User's Guide

to

PARI-GP

C. Batut, D. Bernardi, H. Cohen, M. Olivier

last updated October 25, 1993
for version 1.38.1

Table of Contents

Chapter 1: Overview of the **PARI** System

1.1 Introduction.

The PARI system is a package which is capable of doing formal computations on recursive types at high speed; it is primarily aimed at number theorists, but can be used by people whose primary need is speed.

Although quite an amount of symbolic manipulation is possible in PARI, this system does very badly compared to much more sophisticated systems like Reduce, Macsyma, Maple, Mathematica or Scratchpad (now Axiom) on such manipulations (e.g. multivariate polynomials, formal integration, etc...). On the other hand, the two main advantages of the system is its speed (which can be between 5 and 100 times better on many computations than the above mentioned systems), and the possibility of using directly data types which are familiar to mathematicians.

It is possible to use PARI in two different ways:

1) as a library, which can be called from any upper-level language application (for instance written in C, C++, Pascal or Fortran);

2) as a sophisticated programmable calculator, named **GP**, which contains most of the standard control instructions of a standard language like C.

The use of GP is explained in chapters 2 and 3, and the programming in library mode is explained in chapters 3, 4 and 5. In the present chapter 1, we give an overview of the system.

Implementation notes.

Version 1.38.1 of the PARI package contains essentially four versions. The first one is a specific implementation for 68020/68030/68040/68060 based computers which contains a kernel (for the elementary arithmetic operations on multiprecise integers and real numbers, and binary/decimal conversion routines) entirely written in MC68020 assembly language (around 7000 lines), the rest being at present entirely written in ANSI C with a C++-compatible syntax (previous versions were written in K&R C, but this has now been abandoned). The system runs on SUN-3/xx, Sony News, NeXT cubes and on 680x0 based Macs with x$\geq$2. It should be very easy to port on any other 680x0 based machine like for instance the Apollo Domain workstations.

Note that the assembly language source code uses the SUN syntax, which for some strange reason differs from the Motorola standard used by most other 680x0 machines in the world. On the Mac disquettes, we have included a program which automatically converts from the SUN syntax into the standard one, at least for the needed PARI assembly file. On the Unix distribution, we have included other versions of the assembly file, using different syntaxes.

The second version is a specific implementation for SPARC version 7 based workstations, like the SPARCstation 1, 1+, 2 and the Solbourne 4000. This version contains only a few hundred lines of assembly code, and is usually slightly faster on the SPARCstation 1+ than on a SUN-3/60 or SUN-3/80. A variant of this version for SPARC version 8 (both Supersparc and Microsparc) is also included, see appendix A for more details.

The third version is a specific implementation for the Hewlett-Packard HP-PA architecture. Like the Sparc versions, this contains a few hundred lines of assembly code, and is quite fast.

Finally, a fourth version is written entirely in C, and should be portable with not too much trouble to any 32-bit computer having no real memory constraints. It is again at least 2 times slower than the SPARC version. This version has been tested for example on the DECstations 3100 and 5000. A 64-bit version is currently under development and should be available soon.

In addition to Unix workstations and Macs, Pari has been ported to a considerable number of smaller and larger machines, for example the VAX, 68000-based machines like the Atari, Mac Classic or Amiga 500, 68020 machines such as the Amiga 2500 or 3000, and even to MS-DOS 386 or better machines, using the `djgpp` port of the GNU C compiler and DOS-extender.

1.2 The PARI types.

The crucial word in PARI is recursivity: most types which are handled are recursive. For example, the basic type **Complex** exists. However, the components (i.e. the real and imaginary part) of such a "complex number" can be of any type. The only sensible ones are integers (we are then in $\mathbf{Z}[i]$), rational numbers ($\mathbf{Q}[i]$), real numbers ($\mathbf{R}[i] = \mathbf{C}$), or even elements of $\mathbf{Z}/n\mathbf{Z}$ ($(\mathbf{Z}/n\mathbf{Z})[i]$ when this makes sense), or p-adic numbers when $p \equiv 3 \bmod 4$ ($\mathbf{Q}_p[i]$).

This feature must of course not be used too rashly: for example you are in principle allowed to create objects which are "complex numbers of complex numbers", but don't expect PARI to make sensible use of such objects: you will mainly get nonsense.

On the other hand, one thing which *is* allowed is to have components of different, but compatible, types. For example, taking again complex numbers, the real part could be of type integer, and the imaginary part of type rational.

By compatible, we mean types which can be freely mixed in operations like $+$ or $*$. For example if the real part is of type real, the imaginary part cannot be of type integermod (integers modulo a given number n).

Let us now describe the types. As explained above, they are built recursively from basic types which are as follows. We use the letter T to designate any type; the type number corresponds to the internal representation of the type.

type 1:	$\mathbf{Z}$	Integers (with arbitrary precision)
type 2:	$\mathbf{R}$	Real numbers (with arbitrary precision)
type 3:	$\mathbf{Z}/n\mathbf{Z}$	Integermods (integers modulo n)
type 4:	$\mathbf{Q}$	Rational numbers (in irreducible form)
type 5:	$\mathbf{Q}$	Rational numbers (in non necessarily irreducible form)
type 6:	$T[i]$	Complex numbers
type 7:	$\mathbf{Q}_p$	p-adic numbers
type 8:	$T[w]$	Quadratic numbers (where $[\mathbf{Z}[w] : \mathbf{Z}] = 2$)
type 9:	$T[X]/P(X)T[X]$	Polymods (polynomials modulo P)
type 10:	$T[X]$	Polynomials
type 11:	$T((X))$	Power series (finite Laurent series)
type 13:	$T(X)$	Rational functions (in irreducible form)
type 14:	$T(X)$	Rational functions (in non necessarily irreducible form)
type 17:	T^n	Row (i.e. horizontal) vectors
type 18:	T^n	Column (i.e. vertical) vectors
type 19:	$\mathcal{M}_{m,n}(T)$	Matrices.

In addition, there exist type 15 for binary quadratic forms of positive discriminant, type 16 for binary quadratic forms of negative discriminant which can be used in specific operations, but which may disappear in future versions, hence their use is not documented in this manual.

Let us look closely at these types.

1.2.1 Integers and reals: they are of arbitrary and varying length (each number having coded internally its own length or precision) with the following mild restrictions: integers must be in absolute value less than $2^{1048480}$ (i.e. roughly 315623 digits). The precision of real numbers is also at most 315623 significant decimal digits, and the binary exponent must be in absolute value less than $2^{23} = 8388608$.

Note that PARI has been optimized so that it works as fast as possible on numbers with 40 to 100 decimal digits. In particular, no fancy multiplication technique is used. Hence, although it is possible to use PARI to do computations with 300000 decimal digits, much better programs can be written for these loooong numbers.

Integers and real numbers are completely non recursive types and are sometimes called the **leaves**.

1.2.2 Integermods, rational numbers (irreducible or not), p-adic numbers, polymods, and rational functions: they are recursive, but in a forced manner:

For integermods or polymods, there are two components: the modulus, which must be of type integer (resp. polynomial), and the number (resp. polynomial).

For rational numbers or rational functions, there are also only two components: the numerator and the denominator, which must both be of type integer (resp. polynomial).

Finally, p-adic numbers have three components: the prime p, the "modulus" p^k, and an approximation to the p-adic number. Here $\mathbf{Z}_p$ is considered as $\varprojlim \mathbf{Z}/p^k\mathbf{Z}$, and $\mathbf{Q}_p$ as its field of fractions. Like real numbers, the codewords contain also the information on the precision of the number (which is in fact redundant with p^k, but it is apparently more efficient to keep this redundancy).

1.2.3 Complex numbers ($a + bi$) and quadratic numbers: quadratic numbers are numbers of the form $a + bw$, where w is such that $[\mathbf{Z}[w] : \mathbf{Z}] = 2$, and more precisely $w = \sqrt{d}/2$ when $d \equiv 0 \bmod 4$, and $w = (1+\sqrt{d})/2$ when $d \equiv 1 \bmod 4$, where d is the discriminant of a quadratic order.

Complex and quadratic numbers are partially recursive: the two components a and b can be of type integer, real, rational, integermod or p-adic, and can be mixed, subject to the limitations mentioned above. For example, $a + bi$ with a and b p-adic is in $\mathbf{Q}_p[i]$, but this is equal to $\mathbf{Q}_p$ when $p \equiv 1 \bmod 4$, hence we must exclude these p when one explicitly uses a complex p-adic type.

1.2.4 Polynomials, power series, vectors and matrices: they are completely recursive: their components can be of any type, and types can be mixed (however beware when doing operations). Note in particular that a polynomial in two variables is simply a polynomial with polynomial coefficients.

Note that in the present version 1.38.1 of PARI, there is a bug in the handling of power series of power series (i.e. power series in several variables). However power series of polynomials (which are power series in several variables of a special type) are OK. The reason for this bug is known, but it is difficult to correct because the mathematical problem itself contains some amount of imprecision.

1.2.5 Exact and imprecise objects: we have already said that integers and reals are called the leaves because they are ultimately at the end of every branch of a tree representing a PARI object. Another important notion is that of an **exact object**: by definition, numbers of basic type real, p-adic or power series are imprecise, and we will say that a PARI object having one of these imprecise types anywhere in its tree is not exact. All other PARI objects will be called exact. This is a very important notion since no numerical analysis is involved when dealing with exact objects.

1.2.6 What is zero?: this is a crucial question in all computer systems. The answer we give in PARI is the following. For exact types, all zeros are equivalent and are exact, so are usually represented as an integer zero. The problem becomes nontrivial for imprecise types. For p-adics the answer is as follows: every p-adic number (including 0) has an exponent e and a "mantissa" u which is a p-adic unit, except when the number is zero (in which case u is zero), the mantissa having a certain "precision" k (i.e. is defined modulo p^k). Then this p-adic zero is understood to be equal to $O(p^e)$, i.e. there is an infinite number of zeros. The number k is thus irrelevant.

For power series the situation is similar, with p replaced by X, i.e. a power series zero will be $O(X^e)$, the number k (here the length of the power series) being also irrelevant.

For real numbers, the precision k is also irrelevant, and a real zero will in fact be $O(2^e)$ where e is now usually a negative binary exponent. This of course will be printed as usual for a real number ($0.0000\cdots$ in `f` format or $0.Exx$ in `e` format) and not with a $O()$ symbol as with p-adics or power series.

1.2.7 Scalar types: basic types with type number less than or equal to 9 will be called scalar types because they essentially occur as coefficients of some more complicated object. Note that type 9, i.e. polymod, is used to define algebraic extensions of a base ring, and as such is a scalar type.

1.3 Operations and functions.

1.3.1 The PARI philosophy. The basic philosophy which governs PARI is that operations and functions should first give as exact a result as possible, and second, be permitted if they make any kind of sense.

More specifically, if you do an operation (not a transcendental one) between exact objects, you will get an exact object. For example, dividing 1 by 3 does not give $0.33333\cdots$ as you could expect, but simply the rational number (1/3). If you really want the result in type real, evaluate 1./3 or add 0. to (1/3).

The result of operations between imprecise objects will be as precise as possible. Consider for example one of the most difficult cases, that is the addition of two real numbers x and y. The accuracy of the result is *a priori* unpredictable; it depends on the precisions of x and y, on their sizes (i.e. their exponents), and also on the size of $x+y$. PARI works out automatically the right precision for the result, even when it is working in calculator mode GP where there is a default precision. See the technical reference manual for the precise formulas giving the precision of the result.

The second part of the PARI philosophy is that PARI operations are in general quite permissive. For instance taking the exponential of a vector should not make sense. However, it is frequent that a computation comes out with a result which is a vector with many components, and one wants to get the exponential of each one. This could easily be done either under GP or in library mode, but in fact PARI assumes that this is exactly what you want to do when you take the exponential of a vector, so no work is necessary. Most transcendental functions work in the same way (see chapter 3 for details).

An ambiguity would arise with square matrices. PARI always considers that you want to do componentwise function evaluation, hence to get for example the exponential of a square matrix you would need to use a function with a different name, `matexp` for instance. In the present version 1.38.1, this is not yet implemented. See however the program in Appendix C, which is a first attempt for this particular function.

The available operations and functions in PARI are described in detail in chapter 3. Here is a brief summary:

1.3.2 Standard operations.

The four standard operators $+$, $-$, $*$, $/$ of course exist. It should be once more emphasized that division is, as far as it can, an exact operation (4 divided by 3 gives (4/3)). In addition to this, operations on integers or polynomials, like \ (Euclidean division), % (Euclidean remainder) exist. There is also the exponentiation operator ^, when the exponent is of type integer (otherwise it is considered as a transcendental function). Finally, the logical operators && (`and` operator), || (`or` operator) exist, giving as results 1 (true) or 0 (false). Note that the C-syntax & and | is also accepted as a synonym. However, there is no bitwise `and` or `or`.

1.3.3 Conversions and similar functions.

Many conversion functions are available to convert between different types. For example floor, ceiling, rounding, truncation, etc.... Other simple functions are included like real and imaginary part, conjugation, norm, absolute value, changing precision or creating an integermod or a polymod.

1.3.4 Transcendental functions.

They usually operate on any object in $\mathbf{C}$, and some also on p-adics. The list is everexpanding and of course contains all the elementary functions, plus already a number of others. Recall that by extension, PARI usually allows a transcendental function to operate componentwise on vectors or matrices.

1.3.5 Arithmetic functions.

Apart from a few like the factorial function or the Fibonacci numbers, these are functions which explicitly use the prime factor decomposition of integers. The standard functions are included. In the present version 1.38.1, a primitive, but useful version of Lenstra's Elliptic Curve Method (ECM) has been implemented. More sophisticated functions are implemented, like finding integral bases and discriminants of number fields, computing class numbers and regulators (for quadratic fields only), and also many functions dealing with elliptic curves over $\mathbf{Q}$ or over local fields.

1.3.6 Other functions.

Quite a number of other functions dealing with polynomials (e.g. finding complex or p-adic roots, factoring,...), power series (e.g. substitution, reversion), linear algebra (e.g. determinant, characteristic polynomial, linear systems), and different kinds of recursions are also included. In addition, standard numerical analysis routines like Romberg integration (open or closed, on a finite or infinite interval), real root finding when the root is bracketed, polynomial interpolation, series acceleration, and plotting are included. See the last sections of chapter 3 for details.

Originally, GP was designed as a debugging tool for the PARI system library, and hence not much thought had been given to make it user-friendly. The situation has now changed, and GP is very useful as a standalone tool. Hence many new functionalities have been added. The operations and functions available in PARI and GP will be described in the next chapter; in the present one we describe the specific use of the GP programmable calculator.

GP (and in fact any program using the PARI library) needs a stack in which to do its computations, and a table of primes, which may be very small. The default stack size is 1000000 bytes on the Mac, and 4000000 on other machines. The prime number limit is by default 200000 on the MacII, and 500000 on other machines. These defaults can be changed by adding parameters to the input line. To start the calculator, the general syntax is:

```
gp [-s stacksize] [-p primelimit] [-b buffersize]
```

where as usual items within brackets are optional. The buffersize option is to allow for overly large input files read into GP, for example large matrices. The default is 30000, which is usually more than plenty.

(Note that on the Macintosh, even after clicking on the gp icon, once in the MPW Shell, you still need to type explicitly the above command. If you are uusing the standalone version SAGp, the limits are hardwired and cannot be changed.)

If you have GNUemacs, it is strongly advised to work in a special emacs shell (see section 3.12), which is started by typing `M-x gp` (where as usual `M` is the Meta key) if you accept the default stack, prime and buffer sizes, or `C-u M-x gp` which will ask you for the name of the gp executable, the stack size, the prime limit and the buffersize.

A copyright message then appears which includes the version number. Please note this number, so as to be sure to have the most recent version if you wish to have updates of PARI. The present manual is written for version 1.38.1, but only details will change (like new and faster functions, less bugs, better output, etc...) until the next major revision 2.00.

After the copyright, the computer works for a few seconds (it is in fact computing and storing a table of primes), writes the top-level help information, the initial defaults, and then waits after its prompt, initially "?".

Note that at any point the user can type "Ctrl-C" in GP, and the current computation is interrupted and control is given again to the user at the GP prompt.

2.1 Metacommands.

The top-level help information tells you that (as in many systems), to get help you should type a "`?`". When you do this, a menu appears, describing the ten main categories of available functions and what to do to get more precise help. If you now type "`?n`" with $1 \leq n \leq 11$ you will get the list of commands corresponding to category n and simultaneously to Section $3.n$ of this manual.

If you type "`?functionname`" where `functionname` is the name of a PARI function, you will get a short explanation about this function.

Finally, if you type "`?\`" you will get a short description of the metacommands. In the rest of this section, we will describe these commands in more detail.

As a general rule, under GP, commands starting with "\" and some other symbols like "?", are not computing commands, but are metacommands which correspond to information exchange with GP. The available commands of this sort can be divided among simple commands (explained here) and default setting commands (explained in the next section).

Simple commands are case insensitive. For example, \Q is identical to \q.

2.1.1 \\: comment. The rest of the line is ignored by GP.

2.1.2 **\a**n: print the object number n ($\%n$) in raw format. If the number n is omitted, print the latest computed object (%).

2.1.3 **\b**n: print the object number n ($\%n$) in prettyprint (i.e. beautified) format. If the number n is omitted, print the latest computed object (%).

2.1.4 **\c**: prints the list of all available functions under GP, not including functions using a special character (specifically + , − , ∗ , /, \ , ^ , ! , ~ , _ , % , &, |). These names being in general not informative enough, more information can be obtained by typing "**?**command", which gives in short form the effect of the command and the name and syntax of the corresponding library function. Of course, to have complete information, read chapter 3 of this manual. Much better help can be obtained if you work under GNUemacs (see section 3.12). As already mentioned, if you type "**?**n" where n is a number from 1 to 11, you will only get the list of functions in section 3.n of the manual (the list of sections being obtained by simply typing "**?**").

2.1.5 **\d**: prints the defaults (see the next section), including the stacksize, primelimit and buffersize. This command is always executed (without user intervention), when GP is started.

2.1.6 **\e**: switches the echo mode on and off (the default is off). When echo mode is on, the command is reprinted before being executed. This can be useful when reading a file by the \r command. It is for example the first command of the test file `testin` used to check the correctness of GP installation.

2.1.7 **\g**n: set the debugging level equal to the positive integer n. The default is $n = 0$. If the debugging level is nonzero, some messages may be printed (in French) according to what is going on.

2.1.8 **\k**: kill and reinitialize GP. Apart from the initial message, this is equivalent to exiting from GP and starting it again, apart from the fact that the variables which have been used are still known to GP.

Warning. In the present version 1.38.1, do not use this command after a stack doubling has occured. In this case, if you really want to kill the session, simply finish the session and restart.

2.1.9 **\l**: switch the logfile on/off. Initially, the GP session does not have a log file where all the commands and results are stored. Typing \l will open a file named `pari.log` (and destroy any previous file with that name), and until a subsequent \l, all the commands and results will be written in that file.

2.1.10 **\m**: print the object number n ($\%n$) in prettymatrix format. If the number n is omitted, print the latest computed object (%).

2.1.11 **\p**: changes the default format from raw to prettymatrix to prettyprint (i.e. beautified) to raw format.

2.1.12 \q: quits GP and returns to the system.

2.1.13 \r filename: reads into GP all the objects written on the named file using the \w command (see 2.1.13.).

2.1.14 \s: prints the state of the PARI stack, and in particular the amount of memory used, expressed as a percentage. This enables the user to see whether the stack is getting too full.

2.1.15 \s(n): prints the state of the PARI stack, and in addition the explicit hexadecimal representations of the first n longwords of the stack (going down) if n is positive, and the complete stack if n is negative. This is used primarily as a debugging device for PARI, and is not intended for the casual user.

2.1.16 \t: prints the internal longword format of all the PARI types. The detailed bit or byte format of the initial codeword(s) is explained in the technical reference manual, but its knowledge is not necessary for a GP user.

2.1.17 \v: prints the version number type of implementation (680x0, sparc, other) of the program you are using. Note that the subversion may be omitted (e.g. 1.37 instead of 1.37.3).

2.1.18 \w$n$ filename: write the object number n ($\%n$) on the named file, in raw format. If the number n is omitted, write the latest computed object (%).

2.1.19 \x: print the complete tree with addresses and contents in hexadecimal, of the internal representation of the latest computed object in GP. As for \s, this is used primarily as a debugging device for PARI, and is not intended for the casual user. However, used on a PARI integer, it can be used as a decimal→hexadecimal converter.

2.1.20 \y: switches the automatic simplification feature on or off. When the Pari library makes a computation, the type of the result is not always the simplest possible. The only type conversions which the Pari library does automatically are rational numbers to integers (when they are of type 4 and equal to integers), and similarly rational functions to polynomials (when they are of type 13 and equal to polynomials). This feature is useful in many cases, and saves time, but can be often annoying in practice. Hence there exists the function `simplify` (see Chapter 3) which allows to simplify to the simplest possible types recursively.

When automatic simplification is on, the function `simplify` is called at the end of any GP computation, so that all printed results are simplified. However, the intermediate results will not be simplified otherwise the loss of time would be too large. By default, automatic simplification is turned on.

2.1.21 #: switches the timer on or off. The timer counts electrical cycles, so is precise only to ±20ms in Europe and ±16.7ms in North America. The time measured is the user CPU time, *not* including the time for printing. By default, when GP is launched, the timer is switched off.

2.2 Defaults and output formats.

The other commands starting with "\" are for setting the defaults. Note that in the syntax given below, the "=" sign must be written explicitly, and n represents a non negative integer.

2.2.1 \precision= n: Sets to n the number of significant digits, and at the same time the number of printed digits of real numbers. The initial default is 28. Note that one can also use the function `setprecision` (see chapter 3) to achieve the same goal, and this *must* be done if it is inside a GP expression or program.

2.2.2 \serieslength= n: Sets to n the number of significant terms in power series. The initial default is 16. Note that one can also use the function `setserieslength` (see chapter 3) to achieve the same goal, and this *must* be done if it is inside a GP expression or program.

2.2.3 \format=x$n.m$ (where x is a letter): if the letter is "f", real numbers will be printed in fixed floating point format with no explicit exponent (e.g. 0.000033); if the letter is "e", they will be printed in scientific format, always with an explicit exponent (e.g. 3.3e-5). If the letter is "g", real numbers will be printed in "f" format, except when their absolute value is less than 2^{-32}, in which case they are printed in "e" format. The initial default is "g".

The number n is the number of significant digits printed for real numbers except if $n \leq 0$ where all the significant digits will be printed (initial default 28), and the number m is the number of characters to be used for printing integers, but is ignored if is equal to zero (initial default 0). This is a feeble attempt at formatting.

2.2.4 \prompt=string: Sets the prompt equal to the given string. The initial default is "?".

2.2.5 Raw, prettymatrix and beautified format: in addition to the default formats which can be set as explained above, you can print results in three ways. Either in "raw" format, i.e. a format which is equivalent to what you input, including explicit multiplication signs, and everything typed on a line instead of two dimensional boxes. This can have several advantages, for instance it allows you to pick the result with a mouse or an editor, and to put it somewhere else.

The second format is the "prettymatrix". The only difference with the raw format is that matrices are printed as boxes instead of horizontally. This is prettier, but takes more space and cannot be used for input. Column vectors are still printed horizontally. This is the default.

The third format is the "prettyprint" or beautified format. In the present version 1.38.1, this is not beautiful at all. The command $\backslash p$ is used to navigate between these formats as explained above, and independently of the default format, an object can be printed in any of the three formats using the commands $\backslash a$, $\backslash m$ and $\backslash b$ respectively.

2.2.6 Note on the output formats.

A zero real number is printed in "e" format as $0.Exx$ where xx is the (usually negative) *decimal* exponent of the number (cf. 1.2.6.). This allows the user to check the accuracy of the zero in question (this could also be done using \x, but it would be more technical).

When the integer part of a real number x is not significant because the exponent of x is greater than the internal precision, an approximation to the integer part is printed, followed by a "$*$" instead of a decimal point, indicating corrupt digits on the left.

Note also that a number of type integer or real is written without enclosing parentheses, while most other types have them. Hence, if you see printed the expression (3.14), it is not of type real, but probably of type complex with zero imaginary part (if you want to be sure, type \x).

Finally note that in the present version 1.38.1, the printing of 0, 1, and -1 as coefficients in beautified format, although understandable, is buggy. Hence the default is the raw format until this is improved.

2.3 Input formats for the PARI types.

Apart from commands, the general input in GP after the prompt is a sequence of legal expressions. Before describing this in detail in the next section, let us see here how to input the different types of PARI.

Note that blanks are ignored in an expression.

2.3.1 Integers (type 1): type the integer (with an initial + or − if desired) with no decimal point.

2.3.2 Real numbers (type 2): type the number with a decimal point. The internal precision of the real number will be the supremum of the input precision and the default precision. For example, if the default precision is 28 digits, typing 2. will give a number with internal precision 28, but typing a 45 significant digit real number will give a number with internal precision at least 45 (although less may be printed).

You can also use scientific notation with the letter "E" or "e" (like 6.02 E 23 or 1e-5).

2.3.3 Integermods (type 3): to enter $n \bmod m$, type mod(n, m), not $n\%m$ (see 3.2.15.).

2.3.4 Rational numbers (types 4 and 5): under GP, all fractions are automatically reduced to lowest terms, so it is not possible to enter a non-irreducible fraction (type 5), although of course in library mode this is easy. To enter n/m just type it as written. As explained in section 3.1.4., division will *not* be performed, only reduction to lowest terms.

2.3.5 Complex numbers (type 6): to enter $x + iy$, type $x + I * y$ or $x + i * y$. The letters I and i stand for $\sqrt{-1}$. Recall from chapter 1 that x and y can be of type integer, real, integermod fraction, or p-adic.

2.3.6 p-adic numbers (type 7): to enter a p-adic number, simply write a rational or integer expression and add to it $O(p\hat{\ }k)$, where p and k are integers. This last expression indicates three things to GP: first that it is dealing with a p-adic (the fact that p is an integer), second the "prime" p (note that no verification is done that p is indeed prime; you can work on 10-adics if you want, but beware of disasters as soon as you do something non trivial like taking a square root), and finally the number of significant p-adic digits k.

For example, to obtain the 7-adic number

$$2 * 7\hat{\ }(-1) + 3 + 4 * 7 + 2 * 7\hat{\ }2 + O(7\hat{\ }3),$$

you can type it exactly as shown, or equivalently as

$$905/7 + O(7\hat{\ }3).$$

2.3.7 Quadratic numbers (type 8): first, you must define the default quadratic order or field in which you want to work. This is done using the `quadgen` function, in the following way. Write something like

```
w=quadgen(d)
```

where `d` is the *discriminant* of the quadratic order in which you want to work (hence $d \equiv 0$ or $1 \bmod 4$). The name `w` is of course just a suggestion, but corresponds to traditional usage. You can of course use any variable name that you like. However, quadratic numbers are always printed with a `w`, regardless of the discriminant. So beware, two numbers can be printed in the same way and not be equal. However GP will refuse to add or multiply them for example.

Then, $(1, w)$ will be the "canonical" integral basis of the quadratic order (i.e. $w = \sqrt{d}/2$ if $d \equiv 0 \bmod 4$, and $w = (1+\sqrt{d})/2$ if $d \equiv 1 \bmod 4$, where d is the discriminant), and to enter $x + yw$ you just type $x + y * w$ or $x + y * W$.

2.3.8 Polymods (type 9): exactly as for integermods, to enter $x \bmod y$ (where x and y are polynomials), type mod(x, y), not $x\%y$ (see 3.2.12.). Note that when y is an irreducible polynomial in one variable, polymods whose modulus is y are simply algebraic numbers in the finite extension defined by the polynomial y. This allows us to work easily in number fields, finite extensions of the p-adic field $\mathbf{Q}_p$, or finite fields.

Important remark. Since the variables occurring in a polymod are not free variables, it is essential in order to avoid inconsistencies, that all polymods have the same structure, i.e. the main variable will be number 0, the next number 1, etc.... In other words, since 0 is the variable number of "X", do **not** use expressions like `mod(Y,Y^2+1)` since PARI will not recognize that this is identical to `mod(X,X^2+1)`. Another consequence is that an operation like `X+mod(X,X^2+1)` gives exactly the same thing as a result (meaning $X + i$ where $i^2 = -1$), and not `mod(2*X,X^2+1)`.

2.3.9 Polynomials (type 10): type the polynomial in a natural way, not forgetting to put a "$*$" between a coefficient and a formal variable (this $*$ does not appear on beautified output). Any variable name can be used except for the reserved names `i` (used exclusively for the square root of -1), `pi` (3.14...) and `euler` (Euler's constant). The total number of different variable names is limited to 256, which should be enough. If you ever need hundreds of variables, you should probably use vectors instead.

2.3.10 Power series (type 11): type a rational function or polynomial expression and add to it $O(expr\hat{}k)$, where *expr* is an expression having a non zero valuation, like a polynomial, power series, or a rational function (the most common cases being simply *expr* =variable name). This indicates to GP that it is dealing with a power series, and the desired precision is k times the valuation of *expr* with respect to its main variable (to check the order of the variables, or to modify it, use the function `reorder` see 3.11.4.6).

2.3.11 Rational functions (types 13 and 14): under GP, all fractions are automatically reduced to lowest terms, so it is not possible to enter a non-irreducible rational function (type 14), although of course in library mode this is easy. To enter x/y just type it as written. As explained in section 3.1.4., division will *not* be performed, only reduction to lowest terms.

2.3.12 Binary quadratic forms of positive discriminant (type 15): enter a 4-component (row or column) vector, and transform it into a binary quadratic form using the function `qfr` (see 3.4.45). The fourth component (which is related to the regulator, more precisely to Shanks' "distance") must be a real number. See also the function `pf` which directly creates a prime form of given discriminant (see 3.4.37).

2.3.13 Binary quadratic forms of negative discriminant (type 16): enter a 3-component (row or column) vector, and transform it into a binary quadratic form using the function `qfi` (see 3.4.44). See also the function `pf` which directly creates a prime form of given discriminant (see 3.4.37).

2.3.14 Row and column vectors (types 17 and 18): to enter a row vector, type the components separated by commas ",", and enclosed between brackets "[" and "]", e.g. $[x, y, z]$. To enter a column vector, type the vector horizontally, and add a tilde "~" to transpose.

2.3.15 Matrices (type 19): to enter a matrix, type the components line by line, the components being separated by commas ",", the lines by semicolons ";", and everything enclosed in brackets "[" and "]", e.g. $[x, y; z, t; u, v]$.

Note that although the internal representation is essentially the same, (only the type number being different), a row vector of column vectors is *not* a matrix; for example, multiplication will not work in the same way. To transform it into a matrix, use the function `mat` (see 3.8.38).

2.4 The general GP input line.

A general GP session goes as follows: a sequence of characters is typed by the user after the prompt. This can be either a function definition, or an expression or a sequence of expressions (i.e. a program). In this case, after the last expression is computed the result is put in an internal array. The successive elements of this array are called %1, %2, ...under GP, and they can also be used in library mode with the names `g[1], g[2], ...` In this case, the array is not automatically filled for you. As a shortcut, the last computed expression can also be called %.

If you want to suppress the printing of the result, for example because it is a long unimportant intermediate result, end the expression with a ";" sign. This same sign is used as an instruction separator when several instructions are written on the same line (note that the ":" sign can also be used, but we will try to keep C-style conventions). Note that the last expression you have computed, even if not printed, still has the name (%1, %2, ...) so care must be used when using it subsequently since its number does not appear explicitly. Of course, if it is simply used on the next line, use % as usual.

Any legal expressions can be typed, and it is evaluated using the usual conventions about operator priorities and left to right evaluation (including raising to a power), using the available operator symbols, function names (including user-defined functions), and special variables. Please note that in general there is no distinction between lowercase. and uppercase Also, note that blanks are completely ignored in the input to GP.

The special variable names used by GP are `euler` (Euler's constant=0.577...), `i` (the square root of -1), `pi` (3.14...), which are really functions without parameters, and `O` whose syntax is the following:

`O`$(expr\hat{\ }k)$

When *expr* is an integer or a rational number, creates a *expr*-adic number (zero in fact) of precision k. When *expr* is a polynomial, a power series or a rational function whose main variable is X, say, this creates a power series (also zero) of precision $v * k$ where v is the X-adic valuation of *expr* (see 2.3.6 and 2.3.9.).

Note: as usual, lowercase **o** is allowed, but we do not advise to use it since it has a different mathematical meaning.

The functions and operators available in PARI and in the GP/PARI calculator are numerous and everexpanding. Here is a description of the ones available in version 1.38.1. It should be noted that many of these functions accept quite different types as arguments, but others are more restricted. The list of acceptable types will be given for each function or class of functions. Except when stated otherwise, it is understood that a function or operation which should make natural sense is legal. In this chapter, we will describe the functions according to a rough classification. For the functions in alphabetical order, see the general index.

3.1 Standard monadic or dyadic operators.

3.1.1 $\pm$ $+x$ and $-x$ refer to monadic operators ($+x$ does nothing and $-x$ negates x).

The library syntax is **gneg**(x) for $-x$.

3.1.2 +, − $x+y$ and $x-y$ are the sum and difference of x and y. Among the prominent impossibilities are addition/subtraction between a scalar type and a vector or a matrix, between vector/matrices of incompatible sizes and between an integermod and a real number.

The library syntax is **gadd**(x, y) for $x+y$, **gsub**(x, y) for $x-y$.

3.1.3 $*$ $x*y$ is the product of x and y. Among the prominent impossibilities are multiplication between vector/matrices of incompatible sizes, between an integermod and a real number. Note that because of vector and matrix operations, $*$ is not necessarily commutative. Note also that since multiplication between two column or two row vectors is not allowed, to obtain the scalar product of two vectors of the same length, you must multiply a line vector by a column vector, if necessary by transposing one of the vectors (using `~` or `trans( )`, see section 8).

The library syntax is **gmul**(x, y) for $x*y$.

3.1.4 / x/y is the quotient of x and y. In addition to the impossibilities for multiplication, note that if the divisor is a matrix, it must be an invertible square matrix, and in that case the result is $x*y^{-1}$. Furthermore note that the result is as exact as possible: in particular, division of two integers always gives a rational number (which may be an integer if the quotient is exact) and *not* the Euclidean quotient (see $x\backslash y$ for that), and similarly the quotient of two polynomials is a rational function in general. To obtain the approximate real value of the quotient of two integers, add 0. to the result; to obtain the approximate p-adic value of the quotient of two integers, add $O(p^k)$ to the result; finally, to obtain the taylor series expansion of the quotient of two polynomials, add $O(X\hat{}k)$ to the result or use the `taylor` function (see section 7).

The library syntax is **gdiv**(x, y) for x/y.

3.1.5 \ $x\backslash y$ is the euclidean quotient of x and y. The types must be either both integer or both polynomials. The result is the Euclidean quotient. In the case of integer division, the quotient is such that the corresponding remainder is nonnegative.

The library syntax is **gdivent**(x, y) for $x\backslash y$.

3.1.6 % $x\%y$ is the euclidean remainder of x and y. The modulus y must be of type integer or polynomial. The result is the remainder, always nonnegative in the case of integers. Allowed dividend types are scalar exact types when the modulus is an integer, and polynomials, polymods and rational functions when the modulus is a polynomial.

The library syntax is **gmod**(x, y) for $x\%y$.

3.1.7 divres(x, y) creates a vector with two components, the first being the Euclidean quotient, the second the Euclidean remainder, of the division of x by y. This avoids the need to do two divisions if one needs both the quotient and the remainder. The arguments must be both integers or both polynomials, and in the case of integers the remainder is of the sign of the dividend.

The library syntax is **gdiventres**(x, y).

3.1.8 ^ x^y is powering. If the exponent is an integer, then exact operations are performed using binary powering techniques. In particular, in this case the first argument cannot be a vector or matrix unless it is a square matrix (and moreover invertible if the exponent is negative). If the exponent is not of type integer, this is treated as a transcendental function (see section 3), and in particular has the effect of componentwise powering on vector or matrices.

The library syntax is **gpui**$(x, y, prec)$ for x^y where the precision parameter prec must be given only if y is not integral but x is an exact (real or complex) type.

In addition, there are two shift functions (i.e. multiplication and division by powers of 2) which are faster than the general multiplication or division routines:

3.1.9 shift(x, n): shift x componentwise left by n bits if $n \geq 0$ and right by $-n$ bits if $n < 0$. A left shift by n corresponds to multiplication by 2^n, and a right shift by $-n$ corresponds to Euclidean division of x by 2^{-n}.

The library syntax is **gshift**(x, n) where n is a C-integer.

3.1.10 shiftmul(x, n): multiply x by 2^n. The difference with `shift` is that when $n < 0$, the ordinary division takes place, hence for example if x is an integer the result may be a fraction, while for `shift` Euclidean division takes place when $n < 0$ hence if x is an integer the result is still an integer.

The library syntax is **gmul2n**(x, n) where n is a C-integer.

3.1.11 comparison and boolean operators. The six standard comparison operators $<=$, $<$, $>=$, $>$, $==$, $!=$ are available in GP, and in library mode under the names **gle, glt, gge, ggt, geq, gne** respectively. The library syntax is **co**(x, y), where co is the comparison operator. The result is 1 if the comparison is true, 0 if it is false.

The standard boolean functions || (inclusive or) and && (and) are also available, and the library syntax is **gor**(x, y) and **gand**(x, y) respectively. Note that to avoid confusion with the factorial function, there is no ! (not) operator, but this can easily be circumvented.

In library mode, it is in fact usually preferable to use the two basic functions which are **gcmp**(x, y) which gives the sign (1, 0, or -1) of $x - y$, where x and y must be in $\mathbf{R}$, and **gegal**(x, y) which can be applied to any two PARI objects x and y and gives 1 (i.e. true) if they are equal (but not necessarily identical), 0 (i.e. false) otherwise. Particular cases of gegal which should be used are **gcmp0**(x) ($x = 0$?), **gcmp1**(x) ($x = 1$?), and **gcmp_1** (x) ($x = -1$?).

Note that **gcmp0**(x) tests whether x is equal to zero, even if x is not an exact object. To test whether x is an exact object which is equal to zero, one must use **isexactzero**.

GP accepts the following synonyms for some of the above functions: since there is no bitwise `and` or bitwise `or`, | and & are accepted as synonyms of || and && respectively. Also, <> is accepted as a synonym for ! =. On the other hand, = is definitely *not* a synonym for == since it is the assignment statement.

3.1.12 lex(x, y): gives the result of a lexicographic comparison between x and y. This is to be interpreted in quite a wide sense. For example, the vector $[1, 3]$ will be considered smaller than the longer vector $[1, 3, -1]$ (but of course larger than $[1, 2, 5]$), i.e. `lex([1,3],[1,3,5])` will give -1 as a result.

The library syntax is **lexcmp**(x, y).

3.1.13 sign(x): sign of x, which must be of type integer, real or fraction. The result (0, 1 or -1) is a C-integer.

The library syntax is **gsigne**(x).

3.1.14 max(x, y) and **min**(x, y) create the maximum and minimum of x and y when they can be compared.

The library syntax is **gmax**(x, y) and **gmin**(x, y).

Important remark. In all the above library syntaxes, we have given only the basic names of the functions. For example `gadd`(x, y) assumes that x and y are PARI objects (the GEN type) and *creates* the result $x + y$ on the PARI stack. From most of these basic names, the names and effects of other functions can be obtained in the following way. We give the example for gadd, but the same is true for all the other functions above, and a few more that we have not given:

gaddgs(x, y): here x is a GEN, y is an ordinary 32-bit integer.

gaddsg(x, y): here x is an ordinary 32-bit integer, y is a GEN.

gaddz(x, y, z), **gaddgsz**(x, y, z), **gaddsgz**(x, y, z): here z is a preexisting GEN and the result of the corresponding operation is put in z. The size of the PARI stack does not change.

As simplification to programming, for many functions beginning with a **g**, such as gadd, one can replace the **g** by an **l** to obtain a **long** result instead of a GEN (i.e. pointer to long) result. For instance **ladd**(x, y) is identical with ((`long`)`gadd`(x, y)). These macros are written in the files listed in `gencom.h`.

3.2 Conversions and similar elementary functions

Many of these conversion functions are rounding or truncating operations. In this case, if the argument is a rational function, the result is the Euclidean quotient of the numerator by the denominator, and if the argument is a vector or a matrix, the operation is done componentwise. This will not be restated for every function.

3.2.1 binary(x): outputs the vector of the binary digits of $|x|$. Here x can be an integer, a real number (in which case the result has two components, one for the integer part, one for the fractional part) or a vector/matrix.

The library syntax is **binaire**(x).

3.2.2 bittest(x, n): outputs the n^{th} bit of $|x|$ starting from the right (i.e. the coefficient of 2^n in the binary expansion of x. The result is 0 or 1.

The library syntax is **bittest**(x, n) where n as well as the result are C-integers.

3.2.3 bytesize(x): outputs the total number of bytes occupied by the tree representing the Pari object x.

The library syntax is **taille2**(x) which returns a C `long`. The function `taille` returns the number of *words* instead.

3.2.4 ceil(x): ceiling of x. When x is in **R**, the result is the smallest integer greater than or equal to x.

The library syntax is **gceil**(x).

3.2.5 centerlift(x): lifts an element $x = a \bmod n$ of $\mathbf{Z}/n\mathbf{Z}$ to a in $\mathbf{Z}$, and similarly lifts a polymod to a polynomial. This is the same as `lift` except that in the particular case of elements of $\mathbf{Z}/n\mathbf{Z}$, the lift y is such that $-n/2 < y \leq n$. If x is of type fraction, complex, quadratic, polynomial, power series, rational function, vector or matrix, the lift is done for each coefficient. Forbidden types for x are reals and p-adics. Note that the main variable of the lift of a polymod will be variable number 0 (i.e. 'X') if the rules concerning the creation of polymods explained in chapter 2 are observed.

The library syntax is **centerlift**(x).

3.2.6 changevar(x, y): change the variables in the object x according to the permutation specified by the vector y. For example, assume that the variables have been introduced in the order `x`, `a`, `b`, `c`. Then, if y is the vector `[x,c,a,b]`, the variable `a` will be replaced by `c`, `b` by `a`, and `a` by `b`, `x` being unchanged. Note that the permutation must be completely specified, e.g. `[c,a,b]` would not work, since this would replace `x` by `c`, and leave `a` and `b` unchanged.

The library syntax is **changevar**(x, y).

3.2.7 components of a PARI object:

There are essentially three ways to extract the components from a PARI object.

The first and most general, is the function **compo**(x, n) which extracts the n^{th}-component of x. This is to be understood as follows: every PARI type has one or two initial code words. The components are counted, starting at 1, after these code words. In particular if x is a vector, this is indeed the n^{th}-component of x, if x is a matrix, the n^{th} column, if x is a polynomial, the n^{th} coefficient (i.e. of degree $n-1$), and for power series, the n^{th} significant coefficient. The use of the function `compo` implies the knowledge of the structure of the different PARI types, which can be recalled by typing \t under GP.

The library syntax is **compo**(x, n), where n is a 32-bit C-integer.

The two other methods are more natural but more restricted. First, the function **coeff**(x, n) gives the coefficient of degree n of the polynomial or power series x, with respect to the main variable of x (to see the order of the variables or to change it, use the function `reorder`, see section 11). In particular if n is less than the valuation of x or in the case of a polynomial, greater than the degree, the result is zero (contrary to `compo` which would send an error message). If x is a power series and n is greater than the largest significant degree, then an error message is issued.

For greater flexibility, vector or matrix types are also accepted for x, and the meaning is then identical with that of `compo`.

Finally note that a scalar type is considered by `coeff` as a polynomial of degree zero.

The library syntax is **truecoeff**(x, n).

The third method is specific to vectors or matrices under GP. If x is a (row or column) vector, then `x[n]` represents the n^{th} component of x, i.e. `compo(x,n)`. It is more natural and shorter to write. If x is a matrix, `x[m,n]` represents the coefficient of row `m` and column `n` of the matrix, `x[m,]` represents the m^{th} *row* of x, and `x[,n]` represents the n^{th} *column* of x.

Finally note that in library mode, the macro `coeff(x,m,n)` exists with exactly the meaning of `x[m,n]` under GP when x is a matrix. This macro should not be confused with the two-variable macro `coeff` used primarily for polynomials and power series under GP.

3.2.8 conj(x) or x_: conjugate of x. The meaning of this is clear, except that for real quadratic numbers, it means conjugation in the real quadratic field. This function has no effect on integers, reals, integermods, fractions or p-adics. The only forbidden type is polymod (see `conjvec` for this).

The library syntax is **gconj**(x).

3.2.9 conjvec(x): conjugate vector representation of x. If x is a polymod `mod(a,q)`, this gives a vector of length degree(q) containing the complex embeddings of the polymod if `q` has integral or rational coefficients, and the conjugates of the polymod if `q` has some integermod coefficients. If x is a integer or a rational number, the result is x. If x is a (row or column) vector, the result is a matrix whose columns are the conjugate vectors of the individual elements of x.

The library syntax is **conjvec**$(x, prec)$, where prec is a C integer.

3.2.10 cvtoi(x): If x is in $\mathbf{R}$, truncates x to an integer. If the result is not significant because the exponent of x is too large compared to its precision, no error occurs, but the number of lost significant bits is put at the address specified in a second argument. This is transparent under GP, but this second argument must be given in library mode. If no digits are lost, this second argument contains the negative of the number of significant bits of the fractional part.

The library syntax is **gcvtoi**$(x, \&e)$ where e is a 32-bit C-integer.

3.2.11 denom(x): lowest denominator of x. The meaning of this is clear when x is a rational number or function. When x is an integer or a polynomial, the result is equal to 1. When x is a vector or a matrix, the lowest common denominator of the components of x is computed. All other types are forbidden.

