Q Quadratic: PARI guide

(version 0.1)

A PARI/GP package for integral binary quadratic forms (and coming soon: quaternion algebras) over $\mathbb{Q}$, with an emphasis on indefinite quadratic forms and indefinite quaternion algebras.

James Rickards
Department of Mathematics and Statistics,
McGill University, Montreal
Personal homepage
Github repository

Contents

1 Introduction

The roots for this library came from my thesis project, which involved studying intersection numbers of geodesics on modular and Shimura curves. To be able to do explicit computations, I wrote many GP scripts to deal with indefinite binary quadratic forms, and indefinite quaternion algebras. This library is a revised version of those scripts, rewritten in PARI ([The20]) for optimal efficiency.

While there already exist some PARI/GP methods to compute with quadratic forms and quaternion algebras (either installed or available online), I believe that this is the most comprehensive set of methods yet.

The package has been designed to be easily usable with GP, with more specific and powerful methods available to PARI users. More specifically, the GP functions are all given wrappers so as to not break, and the PARI methods often allow passing in of precomputed data like the discrimiant, the reduced orbit of an indefinite quadratic form, etc. If you only intend on using this library in GP, please consult the GP manual instead.

Note that the current version (0.1) only includes algorithms for quadratic forms; the quaternionic algorithms will be in the next update, which will hopefully be ready by October 2020.

1.1 Overview of the main available methods

For integral binary quadratic forms, there are methods available to:

- Generate lists of (fundamental, coprime to a given integer n) discriminants;
- Compute the basic properties, e.g. the automorph, discriminant, reduction, and equivalence of forms;
- For indefinite forms, compute all reduced forms, the Conway river, left and right neighbours of river/reduced forms;
- Compute the narrow class group and a set of generators, as well as a reduced form for each equvalence class in the group;
- Output all integral solutions (x, y) to $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = n$ for any integers A, B, C, D, E, F, n;
- Solve the simultaneous equations $Ax^2+By^2+Cz^2+Dxy+Exz+Fyz = n_1$ and $Ux+Vy+Wz = n_2$ for any integers $A, B, C, D, E, F, U, V, W, n_1, n_2$.

1.2 Upcoming methods

The next project is to implement methods relating to quaternion algebras over $\mathbb{Q}$. Planned methods include:

- Initialize the algebra given the ramification, and initialize maximal/Eichler orders (with specific care given to algebras ramified at $<= 2$ finite places);

- Compute the fundamental domain of unit groups of Eichler orders in indefinite algebras (Shimura curves);
- Solve the principal ideal problem in indefinite quaternion algebras;
- Compute all optimal embeddings of a quadratic order into a quaternion algebra, and arrange them with respect to the class group action and their orientation.

I will also be adding methods to compute intersections of indefinite binary quadratic forms and closed geodesics on Shimura curves, as this was one of the goals of my thesis project.

1.3 How to use the library

As a first word of warning, this library is only guaranteed to work on Linux. The essential files (.so) were created with GP2C, and they are not usable with Windows (I don't think it works on Mac, but I don't know). However, the workaround for Windows is to install the Linux Subsystem for Windows, and install PARI/GP there (in fact, this is my current setup, and it works well).

If you plan on modifying the library, then you just need **qquadraticdecl.h**, **c_base.c**, and **c_bqf.c**; I will assume that you know what to do from there.

Otherwise, if you plan on only using and not modifying the library, then you need the files **qquadraticdecl.h**, **libqquadratic.so**, and **qquadratic.gp**. When writing your program include the .h declaration file, and if you load the program in GP, make sure that qquadratic.gp has also been loaded (or at least one function from it has). If you do not do this, your function will fail to find the reference to my function and the program will exit! I think there should be a way to not have this happen, but I am not a C programming expert, so this will do for now.

1.4 Validation of methods

Unless otherwise noted, non-static methods are stack clean (the main exceptions come from lists). I have made a good effort to ensure that this claim is true, and to validate that my methods do exactly what is claimed.

To test this, I have written a series of validation methods. For each function, I generate some "random data," enter it in the function, and test that the result is stack clean and that basic properties are obeyed (and then repeat this a few thousand times). For example, I can compute the narrow class group for a given D, check that the generators are all of the right order (by powering them and testing for equivalence with the identity), generating the whole group by composing the generators together, and checking that the group elements are all non-equivalent. Of course this isn't "proof" that I have no errors lurking in obscure parts of the algorithm, but it does provide good support.

If you do happen to find a bug, then please let me know! At the moment I have not posted my validation methods, but if for some reason you would like them, then I can send you the files directly.

1.5 Programming style

I have gone for optimal efficiency by using the most specific methods (e.g. addii, passing in precomputed data, etc.) whenever possible. What this means is that it is very easy to break the library if you feed in bad inputs! However, when working with GP we really do not want this to happen, so every method available to GP has a wrapper for protection against segmentation breaks (if it is required). This wrapper is always indicated by adding the suffix "_typecheck" to the method, and it's function is basically to check that the inputs are kosher and feed them on to the appropriate method. There are some methods that do not require this wrapper, and they do not come with it.

If there is a piece of "standard data" that is very useful in a method, there is typically a method that does not require passing in of this data, and another that does. This allows the user to maximize efficiency by not recomputing this standard data repeatedly. For example, if you are doing a lot of computations with a fixed quadratic form q, then you can store the discriminant of q and pass it along with q when this is available. If you are more concerned with getting it correct and not breaking your programs, then you can just use the "_typecheck" methods, as they require no precomputed data and are hard to break. Once you have a working program, you can revisit this to optimize for efficiency.

1.6 How to use this manual

Sections 2-3 contain detailed descriptions of every function: the input, output, and what the function does. The sections are labeled by source files, and are divided into subsections of "similar" methods. If you are seeking a function for a certain task, have a look through here.

Section 4 contains simply the method declarations, and is useful as a quick reference. Clicking the name of a method in this section will take you to its full description in Sections 2-3, and clicking on the name there will take you back to Section 4.

Methods accessible to GP are given a green background, static methods are given a blue background, and precise non-GP accessible and non-static methods are in yellow. The methods are generally alphabetized, with static methods appearing at the end of sections. Any non-static method that is not stack clean is given a red background, and will be appropriately noted in the description. Start by looking at the green methods, and when you want to use one check the surrounding yellow methods for the precise version you want. Unless you are modifying these methods directly, you can ignore the blue methods.

