	212
RgV_isin_i	192	RgXn_reverse	213
RgV_isscalar	192	RgXn_sqr	212
RgV_is_arithprog	182	RgXn_sqrt	212
RgV_is_FpV	112	RgXQC_red	214
RgV_is_prV	300	RgXQM_mul	214
RgV_is_QV	182	RgXQM_red	214
RgV_is_ZMV	188	RgXQV_factorback	214
RgV_is_ZV	182	RgXQV_red	214
RgV_is_ZVnon0	182	RgXQV_RgXQ_mul	214
RgV_is_ZVpos	182	RgXQX_div	214
RgV_kill0	191	RgXQX_divrem	214
RgV_neg	189	RgXQX_mul	214
RgV_nffix	331	RgXQX_powers	214
RgV_polint	191	RgXQX_pseudodivrem	210
RgV_prod	190	RgXQX_pseudorem	210
RgV_RgC_mul	190	RgXQX_red	214
RgV_RgM_mul	190	RgXQX_rem	214
RgV_Rg_mul	190	RgXQX_RgXQ_mul	214
RgV_sub	189	RgXQX_RgXQ_translate	214
RgV_sum	190	RgXQX_sqr	214
RgV_sumpart	190	RgXQ_charpoly	213
RgV_sumpart2	191	RgXQ_inv	213
RgV_to_F2v	120	RgXQ_matrix_pow	213
RgV_to_F3v	121	RgXQ_minpoly	213
RgV_to_Flv	172	RgXQ_mul	213
RgV_to_FpV	113	RgXQ_norm	213
RgV_to_RgM	226	RgXQ_pow	213
RgV_to_RgX	225	RgXQ_powers	213
RgV_to_RgX_reverse	225	RgXQ_powu	213
RgV_to_ser	226	RgXQ_ratlift	213
RgV_to_str	269, 270	RgXQ_reverse	214
RgV_type	112	RgXQ_sqr	213
RgV_type2	112	RgXQ_trace	213
RgV_zc_mul	173	RgXV_maxdegree	208
RgV_zm_mul	173	RgXV_prod	209
RgXnV_red_shallow	213	RgXV_rescale	211
RgXn_div	212	RgXV_RgV_eval	209
RgXn_div_i	212	RgXV_to_FlxV	171
RgXn_eval	213	RgXV_to_RgM	226
RgXn_exp	213	RgXV_unscale	211
RgXn_expint	213	RgXX_to_Kronecker	134, 203, 207, 208
RgXn_inv	212	RgXX_to_Kronecker_spec	208

RgXX_to_Kronecker_var	208	RgX_homogenous_derivn	212
RgXX_to_RgM	226	RgX_homogenous_eval	212
RgXY_cxevalx	246	RgX_homogenous_evalpow	212
RgXY_degreex	226	RgX_inflate	207
RgXY_derivx	226	RgX_integ	211
RgXY_swap	226	RgX_isscalar	206
RgXY_swapspec	226	RgX_is_FpX	123
RgX_act_Gl2Q	212	RgX_is_FpXQX	128
RgX_act_ZGl2Q	212	RgX_is_monomial	206
RgX_add	208	RgX_is_QX	206
RgX_addmulXn	210	RgX_is_rational	206
RgX_addmulXn_shallow	210	RgX_is_ZX	206
RgX_addspec	210	RgX_is_ZXX	206
RgX_addspec_shallow	210	RgX_mul	209
RgX_affine	211	RgX_mul2n	209
RgX_blocks	143, 149, 206	RgX_mulhigh_i	212
RgX_check_QX	203	RgX_muls	209
RgX_check_ZX	197	RgX_mulspec	210
RgX_check_ZXX	203	RgX_mulXn	210
RgX_chinese_coprime	211	RgX_mul_i	209
RgX_coeff	206	RgX_mul_normalized	209
RgX_copy	206	RgX_neg	208
RgX_cxeval	246	RgX_nffix	331
RgX_deflate	207	RgX_normalize	209
RgX_deflate_max	207	RgX_pseudodivrem	210
RgX_deflate_order	207	RgX_pseudorem	210
RgX_degree	206	RgX_recip	207
RgX_deriv	211	RgX_recip_i	207
RgX_derivn	211	RgX_recip_shallow	207
RgX_digits	210	RgX_rem	210
RgX_disc	211	RgX_renormalize	206
RgX_div	210	RgX_renormalize_lg	206
RgX_divrem	209	RgX_rescale	211
RgX_divs	209	RgX_rescale_to_int	207
RgX_div_by_X_x	210	RgX_resultant_all	211
RgX_equal	206	RgX_RgMV_eval	246
RgX_equal_var	206	RgX_RgM_eval	246
RgX_even_odd	155, 206	RgX_RgV_eval	209
RgX_extgcd	211	RgX_RgXnV_eval	213
RgX_extgcd_simple	211	RgX_RgXn_eval	213
RgX_fpnorml2	211	RgX_RgXQV_eval	213
RgX_gcd	210, 211	RgX_RgXQ_eval	213
RgX_gcd_simple	211	RgX_Rg_add	208
RgX_Graeffe	207	RgX_Rg_add_shallow	208
RgX_gtofp	211	RgX_Rg_div	209
RgX_halfgcd	211	RgX_Rg_divexact	209
RgX_halfgcd_all	211	RgX_Rg_eval_bk	209
RgX_homogenize	207	RgX_Rg_mul	208
RgX_homogenize_deg	207	RgX_Rg_sub	208