The library syntax is **denom**(x).

3.2.12 floor(x): floor of x. When x is in $\mathbf{R}$, the result is the largest integer smaller than or equal to x.

The library syntax is **gfloor**(x).

3.2.13 frac(x): fractional part of x. Identical to $x - \text{floor}(x)$. If x is real, the result is in $[0, 1[$.

The library syntax is **gfrac**(x).

3.2.14 getrand$()$: returns the current value of the random number seed. Useful mainly for debugging purposes.

The library syntax is **getrand**$()$. The value returned is a GEN.

3.2.15 getstack$()$: returns the current value of `top-avma`, i.e. the number of bytes used up to now on the stack. Useful mainly for debugging purposes.

The library syntax is **getstack**(). The value returned is a GEN.

3.2.16 gettime(): returns the time (in milliseconds) elapsed since either the last call to `gettime`, or to the beginning of the containing GP instruction (if inside GP), whichever came last.

The library syntax is **gettime**(). The value returned is a GEN.

3.2.17 imag(x): imaginary part of x. When x is a quadratic number, this is the coefficient of ω in the "canonical" integral basis $(1,\omega)$.

The library syntax is **gimag**(x).

3.2.18 length(x): number of non-code words in x really used (i.e. the effective length for integers and polynomials). In particular, the degree of a polynomial is equal to its length minus 1.

The library syntax is **glength**(x).

3.2.19 lift(x): lifts an element $x = a \bmod n$ of $\mathbf{Z}/n\mathbf{Z}$ to a in $\mathbf{Z}$, and similarly lifts a polymod to a polynomial. If x is of type fraction, complex, quadratic, polynomial, power series, rational function, vector or matrix, the lift is done for each coefficient. Forbidden types for x are reals and p-adics. Note that the main variable of the lift of a polymod will be variable number 0 (i.e. 'X') if the rules concerning the creation of polymods explained in chapter 2 are observed.

The library syntax is **lift**(x).

3.2.20 mod(x, y): creates the PARI object ($x \bmod y$), i.e. an integermod or a polymod. y must be an integer or a polynomial. If y is an integer, x must be an integer. If y is a polynomial, x must be a scalar or a polynomial. The result is put on the PARI stack.

This function is not the same as $x\%y$, the result of which is an integer or a polynomial.

The library syntax is **gmodulcp**(x, y).

3.2.21 modp(x, y): same effect as `mod`, except that the created result is put on the heap and not on the stack, and hence becomes a permanent copy which cannot be erased later by garbage collecting (see 4.2.4). In particular, care should be taken to avoid creating too many such objects, since the heap is very small (typically a few thousand objects at most).

The library syntax is **gmodulo**(x, y).

3.2.22 norm(x): algebraic norm of x, i.e. the product of x with its conjugate (no square roots are taken), or conjugates for polymods. For vectors and matrices, the norm is taken componentwise and hence is not the L^2-norm (see `norml2` below). Note that the norm of an element of $\mathbf{R}$ is its square, so as to be compatible with the complex norm.

The library syntax is **gnorm**(x).

3.2.23 norml2(x): square of the L^2-norm of x. x must be a (row or column) vector.

The library syntax is **gnorml2**(x).

3.2.24 numer(x): numerator of x. When x is a rational number or function, the meaning is clear. When x is an integer or a polynomial the result is x itself. When x is a vector or a matrix, then `numer(x)` is defined to be `denom(x)*x`. All other types are forbidden.

The library syntax is **numer**(x).

3.2.25 permutation(n, k): generates the k-th permutation (as a vector of length n) of the numbers 1 to n. The number k is taken modulo $n!$.

The library syntax is **permute**(n, k), where n is a C-integer.

3.2.26 poly(x, v): transform the object x into a polynomial with main variable v. If x is a scalar, this gives a constant polynomial. If x is a power series, the effect is identical to `trunc` (see below), i.e. it chops off the $O(X^k)$. If x is a vector, this function creates the polynomial whose coefficients are given in x, with $x[1]$ being the leading coefficient (which can be zero).

Warning: this is *not* a substitution function. It is intended to be quick and dirty. So if you try `poly(a,y)` on the polynomial `a=x+y`, you will get `y+y`, which is not a valid PARI/GP object.

The library syntax is **gtopoly**(x, v), where v is a variable number.

3.2.27 polyrev(x, v): transform the object x into a polynomial with main variable v. If x is a scalar, this gives a constant polynomial. If x is a power series, the effect is identical to `trunc` (see below), i.e. it chops off the $O(X^k)$. If x is a vector, this function creates the polynomial whose coefficients are given in x, with $x[1]$ being the constant term. Note that this is the reverse of `poly` if x is a vector, otherwise it is identical to `poly`.

The library syntax is **gtopolyrev**(x, v), where v is a variable number.

3.2.28 prec(x, n): x being a PARI object and n a 32-bit C-integer, creates a new object equal to x but with a new precision n. This is to be understood as follows:

For exact types, no change. For x a vector or a matrix, the operation is done componentwise.

For real x, n is the number of desired significant *decimal* digits. If n is smaller than the precision of x, x is truncated, otherwise x is extended with zeros.

For x a p-adic or a power series, n is the desired number of significant p-adic or X-adic digits, where X is the main variable of x.

Note that the function `prec` never changes the type of the result. In particular it is not possible to use `prec` to obtain a polynomial from a power series. For that, see `trunc` (see below).

The library syntax is **gprec**(x, n), where n is a C-integer.

3.2.29 quadgen(x): creates the quadratic number $\omega = (a+\sqrt{x})/2$ where $a = 0$ if $x \equiv 0 \bmod 4$, $a = 1$ if $x \equiv 1 \bmod 4$, so that $(1, \omega)$ is an integral basis for the quadratic order of discriminant x. x must be an integer congruent to 0 or 1 modulo 4.

The library syntax is **quadgen**(x).

3.2.30 quadpoly(x): creates the "canonical" quadratic polynomial corresponding to the discriminant x, i.e. the minimal polynomial of `quadgen(x)`. x must be an integer congruent to 0 or 1 modulo 4.

The library syntax is **quaddisc**(x).

3.2.31 random$()$: gives a random integer between 0 and $2^{31} - 1$. This is simply the standard `rand()` function of C, hence in particular the low order bits are not very random.

The library syntax is **genrand**$()$.

3.2.32 real(x): real part of x. In the case where x is a quadratic number, this is the coefficient of 1 in the "canonical" integral basis $(1, \omega)$.

The library syntax is **greal**(x).

3.2.33 rndtoi(x): same as `cvtoi` (see above) except that truncation is replaced by rounding (see `round` below).

The library syntax is **grndtoi**($x, \&e$).

3.2.34 round(x): If x is in **R**, rounds x to the nearest integer. If the exponent of x is too large compared to its precision, the result is undefined and an error message occurs. Use `rndtoi` above to suppress this error handling.

Important remark: note that, contrary to the other truncation functions (except `rndtoi` which is essentially the same), this function operates on every coefficient at every level of a PARI object. For example

$$\mathrm{trunc}\left(\frac{2.4 * X^2 - 1.7}{1.2 * X}\right) = 2.0 * X,$$

while

$$\mathrm{round}\left(\frac{2.4 * X^2 - 1.7}{1.2 * X}\right) = \frac{2 * X^2 - 2}{X}.$$

An important use of `round` is to get exact results after a long approximate computation, when theory tells you that the coefficients must be integers.

The library syntax is **ground**(x).

3.2.35 rounderror(x): maximum error in decimal digits observed when rounding x. This functions allows the user to easily check the precision of the rounding of a result whose components are supposed to be integers.

The library syntax is **rounderror**(x) and the result is a C-integer.

3.2.36 series(x, v): transform the object x into a power series with main variable v. If x is a scalar, this gives a constant power series with precision given by the global variable `precdl` (transparent under GP). If x is a polynomial, the precision is the greatest of `precdl` and the degree of the polynomial. If x is a vector, the precision is similarly given, and the coefficients of the vector are understood to be the coefficients of the power series starting from the constant term (i.e. the reverse of the function `poly`, see above).

The warning given for `poly` applies here: this is not a substitution function.

The library syntax is **gtoser**(x, v), where v is a variable number.

3.2.37 setprecision(n): sets the current default precision equal to n decimal digits if $n > 0$. In any case, it returns the value of the default precision before the change. Apart from the fact that it returns a value, this is identical to the \precision=n command, with the difference that it can be used inside any GP expression or program.

The library syntax is **setprecr**(n), where n is a C-integer, as is the returned value.

3.2.38 setrand(n): reseeds the random number generator to the value n. The initial seed (in Unix systems at least) is $n = 1$.

The library syntax is **setrand**(n) where n is a C-integer. The value returned is n once again, but as a GEN.

3.2.39 setserieslength(n): sets the current default series length equal to n if $n > 0$. In any case, it returns the value of the default length before the change. Apart from the fact that it returns a value, this is identical to the \serieslength=n command, with the difference that it can be used inside any GP expression or program.

The library syntax is **setserieslength**(n), where n is a C-integer, as is the returned value.

3.2.40 settype(x, t): makes a copy of x and sets its type equal to type number t. This function must be used with extreme caution, otherwise disasters may occur, but one instance where it is extremely useful is `settyp(x,14)` when x is a rational function (type 13). In this case, the created object, as well as the objects created from it, will not be reduced automatically, making the operations much faster. In fact this function is the *only* way to create reducible rationals (type 5) or rational functions (type 14) in GP.

The library syntax is **gsettype**(x, t), where t is a long, but need not be used since the internal function `settyp` is available. Note that `settyp` does *not* create a copy of x, while `gsettype` does.

3.2.41 simplify(x): this function tries to simplify the object x as much as it can. The simplifications do not concern rational functions (which Pari automatically tries to simplify), but type changes. Specifically, a complex or quadratic number whose imaginary part is exactly equal to 0 (i.e. not a real zero) is converted to its real part, and a polynomial of degree zero is converted to its constant term. For all types, this of course occurs recursively. This function is useful in any case, but in particular before the use of arithmetic functions which expect integer arguments, and not for example a complex number of 0 imaginary part an integer real part (which is however printed as an integer).

The library syntax is **simplify**(x).

3.2.42 size(x): gives the maximal number of decimal digits minus one of the components of x.

The library syntax is **size**(x), and the result is a C-integer.

3.2.43 trunc(x): truncation of x. When x is in $\mathbf{R}$, this means that the part after the decimal point is chopped away. If the result is not significant because the exponent is too large compared to the precision, an error occurs. Use `cvtoi` (see above) to suppress this error handling.

Note a very special use of `trunc`: when applied to a power series, it transforms it into a polynomial or a rational function with denominator a power of X, by chopping away the $O(X^k)$. Similarly, when applied to a p-adic number, it transforms it into an integer or a rational number by chopping away the $O(p^k)$.

The library syntax is **gtrunc**(x).

3.2.44 type(x): internal type number (from 1 to 19) of the pari object x. This is useful only under GP. See also the function `settype`.

The library syntax is **gtype**(x), but need not be used since the internal function `typ` is available. Note that `gtype` gives a GEN while `typ` gives a C-integer.

3.2.45 valuation(x, p): Computes the highest exponent of p dividing x. If p is of type integer, x must be an integer, an integermod whose modulus is divisible by p, a fraction, a q-adic number with $q = p$, or a polynomial or power series in which case the valuation is the minimum of the valuation of the coefficients.

If p is of type polynomial, x must be of type polynomial or rational function, and also a power series if x is a monomial. If $x = 0$, an error is issued except in case of p-adic numbers and power series, in which case the result is the exponent of the zero. Finally, the valuation of a vector, complex or quadratic number is the minimum of the component valuations. Any other type combinations gives an error.

The library syntax is **ggval**(x).

3.2.46 vec(x): transform the object x into a vector. The vector will be with one component only, except when x is a vector/matrix or a quadratic form (in which case the resulting vector is simply the initial object considered as a row vector), but more importantly when x is a polynomial or a power series. In the case of a polynomial, the coefficients of the vector starts with the leading coefficient of the polynomial, while for power series only the significant coefficients are taken into account, but this time by increasing order of degree.

The library syntax is **gtovec**(x).

3.2.47 vecmax(x): if x is a vector or a matrix, returns the maximum of the elements of x, otherwise returns a copy of x. Returns zero if x is empty (as it should).

The library syntax is **vecmax**(x).

3.2.48 vecmin(x): if x is a vector or a matrix, returns the minimum of the elements of x, otherwise returns a copy of x. Gives an error if x is empty.

The library syntax is **vecmin**(x).

3.3 Transcendental functions.

As a general rule, which of course in some cases may have exceptions, transcendental functions operate in the following way:

If the argument is either an integer, a real, a rational, a complex or a quadratic number, it is, if necessary, first converted to a real (or complex) number using the current precision held in the variable **prec**. Under GP this is transparent to the user, but when programming in library mode, care must be taken to supply the parameter `prec` as the last argument of the function if the first argument is an exact object (see 1.2.5.), otherwise disaster will occur.

Then, the function is computed. Note that even if the argument is real, the result may be complex (e.g. acos(2.0) or acosh(0.0)). Note also that the principal branch is always chosen.

If the argument is an integermod or a p-adic, at present only a few functions like `sqrt` (square root), `sqr` (square), `log`, `exp`, powering, `teich` (Teichmüller character) and `agm` (arithmetic-geometric mean) are implemented. Note that in the case of a 2-adic number, `sqr`(x) is not identical to $x * x$: for example if $x = 1 + O(2^5)$ then $x * x = 1 + O(2^5)$ while `sqr`(x) $= 1 + O(2^6)$. (Remark: note that if we wanted to be strictly consistent with the PARI philosophy, we should have $x * y = (4 \bmod 8)$ when both x and y are congruent to 2 modulo 4, or `sqr`(x) $= (4 \bmod 32)$ when x is congruent to 2 modulo 4. However, since an integermod is an exact object, PARI assumes that the modulus must not change, and the result is hence $0 \bmod 4$ in both cases. On the other hand, p-adics are not exact objects, hence are treated differently.)

If the argument is a polynomial, power series or rational function, it is, if necessary, first converted to a power series using the current precision held in the variable **precdl**. Under GP this again is transparent to the user. When programming in library mode, however, the global variable `precdl` must be set before calling the function if the argument has an exact type (i.e. not a power series). Here `precdl` is not an argument of the function, but a global variable.

Then the Taylor series expansion of the function around $X = 0$ (where X is the main variable) is computed to a number of terms depending on the number of terms of the argument and the function being computed.

If the argument is a vector or a matrix, the result is componentwise evaluation of the function. In particular, transcendental functions on square matrices, which are not implemented in the present version 1.38.1 (see Appendix C however), will have a slightly different name if they are implemented some day.

3.3.1 ^ x^y: if y is not of type integer, this has the same effect as $\exp(y * \ln(x))$. It can be applied to p-adic numbers as well as to the more usual types.

The library syntax is **gpui**$(x, y, prec)$.

3.3.2 abs(x): absolute value of x (modulus if x is complex). Polynomials, power series and rational functions are not allowed.

The library syntax is **gabs**$(x, prec)$ where the second argument $prec$ is necessary only in the complex or imaginary quadratic case.

3.3.3 acos(x): principal branch of $\cos^{-1}(x)$, i.e. such that $\mathrm{Re}(\mathrm{acos}(x)) \in [0, \pi]$. If $x \in \mathbf{R}$ and $|x| > 1$ then $\mathrm{acos}(x)$ is complex.

The library syntax is **gacos**$(x, prec)$.

3.3.4 acosh(x): principal branch of $\cosh^{-1}(x)$, i.e. such that $\mathrm{Im}(\mathrm{acosh}(x)) \in [0, \pi]$. If $x \in \mathbf{R}$ and $x < 1$ then $\mathrm{acosh}(x)$ is complex.

The library syntax is **gach**$(x, prec)$.

3.3.5 agm(x, y): arithmetic-geometric mean of x and y. In the case of complex or negative numbers, the principal square root is always chosen. p-adic or power series arguments are also allowed. Note that a p-adic agm exists only if x/y is congruent to 1 modulo p (modulo 16 for $p = 2$). x and y cannot both be vectors or matrices.

The library syntax is **agm**$(x, y, prec)$.

3.3.6 arg(x): argument of the complex number x, such that $-\pi < \arg(x) \leq \pi$.

The library syntax is **garg**$(x, prec)$.

3.3.7 asin(x): principal branch of $\sin^{-1}(x)$, i.e. such that $\mathrm{Re}(\mathrm{asin}(x)) \in [-\pi/2, \pi/2]$. If $x \in \mathbf{R}$ and $|x| > 1$ then $\mathrm{asin}(x)$ is complex.

The library syntax is **gasin**$(x, prec)$.

3.3.8 asinh(x): principal branch of $\sinh^{-1}(x)$, i.e. such that $\mathrm{Im}(\mathrm{asinh})(x) \in [-\pi/2, \pi/2]$.

The library syntax is **gash**$(x, prec)$.

3.3.9 atan(x): principal branch of $\tan^{-1}(x)$, i.e. such that $\mathrm{Re}(\mathrm{atan}(x)) \in]-\pi/2, \pi/2[$.

The library syntax is **gatan**$(x, prec)$.

3.3.10 atanh(x): principal branch of $\tanh^{-1}(x)$. i.e. such that $\mathrm{Im}(\mathrm{atanh}(x)) \in]-\pi/2, \pi/2]$. If $x \in \mathbf{R}$ and $|x| > 1$ then $\mathrm{atanh}(x)$ is complex.

The library syntax is **gath**$(x, prec)$.

3.3.11 bernreal(x): Bernoulli number B_x, where $B_0 = 1$, $B_1 = -1/2$, $B_2 = 1/6, \ldots$, expressed as a real number with the current precision.

The library syntax is **bernreal**$(x, prec)$.

3.3.12 bernvec(x): creates a vector containing, as rational numbers, the Bernoulli numbers B_0, $B_2, \ldots$, B_{2x}. These Bernoulli numbers can then be used as follows. Assume that this vector has been put into a variable, say b. Then we define under GP:

```
bern(x)=if(x==1,-1/2,if((x<0)||(x%2),0,b[x/2+1]))
```

and then `bern(k)` gives the Bernoulli number of index k as a rational number, exactly as `bernreal(k)` gives it as a real number.

The library syntax is **bernvec**(x).

3.3.13 cos(x): cosine of x.

The library syntax is **gcos**$(x, prec)$.

3.3.14 cosh(x): hyperbolic cosine of x.

The library syntax is **gch**$(x, prec)$.

3.3.15 dilog(x): principal branch of the dilogarithm of x, i.e. analytic continuation of the power series $\log_2(x) = \sum_{n\geq 1} x^n/n^2$.

The library syntax is **dilog**$(x, prec)$.

3.3.16 eint1(x): exponential integral $\int_x^\infty \frac{e^{-t}}{t}\,dt$.

The library syntax is **eint1**$(x, prec)$.

3.3.17 erfc(x): complementary error function $(2/\sqrt{\pi})\int_x^\infty e^{-t^2}\,dt$.

The library syntax is **erfc**$(x, prec)$.

3.3.18 eta(x): Dedekind's η function, without the $q^{1/24}$. The meaning is this. If x is a complex number with positive imaginary part, the result is $\prod_{n=1}^\infty (1 - q^n)$, where $q = e^{2i\pi x}$. If x is a power series (or can be converted to a power series) with positive valuation, the result is $\prod_{n=1}^\infty (1 - x^n)$.

The library syntax is **eta**$(x, prec)$.

3.3.19 euler: Euler's constant $0.57721\cdots$.

Note that `euler` is one of the few special reserved names which cannot be used for variables.

The library syntax is **mpeuler**$(prec)$ where *prec must* be given. Note that this creates γ on the PARI stack. If one does not want to create it on that stack but be able to use it later under the global name **geuler** (with no parentheses) use instead **consteuler**$(prec)$.

3.3.20 exp(x): exponential of x. p-adic arguments with positive valuation are accepted.

The library syntax is **gexp**$(x, prec)$.

3.3.21 gamh(x): gamma function evaluated at the argument $x+1/2$. When x is an integer, this is much faster than using gamma($x+1/2$).

The library syntax is **ggamd**($x, prec$).

3.3.22 gamma(x): gamma function of x. In the present version 1.38.1 the p-adic gamma function is not yet implemented.

The library syntax is **ggamma**($x, prec$).

3.3.23 hyperu(a, b, x): U-confluent hypergeometric function with parameters a and b.

The library syntax is **hyperu**($a, b, x, prec$).

3.3.24 incgam(x, y): incomplete gamma function.

The arguments x and y must be positive. The result returned is $\int_y^\infty e^{-t}t^{x-1}\,dt$.

The library syntax is **incgam**($x, y, prec$).

Note: in addition, there exist also the functions `incgam1 and incgam2` which are used for internal purposes.

3.3.25 incgam3(x, y): complementary incomplete gamma function.

The arguments x and y must be positive. The result returned is $\int_0^y e^{-t}t^{x-1}\,dt$. when y is not too large.

The library syntax is **incgam3**($x, y, prec$).

3.3.26 incgam4(x, y, z): incomplete gamma function with given gamma value. This function assumes (of course without checking) that $z=\Gamma(x)$. The arguments x and y must be positive. The result returned is $\int_y^\infty e^{-t}t^{x-1}\,dt$, as in the ordinary `incgam` function, but the computation will be much faster when y is small.

The library syntax is **incgam4**($x, y, z, prec$).

3.3.27 jbesselh(n, x): J-Bessel function of half integral index. More precisely, `jbesselh(n,x)` computes $J_{n+1/2}(x)$ where n must be of type integer, and x is any element of $\mathbf{C}$. In the present version 1.38.1, this function is not very accurate when x is small.

The library syntax is **jbesselh**($n, x, prec$).

3.3.28 jell(x): elliptic j-invariant. x must be a complex number with positive imaginary part, or convertible into a power series or a p-adic number with positive valuation.

The library syntax is **jell**($x, prec$).

3.3.29 kbessel(nu, x): K-Bessel function of index nu (which can be complex) and argument x. Only real and positive arguments x are allowed in the present version 1.38.1.

The library syntax is **kbessel**($nu, x, prec$).

Note: in addition, another implementation of this function which is often faster than kbessel is the function **kbessel2**.

3.3.30 ln(x) or **log**(x): principal branch of the natural logarithm of x, i.e. such that $\mathrm{Im}(\ln(x))\in]-\pi,\pi]$. The result is complex (with imaginary part equal to π) if $x\in\mathbf{R}$ and $x<0$.

p-adic arguments are also accepted for x, with the convention that $\ln(p) = 0$. Hence in particular $\exp(\ln(x))/x$ will not in general be equal to 1 but to a $p-1$-th root of unity (or ± 1 if $p = 2$) times a power of p.

The library syntax is **glog**$(x, prec)$.

3.3.31 lngamma(x): principal branch of the logarithm of the gamma function of x. Can have much larger arguments than `gamma` itself. In the present version 1.38.1 the p-adic lngamma function is not yet implemented.

The library syntax is **glngamma**$(x, prec)$.

3.3.32 logagm(x): principal branch of the natural logarithm of x, computed using an agm formula suggested by Mestre, when x is real, otherwise identical to `log`.

The library syntax is **glogagm**$(x, prec)$.

3.3.33 pi: the constant pi $(3.14159\cdots)$.

The library syntax is **mppi**$(prec)$ where *prec must* be given. Note that this creates π on the PARI stack. If one does not want to create it on that stack but be able to use it later under the global name **gpi** (with no parentheses) use instead **constpi**$(prec)$.

3.3.34 polylog(m, x): m^{th} polylogarithm of x, i.e. analytic continuation of the power series $\log_m(x) = \sum_{n\geq 1} x^n/n^m$. The program uses the power series when $|x|^2 \leq 1/2$, and the power series expansion in $\widetilde{log}(x)$ otherwise. It is valid in a large domain (at least $|x| < 230$), but should not be used too far away from the unit circle since it is then better to use the functional equation linking the value at x to the value at $1/x$, which takes a trivial form for the `polylogd` and `polylogp` functions below. Power series, polynomial, rational and vector/matrix arguments are allowed.

The library syntax is **gpolylog**$(m, x, prec)$.

3.3.35 polylogd(m, x): modified m^{th} polylogarithm of x, called $\tilde{D}_m(x)$ in Zagier, defined for $|x| \leq 1$ by

$$\Re_m\left(\sum_{k=0}^{m-1} \frac{(-\log|x|)^k}{k!}\log_{m-k}(x) + \frac{(-\log|x|)^{m-1}}{m!}\log|1-x|\right)$$

and such that $D_m(1/x) = (-1)^{m-1}D_m(x)$, where $\Re_m$ denotes $\Re$ or $\Im$ depending whether m is odd or even.

The library syntax is **polylogd**$(m, x, prec)$.

3.3.36 polylogdold(m, x): modified m^{th} polylogarithm of x, called $D_m(x)$ in Zagier, defined for $|x| \leq 1$ by

$$\Re_m\left(\sum_{k=0}^{m-1} \frac{(-\log|x|)^k}{k!}\log_{m-k}(x) - \frac{1}{2}\frac{(-\log|x|)^m}{m!}\right)$$

and such that $D_m(1/x) = (-1)^{m-1}D_m(x)$, where $\Re_m$ denotes $\Re$ or $\Im$ depending whether m is odd or even.

The library syntax is **polylogdold**$(m, x, prec)$.

3.3.37 polylogp(m,x): another modified m^{th} polylogarithm of x, called $P_m(x)$ in Zagier, defined for $|x| \le 1$ by

$$\Re_m\left(\sum_{k=0}^{m-1}\frac{2^k B_k}{k!}(\log|x|)^k \log_{m-k}(x) - \frac{2^{m-1}B_m}{m!}(\log|x|)^m\right)$$

and such that $P_m(1/x) = (-1)^{m-1}P_m(x)$, where $\Re_m$ denotes $\Re$ or $\Im$ depending whether m is odd or even.

The library syntax is **polylogp**$(m, x, prec)$.

3.3.38 psi(x): the ψ-function of x, i.e. the logarithmic derivative $\Gamma'(x)/\Gamma(x)$.

The library syntax is **gpsi**$(x, prec)$.

3.3.39 sin(x): sine of x.

The library syntax is **gsin**$(x, prec)$.

3.3.40 sinh(x): hyperbolic sine of x.

The library syntax is **gsh**$(x, prec)$.

3.3.41 sqr(x): square of x. Not identical to $x*x$ in the case of 2-adics (see above), and acts componentwise on vectors and matrices. In particular, if x is a square matrix, $\mathrm{sqr}(x)$ is not the square of x (use instead $x*x$ or x^2).

The library syntax is **gsqr**(x).

3.3.42 sqrt(x): principal branch of the square root of x, i.e. such that $\mathrm{Arg}(\mathrm{sqrt}(x)) \in]-\pi/2,\pi/2]$, or in other words such that $\mathrm{Re}(\mathrm{sqrt}(x)) > 0$ or $\mathrm{Re}(\mathrm{sqrt}(x)) = 0$ and $\mathrm{Im}(\mathrm{sqrt}(x)) >= 0$. Integermod a prime and p-adics are allowed as arguments. If $x \in \mathbf{R}$ and $x < 0$ the result is complex with positive imaginary part.

The library syntax is **gsqrt**$(x, prec)$.

3.3.43 tan(x): tangent of x.

The library syntax is **gtan**$(x, prec)$.

3.3.44 tanh(x): hyperbolic tangent of x.

The library syntax is **gth**$(x, prec)$.

3.3.45 teich(x): Teichmuller character of the p-adic number x.

The library syntax is **teich**(x).

3.3.46 wf(x): Weber's f function, i.e. such that $j = (f^{24}-16)^3/f^{24}$.

The library syntax is **wf**$(x, prec)$.

3.3.47 wf2(x): Weber's f_2 function, i.e. such that $j = (f_2^{24}+16)^3/f_2^{24}$.

The library syntax is **wf2**$(x, prec)$.

3.3.48 zeta(s): Riemann's zeta function $\zeta(s) = \sum_{n\geq 1} n^{-s}$, computed using the Euler-Maclaurin summation formula, except when s is of type integer, in which case it is computed using Bernoulli numbers for $s <= 0$ or $s > 0$ and even, and using modular forms for $s > 0$ and odd.

The library syntax is **gzeta**$(s, prec)$.

3.4 Arithmetic functions.

These functions are by definition functions whose natural domain of definition is either **Z** (or **N**), or sometimes polynomials over a base ring. Functions which concern polynomials exclusively will be explained in the next section. The way these functions are used is completely different from transcendental functions: in general only the types integer and polynomial are accepted as arguments.

In the present version 1.38.1 a primitive but useful version of the ECM method has been implemented. Also, numbers found to be pseudo-primes after 10 successful trials of the Rabin-Miller test are declared primes.

3.4.1 bestappr(x, k): if $x \in \mathbf{R}$, finds the best rational approximation to x with denominator at most equal to k using continued fractions. If x is a generic type, `bestappr` is used on each coefficient independently.

The library syntax is **bestappr**(x, k).

3.4.2 bezout(x, y): finds u and v minimal in a natural sense such that $x*u+y*v = \gcd(x, y)$. The arguments must be both integers or both polynomials, and the result is a row vector with three components u, v, and $\gcd(x, y)$.

The library syntax is either **vecbezout**(x, y) to get the vector, or **gbezout**$(x, y, \&u, \&v)$ which gives as result the address of the created gcd, and puts in u and v the addresses of the corresponding created objects.

3.4.3 bigomega(x): number of prime divisors of x counted with multiplicity. x must be an integer, and the result is a 32-bit C-integer.

The library syntax is **bigomega**(x).

3.4.4 bin(x, y): binomial coefficient $\binom{x}{y}$. Here y must be an integer, but x can be any PARI object.

The library syntax is **binome**(x, y), where y must be a C-integer.

3.4.5 boundcf$(x, lmax)$: creates the row vector whose components are the partial quotients of the continued fraction expansion of x, the number of partial quotients being limited to lmax. If x is a real number, the expansion stops at the last significant partial quotient or after lmax terms, whichever is smallest. x can also be a rational function or a power series.

The library syntax is **gboundcf**$(x, lmax)$, where $lmax$ is a C integer.

3.4.6 boundfact(x, p): For integer x, finds only the prime factors up to p or to the default bound of the prime table created upon initialization of GP, whichever is lowest (except when $p <= 1$ where the effect is identical to `smallfact`. The remaining part is hence not necessarily prime.

The library syntax is **boundfact**(x, p).

3.4.7 buchimag$(D, c, c2, t)$: McCurley's sub-exponential algorithm for computing the class group of an imaginary quadratic field of discriminant D. Should be used instead of `classno` for $|D| > 10^{25}$ or when the *structure* is wanted. The numbers c and $c2$ are positive real numbers which control the execution time and the stack size. To get maximum speed, set $c2 = c$. To get a rigorous result (under GRH) you must take $c2 = 6$. Reasonable values for c are between 0.1 and 2. t is the number of "extra relations" that we want to use in McCurley's algorithm (see below). We must have $t \geq 1$, and usually $t = 10$ is sufficient.

The result of this function is a vector with 4 components. The first is the class number, the second is a vector giving the structure of the class group as a product of cyclic groups, and the third is a vector giving as binary quadratic forms generators of those cyclic groups. The last component is a measure of the correctness of the result. If it is close to 1, the result is correct (under GRH). If it is close to a larger integer, this shows that the class number is off by a factor equal to this integer, and you must start again with a larger value for t or c or a different random seed. In this case, a warning message is printed.

Note: some installations may have this function disabled.

The library syntax is **buchimag**$(D, c, c2, t)$.

3.4.8 buchreal$(D, n, c, c2, t)$: Buchmann's sub-exponential algorithm for computing the class group and the regulator of a real quadratic field of discriminant D. Should be used instead of `classno` and `regula` for $D > 10^{10}$ or when the *structure* is wanted. If the integer n is equal to 0 the function computes the ordinary (or wide) class group and regulator, while if n is non-zero it computes the narrow class group and regulator (this case is not yet implemented in version 1.38.1). The numbers c and $c2$ are positive real numbers which control the execution time and the stack size. To get maximum speed, set $c2 = c$. To get a rigorous result (under GRH) you must take $c2 = 6$. Reasonable values for c are between 0.1 and 2. t is the number of "extra relations" that we want to use in Buchmann-McCurley's algorithm (see below). We must have $t \geq 2$, and usually $t = 10$ is sufficient.

The result of this function is a vector with 5 components. The first is the class number, the second is a vector giving the structure of the class group as a product of cyclic groups, and the third is a vector giving as binary quadratic forms generators of those cyclic groups. The fourth component is the regulator, computed to an accuracy which is the maximum of an internal accuracy determined by the program and the current default (note that once the regulator is known to a small accuracy it is trivial to compute it to very high accuracy. I leave this as an exercise to the reader, or see the tutorial). The last component is a measure of the correctness of the result. If it is close to 1, the results are correct (under GRH). If it is close to a larger integer, this shows that the product of the class number by the regulator is off by a factor equal to this integer, and you must start again with a larger value for t or c or a different random seed. In this case, a warning message is printed.

Note: some installations may have this function disabled.

The library syntax is **buchreal**$(D, n, c, c2, t)$.

3.4.9 cf(x): creates the row vector whose components are the partial quotients of the continued fraction expansion of x. If x is a real number, the expansion stops at the last significant partial quotient. x can also be a rational function.

The library syntax is **gcf**(x).

3.4.10 cf2(b, x): creates the row vector whose components are the partial quotients of the continued fraction expansion of x, the numerators being equal to the coefficients of the vector b. The length of the result is equal to the length of b unless a partial remainder is encountered which is equal to zero, in which case the expansion stops. In the case of real numbers, the stopping criterion in thus different from that of `cf` since if b is too long, some partial quotients may not be significant. x can also be a rational function or a power series.

The library syntax is **gcf2**(b, x).

3.4.11 chinese(x, y): x and y being both integermods or polymods, creates (with the same type) a z in the same residue class as x and in the same residue class as y, if it is possible.

The library syntax is **chinois**(x, y).

3.4.12 classno(x): class number of the quadratic field of discriminant x. In the present version 1.38.1, a simple algorithm is used for $x > 0$, so x should not be too large (say $x < 10^7$) for the time to be reasonable. On the other hand, for $x < 0$ one can reasonably compute classno(x) for $|x| < 10^{25}$, since the method used is Shanks' method which is in $O(|x|^{1/4})$. For larger values of $|D|$, see `buchimag` for $D < 0$ and `buchreal` for $D > 0$.

The library syntax is **classno**(x).

There also exists the function `classno2`, which computes the class number using the functional equation. However, it is in $O(|x|^{1/2})$. Finally, in library mode only there exists the function `classno3` which computes the class number of an imaginary quadratic field by counting reduced forms, a $O(|x|)$ algorithm. See also `hclassno` below.

3.4.13 compimag(x, y) composition of the binary quadratic forms x and y. This is of course a special case of `gmul(x,y)`. See also `nucomp` and `nudupl`.

The library syntax is **compimag**(x, y).

3.4.14 comprealraw(x, y) composition of the binary quadratic forms x and y with positive discriminant, without reduction of the result. This is useful e.g. to compute a generating element of an ideal.

The library syntax is **comprealraw**(x, y).

3.4.15 content(x): computes the gcd of all the coefficients of x, when this gcd makes sense. If x is a scalar, this simply gives x. If x is a polynomial (and by extension a power series), it gives the usual content of x. If x is a rational function, it gives the ratio of the contents of the numerator and the denominator. Finally, if x is a vector or a matrix, it gives the gcd of all the entries.

The library syntax is **content**(x).

3.4.16 divisors(x): creates a row vector whose components are the positive divisors of the integer x in increasing order.

The library syntax is **divisors**(x).

3.4.17 fact(x) or $x!$: factorial of x. The expression $x!$ gives a result which is an integer, while fact(x) gives a real number.

The library syntax is **mpfact**(x) for $x!$ and **mpfactr**$(x, prec)$ for fact(x). x must be a 32-bit C-integer and not a PARI integer.

3.4.18 factfq(x, p, a): factorization in the field $\mathbf{F}_q$ defined by the irreducible polynomial a over $\mathbf{F}_p$ of the polynomial x. The coefficients of x must be operation-compatible with $\mathbf{Z}/p\mathbf{Z}$. The result is a two column matrix, the first column being the irreducible polynomials dividing x, and the second the exponents. It is advised to use for the variable of a (which will be used as variable of a polymod), a name distinct from the other variables used, so that a `lift()` of the result will be legible.

The library syntax is **factmod9**(x, p, a).

3.4.19 factmod(x, p): factorization mod p of the polynomial x. The coefficients of x must be operation-compatible with $\mathbf{Z}/p\mathbf{Z}$. The result is a two column matrix, the first column being the irreducible polynomials dividing x, and the second the exponents. If you want only the *degrees* of the irreducible polynomials (for example for computing an L-function), use the function `simplefactmod`.

The library syntax is **factmod**(x, p).

3.4.20 factor(x): general factorization function. If x is of type integer, rational, polynomial or rational function, the result is a two column matrix, the first column being the irreducibles dividing x (prime numbers or polynomials), and the second the exponents. If x is a vector or a matrix, the factoring is done componentwise (hence the result is a vector or matrix of two column matrices).

The polynomials or rational functions to be factored must have coefficients either integers or integermods. Note that PARI does *not* yet know how to factor multivariate polynomials.

The library syntax is **factor**(x). See also `factpol` `factpol2` and `factornf`.

3.4.21 fibo(x): x^{th} Fibonacci number.

The library syntax is **fibo**(x). x must be a 32-bit C-integer and not a PARI integer.

3.4.22 gcd(x, y): creates the greatest common divisor of x and y. x and y can be of quite general types; for instance both rational numbers. Vector/matrix types are also accepted. Note that for these types, `gcd` is not commutative.

The library syntax is **ggcd**(x, y).

3.4.23 hclassno(x): Hurwitz class number of x, where x is nonnegative and congruent to 0 or 3 modulo 4. See also `classno` above.

The library syntax is **hclassno**(x).

3.4.24 hilb(x, y, p) or **hilbp**(x, y): Hilbert symbol of x and y. If x and y are of type integer or fraction, the function `hilb` must be used with an explicit third parameter p, $p = 0$ meaning the place at infinity. Otherwise, p need not be given, and x and y can be of compatible types integer, fraction, real, integermod or p-adic.

The library syntax is in both cases **hil**(x, y, p) or **hil**(x, y).

3.4.25 isfund(x): true (1) if x is equal to 1 or to the discriminant of a quadratic field, false (0) otherwise.

The library syntax is **isfundamental**(x).

3.4.26 isprime(x): true (1) if x is a strong pseudo-prime for 10 randomly chosen bases, false (0) otherwise.

The library syntax is **isprime**(x).

3.4.27 ispsp(x): true (1) if x is a strong pseudo-prime for a randomly chosen base, false (0) otherwise.

The library syntax is **ispsp**(x).

3.4.28 isqrt(x): integer square root of x, which must be of PARI type integer. A negative x is allowed, and the result in that case is `i*isqrt(-x)`.

The library syntax is **racine**(x).

3.4.29 issqfree(x): true (1) if x is squarefree, false if not. Here x can be an integer or a polynomial.

The library syntax is **issquarefree**(x).

3.4.30 issquare(x): true (1) if x is square, false if not. x can be of any type (for vector/matrix types the function is applied to the components).

The library syntax is **gcarreparfait**(x).

3.4.31 kro(x, y): Kronecker (i.e. generalized Legendre) symbol $\left(\frac{x}{y}\right)$. x and y must be of type integer and the result (0 or ± 1) is a 32-bit C-integer.

The library syntax is **kronecker**(x, y).

3.4.32 lcm(x, y): least common multiple of x and y, i.e. such that $\mathrm{lcm}(x,y) * \mathrm{gcd}(x,y) = x * y$.

The library syntax is **glcm**(x, y).

3.4.33 mu(x): Möbius μ-function of x. x must be of type integer and the result (0 or ± 1) is a 32-bit C-integer.

The library syntax is **mu**(x).

3.4.34 nextprime(x): finds the smallest prime greater than or equal to x. x must be of type integer.

The library syntax is **bigprem**(x).

3.4.35 nucomp(x, y, l) composition of the primitive positive definite binary quadratic forms x and y using the NUCOMP and NUDUPL algorithms of Shanks (a la Atkin). l is any positive constant, but for optimal speed, one should take $l = |D|^{1/4}$ where D is the common discriminant of x and y.

The library syntax is **nucomp**(x, y, l). The auxiliary function **nudupl**(x, l) also exists when $x = y$.

3.4.36 numdiv(x): number of divisors of x. x must be of type integer, and the result is a 32-bit C-integer.

The library syntax is **numbdiv**(x).

3.4.37 nupow(x, n): n-th power of the primitive positive definite binary quadratic form x using the NUCOMP and NUDUPL algorithms (see `nucomp` above).

The library syntax is **nupow**(x, n).

3.4.38 omega(x): number of distinct prime divisors of x. x must be of type integer, and the result is a 32-bit C-integer.

The library syntax is **omega**(x).

3.4.39 order(x): x must be an integer mod n, and the result is the order of x in the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$. Error if x is not invertible.

The library syntax is **order**(x).

3.4.40 pf(x, p): prime binary quadratic form of discriminant x whose first coefficient is the prime number p. Error if x is not a quadratic residue mod p. In the case where $x > 0$, the "distance" component of the form is set equal to zero to the current precision.

The library syntax is **primeform**$(x, p, prec)$, where the third variable $prec$ is a C-integer, but is necessary only when $x > 0$.

3.4.41 phi(x): Euler's ϕ-function of x. x must be of type integer.

The library syntax is **phi**(x).

3.4.42 pnqn(x): When x is a vector or a one row matrix, x is considered as the list of partial quotients $[a_0, a_1, \ldots, a_n]$ of a rational number, and the result is the 2 by 2 matrix $[p_n, p_{n-1}; q_n, q_{n-1}]$ in the standard notation of continued fractions, so $p_n/q_n = a_0 + 1/(a_1 + \ldots + 1/a_n)\ldots)$. If x is a matrix with two rows $[b_0, b_1, \ldots, b_n]$ and $[a_0, a_1, \ldots, a_n]$, this is then considered as a generalized continued fraction and we have similarly $p_n/q_n = 1/b_0(a_0 + b_1/(a_1 + \ldots + b_n/a_n)\ldots)$. Note that in this case one usually has $b_0 = 1$.

The library syntax is **pnqn**(x).

3.4.43 powrealraw(x, n) n-th power of the binary quadratic form x of positive discriminant, computed without doing any reduction (i.e. using `comprealraw` above). Here n must be non-negative and $n < 2^{31}$.

The library syntax is **powrealraw**(x, n) where n must be a C-integer.

3.4.44 prime(x): the x^{th} prime number.

The library syntax is **prime**(x). x must be a 32-bit C-integer.

3.4.45 primes(x): creates a row vector whose components are the first x prime numbers.

The library syntax is **primes**(x). x must be a 32-bit C-integer.

3.4.46 primroot(x): returns a primitive root of x, where x is a prime power.

The library syntax is **gener**(x).

3.4.47 qfi(a, b, c): creates the binary quadratic form $ax^2 + bxy + cy^2$ with $b^2 - 4ac < 0$.

The library syntax is **qfi**(a, b, c).

3.4.48 qfr(a, b, c, d): creates the binary quadratic form $ax^2 + bxy + cy^2$ with $b^2 - 4ac > 0$ and distance function d.

The library syntax is **qfr**(a, b, c, d).

3.4.49 quaddisc(x): discriminant of the quadratic field $\mathbf{Q}(\sqrt(x))$, where $x \in \mathbf{Q}$. Up to trivial 2-factors, this function can also be used to compute the squarefree part of x.

The library syntax is **quaddisc**(x).

3.4.50 redimag(x): reduce the binary quadratic form x with negative discriminant.

The library syntax is **redimag**(x).

3.4.51 redreal(x): reduce the binary quadratic form x with positive discriminant.

The library syntax is **redreal**(x).

3.4.52 redrealnod(x, sq): reduce the binary quadratic form x with positive discriminant D without computing the distance function, where sq is the integer square root of D (no checking is done of this fact).

The library syntax is **redrealnod**(x, sq).

3.4.53 regula(x): regulator of the quadratic field of positive discriminant x. Error if x is not a discriminant (fundamental or not) or if x is a square. See also `buchreal` if x is large.

The library syntax is **regula**$(x, prec)$.

3.4.54 rhoreal(x): perform one reduction step on the binary quadratic form x with positive discriminant.

The library syntax is **rhoreal**(x).

3.4.55 rhorealnod(x, sq): perform one reduction step on the binary quadratic form x with positive discriminant D without computing the distance function, where sq is the integer square root of D (no checking is done of this fact).

The library syntax is **rhorealnod**(x, sq).

3.4.56 sigma(x): sum of the positive divisors of x. x must be of type integer.

The library syntax is **sumdiv**(x).

3.4.57 sigmak(k, x): sum of the k^{th} powers of the positive divisors of x. x must be of type integer, but (in library mode) k must be a 32-bit C-integer.

The library syntax is **sumdivk**(k, x).

3.4.58 simplefactmod(x, p): factorization mod p of the polynomial x. The coefficients of x must be operation-compatible with $\mathbf{Z}/p\mathbf{Z}$. The result is a two column matrix, the first column being the degrees of the irreducible polynomials dividing x, and the second the exponents. If you want the irreducible polynomials themselves, use the function `factmod`.

The library syntax is **simplefactmod**(x, p).

3.4.59 smallfact(x): For integer x, finds only the prime factors up to the default table created when GP was started (usually 500000). The remaining part is hence not necessarily prime. This never takes more than a few seconds, and is particularly advantageous with respect to `factor` when one needs only the small prime divisors and not the complete factorization.

The library syntax is **smallfact**(x).