2 c_base

This is a collection of "basic" functions and structures, which are useful in various places. The main interesting method here is "sqmod", which allows you to compute square roots modulo any integer n, and not just primes (which is already built into PARI/GP).

2.1 Euclidean geometry

These methods will likely be moved the geometry package, when I write that (the geometry package will support finding the fundamental domain for a discrete subgroup of $\mathrm{PSL}(2, \mathbb{R})$).

Name:	`GEN crossratio`
Input:	`GEN a, GEN b, GEN c, GEN d`
Input format:	`a`, `b`, `c`, `d` complex numbers or infinity, with at most one being infinity
Output format:	Complex number or ±`oo`
Description:	Returns the crossratio `[a,b;c,d]`.

Name:	`GEN mat_eval`
Input:	`GEN M, GEN x`
Input format:	`M` a 2x2 matrix and `x` a complex number or infinity
Output format:	Complex number or ±`oo`
Description:	Returns `M` acting on `x` via Mobius transformation.

Name:	`GEN mat_eval_typecheck`
Input:	`GEN M, GEN x`
Input format:	`M` a 2x2 matrix and `x` a complex number or infinity
Output format:	Complex number or ±`oo`
Description:	Checks that `M` is a 2x2 matrix, and returns `mat_eval(M, x)`.

2.2 Infinity

In dealing with the completed complex upper half plane, the projective line over $\mathbb{Q}$, etc., we would like to work with ∞, but currently PARI does not support adding/dividing infinities by finite numbers. The functions here are wrappers around addition and division to allow for this.

Name:	`GEN addoo`
Input:	`GEN a, GEN b`
Input format:	`a, b` complex numbers or infinity
Output format:	Complex number or ±`oo`
Description:	Returns `a+b`, where the output is `a` if `a` is infinite, `b` if `b` is infinite, and `a+b` otherwise.

Name:	`GEN divoo`
Input:	`GEN a, GEN b`
Input format:	`a, b` complex numbers or infinity
Output format:	Complex number or ±`oo`
Description:	Returns `a/b`, where `a/0` will return $\pm\infty$ (depending on the sign of `a`), and $\pm\infty$/`b` will return $\pm\infty$ (depending on the sign of `b`). Note that both `0/0` and ∞/∞ return ∞.

2.3 Linear equations and matrices

`lin_intsolve` is essentially just `gbezout`, but it outputs to a format that is useful to me.

Name:	`GEN lin_intsolve`
Input:	`GEN A, GEN B, GEN n`
Input format:	Integers `A, B, C`
Output format:	`gen_0` or $[[m_x, m_y], [x_0, y_0]]$.
Description:	Solves $Ax+By = n$ using `gbezout`, where the general solution is $x = x_0+m_xt$ and $y = y_0 + m_xt$ for $t \in \mathbb{Z}$.

Name:	`GEN lin_intsolve_typecheck`
Input:	`GEN A, GEN B, GEN n`
Input format:	Integers `A, B, C`
Output format:	`gen_0` or $[[m_x, m_y], [x_0, y_0]]$.
Description:	Checks that `A, B, n` are integral, and returns `lin_intsolve(A, B, n)`.

Name:	`GEN mat3_complete`
Input:	`GEN A, GEN B, GEN C`
Input format:	Integers `A, B, C` with $\gcd(A, B, C) = 1$
Output format:	Matrix
Description:	Returns a `3x3` integer matrix with determinant 1 and first row `A, B, C`.

Name:	`GEN mat3_complete_typecheck`
Input:	`GEN A, GEN B, GEN C`
Input format:	Integers `A, B, C` with $\gcd(A, B, C) = 1$
Output format:	Matrix
Description:	Checks that `A, B, C` are relatively prime integers, and returns `mat3_complete(A, B, C)`.

2.4 Square roots modulo n

In PARI/GP you can take square roots modulo p^e very easily, but there is not support for a general modulus n, and if the number you are square rooting is not a square, an error will occur. `sqmod` is designed to solve this problem, and uses the built in methods of `Zp_sqrt` and `chinese` to build the general solution.

Name:	`GEN sqmod`
Input:	`GEN x, GEN n, GEN fact`
Input format:	`x` a rational number with denominator coprime to `n`, a positive integer, and `fact` the factorization of `n`, which can be passed in as `gen_0` if not precomputed.
Output format:	`gen_0` or `v=[S, m]`.
Description:	Returns the full solution set to $y^2 \equiv x \pmod{n}$, where the solution set is described as $y \equiv s_i \pmod{m}$ for any $s_i \in S$.

Name:	`GEN sqmod_typecheck`
Input:	`GEN x, GEN n`
Input format:	`x` a rational number, `n` a non-zero integer.
Output format:	`gen_0` or `v=[S, m]`.
Description:	Checks that `x` is rational and `n` is a non-zero integer, replaces it by `-n` if negative, and returns `sqmod(x, n, gen_0)`.

Name:	`GEN sqmod_ppower`
Input:	`GEN x, GEN p, long n, GEN p2n, int iscoprime`
Input format:	`x` integer, `p` prime, `n` non-negative integer, `p2n`$=p^n$, `iscoprime=0, 1`
Output format:	`gen_0` or `v=[S, m]`.
Description:	Returns the full solution set to $y^2 \equiv x \pmod{p^n}$, where the solution set is described as $y \equiv s_i \pmod{m}$ for any $s_i \in S$ (m is necessarily a power of p dividing p^n). If `iscoprime=1`, the `x, p` are guaranteed to be coprime; otherwise this assumption is not made.

2.5 Integer vectors

The following methods are typically available for `ZC`, but not for `ZV`.

Name:	`GEN ZV_copy`
Input:	`GEN v`
Input format:	`v` a vector with integer entries
Output format:	Vector
Description:	Returns a copy of the integral vector `v`.

Name:	`int ZV_equal`
Input:	`GEN v1, GEN v2`
Input format:	`v1, v2` vectors with integer entries
Output format:	`0` or `1`
Description:	Returns `1` if `v1=v2` and `0` else.

Name:	`GEN ZV_Z_divexact`
Input:	`GEN v, GEN y`
Input format:	`v` an integral vector, `y` a non-zero integer which divides all components of `v`
Output format:	Vector
Description:	Returns `v/y`.