RgX_Rg_translate	211	Rg_to_Fl	171
RgX_Rg_type	112	Rg_to_Flxq	171
RgX_rotate_shallow	208	Rg_to_Fp	112, 113
RgX_shift	155, 208	Rg_to_FpXk	133
RgX_shift_inplace	208	Rg_to_FpXQ	128
RgX_shift_inplace_init	208	Rg_to_Fq	128
RgX_shift_shallow	208	Rg_to_RgC	226
RgX_splitting	143, 206	Rg_type	110
RgX_sqr	209	RM_round_maxrank	301, 325, 344
RgX_sqrhigh_i	212	rnfabelianconjgen	331
RgX_sqrspec	210	rnfcomplete	306
RgX_sqr_i	209	rnfdisc_factored	331
RgX_sturmpart	252	rnfeltabstorel	330
RgX_sub	208	rnfeltreltoabs	330
RgX_sylvestermatrix	233	rnfeltup	330
RgX_to_F2x	171	rnfequationall	330
RgX_to_Flx	171	rnfnewprec	307
RgX_to_FlxqX	171	rnfnewprec_shallow	307
RgX_to_FpX	123	rnf_build_nfabs	306
RgX_to_FpXQX	128	rnf_COND	329
RgX_to_FqX	128	rnf_get_absdegree	303
RgX_to_nfX	308	rnf_get_alpha	304
RgX_to_RgC	225	rnf_get_degree	303
RgX_to_RgV	226	rnf_get_disc	304
RgX_to_ser	226	rnf_get_idealdisc	304
RgX_to_ser_inexact	226	rnf_get_index	304
RgX_type	112	rnf_get_invzk	304
RgX_type2	112	rnf_get_k	304
RgX_type3	112	rnf_get_map	304, 330
RgX_type_code	111	rnf_get_nf	303
RgX_type_decode	111	rnf_get_nfdegree	303
RgX_type_is_composite	111	rnf_get_nfpol	303
RgX_unscale	211	rnf_get_nfvarn	303
RgX_val	208	rnf_get_nfzk	303, 330
RgX_valrem	208	rnf_get_pol	303
RgX_valrem_inexact	208	rnf_get_polabs	304, 306
Rg_col_ei	228	rnf_get_ramified_primes	304
Rg_get_0	31, 111	rnf_get_varn	303
Rg_get_1	111	rnf_get_zk	304
Rg_is_FF	254	rnf_REL	329
Rg_is_Fp	112	rnf_zkabs	306
Rg_is_FpXk	133	rootmod	161
Rg_is_FpXQ	128	rootmod0	161
Rg_is_QXk	203	rootmod2	161
Rg_nffix	331	rootsof1pow	261
Rg_RgC_sub	190	rootsof1powinit	261
Rg_RgX_sub	208	rootsof1q_cx	261
Rg_to_F2	171	rootsof1u_cx	260
Rg_to_F2xq	171	rootsof1u_Fp	105