3.4.60 unit(x): fundamental unit of the real quadratic field $\mathbf{Q}(\sqrt{x})$ where x is the positive discriminant of the field. If x is not a fundamental discriminant, this probably gives the fundamental unit of the corresponding order. x must be of type integer, and the result is a quadratic number.

The library syntax is **fundunit**(x).

3.5 Functions related to elliptic curves.

We have implemented a number of functions which are useful for number theorists working on elliptic curves. We always use Tate's notations.

The functions assume that the curve is given by a general Weierstrass model

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

where a priori the a_i can be of any scalar type. This curve can be considered as a five component vector `e=[a1,a2,a3,a4,a6]`, but for most functions it is useful to have at one's disposal more information. This is given either by the function `initell` (see below), which gives a 19 component vector (which we will call a long vector in this section), or by the faster function `smallinitell` which gives a 13 component vector (which we will call a medium vector), all these vectors starting in the same way. Consequently, in functions which do not use the extra information given by `initell`, the curve can be given either as a five component vector, or by one of the longer vectors computed by `initell` or `smallinitell`.

Other functions, in particular those relative to height computations (see `hell` below) require also that the curve be in minimal Weierstrass form. This is achieved by the function `globalred` below.

Points on elliptic curves are represented as two component vectors `[x,y]`, except for the point at infinity, i.e. the identity element of the group law, represented by the one-component vector `[0]`.

3.5.1 addell($e, z1, z2$): sum of the points $z1$ and $z2$ on the elliptic curve corresponding to the vector e.

The library syntax is **addell**($e, z1, z2$).

3.5.2 akell(e, n): compute the coefficient a_n of the L-function of the elliptic curve e, i.e. in principle coefficients of a newform of weight 2 assuming Taniyama-Weil. e must be a medium or long vector of the type given by `smallinitell` or `initell`. For this function to work for every n and not just those prime to the conductor, e must be a minimal Weierstrass equation. If this is not the case, use the function `globalred` first before using `akell`.

The library syntax is **akell**(e, n).

3.5.3 anell(e, n): compute the vector of the first n a_k corresponding to the elliptic curve e, i.e. in principle coefficients of a newform of weight 2 assuming Taniyama-Weil. e must be a medium or long vector of the type given by `smallinitell` or `initell`. For this function to work for every n and not just those prime to the conductor, e must be a minimal Weierstrass equation. If this is not the case, use the function `globalred` first before using `anell`.

The library syntax is **anell**(e, n), where n is a C integer.

3.5.4 apell(e, p): compute the a_p corresponding to the elliptic curve e and the prime number p, using the baby-step giant-step method and a trick due to Mestre. No checking is done that p is indeed prime. The number of points of e over $\mathbf{F}_p$ is $p + 1 - a_p$. e must be a medium or long vector of the type given by `smallinitell` or `initell`.

The library syntax is **apell**(e, p).

3.5.5 apell2(e, p): compute the a_p corresponding to the elliptic curve e and the prime number p as a sum of Legendre symbols. This is slower than `apell` as soon as p is greater than 100, say. e must be a medium or long vector of the type given by `smallinitell` or `initell`.

The library syntax is **apell2**(e, p).

3.5.6 bilhell$(e, z1, z2)$: If $z1$ and $z2$ are points on the elliptic curve e, this function computes the value of the canonical bilinear form on $z1$, $z2$, i.e. `hell(e,z1+z2)-hell(e,z1)-hell(e,z2)`, where + denotes of course addition on e. In addition, $z1$ or $z2$ (but not both) can be vectors or matrices.

The library syntax is **bilhell**$(e, z1, z2, prec)$.

3.5.7 chell(e, v): change the data for the elliptic curve e by changing the coordinates using the vector `v=[u,r,s,t]`, i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$. The vector e must be a medium or long vector of the type given by `smallinitell` or `initell`.

The library syntax is **coordch**(e, v).

3.5.8 chptell(x, v): change the coordinates of the point or vector of points x using the vector `v=[u,r,s,t]`, i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$ (see also `chell` above).

The library syntax is **pointch**(x, v).

3.5.9 globalred(e): calculate the arithmetic conductor, the global minimal model of e and the global Tamagawa number c. Here e is an elliptic curve given by a medium or long vector of the type given by `smallinitell` or `initell`, and is supposed to have all its coefficients a_i in $\mathbf{Q}$. The result is a 3 component vector $[N, v, c]$. N is the arithmetic conductor of the curve, v is itself a vector $[u, r, s, t]$ with rational components. It gives a coordinate change for e over $\mathbf{Q}$ such that the resulting model has integral coefficients, is everywhere minimal, a_1 is 0 or 1, a_2 is 0, 1 or -1 and a_3 is 0 or 1. Such a model is unique, and the vector v is unique if we specify that u is positive. To get the new model, simply type `chell(e,v)`. Finally c is the product of the local Tamagawa numbers c_p, which is a quantity which enters in the Birch and Swinnerton-Dyer conjecture.

The library syntax is **globalreduction**(e).

3.5.10 hell(e, z): global Néron-Tate height of the point z on the elliptic curve e. The vector e must be a long vector of the type given by `initell`. This computation is done using sigma and theta-functions and a trick due to J. Silverman.

The library syntax is **ghell**$(e, z, prec)$. The archimedean contribution alone is given by the library function **hell**$(e, z, prec)$.

3.5.11 hell2(e, z): same as `hell`, except that the algorithm used is Tate's 4^n algorithm, and is much slower.

The library syntax is **ghell2**$(e, z, prec)$.

3.5.12 initell(e): compute some fixed data concerning the elliptic curve given by the five component vector e, which will be essential for most further computations on the curve. The result is a 19-component vector E (called a long vector in this section), containing the following information:

The first 13 components contain

$$a_1, a_2, a_3, a_4, a_6, b_2, b_4, b_6, b_8, c_4, c_6, \Delta, j.$$

In particular, the discriminant is $E[12]$, and the j-invariant is $E[13]$.

For the other six components, their content depends on whether the curve is defined over $\mathbf{R}$ or not.

When e is defined over $\mathbf{R}$, $E[14]$ is a vector with three components containing the roots of the associated Weierstrass equation. If the roots are all real, then they are ordered by decreasing value. If only one is real, it is the first component of $E[14]$.

$E[15]$ is the real period of E (integral of $dx/(2y + a_1x + a_3)$ over the connected component of the identity element of the real points of the curve), and $E[16]$ is a complex period. In other words, $\omega_1 = E[15]$ and $\omega_2 = E[16]$ form a basis of the complex lattice defining e, with $\tau = \frac{\omega_2}{\omega_1}$ having positive imaginary part.

$E[17]$ and $E[18]$ are the corresponding values η_1 and η_2 such that $\eta_1\omega_2 - \eta_2\omega_1 = i\pi$.

Finally, $E[19]$ is the volume of the complex lattice defining e.

When e is defined over $\mathbf{Q}_p$, the p-adic valuation of j must be negative. Then $E[14]$ is the vector with a single component equal to the p-adic root of the associated Weierstrass equation corresponding to -1 under the Tate parametrization.

$E[15]$ is equal to the square of the u-value, in the notation of Tate.

$E[16]$ is the u-value itself, if it belongs to $\mathbf{Q}_p$, otherwise zero.

$E[17]$ is the value of Tate's q for the curve e.

$E[18]$ is the value of Mestre's w (this is technical), and $E[19]$ is arbitrarily set equal to zero.

For all other base fields or rings, the last six components are arbitrarily set equal to zero.

Note: temporarily, an older version of `initell` called `initell2` is installed, but will disappear in the near future.

The library syntax is **initell**($e, prec$).

3.5.13 isoncurve(e, z): gives 1 (i.e. true) if the point z is on the elliptic curve e, 0 otherwise. Here e can be a five-component vector or a long vector.

The library syntax is **oncurve**(e, z), and the result is a C-integer.

3.5.14 localred(e, p): calculate the Kodaira type of the local fiber of the elliptic curve e at the prime p. e must be given by a medium or long vector of the type given by `smallinitell` or `initell`, and is assumed to have all its coefficients a_i in $\mathbf{Z}$. The result is a 4 component vector $[f, kod, v, c]$. Here f is the exponent of p in the arithmetic conductor of e, kod is the Kodaira type which is coded as follows:

1 means good reduction (type I_0), 2, 3 and 4 mean types II, III and IV respectively, $4+\nu$ with $\nu > 0$ means type I_ν; finally the opposite values -1, -2, etc. refer to the starred types I_0^*, II^*, etc. The third component v is itself a vector $[u, r, s, t]$ giving the coordinate changes done during the local reduction. Normally, this has no use if u is 1, that is, if the given equation was already minimal. Finally, the last component c is the local Tamagawa number c_p.

The library syntax is **localreduction**(e, p).

3.5.15 lseriesell(e, s, N, A): e being a medium or long vector given by `smallinitell` or `initell`, this computes the value of the L-series of e at s. It is assumed that e is a minimal model over **Z**, N is equal to plus or minus the conductor depending on the sign of the functional equation, that the curve is a Weil curve, and A is a cutoff point for the integral, which must be chosen close to 1 for best speed. The result being independent of A, this function can be used to compute the sign of the functional equation, and even N if one wants, although it is always much faster to apply the function `globalred`.

Note that in the present version 1.38.1, this function is accessible externally in GP, but in future versions it may only be accessible in library mode since it will be superseded.

Note also that if the conductor of the curve is large, say greater than 10^{12}, this function will take an unreasonable amount of time since it uses a $O(N^{1/2})$ algoorithm.

The library syntax is **lseriesell**$(e, s, N, A, prec)$ where $prec$ is a C-integer.

3.5.16 mathell(e, x): x being a vector of points, this function outputs the Gram matrix of x with respect to the Néron-Tate height, in other words, the `(i,j)` component of the matrix is equal to `hell(e,x[i]+x[j])-hell(e,x[i])-hell(e,x[j])`, where `x[i]+x[j]` denotes of course the sum of the points on the curve e. The rank of this matrix, at least in some approximate sense, gives the rank of the set of points, and if x is a basis of the Mordell-Weil group of e, its determinant is equal to the regulator of e.

The library syntax is **mathell**$(e, x, prec)$.

3.5.17 ordell(e, x): gives a 0, 1 or 2-component vector containing the y-coordinates of the points of the curve e having x as x-coordinate.

The library syntax is **ordell**(e, x).

3.5.18 orderell(e, z): gives the order of the point z on the elliptic curve e if it is a torsion point, zero otherwise. In the present version 1.38.1, this is implemented only for elliptic curves defined over **Q**.

The library syntax is **orderell**(e, z), and the result is a GEN.

3.5.19 pointell(e, z): e being a long vector, computes the coordinates $[x, y]$ on the curve e corresponding to the complex number z. Hence this is the inverse function of `zell`. In other words, if the curve is put in Weierstraß form, $[x, y]$ represent the Weierstraß $\wp$-function and its derivative. If z is in the lattice defining e over **C**, the result is the point at infinity $[0]$.

The library syntax is **pointell**$(e, z, prec)$.

3.5.20 powell(e, n, z): computes n times the point z for the group law on the elliptic curve e. Here n is in **Z**.

The library syntax is **powell**(e, n, z).

3.5.21 smallinitell(e): compute some fixed data concerning the elliptic curve given by the five component vector e, which may be useful for further computations on the curve. The result is a 13-component vector E (called a medium vector in this section), containing the first 13 components of the vector given by `initell`, in other words the vector

$$[a_1, a_2, a_3, a_4, a_6, b_2, b_4, b_6, b_8, c_4, c_6, \Delta, j].$$

In particular, the discriminant is $E[12]$, and the j-invariant is $E[13]$.

The library syntax is **smallinitell**(e).

3.5.22 subell($e, z1, z2$): difference of the points $z1$ and $z2$ on the elliptic curve corresponding to the vector e.

The library syntax is **subell**($e, z1, z2$).

3.5.23 taniyama(e): computes the modular parametrization of the elliptic curve e (where e is given in the format output by `initell` or `smallinitell`), in the form of a two component vector $[u, v]$ of power series, given to the current default series precision. This vector is characterized by the following two properties. First the point $(x, y) = (u, v)$ satisfies the equation of the elliptic curve. Second, the differential $du/(2v + a_1u + a_3)$ is equal to $f(z)dz$, a differential form on $H/\Gamma_0(N)$ where N is the conductor of the curve. The variable used in the power series for u and v is x, which is implicitly understood to be equal to $\exp(2 * i * \pi * z)$. The equation of the curve e must be minimal (use `globalred` to get a minimal equation).

The library syntax is **taniyama**(e), and the precision of the result is determined by the global variable `precdl`.

3.5.24 torsell(e): If e is an elliptic curve defined over $\mathbf{Q}$, this outputs the torsion subgroup of e as a 3 component vector `[t,v1,v2]` with the following meaning. `t` is the order of the torsion group. `v1` is a 0, 1 or 2-component vector giving the structure of the torsion group as a product of cyclic groups, with `v1[2]=2` if it exists. Finally `v2`, which is a vector of the same length as `v1`, gives generators for these cyclic groups. e must be a vector as output by `initell` or `smallinitell`.

The library syntax is **torsell**(e).

3.5.25 zell(e, z): If e is an elliptic curve with coefficients in $\mathbf{R}$, this computes a complex number t (modulo the lattice defining e) corresponding to the point z, i.e. such that, in the standard Weierstrass model, $\wp(t) = z[1], \wp'(t) = z[2]$. In other words, this is the inverse function of `pointell`.

If e has coefficients in $\mathbf{Q}_p$, then either Tate's u is in $\mathbf{Q}_p$, in which case the output is a p-adic number t corresponding to the point z under the Tate parametrization, or only its square is, in which case the output is $t + 1/t$. e must be a long vector of the type given by `initell`.

The library syntax is **zell**($e, z, prec$).

3.6 Functions related to general number fields.

In this section can be found functions which are used almost exclusively for working in general number fields. Other less specific functions can be found in the next section on polynomials. Functions related to quadratic number fields can be found in section 4 above.

Unless otherwise noted, nf will denote a number field, i.e. a 9-component vector in the format output by `initalg` or `initalg0`. An *ideal* (usually x or y) will mean the following:

x can be a $\mathbf{Z}$-basis, in HNF or not. In this case x is a square matrix.

x can be an *idele*, i.e. a 2-component vector, the first being an ideal given as a $\mathbf{Z}$-basis, the second being a $r_1 + r_2$-row vector giving the complex logarithmic archimedean information.

x can be a $\mathbf{Z}_K$-generating system for an ideal.

x can be a prime ideal, i.e. a 5-component vector in the format output by `primedec`.

x can be a polymod, i.e. an algebraic integer, in which case x is treated as being the principal idele generated by x.

Finally, x can be an integer or a rational number, also treated as a principal idele.

3.6.1 base(x): integral basis of the number field defined by the monic irreducible polynomial x, using the round 4 algorithm. For small degrees and coefficient size, `base2` is a little faster. For larger degrees, `base` is much faster.

This program is the translation in C by D. Letard of a program written by David Ford in Maple. If only a local integral basis is desired, use `factoredbase`.

The library syntax is **base**($x, \&d$), where d will receive the discriminant of the number field (*not* of the polynomial x).

3.6.2 base2(x): integral basis of the number field defined by the monic irreducible polynomial x, using the round 2 algorithm. This program is the translation by in C of a program written by David Ford in Algeb.

The library syntax is **base2**($x, \&d$), where d will receive the discriminant of the number field (*not* of the polynomial x).

3.6.3 buchgen($P, c, c2, t$): identical to `buchgenfu` (see below) except that the fundamental units and roots of unity are not computed. Hence the result has only 7 components.

The library syntax is **buchgen**($P, c, c2, t, prec$).

3.6.4 buchgenfu($P, c, c2, t$): Buchmann's sub-exponential algorithm for computing the class group, the regulator and a system of fundamental units of the general algebraic number field K defined by the irreducible monic polynomial P with integer coefficients.

Note that this is a completely experimental program which is constantly being improved.

The numbers c and $c2$ are positive real numbers which control the execution time and the stack size. To get maximum speed, set $c2 = c$. To get a rigorous result (under GRH and more) you must take $c2 = 12$ (or $c2 = 6$ in the quadratic case, but then you should use the much faster functions `buchimag` and `buchreal` below). Reasonable values for c are between 0.1 and 2. t is the number of "extra relations" that we want to use in Buchmann-McCurley's algorithm (see below). We must have $t \geq 1$, and usually $t = 10$ is sufficient.

The result of this function is a vector v with 10 components, which for ease of presentation is in fact output as a one column matrix. $v[1]$ is equal to the polynomial P. Note that for optimum performance, P should have gone through `polred` or `initalg`. $v[2]$ is the 2-component vector $[r1, r2]$ (where $r1$ and $r2$ are as usual the number of real and half the number of complex embeddings of the number field K). $v[3]$ is the 2-component vector containing the field discriminant and the index, $v[4]$ is an integral basis in Hermite normal form. $v[5]$ is a 3-component vector containing the class number, the structure of the class group as a product of cyclic groups of order n_i, and the corresponding generators of the class group of respective orders n_i. $v[6]$ is the regulator computed to an accuracy which is the maximum of an internally determined accuracy and of the default. $v[7]$ is a measure of the correctness of the result. If it is close to 1, the results are correct (under GRH and a few more reasonable assumptions). If it is close to a larger integer, this shows that the product of the class number by the regulator is off by a factor equal to this integer, and you must start again with a larger value for t or c or a different random seed. (Since the computation involves a random process, starting again with exactly the same parameters may give the correct result.) In this case a warning message is printed. $v[8]$ a vector with 2 components, the first being the number w of roots of unity in K and the second a primitive w-th root of unity expressed as a polynomial. $v[9]$ is a system of fundamental units also expressed as polynomials. Finally, $v[10]$ gives a measure of the correctness of the computations of the fundamental units (not of the regulator), expressed as a number of bits. If this number is greater than 20, say, everything is OK. If $v[10] \leq 0$, then we have lost all accuracy in computing the units (usually an error message will be printed and the units not given). In the intermediate cases, one must proceed with caution (for example by increasing the current precision).

Note: some installations may have this function disabled.

The library syntax is **buchgenfu**$(P, c, c2, t, prec)$.

3.6.5 buchinit$(P, c, c2, t)$: essentially identical to `buchgen` except that the output is much longer, and should not be printed out explicitly in general. The result of buchinit is used in programs such as `isprincipal` and `isunit`. The result is an 8-component vector v, whose first 6 components are technical and will not be explained here (see the source code in the file `buch.c`). $v[7]$ is equal to the number field data nf given by `initalg0`, and $v[8]$ is a 4-component vector containing the 5-th, 6-th, 7-th and 8-th components of the result of `buchgenfu`, i.e. the class group, the regulator, the "check" number which should be close to 1 and the number of roots of unity and a generator.

The library syntax is **buchinit**$(P, c, c2, t, prec)$.

3.6.6 buchinitfu$(P, c, c2, t)$: same as `buchinit` except that the 8-th component contains the last 6 components of `buchgenfu`, i.e. also the fundamental units and the check on their computation.

The library syntax is **buchinitfu**$(P, c, c2, t, prec)$.

3.6.7 discf(x): field discriminant of the number field defined by the monic irreducible polynomial x.

The library syntax is **discf**(x). See also `base` above.

3.6.8 factoredbase(x, f): integral basis of the number field defined by the monic irreducible polynomial x, using the round 2 algorithm, where f is the two column matrix of the factorization of the discriminant of the polynomial x. Note that f does *not* need to be a complete factorization of the discriminant. This is especially useful if only a local integral basis at a prime or primes is desired: only factors with exponents greater or equal to 2 will be considered.

The library syntax is **factoredbase**$(x, p, \&d)$, where d will receive the discriminant of the number field.

3.6.9 factoreddiscf(x,p): field discriminant of the number field defined by the monic irreducible polynomial x, using the round 2 algorithm, where p is the two column matrix of the factorization of the discriminant of the polynomial x.

The library syntax is **factoreddiscf**(x,p).

3.6.10 factoredpolred(x,p): same as `polred` (see below) except that p is the two column matrix of the factorization of the discriminant of the polynomial x.

The library syntax is **factoredpolred**$(x,p,prec)$.

3.6.11 factoredpolred2(x,p): same as `polred2` (see below) except that p is the two column matrix of the factorization of the discriminant of the polynomial x.

The library syntax is **factoredpolred2**$(x,p,prec)$.

3.6.12 factornf(x,t): factorization of the univariate polynomial x over the number field defined by the (univariate) polynomial t. x may have coefficients in $\mathbf{Q}$ or in the number field. The main variable of t must be of lower priority than that of x (in other words the variable number of t must be greater than that of x). However if the coefficients of the number field occur explicitly (as polymods) as coefficients of x, the variable of these polymods *must* be the same as the main variable of t. For example `factornf(x^2+mod(y,y^2+1),y^2+1)` and `factornf(x^2+1,y^2+1)` are legal but `factornf(x^2+mod(z,z^2+1),y^2+1)` is not.

The library syntax is **polfnf**(x,t).

3.6.13 galois(x): Galois group of the nonconstant polynomial $x \in \mathbf{Q}[X]$. In the present version 1.38.1, x must be irreducible and the degree of x must be less than or equal to 7 (for degrees 8,9,10, and 11 we have specific Pari programs available upon request).

The output is a 3-component vector $[n,s,k]$ with the following meaning: n is the cardinality of the group, s is its signature ($s=1$ if the group is a subgroup of the alternating group A_n, $s=-1$ otherwise), and k is the number of the group corresponding to a given pair (n,s) ($k=1$ except in 2 cases). Specifically, the groups are coded as follows, using standard notations:

In degree 1: $S_1=[1,-1,1]$.

In degree 2: $S_2=[2,-1,1]$.

In degree 3: $A_3=C_3=[3,1,1]$, $S_3=[6,-1,1]$.

In degree 4: $C_4=[4,-1,1]$, $V_4=[4,1,1]$, $D_4=[8,-1,1]$, $A_4=[12,1,1]$, $S_4=[24,-1,1]$.

In degree 5: $C_5=[5,1,1]$, $D_5=[10,1,1]$, $M_{20}=[20,-1,1]$, $A_5=[60,1,1]$, $S_5=[120,-1,1]$.

In degree 6: $C_6=[6,-1,1]$, $S_3=[6,-1,2]$, $D_6=[12,-1,1]$, $A_4=[12,1,1]$, $G_{18}=[18,-1,1]$, $S_4^-=[24,-1,1]$, $A_4\times C_2=[24,-1,2]$, $S_4^+=[24,1,1]$, $G_{36}^-=[36,-1,1]$, $G_{36}^+=[36,1,1]$, $S_4\times C_2=[48,-1,1]$, $A_5=PSL_2(5)=[60,1,1]$, $G_{72}=[72,-1,1]$, $S_5=PGL_2(5)=[120,-1,1]$, $A_6=[360,1,1]$, $S_6=[720,-1,1]$.

In degree 7: $C_7=[7,1,1]$, $D_7=[14,-1,1]$, $M_{21}=[21,1,1]$, $M_{42}=[42,-1,1]$, $PSL_2(7)=PSL_3(2)=[168,1,1]$, $A_7=[2520,1,1]$, $S_7=[5040,-1,1]$.

The method used is that of resolvent polynomials. Beware that in the case of degree 7 one has to factor polynomials of degree 35 over $\mathbf{Z}$, and this can take a long time. Furthermore there is a risk that the pari stack overflows, so it is prudent in this case to start with a large stack.

The library syntax is **galois**$(x, prec)$.

3.6.14 galoisconj(x): conjugates of a root r of the nonconstant polynomial x expressed as polynomials in r. This can be used even if the corresponding number field is not Galois since some conjugates may lie in the field.

The library syntax is **galoisconj**$(x, prec)$. A slower version but which has a better chance of finding the conjugates when the index is large is **galoisconj2**$(x, prec)$ (not accessible from GP).

3.6.15 idealadd(nf, x, y): sum of the two ideals x and y in the number field nf. When x and y are given by **Z**-bases, this does not depend on nf and can be used to compute the sum of any two **Z**-modules. The result is given in HNF.

The library syntax is **idealadd**(nf, x, y).

3.6.16 idealdiv(nf, x, y): quotient $x \cdot y^{-1}$ of the two ideals x and y in the number field nf, where nf is a 9-component vector given by `initalg` or `initalg0`. The result is given in HNF.

The library syntax is **idealdiv**(nf, x, y).

3.6.17 idealfactor(nf, x): factors into prime ideal powers the ideal x in the number field nf, where nf is a 9-component vector given by `initalg` or `initalg0`. The output format is similar to the `factor` function, and the prime ideals are represented in the form output by the `primedec` function, i.e. as 5-element vectors.

The library syntax is **idealfactor**(nf, x).

3.6.18 idealintersect(nf, x, y): sum of the two ideals x and y in the number field nf. When x and y are given by **Z**-bases, this does not depend on nf and can be used to compute the intersection of any two **Z**-modules. The result is given in HNF.

The library syntax is **idealintersect**(nf, x, y).

3.6.19 idealinv(nf, x): inverse of the ideal x in the number field nf. nf must be a 9-component vector as given by `initalg`, The result is the Hermite normal form of the inverse of the ideal, together with the opposite of the Archimedean information if it is given.

The library syntax is **idealinv**(nf, x).

3.6.20 ideallllred$(nf, x, vdir)$: LLL reduction of the ideal x in the number field nf, along the direction $vdir$. Here $vdir$ must be either an $r1 + r2$-component vector ($r1$ and $r2$ number of real and complex places of nf as usual), or the Pari zero, in which case $vdir$ is assumed to be equal to the vector having only components equal to 1. The notion of reduction along a direction is technical and cannot be explained here. Note that this is *not* the same as the LLL reduction of the lattice x since ideal operations are involved. The result is the Hermite normal form of the LLL-reduced ideal, which is usually, but not always, a reduced ideal. x may also be a 2-component vector, the first being as above, and the second containing a matrix of Archimedean information. In that case, this matrix is suitably updated.

The library syntax is **ideallllred**$(nf, x, vdir, prec)$.

3.6.21 idealmul(nf, x, y): ideal multiplication of the ideals x and y in the number field nf. nf must be a 9-component vector as given by `initalg` or `initalg0`. The result is a generating set for the ideal product with at most n elements, and is in Hermite normal form if x and y are, and this is given together with the sum of the Archimedean information in x and y if both are given.

The library syntax is **idealmul**(nf, x, y).

3.6.22 idealmulred(nf, x, y): same as `idealmul`, except that the result is reduced using `ideallllred`.

The library syntax is **idealmulred**$(nf, x, y, prec)$, where $prec$ is a C-integer representing the precision.

3.6.23 idealpow(nf, x, k): computes the k-th power of the ideal x in the number field nf as given by `initalg` or `initalg0`. k can be positive negative or zero. The result is NOT reduced, it is really the k-th ideal power, and is given in HNF. Use `idealpowred` if you want a reduced result.

The library syntax is **idealpow**(nf, x, k).

3.6.24 idealpowred(nf, x, k): computes a reduced k-th power of the ideal x in the number field nf as given by `initalg` or `initalg0`. To get a non-reduced result, use `idealpow`. Note however that `idealpowred` is NOT the same as as `idealpow` followed by reduction (i.e. `ideallllred`), since the reduction is performed throughout the powering process.

The library syntax is **idealpowred**$(nf, vp, k, prec)$, where $prec$ is a C-integer.

3.6.25 idealval(nf, x, vp): gives the valuation of the ideal x at the prime ideal vp in the number field nf. nf must be a 9-component vector as given by `initalg` or `initalg0`, and vp must be a 6-component vector as given by `primedec`.

The library syntax is **idealval**(nf, x, vp) and the result is a C-integer.

3.6.26 initalg(x): x being a nonconstant irreducible polynomial in $\mathbf{Q}[X]$, computes a 9-component vector v useful in working in the number field K defined by x. $v[1]$ contains a polynomial defining K which is as simple as can easily be found using the `polred` algorithm. $v[2]$ contains $[r1, r2]$, the number of real and complex places of K. $v[3]$ contains the discriminant of the number field K. $v[4]$ contains the index of $v[1]$, i.e. $[\mathbf{Z}_K : \mathbf{Z}[\theta]]$, where θ is any root of $v[1]$. $v[5]$ is a vector containing 5 matrices M, MC, $T2$, T and MD useful for certain computations in the number field K. The matrix M is the $(r1 + r2) \times n$ matrix whose columns represent the numerical values of the conjugates of the elements of the integral basis. The matrix MC is essentially the conjugate of the transpose of M, except that the last $r2$ columns are also multiplied by 2. The matrix $T2$ is an $n \times n$ matrix equal to the product $MC \cdot M$, the so-called T2-matrix. The matrix T is the $n \times n$ matrix whose coefficients are $\mathrm{Tr}(\omega_i\omega_j)$ where the ω_i are the elements of the integral basis. Note that $T = \overline{MC} \cdot M$ and in particular that $T = T_2$ if the field is totally real (in practice T_2 will have real approximate entries and T will have integer entries). Note also that $\det(T)$ is equal to the discriminant of the field K. Finally, the columns of MD express a $\mathbf{Z}$-basis of the different of K on the integral basis $v[7]$.

$v[6]$ is the vector containing the $r1 + r2$ roots of $v[1]$ corresponding to the $r1 + r2$ embeddings of the number field into $\mathbf{C}$ (the first $r1$ components are real, the next $r2$ have positive imaginary part). $v[7]$ is an integral basis in Hermite normal form for $\mathbf{Z}_K$ expressed on the powers of θ. $v[8]$ is the $n \times n$ integral matrix expressing the power basis in terms of the integral basis, and finally $v[9]$ is the $n \times n^2$ matrix giving the multiplication table of the integral basis.

The library syntax is **initalg**$(x, prec)$.

3.6.27 initalg0(x): same as `initalg` except that the polynomial is not changed. This is useful in conjunction with functions expecting a number field `nf` but where one wants to keep the initial polynomial.

The library syntax is **initalg0**$(x, prec)$.

3.6.28 initalg2(P): this is a complement to `initalg`. With the same conditions, it produces a polymod $mod(a, pol)$ equal to $mod(x, P)$ where pol is the polynomial output by `initalg`.

The library syntax is **initalg2**$(x, prec)$.

3.6.29 isincl(x, y): tests whether the number field K defined by the polynomial x is conjugate to a subfield of the field L defined by y (where x and y must be in $\mathbf{Q}[X]$). If they are not, the output is the number 0. If they are, the output is a vector of polynomials, each polynomial a representing an isomorphism of K into L, i.e. being such that $y = x \circ a$.

The library syntax is **nfincl**(x, y).

3.6.30 isisom(x, y): tests whether the two number fields defined by the polynomials x and y (which must be in $\mathbf{Q}[X]$) are isomorphic. If they are not, the output is the number 0. If they are, the output is a vector of polynomials, each polynomial a representing an isomorphism, i.e. being such that $y = x \circ a$.

The library syntax is **nfiso**(x, y).

3.6.31 isprincipal(bnf, x): bnf being the number field data output by `buchinit`, and x being either a $\mathbf{Z}$-basis of an ideal in the number field (not necessarily in HNF) or a prime ideal in the format output by the function `primedec`, this function tests whether the ideal is principal or not. The result is much more complete than a simple true/false answer: it is given as a 3 component vector $[v_1, v_2, e]$ with the following meaning. v_1 is the vector of components c_i of the class of the ideal x in the class group, expressed on the generators g_i given by `buchinit` (specifically `buchinit[8][1][3]`). The c_i are chosen so that $0 \leq c_i < n_i$ where n_i is the order of g_i (the vector of n_i being `buchinit[8][1][2]`).

The second component v_2 gives on the integral basis the components of α such that $x = \alpha \prod_i g_i^{c_i}$. In particular, x is principal if and only if v_1 is equal to the zero vector, and if this the case $x = \alpha \mathbf{Z}_K$ where α is given by v_2. Note that if α is too large to be given, a warning message will be printed and v_2 will be set equal to the empty vector. Finally the third component e is analogous to the last component of `buchgen`: it gives a check on the accuracy of the result, in bits. e should be at least 10, and preferably much more. If this is not the case, the result may be incorrect, and a higher default precision should be taken.

The library syntax is **isprincipal**(bnf, x).

3.6.32 isunit(bnf, x): bnf being the number field data output by `buchinit` and x being an algebraic number (type integer, rational or polymod), this outputs the decomposition of x on the fundamental units an the roots of unity if x is a unit, the empty vector otherwise. More precisely, if u_1,...,u_r are the fundamental units found by `buchgen` and if ζ is the generator of the group of roots of unity found by `buchgen`, the output is a vector $[x_1, \ldots, x_r, x_{r+1}]$ such that $x = u_1^{x_1} \cdots u_r^{x_r} \cdot \zeta^{x_{r+1}}$. The x_i are integers for $i \leq r$ and is an integer modulo the order of ζ for $i = r + 1$.

The library syntax is **isunit**(bnf, x).

3.6.33 modreverse(a): a being a polymod $A(X)$ modulo $T(X)$, finds the "reverse polymod" $B(X)$ modulo $Q(X)$, where Q is the minimal polynomial of a, which must be equal to the degree of T, and such that if θ is a root of T then $\theta = B(\alpha)$ for a certain root α of Q.

This is very useful when one changes the generating element in algebraic extensions.

The library syntax is **polymodrecip**(x).

3.6.34 newtonpoly(x, p): gives the vector of the slopes of the Newton polygon of the polynomial x with respect to the prime number p. The n components of the vector are in decreasing order, and n is equal to the degree of x.

The library syntax is **newtonpoly**(x, p).

3.6.35 ordred(x): finds polynomials with reasonably small coefficients and of the same degree as that of x defining suborders of the order defined by x. One of the polynomials always defines $\mathbf{Q}$ (hence is equal to $(x-1)^n$, where n is the degree), and another always defines the same order as x if x is irreducible.

The library syntax is **ordred**(x).

3.6.36 polred(x): finds polynomials with reasonably small coefficients defining subfields of the number field defined by x. One of the polynomials always defines $\mathbf{Q}$ (hence is equal to $(x-1)$, where n is the degree), and another always defines the same number field as x if x is irreducible.

The library syntax is **polred**$(x, prec)$.

3.6.37 polred2(x): finds polynomials with reasonably small coefficients defining subfields of the number field defined by x. The result is a two column matrix, the first column giving the elements, the second giving the polynomials.

One of the polynomials always defines $\mathbf{Q}$ (hence is equal to $(x-1)$, where n is the degree), and another always defines the same number field as x if x is irreducible.

The library syntax is **polred2**$(x, prec)$.

3.6.38 primedec(nf, p): computes the prime ideal decomposition of the prime number p in the number field nf. nf must be a 9-component vector as given by `initalg` and p must be a (positive) prime number. Note that the fact that p is prime is not checked, so if a non-prime number p is given it may lead to unpredictable results.

The result is a vector of 5-component vectors, each representing one of the prime ideals above p in the number field nf. The representation $vp = [p, a, e, f, b]$ of a prime ideal means the following. The prime ideal is equal to $p\mathbf{Z}_K + \alpha\mathbf{Z}_K$ where $\mathbf{Z}_K$ is the ring of integers of the field and $\alpha = \sum_i a_i\omega_i$ where the ω_i form the integral basis $nf[7]$, e is the ramification index, f is the residual index, and b is an n-component column vector representing some $\beta \in \mathbf{Z}_K$ which will be useful for computing valuations but which the user can ignore.

The library syntax is **primedec**(nf, p).

3.6.39 principalideal(nf, x): creates the principal ideal generated by the algebraic number x (which must be of type integer, rational or polymod) in the number field defined by the 9-component vector nf. The result is a 1-column matrix.

The library syntax is **principalideal**(nf, x).

3.6.40 principalidele(nf, x): creates the principal idele generated by the algebraic number x (which must be of type integer, rational or polymod) in the number field defined by the 9-component vector nf. The result is a 2-component vector, the first being a 1-column matrix representing the corresponding principal ideal, and the second being the vector with $r_1 + r_2$ components giving the complex logarithmic embedding of x.

The library syntax is **principalidele**(nf, x).

3.6.41 rootsof1(nf): computes the number of roots of unity w and a primitive w-th root of unity (expressed on the integral basis) belonging to the number field defined by the vector nf as given by `initalg`. The result is a two component vector $[w, z]$ where z is a column vector expressing a primitive w-th root of unity on the integral basis $nf[7]$.

The library syntax is **rootsof1**(nf).

3.6.42 smallbase(x): integral basis of the number field defined by the monic irreducible polynomial x, using the round 2 algorithm, where one does not take into account squares of primes which are not precomputed.

The library syntax is **smallbase**($x, \&y$), where y will receive the discriminant of the number field (*not* of the polynomial x). This program is the translation in C of a program written by David Ford.

3.6.43 smalldiscf(x): field discriminant of the number field defined by the monic irreducible polynomial x, where one does not take into account squares of primes which are not precomputed.

The library syntax is **smalldiscf**(x). See also `smallbase` above.

3.6.44 smallpolred(x): same as `polred` above except that only a suborder of the maximal order may be used.

The library syntax is **smallpolred**($x, prec$).

3.6.45 smallpolred2(x): same as `polred2` above except that only a suborder of the maximal order may be used.

The library syntax is **smallpolred2**($x, prec$).

3.6.46 tschirnhaus(x): apply a random Tschirnhausen transformation to the nonconstant separable polynomial x so as to obtain a new equation for the number field (or more generally the etale algebra) defined by x. This is for instance useful when computing resolvents, hence is used in the `galois` function above.

The library syntax is **tschirnhaus(x)**.

3.7 Polynomials and power series.

We group here all functions which are specific to polynomials or power series. Many other functions which can be applied on these objects are described in the other sections. Also, some of the functions described here can be applied to other types.

3.7.1 apprpadic(x, a): vector of p-adic roots of the polynomial x congruent to the p-adic number a modulo p (or modulo 4 if $p = 2$), and with the same p-adic precision as a. The number a can be an ordinary p-adic number (type 7, i.e. an element of $\mathbf{Q}_p$) or can be an element of a finite extension of $\mathbf{Q}_p$, in which case it is of type 9 (polymod), where at least one of the coefficients of the polymod is a p-adic number. In this case, the result is the vector of roots belonging to the same extension of $\mathbf{Q}_p$ as a.

The library syntax is **apprgen9**(x, a), but if a is known to be simply a p-adic number (type 7), the syntax **apprgen**(x, a) can be used.

3.7.2 convol(x, y): convolution (or Hadamard product) of the two power series x and y; in other words if $x = \sum a_k * X^k$ and $y = \sum b_k * X^k$ then $\mathrm{convol}(x, y) = \sum a_k * b_k * X^k$.

The library syntax is **convol**(x, y).

3.7.3 cyclo(n): n-th cyclotomic polynomial, where n must be positive.

The library syntax is **cyclo**(n), where n is a C-integer.

3.7.4 deriv(x, y): derivative of x with respect to the simple variable y. x can be any type except polymod. The derivative of a scalar type is zero, and the derivative of a vector or matrix is done componentwise.

The library syntax is **deriv**(x, v), where v is the number of the variable y.

3.7.5 disc(x): discriminant of x. x must be a polynomial. The algorithm used is the subresultant algorithm.

The library syntax is **discsr**(x).

3.7.6 eval(x): replace in x the formal variables by the values that have been assigned to them after the creation of x. This is mainly useful in GP, and not in library mode. Do not confuse this with substitution (see `subst` below).

The library syntax is **geval**(x).

3.7.7 factorpadic(x, p, r): p-adic factorization of the polynomial x to precision r, the result being a two column matrix as in `factor`. r must be in general be strictly larger than the p-adic valuation of the discriminant of x for the result to make any sense. The method used is Ford-Letard's implementation of the round 4 algorithm of Zassenhaus. Another version using an algorithm due to Buchmann and Lenstra is `factorpadic2`, but is less efficient.

The library syntax is **factorpadic4**(x, p, r), where r is a C-integer.

3.7.8 factpol(x, l): x must be a polynomial with coefficients in $\mathbf{Z}$. If $l = 0$, find the complete factorization of x, and if $l > 0$, search only for irreducible factors of degree less than or equal to l. The result is a two column matrix, the first one containing the irreducible factors, the second one the exponents.

The library syntax is **factpol**(x, l). The algorithm used is the standard Hensel lifting of a mod p factorization. Another implementation using instead root finding over $\mathbf{C}$ (instead of implicitly using $\mathbf{Q}_p$) is **factpol2**(x, l).

3.7.9 integ(x, y): formal integration of x with respect to the simple variable y. No logarithmic terms must occur in the result. x can be of any type, but the case where x is a rational function is not implemented in the present version 1.38.1.

The library syntax is **integ**(x).

3.7.10 laplace(x): x must be a power series with only nonnegative exponents. If $x = \sum (a_k/k!) * X^k$ then the result is $\sum a_k * X^k$.

The library syntax is **laplace**(x).

3.7.11 legendre(x): creates the x^{th} Legendre polynomial.

The library syntax is **legendre**(x), where x is a 32-bit C-integer.

3.7.12 polint(xa, ya, x): given the data vectors xa and ya of the same length n (xa containing the x-coordinates, and ya the corresponding y-coordinates), this function finds the interpolating polynomial passing through these points and evaluates it at the value x.

The library syntax is **polint**$(xa, ya, x, \&er)$, where er will contain an error estimate on the returned value.

3.7.13 polsym(x, n): creates the vector of the symmetric powers of the roots of the polynomial x up to power n.

The library syntax is **polsym**(x).

3.7.14 recip(x): reciprocal polynomial of x, i.e. the coefficients are in reverse order. x must be a polynomial.

The library syntax is **polrecip**(x).

3.7.15 resultant(x, y): resultant of the two polynomials x and y with exact entries. The algorithm used is the subresultant algorithm.

The library syntax is **subres**(x, y).

3.7.16 resultant2(x, y): resultant of the two polynomials x and y. The algorithm used is the determinant of Sylvester's matrix.

The library syntax is **resultant2**(x, y).

3.7.17 reverse(x): reverse power series (i.e. x^{-1}, not $1/x$) of x. x must be a power series whose valuation is exactly equal to one.

The library syntax is **recip**(x).

3.7.18 rootmod(x, p): roots modulo p of the polynomial x. Slightly slower than `rootmod2` below for $p < 100$, but much faster for larger values. The particular nonprime value $p = 4$ is accepted, mainly for 2-adic computations. Multiple roots are *not* repeated.

The library syntax is **rootmod**(x, p).

3.7.19 rootmod2(x, p): roots modulo p of the polynomial x. To be used only when p is small. Multiple roots are repeated with their order of multiplicity.

The library syntax is **rootmod2**(x, p).

3.7.20 rootpadic(x, p, r): vector of p-adic roots of the polynomial x with p-adic precision equal to r. Multiple roots are *not* repeated.

The library syntax is **rootpadic**(x, p, r), where r is a C-integer.

3.7.21 roots(x): complex roots of the polynomial x, given as a column vector where each root is repeated according to its multiplicity. The precision is given as for transcendental functions: under GP it is kept in the variable `prec` and is transparent to the user, but it must be explicitly given as a second argument in library mode.

The algorithm used is a variant of the Newton-Raphson method and is not guaranteed to converge, but is rather fast. If you get the messages "too many iterations in roots" or "INTERNAL ERROR: incorrect result in roots", try modifying your polynomial to get the roots, or try to use the slower function `rootslong`.

The library syntax is **roots**($x, prec$).

3.7.22 rootslong(x): same as `roots` except that the program is a little more robust and slower.

The library syntax is **rootslong**($x, prec$).

3.7.23 sturm(x): number of real roots of the real polynomial x, using Sturm's algorithm.

The library syntax is **sturm**(x). The result is a 32-bit C-integer.

3.7.24 sturmpart(x, a, b): number of real roots of the real polynomial x in the interval $(a, b]$, using Sturm's algorithm.

The library syntax is **sturmpart**(x). The result is a 32-bit C-integer.

3.7.25 subst(x, y, z): replace the simple variable y by the argument z in expression x. Every non-scalar type is allowed for x. If x is a power series, z must be either a polynomial, a power series, or a rational function. y must be a simple variable name.

The library syntax is **gsubst**(x, v, z), where v is the number of the variable y.

3.7.26 taylor(x, y): Taylor expansion around 0 of x with respect to the simple variable y. x can be of any reasonable type, for example a rational function. The number of terms of the expansion is transparent to the user under GP, but must be given as a second argument in library mode.

The library syntax is **tayl**(x, y, n), where the 32-bit C-integer n is the desired number of terms in the expansion.

3.7.27 tchebi(x): creates the x^{th} Tchebicheff polynomial.

The library syntax is **tchebi**(x), where x is a 32-bit C-integer.

3.8 Vectors, matrices and linear algebra.

3.8.1 adj(x): adjoint matrix of x, i.e. the matrix y of cofactors of x such that $x * y = \det(x) * \mathrm{Id}$. x must be a (non necessarily invertible) square matrix.

The library syntax is **adj**(x).

3.8.2 algdep(x, k): x being real or complex, finds a polynomial of degree at most k having x as approximate root. The algorithm used is a variant of the LLL algorithm due to Hastad, Lagarias and Schnorr (STACS 1986). Note that the polynomial which is obtained is not necessarily the "correct" one. One can check the closeness either by a polynomial evaluation or substitution, or by finding the roots of the polynomial given by algdep.

The library syntax is **algdep**($x, k, prec$), where k is a 32-bit C-integer.

3.8.3 algdep2(x, k, dec): x being real or complex, finds a polynomial of degree at most k having x as approximate root. dec is a number which should be between half the number of decimal digits of precision and that number. The algorithm used is the LLL algorithm. Note that the polynomial which is obtained is not necessarily the "correct" one. One can check the closeness either by a polynomial evaluation or substitution, or by finding the roots of the polynomial given by algdep2.

The library syntax is **algdep2**$(x, k, dec, prec)$, where k and dec are 32-bit C-integers.