Name:	`GEN ZV_Z_mul`
Input:	`GEN v, GEN x`
Input format:	`v` an integral vector, `x` an integer
Output format:	Vector
Description:	Returns `vx`.

2.6 Time

Methods for returning and printing time. Uses the C library time.h, as this is significantly less work than getwalltime().

Name:	`char* returntime`
Input:	–
Input format:	-
Output format:	-
Description:	Returns the current time as a string.

Name:	`void printtime`
Input:	–
Input format:	-
Output format:	-
Description:	Prints the current time.

2.7 Lists

There are three dynamically linked lists implemented: `clist, glist, llist`. A circular list (`clist`) `C` stores a `GEN` (accessible by `C->data`), a pointer to the next element (`C->next`), and the previous element (`C->prev`). A generic list is the same, except is does not store a pointer to the previous element. A long list is the same as a generic list, except it stores the data type `long` instead of `GEN` (and hence does not enter into the PARI stack).

Lists should be initialized by calling `(c/g/l)list *L=NULL;`, and use the following methods to work with them. If you do not call the `togvec` or `tovecsmall` methods, you should free the list using the appropriate `(c/g/l)list_free` methods.

Name:	`void clist_free`
Input:	`clist *l, long length`
Input format:	Pointer to a `clist` l, length of the list `length`
Output format:	-
Description:	Frees the `clist` (each data point had been initialized with `pari_malloc`). If `l` is shorter than `length`, an error will occur, and if it is longer than the remaining part of the list has not been freed.

Name:	`void clist_putbefore`
Input:	`clist **head_ref, GEN new_data`
Input format:	-
Output format:	-
Description:	Adds an element containing `new_data` before the element pointed to by `head_ref`, and updates the list start position.

Name:	`void clist_putafter`
Input:	`clist **head_ref, GEN new_data`
Input format:	-
Output format:	-
Description:	Adds an element containing `new_data` after the element pointed to by `head_ref`, and updates the list start position.

Name:	`GEN clist_togvec`
Input:	`clist *l, long length, int dir`
Input format:	`length` the length of `l`, and `dir=-1` or `1`.
Output format:	Vector
Description:	Returns a vector consisting of the list from `l` and onward of length `length`, and frees the list. If `dir=1` we go through the list via `l->next`, and if `dir=-1` we go through via `l->prev`.

Name:	`void glist_free`
Input:	`glist *l`
Input format:	Pointer to a `glist` l
Output format:	-
Description:	Frees the `glist` (each data point had been initialized with `pari_malloc`).

Name:	`GEN glist_pop`
Input:	`glist **head_ref`
Input format:	-
Output format:	-
Description:	Removes the last element of the list, frees this list element, and returns the data entry. This does NOT copy the return element, so it is NOT stack safe.

Name:	`void glist_putstart`
Input:	`glist **head_ref, GEN new_data`
Input format:	-
Output format:	-
Description:	Adds an element containing `new_data` before the element pointed to by `head_ref`, and updates the list start position.

Name:	`GEN glist_togvec`
Input:	`glist *l, long length, int dir`
Input format:	`length` the length of `l`, and `dir=-1` or `1`.
Output format:	Vector
Description:	Returns a vector consisting of the list from `l` and onward of length `length`, and frees the list. If `dir=1` we fill in the return vector from index `1` to `length`, and if `dir=-1` we go in the opposite direction.

Name:	`void llist_free`
Input:	`llist *l`
Input format:	Pointer to a `clist` l
Output format:	-
Description:	Frees the `llist` (each data point had been initialized with `pari_malloc`). If `l` is shorter than `length`, an error will occur, and if it is longer than the remaining part of the list has not been freed.

Name:	`long llist_pop`
Input:	`llist **head_ref`
Input format:	-
Output format:	-
Description:	Removes the last element of the list, frees this list element, and returns the data entry.

Name:	`void llist_putstart`
Input:	`llist **head_ref, GEN new_data`
Input format:	-
Output format:	-
Description:	Adds an element containing `new_data` before the element pointed to by `head_ref`, and updates the list start position.

Name:	`GEN llist_togvec`
Input:	`llist *l, long length, int dir`
Input format:	`length` the length of `l`, and `dir=-1` or `1`.
Output format:	Vector
Description:	Returns a vector consisting of the list from `l` and onward of length `length`, and frees the list. If `dir=1` we fill in the return vector from index `1` to `length`, and if `dir=-1` we go in the opposite direction.

Name:	`GEN llist_tovecsmall`
Input:	`llist *l, long length, int dir`
Input format:	`length` the length of `l`, and `dir=-1` or `1`.
Output format:	Vecsmall
Description:	Returns a Vecsmall consisting of the list from `l` and onward of length `length`, and frees the list. If `dir=1` we fill in the return vector from index `1` to `length`, and if `dir=-1` we go in the opposite direction.

3 c_bqf

These methods primarily deal with primitive integral homogeneous positive definite/indefinite binary quadratic forms. Such a form is represented by the vector $[A, B, C]$, which represents $AX^2+BXY+CY^2$. Some of the basic methods support non-primitive, negative definite, or square discriminant forms (like `bqf_disc` or `bqf_trans`), but more complex ones (like `bqf_isequiv`) may not.

On the other hand, the method `bqf_reps` allows non-primitive forms, as well as negative definite and square discriminant forms. Going further, `bqf_bigreps` allows non-homogeneous binary quadratic forms (but the integral requirement is never dropped).

In this and subsequent sections, a **BQF** is an integral binary quadratic form, an **IBQF** is an indefinite BQF, a **DBQF** is a positive definite BQF, a **PIBQF/PDBQF** is a primitive indefinite/positive definite BQF respectively, and a **PBQF** is either a PIBQF or a PDBQF.

In general, a method taking in a BQF will start with `bqf_`. This is further specialized to indefinite/positive definite/square discriminant/zero discriminant forms be adding the prefixes `i/d/s/z` respectively.

3.1 Discriminant methods

These methods deal with discriminant operations that do not involve quadratic forms.

Name:	`GEN disclist`
Input:	`GEN D1, GEN D2, int fund, GEN cop`
Input format:	Integers `D1`, `D2`, `fund=0, 1`, `cop` an integer
Output format:	Vector
Description:	Returns the set of discriminants (non-square integers equivalent to `0, 1` modulo 4) between `D1` and `D2` inclusive. If `fund=1`, only returns fundamental discriminants, and if `cop`≠`0`, only returns discriminants coprime to `cop`.