rootsof1_cx	260	sd_echo	293
rootsof1_Fl	105	sd_factorlimit	293
rootsof1_Fp	105	sd_factor_add_primes	293
roots_from_deg1	233	sd_factor_proven	293
roots_to_pol	233	sd_format	293
roots_to_pol_r1	233	sd_graphcolormap	294
roundr	89	sd_graphcolors	294
roundr_safe	89	sd_help	294
row vector	31	sd_histfile	294
row	284	sd_histsize	294
rowcopy	284	sd_intarray	295
rowpermute	284	sd_lines	294
rowslice	284	sd_linewrap	294
rowslicepermute	284	sd_log	294
rowsplice	284	sd_logfile	294
row_i	284	sd_nbthreads	294
row_Q_primpart	242	sd_new_galois_format	294
rtodbl	25, 223	sd_output	294
rtor	88	sd_parisize	294
rtor_lg	88	sd_parisizemax	294
R_abs	249	sd_path	294
R_abs_shallow	249	sd_plotsizes	294
R_from_QR	194	sd_prettyprinter	294
S			
scalarcol	228	sd_primelimit	294
scalarcol_shallow	233	sd_prompt	294
scalarmat	228	sd_prompt_cont	294
scalarmat_s	228	sd_psfile	294
scalarmat_shallow	233	sd_readline	294
scalarpol	227	sd_realbitprecision	294
scalarpol_shallow	233	sd_realprecision	294
scalarser	227	sd_recover	294
scalar_Flm	116	sd_secure	294
scalar_ZX	197	sd_seriesprecision	294
scalar_ZX_shallow	197	sd_simplify	294
sdivsi	99	sd_sopath	294
sdivsi_rem	99	sd_strictargs	294
sdivss_rem	98	sd_strictmatch	294
sdomain_isincl	370	sd_string	296
sd_breakloop	293	sd_TeXstyle	293
sd_colors	293	sd_threadsize	294
sd_compatible	293	sd_threadsizemax	295
sd_datadir	293	sd_timer	294
sd_debug	293	sd_toggle	295
sd_debugfiles	293	sd_ulong	295
sd_debugmem	293	secure	55
sd_doctest	293	serchop0	227
		serchop_i	227
		sertoser	252
		ser_inv	252

ser_isexactzero	252	smallpolred	333
ser_normalize	252	smallpolred2	333
ser_unscale	252	SMALL_ULONG	85
setabssign	63	smith	342
setalldebug	39	smithall	342
setdefault	54, 55, 293	smithclean	343
setexpo	28, 31, 63	smodis	99
setisclone	26	smodsi	99
setlg	26, 62	smodss	99
setlgefint	27, 62	snm_closure	292
setprecp	29, 63	solvestep0	291
setrand	102	sort	237
setrealprecision	258	sort_factor	238
setsigne	27, 30, 31, 62	split_realimag	193
settyp	26, 62	sprintf	37
setunion_i	238	sqrfrac	249
setvalp	29, 63	sqri	93
setvalser	31, 63	sqrr	93
setvarn	24, 30, 31, 63, 233	sqrs	93
set_avma	68	sqrtdi	101
set_lex	291	sqrtnr	258
set_sign_mod_divisor	321	sqrtnr_abs	258
shallow	51	sqrtr	258
shallowconcat	283, 284	sqrtrmi	101
shallowconcat1	283	sqrtr_abs	259
shallowcopy	25, 283	sqrui	93
shallowextract	284	sstoQ	228
shallowmatconcat	284	stack	13, 17
shallowmatextract	284	stackdummy	70, 87
shallowtrans	283	stack_calloc	67
shiftaddress	68	stack_calloc_align	67
shiftaddress_canon	68	stack_lim	22
shifti	90	stack_malloc	67, 269
shiftdi	82	stack_malloc_align	67
shiftdr	82	stack_sprintf	269
shiftr	90	stack_strcat	269
shiftr_inplace	90	stack_strdup	269
shift_left	90	stderr	271
shift_right	90	stdout	36, 271
SIGNBITS	65	stoi	25, 88
signe	27, 30, 31, 60	stor	88
SIGNnumBITS	65	stor_lg	88
SIGNSHIFT	65	Str	270
simplefactmod	161	strexpand	270
simplify	72	strftime_expand	273
simplify_shallow	72	string_context	77
sisfundamental	107	strntoGENstr	269
sizedigit	61	strprintf	37
SL2_inv_shallow	186	strtex	270