3.8.4 char(x, y): characteristic polynomial of x with respect to the variable y, i.e. determinant of $y*I-x$ if x is a square matrix, determinant of the map multiplication by x if x is a polymod, error if x is of any other type. For matrices, the method used is essentially the same as for computing the adjoint matrix, i.e. computing the traces of the powers of x.

The library syntax is **caradj0**(x, v), where v is the variable number, or even **caradj**$(x, v, 0)$. Note that the function `caradj(x,v,pty)` returns the characteristic polynomial of x in the variable v, and if `pty` is not equal to 0, puts in `pty` the address of the adjoint matrix of x, so both can be obtained at once.

3.8.5 char1(x, y): characteristic polynomial of x with respect to the variable y, i.e. determinant of $y * I - x$ if x is a square matrix, determinant of the map multiplication by x if x is a polymod, error if x is of any other type. For matrices, the method used is that of Lagrange interpolation and is almost always slower than `char`. Identical to `char` for polymods.

The library syntax is **caract**(x, v), where v is the variable number.

3.8.6 char2(x, y): characteristic polynomial of the square matrix x with respect to the variable y using the Hessenberg form. This is faster than `char` when the coefficients are integermod a prime or real numbers, but is usually slower in other base rings.

The library syntax is **carhess**(x, v), where v is the variable number.

3.8.7 concat(x, y): concatenation of x and y. If x or y is not a vector or matrix, it is considered as a one dimensional vector. All types are allowed for x and y, but the sizes must be compatible. Note that matrices are concatenated horizontally, i.e. the number of rows stays the same. Using transpositions, it is easy to concatenate them vertically.

To concatenate vectors sideways (i.e. to obtain a 2-row or two column matrix), first transform the vector into a 1-row or 1-column matrix using the function `mat` below.

The library syntax is **concat**(x, y).

3.8.8 det(x): determinant of x. x must be a square matrix. Another program called **det2**(x) is better when the entries of the matrix are reals or integers for example, but can be much worse for more complicated entries like multivariate polynomials.

The library syntax is **det**(x) and **det2**(x).

3.8.9 detint(x): x being an $m \times n$ matrix with integer coefficients, this function computes a multiple of the determinant of the lattice generated by the columns of x if it is of rank m, and returns zero otherwise. This function can be useful in conjunction with the function `hermitemod` which needs to know such a multiple. Other ways to obtain this determinant (assuming the rank is maximal) is `det(lllkerim(x)[2]*x)` or simply `det(hermite(x))`. Try to see which is faster in your case.

The library syntax is **detint**(x).

3.8.10 eigen(x): gives the eigenvectors of x as columns of a matrix.

The library syntax is **eigen**(x).

3.8.11 extract(x, y): extraction of components of the vector or matrix x according to y. x must be a vector or a matrix. In the case of a matrix, the components are as usual the *columns* of

x. y must be either a number of PARI type integer, in which case it is considered as a mask: The binary bits of y are read from right to left, but correspond to taking the components from left to right. For example, if $y = 13 = (1101)_2$ then the components 1,3 and 4 are extracted.

Or y can be a vector, with integer entries, in which case these entries correspond to the component number to be extracted, in the order specified.

In the case of a matrix, the difference between extract and matextract is that only columns are extracted.

The library syntax is **extract**(x, y).

3.8.12 gauss(x, y): x being a square matrix and y a column vector, finds the solution u of $x * u = y$, using Gaussian elimination. This has the same effect, but is much faster, than $x^{-1} * y$.

The library syntax is **gauss**(x, y).

3.8.13 hermite(x): if x is a (not necessarily square) matrix of maximal rank, finds the *upper triangular* Hermite Normal Form of x If the rank of x is equal to its number of rows, the result is a square matrix. In general, the columns of the result form a basis of the lattice spanned by the columns of x.

The library syntax is **hnf**(x).

3.8.14 hermitemod(x, d): if x is a (not necessarily square) matrix of maximal rank, and d is the (non-zero) determinant of the lattice spanned by the columns of x, finds the *upper triangular* Hermite Normal Form of x If the rank of x is equal to its number of rows, the result is a square matrix. In general, the columns of the result form a basis of the lattice spanned by the columns of x. This is much faster than `hermite` when d is known.

The library syntax is **hnfmod**(x, d).

3.8.15 hess(x): Hessenberg form of the square matrix x.

The library syntax is **hess**(x).

3.8.16 hilbert(x): x being a 32-bit C-integer, creates the Hilbert matrix of order x, i.e. the matrix whose coefficient (i,j) is $\frac{1}{i+j-1}$.

The library syntax is **hilbert**(x).

3.8.17 idmat(n): creates the $n \times n$ identity matrix.

The library syntax is **idmat**(n) where n is a C-integer. Related functions are **gscalmat**(x, n) which creates x times the identity matrix (x being a GEN and n a C-integer) and **gscalsmat**(x, n) which is the same when x is a C-integer.

3.8.18 image(x): gives a basis for the image of the matrix x as columns of a matrix. A priori the matrix can have entries of any type.

The library syntax is **image**(x).

3.8.19 imagecompl(x): gives the vector of the column indices which are not extracted by the function `image`. Hence the number of components of `imagecompl(x)` plus the number of columns of `image(x)` is equal to the number of columns of the matrix x.

The library syntax is **imagecompl**(x).

3.8.20 imager(x): gives a basis for the image of the matrix x as columns of a matrix, where x has elements which can be nonexact real or complex numbers. In that case, the precision of the matrix entries determines what is meant by an element of the image. In particular, if the matrix is ill conditioned, the results may not be what you expect.

The library syntax is **imagereel**(x).

3.8.21 indexrank(x): x being a matrix of rank r, gives two vectors of length r giving a list of rows and columns respectively (starting from 1) such that the extracted matrix obtained from these 2 vectors using `matextract` is invertible.

The library syntax is **indexrank**(x).

3.8.22 indsort(x): indirect sorting of the vector x, i.e. if x is an n-dimensional vector, creates the permutation of $[1, 2, \ldots, n]$ which applied to the components of x sorts x in increasing order.

The library syntax is **indexsort**(x).

3.8.23 intersect(x, y): x and y being two matrices with the same number of rows whose columns are independent, finds a basis of the **Q**-vector space equal to the intersection of the spaces spanned by the columns of x and y respectively.

The library syntax is **intersect**(x, y).

3.8.24 inverseimage(x, y): gives one column vector belonging to the inverse image of the column vector y by the matrix x if one exists, the empty vector otherwise. To get the complete inverse image, it suffices to add to the result any element of the kernel of x obtained for example by `ker`.

The library syntax is **inverseimage**(x, y).

3.8.25 jacobi(x): x being a real symmetric matrix, this gives a vector having two components: the first one is the vector of eigenvalues of x, the second is the corresponding orthogonal matrix of eigenvectors of x. The method used is Jacobi's method for symmetric matrices.

The library syntax is **jacobi**(x).

3.8.26 ker(x): gives a basis for the kernel of the matrix x as columns of a matrix. A priori the matrix can have entries of any type. See also `keri` and `kerr`.

The library syntax is **ker**(x).

3.8.27 keri(x): same as `ker`, except that it assumes that the matrix has entries of type integer. In that case it is much faster.

The library syntax is **keri**(x).

3.8.28 kerint(x): gives an LLL-reduced **Z**-basis for the lattice equal to the kernel of the matrix x as columns of the matrix x with integer entries (rational entries are not permitted) using a modified integer LLL algorithm.

The library syntax is **kerint**(x).

3.8.29 kerint1(x): like `kerint`, but using `matrixqz3`, gives an LLL-reduced **Z**-basis for the lattice equal to the kernel of the matrix x as columns of the matrix x with integer or rational entries. If LLL reduction of the final result is not desired, use `matrixqz3(ker(x))` instead.

The library syntax is **kerint1**(x).

3.8.30 kerint2(x): like `kerint`, but using another modified LLL algorithm, gives an LLL-reduced **Z**-basis for the lattice equal to the kernel of the matrix x as columns of the matrix x with integer entries (rational entries are not permitted). In the present version 1.38.1, only independent rows are allowed.

The library syntax is **kerint2**(x).

3.8.31 kerr(x): gives a basis for the kernel of the matrix x as columns of a matrix, where x has elements which can be nonexact real or complex numbers. In that case, the precision of the matrix entries determines what is meant by an element of the kernel. In particular, if the matrix is ill conditioned, the results may not be what you expect.

The library syntax is **kerreel**(x).

3.8.32 lexsort(x) sort the elements of the vector x by ascending lexicographic order (see `lex` above).

The library syntax is **lexsort**(x).

3.8.33 lindep(x): x being a vector with real or complex coefficients, finds a small integral linear combination among these coefficients using a variant of the LLL algorithm due to Hastad, Lagarias and Schnorr (STACS 1986).

The library syntax is **lindep**($x, prec$).

3.8.34 lindep2(x, dec): x being a vector with real or complex coefficients, finds a small integral linear combination among these coefficients using the LLL algorithm. dec is a parameter which should be between one half the number of decimal digits of precision and that number.

The library syntax is **lindep2**($x, dec, prec$) where dec is a C-integer.

3.8.35 lll(x): LLL algorithm applied to the **columns** of the (non necessarily square) matrix x. The columns of x must however be of maximal rank (otherwise if x has integral coefficients use `lllint`. The result is a square transformation matrix T such that $x \cdot T$ is an LLL-reduced basis of the lattice generated by the column vectors of x. The computations are done with real numbers (i.e. not with rational numbers) hence are fast but as presently programmed (version 1.38.1) are numerically unstable. An older version of this program is `lll1`.

The library syntax is **lll**($x, prec$).

3.8.36 lllgram(x): same as LLL except that the matrix x which is now square is the gram matrix of the lattice vectors, and not the coordinates of the vectors themselves. The result is again the transformation matrix T which gives (as columns) the coefficients with respect to the initial basis vectors. Same remarks as for LLL about numerical instability in the present version 1.38.1. An older version of this program is `lllgram1`.

The library syntax is **lllgram**($x, prec$).

3.8.37 lllgramint(x): The matrix x being the gram matrix of a basis of some lattice, is assumed to have integral entries. The result is the unimodular transformation matrix T which gives (as columns) the coefficients with respect to the initial basis vectors. The computations are done entirely with integers and is both accurate and quite fast.

The library syntax is **lllgramint**(x).

3.8.38 lllgramkerim(x): LLL algorithm applied to the gram matrix of some lattice, which is assumed to have integral entries. Here the columns can be linearly dependent. The result is a two component vector of matrices whose concatenation form a unimodular transformation matrix T which gives as columns the coefficients with respect to the initial basis vectors. The first matrix will give the kernel, the second will give an LLL-reduced basis of the image.

The library syntax is **lllgramkerim**(x).

3.8.39 lllint(x): LLL algorithm applied to the **columns** of the (non necessarily square) matrix x. It is assumed that the corresponding gram matrix is integral. The result is a square transformation matrix T such that $x \cdot T$ is an LLL-reduced basis of the lattice generated by the column vectors of x. The computations are done entirely with integers and is both accurate and quite fast.

The library syntax is **lllint**(x).

3.8.40 lllkerim(x): LLL algorithm applied to the columns of the (non necessarily square) matrix x which is assumed to have integral entries. Here the columns can be linearly dependent. The result is a two component vector of matrices, the columns of the first matrix representing a basis of the integer kernel of x (not necessarily LLL-reduced) and the columns of the second matrix being an LLL-reduced $\mathbf{Z}$-basis of the image of the matrix x.

The library syntax is **lllkerim**(x).

3.8.41 lllrat(x): same as LLL except that the computations are all done in rational numbers. Hence no risk of numerical instability, but extremely slow. This function is essentially superseded by `lllint` so will soon disappear.

The library syntax is **lllrat**(x).

3.8.42 mat(x): transform the object x into a matrix. If x is not a vector or a matrix, this creates a 1×1 matrix. If x is a row (resp. column) vector, this creates a 1-row (resp. 1-column) matrix. If x is already a matrix, a copy of x is created.

This function can be useful in connection with the function `concat` (see above).

The library syntax is **gtomat**(x).

3.8.43 matextract(x, y, z): extraction of a matrix from the matrix x, the line mask or vector y and the column mask or vector z. x must be a matrix, and y and z either numbers of PARI type integer or vectors. The extraction is done using the same rules as for `extract` (see above). The difference with `extract` is that both lines and columns are extracted.

The library syntax is **matextract**(x, y, z).

3.8.44 matinvr(x): computes the inverse matrix of x assumed to have real entries. This may be more efficient than using `ginv` (or x^{-1} under GP).

The library syntax is **invmatreel**(x). A related function is **invmulmatreel**(x, y) which directly computes $x^{-1} * y$ for matrices x and y.

3.8.45 matsize(x): x being a vector or matrix, returns a row vector with two components, the first being the number of rows (1 for a row vector), the second the number of columns (1 for a column vector).

The library syntax is **matsize**(x).

3.8.46 matrixqz(x, p): x being an $m \times n$ matrix with $m \geq n$ with rational or integer entries and of maximal rank, this function returns a matrix having only integral entries, having the same image as x (so is which is obtained from x by right multiplication with an $n \times n$ invertible matrix) and such that the GCD of all its $n \times n$ subdeterminants is equal to 1 when p is equal to 0, or non divisible by p otherwise. Here p must be a prime number (when it is nonzero). However, if the function is used when p has no small prime factors, it will either work or give the message "impossible inverse modulo" and a nontrivial divisor of p.

The library syntax is **matrixqz**(x, p).

3.8.47 matrixqz2(x): x being an $m \times n$ matrix with rational or integer entries, this function returns a matrix whose columns form a basis of the lattice equal to $\mathbf{Z}^n$ intersected with the lattice generated by the columns of x.

The library syntax is **matrixqz2**(x).

3.8.48 matrixqz3(x): x being an $m \times n$ matrix with rational or integer entries, this function returns a matrix whose columns form a basis of the lattice equal to $\mathbf{Z}^n$ intersected with the $\mathbf{Q}$-vector space generated by the columns of x.

The library syntax is **matrixqz3**(x).

3.8.49 minim(x, b, m): x being a square and symmetric matrix with integer entries representing a positive definite quadratic form, gives the number of vectors of square norm less than or equal to b (for the norm defined by x), and at most m of these vectors. The result is a three component vector, the first component being the number of vectors, the second being the maximum norm found, and the last vector is a matrix whose columns are the vectors found (at most m).

In the special case where $b = 0$, the function searches for the minimal non-zero vectors.

The library syntax is **minim**(x, b, m)

3.8.50 perf(x): x being a square and symmetric matrix with integer entries representing a positive definite quadratic form, looks for the perfection of the form: gives the rank of the family of the s symetric matices $v_i v_i^t$, the v_i $(1 \leq i \leq s)$ beeing the minimal vectors.

The library syntax is **perf**(x)

3.8.51 pascal(x): creates as a matrix the lower triangular pascal triangle of order $x + 1$ (i.e. with binomial coefficients up to x).

The library syntax is **pascal**(x), where x is a 32-bit C-integer.

3.8.52 rank(x): rank of the matrix x.

The library syntax is **rank**(x), and the result is a 32-bit C-integer.

3.8.53 signat(x): signature of the quadratic form represented by the symmetric matrix x. The result is a two-component vector.

The library syntax is **signat**(x)

3.8.54 smith(x): if x is a (singular or non-singular) square matrix, outputs the vector of elementary divisors of x (i.e. the diagonal of the Smith normal form of x)

The library syntax is **smith**(x).

3.8.55 smith2(x): if x is a (singular or non-singular) square matrix, outputs a 2-component vector $[u, v]$ where u and v are two unimodular matrices such that uxv is the diagonal mattrix whose diagonal is `smith(x)`.

The library syntax is **smith2**(x).

3.8.56 sort(x): sort the vector x in ascending order, using the heapsort method. x must be a vector, and its components integers, reals, or fractions. A related function is `indsort` (see above) which gives the indexes of the sorted vector in terms of the initial one. For example, `extract(x,indsort(x))` is equivalent to `sort(x)`.

The library syntax is **sort**(x).

3.8.57 sqred(x): decomposition into squares of the quadratic form represented by the symmetric matrix x. The result is a matrix whose diagonal entries are the coefficients of the squares, and the non diagonal entries represent the bilinear forms.

The library syntax is **sqred**(x).

3.8.58 supplement(x): assuming that the columns of the matrix x are linearly independent (if they are not, an error message is issued), find a square invertible matrix whose first columns are the columns of x, i.e. supplement the columns of x to a basis of the whole space.

The library syntax is **suppl**(x).

3.8.59 trace(x): This applies to quite general x. If x is not a matrix, it is equal to the sum of x and its conjugate, except for polymods where it is the trace as an algebraic number.

For x a square matrix, it is the ordinary trace. If x is a non square matrix (but not a vector), an error occurs.

The library syntax is **trace**(x).

3.8.60 trans(x) or x~: transpose of x. This has an effect only on vectors and matrices.

The library syntax is **gtrans**(x).

3.8.61 vecsort(x, k): sort the vector of vectors or matrix x in ascending order of the k-th component, using the heapsort method. The components being compared must be of type integer, real or fraction. k can also be a vector, in which case the sorting is done lexicographically according to the components listed in the vector k. For example, if $k = [2, 1, 3]$, sorting will be done with respect to the 2nd-component, and when these are equal with respect to the 1st, and when these are equal with respect to the 3rd.

The library syntax is **vecsort**(x, k).

3.9 Sums, products, integrals and similar functions.

Although the GP calculator is programmable, it is useful to have preprogrammed a number of loops, including sums, products, and a certain number of recursions. Also, a number of functions from numerical analysis like numerical integration and summation of series will be described here.

One of the parameters in these loops must be the control variable, hence a simple variable name. The last parameter must be any legal PARI expression, including of course expressions using loops. Since it is much easier to program directly the loops in library mode, these functions are mainly useful for GP programming. The use of these functions in library mode is a little tricky and its explanation will be omitted, although the reader can try and figure it out by himself by reading the source code. A brief explanatory sketch is given in section 11. Hence in this section we only give the library syntax, with no semantic explanation.

The letter X will always denote any simple variable name, and represents the formal parameter used in the function.

3.9.1 divsum$(n, X, expr)$: sum of expression $expr$ over the positive divisors of n.

In the present version 1.38.1, n is restricted to being less than 2^{31}.

The library syntax is **divsomme**`(entree *ep, GEN num, char *ch)`.

3.9.2 hvector$(n, X, expr)$: creates a row vector with n components whose components are the expression $expr$ evaluated at the integral points between 1 and n.

The library syntax is **vecteur**`(entree *ep, GEN nmax, char *ch)`.

3.9.3 (numerical) integration: A number of Romberg-like integration methods are implemented. The user should not require too much accuracy: 18 or 28 decimal digits is OK, but not much more. In addition, analytical cleanup of the integral must have been done: there must be no singularities in the interval or at the boundaries. In practice this can be accomplished with a simple change of variable. Furthermore, for improper integrals, where one or both of the limits of integration are plus or minus infinity, the function must decrease sufficiently rapidly at infinity. This can often be accomplished through integration by parts.

Note that infinity can be represented with essentially no loss of accuracy by 1e4000. However beware of real underflow when dealing with rapidly decreasing functions. For example, if one wants to compute the $\int_0^\infty e^{-x^2}\,dx$ to 28 decimal digits, one should set infinity equal to 10 for example, and certainly not to 1e4000.

The integrand may have values belonging to a vector space over the real numbers; in particular, it can be complex-valued or vector-valued.

3.9.4 intgen$(X = a, b, expr)$: general driver routine for doing numerical integration from a to b of the expression $expr$, with respect to the formal variable X.

The library syntax is **rombint**`(entree *ep, GEN a, GEN b, char *ch, long int prec)`.

3.9.5 intinf$(X = a, b, expr)$: numerical integration from a to b, tailored for being used when a or b are infinite. One *must* have $ab > 0$, and in fact if for example $b = \infty$, then it is preferable to have a as large as possible, at least $a \geq 1$.

The library syntax is **qromi**`(entree *ep, GEN a, GEN b, char *ch, long int prec)`.

3.9.6 intnum$(X = a, b, expr)$: simple numerical integration from a to b. This is the fastest method, and should be used when a and b are not too large, the function is smooth, and can be evaluated exactly everywhere on the interval $[a, b]$.

The library syntax is **qromb**`(entree *ep, GEN a, GEN b, char *ch, long int prec)`.

3.9.7 intopen$(X = a, b, expr)$: numerical integration from a to b. This method should be used, again when a and b are not too large, but when the function is now allowed to be undefined (but continuous) at a or b, for example the function $\sin(x)/x$ at $x = 0$.

The library syntax is **qromo**`(entree *ep, GEN a, GEN b, char *ch, long int prec)`.

Two summation methods are also implemented, for alternating series, and for series with terms of the same sign (see `sumalt` and `sumpos` below).

3.9.8 matrix$(m, n, X, Y, expr)$: creation of the $m \times n$ matrix whose coefficients are given by the expression *expr*. There are two formal parameters in *expr*, the first one (X) corresponding to the rows, the second (Y) to the columns, and X goes from 1 to m, Y goes from 1 to n.

The library syntax is **matrice**`(entree *ep1, entree *ep2, GEN nlig, GEN ncol, char *ch)`.

3.9.9 prod$(x, X = a, b, expr)$: product of expression *expr*, initialized at x, the formal parameter X going from a to b. As for `sum`, the initialization parameter x must be given: its main purpose is to force the type of the operations being performed. For example if it is put equal to the integer 1, operations will start being done exactly. If it is put equal to the real 1., they will be done using real numbers having the default precision. If it is put equal to the power series $1 + O(X^k)$ for a certain k, they will be done using power series of precision at most k. These are the three most common initializations.

As an extreme example, compare

$$\mathrm{prod}(1, j = 1, 100, (1 - X\hat{\ }j))$$

with

$$\mathrm{prod}(1 + O(X\hat{\ }101), j = 1, 100, (1 - X\hat{\ }j)).$$

The library syntax is **produit**`(entree *ep, GEN x, GEN a, GEN b, char *ch)`.

3.9.10 prodeuler$(X = a, b, expr)$: product of expression *expr*, initialized at 1., the formal parameter X ranging over the prime numbers between a and b.

The library syntax is **prodeuler**`(entree *ep, GEN a, GEN b, char *ch, long int prec)`.

3.9.11 prodinf$(X = a, expr)$: infinite product of expression *expr*, the formal parameter X starting at a. The evaluation stops when the relative error of the expression minus 1 is less than the default precision. The expressions must always evaluate to an element of $\mathbf{C}$.

The library syntax is **prodinf**`(entree *ep, GEN a, char *ch, long int prec)`.

3.9.12 prodinf1$(X = a, expr)$: infinite product of expression $1 + expr$, the formal parameter X starting at a. The evaluation stops when the relative error of the expression is less than the default precision. The expressions must always evaluate to an element of $\mathbf{C}$.

The library syntax is **prodinf1**`(entree *ep, GEN a, char *ch, long int prec)`.

3.9.13 solve$(X = a, b, expr)$: real root of expression *expr* between a and b, under the condition $expr(X = a) * expr(X = b) \leq 0$. The method used is Brent's method.

The library syntax is **zbrent**`(entree *ep, GEN a, GEN b, char *ch, long int prec)`.

3.9.14 sum$(x, X = a, b, expr)$: sum of expression *expr*, initialized at x, the formal parameter going from a to b. As for `prod`, the initialization parameter x must be given: its main purpose is to force the type of the operations being performed. For example if it is put equal to the integer

0, operations will start being done exactly. If it is put equal to the real 0., they will be done using real numbers having the default precision. If it is put equal to the power series $O(X^k)$ for a certain k, they will be done using power series of precision at most k. These are the three most common initializations.

As an extreme example, compare

$$\mathrm{sum}(0, j = 1, 1000, 1/j)$$

with

$$\mathrm{sum}(0., j = 1, 1000, 1/j).$$

The library syntax is **somme**`(entree *ep, GEN x, GEN a, GEN b, char *ch)`.

3.9.15 sumalt($X = a, expr$): numerical summation of the series *expr*, which should be an alternating series, the formal variable X starting at a. The algorithm used is Euler's method of finite differences combined with a trick due to van Wijngaarden. Beware that the stopping criterion is that the term gets small enough, hence terms which are equal to 0 will create problems and should be removed.

Divergent alternating series can sometimes be summed by this method, as well as series which are not exactly alternating (see for example section 11).

The library syntax is **sumalt**`(entree *ep, GEN a, char *ch, long int prec)`.

3.9.16 suminf($X = a, expr$): infinite sum of expression *expr*, the formal parameter X starting at a. The evaluation stops when the relative error of the expression is less than the default precision. The expressions must always evaluate to an element of **C**.

The library syntax is **suminf**`(entree *ep, GEN a, char *ch, long int prec)`.

3.9.17 sumpos($X = a, expr$): numerical summation of the series *expr*, which must be a series of terms having the same sign, the formal variable X starting at a. The algorithm used is van Wijngaarden's trick for converting such a series into an alternating one, and is quite slow. Beware that the stopping criterion is that the term gets small enough, hence terms which are equal to 0 will create problems and should be removed.

The library syntax is **sumpos**`(entree *ep, GEN a, char *ch, long int prec)`.

3.9.18 vector($n, X, expr$): creates a row vector with n components whose components are the expression *expr* evaluated at the integral points between 1 and n (identical with `hvector`).

The library syntax is **vecteur**`(entree *ep, GEN nmax, char *ch)`.

3.9.19 vvector($n, X, expr$): creates a column vector with n components whose components are the expression *expr* evaluated at the integral points between 1 and n.

The library syntax is **vvecteur**`(entree *ep, GEN nmax, char *ch)`.

3.10 Plotting functions.

Although plotting is not the main purpose of Pari, a number of plotting functions are provided. These functions are either high-level plotting functions (all the functions starting with `plot`) in which the user has little to do but explain what type of plot he wants, and whose syntax is similar to the one used in the preceding section. Or they can be low-level plotting functions, where every drawing primitive (point, line, box, etc...) must be specified by the user. These low-level functions (called rectplot functions) work as follows. You have at your disposal 16 virtual windows which are filled independently, and can then be physically ORed on a single window at user-defined positions. These rectwindows are numbered from 0 to 15, and must be initialized before being used by the function `initrect`, which specifies the height and width of the virtual window. At all times, a virtual cursor (initialized at $[0,0]$) is associated to the window, and its current value can be obtained using the function `cursor`.

A number of primitive graphic objects can then be drawn in these windows, and only the part of the object which is inside the window will be drawn, with the exception of polygons and strings which are drawn entirely (but the virtual cursor can move outside of the window). Finally, the actual physical drawing is done using the function `draw`. Note that the windows are preserved so that further drawings using the same windows at different positions or different windows can be done without extra work. If you want to erase a window (and free the corresponding memory), use the function `killrect`. It is not possible to partially erase a window. Erase it completely, initialize it again and then fill it with the graphic objects that you want to keep.

In addition to initializing the window, you may want to have a scaled window to avoid unnecessary conversions. For this, use the function `scale` below. As long as this function is not called, the scaling is simply the number of pixels, the origin being at the upper left and the y-coordinates going downwards.

Finally, note that in the same way that `texprint` allows you to have a TeX output corresponding to printed results, the functions starting with `post` allow you to have `postscript` output of the plots. This will not be absolutely identical with the screen output, but will be sufficiently close. Note that you can use postscript output even if you do not have the plotting routines enabled. The postscript output is written in a file called `pari.ps`. Each time a new postscript output is asked for the postscript output is appended to that file. Hence the user must remove this file first if he does not want unnecessary drawings from preceding sessions to appear. On the other hand, in this manner as many plots as desired can be kept in a single file.

3.10.1 box$(w, x2, y2)$: let $(x1, y1)$ be the current position of the virtual cursor. Draw in the rectwindow w the outline of the rectangle which is such that the points $(x1, y1)$ and $(x2, y2)$ are opposite corners. Only the part of the rectangle which is in w is drawn. The virtual cursor does *not* move.

The library syntax is **rectbox**$(w, x2, y2)$ where w is a C-integer.

3.10.2 cursor(w): give as a 2-component vector the current (scaled) position of the virtual cursor corresponding to the rectwindow w.

The library syntax is **rectcursor**(w) where w is a C-integer.

3.10.3 draw$(list)$: physically draw the rectwindows given in $list$ which must be a vector whose number of components is divisible by 3. If $list = [w1, x1, y1, w2, x2, y2, \ldots]$, the windows $w1$, $w2$, etc... are physically placed with their upper left corner at physical position $(x1, y1)$, $(x2, y2)$, $\ldots$respectively, and are then drawn together. Overlapping regions will thus be drawn twice, and the windows are considered transparent.

The library syntax is **rectdraw**$(list)$.

Note that in the present version 1.38.1 this function has been written for the X11-window system (hence also for GUI's based on X11 such as Openwindows and Motif), and for Sunview/Suntools.

3.10.4 initrect(w, x, y): initialize the rectwindow w to width x and height y, and position the virtual cursor at $(0, 0)$.

The library syntax is **initrect**(w, x, y) where all 3 arguments are C-integers.

3.10.5 killrect(w): erase rectwindow w and free the corresponding memory. Note that if you want to use the rectwindow w again, you must use `initrect` first to specify the new size.

The library syntax is **killrect**(w) where w is a C-integer.

3.10.6 line$(w, x2, y2)$: let $(x1, y1)$ be the position of the virtual cursor. Draw in the rectwindow w the part of the segment $(x1, y1) - (x2, y2)$ which is inside w, and move the virtual cursor to $(x2, y2)$ (even if it is outside the window). If you want to draw a line from $(x1, y1)$ to $(x2, y2)$ where $(x1, y1)$ is not necessarily the position of the virtual cursor, use `move(w,x1,y1)` before using the `line` function.

The library syntax is **rectline**$(w, x2, y2)$ where w is a C-integer.

3.10.7 lines$(w, listx, listy)$: draw on the rectwindow w the polygon such that the (x,y)-coordinates of the vertices are in the vectors of equal length $listx$ and $listy$. The virtual cursor does not move. This is basically the same function as the function `ploth2`, but with no scaling factors. Note that to obtain a closed polygon, the first and last point must be the same. Finally, for simplicity, the whole polygon is drawn, not only the part of the polygon which is inside the rectwindow.

The library syntax is **rectlines**$(w, listx, listy)$ where w is a C-integer.

3.10.8 move(w, x, y): move the virtual cursor of the rectwindow w to position (x, y).

The library syntax is **rectmove**(w, x, y) where w is a C-integer.

3.10.9 plot$(X = a, b, expr)$: crude plot of the function represented by expression $expr$ from a to b.

The library syntax is **plot**`(entree *ep, GEN a, GEN b, char *ch)`.

3.10.10 ploth$(X = a, b, expr)$: high precision plot of the function $y = f(x)$ represented by the expression $expr$, x going from a to b. Since this involves around 1000 function calls, it is advised to keep the current precision to a minimum (i.e. 9) before calling `ploth`.

Note that in the present version 1.38.1 this function has been written for the Macintosh, the X11-window system (hence also for GUI's based on X11 such as Openwindows and Motif), and for Sunview/Suntools. An Atari/gem version is also available upon request.

The library syntax is **ploth**`(entree *ep, GEN a, GEN b, char *ch, long int prec)`.

3.10.11 ploth2$(T = a, b, expr)$: high precision plot of the curve represented in parametric form by $x = f(t), y = g(t)$, represented by the expression $expr$, t going from a to b. The expression must be a two component vector. Since this involves around 2000 function calls, it is advised to keep the current precision to a minimum (i.e. 9) before calling `ploth2`.

Note that in the present version 1.38.1 this function has been written for the Macintosh, the X11-window system (hence also for GUI's based on X11 such as Openwindows and Motif), and for Sunview/Suntools. An Atari/gem version is also available upon request.

The library syntax is **ploth2**`(entree *ep, GEN a, GEN b, char *ch, long int prec)`.

3.10.12 plothraw$(listx, listy)$: high precision plot of the points whose (x,y)-coordinates are in the vectors of equal length listx and listy. Automatic positioning and scaling is done, but with the same scaling factor on x and y.

The library syntax is **plothraw**$(listx, listy)$.

Note that in the present version 1.38.1 this function has been written for the X11-window system (hence also for GUI's based on X11 such as Openwindows and Motif), and for Sunview/Suntools.

3.10.13 point(w, x, y): draw the point (x, y) on the rectwindow w (if it is inside w), and in any case move the virtual cursor to position (x, y).

The library syntax is **rectpoint**(w, x, y) where w is a C-integer.

3.10.14 points$(w, listx, listy)$: draw on the rectwindow w the points whose (x,y)-coordinates are in the vectors of equal length $listx$ and $listy$ and which are inside w. The virtual cursor does not move. This is basically the same function as the function `plothraw`, but either with no scaling factor or with a scale chosen using the function `scale`.

The library syntax is **rectpoints**$(w, listx, listy)$ where w is a C-integer.

3.10.15 postdraw$(list)$: same as `draw`, except that the output is a postscript program appended to the file `pari.ps`.

The library syntax is **postdraw**$(list)$.

3.10.16 postploth$(X = a, b, expr)$: same as `ploth`, except that the output is a postscript program appended to the file `pari.ps`.

The library syntax is **postploth**`(entree *ep, GEN a, GEN b, char *ch)`.

3.10.17 postploth2$(T = a, b, expr)$: same as `ploth2`, except that the output is a postscript program appended to the file `pari.ps`.

The library syntax is **postploth2**`(entree *ep, GEN a, GEN b, char *ch)`.

3.10.18 postplothraw$(listx, listy)$: same as `plothraw`, except that the output is a postscript program appended to the file `pari.ps`.

The library syntax is **postplothraw**$(listx, listy)$.

3.10.19 rbox(w, dx, dy): draw in the rectwindow w the outline of the rectangle which is such that the points $(x1, y1)$ and $(x1 + dx, y1 + dy)$ are opposite corners, where $(x1, y1)$ is the current position of the cursor. Only the part of the rectangle which is in w is drawn. The virtual cursor does *not* move.

The library syntax is **rectrbox**(w, dx, dy) where w is a C-integer.

3.10.20 rline(w, dx, dy): draw in the rectwindow w the part of the segment $(x1, y1) - (x1 + dx, y1 + dy)$ which is inside w, where $(x1, y1)$ is the current position of the virtual cursor, and move the virtual cursor to $(x1 + dx, y1 + dy)$ (even if it is outside the window).

The library syntax is **rectrline**(w, dx, dy) where w is a C-integer.

3.10.21 rmove(w, dx, dy): move the virtual cursor of the rectwindow w to position $(x1 + dx, y1 + dy)$, where $(x1, y1)$ is the initial position of the cursor (i.e. to position (dx, dy) relative to the initial cursor).

The library syntax is **rectrmove**(w, dx, dy) where w is a C-integer.

3.10.22 rpoint(w, dx, dy): draw the point $(x1 + dx, y1 + dy)$ on the rectwindow w (if it is inside w), where $(x1, y1)$ is the current position of the cursor, and in any case move the virtual cursor to position $(x1 + dx, y1 + dy)$.

The library syntax is **rectrpoint**(w, dx, dy) where w is a C-integer.

3.10.23 scale$(w, x1, x2, y1, y2)$: scale the local coordinates of the rectwindow w so that x goes from $x1$ to $x2$ and y goes from $y1$ to $y2$ ($x2 < x1$ and $y2 < y1$ being allowed). Initially, after the initialization of the rectwindow w using the function `initrect`, the default scaling is the graphic pixel count, and in particular the y axis is oriented downwards since the origin is at the upper left. The function `scale` allows to change all these defaults and should be used whenever functions are graphed.

The library syntax is **rectscale**$(w, x1, x2, y1, y2)$ where w is a C-integer.

3.10.24 string(w, x): draw on the rectwindow w at the current position of the cursor the string corresponding to x. x can either be a string enclosed in quotes ("), or x can be a number which can be converted into a real number. In that case it is printed in the format 9.3lf, i.e. 9 characters before the decimal point, and 3 after.

The library syntax is **rectstring**(w, x) where w is a C-integer and x is a GEN which either represents a vector of ASCII values, or can be converted into a real number.

3.11 Programming under GP and user-defined functions.

3.11.1 Variables and symbolic expressions.

In GP you can use up to 256 variable names. These names can be any standard identifier names, i.e. they must start with a letter and contain only alphanumeric characters. To avoid confusion with other symbols, you must not use non alphanumeric symbols like '_', '$', or '.'. In addition to the function names which you must not use (see the list with \c) there are exactly three special variable names which you are not allowed to use: **pi** and **euler**, which represent well known constants, but also **i** which is always equal to the square root of -1. Since usually you are not going to use the variable name `pi` for anything else but the number π, and similarly for `euler`, these two will not create any problems. On the other hand, you will have to get used to calling a loop control variable with a name different from `i`. We realize that this may initially be the cause for some confusion, but we do not have any replacement symbol for the square root of -1.

Now the main thing to understand is that PARI/GP is *not* a symbolic manipulation package, although it shares some of the functionalities. One of the main consequences of this fact is that all expressions are evaluated as soon as they are written, they never stay in a purely abstract form. As an important example, consider what happens when you use a variable name *before* assigning a value into it. This is perfectly acceptable to GP, who considers this variable in fact as a polynomial of degree 1, with coefficients 1 in degree 1, 0 in degree 0, whose variable is the variable name you used.

If later you assign a value to that variable, the objects which you have created before will still be considered as polynomials. If you want to obtain their value, use the function `eval` (see section 7 above).

Another consequence is that the variables are numbered in the order that they appear, and the main variable of an expression is always the lowest numbered variable. Hence if you are working with expressions involving several variables and want to have them ordered in a specific manner *in the internal representation*, the simplest is just to write down the variables one after the other under GP before starting any real computations. If you already have started working and want to change the names of the variables in an object, use the function `changevar` (see section 2). If you only want to have them ordered when the result is printed, you can also use the function `reorder` (see below).

Finally, note that if x is a vector, you can assign a result to $x[m]$ (i.e. write something like $x[k] = expr$). If x is a matrix, you can assign a result to $x[m,n]$, but *not* to $x[m]$, which is a vector. If you want to assign an expression to the m-th column of a matrix x, use $x[,m] = expr$ instead. Similarly, use $x[m,] = expr$ to assign an expression to the m-th row of x.

3.11.2 Expressions and expression sequences.

An expression is formed by combining the GP operators, functions (including user-defined functions) and control statements. It may be preceded by an assignment statement '=' into a variable. It always has a value, which can be any PARI object.

Several expressions can be combined on a single line by separating them with semicolons (';') and also with colons (':') for those who are used to BASIC. Such an expression sequence will be called simply a *seq*. A *seq* also has a value, which is the value of the last nonempty expression in the sequence. Under GP, the value of the *seq* is always put on the stack (i.e. it will become the next object $\%n$), and only that value. The values of the other expressions in the *seq* are discarded after the execution of the *seq* is complete, except of course if they were assigned into variables. In addition, the value of the *seq* (or of course of an expression if there is only one) is printed if the line does not end with a semicolon (';').

3.11.3 Control statements.

A number of control statements are available under GP. They are simpler and have a slightly different syntax than their C counterparts, but are quite powerful enough to write any kind of program. Some of them are specific to GP, since they are made for number theorists. They are as follows. As usual, X will denote any simple variable name, and *seq* will always denote a sequence of expressions, including the empty sequence.

3.11.3.1 for$(X = a, b, seq)$: the formal variable X going from a to b, the *seq* is evaluated. Nothing is done if $a > b$. a and b must be in $\mathbf{R}$.

3.11.3.2 fordiv(n, X, seq): the formal variable X ranging through the positive divisors of n, the sequence *seq* is evaluated. n must be of type integer.

3.11.3.3 forprime$(X = a, b, seq)$: the formal variable X ranging over the prime numbers between a to b (including a and b if they are prime), the *seq* is evaluated. Nothing is done if $a > b$. Note that a and b must be in $\mathbf{R}$.

3.11.3.4 forstep$(X = a, b, s, seq)$: the formal variable X going from a to b, in increments of s, the *seq* is evaluated. Nothing is done if $s > 0$ and $a > b$ or if $s < 0$ and $a < b$. a, b and s must be in $\mathbf{R}$, and s must be nonzero.

3.11.3.5 goto(a): a being an integer between 0 and 99, go to the position in the seq referred to by label a. This can be either a forward skip or a backwards skip. In case of multiple labels, control is transferred in the following manner. If one such label is to the left of the goto instruction, control is transferred to the rightmost one among those which are to the left. Otherwise, control is transferred to the rightmost label with a given number. In any case, multiple labels in a given seq should be avoided.

Also, good programming practice implies that the `goto` instruction should be used only in special cases, such as getting out of a deeply nested set of control instructions in one step. In fact, in the present version 1.38.1, the `goto` instruction almost always leads to syntax errors hence should be avoided, except as a means to stop a program catastrophically without getting out of GP. This may be corrected in a future release.

Note that usually, it is possible to use the `goto` instruction with an a which is not an absolute constant, but may vary in the seq ("computed goto").

3.11.3.6 if($a, seq1, seq2$): if a is nonzero, the expression sequence $seq1$ is evaluated, otherwise the expression $seq2$ is evaluated. Of course, $seq1$ or $seq2$ may be empty, but the syntax must stay the same: `if(a,seq,)` evaluates seq if a is not equal to zero, does nothing otherwise; `if(a,,seq)` evaluates seq if a is equal to zero, does nothing otherwise.

3.11.3.7 label(a): a being an integer between 0 and 99, label this position as label number a for use by the `goto` instruction. Labels are specific to a given seq.

3.11.3.8 until(a, seq): evaluate expression sequence seq until a is not equal to 0 (i.e. until a is true). If a is initially not equal to 0, seq is evaluated once (more generally, the condition on a is tested *after* execution of the seq, not before as in `while`).

3.11.3.9 while(a, seq): while a is nonzero evaluate the expression sequence seq. The test is made *before* evaluating the seq, hence in particular if a is initially equal to zero the seq will not be evaluated at all.

3.11.4 Specific functions used in GP programming.

In addition to the general PARI functions, it is necessary to have some functions which will be of use specifically for GP. They are as follows:

3.11.4.1 kill(x): kills the present value of the variable or user-defined function x. After `kill` of a variable, since the variable does not have a value, as has been explained above it is again considered as a monic polynomial of degree 1 with no constant term.

For the following four printing functions, *list* represents a list (separated by commas) either of PARI objects or of character strings between double quotes '`"`', which are always printed as they are.

3.11.4.2 pprint(*list*): output *list* in prettyprint (beautified) format, ending with a newline.

3.11.4.3 pprint1(*list*): output *list* in prettyprint (beautified) format, without ending with a newline.

3.11.4.4 print(*list*): output *list* in raw format, ending with a newline.

3.11.4.5 print1(*list*): output *list* in raw format, without ending with a newline.

3.11.4.6 read(): reads a string, interpreted as a GP expression, from the input file, usually standard input (i.e. the keyboard). If a sequence of expressions is given, the result is the result of the last expression of the sequence. When using this instruction, it is useful to prompt for the string by using the `print1` function. Note that in the present version 2.12 of `pari.el`, when using GP under gnuemacs (see section 11) one *must* prompt for the string, with a string which ends with the same prompt as the current one (usually a '?').

3.11.4.7 reorder(x): x must be a vector. If x is the empty vector, this gives the vector whose components are the existing variables in increasing order (i.e. in decreasing importance). If x is nonempty, it must be a permutation of variable names, and this permutation gives a new order of importance of the variables. For example, if the existing order is `[x,y,z]`, then after `reorder([z,x])` the order of importance of the variables will be `[z,y,x]`.

3.11.4.8 texprint(*list*): outputs *list* in TEX format. This output can then be used in a TEX manuscript.

Warning: in the present version 1.38.1, the printing is done on the standard output, hence to be able to get the output into a file, you must be either in an emacs session or shell (see section 11), in a windowing system (which enables you to grab the output with the mouse), under a UNIX script command, or with the logfile enabled (see the command \l). This will be improved.

3.11.5 User defined functions.

It is very easy to define a new function under GP, which can then be used like any other function. The syntax is as follows:

```
name(list of true formal variables, list of local variables)=seq
```

where `name` is the name that you want to give to your function (same syntactic restrictions as for variable names). `list of true formal variables` is the list of variables corresponding to those which you will actually use when calling your function, the variables being separated by commas, and the list being allowed to be empty. `list of local variables` is the list of the additional local variables which you will use in the definition of the function. If you omit some or all of these local variable declarations, the function will probably still work, but the nondeclared variables will become global, hence known outside of the function, and this may have undesirable side-effects. On the other hand, in some cases it may also be what you want. Finally, as usual `seq` is any expression sequence.

Once the function is defined using the above syntax, you can use it like any other function. In addition, you can also recall its definition exactly as you do for predefined functions, that is by writing `?name`. One small difference with predefined functions is that you can never redefine such a function, while you can redefine a user-defined function as many times as you want, without using the `kill` instruction.

In a given session you can give any identifier name to a function, *except* those of predefined functions (of course) but also those of any variables that have been used, even if they have been killed. On the other hand, if you want to use as a variable name the name of a user-defined function, it is enough (and necessary) to kill this function name first.

An amusing example of a user-defined function is the following. It is intended to illustrate both the use of user-defined functions and the power of the `sumalt` function. Although since version 1.34 the Riemann zeta-function is included in the standard functions, let us assume that this is not the case (or that we want another implementation). One way to define it, which is probably the simplest (but certainly not the most efficient), is as follows:

```
zet(s,j)=sumalt(j=1,(-1)^(j-1)*j^(-s))/(1-2^(1-s))
```

Then this gives reasonably good accuracy and speed as long as you are not too far from the domain of convergence. Try it for s integral between -5 and 5, say, or for $s = 0.5 + i * t$ where $t = 14.134\ldots$ Of course, the call to the function is done by `zet(s)`, the variable `j` must not be given.

3.11.6 Special editing characters. A GP program can of course have more than one line. Since GP executes your commands as soon as you have finished typing them, there must be a way to tell it to wait for the next line or lines of input before doing anything. There are two ways of doing this.