Name:	`GEN discprimeindex`
Input:	`GEN D, GEN facs`
Input format:	Discriminant `D`, `facs=0` or the factorization of `D` (the output of `Z_factor`)
Output format:	Vector
Description:	Returns the set of primes p for which D/p^2 is a discriminant.

Name:	`GEN discprimeindex_typecheck`
Input:	`GEN D`
Input format:	Discriminant `D`
Output format:	Vector
Description:	Checks that `D` is a discriminant, and returns `discprimeindex(D, gen_0)`.

Name:	`GEN fdisc`
Input:	`GEN D`
Input format:	Discriminant `D`
Output format:	Integer
Description:	Returns the fundamental discriminant associated to `D`.

Name:	`GEN fdisc_typecheck`
Input:	`GEN D`
Input format:	Discriminant `D`
Output format:	Integer
Description:	Checks that `D` is a discriminant, and returns `fdisc(D)` if so. Returns `gen_0` if not a discriminant.

Name:	`int isdisc`
Input:	`GEN D`
Input format:	-
Output format:	`0` or `1`
Description:	Returns `1` if `D` is a discriminant and `0` else.

Name:	`GEN pell`
Input:	`GEN D`
Input format:	Positive discriminant `D`
Output format:	`[T, U]`
Description:	Returns the smallest solution in the positive integers to Pell's equation $T^2 - DU^2 = 4$.

Name:	`GEN pell_typecheck`
Input:	`GEN D`
Input format:	Positive discriminant `D`
Output format:	`[T, U]`
Description:	Checks that `D` is a positive discriminant, and returns `pell(D)`.

Name:	`GEN posreg`
Input:	`GEN D, long prec`
Input format:	Positive discriminant `D`, precision `prec`
Output format:	Real number
Description:	Returns the positive regulator of $\mathcal{O}_D$, i.e. the logarithm of the fundamental unit of norm 1 in the unique order of discriminant D.

Name:	`GEN posreg_typecheck`
Input:	`GEN D, long prec`
Input format:	Positive discriminant `D`, precision `prec`
Output format:	Real number
Description:	Checks that `D` is a positive discriminant, and returns `posreg(D, prec)`.

Name:	`GEN quadroot`
Input:	`GEN D`
Input format:	Discriminant `D`
Output format:	`t_QUAD`
Description:	Outputs the `t_QUAD` `w` for which $w^2 = D$.

Name:	`GEN quadroot_typecheck`
Input:	`GEN D`
Input format:	Discriminant `D`
Output format:	`t_QUAD`
Description:	Checks that `D` is a discriminant and returns `quadroot(D)`.

3.2 Basic methods for binary quadratic forms

Recall that the BQF $AX^2 + BXY + CY^2$ is represented as the vector $[A, B, C]$.

Name:	`GEN bqf_automorph_typecheck`
Input:	`GEN q`
Input format:	PBQF q
Output format:	Matrix
Description:	Returns the invariant automorph `M` of `q`, i.e. the $\mathrm{PSL}(2, \mathbb{Z})$ matrix with positive trace that generates the stabilizer of `q` (a cyclic group of order 1, 2, 3, or ∞).

Name:	`int bqf_compare`
Input:	`void *data, GEN q1, GEN q2`
Input format:	`*data=NULL`, `q1` and `q2` BQFs
Output format:	`-1, 0, 1`
Description:	Lexicographically compares `q1` and `q2`, returning `-1` if $q_1 < q_2$, `0` if $q_1 = q_2$, and `1` if $q_1 > q_2$. This method is used to sort and search a set of BQFs more efficiently (with `gen_sort` and `gen_search`).

Name:	`int bqf_compare_tmat`
Input:	`void *data, GEN d1, GEN d2`
Input format:	`*data=NULL`, `di=[qi, mi]` with `qi` a BQF for `i=1,2`
Output format:	`-1, 0, 1`
Description:	Lexicographically compares `q1` and `q2`, returning `-1` if $q_1 < q_2$, `0` if $q_1 = q_2$, and `1` if $q_1 > q_2$. This method is used to sort and search a set of BQFs where we are also keeping track of an extra data point `mi`, often a transition matrix (whose value has no effect on the output of this method).

Name:	`GEN bqf_disc`
Input:	`GEN q`
Input format:	BQF `q`
Output format:	Integer
Description:	Returns the discriminant of `q`, i.e. $B^2 - 4AC$ where `q=[A, B, C]`.

Name:	`GEN bqf_disc_typecheck`
Input:	`GEN q`
Input format:	BQF `q`
Output format:	Integer
Description:	Checks that `q` is a BQF, and returns `bqf_disc(q)`.

Name:	`GEN bqf_isequiv`
Input:	`GEN q1, GEN q2, GEN rootD, int Dsign, int tmat`
Input format:	PBQFs `q1, q2` of the same discriminant D, `rootD` the real square root of D if $D > 0$ (and anything if $D < 0$), `Dsign` the sign of D, `tmat=0, 1`
Output format:	`gen_0`, `gen_1`, or a matrix
Description:	Determines if `q1` and `q2` are equivalent or not. If `tmat=0`, returns `gen_0` or `gen_1`, and if `tmat=1`, returns `gen_0` if not equivalent and a $\mathrm{SL}(2, \mathbb{Z})$ transition matrix taking `q1` to `q2` if they are equivalent.

Name:	`GEN bqf_isequiv_set`
Input:	`GEN q, GEN S, GEN rootD, int Dsign, int tmat`
Input format:	PBQFs `q` and set of PBQFs `S`, all of the same discriminant D, `rootD` the real square root of D if $D > 0$ (and anything if $D < 0$), `Dsign` the sign of D, `tmat=0, 1`
Output format:	Integer or `[i, M]`
Description:	Determines if `q` and an element of `S` are equivalent or not. If `tmat=0`, returns `gen_0` if not and an index `i` such that `q` is equivalent to `S[i]` if they are equivalent. If `tmat=1`, returns `gen_0` if not equivalent and `[i, M]` if they are, where $M \circ q = S[i]$.