vecsmall_lexcmp	287
vecsmall_max	286
vecsmall_min	287
vecsmall_pack	287
vecsmall_prefixcmp	287
vecsmall_prepend	287
vecsmall_prod	287
vecsmall_reverse	286
vecsmall_shorten	286
vecsmall_sort	286
vecsmall_to_col	286
vecsmall_to_vec	286
vecsmall_to_vec_inplace	286
vecsmall_uniq	287
vecsmall_uniq_sorted	287
vecsort	237
vecssplice	284
vecsqr	285
vecsquarefreeu	177
vec trunc_append	58
vec trunc_append_batch	58
vec trunc_init	58
vecvecsmall_indexsort	288
vecvecsmall_max	288
vecvecsmall_search	288
vecvecsmall_sort	287
vecvecsmall_sort_inplace	287
vecvecsmall_sort_shallow	287
vecvecsmall_sort_uniq	288
veczeta	260
vec_append	286
vec_ei	228
vec_equiv	286
vec_insert	286
vec_is1to1	286
vec_isconst	285
vec_lengthen	286
vec_prepend	286
vec_Q_primpart	242
vec_reduce	286
vec_setconst	285
vec_shorten	286
vec_to_vecsmall	286
Vgaeasytheta	371

W

walltimer_delay	39
walltimer_get	39

walltimer_start	39
warner	38
warnfile	38
warnmem	38
warnprec	38
whilepari	291
writebin	36, 272

Z

Z2_sqrt	166
ZabM_indexrank	310
ZabM_inv	310
ZabM_inv_ratlift	311
ZabM_ker	310
ZabM_pseudoinv	311
zbrent0	291
zCs_to_ZC	196
ZC_add	183
ZC_copy	183
ZC_divexactu	183
ZC_hnfdivrem	345
ZC_hnfreem	345
ZC_is_ei	186
ZC_lincomb	183
ZC_lincomb1_inplace	183
ZC_lincomb1_inplace_i	183
ZC_modhnf	340
ZC_neg	183
ZC_nfval	319
ZC_nfvalrem	320
ZC_prdvd	320
ZC_Q_mul	185
ZC_reducemod111	345
ZC_reducemodmatrix	345
ZC_sub	183
zc_to_ZC	173
ZC_union_shallow	184
ZC_ZV_mul	183
ZC_Z_add	183
ZC_Z_div	183
ZC_Z_divexact	183
ZC_z_mul	173
ZC_Z_mul	183
ZC_Z_sub	183
zerocol	228
zeromat	228
zeromatcopy	228
zeropadic	227

zeropadic_shallow	233	ZlM_gauss	170
zeropol	228	zlxX_translate1	149
zeroser	227	zlx_translate1	141
zerovec	228	ZM2_mul	184
zerovec_block	228	ZM2_sqr	185
zero_F2m	120	ZMrow_equal0	185
zero_F2m_copy	120	ZMrow_ZC_mul	185
zero_F2v	120	zMs_compress	196
zero_F2x	155	zMs_to_ZM	196
zero_F3m_copy	121	zMs_ZC_mul	196
zero_F3v	121	ZMV_prod	185
zero_Flm	118	ZMV_producttree	163
zero_Flm_copy	118	ZMV_to_FlmV	189
zero_Flv	118	ZMV_to_zmV	189
zero_Flx	142	zmV_to_ZMV	189
zero_FlxC	145	ZM_add	184
zero_FlxM	145	ZM_charpoly	186
zero_FlxM_copy	145	ZM_copy	184
zero_zm	188	zm_copy	188
zero_zv	188	ZM_det	185
zero_zx	205	ZM_detmult	185
ZGCs_add	195	ZM_det_triangular	186
ZGC_G_mul	195	ZM_diag_mul	185
ZGC_G_mul_inplace	195	ZM_divexactu	185
ZGC_Z_mul	195	ZM_equal	184
ZG_add	195	ZM_equal0	184
ZG_G_mul	195	ZM_famat_limit	312
ZG_mul	195	ZM_gauss	186
ZG_neg	195	ZM_hnf	338, 342
ZG_normalize	195	ZM_hnfall	338, 339, 340, 342, 343
ZG_sub	195	ZM_hnfall_i	339
ZG_Z_mul	195	ZM_hnfcenter	340
Zideallog	348	ZM_hnfdivrem	345
zidealstar	332	ZM_hnfintersectmod	338
zidealstarinit	332	ZM_hnflll	340
zidealstarinitgen	332	ZM_hnfmod	338, 342
zkchinese	315	ZM_hnfmodall	339
zkchinese1	315	ZM_hnfmodall_i	339
zkchineseinit	315	ZM_hnfmodid	338, 339, 342
zkC_multable_mul	309	ZM_hnfmodprime	338
zkmodprinit	319	ZM_hnfperm	339
zkmultable_capZ	310	ZM_hnfrem	345
zkmultable_inv	310	ZM_hnf_knapsack	339
zk_inv	309, 310	ZM_imagecompl	186
zk_multable	309, 314	ZM_incremental_CRT	162
zk_scalar_or_multable	310, 316	ZM_indeximage	186
zk_to_Fq	319	ZM_indexrank	186
zk_to_Fq_init	319	ZM_init_CRT	162
zlm_echelon	169	ZM_inv	186