The first one is simply to use the backslash character '\' at the end of the line that you are typing, just before hitting <return>. This tells GP that what you will write on the next line is the physical continuation of what you have just written. In other words, it makes GP forget your newline character. For example if you use this while defining a function, and if you ask for the definition of the function using `?name`, you will see that your backslash has disappeared and that everything is on the same line. You can type a \ anywhere. It will be interpreted as above only if it is immediately followed by a newline. For example, you can type

```
3+\
4
```

instead of typing `3+4`.

The second one cannot be used everywhere, but is in general more useful. It is the use of braces '{' and '}'. When GP sees an opening brace ('{') at the beginning of a line, it understands that you are writing a program, and newlines will be ignored (but registered, contrary to when you use a backslash) until you type a closing brace '}'. However, there is an important (but easily obeyed) restriction: inside an open brace-close brace pair, all newlines must occur after a semicolon (';'), i.e. between two expressions forming a *seq*.

Of course you can combine the use of backslash and braces.

3.11.7 The GP/PARI programming language.

As we have seen, the GP calculator uses a primitive purely interpreted language. The structure of this language is in fact more reminiscent of LISP with a functional notation, `f(x,y)` rather than `(f x y)`: all programming constructs, such as `if, while,` etc... are functions, and the main loop does not really execute, but rather evaluates (sequences of) expressions.

Of course, it is by no means a true LISP. Function names are distinct from variable names: once a name has been used as a variable name, this cannot be changed in the same session. Predefined function names cannot be used for anything else, and the number of actual parameters must match the declaration given in the online help. Identifiers that are used for user-defined functions can be freely redefined for other functions, or totally forgotten (using `kill`) and later reused for a variable, for example.

Each variable has a stack of values, implemented as a linked list. When a new scope is entered, during a function call, the value of the actual parameter is pushed on the stack. If the parameter is not supplied, a special 0 value called **gnil** is pushed on the stack (this value is not printed if it is returned as the result of a GP expression sequence). Upon exit, the stack decreases. You can `kill` a variable, decreasing the stack yourself. This should be used only at the top level of GP, to undo the effect of an assignment, not from a function. However, the stack has a bottom: the value of a variable is the monomial of degree 1 in this variable, as is natural for a mathematician.

Note that the iterative constructs which use a variable name (`for, fordiv, forprime, prod, sum, vector, matrix, plot,` etc.) also consider the given variable to be local to the construct. A value is pushed on entry and pulled on exit. So, it is not necessary for a function using such a construct to declare the variable as a dummy formal parameter. In particular, in our `zet` example above, the variable `j` need not be declared.

Otherwise, it is strongly recommended to declare in this way all other variables that are used inside a function: If a function accesses a variable which is not one of its formal parameters, the value used will be the one which happens to be on top of the stack at the time of the call. This could be a "global" value, or a local value belonging to any function higher in the call chain. So, be warned.

Implementation note. For the curious reader, here is how these stacks are handled: a hashing function is computed from the identifier, and used as an index in `hashtable`, a table of pointers. Each of these pointers begins a linked list of structures (type `entree`). The linked list is searched linearly for the identifier (each list will have less than 7 components or so). When the correct `entree` is found, it points to the top of the stack of values for that identifier if it is a variable, to the function itself if it is a predefined function, and to a copy of the text of the function if it is a user-defined function. When an error occurs, all of this maze (rather a tree, in fact) is searched and (hopefully) replaced in the situation preceding the last call of the main evaluator.

3.12 Using GP under gnuemacs.

Thanks to the help of Annette Hoffman from the university of Saarbrücken, and since version 1.34.05 from David Carlisle from the university of Manchester, it is possible and in fact desirable to use GP as a subprocess of gnuemacs. This is of course possible only if gnuemacs has been installed on your machine.

To use this, you must include in your `.emacs` file the following line:

```
(load "pari" nil t)
```

where `pari.el` is the name of the file that will have to be loaded by gnuemacs (if you have changed the name, or if you have the file in a different directory, you must of course supply the correct name). This file is included in the PARI distribution.

Once this is done, under gnuemacs if you type `M-x gp` (where as usual M is the Meta key, i.e. Escape, or on SUN keyboards, the Left key), a special shell will be started, which in particular launches GP with the default stack size, prime limit and input buffer size. If you type instead `C-u M-x gp`, you will be asked for the name of the GP executable, the stack size, the prime limit and the input buffer size before the execution of GP begins. If for any of these you simply type return, the default value will be used (on Unix machines it will be `/usr/local/bin/gp` for the executable, 4000000 for the stack, 500000 for the prime limit and 30000 for the buffer size).

You can then work as usual under GP, but with two notable advantages. First and foremost, you have at your disposal all the facilities of a text editor like emacs, in particular for correcting or copying blocks. Second, you can have an on-line help which is much more complete than what you obtain by typing `?name`. This is done by typing M-? (where M is the Meta key). In the minibuffer, emacs asks what function you want to describe, and after your reply you obtain the description which is in the users manual, including the description of functions (such as \, %) which use special symbols.

This help system can also be menu-driven, by using the command `M-\c` which opens a help menu window which enables you to choose the category of commands for which you want an explanation.

Finally you can use command completion in the following way. After the prompt, type the first few letters of the command, then M-TAB where TAB is the TAB key (of course not the letters T,A,B). If there exists a unique command starting with the letters you have typed, the command name will be completed. If not, either the list of commands starting with the letters you typed will be displayed in a separate window (which you can then kill by typing as usual C-x 1 or by typing in more letters), or "no match found" will be displayed in the Emacs command line.

You also have at your disposal a few other commands. Read the file pari.txt for details.

Note that if for some reason the session crashes (due to a bug in your program or in the PARI system), you will usually stay under emacs, but the GP buffer will be killed. To recover it, simply type again M-x gp (or C-u M-x gp), and a new session of GP will be started after the old one, so you can recover what you have typed. Note that this will of course *not* work if for some reason you exited emacs before coming back (except for the C-z temporary stopping command).

4.1 Introduction: initializations, universal objects, input and output.

To be able to use PARI in library mode, you must write a C program and link it to the PARI library and the PARI include files. See the installation guide (given in Appendix A) on how to create and install the PARI library and include files. A sample Makefile is also given in Appendix B.

Probably the best way to understand how programming is done is to work through a complete example. We will write such a program in section 4.3. Before doing this, a few explanations are in order.

4.1.1 Initializations and universal objects.

First, one must explain to the outside world what kind of objects and programs we are going to use. This is done simply with the statement

```
#include <genpari.h>
```

Note that this is usually a link, created when you make the library, to a file called either `genpari68k.h` (for 680x0-based machines with x at least 2) or `genpariother.h` (for all other machines).

This file `genpari.h`, first exports all the necessary constants, variables and functions, defines some important macros, and also defines the fundamental type for all PARI objects: the type **GEN**, which is simply a pointer to `long`.

Technical note: we would have liked to define a type GEN to be a pointer on itself. This unfortunately is not possible in C, except by using structures, but then the names become unwieldy. On the other hand, by a simple trick, it can be done in Pascal for example. The result of this is that when we will use a component of a PARI object, it will be a *long*, hence will need to be typecasted to a GEN again if we want to avoid warnings from the compiler. This will sometimes be quite tedious, but of course is trivially done.

To take an example, a polynomial P of degree 2 will be represented by a chunk of memory pointed to by the GEN P. $P[0]$ and $P[1]$ contain code information, in particular the type of the object, the degree of P, etc.... $P[2]$, $P[3]$, $P[4]$ contain pointers to the coefficients of degree 0, 1, and 2 of P respectively (note the ascending order). This is where typecasting will be necessary: in principle $P[i]$ (for $i = 2, 3, 4$) is a long, but we will want to use it as a GEN. The coefficients $P[i]$ themselves are in chunks of memory whose complexity depends on the types of the coefficients, and so on.

Now we must state the most important law about programming in PARI, which must be respected if one wants to avoid disasters:

Apart from universal objects (see below) **the chunks of memory used by a given PARI object must be in consecutive memory locations**.

Don't panic: let's see the reason and the meaning of this, and how it can be achieved.

When doing large computations, unwanted intermediate results clutter up memory very fast so some kind of garbage collecting is needed. Most large systems do garbage collecting when the cluttering gets heavy, and this slows down the performance. In PARI we have taken a different approach: you must do your own cleaning up as soon as the intermediate results are not needed. Special purpose routines have been written to do this, but the primary requirement is exactly as stated above: a PARI object must be (essentially) connected. As a consequence of this explanation, one also sees that there is an evident exception to the above law: if your computation is small enough so that you don't need to do any garbage collecting, then just go ahead, PARI won't mind disconnected objects in most cases. However, since PARI routines do their own garbage collecting, watch carefully what you are doing.

The notion of universal object alluded to above is quite simple: during the execution of your program, a number of objects will have been defined (by the system or by yourself) with the idea that they stay permanently with the same values. Examples are the integer 1, the fraction $\frac{1}{2}$, the polynomial X, or a prime p which is used as a base modulus for integermods or p-adics. These universal objects are of course allowed to be disconnected from the other PARI objects.

After declaring the use of the file genpari.h, the first executable statement of a main program should be to initialize the PARI system, and in particular the PARI stack which will contain all the computations. This is simply done with a call to the two variable function **init**, like `init(4000000,100000)`. The first argument (here 4 million) is the number of bytes given to PARI to work with (it should not reasonably drop under 500000), and the second is the upper limit on a precomputed prime number table. If you don't want prime numbers, just put 2, but put an argument anyway because `init()` expects one.

We have now at our disposal:

• the following universal objects: the integer 0 (**gzero** as a GEN, `zero` as a long), the integer 1 (**gun** as a GEN, `un` as a long), the integer 2 (**gdeux** as a GEN, `deux` as a long), the fraction $\frac{1}{2}$ (**ghalf** as a GEN, `lhalf` as a long), the complex number i, **gi** as a GEN. In addition, space is reserved for the polynomials 1 (**polun[]** as a GEN, `lpolun` as a long), and the polynomials x_v, (**polx[]** as GENs, `lpolx` as longs), where x_v is the name of variable number v, where $0 \leq v \leq 255$. However, they are not created upon initialization, and it is the programmer's responsibility to fill them before use. Since this is not very easy, we advise the user to use the function `lisseq` which has essentially the same effect as `lisexpr` except that it can execute a sequence of expressions and not only a single expression. For example, to prepare for use the variables a,b,c,x, write

```
lisseq("x;a;b;c")
```

Note that `polun` and `polx` are arrays, the index being the polynomial variable number.

• a large PARI stack containing nothing but (in the present version) the 167 long words (668 bytes) of the predefined universal objects.

• a heap which is dealt with in a different way from the stack, and will contain other permanent universal objects.

• a table of primes.

• access to all the built-in functions of the PARI library.

We have already described many of these functions in the preceding chapters. However some of them are specific to library mode and thus will be explained in this chapter.

4.1.2 Input and output.

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

For input, PARI provides you with two powerful high level functions which enables you to input your objects as if you were under GP. In fact, the second one *is* essentially the GP syntactical parser, hence you can use it not only for input but for any computation that you can do under GP. These functions are called **lisexpr** and **lisseq**. The first one has the following syntax:

```
GEN lisexpr(char* s);
```

Its effect is to analyze the input string s and to compute the result as in GP. However it is limited to one expression. If you want to read and evaluate a sequence of expressions, use

```
GEN lisseq(char* s);
```

Warning: there is a slight difference between these functions and the GP syntactical parser: the expressions and sequences which you use must not contain any spaces.

Once in a while, it may be useful to have the evaluation of the string involving a call to a function you have defined in C. The function **install** allows you to give a name to a function taking 0, 1, 2 or 3 GEN arguments and returning a single GEN. The syntax is

```
void install(GEN (*f)(), char *name, int valence)
```

where `f` is the (address of) the function, `name` its new name, and `valence` the number of its arguments, an integer between 0 and 3.

For output, there exist four different functions. First, you can use the function **sor** with the following syntax:

```
void sor(GEN x,char format,long dec,long field);
```

Here format is either `'e'`, `'f'` or `'g'` corresponding to the three output formats of GP, `dec` is the number of printed significant digits for real numbers, and should be put equal to -1 if all of them are wanted, and `field` corresponds to the field width of GP used for printing integers.

A default use of this function is to use the macro `outbeaut(GEN x)` which is equivalent to `sor(x,'g',-1,0)` followed by a newline and a buffer flush.

The second format corresponds to the "raw" format of GP (see section 2.2.5) and is obtained by using the function **brute** with the following syntax:

```
void brute(GEN x,char format,long dec);
```

A default use of this function is to use the macro `output(GEN x)` which is equivalent to `brute(x,'g',-1)` followed by a newline and a buffer flush.

Note that during debugging, the function `brute` cannot be nicely used since the buffer is not flushed, and the macro `output` cannot be executed, being a macro. For this reason only, Pari provides you with the fuunction `void imprimer(GEN x)` which has the same effect as `output`, but can now be called under a debugger. For example, under GDB, if x is a GEN, to see the contents of x simply type `p imprimer(x)`.

The third format corresponds to the `texprint` function of GP, and gives a TEX output of the result. It is obtained by using the function **texe** with the following syntax:

```
void texe(GEN x,char format,long dec);
```

Finally, you can use the hexadecimal tree output corresponding to the GP command "x using the function **voir** with the following syntax:

```
void voir(GEN x,-1);
```

Again the last parameter must be given and put equal to -1. In principle this last type of output is used only for debugging purposes.

4.2 Creation, destruction, and implementation of the PARI objects.

By far the most important functions which are specific to the library mode are the functions which allow the programmer to create and delete PARI objects, and the assignment statements.

4.2.1 Creation of PARI objects. The basic function which creates a PARI object is the function **cgetg** whose syntax is:

```
cgetg(long length,long type);
```

Here length is a long specifying the number of longwords to be allocated to the object, and type is a long which is the type number of the object (see chapter 1 or below for the list of these type numbers). Note that the length is the first argument, and type the second. The precise effect of this function is as follows: it first creates on the stack a chunk of memory of size "length" longwords, and puts the address of the chunk as the returned value. If not enough memory is available, a message to the effect that the PARI stack overflows will be printed. Otherwise, it sets the type of the chunk to "type", sets the length, and sets the reference count to 1. In effect, it fills correctly and completely the first codeword (z[0] or *z) of the PARI object. Many PARI objects having also a second codeword (types 1,2,7,10,11), *it is the programmer's responsibility to fill this second codeword*, either explicitly (see below), or implicitly using an assignment statement.

Note that the argument "length" is forced for a number of types: 3 for types 3,4,5,6,9,13,14 and 16, 4 for type 8 (Quad), and 5 for type 7 (p-adic). However for efficiency's sake no checking is made in the function `cgetg` so disastrous results can occur if you give an incorrect length.

Notes: 1) the creation of leaves, i.e. integers or reals, being so common, **cgeti** and **cgetr** should be used instead of `cgetg( ,1)` and `cgetg( ,2)`.

2) the macros `lgetg`, `lgeti` and `lgetr` are predefined as `(long)cgetg`, `(long)cgeti` and `(long)cgetr` respectively.

Examples:

`z = cgeti(6);` (or `cgetg(6, 1);`) creates an integer type which can hold numbers of absolute value less than 2^{128} (2 codewords + 4 mantissa longwords).

`z = cgetg(3, 6); z[1] = lgetr(5); z[2] = lgetr(5);` creates a complex type whose real and imaginary part can hold real numbers of precision 96 bits.

`z = cgetg(4, 19); for(i=1; i<4; i++) z[i] = lgetg(5, 18);` creates a matrix type for 4×3 matrices. One must also create space for the matrix elements themselves using a double loop.

These last two examples illustrates 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 get the whole tree. For matrices, a common mistake is to think that `z = cgetg(4, 19)` (for example) will create the root of the matrix: one needs also to create the column vectors of the matrix. This is because a matrix is really nothing else than a line 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.

4.2.2 Implementation of the PARI types.

Although it is a little tedious, we must go through each type and explain their implementation. Let z be a PARI object. Common to all the types is the first codeword (z[0]), which we don't have to worry about since this is taken care of by `cgetg`. However we need its precise description: the first byte is the type number, the second byte is the reference count (used only for garbage collecting purposes), and the last two bytes (the low order word if you prefer) is the length of the root in longwords. For instance in the example `z = cgeti(6)` above, z[0] will be set to `0x01010006`.

These bytes can be handled through the following functions:

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

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

`long pere(GEN z);` returns the reference count of z.

`void setpere(GEN z, long n);` sets equal to `n` the reference count of z.

`void incpere(GEN z);` increments the reference count of z (with saturation at 255).

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

`long setlg(GEN z, long l);` sets equal to `l` the length of z (you should not have to use this function if you use `cgetg`; however, see an example of the use of this function in the matexp function given in section 4.3).

Let us now look precisely at the types:

Type 1 (integers): this type has a second codeword z[1] which contains the following information. The first byte is the sign of z, i.e. 1 if $z > 0$, 0 if $z = 0$, -1 if $z < 0$. The second byte is unused. The low order word is the **effective length** of z, i.e. the total number of significant longwords. This means the following: apart from the integer 0 (whose second codeword is equal to 2), every integer is "normalized", meaning that the first mantissa longword (i.e. z[2]) is non zero. However, the integer may have been created with a longer length. Hence the "length" which is in z[0] can be larger than the "effective length" which is in z[1]. In fact, it would be a disaster to try to access z[i] for i larger than or equal to the effective length.

This information can be handled using the following functions:

`long signe(GEN z);` returns the sign of z.

`void setsigne(GEN z, long s);` sets equal to `s` the sign of z.

`long lgef(GEN z);` returns the effective length of z.

`void setlgef(GEN z, long l);` sets equal to `l` the effective length of z.

The integer 0 can be characterized either by its sign equal to 0, or by its effective length equal to 2. Apart from $z = 0$, z[2] exists and is non zero, and the absolute value of z is (z[2],z[3],...,z[lgef(z)-1]) in base 2^{32}, where as usual in this notation z[2] is the high order longword.

Type 2 (real numbers): this type has a second codeword z[1] whose first byte is also the sign (obtained or set using the same functions as for the integers), but whose 3 low order bytes contains a biased binary exponent (i.e. the exponent plus 2^{23}). This exponent can be handled using the following functions:

`long expo(GEN z);` returns the true (unbiased) exponent of z. This is defined even when z is equal to zero, see section 1.2.6.

Note also the function `long gexpo(GEN z)` which tries to return an exponent for z even if z is not a real number.

`void setexpo(GEN z, long e);` sets equal to `e` the exponent of z, of course after adding the bias.

The real zero is characterized by having its sign equal to 0. However, usually the first mantissa word z[2] is defined and equal to 0. This fact must *never* be used to characterize 0. If z is not equal to 0, the first mantissa word z[2] is normalized, i.e. its first bit is 1. The mantissa is (z[2],z[3],...,z[lg(z)-1]) in base 2^{32} where z[2] is the most significant longword, and the mantissa is understood to be between 1 and 2. Thus the real number -3.5 is represented (if the length is 4) as `0x02010004, 0xff800001, 0xe0000000, 0`.

Type 3 (integermods): z[1] points to the modulus, and z[2] on the number representing the class z. Both must be of type integer. In principle z[1] > 0 and 0 <= z[2] < z[1], but this rule does not have to be strictly obeyed by the user. Any integermod obtained after a PARI operation will in principle satisfy these conditions.

Types 4 and 5 (rational numbers): z[1] points to the numerator, and z[2] on the denominator. Both must be of type integer. In principle z[2] > 0, but this rule does not have to be strictly obeyed either.

Type 6 (complex numbers): z[1] points on the real part, and z[2] on the imaginary part. A priori z[1] and z[2] can be of any type, but only certain types are useful and make sense.

Type 7 (*p*-adic numbers): this type has a second codeword z[1] which contains the following information: the first *word* contains 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. the second word contains the biased exponent of z (the bias being here equal to 2^{15}). This information can be handled using the following functions:

`long precp(GEN z);` returns the p-adic precision of z, i.e. one less than the number of significant p-adic digits.

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

`long valp(GEN z);` returns the p-adic valuation of z (i.e. the unbiased exponent). This is defined even if z is equal to 0, see section 1.2.6.

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

In addition to this codeword, z[2] points to the prime p, z[3] points to $p^{\mathrm{precp(z)}}$, and z[4] points to an integer representing the p-adic unit associated to z modulo z[3] (or points to zero if z is zero). To summarize, we have the equality:

$$z = p^{\mathrm{valp(z)}} * (z[4] + O(z[3])) = p^{\mathrm{valp(z)}} * (z[4] + O(p^{\mathrm{precp(z)}}))$$

Type 8 (quadratic numbers): z[1] points on the polynomial defining the quadratic field, z[2] on the "real part" and z[3] on the "imaginary part", where this is to be taken as the coefficients of z on the "canonical" basis (1,w) (see section 1.2.3). Complex numbers are particular cases of quadratics but deserve a separate type.

Type 9 (polymods): exactly as for integermods, z[1] points on the modulus, and z[2] on a polynomial representing the class of z. Both must be of type polynomial. However, one must obey the rules explained in chapter 2 concerning the hierarchical structure of the variables of a polymod.

Type 10 (polynomials): this type has a second codeword which is analogous to the one for integers. The first byte is the "sign", i.e. 0 if the polynomial is equal to 0, and 1 if not (see however the important remark below). The second byte is the variable number (e.g. 0 for X, 1 for Y, etc...). This number can be handled with the following functions:

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

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 32767.

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

The least significant word of the second codeword is the effective length of the polynomial. Note that the degree of the polynomial is equal to the effective length minus three. The functions used to handle this codeword are the same as for integers. The components z[2], z[3],...z[lgef(z)-1] point to the coefficients of the polynomial *in ascending order*, with z[2] being the constant term and so on.

Important remark. A zero polynomial can be characterized by the fact that its sign is 0. However, its effective length may be equal to 2, or greater than 2. If it is greater than 2, this means that all the coefficients of the polynomial are equal to zero (as they should for a zero polynomial), but not all of these zeros are exact zeros, and more precisely the leading term z[lgef(z)-1] is not an exact zero.

Type 11 (power series): This type also has a second codeword. Its first byte is the "sign", i.e. 0 if the power series is 0, and 1 if not. The second byte is the variable number, as for polynomials. The last two bytes code for a biased exponent with a bias of 2^{15}. This information can be handled with the following functions:

`signe, setsigne, varn, setvarn` as for polynomials, and `valp, setvalp` for the exponent. Beware not to use `expo` and `setexpo` for power series.

If the power series is non zero, z[2], z[3],...z[lg(z)-1] point to the coefficients of z in ascending order, z[2] being the first non-zero coefficient. Note that the exponent of a power series can be negative, i.e. we can deal with Laurent series with a finite number of negative terms.

Types 13 and 14 (rational functions): z[1] points on the numerator, and z[2] on the denominator. Both must be of type polynomial.

Type 16 (binary quadratic forms): z[1], z[2], z[3] point on the three coefficients of the form. All three should be of type integer.

Types 17 and 18 (vectors): z[1], z[2],...z[lg(z)-1] point on the components of the vector.

Type 19 (matrices): z[1], z[2],...z[lg(z)-1] point on the column vectors of z, i.e. they must be of type 18 and of the same length.

4.2.3 Assignment and copying of PARI objects.

The general assignment function is the function **gaffect** with the following syntax:

`void gaffect(GEN x, GEN y);`

Its effect is to assign the PARI object x into the (preexisting) object y. Many conditions must be met for the assignment to be possible. For instance it is allowed to assign an integer into a real number, but the converse is forbidden. For that, you must use the truncation or rounding function of your choice (see section 3.2). It can also happen that y is not large enough or does not have the proper tree structure to receive the object x. As an extreme example, assume y is the zero integer with length equal to 2. Then all assignments of a non zero integer into y will result in an error message since y is not large enough to fit a non zero integer. In general common sense will tell you what is possible, keeping in mind the PARI philosophy which says that if it makes sense it is legal (the assignment of an imprecise object into a precise one does *not* make sense. However, a change in precision of imprecise objects is allowed).

It is also necessary to assign ordinary 32-bit long numbers of C into a PARI object. This is done using the function **gaffsg** with the following syntax:

```
void gaffsg(long s, GEN y);
```

It is also very useful to copy a PARI object, not just by moving around a pointer, but by physically creating a copy of the whole tree structure. The function which does this is called **gcopy**, with the predefined macro `lcopy` as a synonym for `(long)gcopy`. Its syntax is:

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

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

However, it may also be useful to create a *permanent* copy of a PARI object. This will not be created on the stack but on the heap. The function which does this is called **gclone**, with the predefined macro `lclone` as a synonym for `(long)gclone`. Its syntax is:

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

A final class of functions should be mentioned in this subsection: functions which convert from C objects to PARI objects (with creation of the objects on the stack). They are as follows:

```
GEN stoi(long s);
```

```
GEN dbltor(double s);
```

Their meaning is clear from their names. The converse functions

```
long gtolong(GEN z);
```

```
double gtodouble(GEN z);
```

also exist, but are seldom used.

4.2.4 Destruction of PARI objects and garbage collection.

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 by a cgetg or a function), this is very simple: just use the function **cgiv** with the syntax:

```
void cgiv(z);
```

where z is the object (which can be a tree), which you want to give back. Unfortunately life is not so simple, and in general you are doing a computation and want to give back accumulated garbage. A simple example is the following.

Let x and y be two preexisting PARI objects and suppose that we want to compute x*x+y*y. This can trivially be done using the following program (we skip the necessary declarations):

```
p1=gmul(x,x);p2=gmul(y,y);z=gadd(p1,p2);
```

The GEN z will indeed point on the PARI object equal to x*x+y*y. However, consider the stack. It contains as unnecessary garbage p1 and p2. More precisely it contains (in that order) z, p1, p2. We need a way to get rid of this garbage (in this case it causes no harm except that it occupies memory space, but in other cases it could disconnect PARI objects and this is forbidden). It would not have been possible to get rid of p1, p2 before since they are used in the final operation. It is not possible to use the function `cgiv` since p1 and p2 are not at the bottom of the stack. Hence PARI provides us with a much more powerful tool whose correct handling is not always easy: the function **gerepile** (with the macro `lpile` as a synonym to `(long)gerepile`). This function has the following syntax:

```
GEN gerepile(long ltop, long lbot, GEN z);
```

The understanding of the behavior of this function is certainly the most difficult part of this chapter, hence we will try to go slowly in the explanation.

First, the PARI stack has a current stack pointer which is a global variable (hence must not be declared in a program) called **avma** (which stands for **av**ailable **m**emory **a**ddress, which as its name indicates points always just after the first free address on the stack (remember that the stack grows from high to low addresses). For certain reasons this variable is of type long and not of type GEN as would be natural.

Now let us see the effects of the function `gerepile`. For this function to work one must have lbot < ltop. As a first approximation, we describe the effects of this function as follows. Give back the memory locations between lbot and ltop, and move the object z upwards so that no space is lost. Specifically, we rewrite our previous example as follows:

```
ltop=avma; /* keep in mind the current address of the top of the stack */
p1=gmul(x,x); p2=gmul(y,y);
lbot=avma; /* keep the address of the bottom of the garbage pile */
z=gadd(p1,p2);
z=gerepile(ltop,lbot,z); /* do the garbage collecting */
```

The last two instructions could also have been written more simply:

```
z=gerepile(ltop,lbot,gadd(p1,p2));
```

As you can see, in simple conditions the use of `gerepile` is not really difficult. However it is absolutely necessary to understand what has happened. When entering an instruction such as `gerepile(ltop,lbot,z)`, we must have avma <= lbot < ltop. As we will see, the variable z is in fact used very little. The function then considers all the PARI objects between avma and lbot (i.e. the ones that we want to keep) and looks at every component of such an object which is not a codeword. This component is a pointer on an object whose address is either between avma and lbot, in which case it will be suitably updated, larger than or equal to ltop, in which case it will not change, or between lbot and ltop in which case `gerepile` will scream an error message at you. If all goes well, the pointers are suitably updated, the chunk of memory from avma to lbot−1 is physically copied to addresses avma+ltop−lbot to ltop−1, avma is updated (to the value avma+ltop−lbot), and finally the displacement ltop−lbot is added to z and given as a result of the function. In addition, if z is not 0 (in the sense of being address 0), the PARI object pointed to by ltop is checked to see whether certain of its pointers need also to be updated. This is a horrible hack which should not be used.

Important remark: as we will see presently it is often necessary to do several `gerepile` in a computation. However, the least the better. The only condition for `gerepile` to work is that the garbage be connected. If the computation can be arranged so that there is a minimal number of connected pieces of garbage then it should be done.

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

```
p1=gmul(x,x);p2=gadd(p1,y);p3=gmul(y,y);p4=gadd(p3,x);
z=cgetg(3,17);z[1]=(long)p2;z[2]=(long)p4;
```

This leaves a dirty stack with (in that order) z,p4,p3,p2,p1 and p1 and p3 being garbage. But if we compute p3 *before* p2 then the garbage becomes connected and we get the following program with garbage collecting:

```
ltop=avma;p1=gmul(x,x);p3=gmul(y,y);lbot=avma;
z=cgetg(3,17);z[1]=ladd(p1,y);z[2]=ladd(p3,x);
z=gerepile(ltop,lbot,z);
```

We take our next example directly from the implementation of `gmul` in the file `gen2.c`, case 6 times case 6. In other words we want to write a program to compute the product of two complex numbers x and y, using a method which takes only 3 multiplications instead of 4. Let z=x*y, and set x=xr+i*xi and similarly for y and z. The well known trick is to compute p1=xr*yr, p2=xi*yi, p3=(xr+xi)*(yr+yi), and then we have zr=p1−p2, zi=p3−p1−p2. The program is essentially as follows:

```
ltop=avma;p1=gmul(x[1],y[1]);p2=gmul(x[2],y[2]);
p3=gmul(gadd(x[1],x[2]),gadd(y[1],y[2]));p4=gadd(p1,p2);lbot=avma;
z=cgetg(3,6);z[1]=lsub(p1,p2);z[2]=lsub(p3,p4);
z=gerepile(ltop,lbot,z);
```

Let us now look at a less trivial example where more than one `gerepile` is needed in practice (in theory one can always use only one, see below, but generally at the expense of efficiency). Suppose for instance that we want to write a function which multiplies a line vector by a matrix (such a function is of course already part of `gmul`, but let's ignore this for a moment). Then the most natural way is to do a `cgetg` of the result immediately, and then for each coefficient of the result vector to do a `gerepile` to get rid of the garbage which has accumulated for that particular coefficient. We leave the details to the reader, who can look at the answer in the file gen2.c, in the function gmul, case 17 times case 19. As was said above, it would theoretically be possible to have a single connected piece of garbage, but it would be a much less natural and unnecessarily complicated program.

Finally, let us take an example which is probably the least trivial way of using `gerepile`, but is unfortunately sometimes necessary. Although it is not an infrequent occurrence, we will not give a specific example but a general one. Suppose that we want to do a computation (usually inside a larger function) giving more than one PARI object as a result, say two for instance. Then even if we set up the work properly, before cleaning up we will have a stack which has the desired results z1, z2 (say), and then connected garbage from lbot to ltop. If we write

```
z1=gerepile(ltop,lbot,z1);
```

then the stack will be cleaned, the pointers OK, but we will have lost the address of z2. Hence the best way in this case is to write the following, assuming that `dec` has been declared as a long and not as a GEN:

```
dec=lpile(ltop,lbot,0)/4;z1+=dec;z2+=dec;
```

This works because `gerepile` thinks that 0 is a valid address, hence puts in `dec` the displacement in bytes. Since z1 and z2 are pointers, we need to divide `dec` by 4 to get a longword displacement.

If you followed us this far, congratulations, and rejoice, the rest is much easier.

4.2.5 Some tricks and hints. Even for the authors, the use of `gerepile` was not evident at first. Hence we give some indications on how to avoid most problems connected with `gerepile`, at the expense of speed.

First, although it looks complicated, `gerepile` has turned out to be a very flexible and fast garbage collector, which can compare very favorably with much more sophisticated methods used in other systems. A few tests that we have made indicate that the price paid for using `gerepile` is never more than 10 percent, and usually around 5 or 6 percent of the total time, which is quite acceptable.

Second, in many cases, in particular when the tree structure and the size of the PARI objects which will appear in a computation are under control, one can avoid `gerepile` altogether by creating sufficiently large objects at the beginning (using `cgetg`), and then using assignment statements and operations ending with z (such as `gaddz`). Coming back to our first example, note that if we know that x and y are of type real and of length less than or equal to 5, we can program without using `gerepile` at all:

```
z=cgetr(5);ltop=avma;p1=gmul(x,x);p2=gmul(y,y);gaddz(p1,p2,z);avma=ltop;
```

Here the cleaning up is done using the simple assignment avma=ltop which takes essentially no time at all.

Third, the philosophy of `gerepile` is the following: keep the value of the stack pointer avma at the beginning, and just **before** the last operation. Afterwards, it would be too late since the garbage address would be lost.

Finally, if everything seems hopeless, at the expense of speed you can do the following: after keeping in ltop the value of avma, perform your computation as you wish, in any order, leaving a messy stack. Let z be your result (let's assume you have only one. If you have more than one, you can always create a vector which is a single PARI object whose components are your results; after all, we are already cheating, we can cheat some more). Then write the following:

```
lbot=avma;z=gerepile(ltop,lbot,gcopy(z));
```

The trick is to force a copy of z at the bottom of the stack, hence all the rest including the initial z becomes connected garbage. Note that this may not work in some cases where you have created new universal objects between lbot and ltop, for instance moduli for mod p or p-adic computations. In that case you must also copy these objects on top of the stack and modify the components of your results which point to these objects. This in fact involves rewriting a simplified version of a part of `gerepile`, hence is not recommended. If you are in this situation and still want to use this trick, we strongly advise you to create all your new universal objects before starting your computation, hence before the instruction `ltop=avma;`. Then you should have no problems.

Another solution is to use clones, i.e. the function `gclone` (see 4.2.3), which creates a permanent object on the heap, and not on the stack.

4.3 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 work through a detailed nontrivial example of a main program. We will write a program which computes the exponential of a square matrix x. The complete listing is given in Appendix C, but each part of the program will be written and commented here. We will use an algorithm which is not optimal but is not far from the one used for the PARI function `gexp` (in fact in the function `mpexp1`). This consists in calculating the sum of the series:

$$e^{x/(2^n)} = \sum_{k=0}^{\infty} \frac{(x/(2^n))^k}{k!}$$

for a suitable positive integer n, and then computing e^x by repeated squarings. First, we will need to compute the L^2-norm of the matrix x, i.e. the quantity:

$$z = \|x\|_2 = \sqrt{\sum_{i,j} x_{i,j}^2}.$$

We will then choose the integer n such that the L^2-norm of $x/(2^n)$ is less than or equal to 1, i.e.

$$n = \left\lceil \frac{\ln(z)}{\ln(2)} \right\rceil$$

if $z >= 1$. Then the convergence of the series will be at least as good as the one for e^1, and will be easy to estimate. In fact a larger value of n would be preferable, but this is slightly machine dependent and more complicated, and will be left to the reader.

We will now start writing our program. So as to be able to use it in other contexts, we will structure it in the following way: a main program which will do the input and output, and a function which we shall call **matexp** which does the real work. The main program is easy to write. It can be something like this:

```
#include <genpari.h>
GEN matexp();

main()
{
  GEN x,y;
  long prec,d;
  char s[512];

  init(1000000,2); /* take a million bytes of memory for the stack */
  printf("precision of the computation in decimal digits?  ");
  scanf("%d",&d);if(d<0) prec=3;else prec=(long)(d*K1+3);
  printf("input your matrix in GP format:\n");
  s[0]=0;while(!s[0]) gets(s);x=lisexpr(s);
  y=matexp(x,prec);
  sor(y,'g',d,0);
}
```

The variable `prec` represents the length in longwords of the real numbers used. K1 is the constant (defined in `gencom.h`) equal to $\ln(10)/(32 * \ln(2))$, which allows us to convert from a number of decimal digits to a number of longwords. The function **lisexpr** was mentioned earlier and avoids any trouble in the input. In fact, as was also mentioned, **lisexpr** can take any legal GP expression hence can do not only input but also computations. Note that we have used the construction `s[0]=0;while(!s[0]) gets(s);` to get our string, instead of **scanf** which would make trouble in this case.

Finally, **sor** is the general output routine. We have chosen to give d significant digits since this is what was asked. Note that there is a trick hidden here: if d was given to be negative, then the computation will be done in precision 3 (i.e. about 9.7 decimal digits) and in the function **sor**, giving a negative third argument outputs all the significant digits, hence nothing is wrong.

Now let us attack the main course, the function matexp:

```
GEN matexp(x,prec)
    GEN x;
    long prec;

{
  GEN y,r,s,p1,p2;
  long tx=typ(x),lx=lg(x),i,k,n,lbot,ltop;

/* check that x is a square matrix */

  if(tx!=19) {printf("This expression is not a matrix\n");return(gzero);}
  if(lx!=lg((GEN)x[1])) {printf("Not a square matrix\n");return(gzero);}

/* compute the L2 norm of x */

  ltop=avma;s=gzero;
  for(i=1;i<lx;i++) s=gadd(s,gnorml2((GEN)x[i]));
  if(typ(s)==2) setlg(s,3);
  s=gsqrt(s,3); /* we do not need much precision on s */

/* if s<1, we are happy */

  if(expo(s)<0) {n=0;p1=x;}
  else {n=expo(s)+1;p1=gmul2n(p1,-n);setexpo(s,-1);}
```

Before continuing, a few remarks are in order. First, after printing an error message we need to return a GEN value otherwise a fussy compiler will complain. Hence we return gzero, which in any case is an impossible result for an exponential. We also have at our disposal the GEN gnil, which is also equal to zero, but has the peculiarity of not being printed.

Second, to compute the square of the L^2-norm of x we just add the squares of the L^2-norms of the column vectors which we obtain using the library function `gnorml2`. Had this function not existed, it would of course have been just as easy, but we would have had to make a double loop. Then, we take the square root of s, in precision 3. For that we use the function `setlg` which tells s to be in such a precision, only if s is of type real. Note that the matrix x is allowed to have complex entries, but the function `gnorml2` guarantees that s is a nonnegative number of type real. If we had not known this fact, we would simply have added the instruction `s = greal(s);` just after the `for` loop.

Third, before starting this computation which will produce garbage on the stack, we have carefully kept the value of the stack pointer avma in ltop. Note that we are going to assume throughout that the garbage does not overflow the memory that we allocated on the stack. If it did, we would have two solutions. Either allocate more memory in the main program (for instance change 1000000 into 2000000), or do some `gerepile` along the way.

Fourth, note that we initialized the sum `s` to gzero, which is an exact zero. This is logical, but has some disadvantages: if all the entries of the matrix are integers (or rational numbers), the computation will be rather long, about twice as long as with real numbers of the same length. It would be better to initialize `s` to a real zero, using for instance the instructions:

`s=cgetr(prec+1);gaffsg(0,s);`.

However, this would not make much sense. In fact you should avoid making an assignment of an exact zero (essentially the integer zero) into a real number: which real zero is it going to give as a result? In fact a choice has been made, and it will give you the zero with exponent equal to -32 times the number of longwords in the mantissa, but this is not really satisfactory. Perhaps PARI should give an error message in that case, or at least a warning.

Fifth, we will want to look at the approximate size of real numbers, and the fastest way to do this is to look at their binary exponents. Hence we need to have `s` as a real number, and not as an integer or a rational number. This is done automatically when we take the square root.

Finally, note the use of the function **gmul2n**. This function has the following syntax:

`GEN gmul2n(GEN x,long n);`

The effect is simply to multiply `x` by 2^{n}, where `n` can be positive or negative. This is much faster than gmul or gmulgs.

There is another function **gshift** with exactly the same syntax. When `n` is nonnegative, the effects of these two functions is the same. However when `n` is negative, gshift acts like a right shift of `-n`, hence does not give the same effect on integers. The function gshift is the PARI analogue of the C operators `<<` and `>>`.

We now come to the heart of the function. We have now a GEN p1 which points to a certain matrix of which we want to take the exponential. We will want to transform this matrix into a matrix with real (or complex of real) entries before starting the computation. To do this, we simply multiply by the real number 1 in precision prec+1 (to be safe). Then, to compute the series we will keep three variables: a variable p2 which at stage k will contain $\mathtt{p1}^k/k!$, a variable y which will contain $\sum_{i=0}^{k}\mathtt{p1}^i/i!$, and the variable r which will contain $\mathtt{s}^k/k!$. Note that we do not use Horner's rule. This is simply because we are lazy and do not want to compute in advance the number of terms that we need. We leave this modification (and many other improvements!) to the reader. The program continues as follows:

```
/* initializations before the loop */

  r=cgetr(prec+1);gaffsg(1,r);p1=gmul(r,p1);
  y=gscalmat(r,lx-1); /* this creates the scalar matrix r*I_{lx-1} */
  p2=p1;r=s;k=1;
  y=gadd(y,p2);

/* now the main loop */
```

```
  while(expo(r) >= -32*(prec-1))
    {k++;p2=gdivgs(gmul(p2,p1),k);r=gdivgs(gmul(s,r),k);y=gadd(y,p2);}

/* now square back n times if necessary */

  if(!n){lbot=avma;y=gerepile(ltop,lbot,gcopy(y));}
  else
  {
    for(i=0;i<n;i++){lbot=avma;y=gmul(y,y);}
    y=gerepile(ltop,lbot,y);
  }
  return(y);
}
```

A few remarks once again. First note the use of the function **gscalmat** with the following syntax:

```
GEN gscalmat(GEN x, long l);
```

The effect of this function is to create the `l` $\times$ `l` scalar matrix whose diagonal entries is the GEN x. Hence the length of the matrix including the codeword will in fact be `l+1`. There is a corresponding function **gscalsmat** which takes a long as a first argument.

If we refer to what has been said above, the main loop is clear.

When we do the final squarings, according to the fundamental theorem on the use of `gerepile` we keep the value of avma in lbot just *before* the squaring, so that if it is the last one, lbot will indeed be the bottom address of the garbage pile, and `gerepile` will work. Note that it takes a completely negligible time to do this in each loop compared to a matrix squaring. However, when n is initially equal to 0, no squarings have to be done, and we have our final result ready but we lost the address of the bottom of the garbage pile. Hence we use the trick of copying y again on top of the stack. This is inefficient, but does the work. If we wanted to avoid this, the best thing to do would be to put the instruction `lbot=avma` just before both occurrences of the instruction `y=gadd(p2,y);`.

Remarks on the program: as such, the program should work most of the time if x is a square matrix with real or complex entries. Indeed, since essentially the first thing that we do is to multiply by the real number 1, the program should work for integer, real, rational, complex or quadratic entries. This is in accordance with the behavior of transcendental functions.

Furthermore, since this program is intended to be only an illustrative example, it has been written a little sloppily. In particular many error checks have been omitted, and the efficiency is far from optimal. An evident improvement is to avoid the unnecessary gcopy by inserting a couple of extra `lbot=avma;` instructions. Another improvement is to multiply the matrix x by the real number 1 right at the beginning, speeding up the computation of the L^2-norm in many cases. These improvements are included in the version given in Appendix C. Still another improvement would come from a better choice of n. If the reader takes a look at the implementation of the function `mpexp1` in the file trans1.c he can make himself the necessary changes. Finally, there exist other algorithms of a different nature to compute the exponential of a matrix.

Remark: While writing this program, we have seen a few new functions (the complete list of available functions is given by alphabetical order in the index). However, if you care to look at the file `gencom.h`, you will notice that many more functions are defined. But in every case these missing functions are particular cases of general functions. For example, we have the function `gneg`, which takes the negative of a PARI object. However, there also exist functions like `negi` (for type integer), `negr` (for type real), and `mpneg` (for type integer or real).

These functions can of course be called by the user but we feel that the few microseconds lost in calling more general functions (in this case `gneg`) is compensated by the fact that one needs to remember a much smaller number of functions, and also because there is a hidden danger here: the type of the objects that you use, if they are themselves results of a previous computation, is not completely predetermined. For instance the multiplication of a type real by a type integer *usually* gives a result of type real, except when the integer is 0, in which case according to the PARI philosophy the result is the exact integer 0. Hence if afterwards you call a function which specifically needs a real type argument, you are in trouble.

If you really want to use these functions, their names are self-explanatory once you know that **i** stands for a PARI integer, **r** for a PARI real, **mp** for i or r, **s** for an ordinary 32-bit signed long, **z** (as a suffix) meaning as usual that the result is not created on the PARI stack but assigned to a preexisting GEN given as an extra argument.

For completeness, in Chapter 5 we have given a description of all these low-level functions.

Please note that in the present version 1.38.1 the names of the functions are not always consistent. This will be changed. Hence anyone programming in PARI must be aware that the names of almost all functions that he uses will change eventually. If the need arises (i.e. if there really are people out there who delve into the innards of PARI), updated versions with no name changes will be released.

4.4 Adding functions to PARI.

As already mentioned, modified versions of the PARI package should NOT be spread without our prior approval. If you do modify PARI, however, it is certainly for a good reason, hence we would like to know about it, so that everyone can benefit from it. There is then a good chance that the modifications that you have made will be incorporated into the next release.

(Recall the e-mail: `pari@alioth.greco-prog.fr`).

Roughly 3 types of modifications can be made. The first type is to modify the code, include files and the Makefile so that compilation is possible on a new system.

The second type is to modify existing code, either to correct bugs, to add new functionalities, or to improve efficiency.

Finally the third 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.

First choose the appropriate file where you are going to code your function. For example alglin1.c or alglin2.c if it is a function dealing with vectors or matrices (linear algebra), trans1.c, trans2.c, trans3.c if it is a transcendental function, arith1.c or arith2.c if it is an arithmetic function, bibli1.c or bibli2.c for miscellaneous functions, etc... Note that the files gp.c, mp.c and versionXXX.c as well as the assembly files have a special role and should not be modified.