Name:	`GEN bqf_isequiv_typecheck`
Input:	`GEN q1, GEN q2, int tmat, long prec`
Input format:	`q1` a PBQF, `q2` a PBQF or a set of PBQFs, `tmat=0, 1`, `prec` the precision
Output format:	Integer or matrix or `[i, M]`
Description:	Checks if `q1` is a PBQF, `q2` is a PBQF or a set of PBQFs, and returns `bqf_isequiv` or `bqf_isequiv_set` on `q1` and `q2` as appropriate. Elements of `q2` need not have the same discriminant as each other or `q1`.

Name:	`int bqf_isreduced`
Input:	`GEN q, int Dsign`
Input format:	`q` a PBQF of discriminant D, `Dsign` the sign of D
Output format:	`0, 1`
Description:	Returns `1` if `q` is reduced, and `0` is `q` is not reduced. We use the standard reduced definition when $D < 0$, and the conditions $AC < 0$ and $B > \lvert A+C \rvert$ when $D > 0$.

Name:	`int bqf_isreduced_typecheck`
Input:	`GEN q`
Input format:	`q` a PBQF
Output format:	`0, 1`
Description:	Checks that `q` is q PBQF and returns `1` if reduced, and `0` if not reduced.

Name:	`GEN bqf_random`
Input:	`GEN maxc, int type, int primitive`
Input format:	`maxc` a positive integer, `type, primitive=0, 1`
Output format:	BQF
Description:	Returns a random BQF of non-square discriminant with coefficient size at most `maxc`. If `type=-1` it will be positive definite, `type=1` indefinite, and `type=0` either type. If `primitive=1` the form will be primitive, otherwise it need not be.

Name:	`GEN bqf_random_D`
Input:	`GEN maxc, GEN D`
Input format:	`maxc` a positive integer, `D` a discriminant
Output format:	BQF
Description:	Checks that `maxc` is a positive integer and `D` is a discriminant, and returns a random primitive BQF of discriminant `D` (positive definite if $D < 0$).

Name:	`GEN bqf_red`
Input:	`GEN q, GEN rootD, int Dsign, int tmat`
Input format:	`q` a PBQF of discriminant D, `rootD` the square root of D if $D > 0$ (and anything if $D < 0$), `Dsign` the sign of D, `tmat=0,1`
Output format:	BQF or `[q', M]`
Description:	Outputs the reduction of `q`. If `tmat=0` this is a BQF, otherwise this is `[q', M]` where the reduction is `q'` and the transition matrix is `M`.

Name:	`GEN bqf_red_typecheck`
Input:	`GEN q, int tmat, long prec`
Input format:	`q` a PBQF, `tmat=0,1`, `prec` the precision
Output format:	BQF or `[q', M]`
Description:	Checks that `q` is a PBQF, and returns `bqf_red(q,...)`.

Name:	`GEN bqf_roots`
Input:	`GEN q, GEN D, GEN w`
Input format:	BQF `q` of discriminant `D`, with $w^2 = D$ where `w` is a `t_QUAD` if `D` is not a square
Output format:	`[r1, r2]`
Description:	Returns the roots of `q(x,1)=0`, with the first root coming first. If `D` is not a square, these are of type `t_QUAD`, and otherwise they will be rational or infinite. If `D=0`, the roots are equal.

Name:	`GEN bqf_roots_typecheck`
Input:	`GEN q`
Input format:	BQF `q`
Output format:	`[r1, r2]`
Description:	Checks that `q` is a BQF, and returns `bqf_roots(q,...)`.

Name:	`GEN bqf_trans`
Input:	`GEN q, GEN M`
Input format:	BQF `q`, $M \in \mathrm{SL}(2,\mathbb{Z})$
Output format:	BQF
Description:	Returns $M \circ q$.

Name:	`GEN bqf_trans_typecheck`
Input:	`GEN q, GEN M`
Input format:	BQF `q`, $M \in \mathrm{SL}(2,\mathbb{Z})$
Output format:	BQF
Description:	Checks that `q` is q BQF and $M \in \mathrm{SL}(2,\mathbb{Z})$, and returns `bqf_trans(q, M)`.

Name:	`GEN bqf_transL`
Input:	`GEN q, GEN n`
Input format:	BQF `q`, integer `n`
Output format:	BQF
Description:	Returns $\left(\begin{smallmatrix}1 & n\\ 0 & 1\end{smallmatrix}\right) \circ q$.

Name:	`GEN bqf_transR`
Input:	`GEN q, GEN n`
Input format:	BQF `q`, integer `n`
Output format:	BQF
Description:	Returns $\left(\begin{smallmatrix}1 & 0\\ n & 1\end{smallmatrix}\right) \circ q$.

Name:	`GEN bqf_transS`
Input:	`GEN q`
Input format:	BQF `q`
Output format:	BQF
Description:	Returns $\left(\begin{smallmatrix}0 & 1\\ -1 & 0\end{smallmatrix}\right) \circ q$.

Name:	`GEN bqf_trans_coprime`
Input:	`GEN q, GEN n`
Input format:	BQF `q`, integer `n` coprime to $\gcd(q)$
Output format:	BQF
Description:	Returns a BQF equivalent to `q` whose first coefficient is coprime to `n`.

Name:	`GEN bqf_trans_coprime_typecheck`
Input:	`GEN q, GEN n`
Input format:	BQF `q`, non-zero integer `n`
Output format:	BQF
Description:	Checks that `q` is a BQF and `n` is coprime to $\gcd(q)$, and returns `bqf_trans_coprime(q, n)`.

Name:	`GEN ideal_tobqf`
Input:	`GEN numf, GEN ideal`
Input format:	`numf` a quadratic number field, `ideal` an ideal in `numf`
Output format:	BQF
Description:	Converts the ideal to a BQF and returns it.

3.3 Basic methods, but specialized

These are the above basic methods, but specialized to the positive definite/indefinite cases.

Name:	`GEN dbqf_automorph`
Input:	`GEN q, GEN D`
Input format:	PDBQF `q` of discriminant `D`
Output format:	Matrix
Description:	Returns the invariant automorph `M` of `q` in $\mathrm{PSL}(2,\mathbb{Z})$, which is trivial if $D < -4$, has order 2 if $D = -4$, and order 3 if $D = -3$.

Name:	`GEN dbqf_isequiv`
Input:	`GEN q1, GEN q2`
Input format:	DBQFs `q1, q2` of the same discriminant
Output format:	`gen_0`, `gen_1`
Description:	Returns 1 if $q1$ is equivalent to $q2$ and 0 if not.