ZM_inv_ratlift	186	ZM_zc_mul	173
ZM_isdiagonal	186	ZM_ZC_mul	185
ZM_ishnf	186	zm_zc_mul	187
ZM_isidentity	186	ZM_zm_mul	173
ZM_isscalar	186	ZM_ZV_mod	185
ZM_ker	186	ZM_ZX_mul	185
ZM_lll	343, 344	ZM_Z_div	185
ZM_lll_norms	344	ZM_Z_divexact	185
ZM_max_expi	186	ZM_Z_mul	185
ZM_max_lg	186	zncharcheck	347
ZM_merge_factor	238	zncharconj	347
ZM_modhnf	340	znchardiv	347
ZM_mul	184	znchareval	347
zm_mul	188	zncharker	347
ZM_multosym	185	zncharm mul	347
ZM_mul_diag	185	zncharorder	347
ZM_neg	184	zncharpow	347
ZM_nm_mul	173	znchar_quad	348
ZM_nv_mod_tree	163	znconreyfromchar	347
ZM_permanent	185	znconreyfromchar_normalized	347
zm_permanent	188	znconreylog_normalize	348
ZM_pow	185	znconrey_check	347
ZM_powu	185	znconrey_normalized	347
ZM_pseudoinv	186	znstar_get_conreycyc	306
ZM_Q_mul	185	znstar_get_conreygen	306
ZM_rank	186	znstar_get_cyc	305
ZM_reducemodlll	345	znstar_get_faN	305
ZM_reducemodmatrix	345	znstar_get_gen	305
zm_row	188	znstar_get_N	305
ZM_snf	340	znstar_get_no	305
ZM_snfall	340	znstar_get_pe	305
ZM_snfall_i	340	znstar_get_U	306
ZM_snfclean	341	znstar_get_Ui	306
ZM_snf_group	341	Zn_ispower	105
ZM_sqr	184	Zn_issquare	105
ZM_sub	184	Zn_quad_roots	105
ZM_supnorm	186, 245	Zn_sqrt	104
ZM_togglesign	184	Zn_sqrtall	104
ZM_to_F2m	120	ZpMs_ZpCs_solve	197
ZM_to_F3m	122	ZpM_echelon	169
ZM_to_Flm	172	ZpM_invlift	166
zm_to_Flm	173	ZpM_ZpC_gauss	166
ZM_to_zm	172	ZpXQM_prodFrobenius	168
zm_to_ZM	173	ZpXQXQ_div	169
zm_to_zxV	174	ZpXQXQ_inv	169
ZM_trace	186	ZpXQXQ_invlift	169
ZM_transmul	185	ZpXQX_digits	169
ZM_transmultosym	185	ZpXQX_divrem	169
zm_transpose	188	ZpXQX_liftfact	168