Then code your function, using as a guide other functions in the Pari sources. One important thing to remember is to always clean the stack before exiting your function (usually using the function `gerepile`) otherwise the successive calls to the function will clutter the stack with unnecessary garbage and stack overflow will occur sooner.

If error messages are to be generated in your function, use the general error handling routine `err`. Initially, use the syntax

```
err(talker,"error message");
```

where `error message` is the message that you want printed. This function does not return, but ends with a `longjmp` statement.

Once the program is debugged, you can replace the above line by

```
err(errornumber);
```

where `errornumber` must be defined in the file `erreurs.h`, and the corresponding error message must be placed at the corresponding spot in the file `errmessages.c`. A look at these two files will tell you exactly what to do. Note however that to have the correct numbering of errors once again, hence the correct error messages, you must recompile the complete Pari distribution (for example after `touch *.[chs]`). This is not necessary if you use the `talker` syntax.

Finally, find a reasonable place in the file `gencom.h` to declare your function so that it is known to all other files and functions.

Your function is now ready to be used in library mode after compilation and creation of the library. It is however still inaccessible from GP.

To make your function accessible to GP, you must do the following. First, find a name for your function under GP, which does not have to be identical to the one used in library mode, but must use only lower case alphabetic characters and digits, the first character as usual being alphabetic. Then in the file `anal.c` place in exact alphabetical order of the name that you want to use under GP the following line (note that digits¡letters):

```
{"gpname",V,(void*)libname,secno,0},
```

where `libname` is the name of your function in library mode, `gpname` the name that you have chosen to call it under GP, `secno` is the section number of chapter 3 in which this function should belong (type `?` in GP to see the list), and V is a number between 0 and 99, chosen so as to correspond to the type of the function and its arguments. The first digit identifies the number of arguments, not counting the variable `prec`.

The most common values for V are the following, expressed as C-function prototypes. You can find the others by looking at the function `identifier` in the file `anal.c`. Note that the parameter `prec` (which denotes the real precision) is included in most of the given prototypes, but can be omitted if the function does not use it. Note also that we implicitly identify `int` with `long`.

```
V=0:  GEN name(long prec);

V=1:  GEN name(GEN x, long prec);

V=2:  GEN name(GEN x, GEN y, long prec);

V=3:  GEN name(GEN x, GEN y, GEN z, long prec);

V=10:  long name(GEN x);

V=11:  GEN name(long x,long prec);

V=15:  long name(long x);

V=20:  long name(GEN x, GEN y);
```

```
V=23:  GEN name(GEN x, long y, long prec);

V=24:  GEN name(long x, GEN y, long prec);

V=29:  long name(GEN x, long y);

V=30:  long name(GEN x, GEN y, GEN z);

V=32:  GEN name(GEN x, GEN y, long z);

V=33:  GEN name(GEN x, long y, long z);

V=35:  GEN name(long x, GEN y, GEN z);
```

If your function does not correspond to any of the above prototypes, or to the few others which you can find in the function `identifier`, you can make up your own. The best at this point is to read the code for the existing values of V, and to modify it according to your needs. One important point must be stressed if you add a value of V to the function `identifier`: you must also correspondingly add it to the function `skipidentifier`. Read the code for that function to see how to proceed.

Once this has been done, in the file `helpmessages.c` write in exact alphabetical order a short message describing the effect of your function:

```
"name(x,y,...)=short descriptive message",
```

The message must be a single line (which may have more than 80 characters). If the printed message would be more than one line, insert `\n` as appropriate (see the other messages for comparative purposes).

Finally create (or append to) a file called, say, `Mychanges`, which lists the modifications that you have made. Never touch the file `Changes` itself otherwise you would corrupt the Pari distribution. After compiling and debugging, you now have a new function available under GP (and we would very much like to hear about it!).

In this chapter, we give a description of almost all low-level functions of the PARI system. These essentially include functions for handling all the PARI types. Higher level functions, such as arithmetic or transcendental functions, are described fully in Chapter 3 of this manual.

5.1 Level 0 kernel (operations on unsigned longs)

The type "`ulong`" is defined in the file `gencom.h` as `unsigned long`. The global `ulong` variables `overflow` (which will contain only 0 or 1) and `hiremainder` are defined in the file `versionXXX.c`.

For the non-68k versions, we need level 0 operations simulating basic operations of the 68020 processor. For compatibility reasons, and also for very rare cases, they are also included in the 68k version, although they are not really needed. Note that in the given prototypes ulongs are sometimes implicitly typecasted as ints or longs.

`int addll(int x, int y)`: add the ulongs x and y, output the lower 32 bits and put the overflow bit in `overflow`.

`int addllx(int x, int y)`: add `overflow` to the sum of the ulongs x and y, output the lower 32 bits and put the overflow bit in `overflow`.

`int subll(int x, int y)`: subtract the ulongs x and y, output the lower 32 bits and put the overflow bit in `overflow`.

`int subllx(int x, int y)`: subtract `overflow` from the difference of the ulongs x and y, output the lower 32 bits and put the overflow bit in `overflow`.

`int shiftl(ulong x, ulong y)`: left shift the ulong x by y bits, output the lower 32 bits and put the higher 32 bits in `hiremainder`. We must have $y \leq 32$.

`int shiftlr(ulong x, ulong y)`: right shift the ulong $x << 32$ by y bits, output the higher 32 bits and put the lower 32 bits in `hiremainder`. We must have $y \leq 32$.

`int bfffo(ulong x)`: output the number of bits that the ulong x should be shifted left so that its leftmost bit is equal to 1 (32 if x is equal to 0).

`int mulll(ulong x, ulong y)`: multiply the ulong x by the ulong y, output the lower 32 bits and put the higher 32 bits in `hiremainder`.

`int addmul(ulong x, ulong y)`: add `hiremainder` to the product of the ulong x by the ulong y, output the lower 32 bits and put the higher 32 bits in `hiremainder`.

`int divll(ulong x, ulong y)`: output the Euclidean quotient of the ulong x by the ulong y and put the remainder in `hiremainder`.

`long mulmodll(ulong x, ulong y, ulong z)`: output $x*y$ modulo z.

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

In this section as elsewhere, "long" denotes a 32-bit signed C-integer, "integer" denotes a PARI multiprecise integer (type 1), "real" denotes a PARI multiprecise real (type 2). Refer to the users manual for general background.

5.2.1 Basic unit and subunit handling functions

`long typ(GEN x)`: gives the type number of x.

`long lg(GEN x)`: gives the length in 32-bit words of x.

`long lgef(GEN x)`: gives the effective length in 32-bit words of x. This makes sense only if x is an integer (type 1) or a polynomial (type 10).

`long mant(GEN x,long n)`: gives the n-th 32-bit mantissa word of the integer or real x (the most significant word is n=1).

`long signe(GEN x)`: gives the sign (-1, 0 or 1) of x. Should be used for integers, reals, polynomials and power series (for these last two only 0 or 1 are possible). For all other types, use `gsigne(GEN x)`.

`long expo(GEN x)`: gives the unbiased 24-bit binary exponent of the real number x. If x is not a real number, one can define such a quantity, but in that case use the function `gexpo(GEN x)`.

`long expi(GEN x)`: binary exponent of the real number equal to the integer x. This is the special case of the function `gexpo(GEN x)` corresponding to the case where x is of type integer.

`long pere(GEN x)`: gives the number of objects pointing at x, saturating at 255. Now unused.

`long valp(GEN x)`: gives the unbiased 16-bit p-adic valuation (for a p-adic) or X-adic valuation (for a power series) of x.

`long precp(GEN x)`: gives the precision of the p-adic x.

`long varn(GEN x)`: gives the variable number of x (between 0 and 255). Should be used only for polynomials and power series. For all other types, use `gvar(GEN x)`.

`void settyp(GEN x, long s)`: set equal to s the type number of x. This should be used with extreme care since usually the type is set otherwise.

`void setlg(GEN x, long s)`: set equal to s the length of x. Again this should be used with extreme care since usually the length is set otherwise.

`void setlgef(GEN x, long s)`: set equal to s the effective length of x, where x is an integer or a polynomial. The number s must be less than or equal to the length of x.

`void setmant(GEN x, long n, long s)`: set equal to s the n-th mantissa word of the integer or real number x.

`void setsigne(GEN x, long s)`: set equal to s the sign of x. If x is an integer or real, s must be equal to -1, 0 or 1, and if x is a polynomial or a power series, s must be equal to 0 or 1.

`void setexpo(GEN x, long s)`: set equal to s the binary exponent of the real number x. s must be a 24-bit signed number.

`void setpere(GEN x, long s)`: set equal to s the number of objects pointing to x, where $0 \leq s \leq 255$. Now unused.

`void incpere(GEN x)`: increase by 1 the number of objects pointing to x, saturating at 255. Now unused.

`void setvalp(GEN x, long s)`: set equal to s the p-adic valuation of x (if x is a p-adic) or the X-adic valuation of x (if x is a power series).

`void setprecp(GEN x, long s)`: set equal to s the p-adic precision of the p-adic number x.

`void setvarn(GEN x, long s)`: set equal to s ($0 \leq s \leq 255$) the variable number of the polynomial or power series x.

5.2.2 Memory allocation on the PARI stack

`GEN cgetg(long n, long t)`: allocate memory on the PARI stack for an object of length n and type t, and initialize its first codeword (containing `typ` (1 byte), `pere` (1 byte), `lg` (2 bytes).

`GEN cgeti(long n)`: allocate memory on the PARI stack for an integer of length n, and initialize its first codeword. Identical to `cgetg(n,1)`.

`GEN cgetr(long n)`: allocate memory on the PARI stack for a real of length n, and initialize its first codeword. Identical to `cgetg(n,2)`.

`void cgiv(GEN x)`: free object x if it is the last created on the PARI stack (otherwise disaster occurs).

`GEN gerepile(long p, long q, GEN x)`: general garbage collector for the PARI stack. See chapter 4 of the manual for a detailed explanation.

5.2.3 Assignments, conversions and integer parts

`void mpaff(GEN x, GEN z)`: assign x into z (x and z integers or reals).

`void affsz(long s, GEN z)`: assign the long s into the integer or real z.

`void affsi(long s, GEN z)`: assign the long s into the integer z.

`void affsr(long s, GEN z)`: assign the long s into the real z.

`void affii(GEN x, GEN z)`: assign the integer x into the integer z.

`void affir(GEN x, GEN z)`: assign the integer x into the real z.

`void affrs(GEN x, long s)`: assign the real x into the long s. This is a forbidden assignment in PARI, so an error message is issued.

`void affri(GEN x, GEN z)`: assign the real x into the integer z. This is a forbidden assignment in PARI, so an error message is issued.

`void affrr(GEN x, GEN z)`: assign the real x into the real z.

`GEN stoi(long s)`: create the PARI integer corresponding to the long s.

`long itos(GEN x)`: convert to long the PARI integer x (if possible, otherwise an error message is issued).

`GEN mptrunc(GEN x)`: truncation of the integer or real x (not the same as the integer part if x is noninteger and negative).

`void mptruncz(GEN x, GEN z)`: assign the truncation of the integer or real x into z.

`GEN mpent(GEN x)`: true integer part of the integer or real x (i.e. the `floor` function).

`void mpentz(GEN x, GEN z)`: assign the true integer part of the integer or real x into z.

5.2.4 Valuation and shift

`long vals(long s)`: 2-adic valuation of the long s. Returns -1 if s is equal to 0, with no error.

`long vali(GEN x)`: 2-adic valuation of the integer x. Returns -1 if s is equal to 0, with no error.

`GEN mpshift(GEN x, long n)`: shift by n the real or integer x. If n is positive, this is a left shift, i.e. multiplication by 2^n. If n is negative, it is a right shift, hence the truncation of the division of x by 2^{-n}.

`void mpshiftz(GEN x, long n, GEN z)`: assign into z the real or integer x shifted by n.

`GEN shifts(long s, long n)`: convert the long s into a PARI integer and shift by n.

`GEN shifti(GEN x, long n)`: shift by n the integer x.

`GEN shiftr(GEN x, long n)`: shift by n the real x.

5.2.5 Unary operations

Let "op" be some unary operation of type GEN. The names and prototypes of the low-level functions corresponding to op will be as follows.

`GEN mpop(GEN x)`: create the result of op applied to the integer or real x.

`GEN ops(long s)`: create the result of op applied to the long s.

`GEN opi(GEN x)`: create the result of op applied to the integer x.

`GEN opr(GEN x)`: create the result of op applied to the real x.

`void mpopz(GEN x, GEN z)`: assign into the integer or real z the result of applying op to the integer or real x.

Remark: it has not been considered useful to include in PARI the functions `void opsz(long s, GEN z)`, `void opiz(GEN x, GEN z)` and `void oprz(GEN x, GEN z)`.

This can be applied to the following operators.

op=neg: negation ($-$x). The result is the same type as x.

op=abs: absolute value ($|$x$|$). The result is the same type as x.

In addition, there exist the following special unary functions with assignment.

`void mpinvz(GEN x, GEN z)`: assign the inverse of the integer or real x into the real z. The inverse is computed as a quotient of real numbers, not as a Euclidean division.

`void mpinvsr(long s, GEN z)`: assign the inverse of the long s into the real z.

`void mpinvir(GEN x, GEN z)`: assign the inverse of the integer x into the real z.

`void mpinvrr(GEN x, GEN z)`: assign the inverse of the real x into the real z.

5.2.6 Comparison operators

`long mpcmp(GEN x, GEN y)`: compare the integer or real x with the integer or real y. The result is the sign of x$-$y.

`long cmpss(long s, long t)`: sign of s$-$t.

`long cmpsi(long s, GEN x)`: compare the long s with the integer x.

`long cmpsr(long s, GEN x)`: compare the long s with the real x.

`long cmpis(GEN x, long s)`: compare the integer x with the long s.

`long cmpii(GEN x, GEN y)`: compare the integer x with the integer y.

`long cmpir(GEN x, GEN y)`: compare the integer x with the real y.

`long cmprs(GEN x, long s)`: compare the real x with the long s.

`long cmpri(GEN x, GEN y)`: compare the real x with the integer y.

`long cmprr(GEN x, GEN y)`: compare the real x with the real y.

5.2.7 Binary operations

Let "op" be some operation of type GEN. The names and prototypes of the low-level functions corresponding to op will be as follows.

`GEN mpop(GEN x, GEN y)`: create the result of op applied to the integer or reals x and y.

`GEN opss(long s, long t)`: create the result of op applied to the longs s and t.

`GEN opsi(long s, GEN x)`: create the result of op applied to the long s and the integer x.

`GEN opsr(long s, GEN x)`: create the result of op applied to the long s and the real x.

`GEN opis(GEN x, long s)`: create the result of op applied to the integer x and the long s.

`GEN opii(GEN x, GEN y)`: create the result of op applied to the integers x and y.

`GEN opir(GEN x, GEN y)`: create the result of op applied to the integer x and the real y.

`GEN oprs(GEN x, long s)`: create the result of op applied to the real x and the long s.

`GEN opri(GEN x, GEN y)`: create the result of op applied to the real x and the integer y.

`GEN oprr(GEN x, GEN y)`: create the result of op applied to the reals x and y.

`GEN mpopz(GEN x, GEN y, GEN z)`: assign into the integer or real z the result of op applied to the integer or reals x and y.

`GEN opssz(long s, long t, GEN z)`: assign into the integer or real z the result of op applied to the longs s and t.

`GEN opsiz(long s, GEN x, GEN z)`: assign into the integer or real z the result of op applied to the long s and the integer x.

`GEN opsrz(long s, GEN x, GEN z)`: assign into the integer or real z the result of op applied to the long s and the real x.

`GEN opisz(GEN x, long s, GEN z)`: assign into the integer or real z the result of op applied to the integer x and the long s.

`GEN opiiz(GEN x, GEN y, GEN z)`: assign into the integer or real z the result of op applied to the integers x and y.

`GEN opirz(GEN x, GEN y, GEN z)`: assign into the integer or real z the result of op applied to the integer x and the real y.

`GEN oprsz(GEN x, long s, GEN z)`: assign into the integer or real z the result of op applied to the real x and the long s.

`GEN opriz(GEN x, GEN y, GEN z)`: assign into the integer or real z the result of op applied to the real x and the integer y.

`GEN oprrz(GEN x, GEN y, GEN z)`: assign into the integer or real z the result of op applied to the reals x and y.

This can be applied to the following operators.

op=add: addition (x+y). The result is real except if both x and y are integers.

op=sub: subtraction (x−y). The result is real except if both x and y are integers.

op=mul: multiplication (x∗y). The result is real except if both x and y are integers OR if x or y is the integer or long zero.

op=div: division (x/y). In the case where x and y are both integers or longs, the result is the Euclidean quotient, the remainder being of the sign of the dividend x. If one of x or y is real, the result is real except if x is the integer or long zero. Division by zero error if y is equal to zero.

op=res: remainder (x%y). This operation exists only when x and y are longs or integers. The result is the Euclidean remainder corresponding to div, i.e. its sign is that of the dividend x. The result is always an integer.

op=mod: remainder (x%y). This operation exists only when x and y are longs or integers. The result is the true Euclidean remainder, i.e. nonnegative and less than the absolute value of y.

5.2.8 Division with remainder

`GEN dvmdss(long s, long t, GEN* r`: create the Euclidean quotient and remainder of the longs s and t. If r is not 0 or −1. put the address of the remainder in r, and output the quotient. If r is equal to 0, only the quotient is output. If r is equal to −1, the remainder is output instead of the quotient. The remainder is always of the sign of the dividend s.

`GEN dvmdsi(long s, GEN x, GEN* r)`: create the Euclidean quotient and remainder of the long s by the integer x.

`GEN dvmdis(GEN x, long s, GEN* r)`: create the Euclidean quotient and remainder of the integer x by the long s.

`GEN dvmdii(GEN x, GEN y, GEN* r)`: create the Euclidean quotient and remainder of the integer x by the integer y.

`void mpdvmdz(GEN x, GEN y, GEN z, GEN* r)`: assign into the integer or real z the Euclidean quotient of the integers x by y, putting the address of the remainder in r (except if r is equal to 0 or −1 as above).

`void dvmdssz(long s, long t, GEN z, GEN* r)`: assign into the integer or real z the Euclidean quotient of the longs s by t, putting the address of the remainder in r (except if r is equal to 0 or −1 as above).

`void dvmdsiz(long s, GEN x, GEN z, GEN* r`: assign into the integer or real z the Euclidean quotient of the long s by the integer x, putting the address of the remainder in r (except if r is equal to 0 or −1 as above).

`void dvmdisz(GEN x, long s, GEN z, GEN* r)`: assign into the integer or real z the Euclidean quotient of the integer x by the long s, putting the address of the remainder in r (except if r is equal to 0 or −1 as above).

`void dvmdiiz(GEN x, GEN y, GEN z, GEN* r)`: assign into the integer or real z the Euclidean quotient of the integers x by y, putting the address of the remainder in r (except if r is equal to 0 or -1 as above).

5.2.9 Miscellaneous functions

`long mpdivis(GEN x, GEN y, GEN z)`: if the integer y divides the integer x, assign the quotient in the integer z and return 1 (true), otherwise return 0 (false).

`long divise(GEN x, GEN y)`: if the integer y divides the integer x, return 1 (true), otherwise return 0 (false).

`GEN convi(GEN x)`: converts the absolute value of the integer x in base 10^9. The result is an integer z such that z[1] is set artificially to -1, z[2], $\ldots$, z[m] contain the base 10^9 digits of $|x|$, and the output is the address of z[m+1].

`GEN confrac(GEN x)`: converts the fractional part of the real x in base 10^9. The result is an array z on the PARI stack such that z[0] contains the number of significant decimal digits, and z[1],...the base 10^9 expansion of the fractional part of x.

Warning: since z is not a standard PARI object, the stack must be cleaned by a simple avma=oldavma statement, and not by the `gerepile` garbage collector after execution of this function.

`void addsii(long s, GEN x, GEN z)`: assign into the integer z the sum of the long s and the integer x (essentially identical to `addsiz` except that z is specifically an integer).

`void mulsii(long s, GEN x, GEN z)`: assign into the integer z the product of the long s and the integer x (essentially identical to `mulsiz` except that z is specifically an integer).

`long divisii(GEN x, long s, GEN z)`: assign into the integer z the Euclidean quotient of the integer x by the long s, and return the remainder as a long.

5.3 Level 2 kernel (operations on general PARI objects)

In this section, functions followed by an assignment statement will be mentioned as in the following example:

`GEN gadd[z](GEN x, GEN y[, GEN z])`: the explanation is given for the function `gadd`, and the brackets are there to say that the `void` function `void gaddz(GEN x, GEN y, GEN z)` also exists, where the result is assigned into z.

All these [z] functions are obtained using macros (see the file `gencom.h`) hence if an extension of the list is desired, it is trivial to add the suitable macro in the file `gencom.h`.

The functions available to handle subunits are the following.

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

`GEN truecoeff(GEN x, long n)`: create a copy of the coefficient of degree n of x if x is a scalar, polynomial or power series, otherwise of the n-th component of x.

`long coeff(GEN x, long i, long j)`: x being a matrix (type 19), this gives the address of the coefficient of row i and column j of the matrix x. This is NOT a function, simply a macro, and can be used on the left or right hand side of an assignment statement.

5.3.1 Copying and conversion

`GEN cgetp(GEN x)`: create p-adic space sufficient to put x into, and set the prime p and the p-adic precision to that of x, but do not copy x.

`GEN gcopy(GEN x)`: create a new copy of the object x on the PARI stack.

`GEN forcecopy(GEN x)`: same as copy except that even permanent pointers are copied onto the stack.

`long taille(GEN x)`: total number of 32-bit words occupied by the tree representing x.

`GEN gclone(GEN x)`: create a new copy of the object x on the heap.

`GEN greffe(GEN x, long l)`: x being a polynomial (type 10), create a power series (type 11) of length l starting with x, but without actually copying the coefficients, just the pointers. For internal use.

`double rtodbl(GEN x)`: x being a real (type 2), convert x into a C double if possible.

`GEN dbltor(double x)`: convert the C double x into a PARI real.

`double gtodouble(GEN x)`: x being in the real numbers (but not necessarily of type real), convert x into a C double if possible.

`long gtolong(GEN x)`: x being in the integers (but not necessarily of type integer), convert x into a C long if possible.

`GEN gtopoly(GEN x, long v)`: convert or truncate the object x into a polynomial with main variable number v.

`GEN gtopolyrev(GEN x, long v)`: convert or truncate the object x into a polynomial with main variable number v, but vectors are converted in reverse order.

`GEN gtoser(GEN x, long v)`: convert the object x into a power series with main variable number v.

`GEN gtovec(GEN x)`: convert the object x into a (row) vector.

`GEN co8(GEN x, long l)`: x being a quadratic number (type 8), convert x into a real number or complex number depending on the sign of the discriminant of x, to precision l 32-bit words.

`GEN gcvtop(GEN x, GEN p, long l)`: convert x into a p-adic number of precision l.

`GEN gmodulcp(GEN x, GEN y)`: create the object mod(x,y) on the PARI stack, where x and y are either both integers, and the result is an integermod (type 3) or x is a scalar or a polynomial and y a polynomial, and the result is a polymod (type 9).

`GEN gmodulo(GEN x, GEN y)`: same as `gmodulcp` except that the modulus y is copied on the heap and not in the PARI stack.

`long gexpo(GEN x)`: return the binary exponent of x or the maximal binary exponent of the coefficients of x; return $-(2^{15}-1)$ if x has no components. Error if x is exactly zero.

`long gsize(GEN x)`: return 0 if x is exactly 0, otherwise return `gexpo(x)` multiplied by $\log(2)/\log(10)$. This gives a crude estimate for the maximal number of decimal digits of the components of x.

`long gsigne(GEN x)`: return the sign of x (-1, 0 or 1) when x is an integer, real or (irreducible or reducible) fraction. Error for all other types.

`long gvar(GEN x)`: return the main variable of x. If no component of x is a polynomial or power series, return $2^{15}-1$.

`int precision(GEN x)`: If x is of type real, return the precision of x (the length of x in 32-bit words if x is not zero, and a reasonable quantity obtained from the exponent of x if x is equal to zero). If x is of type complex, return the minimum of the precisions of the real and imaginary part. Otherwise, return 0 (coding in fact for ∞).

5.3.2 Comparison operators and valuations

`int gcmp0(GEN x)`: 1 (true) if x is equal to 0, 0 (false) otherwise.

`int isexactzero(GEN x)`: 1 (true) if x is exactly equal to 0, 0 (false) otherwise.

`int gcmp1(GEN x)`: 1 (true) if x is equal to 1, 0 (false) otherwise.

`int gcmp_1(GEN x)`: 1 (true) if x is equal to -1, 0 (false) otherwise.

`long gcmp(GEN x, GEN y)`: comparison of x with y (sign of $x-y$).

`long gcmpsg(long s, GEN x)`: comparison of the long s with x.

`long gcmpgs(GEN x, long s)`: comparison of x with the long s.

`long lexcmp(GEN x, GEN y)`: comparison of x with y for the lexicographic ordering.

`long gegal(GEN x, GEN y)`: 1 (true) if x is equal to y, 0 otherwise.

`long gegalsg(long s, GEN x)`: 1 (true) if the long s is equal to x, 0 otherwise.

`long gegalgs(GEN x, long s)`: 1 (true) if x is equal to the long s, 0 otherwise (it should not be very different from `gegalsg`!).

`long iscomplex(GEN x)`: 1 (true) if x is in the complex numbers but not in the reals, 0 if x is in the reals, error if x is not in the complex numbers.

`long ismonome(GEN x)`: 1 (true) if x is a nonzero monomial in its main variable, 0 otherwise.

`long ggval(GEN x, GEN p)`: greatest exponent of p dividing x, when this makes sense.

`long gval(GEN x, long v)`: highest power of the variable number v dividing the polynomial x.

`int pvaluation(GEN x, GEN p, GEN*r)`: x and p being integers, returns the highest power of p dividing x, creates the coprime to p part of x and returns its address in r.

5.3.3 Assignment statements

`void gaffsg(long s, GEN x)`: assign the long s into the object x.

`void gaffect(GEN x, GEN y)`: assign the object x into the object y.

5.3.4 Unary operators

`GEN gneg[z](GEN x[, GEN z])`: create $-x$.

`GEN gabs[z](GEN x[, GEN z])`: create $|x|$.

`GEN gsqr(GEN x)`: create the square of x,

`GEN ginv(GEN x)`: create the inverse of x.

`GEN gfloor(GEN x)`: create the floor of x, i.e. the (true) integral part.

`GEN gfrac(GEN x)`: create the fractional part of x, i.e. x minus the floor of x.

`GEN gceil(GEN x)`: create the ceiling of x.

`GEN ground(GEN x)`: round the components of x to the nearest integer.

`GEN grndtoi(GEN x, long*e)`: same as round, but in addition put in *e minus the number of significant binary bits left after rounding. If *e is positive, all significant bits are lost. This gives an error message in `ground` but not in `grndtoi`.

`GEN gtrunc(GEN x)`: truncate x. This is the (false) integer part if x is an integer, converts a series into a polynomial or rational function, takes the polynomial part of a rational function.

`GEN gcvtoi(GEN x, long*e)`: same as `grndtoi` except that rounding is replaced by truncation.

`GEN gred[z](GEN x[, GEN z])`: reduce x to lowest terms if x is a fraction or rational function (types 4, 5, 13 and 14), otherwise create a copy of x.

`void gredsp(GEN* x)`: reduce *x to lowest terms and put the result in *x again. The type of the result may be changed to integer or polynomial. A call to this function is the nomral way of finishing a computation whose result gives a fraction or rational function which is not necessarily in lowest terms.

`GEN content(GEN x)`: create the GCD of all the components of x.

`GEN primpart(GEN x)`: x divided by its content.

`void normalize(GEN* px)`: x=*px being an unnormalized power series (i.e. type 11 with all coefficients correctly set except that x[2] is not necessarily different from zero), normalize x correctly in place. For internal use.

`void normalizepol(GEN* px)`: x=*px being an unnormalized polynomial (i.e. type 10 with all coefficients correctly set except that x[2] is not necessarily different from zero), normalize x correctly in place. For internal use.

5.3.5 Binary operators

`GEN gmax[z](GEN x, GEN y[, GEN z])`: create the maximum of the objects x and y if they can be compared.

`GEN gmaxsg[z](long s, GEN x[, GEN z])`: create the maximum of the long s and the object x.

`GEN gmaxgs[z](GEN x, long s[, GEN z])`: create the maximum of the object x and the long s.

`GEN gmin[z](GEN x, GEN y[, GEN z])`: create the minimum of the objects x and y if they can be compared.

`GEN gminsg[z](long s, GEN x[, GEN z])`: create the minimum of the long s and the object x.

`GEN gmings[z](GEN x, long s[, GEN z])`: create the minimum of the object x and the long s.

`GEN gadd[z](GEN x, GEN y[, GEN z])`: create the sum of the objects x and y.

`GEN gaddsg[z](long s, GEN x[, GEN z])`: create the sum of the long s and the object x.

`GEN gaddgs[z](GEN x, long s[, GEN z])`: create the sum of the object x and the long s.

`GEN gaddpex(GEN x, GEN y)`: create the sum of the integer or fraction x and the p-adic y. For internal use.

`GEN gsub[z](GEN x, GEN y[, GEN z])`: create the difference of the objects x and y.

`GEN gsubgs[z](GEN x, long s[, GEN z])`: create the difference of the object x and the long s.

`GEN gsubsg[z](long s, GEN x[, GEN z])`: create the difference of the long s and the object x.

`GEN gmul[z](GEN x, GEN y[, GEN z])`: create the product of the objects x and y.

`GEN gmulsg[z](long s, GEN x[, GEN z])`: create the product of the long s with the object x.

`GEN gmulgs[z](GEN x, long s[, GEN z])`: create the product of the object x with the long s.

`GEN gshift[z](GEN x, long n[, GEN z])`: left shift of the components of x by n (if n is non-negative) or right shift by $-n$ (if n is negative). Applies only to integers, reals and vectors/matrices of such. For other types, it is simply multiplication by 2^n.

`GEN gmul2n[z](GEN x, long n[, GEN z])`: create x multiplied by 2^n. The difference with `gshift` occurs when n is negative and x is of type integer: `gshift` truncates, while `gmul2n` creates a fraction if necessary.

`GEN gdiv[z](GEN x, GEN y[, GEN z])`: create the quotient of the objects x and y.

`GEN gdivgs[z](GEN x, long s[, GEN z])`: create the quotient of the object x with the long s.

`GEN gdivsg[z](long s, GEN x[, GEN z])`: create the quotient of the long s with the object x.

`GEN gdivent[z](GEN x, GEN y[, GEN z])`: create the true Euclidean quotient of x by the integer or polynomial y.

`GEN gdiventsg[z](long s, GEN x[, GEN z])`: create the true Euclidean quotient of the long s by the integer x.

`GEN gdiventgs[z](GEN x, long s[, GEN z])`: create the true Euclidean quotient of the integer x by the long s.

`GEN gdiventres(GEN x, GEN y)`: create a 2-component vertical vector whose components are the true Euclidean quotient and remainder of x by y.

`GEN gdivmod(GEN x, GEN y, GEN*r)`: If r is not equal to 0 or -1, create the (false) Euclidean quotient of x by y, and put the address of the remainder in r. If r is equal to 0, do not create the remainder, and if r is equal to -1, create and output only the remainder.

`GEN poldivres(GEN x, GEN y, GEN*r)`: same as `gdivmod` but specifically for x and y polynomials.

`GEN gdeuc[z](GEN x, GEN y[, GEN z])`: create the Euclidean quotient of the polynomial x by the polynomial y.

`GEN gmod[z](GEN x, GEN y[, GEN z])`: create the true remainder of x modulo the integer or polynomial y.

`GEN gmodsg[z](long s, GEN x[, GEN z])`: create the true remainder of the long s modulo the integer x.

`GEN gmodgs[z](GEN x, long s[, GEN z])`: create the true remainder of the integer x modulo the long s.

`GEN gres(GEN x, GEN y)`: create the Euclidean remainder of the polynomial x divided by the polynomial y.

`GEN ginvmod(GEN x, GEN y)`: create the inverse of x modulo y when it exists.

`GEN gpui(GEN x, GEN y, long l)`: create x^y. The precision l is taken into account only if y is not an integer and x is an exact object. If y is an integer, binary powering is done. Otherwise, the result is $\exp(y * \log(x))$ computed to precision l.

`GEN ggcd(GEN x, GEN y)`: create the GCD of x and y.

`GEN glcm(GEN x, GEN y)`: create the LCM of x and y.

`GEN subres(GEN x, GEN y)`: resultant of the polynomials x and y computed using the subresultant algorithm.

`GEN gpuigs(GEN x, long n, long l)`: create x^n using binary powering. The precision l is used *only* when x is of type 15 (real quadratic forms), hence can almost always be ommited.

`GEN gsubst(GEN x, long v, GEN y)`: substitute in x the object y for the variable number v.

`int gdivise(GEN x, GEN y)`: return 1 (true) if y divides x, 0 otherwise.

`GEN gbezout(GEN x, GEN y, GEN* u, GEN* v)`: create the GCD of x and y, and put in u and v the adresses of objects such that $*ux+*vy=\gcd(x,y)$.

This chapter is intended to be a guided tour and a tutorial to the GP calculator. Many examples will be given, but each time a new function is used, the reader should look at the appropriate section in the manual for detailed explanations. Hence although this chapter can be read independently of the rest (for example to get rapidly acquainted with the possibilities of GP without having to read the whole reference manual), the reader will profit most from it by reading it in conjunction with the reference manual.

6.1 Greetings!

So you are sitting in front of your workstation (or terminal, or PC,...), and you type **gp** to get the program started (remember to always hit the `enter` key and not the `return` key on a Macintosh computer).

It says hello in its particular manner, and then waits for you after its `prompt`, initially `?`.

Type $2+2$. What happens? Maybe not what you expect. First of all, of course, you should tell GP that your input is finished, and this is done by hitting the `return` (or `newline`) key, or the `enter` key on the Mac. If you do exactly this, you will get the expected answer. However some of you may be used to other systems like Macsyma or Maple. In this case, you will have subconsciously ended the line with a semicolon `;` before hitting `return`, since this is how it is done on those systems. In that case, you will simply see GP answering you with a smug expression, i.e. a new prompt and no answer! This is because a semicolon at the end of a line in GP tells it to keep the result, but not to print it (you will certainly want to use this feature if the output is several pages long).

Try $27 * 37$. Wow! even multiplication works.

Now this session is getting lengthy, so the second thing one needs to learn is to quit. Each system has its quit signal (to name a few: `quit`, `quit();`, `exit`, `system`,...). In GP it is `\q` (backslash), the `q` being of course for quit. Try it.

Now you've done it! You're out of GP, so how do you want to continue studying this tutorial? Get back in please (see above).

Let's get to more serious stuff. Let's see, I seem to remember that the decimal expansion of 1/7 has some interesting properties. Let's see what GP has to say about this. Type 1/7. What? This computer is making fun of me, it just spits back to me my own input, that's not what I want!

Now stop complaining, and think a little. This system has been written mainly for pure mathematicians, and not for physicists (although they are welcome to use it :-). And mathematically, 1/7 is an element of the field $\mathbf{Q}$ of rational numbers, so how else but 1/7 can the computer give the answer to you? (well maybe 2/14, but why complicate matters?). Seriously, the basic point here is that PARI (hence GP) will almost always try to give you a result which is as precise as possible (we will see why "almost" later), hence since here you gave an operation whose result can be represented exactly, that's what it gives you.

OK, but I still want the decimal expansion of 1/7. No problem. Type one of the following: 1./7, 1/7., 1./7., 0. + 1/7,.... Immediately 28 decimals of this fraction appear, and the repeating pattern is 142857. The reason is that you have included in the operations numbers like 0., 1. or 7. which are *imprecise* real numbers, hence GP cannot give you an exact result.

Why 28 decimals by the way? well, it is the default initial precision, as indicated when you launch GP. This has been chosen so that the computations are very fast, and give already 12 decimals more accuracy than conventional double precision floating point operations. Only large mainframes or supercomputers have 28 digits of precision in their standard libraries, and that is the absolute limit.

Not here of course. You can extend the precision (almost) as much as you like as we will see in a moment.

I'm getting bored, why don't we get on with some more exciting stuff? Well, try `exp(1)`. Presto, comes out the value of e to 28 digits. Try `log(exp(1))`. Well, it's not exactly equal to 1, but pretty close! That's what you lose by working numerically.

Now try `exp(pi*sqrt(163))`. Hmmm, since from our last example we suspect that the last digit may be wrong, can this really be an integer? This is the time to change precision. Type `\precision=50`, then try `exp(pi*sqrt(163))` again. We were right to suspect that the last decimal was incorrect, since we get even more nines than before, but it is now convincingly clear that this is not an integer. Maybe it's a bug in PARI, and the result is really an integer? Type `sqr(log(%)/pi)` immediately after the preceding computation (`%` means the result of the last computed expression. More generally, the results are numbered `%1, %2, ...` **including** the results that you do not want to see printed by putting a semicolon at the end of the line, and you can evidently use all these quantities in any further computations. `sqr` is the square function (`sqr(x)=x*x`), not to be confused with `sqrt` which is the square root function). The result seems to be indistinguishable from 163, hence it does not seem to be a bug.

In fact it is a known fact that `exp(pi*sqrt(n))` not only is not an integer or a rational number, but is even a transcendental number when n is a positive rational number.

So GP is just a fancy calculator, able to give me more decimals than I will ever need? Not so, GP is incredibly more powerful than an ordinary calculator, even independently of its arbitrary precision possibilities.

Additional comments

You are supposed to skip this at first, and come back later.

1) It seems that the text implicitly says that as soon as an imprecise number is entered, the result will be imprecise. Is this always true? There is a unique exception: when you multiply an imprecise number by the exact number 0, you will get the exact 0. Try: $0 * 1.4$, and compare with $0. * 1.4$.

2) Not only can the number of decimal places of real numbers be large, but the number of digits of integers also. Try 100!. It is never necessary to tell in advance to GP the size of the integers that it will encounter, since this is adjusted automatically. On the other hand, for many computations with real numbers, it is necessary to specify a default precision (initially 28 digits).

3) Come back to 28 digits of precision (`\precision=28`), and type `exp(24*pi)`. What is this $*$ that you get at the end of the result?

Well, GP never wants you to believe that a result is correct when it is not. We are working with 28 digits of precision, but the integer part of `exp(24*pi)` has 33 decimal digits. Hence GP dutifully prints out 33 digits, but it prints a $*$ to warn you that the last few digits are wrong (without telling you how many). The $*$ can thus be interpreted as a decimal point with a warning message.

4) To avoid this $*$, there are two ways. One is of course to increase the precision to more than 33 decimals. Let's try it. To give it a wide margin, we set the precision to 40 decimals. Then we recall our last result (`%` or `%n` where `n` is the number of the result). What? We still have the $*$! Do you understand why?

Again let's try to see what's happening. The number you recalled had been computed only to 28 decimals, and even if you set the precision to 1000 decimals, GP knows that your number has only 28 digits of accuracy but an integral part with 33 digits. So you haven't improved things by increasing the precision. Or have you? What if we retype `exp(24*pi)` now that we have 40 digits? Try it. Now we do not have any warning, and we see the beginning of what's after the decimal point. We also see that at 28 decimals the last 6 digits were wrong.

5) There is still another method to avoid the warning: put yourself back into precision 28. Now type `\format=e0.28`. The e instead of the g which is printed when GP greets you, means that real numbers (by opposition to integers or rational numbers) will be printed in *exponential* format, i.e. with a E xxx denoting powers of 10. Try again `exp(24*pi)` in precision 28. Now we don't have any warning, and the reason is clear: GP has to print only a single digit for the integer part, and then it prints out as many decimals as was asked by the format (here 28).

6) *Warning*. Try the following: starting in precision 28, type successively `\format=e0.50`, then `exp(24*pi)`. Do you understand why the result is so bad, and why there are lots of zeros at the end? Convince yourself by typing `log(exp(1))`. The moral is that the `format` command changes only the output format, but *not* the default precision. On the other hand, the `precision` command changes both the precision and the output format.

6.2 Warming up

Another thing you better get used to pretty fast is error messages. Try typing 1/0. Couldn't be clearer. Taking again our universal example in precision 28, type `floor(exp(24*pi))` (`floor` is the mathematicians' integer part, not to be confused with `trunc`, which is the computer scientists': `floor`(-3.4) is equal to -4 while `trunc`(-3.4) is equal to -3). You get a more cryptic error message, which you would immediately understand if you had read the additional comments of the preceding section. Since I told you not to read them, the explanation is simply that GP is unable to compute the integer part of `exp(24*pi)` given only 28 decimals of accuracy, since it has 33 digits.

Some error messages are even much more cryptic than that and are sometimes not so easy to understand (well, it's nothing compared to TEX's error messages! :-).

For instance, try `log(x)`. Not really clear, is it? But no matter, it simply tells you that GP simply does not understand what `log(x)` is (although it does know the `log` function).

Now let's try `sqrt(-1)` to see what error message we get now. Haha! GP even knows about complex numbers, so impossible to trick it that way. Similarly, try typing `log(-2)`, `exp(i*pi)`, `i^i`,.... So we have a lot of real and complex analysis at our disposal (note that there is always a specific branch of multivalued complex transcendental functions which is taken, specified in the manual).

Just for fun, let's try `6*zeta(2)/pi^2`. Pretty good? (If you don't know what the Riemann zeta function is, don't worry, we won't use it in this tutorial).

Now GP didn't seem to know what `log(x)` was, although it did know how to compute numerical values of `log`. This is annoying. Maybe it knows the exponential function? Let's give it a try. Type `exp(x)`. What's this? If you had had any experience with other systems, the answer should have simply been `exp(x)` again. But here the answer is the Taylor expansion of the function around `x=0`, to 16 terms (16 is the default `serieslength`, which can be changed by typing `\serieslength=n`, where `n` is the number of terms that you want in your power series (note the `O(x^16)` which ends the series, and which is trademark of this type of object in GP. It is the familiar "bigoh" notation of analysis).

You will thus automatically get the Taylor expansion of any function that can be expanded around `x=0`, and incidentally this explains why we weren't able to do anything with `log(x)` which is not defined at 0. By if we try `log(1+x)`, then it works. But what if we wanted the expansion around a point different from 0? Well, you're able to change `x` into `x-a`, aren't you? So for instance you can type `log(x+2)` to have the expansion of `log` around `x=2`.

As exercises, try `cos(x)`, `cos(x)^2+sin(x)^2`, `exp(cos(x))`, `exp(exp(x)-1)`, `gamma(1+x)`, with different values of `serieslength` if you like.

Let's try something else: type `(1+x)^3`. No bigoh here, since the result is a polynomial. Haha, but I have learnt that if you do not take exponents which are integers greater or equal to 0, you obtain a power series with an infinite number of nonzero terms. Let's try. Type `(1+x)^(-3)` (the parentheses around -3 are not necessary but make things easier to read). Surprise! Contrary to what we expected, we don't get a power series but a rational function. Again this is for the same reason that 1/7 just gave you 1/7: the result being exact, PARI doesn't see any reason to make it nonexact.

But I still want that power series. To obtain it, just do as in the 1/7 example: type

`(1+x)^(-3)+O(x^16)`, `(1+O(x^16))*(1+x)^(-3)`, `(1+x+O(x^16))^(-3)`,etc...

Now try `(1+x)^(1/2)`. Now we obtain a power series, since the result is an object which PARI does not know how to represent exactly (we could teach PARI about algebraic functions, but then take `(1+x)^pi` as another example). This gives us still another solution to our preceding exercise: we can type `(1+x)^(-3.)`. Since $-3.$ is not an exact quantity, PARI has no means to know that we are dealing with a rational function, and will instead give you the power series.

Finally, if you want to be really fancy, you can type `taylor((1+x)^(-3),x)` (look at the entry for `taylor` for the description of the syntax), but this is in fact almost never used.

To summarize, in this section we have seen that in addition to integers, real numbers and rational numbers, PARI can handle complex numbers, polynomials, power series, rational functions. A large number of functions exist which handle these types, but in this tutorial we will only look at a few.

Additional comments

As before, you are supposed to skip this at first, and come back later.

1) A complex number has a real part and an imaginary part (who would have guessed?). However, beware that when the imaginary part is the exact integer zero, it is not printed, but the

complex number is not converted to a real number. Hence it may *look* like a real number without being one, and this may cause some confusion in programs which expect real numbers. For example, type `3+i-i`. The answer is, as expected 3. But it is still a complex number. Hence if you then type `(1+x)^%`, instead of getting the polynomial `1+3*x+3*x^2+x^3` as expected, you obtain a power series. Worse, when you try to apply an arithmetic function, say the Euler `phi` function, you get the error message which says that "arithmetic functions want integer arguments", which you would have guessed yourself, but the message is difficult to understand since 3 looks like a genuine integer!!! (Please read again if this is not clear. It is a common source of incomprehension).

Similarly, `3+x-x` is not the integer 3 but the constant polynomial equal to 3.

If you want the final expression to be in the simplest form possible (for example before applying an arithmetic function, or simply because things will go faster afterwards), apply the function `simplify` to the result.

2) As already stated, power series expansions are always implicitly around `x=0`. If you want them around `x=a`, replace `x` by `z+a` in the function that you want to expand. For example, to have the expansion of `exp(x)` around `x=1`, we simply type `exp(z+1)`, where `z` stands for `x-a`. For complicated functions, it may be simpler to use the substitution function `subst`. For example, if `p=1/(x^4+3*x^3+5*x^2-6*x+7)`, you may not want to retype this, replacing `x` by `z+a`, so you can write `subst(p,x,z+a)` (look up the exact description of the `subst` function).