Name:	`long dbqf_isequiv_set`
Input:	`GEN q, GEN S`
Input format:	DBQF `q` and a set of DBQFs `S`, with all forms in `S` and `q` having the same discriminant
Output format:	Integer
Description:	Returns `gen_0` if `q` is not similar to any form in `S`, and an index `i` such that `q` is similar to `S[i]` otherwise.

Name:	`GEN dbqf_isequiv_set_tmat`
Input:	`GEN q, GEN S`
Input format:	DBQF `q` and a set of DBQFs `S`, with all forms in `S` and `q` having the same discriminant
Output format:	`gen_0` or `[i, M]`
Description:	Returns `gen_0` if `q` is not similar to any form in `S`, and otherwise returns `[o, M]`, where `q` is similar to `S[i]` with transition matrix `M`.

Name:	`GEN dbqf_isequiv_tmat`
Input:	`GEN q1, GEN q2`
Input format:	DBQFs `q1, q2` of the same discriminant
Output format:	`0`, matrix
Description:	Returns `0` if `q1` is not equivalent to `q2` and a possible transition matrix if it is.

Name:	`GEN dbqf_red`
Input:	`GEN q`
Input format:	`q` a DBQF
Output format:	DBQF
Description:	Returns the reduction of `q`.

Name:	`GEN dbqf_red_tmat`
Input:	`GEN q`
Input format:	`q` a DBQF
Output format:	`[q', M]`
Description:	Returns `[q', M]`, where the reduction of `q` is `q'` and the transition matrix is `M`.

Name:	`GEN ibqf_automorph_D`
Input:	`GEN q, GEN D`
Input format:	`q` a PIBQF of discriminant `D`
Output format:	Matrix
Description:	Returns the invariant automorph of `q`, i.e. the generator with positive trace (and infinite order) of the stabilizer of `q` in $\mathrm{PSL}(2,\mathbb{Z})$. This method calls `pell(D)`, so if this is already computed, use `ibqf_automorph_pell` instead.

Name:	`GEN ibqf_automorph_pell`
Input:	`GEN q, GEN qpell`
Input format:	`q` a PIBQF of discriminant `D`, where `qpell` is the output of `pell(D)`
Output format:	Matrix
Description:	Returns the invariant automorph of `q`. If you don't care about the output of `pell(D)`, then use `ibqf_automorph_D` instead.

Name:	`GEN ibqf_isequiv`
Input:	`GEN q1, GEN q2, GEN rootD`
Input format:	PIBQFs `q1, q2` of the same discriminant D, `rootD` the real square root of D
Output format:	`gen_0`, `gen_1`
Description:	Determines if `q1` and `q2` are equivalent or not, and returns the answer.

Name:	`long ibqf_isequiv_set_byq`
Input:	`GEN q, GEN S, GEN rootD`
Input format:	PIBQFs `q` and set of PIBQFs `S`, all of the same discriminant D, `rootD` the real square root of D
Output format:	Integer
Description:	Determines if `q` and an element of `S` are equivalent or not. Returns `0` if not equivalent, and an index `i` such that `q` is equivalent to `S[i]` otherwise. Generally slower than `ibqf_isequiv_set_byS`, so not recommended for use.

Name:	`long ibqf_isequiv_set_byq_presorted`
Input:	`GEN qredsorted, GEN S, GEN rootD`
Input format:	`qredsorted` is `ibqf_redorbit_posonly(q, rootD)`, sorted with `gen_sort_inplace(qredsorted, NULL, &bqf_compare, NULL)`, `S` is a set of PIBQFs with all forms in `S` and `q` having discriminant D, and `rootD` is the positive square root of D
Output format:	Integer
Description:	`ibqf_isequiv_set_byq` where the sorted positive reduced orbit of `q` is inputted. Useful if you are making multiple calls to `bqf_is_equiv_set` with the same `q` but varying sets `S`. If this is not the case, `ibqf_isequiv_byS` is generally faster.

Name:	`GEN ibqf_isequiv_set_byq_tmat`
Input:	`GEN q, GEN S, GEN rootD`
Input format:	PIBQFs `q` and set of PIBQFs `S`, all of the same discriminant D, `rootD` the real square root of D
Output format:	`gen_0`, `[i, M]`
Description:	Determines if `q` and an element of `S` are equivalent or not. Returns `gen_0` if not equivalent, and `[i, M]` otherwise, where `q` is equivalent to `S[i]` with transition matrix `M`. Generally slower than `ibqf_isequiv_set_byS_tmat`, so not recommended for use.

Name:	`GEN ibqf_isequiv_set_byq_tmat_presorted`
Input:	`GEN qredsorted, GEN S, GEN rootD`
Input format:	`qredsorted` is `ibqf_redorbit_posonly_tmat(q, rootD)` sorted with `gen_sort_inplace(qredsorted, NULL, &bqf_compare_tmat, NULL)`, `S` is a set of PIBQFs with all forms in `S` and `q` having discriminant D, and `rootD` is the positive square root of D
Output format:	`gen_0`, `[i, M]`
Description:	`ibqf_isequiv_set_byq_tmat` where the sorted positive reduced orbit of `q` is inputted. Useful if you are making multiple calls to `bqf_is_equiv_set` with the same `q` but varying sets `S`. If this is not the case, `ibqf_isequiv_byS` is generally faster.

Name:	`long ibqf_isequiv_set_byS`
Input:	`GEN q, GEN S, GEN rootD`
Input format:	PIBQFs `q` and set of PIBQFs `S`, all of the same discriminant D, `rootD` the real square root of D
Output format:	Integer
Description:	Determines if `q` and an element of `S` are equivalent or not. Returns `0` if not equivalent, and an index `i` such that `q` is equivalent to `S[i]` otherwise. Generally faster than `ibqf_isequiv_set_byq`.

Name:	`long ibqf_isequiv_set_byS_presorted`
Input:	`GEN q, GEN Sreds, GEN perm, GEN rootD`
Input format:	`q`, `S`, and `rootD` as in `ibqf_isequiv_set_byS`, where the forms in `S` are reduced with `ibqf_red_pos` to get `Sreds` and sorted by `gen_sort_inplace(Sreds, NULL, &bqf_compare, &perm)`
Output format:	Integer
Description:	`ibqf_isequiv_set_byS` where we reduce `S` and sort it. Useful if you are making multiple calls to `bqf_is_equiv_set` with the same set `S` but varying forms `q`.