ZpXQX_liftroot	168, 169	ZV_chinese	164
ZpXQX_liftroots	169	ZV_chinesetree	165
ZpXQX_liftroot_vald	169	ZV_chinese_center	164
ZpXQX_roots	169	ZV_chinese_tree	165
ZpXQX_ZpXQXQ_liftroot	169	ZV_cmp	182, 239
ZpXQ_div	168	zv_cmp0	188
ZpXQ_inv	168	ZV_content	183
ZpXQ_invlift	168	zv_content	188
ZpXQ_log	168	zv_copy	188
ZpXQ_sqrt	168	zv_cyc_minimal	347
ZpXQ_sqrtnlift	168	zv_cyc_minimize	347
ZpX_disc_val	167	zv_diagonal	188
ZpX_Frobenius	167	ZV_dotproduct	183
ZpX_gcd	167	zv_dotproduct	188
ZpX_liftfact	167	ZV_dotsquare	183
ZpX_liftroot	166, 168	ZV_dvd	184
ZpX_liftroots	167	ZV_equal	182
ZpX_monic_factor	167	zv_equal	188
ZpX_primedec	167	ZV_equal0	182
ZpX_reduced_resultant	167	zv_equal0	188
ZpX_reduced_resultant_fast	167	ZV_extgcd	102, 183
ZpX_resultant_val	167	ZV_indexsort	184
ZpX_roots	166	ZV_isscalar	192
ZpX_ZpXQ_liftroot	168	ZV_lcm	102
ZpX_ZpXQ_liftroot_ea	168	ZV_lval	92
Zp_div	166	ZV_lvalrem	92
Zp_exp	166	ZV_max_expi	184
Zp_inv	166	ZV_max_lg	184
Zp_invlift	166	zv_neg	187
Zp_issquare	105	ZV_neg_inplace	183
Zp_log	166	zv_neg_inplace	187
Zp_sqrt	166	ZV_nv_mod_tree	163
Zp_sqrtnlift	166	ZV_prod	184
Zp_sqrtnlift	166	zv_prod	188
Zp_teichmuller	166	ZV_producttree	163, 165
ZqX_liftfact	169	zv_prod_Z	188
ZqX_liftroot	169	ZV_pval	92
ZqX_roots	169	ZV_pvalrem	92
ZqX_ZqXQ_liftroot	169	ZV_search	184
Zq_sqrtnlift	168	zv_search	188
zvV_equal	188	ZV_snfall	340
ZVV_nv_mod_tree	163	ZV_snfclean	341
ZVV_to_FlvV	172	ZV_snf_gcd	102, 341
zv_abs	187	ZV_snf_group	341
ZV_abcscmp	182	ZV_snf_rank	342
ZV_allpnqn	102	zv_snf_rank	342
zv_canon_inplace	187	ZV_snf_rank_u	342
ZV_cba	178	ZV_snf_trunc	341
ZV_cba_extend	178	ZV_sort	184

ZV_sort_inplace	184	ZXV_dotproduct	201
ZV_sort_shallow	184	ZXV_equal	201
ZV_sort_uniq	184	ZXV_pval	92
ZV_sort_uniq_shallow	184	ZXV_pvalrem	92
ZV_sum	184	ZXV_remi2n	201
zv_sum	188	ZXV_to_FlxV	171
zv_sumpart	188	ZXV_ZX_fromdigits	199
ZV_togglesign	183	ZXV_Z_mul	201
ZV_to_F2v	120	ZXXT_to_FlxXT	172
ZV_to_F3v	121	ZXXV_to_FlxXV	172
ZV_to_Flv	171	ZXX_evalx0	203
zv_to_Flv	173	ZXX_max_lg	203
ZV_to_nv	172	ZXX_mul_Kronecker	203
ZV_to_zv	172	ZXX_nv_mod_tree	164
zv_to_ZV	173	ZXX_pval	92
zv_to_zx	173	ZXX_pvalrem	92
ZV_union_shallow	184	ZXX_Q_mul	203
ZV_zc_mul	173	ZXX_renormalize	203
ZV_zMs_mul	196	ZXX_resultant	201
zv_ZM_mul	173	ZXX_shifti	203
ZV_ZM_mul	185	ZXX_sqr_Kronecker	203
ZV_ZV_mod	183	ZXX_to_F2xX	157
ZV_Z_dvd	92	ZXX_to_FlxX	171
zv_z_mul	188	zxX_to_FlxX	172
ZXC_nv_mod_tree	163	zxX_to_Kronecker	150
ZXk_divexact	203	ZXX_Z_add_shallow	203
ZXk_divides	203	ZXX_Z_divexact	203
ZXk_gcd	203	ZXX_Z_mul	203
ZXM_incremental_CRT	163	ZX_add	198
ZXM_init_CRT	163	ZX_affine	200
ZXM_nv_mod_tree	164	ZX_composedsum	201
ZXn_mul	202	ZX_compositum	201
ZXn_sqr	202	ZX_content	199
ZXQM_mul	202	ZX_copy	197
ZXQM_sqr	202	ZX_deflate_max	199
ZXQXQ_minpoly	202	ZX_deflate_order	199
ZXQX_dvd	210	ZX_deriv	200
ZXQX_gcd	202	ZX_digits	199
ZXQX_mul	202	ZX_disc	200
ZXQX_sqr	202	ZX_divuexact	198
ZXQX_ZXQ_mul	202	ZX_div_by_X_1	198
ZXQ_charpoly	202	ZX_equal	198, 201
ZXQ_minpoly	202	ZX_equal1	198
ZXQ_mul	201	ZX_eval1	200
ZXQ_powers	202	ZX_factor	200
ZXQ_powu	202	ZX_gcd	199
ZXQ_sqr	201	ZX_gcd_all	199
ZXT_remi2n	201	ZX_graeffe	200
ZXT_to_FlxT	171	ZX_incremental_CRT	163