Now try typing `p=1+x+x^2+O(x^10)`, then `subst(p,x,z+1)`. Do you understand why you get an error message?

6.3 The Rest of the PARI Types

Let's handle first some more the basic PARI types.

Type `p=x*exp(-x)`. As expected, you get the power series expansion to 16 terms (if you have not changed the default). Now type `pr=reverse(p)`. You are asking here for the *reversion* of the power series p, in other words the inverse function. This is possible only for power series whose first non-zero coefficient is that of x^1. To check the correctness of the result, you can type `subst(p,x,pr)` or `subst(pr,x,p)` and you should get back `x+O(x^17)`.

Now the coefficients of `pr` obey a very simple formula. First, we would like to multiply the coefficient of `x^n` by `n!` (in the case of the exponential function, this would simplify things considerably!). The PARI function `laplace` does just that. So type `ps=laplace(pr)`. The coefficients now become integers, which can be immediately recognized by inspection. The coefficient of `x^n` is now equal to $\mathrm{n}^{\mathrm{n}-1}$. In other words, we have

$$\mathrm{pr} = \sum_{\mathrm{n}\geq 1} \frac{\mathrm{n}^{\mathrm{n}-1}}{\mathrm{n}!} X^{\mathrm{n}}.$$

Do you know how to prove this? (If you have never seen this, the proof is difficult.)

Of course PARI knows about vectors (rows and columns are distinguished, even though mathematically there is no difference) and matrices. Type for example `[1,2,3,4]`. This gives the row vector whose coordinates are 1, 2, 3 and 4. If you want a column vector, type `[1,2,3,4]~`, the tilde meaning of course transpose. You don't see much difference in the output, except for the tilde at the end. However, now type `\b`: lo and behold, the vector does become a column. This command is used mainly for this purpose.

Type `m=[a,b,c;d,e,f]`. You have just entered a matrix with 2 rows and 3 columns. Note that the matrix is entered by *rows* and the rows are separated by semicolons ";". The matrix is printed naturally in a rectangle shape. If you want it printed horizontally just as you typed it, type `\r`, or if you want this type of printing to be the permanent default type `\p` twice (the first time puts GP in a mode which in the present version you do not want to use). Type `\p` a third time if you want to come back to the original default.

Now type `m[1,2]`, `m[1,]`, `m[,2]`. Are explanations necessary? (In an expression such as `m[j,k]`, the `j` always refers to the row number, and the `k` to the column number, and the first index is always 1, never 0. This default cannot be changed.)

Even better, type `m[1,2]=5;m` (the semicolon allows us to put several instructions on the same line. It is an absolute necessity when we program in GP). Now type `m[1,]=[15,−17]`. No problem. Finally type `m[,2]=[j,k]`. You have an error message since you have typed a row vector, while `m[,2]` is a column vector. If you type instead `m[,2]=[j,k]~` it works.

Type now `h=hilbert(20)`. You get the so-called "Hilbert matrix" whose coefficient of row `i` and column `j` is equal to `1/(i+j−1)`. Incidentally, the matrix `h` takes too much room. If you don't want to see it, simply type a semi-colon ";" at the end of the line (`h=hilbert(20);`). This is an example of a "precomputed" matrix, built-in PARI. There are only a few. We will see later an example of a much more general construction.

What is interesting about Hilbert matrices is that first their inverses and determinants can be computed explicitly (and the inverse has integer coefficients), and second they are numerically very unstable, which make them a severe test for linear algebra packages in numerical analysis. Of course with PARI, no such problem can occur: since the coefficients are given as rational numbers, the computation will be done exactly, so there cannot be any numerical error. Try it. Type `d=det(h)` (you have to be a little patient, this is quite a complicated computation). The result is a rational number (of course) of numerator equal to 1 and denominator having 226 decimal digits. How do I know, by the way? I did not count! Instead, simply type `1.*d`. The result is now in exponential format, and the exponent gives us the answer.

Now type `hr=1.*h;` (do not forget the semicolon, we don't want to see all the junk!), then `dr=det(hr)`. You notice two things. First the computation, although not instantaneous, is much faster than in the rational case. The reason for this is that PARI is handling real numbers with 28 digits of accuracy, while in the rational case it is handling integers having up to 226 decimal digits.

The second more important fact is that the result is terribly wrong. If you compare with `1.*d` computed earlier, which is correct, you will see that only 2 decimals agree! This catastrophic instability is as already mentioned one of the characteristics of Hilbert matrices. In fact, the situation is much worse than that. Type `norml2(1/h−1/(h*1.))` (the function `norml2` gives the square of the L^2 norm, i.e. the sum of the squares of the coefficients). Again be patient since computing `1/h` will take even more time (not much) than computing `det(h)`. The result is larger than 10^{50}, showing that some coefficients of `1/(h*1.)` are wrong by as much as 10^{24} (the largest error is in fact equal to $7.644 10^{24}$ for the coefficient of row 15 and column 14, which is a 28 digit integer).

To obtain the correct result after rounding for the inverse, we have to use a default precision of 56 digits (try it).

Although vectors and matrices can be entered manually, by typing explicitly their elements, very often the elements satisfy a simple law and one uses a different syntax. For example, assume that you want a vector whose `k`-th coordinate is equal to `k^2`. No problem, type for example `vector(10,k,k^2)` if you want a vector of length 10. Similarly, if you type

```
matrix(5,5,j,k,1/(j+k-1))
```

you will get the Hilbert matrix of order 5 (hence the `hilbert` function is unnecessary). The `j` and `k` represent dummy variables which are used to number the rows and columns respectively (in the case of a vector only one is present of course). You must not forget, in addition to the dimensions of the vector or matrix, to indicate explicitly the names of these variables.

Important warning. The letter `i` is reserved for the complex number equal to the square root of -1. Hence it is absolutely forbidden to use it as a variable. Try typing `vector(10,i,i^2)`, the error message that you get clearly indicates that GP does not consider `i` as a variable. Also remember that (for now) GP is case insensitive so the variable `I` is just as forbidden.

When creating vectors or matrices, it is often useful to use boolean operators and the `if()` statement (see the section on programming for details on using this statement). Indeed, an `if` expression has a value, which is of course equal to the evaluated part of the `if`. So for example you can type

```
matrix(8,8,j,k,if((j-k)%2,x,0))
```

to get a checkerboard matrix of `x` and `0`. Note however that a vector or matrix must be *created* first before being used. For example, it is forbidden to write

```
for(j=1,5,v[j]=1/j)
```

if the vector `v` has not been created beforehand (for example by a `v=vector(5,j,0)` command).

Important remark

A drawback of the present GP implementation is that each time a vector or matrix is modified, even in a single component, the whole vector is copied (this is of course not the case in library mode). This implies that the necessary time for filling up a vector recursively is not proportional to its length but to the the square of it length, which can be painful if the length is at all large. The only exception to this, and an important one of course, is the direct use of the `vector` or `matrix` functions which do *not* fill the vector or matrix recursively but directly, hence take linear time.

Consider the following example. We want to store the first 400 values of the sequence defined by $u_0 = 0$, $u_1 = 5$, $u_{n+1} = \sqrt{u_n^2 + 3}\text{-}u_{n-1}$. A reasonable way to do this is through the following GP program:

```
a=0;b=5;v=[a,b];for(j=3,400,c=sqrt(b*b+3)-a;a=b;b=c;v=concat(v,c))
```

If you time the execution of this program and then try the same thing with 400 replaced by 800 you will see that the time has almost been multiplied by 4 although the computations are not harder for the last 400 iterations than for the first.

One method to bypass this drawback of the GP implementation is to try to limit to short vectors. In the example above, we would for example store each hunk of 100 consecutive values in *different* vectors, and start with the last values reached to fill in the next hunk. At the end we can concatenate all the vectors together to get a single vector as desired.

The last PARI types which we have not yet played with are types which are closely linked to number theory (hence people not interested in number theory can skip).

The first is the type "integermod". Let us see an example. Type `n=10^15+3`. We want to know whether `n` is prime or not. Of course we could make use of the builtin facilities of PARI, but let us do otherwise. We first trial divide by the builtin table of primes, by default up to 500000. We slightly cheat here and use the function `smallfact` which does exactly this. So type `smallfact(n)`. The result is a 2 column matrix (as all factoring programs), the first column giving the primes and the second their exponents. Here we get a single row, telling us that `n` is not divisible by any prime up to the default (500000). We could now trial divide further, or even cheat completely and call the PARI function `factor`, but before we do this let us see how to get an answer ourselves.

By Fermat's little theorem, if `n` is prime we must have $\mathrm{a}^{\mathrm{n}-1} \equiv 1 \pmod{\mathrm{n}}$ for all `a` not divisible by `n`. Hence we could try this with `a=2` for example. But $2^{\mathrm{n}-1}$ is a number with approximately $3 \cdot 10^{14}$ digits, hence impossible to write down, even less to compute. But instead type `a=mod(2,n)`. This creates the number 2 considered now as an element of the ring $R = \mathbf{Z}/\mathrm{n}\mathbf{Z}$. The elements of R, called integermods, can always be represented by numbers smaller than `n`, hence very small. Fermat's theorem can be rewritten $\mathrm{a}^{\mathrm{n}-1}$ `=mod(1,n)` in the ring R, and this can be computed very efficiently. Type `a^(n-1)`. The result is definitely *not* equal to `mod(1,n)`, thus *proving* that `n` is not a prime (if we had obtained `mod(1,n)` on the other hand, it would have given us a hint that `n` is maybe prime, but never a proof).

To find the factors is another story. One must use less naive techniques than trial division (or be very patient). To finish this example, type `factor(n)` to see the factors. Since the smallest factor is 14902357, you would have had to be very patient with trial division!

The second specifically number-theoretic type is the p-adic numbers. I have no room for definitions, so please skip if you have no use for such beasts. A p-adic number is entered as a rational or integer valued expression to which is added `O(p^n)` (or simply `O(p)` if `n=1`) where `p` is the prime and `n` the p-adic precision. Apart from the usual arithmetic operations, you can apply a number of transcendental functions. For example, type `n=569+O(7^8)`. Then type `s=sqrt(n)`, you obtain one of the square roots of `n` (if you want to check, type `s*s-n`). Type now `l=log(n)`, then `e=exp(l)`. If you know about p-adic logarithms, you will not be surprised that `e` is not equal to `n`. Type `(n/e)^6`: `e` is in fact equal to `n` times a $(\mathrm{p}-1)$-st root of unity.

Incidentally, if you want to get back the integer 569 from the p-adic number `n`, type `trunc(n)`.

The third number-theoretic type is the type "quadratic number". This type is specially tailored so that we can easily work in a quadratic extension of a base field (usually $\mathbf{Q}$). It is a generalization of the type "complex". To start, we must specify which quadratic field we want to work in. For this, we use the function `quadgen` applied to the *discriminant* `d` (as opposed to the radicand) of the quadratic field. This returns a number (always printed as `w`) equal to `(d+a)/2` where `a` is equal to 0 or 1 according to whether `d` is even or odd. The behavior of `quadgen` is a little special: although its result is always printed as `w`, the variable `w` itself is not set to that value. Hence it is necessary to write systematically `w=quadgen(d)` using the variable name `w` (or `w1` etc.. if you have several quadratic fields), otherwise you will be confused.

So type `w=quadgen(-163)`, then `char(w,x)` which asks for the characteristic polynomial of `w` expressed with the variable `x`. The result shows what `w` will represent. You can also ask for `1.*w` to see which root of the quadratic has been taken, but this is rarely necessary. We can now play in the field $\mathbf{Q}(\sqrt{-163})$. Type for example `w^10, norm(3+4*w), 1/(4+w)`. More interesting, type `a=mod(1,23)*w` then `b=a^264`. This is a generalization of Fermat's theorem to quadratic fields. If you do not want to see the 23 all the time, type `lift(b)`. Another example, type `p=` $\mathrm{x}^2 + \mathrm{w} * \mathrm{x} + 5 * \mathrm{w} + 7$, then `norm(p)` (or even better `simplify(norm(p))`, see above). We thus obtain the quartic equation over **Q** corresponding to the relative quadratic extension over **Q**(w) defined by `p`.

On the other hand, if you type `wr=sqrt(w^2)`, do not expect to get back `w`. Instead, you get the numerical value, the function `sqrt` being considered as a "transcendental" function, even though it is algebraic. Type `algdep(wr,2)`. This is one way to get `w` back. Similarly, type `algdep(sqrt(3*w+5),4)`. See the users manual for the function `algdep`.

The fourth number-theoretic type is the type "polymod", i.e. polynomial modulo another polynomial. This type is used to work in algebraic extensions, for example elements of number fields (if the base field is **Q**), or elements of finite fields (if the base field is $\mathbf{Z}/p\mathbf{Z}$ for a prime p, defined by an integermod). In a sense it is a generalization of the type quadratic number. The syntax used is the same as for integermods. For example, instead of typing `w=quadgen(-163)`, you can type `w=mod(x,x^2-x+41)`. Then exactly as in the quadratic case you can type `w^10`, `norm(3+4*w)`, `1/(4+w)`, `a=mod(1,23)*w`, `b=a^264`, obtaining of course the same results (type `lift(...)` if you don't want to see the polynomial `x^2-x+41` repeated all the time). Of course, the basic interest is that you can work in any degree, not only quadratic.

There is however a slight difference in behavior. Keeping our polymod `w`, type `1.*w`. As you can see, the result is not the same. Type `sqrt(w)`. Here, we obtain a vector with 2 components, the two components being the principal branch of the square root of all the possible embeddings of `w` in **C** (NOT the two square roots). More generally, if `w` was of degree n, we would get an n-component vector, and similarly for transcendental functions.

We have at our disposal the usual arithmetic functions, plus a few others. For example, type `a=mod(x,`$\mathtt{x}^3-\mathtt{x}-1$`)` defining a cubic extension. We can for example ask for `b=a^7`. But now assume we want to express `a` as a polynomial in `b`. This is possible since `b` is also a generator of the same field. No problem, type `modreverse(b)`. This gives a new defining polynomial for the same field (i.e. the characteristic polynomial of `b`), and expresses `a` in terms of this new polymod, i.e. in terms of `a`. We will see this in much more detail in the number field section.

6.4 Elementary Arithmetic Functions

Since PARI is aimed at number theorists, it is not surprising that there exists a large number of arithmetic functions (see the list in the corresponding section of the users manual). We have already seen several, such as `factor`. Note that `factor` factors not only integers, but also polynomials. Type for example `factor(x^15-1)`. You can also ask to factor a polynomial modulo a prime p (`factmod`) and even in a finite field which is not the prime field (`factfq`).

Evidently, you have functions for computing GCD's, extended GCD's, solving the Chinese remainder theorem and so on.

In addition to the factoring facilities, you have a few functions related to primality testing such as `isprime`, `ispsp` and `nextprime`. Note that in all these cases, only strong pseudoprimes are produced, there is no sophisticated primality test.

The usual multiplicative arithmetic functions exist (the Möbius μ function (`mu`) the Euler ϕ function (`phi`), the ω and Ω functions (`omega` and `bigomega`), the sum of divisors functions (`sigma` and `sigmak`), etc...

You can compute continued fractions. For example, if you type `\precision=1000` then `cf(exp(1))` you obtain the continued fraction of the base of natural logarithms, which as you can see obeys a very simple pattern (can you prove it?).

In many cases, one wants to perform some task only when an arithmetic condition is satisfied. GP gives you the following functions: `isprime` as mentioned above, `issquare`, `isfund` to test whether an integer is a fundamental discriminant (i.e. the discriminant of the ring of integers of a quadratic field), and the `forprime` and `divsum` loops. Assume for example that we want to compute the product of all the divisors of a positive integer n. One way is to write:

```
round(exp(divsum(n,d,log(d))))
```

(there is a very simple formula for this product: find and prove it).

If we want to know the list of primes p less than 1000 such that 2 is a primitive root modulo p, one way would be to write:

```
forprime(p=3,1000,if(primroot(p)==2,print(p),))
```

Note that this assumes that `primroot` returns the smallest primitive root, and this is indeed the case.

Functions related to quadratic fields, binary quadratic forms and general number fields will be seen in the next sections.

6.5 Working in Quadratic Number Fields

The simplest of all number fields outside **Q** are quadratic fields. Such fields are characterized by their discriminant, and even better, any nonsquare integer D congruent to 0 or 1 modulo 4 is the discriminant of a specific order in a quadratic field. We can check whether this order is maximal by using the command `isfund(D)`. Elements of a quadratic field are of the form $a + b\omega$, where ω is chosen as $\sqrt{D}/2$ if D is even and $(1+\sqrt{D})/2$ if D is odd, and are represented in Pari by quadratic numbers. To initialize working in a quadratic order, one should start by the command

```
w=quadgen(D)
```

This sets `w` equal to ω as above, and is printed `w`. Note however that if several different quadratic orders are used, a printed `w` may have several different meanings. For example if you type

```
a=quadgen(-23);b=quadgen(-15);
```

Then ask for the value of `a` and `b`, both will be equal to `w`, but of course they are not equal. Hence beware when dealing with several quadratic orders at once.

In addition to elements of a quadratic order, we also want to be able to handle ideals of such orders. In the quadratic case, it is equivalent to handle binary quadratic forms, and this has been chosen in Pari. For negative discriminants, quadratic forms are triples (a, b, c) representing the form $ax^2 + bxy + cy^2$. Such a form will be printed as (and can be created by)

```
qfi(a,b,c)
```

(`i` stands for imaginary).

Such forms can be multiplied, divided, powered as many Pari objects using the usual operations, and they can also be reduced using the function `redimag` (it is not the purpose of this tutorial to explain what all these things mean). In addition, Shanks's NUCOMP algorithm has been implemented (functions `nucomp` and `nupow`), and this is usually a little faster.

Finally, you have at your disposal the functions `classno` which gives (usually) the class number, the function `hclassno` which gives the Hurwitz class number, and the much more sophisticated `buchimag` function which gives the class number and class group structure, even for large discriminants.

Let us see examples of all this at work.

Type `classno(-10007)`. GP tells us that the result is 77. However, you may have noticed in the explanation above that the result is only usually correct. This is because the implementers of the algorithm have been lazy and have not put the complete Shanks algorithm in Pari, causing it to fail in certain very rare cases. In practice, it is almost always correct, and the much more powerful `buchimag` program, which *is* complete, can give confirmation.

So we may be a little suspicious of this class number. Let us check it. First, we need to find a quadratic form of discriminant -10007. Since this discriminant is congruent to 1 modulo 8, we know that there is an ideal of norm equal to 2, i.e. a binary quadratic form (a, b, c) with $a = 2$. To compute it we type `f=pf(-10007,2)` (`pf` stands for primeform, i.e. binary forms corresponding to prime ideals). OK, now we have a form. If the class number is correct, the very least is that this formed raised to the power 77 should equal the identity. Let's check this. Type `f^77`. We get a form starting with 1, i.e. the identity, so this test is OK. Raising f to the power 11 and 7 does not give the identity, hence we now know that the order of f is exactly 77, hence the class number is a multiple of 77. But how can we be sure that it is exactly 77 and not a multiple? Well, type

```
prodeuler(p=2,500,1./(1-kro(-10007,p)/p))*sqrt(10007)/pi
```

This is nothing else than an approximation to the Dirichlet class number formula. The function `kro` is the Kronecker symbol, in this case simply the Legendre symbol. Note also that we have written `1./(1-...)` with a dot after the 1. Otherwise, Pari may want to compute the whole thing as a rational number, which would be terribly long and useless. In fact Pari does no such thing in this particular case, but you never know. Better safe than sorry!

We find 77.77, pretty close to 77, so things seem in order. Explicit bounds on the prime limit to be used in the Euler product can be given which make the above reasoning rigorous.

Let us try the same thing with $D = -3299$. `classno` and the Euler product convince us that the class number must be 27. However, we get stuck when we try to prove this in the simple-minded way above. Indeed, we type `f=pf(-3299,3)` (here 2 is not the norm of a prime ideal but 3 is), and we see that f raised to the power 9 is equal to the identity. This is the case for any other quadratic form we choose. So we suspect that the class group is not cyclic. Indeed, if we list all 9 distinct powers of f, we see that the primeform `pf(-3299,5)` is not on the list (although its cube is as it must). This implies that the class group is probably equal to a product of a cyclic group of order 9 by a cyclic group of order 3. The Euler product plus explicit bounds prove this.

Another way to check it is to use the `buchimag` function by typing for example

```
buchimag(-3299,1,6,10)
```

Indeed the class group is as we thought (see Chapter 3 for the description of the different components of the result).

Working in real quadratic fields instead of complex ones, i.e. with $D > 0$, is not very different.

The same `quadgen` function is used to create elements. Ideals are again represented by binary quadratic forms (a, b, c), this time indefinite. However, the Archimedean valuations of the number field start to come into play (as is clear if one considers ideles instead of ideals), hence in fact quadratic forms with positive discriminant will be represented as a quadruplet (a, b, c, d) where the quadratic form itself is $ax^2 + bxy + cy^2$ with a, b and c integral, and d is the Archimedean component, a real number. For people familiar with the notion, d represents a "distance" as defined by Shanks and Lenstra.

To create such a form we write

```
qfr(a,b,c,d)
```

(`r` stands for real).

Again these forms can be multiplied, divided, powered, and they can be reduced using the function `redreal`. This function is in fact a succession of elementary reduction steps corresponding essentially to a continued fraction expansion, and a single one of these steps can be achieved using the function `rhoreal`. Since handling the fourth component d usually involves computing logarithms, the functions `redrealnod` and `rhorealnod` are also provided (`nod` meaning no distance) so that the fourth component is ignored. Finally, it is sometimes useful to operate on forms of positive discriminant without performing any reduction (this is useless in the negative case), the functions `comprealraw` and `powrealraw` are available which avoid any reduction.

Again, the function `pf` gives a prime form, but the form which is given corresponds to an ideal of prime norm which is usually not reduced. If desired, it should be reduced with one of the function `redreal` or `redrealnod`.

Finally, you have at your disposal the function `classno` which gives the class number (this time guaranteed correct), `regula` which gives the regulator, and the much more sophisticated `buchreal` giving both the class group structure and its generators, and the regulator, even for large discriminants. The `classno` and `regula` program use a $O(\sqrt{D})$ algorithm hence become very slow for discriminants having more than 10 digits. `buchreal` can be used on a much larger range.

Let us see examples of all this at work and learn some little known number theory at the same time. First of all, type `d=3*3299;classno(d)`. We see that the class number is 3 (we know in advance that it must be divisible by 3 from the `d=-3299` case above and Sholtz's theorem). Let us create a form by typing `f=redrealnod(pf(d,2),isqrt(d))`. This gives us a prime ideal of norm equal to 2. Is this ideal principal ? Well, one way to check this, which is not the most efficient but will suffice for now, is to look at the complete cycle of reduced forms equivalent to `f`, using the function `rhoreal`. Type `g=f;for(j=1,20,g=rhoreal(g);print(g))`. We see that we come back to the form `f` without having the principal form (starting with ± 1) in the cycle, so the ideal corresponding to `f` is not principal.

Since the class number is equal to 3, we know however that `f^3` will be a principal ideal $\alpha \mathbf{Z}_K$. How do we find α? For this, type `f3=powrealraw(f,3)`. This computes the cube of `f`, without reducing it. Hence it corresponds to an ideal of norm equal to $8 = 2^3$, so we already know that the norm of α is equal to ± 8. We need more information, and this will be given by the fourth component of the form. Apply `rhoreal` to `f3` until you reach the unit form (you must do this 6 times).

Extract the Archimedean component by typing `c=compo(%,4)`. By definition of this component, we know that

$$\frac{\alpha}{\sigma(\alpha)} = \pm e^{2c},$$

where σ denotes real conjugation in our quadratic field. Thus, if we type

```
a=sqrt(8*exp(2*c))
```

and then `sa=8/a`, we know that up to sign, `a` and `sa` are numerical approximations of α and $\sigma(\alpha)$. Of course, α can always be chosen to be positive, and a quick numerical check shows that the difference of `a` and `sa` is close to an integer, and not the sum, so that in fact the norm of α is equal to -8 and the numerical approximation to $\sigma(\alpha)$ is `-sa`. Thus we type

```
p=x^2-round(a-sa)*x-8
```

and this is the characteristic polynomial of α. We can check that the discriminant of this polynomial is a square multiple of `d`, so α is indeed in our field. More precisely, solving in α and using the numerical approximation that we have to solve the sign ambiguity in the square root, we get explicitly $\alpha = (15221 + 153\sqrt{d})/2$. Note that this can also be done automatically using the functions `polred2` and `modreverse`, as we will see later in the general number field case, or by solving a system of 2 linear equations in 2 variables.

Exercise: now that we have α explicitly, check that it is indeed a generator of the ideal corresponding to the form `f3`.

Let us now play a little with cycles. Type

```
buchreal(10000001,0,1,6,10)
```

We get as a result a 5-component vector, which tells us that the class number is equal to 1, and the regulator is equal to $2641.5\ldots$. Now just as an example, let's assume that we want the regulator to 500 decimals, say. I claim that by simply knowing the crude approximation above, this can be computed with no effort (of course we could redo the whole computation of `buchreal` after having changed the precision, but it would be a waste of time).

This time, we want to start with the unit form. Since $D = 10000001$ is odd, after having set D equal to that value we can type

```
u=redrealnod(qfr(1,1,(1-D)/4,0.),isqrt(D))
```

We use the function with no distance since we want the initial distance to be equal to 0. The function `redrealnod` asks for a second argument which is the integer part of the square root of the discriminant, obtained using the function `isqrt`.

Now we type `f=rhoreal(u)`. This is the first form encountered along the principal cycle. For the moment, keep the precision low, for example the initial default precision. The distance from the identity of `f` is around 4.253. Very crudely, since we want a distance of 2641.5, this should be encountered approximately at $2641.5/4.253 = 621$ times the distance of `f`. Hence, as a first try, we type `f^621`. Oops, we overshot, since the distance is now 3173.02. Now we can refine our initial estimate and believe that we should be close to the correct distance if we raise `f` to the power $621 * 2641.5/3173$ which is close to 517. Now if we compute `f^517` we hit the principal form right on the dot. Note that this is not a lucky accident: we will always land extremely close to the correct target using this method, and usually at most one `rhoreal` correction step is necessary. Of course, only the distance component can tell us where we are along the cycle.

Up to now, we have only worked to low precision. The goal was to obtain this unknown integer 517. Note that this number has absolutely no mathematical significance: indeed the notion of reduction of a form with positive discriminant is not well defined since there are usually many reduced forms equivalent to a given form. However, when Pari makes its computations, the specific order and reductions that it performs is dictated entirely by the coefficients of the quadratic form itself, and not by the distance component, hence the precision used has no effect.

Hence we now start again by setting the precision to (for example) 500, we retype the definition of `u` (why is this necessary?), and then `f=rhoreal(u)` and finally `f^517`. Of course we know in advance that we land on the unit form, and the fourth component gives us the regulator to 500 decimal places with no effort at all.

In a similar way, we could obtain the so-called *compact representation* of the fundamental unit itself, or p-adic regulators. I leave this as exercises for the interested reader.

6.6 Working in General Number Fields

(*Note: the sequence of instructions described in this section can be found as a GP-readable file in the directory* `examples` *under the name* `tutnf.gp`.)

The situation here is of course more difficult. First of all, we need to find a reasonable equation defining our field. This is done using a variant of the `polred` function which initializes a number of things which will be needed. For example assume that we want to work in the number field K defined by one of the roots of the equation $x^4 + 24x^2 + 585x + 1791 = 0$. We can type

```
T=x^4+24*x^2+585*x+1791;polred(T)
```

and see from the third component that the polynomial $x^4 - x^3 - 21x^2 + 17x + 133$ defines the same field with smaller coefficients. The `polred` function gives a (usually) simpler polynomial, and also sometimes some information on the existence of subfields. For example in this case, the second component of `polred` tells us that the field defined by $x^2 - x + 1 = 0$, i.e. the field of cube roots of unity, is a subfield of our number field K.

However to start working in a number field it is preferable to use a variant of `polred` which gives a single polynomial and stores some useful information which will be needed later. This is the function `initalg`, which of course stands for initialization of algebraic number field.

Type `nf=initalg(T)`. You get a vector containing 9 components. The first is the new polynomial which Pari chooses to use for working in your number field. The second contains the signature (r_1, r_2) of the field, the third the field discriminant, the seventh contains an integral basis, etc...See Chapter 3 for the complete list.

Call θ a root of your initial polynomial, and α a root of the one that `initalg` has found. These are algebraic numbers, and as already mentioned are represented as polymods. For example, in our special case θ is equal to the polymod

```
mod(x,x^4+24*x^2+585*x+1791)
```

while α is equal to the polymod

```
mod(x,x^4-x^3-21*x^2+17*x+133)
```

Here of course we are considering only the algebraic aspect, and hence ignore completely *which* root θ or α is chosen.

Now probably you may have a number of elements of your number field which are expressed as polymods with respect to your old polynomial, i.e. as polynomials in θ. Since we are now going to work with α instead, it is necessary to convert these numbers to a representation using α. This is done using the function `initalg2`. For example if you type `initalg2(T)` you obtain as a result the polymod

```
mod(x^2+x-2,x^4-x^3-21*x^2+17*x+133)
```

meaning that $\theta = \alpha^2 + \alpha - 2$, and hence the conversion from a polynomial in θ to one in α is easy. If we want to do the reverse, i.e. go back from a representation in α to a representation in θ, we use the function `modreverse` on the polymod obtained by `initalg2`. Try it. The result has a big denominator (147), essentially because our initial polynomial T was too large.

From now on, we completely forget about our initial polynomial T, and use only the polynomial $x^4 - x^3 - 21x^2 + 17x + 133$ defining α, and the components of the vector `nf` which gives information on our number field K. Note that even if the polynomial has small coefficients, it is necessary to use the `initalg` function before being able to do **ideal** operations in the number field. Of course `initalg` is not really necessary if only **element** operations are performed.

Let us first perform some operations on elements. Type `A=nf[1]` to get the new polynomial, and type

`u=mod(x^3-5*x^2-8*x+56,A)/7`

We may for instance compute `u^3`. Try it. Or, we can type `1/u`. Better yet, if we want to know the norm from K to $\mathbf{Q}$ of `u`, we type `norm(u)` (what else?). Still better, if we want all the symmetric functions of `u` and not only the norm, we type `char(u,x)` (do not forget the `x` variable which tells GP which variable to use for the characteristic polynomial. Note that this gives the characteristic polynomial of `u`, and not in general the minimal polynomial. Exercises: how does one (easily) find the minimal polynomial from this? Find a simpler expression for `u`.)

OK, now let's work on the field itself. The `initalg` command already gave us some information. The field is totally complex (signature $[0,2]$), its discriminant is $D = 18981$ and $(1, \alpha, \alpha^2, (\alpha^3 + 2\alpha^2 + 6\alpha)/7)$ is an integral basis. The Galois group of its Galois closure can be obtained by typing `galois(A)`. The answer ($[8, -1, 1]$) shows that it is equal to D_4, the dihedral group with 8 elements, i.e. the group of symmetries of a square.

This implies that the field is "partially Galois", i.e. that there exists at least one non-trivial field isomorphism which fixes K. To find out which it is, we use the function `galoisconj`. This uses the LLL algorithm to find linear relations. Unfortunately, in the present version 1.38.1 of Pari this is quite unstable so the results are not as nice as we would like. Anyway, let's type `galoisconj(A)`. Oops, we get an error message (in certain versions you may not, in which case ignore the next 2 sentences). This is due to the bad LLL implementation and should disappear someday. In this particular instance, the problem disappears if we *lower* the precision to 18 decimals, say. The result tells us that, apart from the trivial automorphism, the map $\alpha \mapsto (-\alpha^3 + 5\alpha^2 + \alpha - 49)/7$ is a field automorphism. Indeed, if we type

`char(mod((-x^3+5*x^2+x-49)/7,A),x)`

we obtain the polynomial `A` once again.

Before working with ideals, let us note one more thing. The main part of the work of the `polred` or `initalg` function is to compute an integral basis, i.e. a $\mathbf{Z}$-basis of the maximal order $\mathbf{Z}_K$ of K. For a large polynomial, this implies factoring the discriminant of the polynomial, which is very often out of the question. There are two ways in which the situation may be improved. First, it is often the case that the polynomial that one considers is of quite a special type, giving some information on the discriminant. For example, one may know in advance that the discriminant is a square. Hence we can "help" Pari by giving it that information. More precisely, using the extra information that we have we may be able to factor the discriminant of the polynomial. We can then use the function `factoredbase` or `factoredpolred` (see Chapter 3) to inform Pari of this factorization. The function `factoredinitalg` has not been written, but it would of course be a trivial task.

The second way in which the situation may be improved is that often we do not need the complete information on the maximal order, but only require that the order be p-maximal for a certain number of primes p. This can again be done using the function `factoredbase` by cheating, and giving on purpose an incomplete factorization of the discriminant involving only the primes which we want. For example coming back to our initial polynomial T, the discriminant of the polynomial is $3^7 \cdot 7^6 \cdot 19 \cdot 37$. If we only want the 7-maximal order, we simply type

```
factoredbase(T,[7,6;1537461,1])
```

and the factors of 1537461 will not be looked at.

We now would like to work with ideals (and even with ideles) and not only with elements. An ideal will be represented by a square matrix giving the Hermite Normal Form of a $\mathbf{Z}$-basis of the ideal expressed on the integral basis. Note that, except when mentioned otherwise, we always represent elements on the integral basis and not on a power basis like the powers of α. An idele will be represented as a 2-element vector, the first element being the HNF of the corresponding ideal, which summarizes the non-archimedean information, and the second element is a vector of real and complex numbers with $r_1 + r_2$ components, the first r_1 being real, the second complex. In fact, to avoid certain ambiguities, the first r_1 components are allowed to have an imaginary part which is a multiple of π. These $r_1 + r_2$ components correspond to the Archimedean places of the number field K.

Finally, a prime ideal may either be represented as an ordinary ideal as explained above, but usually will be represented as a 5-element vector `[p,a,e,f,b]` as explained in the `primedec` heading of Chapter 3.

Let us keep our number field K above, and hence the vector `nf`. Type

```
pd=primedec(nf,7)
```

This gives the decomposition of the prime number 7 into prime ideals. We have chosen 7 because it is the index of $\mathbf{Z}[\theta]$ in $\mathbf{Z}_K$ (which is given as `nf[4]`) hence is the most difficult case to treat.

The result is a vector with 4 components, showing that 7 is totally split in the field K into prime ideals of norm 7. Let us take one of these ideals, say the first, so type `pr1=pd[1]`. To work further, we would like to have the Hermite Normal Form of this ideal. No problem: since ideal multiplication always gives the result in HNF, we will simply multiply our prime ideal by $\mathbf{Z}_K$, which is represented by the identity matrix. So type

```
hp=idealmul(nf,idmat(4),pr1)
```

and we have the desired HNF.

We can now perform ideal operations. For example type

```
hp3=idealmul(nf,hp,idealmul(nf,hp,hp))
```

or more simply

```
hp3=idealpow(nf,hp,3)
```

to get the cube of the ideal `pr1`. Since the norm of this ideal is equal to $343 = 7^3$, to check that it is really the cube of `pr1` and not of other ideals above 7, type

```
for(j=1,4,print(idealval(nf,hp3,pd[j])))
```

and we see that the valuation at `pr1` is equal to 3, while the others are equal to zero.

Let us now "idelize" hp3 by typing `hpi3=[hp3,[0.,0.]]`. (we need $r_1 + r_2 = 2$ components for the second vector). Then type `hr1=ideallllred(nf,hpi3,0)`. We get a new ideal which is equivalent to the first, and the Archimedean component is non-trivial. Now, just for fun type

```
hr=hr1;for(j=1,3,hr=ideallllred(nf,hr,[1,5]);print(hr))
```

We see that the third `hr` is equal to the initial `hr1`. This means that we have found a unit in our field, and it is easy to extract this unit given the Archimedean information (clearly it would be impossible without). We first form the difference of the Archimedean contributions of `hr` and `hr1` by typing

```
arch=hr[2]-hr1[2];l1=arch[1];l2=arch[2];
```

From this, we obtain the logarithmic embedding of the unit by typing

```
s=real(l1+l2)/4;v1=[l1,l2,conj(l1),conj(l2)]~/2-[s,s,s,s]~;
```

This 4-component vector contains by definition the logarithms of the four complex embeddings of the unit. Since the matrix `nf[5][1]` contains the values of the $r1+r2$ embeddings of the elements of the integral basis, we can obtain the representation of the unit on the integral basis by typing

```
m1=nf[5][1];m=matrix(4,4,j,k,if(j<=2,m1[j,k],conj(m1[j-2,k])));

v=exp(v1);au=gauss(m,v);

vu=round(real(au))
```

Then `vu` is the representation of the unit on the integral basis. The closeness of the approximation of `au` to `vu` gives us confidence that we have made no numerical mistake. To be sure that `vu` represents a unit, type `u=mod(nf[7]*vu,A)`, then typing `norm(u)` we see that it is equal to 1 so `u` is a unit.

There is of course no reason for `u` to be a fundamental unit. Let us see if it is a square. Type `f1=factor(subst(char(u,x),x,x^2))`. We see that the characteristic polynomial of `u` where `x` is replaced by `x^2` stays irreducible, hence `u` is not a square. However, type

```
f1=factor(subst(char(-u,x),x,x^2)).
```

The characteristic polynomial of `-u` factors into 2 polynomials of degree 4 (the second being the same as the first with x replaced by $-x$), showing that `-u` is the square of a unit, the characteristic polynomial of which is given as the component `f1[1,1]` (or `f1[2,1]` which is the characteristic polynomial of the opposite unit).

We now want to find the square root of `-u`. We can again use the `gauss` function as above. For this we need to take the square root of each element of the vector `-v`, and hence there are sign ambiguities. Let's do it anyway. Type `v=sqrt(-v)`. We see that `v[1]` and `v[3]` are conjugates, as well as `v[2]` and `v[4]`, so for the moment the signs seem OK. Now try `au=gauss(m,v)`. The numbers obtained are clearly not integers, hence the last remaining sign change must be performed. Type `v[1]=-v[1];v[3]=-v[3]` (they must stay conjugate) and then again `au=gauss(m,v)`. This time the components are close to integers, so we are done (after typing `vu=round(real(au))` as before).

Anyway, we find that a square root `u2` of `-u` is represented by the vector `vu=[-4,1,1,-1]~` on the integral basis, and this is in fact a fundamental unit. If we want it as an algebraic number, we simply type `u2=mod(nf[7]*vu,A)`.

The use of the function `polred2` gives us another method to find `u2` as follows. We type `q=polred2(f1[1,1])`. We recognize the polynomial `A` as the component `q[3,2]`. To obtain the square root of our unit we then simply type `up2=modreverse(mod(q[3,1],f1[1,1]))` (Exercise: why?). We find that `up2` is represented by the vector `[-3,-1,0,0]~` on the integral basis, which is not the result that we have found before nor its opposite. Where is the error ? (Please think about this before reading on. There is a mathematical subtlety hidden here.)

Have you solved the problem? Good for you! Yes, the problem occurs because as mentioned before (but you may not have noticed since it is not stressed in standard textbooks) although the number field K is not Galois over $\mathbf{Q}$, there does exist non-trivial automorphisms, and we have found it above by using the function `galoisconj`. Indeed, if we apply this automorphism to `up2` (by typing `mod(subst(lift(up2),x,aut),A)` where `aut` is the nontrivial component of `galoisconj(A)` computed above), we find the opposite of `u2`, which is OK.

Still another method which avoids all sign ambiguities and automorphism problems is as follows. Type `r=f1[1,1]%(x^2+u)` to find the remainder of the characteristic polynomial of `u2` divided by `x^2+u`. This will be a polynomial of degree 1 in `x` (with polymod coefficients) and we know that `u2`, being a root of both polynomials will be the root of `r`, hence can be obtained by typing `u2=−coeff(r,0)/coeff(r,1)`. Indeed, we immediately find the correct result with no trial and error.

Now ideals can be used in a wide variety of formats. We have already seen HNF-representations, ideles and prime ideals. We can also use algebraic numbers. For example type `al=mod(x^2−9,A)`, then `principalidele(nf,al)`. We obtain the idele corresponding to `al` (see the manual for the exact description). However it is usually not necessary to compute this explicitly since this is done automatically inside Pari. For example, you can type

```
for(j=1,4,print(j,":  ",idealval(nf,al,pd[j])))
```

We see that the valuation is non-zero (equal to 1) at the prime ideals `pd[1]` and `pd[2]`. In addition, typing `norm(al)` shows that `al` is of norm $49 = 7^2$. Let's check this differently. Type `pd12=idealmul(nf,pd[1],pd[2])` and then `idealmul(nf,al,idmat(4))`. We see that the results are the same, hence the product of the two prime ideals `pd[1]` and `pd[2]` is equal to the principal ideal generated by `al`. There is still something to be done with this examples as we shall see below with the function `isprincipal`. Essentially all functions that you would want on ideals are available:

`idealadd`, `idealdiv`, `idealfactor`, `idealintersect`, `idealinv`, `ideallllred`, `idealmul`, `idealmulred`, `idealpow`, `idealpowred`, `idealval`, `principalideal`, `principalidele`. We will also see `isprincipal` below (don't try it now because you need a much larger `initalg`).

We suggest you play with some of these to get the feel of the algebraic number theory package. Remember simply that when a matrix (usually in Hermite normal form) is output, it is always a $\mathbf{Z}$-basis of the result expressed on the *integral basis* `nf[7]` of the number field, *not* on the power basis, except of course if it is an integral basis.

Apart from the above functions you have at your disposal the very powerful function `buchgenfu` which is of the same type as `buchimag` and `buchreal` seen above, but for general number fields, and hence much slower. See Chapter 3 for a detailed explanation of its use.

Type for example `m=buchgenfu(A,0.2,0.2,10)`, where `A` is the same polynomial as before. After some work, we get a matrix with one column, which we extract by typing `v=m[,1];` Then v is a vector with 10 components. We immediately recognize the first component as being the polynomial `A` once again. The really new things start as the fifth component. We see that the class number $v[5][1]$ is equal to 4, that the class group $v[5][2]$ is cyclic, (and $v[5][3]$ gives us a generator in HNF), that the regulator $v[6]$ is equal to $3.794\ldots$. Then $v[7]$ which is close to 1 tells us that the results should be correct under a lot of assumptions (i.e. the class group or the regulator are not smaller). $v[8][1] = 6$ tells us that the roots of unity in the number field are exactly the sixth roots of 1 and $v[8][2]$ gives us a primitive sixth root of 1 as a polynomial in α, and finally the component

$v[9][1]$ gives us a fundamental unit as a polynomial in α. This unit must be linked to the unit `u2` found above since the unit rank is equal to 1.

To find the relation, let us use the logarithmic embeddings. Type `uf=mod(v[9][1],A)`, `uu=mod(v[8][2],A)` to get the generators of the unit group, then

```
cu2=log(conjvec(u2));cuf=log(conjvec(uf));cuu=log(conjvec(uu));
```

to get the (complex) logarithmic embeddings. Then type

```
lindep(real([cu2[1],cuf[1],cuu[1]]))
```

to find a linear dependence. Unfortunately, the result $[0, 0, 1]$ is to be expected since `uu` is a root of unity, hence the components of `cuu` are pure imaginary. Hence we must not take the real part. However, in that case the logs are defined only up to a multiple of $2i\pi$. Hence we type

```
lindep([cu2[1],cuf[1],cuu[1],2*i*pi])
```

The result $[1, 1, -1, 0]$ shows that `u2*uf` is probably equal to `uu`. Indeed, typing `u2*uf` results in `uu`.

Note that this works, but is not really a good way to obtain the result. A better way will be seen below using the function `isunit`.

Warning: since the fundamental unit obtained depends on the random seed, you may obtain a different unit from the one given here. To be sure to obtain the same result, type `setrand(1)` before the `buchgenfu` instruction.

You may ignore the last component of `v`.

You will notice that the manual explains that the units are given as polynomials in α. You may prefer them as vectors of coordinates on the integral basis. You can go back and forth between those two representations using the components `nf[7]` and `nf[8]` which were given by `initalg` above. For example, to obtain the representation on the integral basis of the primitive sixth root of unity $v[8][2]$, type `ru=nf[8]*vvector(4,j,coeff(v[8][2],j-1))`. Conversely, to come back to the initial representation type `nf[7]*ru`.

The `buchgenfu` program is in fact even more powerful. Instead of using `buchgenfu`, we are going to use a variant, which is going to give us even more information. Type

```
setrand(1);bnf=buchinitfu(A,0.2,0.2,10);.
```

This performs exactly the same computations as `buchgenfu`, but keeps much more information. In particular, you don't *want* to see the result, whence the semicolon.

To get most of the `buchgenfu` components, type `bnf[8]`. This gives you the class group and generators, the regulator, the check component and the unit group. If you don't want the unit group explicitly (it is implicitly contained in the rest of `bnf`), instead of using `buchinitfu` use `buchinit`. Similarly, if you use `buchgenfu` and don't want the units, use instead `buchgen`.

To initialize the number field, no need to use `initalg`: simply type `nf=bnf[7]`. Of course in our case this had already been done. We are now ready to perform more sophisticated operations in the class group. For example, let us take again our prime ideal `hp` above 7. Since we know that the class group is of order 4, we deduce that `hp` raised to the fourth power must be principal. Type `hp4=idealpow(nf,hp,4)` then `vis=isprincipal(bnf,hp4)`. The function `isprincipal` now uses all the information contained in `buchinit` and tells us that indeed `hp4` is principal. More precisely, the first component, $[0]$, gives us the position of the ideal in the class group. Here, $[0]$ means that it is up to equivalence equal to the 0-th power of the generator given by `buchinit` or `buchgen`, in other words that it is a principal ideal.