Name:	`GEN ibqf_isequiv_set_byS_tmat`
Input:	`GEN q, GEN S, GEN rootD`
Input format:	PIBQFs `q` and set of PIBQFs `S`, all of the same discriminant D, `rootD` the real square root of D
Output format:	`gen_0`, `[i, M]`
Description:	Determines if `q` and an element of `S` are equivalent or not. Returns `gen_0` if not equivalent, and `[i, M]` otherwise, where `q` is equivalent to `S[i]` with transition matrix `M`. Generally faster than `ibqf_isequiv_set_byq_tmat`.

Name:	`GEN ibqf_isequiv_set_byS_tmat_presorted`
Input:	`GEN q, GEN Sreds, GEN perm, GEN rootD`
Input format:	`q`, `S`, and `rootD` as in `ibqf_isequiv_set_byS`, where the forms in `S` are reduced with `ibqf_red_pos` to get `Sreds` and sorted by `gen_sort_inplace(Sreds, NULL, &bqf_compare_tmat, &perm)`
Output format:	`gen_0`, `[i, M]`
Description:	`ibqf_isequiv_set_byS_tmat` where we reduce `S` and sort it. Useful if you are making multiple calls to `bqf_is_equiv_set` with the same `q` but varying sets `S`.

Name:	`GEN ibqf_isequiv_tmat`
Input:	`GEN q1, GEN q2, GEN rootD`
Input format:	PIBQFs `q1, q2` of the same discriminant D, `rootD` the real square root of D
Output format:	`gen_0`, matrix
Description:	Determines if `q1` and `q2` are equivalent or not, returning a transition matrix if they are and `gen_0` if not.

Name:	`GEN ibqf_red`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	BQF
Description:	Returns a reduction of `q`.

Name:	`GEN ibqf_red_tmat`
Input:	`GEN q, GEN rootD`
Input format:	`q` a PBQF of discriminant D, `rootD` the square root of D
Output format:	`[q', M]`
Description:	Returns `[q', M]`, where `q'` is a reduction of `q` and `M` is the transition matrix.

Name:	`GEN ibqf_red_pos`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	BQF
Description:	Returns a reduction of `q` with positive first coefficient.

Name:	`GEN ibqf_red_pos_tmat`
Input:	`GEN q, GEN rootD`
Input format:	`q` a PBQF of discriminant D, `rootD` the square root of D
Output format:	`[q', M]`
Description:	Returns `[q', M]`, where `q'` is a reduction of `q` with positive first coefficient and `M` is the transition matrix.

3.4 Basic methods for indefinite quadratic forms

Methods in this section are specific to indefinite forms. The "river" is the river of the Conway topograph; it is a periodic ordering of the forms $[A, B, C] \sim q$ with $AC < 0$. Reduced forms with $A > 0$ occur between branches pointing down and up (as we flow along the river), and reduced forms with $A < 0$ occur between branches pointing up and down.

Name:	`int ibqf_isrecip`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	`0, 1`
Description:	Returns `1` if `q` is reciprocal (`q` is similar to `-q`), and `0` else.

Name:	`int ibqf_isrecip_typecheck`
Input:	`GEN q, long prec`
Input format:	`q` an IBQF of discriminant D, `prec` the precision
Output format:	`0, 1`
Description:	Checks that `q` is an IBQF, and returns `ibqf_isrecip(q, rootD)`.

Name:	`GEN ibqf_leftnbr`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D with $AC < 0$, `rootD` the square root of D
Output format:	IBQF
Description:	Returns the left neighbour of `q`, i.e. the nearest reduced form on the river to the left of `q`.

Name:	`GEN ibqf_leftnbr_tmat`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D with $AC < 0$, `rootD` the square root of D
Output format:	`[q', M]`
Description:	Returns `[q', M]`, with `q'` the left neighbour of `q`, and the transition matrix is `M`.

Name:	`GEN ibqf_leftnbr_typecheck`
Input:	`GEN q, int tmat, long prec`
Input format:	`q` an IBQF of discriminant D with $AC < 0$, `tmat=0, 1`, `prec` the precision
Output format:	IBQF or `[q', M]`
Description:	Checks that `q` is an IBQF on the river, and returns the left neighbour. If `tmat=0` only returns the IBQF, and if `tmat=1` returns the form and transition matrix.

Name:	`GEN ibqf_leftnbr_update`
Input:	`GEN qvec, GEN rootD`
Input format:	`qvec=[q, M]` with `q` an IBQF of discriminant D with $AC < 0$ and $M \in \mathrm{SL}(2, \mathbb{Z})$,`rootD` the square root of D
Output format:	`[q', M']`
Description:	Returns `[q', M']`, where `q'` is the left neighbour of `q`, the transition matrix is `M''`, and `M'=MM''`. This method makes it easier to apply the left neighbour function multiple times while keeping track of the transition matrix from our original starting point.

Name:	`GEN ibqf_redorbit`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	Vector
Description:	Returns the reduced orbit of `q`.

Name:	`GEN ibqf_redorbit_tmat`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	Vector
Description:	Returns the reduced orbit of `q`, where each entry is `[q', M]`, with the reduced form being `q'` and the transition matrix from `q` being `M`.

Name:	`GEN ibqf_redorbit_posonly`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	Vector
Description:	Returns the reduced orbit of `q`, where we only keep the BQFs with positive first coefficient.

Name:	`GEN ibqf_redorbit_posonly_tmat`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	Vector
Description:	Returns the reduced orbit with positive first coefficient of `q`, where each entry is `[q', M]`, with the reduced form being `q'` and the transition matrix from `q` being `M`.

Name:	`GEN ibqf_redorbit_typecheck`
Input:	`GEN q, int tmat, int posonly, long prec`
Input format:	`q` an IBQF, `tmat, posonly=0, 1`, `prec` the precision
Output format:	Vector
Description:	Returns the reduced orbit of `q`. If `tmat=1` each entry is the pair `[q', M]` of form and transition matrix, otherwise each entry is just the form. If `posonly=1`, we only take the reduced forms with positive first coefficient (half of the total), otherwise we take all reduced forms.