ZX_init_CRT	163	ZX_unscale_divpow	200
ZX_is_irred	200	ZX_Uspensky	200
ZX_is_monic	198	ZX_val	199
ZX_is_squarefree	200	ZX_valrem	199
ZX_lval	92	ZX_Zp_root	166
zx_lval	205	ZX_ZXY_resultant	201
ZX_lvalrem	92	ZX_ZXY_rnfequation	201
ZX_max_lg	197	ZX_Z_add	198
ZX_mod_Xnm1	198	ZX_Z_add_shallow	198
ZX_mul	198, 203	ZX_Z_divexact	198
ZX_mulspec	198	zx_z_divexact	205
ZX_mulu	198	ZX_Z_eval	200
ZX_neg	198	ZX_Z_mul	198
ZX_nf_roots	301	ZX_Z_normalize	199
ZX_nv_mod_tree	163	ZX_Z_sub	198
ZX_primitive_to_monic	199	ZX_Z_translate	199
ZX_pval	92	ZX_z_unscale	200
ZX_pvalrem	92	Z_cba	178
ZX_Q_mul	199	Z_chinese	161
ZX_Q_normalize	199, 322	Z_chinese_all	161
ZX_radical	199	Z_chinese_coprime	161
ZX_realroots	201	Z_chinese_post	162
ZX_realroots_irred	201	Z_chinese_pre	162
ZX_rem	198	Z_content	242
ZX_remi2n	198	Z_ECM	177
ZX_renormalize	198	Z_factor	175, 177, 178
zx_renormalize	205	Z_factor_limit	176, 178
ZX_rescale	199	Z_factor_listP	176
ZX_rescale2n	199	Z_factor_until	176
ZX_rescale_lt	199	Z_FF_div	255
ZX_resultant	200	Z_has_prime_3mod4	176
zx_shift	205	Z_incremental_CRT	162
ZX_shifti	198	Z_init_CRT	162
ZX_sqr	198, 203	Z_isanypower	175, 178
ZX_sqrspec	198	Z_isfundamental	180
ZX_squff	200	Z_ispow2	175
ZX_sturm	201	Z_ispower	175
ZX_sturmpart	201, 252	Z_ispowerall	175
ZX_sturm_irred	201	Z_issmooth	176
ZX_sub	198	Z_issmooth_fact	176
ZX_to_F2x	154	Z_issquare	174
ZX_to_Flx	171	Z_issquareall	175
zx_to_Flx	172	Z_issquarefree	180
ZX_to_monic	199	Z_issquarefree_fact	180
zx_to_zv	173	Z_lsmoothen	176
zx_to_ZX	172	Z_lval	91
ZX_unscale	200	z_lval	91
ZX_unscale2n	200	Z_lvalrem	91
ZX_unscale_div	200	z_lvalrem	91

Z_lvalrem_stop	91
Z_nv_mod	163
Z_pollardbrent	177
Z_ppgle	178
Z_ppio	178
Z_ppo	178
Z_pval	91
z_pval	91
Z_pvalrem	91
z_pvalrem	91
Z_smoother	176
Z_to_F2x	154
Z_to_famat	312
Z_to_Flx	173
Z_to_FpX	124
Z_to_perm	264
Z_ZC_sub	183
Z_ZV_mod	163
Z_ZV_mod_tree	163
Z_ZX_sub	198
.	
_evalexpo	62
_evallg	61
_evalprec	62
_evalvalp	62
_evalvalser	62