The second component gives us the algebraic number α such that `hp4=` $\alpha\mathbf{Z}_K$, where $\mathbf{Z}_K$ is the ring of integers of our number field, α being expressed on the integral basis nf[7]. To get α as an algebraic number, we thus type `alpha=mod(nf[7]*vis[2],A)`.

Let us check that the result is correct: first, type `norm(alpha)`. It is indeed equal to $7^4 = 2401$, which is the norm of hp4 (it could also have been equal to -2401). This is only a first check. The complete check is obtained by computing the HNF of the principal ideal generated by `alpha`. To do this, type `idealmul(nf,idmat(4),mat(vis[2]))`. Here, `idmat(4)` represents the ideal $\mathbf{Z}_K$ itself, and we must convert `vis[2]` into a one-column matrix using the function `mat`, otherwise `idealmul` will complain. The result that we obtain is identical to `hp4`, thus showing that `alpha` is correct (not that there was any doubt!). You may ignore the third component of `vis` which just tells us that the accuracy to which `alpha` was computed was largely sufficient.

But `isprincipal` also gives us information for nonprincipal ideals. For example, type

`vit=isprincipal(bnf,hp)`.

The component `vit[1]` is now equal to [3], and tells us that `hp` is ideal-equivalent to the cube of the generator `g` given by `buchinit` or `buchgen`. The second component `vit[2]` gives us α on the integral basis such that `hp=` αg^3.

Let us now take the example of the principal ideal `pd12` that we have seen above. We know that `pd12` is principal, but of couurse we have forgotten the generator that we had found, so we want one. For this, we type `pp=isprincipal(bnf,pd12)`. The first component of the result is `[0]`, telling us that the ideal is indeed principal, as we knew already. But the second component gives us the components of a generator of `pd12` on the integral basis. To get it on the power basis, type `al2=mod(nf[7]*pp[2],A)`. Now we remember suddenly that we already had a generator `al` for the same ideal. Hence type `u3=al2/al`. This must be a unit. To check this, computing its norm would not be sufficient, we need the complete characteristic polynomial, so we type `char(u3,x)`, and indeed it has integral coefficients with constant term 1 so `u3` is a unit. You can again find out what unit it is as we did above. However, as we mentioned, this is not really the best method.

To find the unit, we use explicitly the third component of the vector `bnf` given by `buchinit`. This contains an $(r+1)\times r$ complex matrix whose columns represent the complex logarithmic embedding of the fundamental units. Here $r = r_1 + r_2 - 1$ is the unit rank. We first compute the component of `u3` on the torsion-free part of the group of units by proceeding as follows. Type

`me=concat(bnf[3],[2,2] )`. Indeed, this is a variant of the regulator matrix and is much more practical to use since it is more symmetric and avoids suppressing one row arbitrarily. Now type

`cu3=principalidele(nf,u3)[2]`

to get the complex logarithmic embedding of `u3`. We could of course also have computed this logarithmic embeddings directly using `conjvec` as we did above, but then we must take care of the factors of 1 and 2 occurring.

Then type `xc=gauss(real(me),real(cu3))`

Whatever field we are in, if `u3` is a unit this *must* end with a 0 because of the "spurious" vector $[2,2]$, and the other components (here only one) give the exponents on the fundamental units. Here the only other component is the first, with a coefficient of -1, so we know that `u` is equal to `1/uf` multiplied by a root of unity.

To find this root of unity, we type `xd=cu3−me∗xc` then `xu=principalidele(nf,uu)[2]` and finally `xd[1]/xu[1]`. We find -4 as a result, and since `uu` is of order 6 this is the same as 2 so finally our unit `u3` must be equal to `uu^2/uf`, which is easily checked.

Although the above operations are easily performed, they are a little tedious, hence Pari has incorporated the above in a function called `isunit`. So instead of doing the above, you simply type `isunit(bnf,u3)` and the result $[-1, \mathrm{mod}\,(2,6)]$ tells us that `u3` is equal to `uf^(-1).uu^mod(2,6)`, i.e. to `uu^2/uf` as before.

6.7 Using Transcendental Functions

All the elementary transcendental functions and several higher transcendental functions (gamma function, incomplete gamma function, error function, exponential integral, K-bessel functions, confluent hypergeometric functions, Riemann zeta function, polylogarithms, Weber functions, theta functions) are provided. More could be written if the need arose.

In this type of functions, the default precision plays an essential role. In almost all cases transcendental functions work in the following way. If the argument is exact, the result will be computed using the current default precision. If the argument is not exact, the precision of the argument is used for the computation. A note of warning however: even in this case the *printed* value will be the current real format (usually the same as the default precision).

Let's assume that we have 28 decimals of default precision (this is what we get automatically at the start of a GP session). Type `e=exp(1)`. We get the number $e = 2.718\ldots$ to 28 decimals. Let us check how many correct decimals we really have. The hard (but reasonable) way is to proceed as follows. Change the precision to a substantially higher value, for example by typing `\precision=50`. Then type `e`, then `exp(1)` once again. This last value is the correct value of the mathematical constant e to 50 decimals, while the variable `e` shows the value that was computed to 28 decimals. Clearly they coincide to exactly 29 significant digits.

A simpler way to see how many decimals we have is to type `\x` after the initial `e=exp(1)`. The `02010005` which is the first codeword of the real number indicates that we are dealing with a real number having exactly $3 = 5 - 2$ mantissa words, and since $3\ln(2^32)/\ln(10 \approx 28.9$ we see that we have 28 or 29 significant digits.

Come back to 28 decimals (`\precision=28`). If we type `exp(1.)` you can check that we also obtain 28 decimals. However, type `f=exp(1+10.^(−30))`. Although the default precision is still 28, you can check using the one of the two methods above that we have in fact 59 significant digits! The reason is that `1+10.^(−30)` is computed according to the Pari philosophy, i.e. to the best possible precision. Since 10. has 29 significant digits and 1 has "infinite" precision, the number `1+10.^(−30)` will have $59 = 29 + 30$ significant digits, hence `f` also.

Now type `cos(10.^(−15))`. The result is printed as $1.0000\ldots$, but is of course not exactly equal to 1. Using `\x`, we see that the result has 7 mantissa words, giving us the possibility of having 67 correct significant digits. In fact (look in precision 100), only 60 are correct. Pari gives you as much as it can, and since 6 mantissa words would have given you only 57 digits, it uses 7. But why does it give so precise a result? Well, it is the same reason as before. When x is close to 1, $\cos(x)$ is close to $1 - x^2/2$, hence the precision is going to be approximately the same as this quantity, which will be $1 - 0.5 * 10^{-30}$ where $0.5 * 10^{-30}$ is considered with 28 significant digit accuracy, hence the result will have approximately $28 + 30 = 58$ significant digits.

Unfortunately, this philosophy cannot go too far. For example, when you type `cos(0)`, GP should give you exactly 1. Since it is reasonable for a program to assume that a transcendental function never gives you an exact result, GP gives you $1.000\ldots$ to one more mantissa word than the current precision.

OK, now let's see some more transcendental functions at work. Type `gamma(10)`. No problem (type `9!` to check). Type `gamma(100)`. The number now ends with a $*$, indicating that the printed integral part may be incorrect (in fact it is, type `99!` to get the correct answer). To get the correct value, there are two solutions. The first and most natural one is to increase the precision. Since `gamma(100)` has 157 decimal digits, type `\precision=170` (to be on the safe side), then `gamma(100)` once again. After some work, the printed result is this time perfectly correct.

However, this is probably not the best way to proceed. Come back first to the default precision (type `\precision=28`). The gamma function increasing very rapidly, one usually uses its logarithm. Type `lngamma(100)`. This time the result has a reasonable size, and is exactly equal to `log(99!)`.

Try `gamma(1/2+10*i)`. No problem, we have the complex gamma function. Now type

```
t=1000;z=gamma(1+i*t)*t^(-1/2)*exp(pi/2*t)/sqrt(2*pi),
```

then `norm(z)`. We see that `norm(z)` is very close to 1, in accordance with the complex Stirling formula.

Let's play now with the Riemann zeta function. First put the timer on (type `#`). Type `zeta(2)*6/pi^2`. This seems correct. Type `zeta(3)`. All this takes essentially no time at all. However, type `zeta(3.)`. Although the result is the same, you will notice that the time is substantially larger (if your machine is too fast to see the difference, increase the precision!). This is because Pari uses special formulas to compute `zeta(n)` when `n` is a (positive or negative) integer, even when `n` is odd and positive.

Type `zeta(1+i)`. This also works. Now for fun, let us compute in a very naïve way the first complex zero of `zeta`. We know that it is of the form `1/2+i*t` with between 14 and 15. Thus, we can use the following series of instructions. Type:

```
t1=1/2+14*i;t2=1/2+15*i;z1=zeta(t1);z2=zeta(t2);
if(norm(z2)<norm(z1),t3=t1;t1=t2;t2=t3;z1=z2,);
while(norm(z2)>10^-50,t2=(t1+t2)/2.;z2=zeta(t2);\
if(norm(z2)<norm(z1),t3=t1;t1=t2;t2=t3;z1=z2,);print(t1,":  ",z1))
```

We thus obtain the first zero to 25 significant digits.

As mentioned at the beginning of this tutorial, some transcendental functions can also be applied to p-adic numbers. This is as good a time as any to familiarize yourself with them. Type `a=exp(7+O(7^10))`, then `log(a)`. All seems in order. Now type `b=log(5+O(7^10))`, then `exp(b)`. Is something wrong, since we don't recover the number we started with? Absolutely not. Type `exp(b)*teich(5+O(7^10))`, and we indeed recover our initial number `5+O(7^10)`. The function `teich(x)` is the Teichmüller character, and is characterized by the fact that it is the unique $p-1$-st root of unity (here with $p = 7$) which is congruent to `x` modulo p, assuming that `x` is a p-adic unit.

Let us come back to real numbers for the moment. Type `agm(1,sqrt(2))`. This gives the arithmetic-geometric mean of 1 and $\sqrt{2}$, and is the basic method for computing (complete) elliptic integrals. In fact, type `pi/2/intnum(t=0,pi/2,1/sqrt(1+sin(t)^2))`,

and the result is the same. The elementary transformation `x=sin(t)` gives the mathematical equality

$$\int_0^1 \frac{dx}{\sqrt{1-x^4}} = \frac{\pi}{2\mathrm{agm}(1,\sqrt{2})} \ ,$$

which was one of Gauss's remarkable discoveries in his youth.

Now type `2*agm(1,i)/(1+i)`. As you see, the complex AGM also works, although one must be careful with its definition. The result found is almost identical to the previous one. Exercise: do you see why?

Finally, type `agm(1,1+7+O(7^10))`. As you see, p-adic AGM also work. Note however that since the square root of a p-adic number is not represented in Pari, only certain p-adic AGM can be computed. In addition, when $p = 2$, the congruence restriction is that `agm(a,b)` can be computed only when `a/b` is congruent to 1 modulo 16 (not 8 as could be expected).

Now type `?3`. This gives you the list of all transcendental functions. Instead of continuing with more examples, we suggest that you experiment yourself with the list of functions. In each case, try integer, real, complex and p-adic arguments. You will notice that some have not been implemented (or do not have a reasonable definition).

6.8 Using Numerical Tools

Although not written to be a numerical analysis package, Pari can nonetheless perform some numerical computations. We leave for a subsequent section linear algebra and polynomial computations. A few of the functions are even quite fun and surprising.

You of course know the formula $\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots)$ which is deduced from the power series expansion of `atan(x)`. You also know that π cannot be computed from this formula, since the convergence is so slow. Right? Wrong! Type `\precision=100` (just to make it more astonishing), then `4*sumalt(k=0,(-1)^k/(2*k+1))`. In a few seconds (admittedly more than simply typing `pi`), we get π to 100 significant digits (type `pi` to check). This is due to the combination of a method due to Euler for accelerating alternating sums, and a programming trick due to the Dutch mathematician van Wijngarten (see Numerical Recipes for an explanation).

Similarly, type `\precision=28` (otherwise the time will be too long) and `sumpos(k=1,1/k^2)`. Although once again the convergence is slow, a similar trick allows to compute the sum when the terms are positive (compare with the exact result `pi^2/6`. This is much less impressive because it is much slower, but is still very useful.

Even better, `sumalt` can be used to sum divergent series! Type

```
f(s)=sumalt(k=1,(-1)^(k-1)/k^s)/(1-2^(1-s))
```

(this example has already been given in Chapter 3).

Then for positive values of `s` different from 1, `f(s)` is equal to `zeta(s)` and the series cvonverges, albeit slowly (sumalt doesn't care however). For negative `s`, the series diverges, but `f(s)` still gives the correct result! Try `f(-1)`, `f(-2)`, `f(-1.5)`, and compare with the corresponding values of `zeta`. You should not push the game too far: `f(-2.5)` will probably not work.

Try `f(i)`, and compare with `zeta(i)`. Even (some) complex values work, even though the sum is not alternating any more!

Similarly, try `sumalt(n=1,(−1)^n/(n+i))`.

More traditional functions are the numerical integration functions. For example, type `intnum(t=1,2,1/t)` and presto! you get 26 decimals of `log(2)`. Look at Chapter 3 to see the different integration functions which are available.

With Pari, however, you can go further since complex types are allowed. For example, assume that we want to know the location of the zeros of the function $h(z) = e^z - z$. We use Cauchy's theorem, which tells us that the number of zeros in a disk of radius r centered around the origin is equal to

$$\frac{1}{2i\pi}\int_{C_r}\frac{h'(z)}{h(z)}\,dz\ ,$$

where C_r is the circle of radius r centered at the origin. Hence type

```
f(z,u)=u=exp(z);(u−1)/(u−z)
g(r)=intnum(t=0,2*pi,f(r*exp(i*t))*exp(i*t))*r/(2*pi)
```

The function `g(r)` will then count the number of zeros (we have simply made the change of variable `z=r*exp(i*t)`).

Now type `\precision=9` (otherwise the computation would take too long, and anyway we know that the result is an integer), then `g(1)`, `g(1.5)`. The results tells us that there are no zeros inside the unit disk, but that there are two (necessarily complex conjugate) whose modulus is between 1 and 1.5. For the sake of completeness, let us compute them. Let z be such a zero, and write $z = x + iy$ with x and y real. Then the equation $e^z - z = 0$ implies, after elementary transformations, that $e^{2x} = x^2 + y^2$ and that $e^x\cos(y) = x$. Hence $y = \pm\sqrt{e^{2x} - x^2}$ and hence $e^x\cos(\sqrt{e^{2x} - x^2}) = x$. Therefore, type

```
s(x,u)=u=exp(x);u*cos(sqrt(u*u−x*x))−x
```

Then `s(0)` is positive while `s(1)` is negative. Come back to precision 28 (`\precision=28`) and type

```
x0=solve(x=0,1,s(x))
```

This quickly gives us the value of `x`, and we can then type `z=x0+i*sqrt(exp(2*x0)−x0^2)`. This (together with its complex conjugate) is the required zero. As a check, type `exp(z)−z`, and also `abs(z)`.

Of course you can integrate over contours which are more complicated than circles, but you must perform yourself the changes of variable as we have done above to reduce the integral to a number of integrals on line segments.

The example above also shows the use of the `solve` function. To use `solve` on functions of a complex variable, it is necessary to reduce the problem to a real one. For example, to find the first complex zero of the Riemann zeta function as above, we could try typing

`solve(t=14,15,real(zeta(1/2+i*t)))`,

but this would not work because the real part is positive for `t=14` and `t=15`. As it happens, the imaginary part works. Type

`solve(t=14,15,imag(zeta(1/2+i*t)))`,

and this now works. We could also narrow the search interval and type for instance

```
solve(t=14,14.2,real(zeta(1/2+i*t)))
```

which would also work.

6.9 Working with Elliptic Curves

We have seen that the first thing to do to work in a number field is to use the function `initalg`. Well, similarly, the first thing to do to work with an elliptic curve is to use the function `initell`. So type `e=initell([6,−3,9,−16,−14])`. This computes a number of things about the elliptic curve defined by the affine equation

$$y^2 + 6xy + 9y = x^3 - 3x^2 - 16x - 14 \ .$$

We see that the discriminant e[12] is equal to 37, hence the conductor of the curve is 37. Of course in general it is not so trivial. In fact, the equation of the curve is clearly not minimal, so type `r=globalred(e)`. The first component r[1] tells us that the conductor is 37 as we already knew. The second component is a 4-component vector which will allow us to get the minimal equation: simply type `e=chell(e,r[2])` and the new `e` is now our minimal equation with corresponding data. You can for the moment ignore the third component r[3].

The new `e` tells us that the minimal equation is $y^2 + y = x^3 - x$. Let us now play a little with points on `e`. Type `q=[0,0]`, which is clearly on the curve (type `isoncurve(e,q)` to check). Well, `q` may be a torsion point. Type `hell(q)`, which computes the canonical Neron-Tate height of q. This is non-zero, hence `q` is not torsion. To see this even better, type

```
for(k=1,20,print(powell(e,q,k))),
```

and we see the characteristic parabolic explosion of the size of the points. As a further check, type `hell(e,powell(e,q,20))/hell(e,q)`. We indeed find $400 = 20^2$ as it should be. You can also type `orderell(e,q)` which returns 0, telling you that `q` is non-torsion.

Let's try another curve. Type `e=initell([0,−1,1,0,0])`. This corresponds to the equation $y^2 + y = x^3 - x^2$. Again from the discriminant e[12] we see that the conductor is equal to 11, and if you type `globalred(e)` you will see that the equation for `e` is minimal. Type `q=[0,0]` which is clearly a point on `e`, and `hell(e,q)`. This time we obtain a value which is very close to zero, hence `q` must be a torsion point. Indeed, typing `for(k=1,5,print(powell(e,q,k)))` we see that `q` is a point of order 5 (note that the point at infinity is represented as [0]). More simply, you can type `orderell(e,q)`.

Let's try still another curve. Type `e=initell([0,0,1,−7,6])` to get the curve $y^2 + y = x^3 - 7x + 6$. Typing `globalred(e)` shows that this is a minimal equation and that the conductor, equal to the discriminant, is 5077. There are some trivial integral points on this curve, but let's try to be more systematic.

First, let's study the torsion points. Typing `torsell(e)` shows that the torsion subgroup is trivial, so we won't have to worry about torsion points. Next, the component e[14] gives us the 3 roots of the minimal equation over $\mathbf{C}$, i.e. $Y^2 = X^3 - 7X + 25/4$ (set $(X,Y) = (x, y + 1/2)$) so if (x, y) is a real point on the curve, x must be at least equal to the smallest root, i.e. $x \geq -3$. Finally, if (x, y) is on the curve, its opposite is clearly $(x, -y - 1)$. So we are going to use the `ordell` function (*not* the orderell function, don't confuse the two) and type

```
v=[];for(x=−3,1000,s=ordell(e,x);if(length(s),v=concat(v,[[x,s[1]]]),));v
```

We thus get a large number (18) of integral points. Together with their opposites and the point at infinity, this makes a total of 37 integral points, which is large for a curve having such a small conductor. So we suspect (if we don't know already, since this curve is quite famous!) that the rank of this curve must be high. Let's try and put some order in this (note that we work only with the integral points, but in general rational points should also be considered).

Type `hv=hell(e,v)`. This gives the vector of canonical heights. Let us order the points according to their height. For this, type

`iv=indsort(hv)`, then `hv=extract(hv,iv)` and `v=extract(v,iv)`.

It seems reasonable to take the numbers with smallest height as generators of the Mordell-Weil group. Let's try the first 4: type

`m=mathell(e,extract(v,[1,2,3,4]));det(m)`

Since the curve has no torsion, The fact that the determinant is close to zero shows that the first four points are dependent. To find the dependency, it is enough to find the kernel of the matrix `m`. Since its determinant is not exactly equal to zero, it will be empty. We need an approximate kernel function, and this is the Pari function `kerr`. So type `kerr(m)`, and we indeed get a non-trivial kernel, and the coefficients are (close to) integers as they should. Typing `addell(e,v[1],v[3])` does indeed show that it is equal to `v[4]`.

Taking any other four points, we would in fact always find a dependency. Let's find them all. Type `vp=[v[1],v[2],v[3]]~;m=mathell(e,vp);det(m)`. This is now clearly non-zero so the first 3 points are linearly independent, showing that the rank of the curve is at least equal to 3 (it is in fact equal to 3, and is the curve of smallest conductor having rank 3). We would like to see whether the other points are dependent. For this, we use the function `bilhell`. Indeed, if `Q` is some point which is dependent on `v[1],v[2]` and `v[3]`, then `gauss(m,bilhell(e,vp,Q))` will by definition give the coefficients of the dependence relation. If these coefficients are close to integers, then there is a dependency, otherwise not. This is much safer than using the `kerr` function.

Thus, type `w=vector(18,k,gauss(m,bilhell(e,vp,v[k])))`. We "see" that the coefficients are all very close to integers, and we can prove it by typing `wr=round(w);sqrt(norml2(w-wr))`, which gives an upper bound on the maximum distance to an integer. Thus `wr` is the vector expressing all the components of `v` on its first 3. We are thus led to strongly believe that the curve has rank exactly 3, and this can be proved to be the case.

Let's now explore a few more elliptic curve related functions. Let us keep our curve `e` of rank 3, and type

`v1=[1,0];v2=[2,0];` then `z1=zell(e,v1)` and `z2=zell(e,v2)`.

We thus get the complex parametrization of the curve. To add the points `v1` and `v2`, we should of course type `addell(e,v1,v2)`, but we can also type `pointell(e,z1+z2)` which of course has the disadvantage of giving complex numbers, but illustrates how the group law on `e` is obtained from the addition law on **C**.

Type `f=x*series(anell(e,30),x)`. This gives a power series which, is the Fourier expansion of a modular form of weight 2 on $\Gamma_0(5077)$ (this has been proved directly, but also follows from Wiles's result since `e` is semi-stable). In fact, to find the modular parametrization of the curve, type `modul=taniyama(e)`, then `u=modul[1];v=modul[2];`. Type `(v^2+v)-(u^3-7*u+6)` to see that this parametrizes Now type `x*u'/(2*v+1)`, and we see that this is equal to the modular form `f` found above. The functions `u` and `v`, considered on the upper half plane (with $x = e^{2i\pi\tau}$) are in fact modular *functions* on $\Gamma_0(5077)$.

Finally, let us come back once more to the curve defined by typing `e=initell([0,−1,1,0,0])` where we had seen that the point `q=[0,0]` was of order 5. We know that the conductor of this curve is equal to 11 (type `globalred(e)`). We want the sign of the functional equation. Type `lseriesell(e,1,-11,1.1)`, then `lseriesell(e,1,-11,1)`. Since the values are clearly different, the sign cannot be − (look at Chapter 3 for the explanation of `lseriesell`). In fact there is an algebraic algorithm which would allow to compute this sign, but it has not yet been completely written, although in case of conductors prime to 6 it is very simple.

Now type `ls=lseriesell(e,1,11,1)`, and just as a check `lseriesell(e,1,11,1.1)`. The values agree (approximately) as they should, and give the value of the L-function of `e` at 1. Now according to the Birch and Swinnerton-Dyer conjecture (which is proved for this curve), `ls` is given by the following formula (in this case):

$$L(E,1) = \frac{\Omega \cdot c \cdot \mathrm{Sha}}{|E_{\mathrm{tors}}|^2} \quad ,$$

where Ω is the real period of E, c is the global Tamagawa number, produuct of the local c_p for primes p dividing the conductor, Sha is the order of the Tate-Shafarevitch group, and E_{tors} is the torsion group of E.

Now we know many of these quantities: Ω is equal to `e[15]` (if there had been 3 real roots instead of 1 in `e[14]`, Ω would be equal to `2*e[15]`). The Tamagawa number c is given as the last component of `globalred(e)`, and here is equal to 1. We already know that the torsion subgroup of E contains a point of order 5, and typing `torsell(e)` shows that it is of order exactly 5. Hence type `ls*25/e[15]`. We find 1.00000 . . . as a result, which shows that Sha must be equal to 1.

6.10 Functions Related to Polynomials and Power Series

First a word of warning to the unwary: it is essential to understand the crucial difference between exact and inexact objects (see Chapter 1). This is especially important in the case of polynomials. Let's immediately take a plunge into these problems. Type $\mathtt{gcd}(\mathtt{x}^2-1,\mathtt{x}^2-3*\mathtt{x}+2)$. No problem, the result is `x−1` as expected. But now type $\mathtt{gcd}(\mathtt{x}^2-1.,\mathtt{x}^2-3.*\mathtt{x}+2.)$. You are lucky, the result is almost correct except from a bizzare factor of 3 which come from the way Pari does the computation. In any cas, it is still essentially a reasonable result. But now type `gcd(x−pi,x^2−6*zeta(2))`. Although this should be equal to `x−pi`, Pari finds 1 as a result. This is because the notion of GCD of nonexact polynomials doesn't make much sense. However, type `resultant(x−pi,x^2−6*zeta(2))`. The result is extremely close to zero, showing that indeed the GCD is non-trivial, without telling us what it is. This being said, we will usually use polynomials (and power series) with exact coefficients in our examples.

Type `pol=cyclo(15)`. This gives the 15-th cyclotomic polynomial. Type `r=roots(pol)`. You have the 8 complex roots of pol given to 28 significant digits. To see them better, type `\b`. As you see, they are given as pairs of complex conjugate roots, in a random order. The only ordering done by the function `roots` concern the real roots, which are given first, and in increasing order.

The roots of `pol` are by definition the primitive 15-th roots of unity. To check this, simply type `rc=r^15`. Why, we get an error message! Well, fair enough, vectors cannot be multiplied (even less raised to a power) that easily. However, type `rc=r^15.` with a . at the end. Now it works, because powering to a non-integer (i.e. here real) power is a transcendental function and hence is applied termwise. Note that the fact that 15. is a real number which is represented exactly as an integer has nothing to do with the problem.

We see that the components of the result are very close to 1. It is however tedious to look at all these real and imaginary part. It would be impossible if we had many more. Let's do it automatically. Type `rr=round(real(rc))`, then `sqrt(norml2(rc-rr))`. We see that `rr` is indeed all 1's, and that the L2-norm of `rc-rr` is around 10¯27, reasonable enough when we work with 28 significant digits. Note that the function `norml2`, contrary to what it may seem, does not give the L2 norm but its square, hence we must take the square root (well, not really in this case!).

However, it is necessary to stress the fact that the `roots` function of Pari is *not* very robust, and may sometime give wrong answers. This may be corrected in some future version, but the user should be aware of this. It has been desigened essentially for speed. A slower and sometimes better version is `rootslong`, but even that sometimes fails. For example. type `pol=(x-10000)*(x-10001)`, then `roots(pol)`. The result is clearly wrong. If however we type `rootslong(pol)` the results are now correct. But now type `pol2=subst(pol,x,x*10000)/10^8`, then `rootslong(pol2)`. The result is wrong, even with `rootslong`. So be careful. Essentially, as is well known, the problem occurs when the polynomial has two roots which are very close. Still, `roots` is very useful.

Now type `pol=x^5+x^4+2*x^3-2*x^2-4*x-3`, then `factor(pol)`. The polynomial `pol` factors over $\mathbf{Q}$ (or $\mathbf{Z}$) as a product of two factors. Now type `f(p)=factorpadic(pol,p,10)`. This creates a function `f(p)` which factors `pol` over $\mathbf{Q}_p$ to p-adic precision 10. Type `factor(disc(pol))`. We see that the primes dividing the discriminant are 11, 23 and 37. Type `f(5)`, `f(11)`, `f(23)`, and `f(37)` to see different splittings.

Similarly, we can type `g(p)=lift(factmod(pol,p))`, and `g(2)`, `g(11)`, `g(23)` and `g(37)` which show the different factorizations, this time over $\mathbf{F}_p$.

In fact, even better: type `pol2=x^3+x^2+x-1`, then

```
centerlift(lift(factfq(x^3+x^2+x-1,3,t^3+t^2+t-1)))
```

.

This factors the polynomial `pol2` over the field $\mathbf{F}_3$ adjoined with a root of the polynomial `t^3+t^2+t-1`, which is a form of the field $\mathbf{F}_{27}$. We know that $\mathrm{Gal}(\mathbf{F}_{27}/\mathbf{F}_3)$ is cyclic of order 3 generated by the Frobenius homomorphism $u \mapsto u^3$, and the roots that we have found give the action of the powers of the Frobenius on `t` (if you do not know what I am talking about, please learn it with the help of explicit examples given by Pari).

Similarly, type `pol3=x^4-4*x^2+16` and `fn=factornf(pol3,t^2+1)`, and we get the factorization of the polynomial `pol3` over the number field defined by `t^2+1`, i.e. over $\mathbf{Q}(i)$. To see the result even better, type `lift(fn)`, remembering that `t` stands for the generator of the number field (here equal to $i = \sqrt{-1}$).

Note that it is possible, although ill advised, to use the same variable for the polynomial and the number field. You may for example type `fn2=factornf(pol3,x^2+1)`, and the result is correct. However, the Pari object thus created may give unreasonable results. For example, if you type `lift(fn2)` in the example above, you will get a strange object, with a symbol such as `x*x` typed. This is because Pari knows that the dummy variable `x` is not the same as the explicit variable `x`, but since it must print it when you lift, it has to do something.

To summarize, in addition to being able to factor integers, you can factor polynomials over $\mathbf{C}$ and $\mathbf{R}$ (this is the function `roots`), over $\mathbf{F}_p$ (the function `factmod`, over $\mathbf{F}_{p^k}$ (the function `factfq`), over $\mathbf{Q}_p$ (the function `factorpadic`), over $\mathbf{Q}$ or $\mathbf{Z}$ (the function `factor`), and over number fields (the function `factnf`). In the present version 1.38.1, it is *not* possible to factor over other rings, for example it cannot factor multivariate polynomials.

Functions related to factoring are `apprpadic`, `rootmod`, `rootpadic`, `sturm` and `sturmpart`. Play with them a little.

Now let's type `polsym(pol3,20)`, where `pol3` is the same polynomial as above. This gives the sum of the k-th powers of the roots of `pol3` up to $k = 20$, of course computed using Newton's formula and not using `roots`. You notice that every odd sum is zero (this is trivial since the polynomial is even), but also that the signs follow a regular pattern and that the (nonzero) absolute values are powers of 2. This is true: prove it, and more precisely find an explicit formula for the k-th symmetric power not involving (non-rational) algeberaic numbers.

You can also play a little with the function `karamul` (for Karatsuba multiplication). Put the timer on (`#`), and then set `pol4=(x+1)^200;` (do not forget the semicolon, you don't want to see the result, just the timing. Similarly for the next few commands. Then type `pol5=pol4*pol4;`, and look at the timing. Now type instead `pol6=karamul(pol4,pol4,5);`. This should be about twice as fast (it is implementation dependent so the gain may vary). Try with other values than 5. Also type `pol5-pol6` just to be sure that the result is correct. So if you have polynomials which are that large, it may be worthwhile to use the function `karamul`.

Now let's play a little with power series. We have already done so a little at the beginning. Type

```
8*x+prod(1+O(x^40),n=1,39,if(n%4,1-x^n,1))^8
```

You obtain a power series which has apparently only even powers of `x` appearing. This is surprising, but can be proved using the theory of modular forms. Note that we have initialized the product to `1+O(x^40)` and not simply to 1 otherwise the whole computation would have been done with polynomials, and this would first have been extremely long (try if you want, but don't forget to type `Ctrl-C` when you get tired of waiting) and also totally useless since the coefficients of `x^40` and above are irrelevant anyhow if we stop the product at `n=39`.

While we are on the subject of modular forms (which, together with taylor series expansions of usual functions is another great source of power series), type `\serieslength=122` then `d=x*eta(x)^24`. This gives the first 122 terms of the (Fourier) series expansion of the modular discriminant function Δ of Ramanujan, its coefficients being by definition the Ramanujan τ function which has a number of marvelous properties (look at any book on modular forms for explanations). We would like to see its properties modulo 2. Type `d%2`. Hmm, apparently Pari doesn't like to reduce coefficients of power series (or polynomials for that matter) directly. Can we do it without writing a little program? No problem. Type instead `lift(mod(1,2)*d)` and now this works like a charm.

The pattern in the result is clear. Of course, it now remains to prove it (see Antwerp III). Similarly, type `centerlift(mod(1,3)*d)`. This time the pattern is less clear, but nonetheless there is one. Refer to Anwerp III again.

6.11 Performing Linear Algebra

All the standard linear algebra programs available of course, and many more. In addition, linear algebra over $\mathbf{Z}$, i.e. work on lattices, can also be performed. Let us see how this works. First of all, a vector space (more generally a module) will be given by a generating set of vectors (often a basis) which will be representend as *column* vectors. This set of vectors will in turn be represented as a matrix: in Pari, we have chosen to consider matrices as row vectors of column vectors. The base field (or ring) can be any ring type Pari supports. However, certain operations are specifically written for a real or complex base field, while others are written for $\mathbf{Z}$ as the base ring.

Appendix A : Installation Guide for the UNIX Versions

This is the installation guide to the `Un*x` versions of `Pari/GP`. Starting with release 1.38, we assume that you have either an `ANSI C` or a `C++` compiler available.

If your machine does not have one (for example if you still use `/bin/cc` in `SunOS 4.1.x`), we strongly suggest that you obtain the `gcc/g++` compiler from the Free Software Foundation or by anonymous ftp.

1) To compile the library and the gp calculator.

a) First, create a `Makefile` which is as close as possible to your system by typing `Makemakefile arch`, where `arch` can be the following.

`sun3` if your machine is 68020/68030/68040/68060 based. In that case, you will also have to choose which assembly file to use (see b) below).

`hppa` if your machine has the HP-PA architecture.

`sparcv7` if your machine is based on Sparc version 7 (example: Sparcstations 1, 1+, 2, IPC, IPX).

`sparcv8` if your machine is based on Sparc version 8 (example: Sparcstations 10, Classic, LX). In this case you will have to choose which assembly file to use (see b) below).

`i386` if your machine is based on a 386+387, 486 or Pentium and you use a Un*x OS (it has been successfully tested with linux).

`dos` if your machine is based on a 386+387, 486 or Pentium and you are still in the DOS world. This requires the emx/gcc compiler. Note also that it is incompatible with the Windows environment.

If you do not give any argument to `Makemakefile`, the Makefile which is generated is for other Un*x machines, for example DECstations. A few modifications to the Makefile may be necessary in this case.

Note: starting with the present release, we hope to be able to support 64-bit machines. The present version has been tested on a DEC Alpha, and still has specific bugs, probably due to the compiler (whether gcc or cc) but is usable.

b) In the case of 680x0 or sparcv8 systems you must choose which assembly file to use:

`680x0` systems:

The syntax used by the SUN 3 assembler is not standard. On the MacII distribution, the correct Mac assembler syntax is given. In the present distribution, in addition to `mp.s` which has the SUN 3 syntax, two files called mp.news and mp.ami are included so as to help people having machines with a 680x0 processor (x>= 2) but a more standard syntax.

This may not correspond to the actual syntax used, but may be closer than mp.s. In principle, apart from whitespace and the different syntax, the semantics of the two files should be identical. In case of conflict, correct mp.news or mp.ami (i.e. NEVER touch mp.s).

The file mp.news has been successfully tested on a Sony NEWS, and the file mp.ami on a Commodore Amiga 2500 running Lattice C 5.10.

`sparcv8` systems:

The different implementations of the Sparc Version 8 architecture (at present mainly Microsparc and Supersparc) need slightly different assembly language modules for optimum speed, although they are compatible. If you have a Supersparc (e.g. SparcStation 10), then make a symbolic link from `sparcv8super.s` to `sparcv8.s`. If you have a Microsparc (e.g. Sparc Classic and LX), make a symbolic link from `sparcv8micro.s` to `sparc.s`. By default, the link in the distrinution is from `sparcv8super.s`.

c) Decide if you want to compile with an `ANSI C` or a `C++` compiler. The C++ version of Pari is always a little faster because of inlining, but can be used in library mode only with C++ programs. Hence you may want, for example, to install gp compiled in C++, but the ANSI C library if you want to link it with ANSI C programs. The Makefile is generated by default for ANSI C. If you want it for C++, you must perform the following changes in the `Makefile`: replace `gcc` by `g++` (if this is your C++ compiler), remove the `-ansi` flag, and if you use a non-68k version, replace

```
$(CC) $(CFLAGS) -c mp.c
```

by

```
$(CC) $(CFLAGS) -c -o mp.o mpin.c
```

c) The Makefile assumes by default that you are using `X11` as a windowing system. If this is not the case, modify the Makefile accordingly: you are given the choice of `X11` (the default), `sunview` or no plotting. For this, comment/uncomment the lines corresponding to `PLOTFILE, PLOTCFLAGS` and `PLOTLIBS`. Some slight modifications may also have to be made so that the compiler knows where to access the X11 libraries if they are in some non-standard place.

d) If you have the GNU `readline` library installed (distributed with `gdb`), and want to use its facilities (this is not so useful if, as we advise, you work in an Emacs buffer, see below), comment/uncomment the appropriate lines involving `GPMAIN` in the Makefile. Note: if you are compiling with sunview, you will have an error message about a redefined function `rl_copy`. Since the sun source code is not available, the way out is to recompile the GNU readline library by changing in the file `readline.c` `rl_copy` to some weird name, say `rl_copy_kludge`. The use of the readline and history library (suggested to us by E. Roeder) is not documented but is similar to emacs commands. However note that it is incompatible with SUN commandtools (but not with shelltools).

e) Some special-purpose application programs may be available in Pari and gp. By default in the distribution they are available under gp, but are not linked in with the other library modules during the creation of the file libpari.a. In the present version 1.38.1, these are the functions `buchimag(), buchreal(), buchgen()` and `isprincipal()`, all four in the file `buch.c`. If you want to have the smallest possible gp executable, you will have to do two things. First suppress all occurrences of `buch.c` and `buch.o` in the `Makefile`. Second, in the files `anal.c`, `helpmessages.c` and `gencom.h`, suppress the lines corresponding to `buchimag(), buchreal(), buchgen()` and `isprincipal()`. If on the contrary you want these functions to be available in library mode as well as under gp, in the Makefile add `buch.o` to the files which are listed during the creation of libpari.a.

f) Then simply type `make` in the distribution directory. Be sure to `make clean` before changing to another architecture using the same file system. Note that the 68020 version is especially fast compared to the processor speed because a large part of the kernel is written in assembly language. Note also that the Sparcv7 version runs on Sparcv8 machines, but slower than the specific Sparcv8 versions. On the contrary the Sparcv8 version cannot run on Sparcv7 machines.

g) To test the executable, run gp on the file `testin` as follows:

```
gp < testin >& fileout &
```

Then do a `diff` with the file `testout`. Apart from the header (version number and type) and the very last line which gives the user elapsed time in milliseconds, any difference probably means that something is wrong. Most probably with your installation procedure, but it may be a bug in the Pari system, in which case we would appreciate a report. Note that `testin` is not a severe test and is quite fast (a few minutes), but does check at least one instance of every function. Do NOT forget the & after >, since `testin` tests one special error message.

h) If you want to test the graphic routines, use `gp<testplotin`. You will have to click 6 times on the mouse button after seeing each image (under X11; under suntools you must kill the images). The `testin` script produces a Postscript file `pari.ps` which you can send to a Postscript printer. The printed output should bear some similarity to the screen images (:-).

2) To install the PARI library so that it can be easily used from a user program, type `make install`. This puts the executable gp in `/usr/local/bin`, the library in the directory `/usr/local/lib`, and the necessary include files in `/usr/local/include/pari`, which is created if it does not already exist. If these directories do not suit your installation, change the `LIBDIR` or the `INCLUDEDIR` in the Makefile. You can install separately the library and gp by using the targets `install-lib` and `install-bin`.

3) Once installed, to link to the PARI library just add `-lpari -lm` in your link command and `-I/usr/local/include/pari` to your compilation commands. A sample makefile (`Makesimple`) is given for gp itself. All modifications made to the Makefile should of course be made on the Makesimple file.

4) If you want to use gp under GNU Emacs (see section 3.11 of the users' manual) change the pathnames in the file `pari.el` to suit your installation, and read the file `pari.txt`. If you are familiar with Emacs, we strongly suggest that you do so.

5) If you want to get rid of your `.o` files and the created binaries in the working directory, type `make clean`.

6) For the example of section 4.3 of the user's manual, type `make` in the directory `examples`. Several complete sample gp programs are also given in that directory, for example (Shanks's SQUFOF factoring method, the Pollard rho factoring method, the Lucas-Lehmer primality test for Mersenne numbers, a simple general class group and fundamental unit algorithm, much worse than buchgen, and others). See the file Newin1.37 for some explanations.

7) To print the users' manual, go in the `doc` directory and type `make`. This will automatically create a file `users.dvi` containing the manual with a table of contents and an index in two passes. You must then send the `users.dvi` file to your favorite printer in the usual way. A tutorial for

GP is being written. If you want the available part of the tutorial to be included in the manual, uncomment the line `\input tutorial` in the file `users.tex`.

8) Send bugs, comments, etc... to:

pari@ceremab.u-bordeaux.fr

In any case, we would like to start a Pari mailing list, so if you get this software, even if you are an old-timer of Pari, we would appreciate if you could send us an e-mail message giving us some information about yourself. Put as header of your message "mailing list", so we can recognize it easily. In exchange, we will try to keep you informed on the progress of the system.

11) Good luck and enjoy!

Appendix B : A sample 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, please change them accordingly in the Makefiles.

If the program example that we have given is in the file mattrans.c (say as the first of several matrix transcendental functions), then a sample Makefile is the following:

```
CC = cc
CFLAGS = -O -I/usr/include/pari-include

all:    mattrans

mattrans:    mattrans.c
	$(CC) $(CFLAGS) -o mattrans mattrans.c -lpari -lm
```

Appendix C : A complete program

We give here the listing of the program seen in detail in section 4.3, with the slight modifications explained at the end of that section.

```
#include <genpari.h>

GEN matexp();

main()
{
  GEN x,y;
  long prec,d;
  char s[512];

  init(1000000,2); /* take a million bytes of memory for the stack */
  printf("precision of the computation in decimal digits?  ");
  scanf("%d",&d);if(d<0) prec=3;else prec=d*K1+3;
  printf("input your matrix in GP format:\n");
  s[0]=0;while(!s[0]) gets(s);x=lisexpr(s);
  y=matexp(x,prec);
  sor(y,'g',d,0);
}

GEN matexp(x,prec)
    GEN x;
    long prec;

{
  GEN y,r,s,p1,p2;
  long tx=typ(x),lx=lg(x),i,k,n,lbot,ltop;

/* check that x is a square matrix */

  if(tx!=19) {printf("This expression is not a matrix\n");return(gzero);}
  if(lx!=lg(x[1])) {printf("This matrix is not square\n");return(gzero);}

/* convert x to real or complex of real and compute its L2 norm */

  ltop=avma;s=gzero;
  r=cgetr(prec+1);gaffsg(1,r);p1=gmul(r,x);
  for(i=1;i<lx;i++) s=gadd(s,gnorml2(p1[i]));
  if(typ(s)==2) setlg(s,3);
  s=gsqrt(s,3); /* we do not need much precision on s */

/* if s<1, we are happy */
```

```
  if(expo(s)<0) n=0;
  else {n=expo(s)+1;p1=gmul2n(p1,-n);setexpo(s,-1);}

/* initialisations before the loop */

  y=gscalmat(r,lx-1); /* this creates the scalar matrix r*I_{lx-1} */
  p2=p1;r=s;k=1;
  lbot=avma;y=gadd(y,p2);

/* now the main loop */

  while(expo(r) >= -32*(prec-1))
  {
    k++;p2=gdivgs(gmul(p2,p1),k);r=gdivgs(gmul(s,r),k);
    lbot=avma;y=gadd(y,p2);
  }

/* now square back n times if necessary */

  for(i=0;i<n;i++) {lbot=avma;y=gmul(y,y);}
  y=gerepile(ltop,lbot,y);
  return(y);
}
```

In this appendix we give the list of predefined constants available in the PARI library.

gzero (zero) see 4.1.1.
gun (un) see 4.1.1.
gdeux (deux) see 4.1.1.
ghalf (lhalf) see 4.1.1.
gi see 4.1.1.
polun[] (lpolun[]) see 4.1.1.
polx[] (lpolx[]) see 4.1.1.

geuler. This is Euler's constant, and is in the heap, *not* in the PARI stack. It is not initialized, and if you want to use it you must call **consteuler**(prec) (see 3.3.19.).

gpi. This is the number pi, and is in the heap, *not* in the PARI stack. It is not initialized, and if you want to use it you must call **constpi**(prec) (see 3.3.33.).

bern(i). This is the $2i$-th Bernoulli number ($B_0 = 1$, $B_2 = 1/6$, $B_4 = -1/30$, etc...) The Bernoulli numbers are in the heap and *not* in the PARI stack, and are not initialized. To initialize them you must use the function **mpbern** which has the following syntax:

```
void mpbern(long n, long prec);
```

The effect of this function is to create the even numbered bernoulli numbers up to B_{2n-2} **as real numbers** of precision prec. They can then be used with the macro **bern**(i). Note that this is not a function but simply an abbreviation, hence care must be taken that i is inside the right bounds (i.e. $0 \leq i \leq n-1$) before using it, since no checking is done in PARI itself.

Finally, one has access to a table of (differences of) primes through the pointer **diffptr**. This is used as follows: after `init` has been called, this table is initialized with the differences of primes up to 500000 (default which can trivially be changed by calling `init` with different arguments, see 4.1.1). Then one declares `byteptr d=diffptr;`, where d is the name of the pointer that one uses. This will point to the first difference in the table, i.e. 2. To get to the next difference, just do `d++`.

In addition, some single or double-precision real numbers are predefined, and their list is in the file `gencom.h`

Index

D

E

F

G

H

I

J

K

L

Q

R

U

V

W

X

Z