Name:	`GEN ibqf_rightnbr`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D with $AC < 0$, `rootD` the square root of D
Output format:	IBQF
Description:	Returns the right neighbour of `q`, i.e. the nearest reduced form on the river to the right of `q`.

Name:	`GEN ibqf_rightnbr_tmat`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D with $AC < 0$, `rootD` the square root of D
Output format:	`[q', M]`
Description:	Returns `[q', M]`, with `q'` the right neighbour of `q`, and the transition matrix is `M`.

Name:	`GEN ibqf_rightnbr_typecheck`
Input:	`GEN q, int tmat, long prec`
Input format:	`q` an IBQF of discriminant D with $AC < 0$, `tmat=0, 1`, `prec` the precision
Output format:	IBQF or `[q', M]`
Description:	Checks that `q` is an IBQF on the river, and returns the right neighbour. If `tmat=0` only returns the IBQF, and if `tmat=1` returns the form and transition matrix.

Name:	`GEN ibqf_rightnbr_update`
Input:	`GEN qvec, GEN rootD`
Input format:	`qvec=[q, M]` with `q` an IBQF of discriminant D with $AC < 0$ and $M \in \mathrm{SL}(2, \mathbb{Z})$,`rootD` the square root of D
Output format:	`[q', M']`
Description:	Returns `[q', M']`, where `q'` is the right neighbour of `q`, the transition matrix is `M''`, and `M'=MM''`. This method makes it easier to apply the right neighbour function multiple times while keeping track of the transition matrix from our original starting point.

Name:	`GEN ibqf_river`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	Vector
Description:	Returns the river sequence associated to `q`. The entry `gen_1` indicates going right, and `gen_0` indicates going left along the river.

Name:	`GEN ibqf_river_positions`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	Vector
Description:	Returns `[Lpos, Rpos, riv]`, where `riv` is `ibqf_river(q)` but as a `t_VECSMALL`, `Lpos` is a `t_VECSMALL` of indices `i` for which `riv[i]=0`, and `Rpos` is a `t_VECSMALL` of indices `i` for which `riv[i]=1`.

Name:	`GEN ibqf_river_positions_forms`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	Vector
Description:	Returns `[Lpos, Rpos, rivforms]`, where `rivforms` is the vector of forms on the river `riv` in order, `Lpos` is a `t_VECSMALL` of indices `i` for which `riv[i]=0`, and `Rpos` is a `t_VECSMALL` of indices `i` for which `riv[i]=1`.

Name:	`GEN ibqf_river_typecheck`
Input:	`GEN q, long prec`
Input format:	`q` an IBQF of discriminant D, `prec` the precision
Output format:	Vector
Description:	Checks that `q` is an IBQF and returns `ibqf_river(q)`.

Name:	`GEN ibqf_riverforms`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	Vector
Description:	Returns the forms on the river of `q`, where we only take the forms with first coefficient positive.

Name:	`GEN ibqf_riverforms_typecheck`
Input:	`GEN q, long prec`
Input format:	`q` an IBQF of discriminant D, `prec` the precision
Output format:	Vector
Description:	Checks that `q` is an IBQF, and returns `ibqf_riverforms(q)`.

Name:	`GEN ibqf_symmetricarc`
Input:	`GEN q, GEN D, GEN rootD, GEN qpell, long prec`
Input format:	`q` an IBQF of discriminant `D`, `rootD` the positive square root of `D`, `qpell=pell(D)`, and `prec` the precision
Output format:	$[z, \gamma_q(z)]$
Description:	If γ_q is the invariant automorph of q, this computes the complex number `z`, where `z` is on the root geodesic of `q` and $z, \gamma_q(z)$ are symmetric (they have the same imaginary part). This gives a "nice" upper half plane realization of the image of the root geodesic of `q` on $\mathrm{PSL}(2,\mathbb{Z})\backslash\mathbb{H}$ (a closed geodesic). However, if the automorph of `q` is somewhat large, z and $\gamma_q(z)$ will be very close to the $x-$axis, and this method isn't very useful.

Name:	`GEN ibqf_symmetricarc_typecheck`
Input:	`GEN q, long prec`
Input format:	`q` an IBQF of discriminant D, `prec` the precision
Output format:	$[z, \gamma_q(z)]$
Description:	Checks that `q` is an IBQF, and returns `ibqf_symmetricarc(q, ...)`.

Name:	`GEN ibqf_toriver`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	IBQF
Description:	Reduces `q` to the river and returns it. Mostly useful as a supporting method for `ibqf_red`.

Name:	`GEN ibqf_toriver_tmat`
Input:	`GEN q, GEN rootD`
Input format:	`q` an IBQF of discriminant D, `rootD` the square root of D
Output format:	`[q', M]`
Description:	Reduces `q` to the river and returns the reduction `q'` and the transition matrix `M`. Mostly useful as a supporting method for `ibqf_red_tmat`.

Name:	`GEN mat_toibqf`
Input:	`GEN M`
Input format:	$M \in \mathrm{SL}(2,\mathbb{Z})$
Output format:	PBQF
Description:	Output the PBQF corresponding to the equation `M(x)=x`. Typically used when `M` has determinant `1` and is hyperbolic, so that the output is a PIBQF (this method is inverse to `ibqf_automorph_D` in this case).

Name:	`GEN mat_toibqf_typecheck`
Input:	`GEN M`
Input format:	$M \in \mathrm{SL}(2, \mathbb{Z})$
Output format:	PBQF
Description:	Checks that `M` is a `2x2` integral matrix, and returns `mat_toibqf(M)`. This method does not check that `M` is hyperbolic or that it has determinant `1`.

3.5 Class group and composition of forms

This section deals with class group related computations. To compute the class group we take the built-in PARI methods, which cover the cases when D is fundamental and when the narrow and full class group coincide. For the remaining cases, we "boost up" the full class group to the narrow class group with `bqf_ncgp_nonfundnarrow`.

Name:	`GEN bqf_comp`
Input:	`GEN q1, GEN q2`
Input format:	PBQFs `q1, q2` of the same discriminant
Output format:	PBQF
Description:	Returns the composition of `q1` and `q2`.

Name:	`GEN bqf_comp_red`
Input:	`GEN q1, GEN q2, GEN rootD, int Dsign`
Input format:	PBQFs `q1, q2` of the same discriminant D, `rootD` the positive square root of D if $D > 0$ (and anything if $D < 0$), `Dsign` the sign of D
Output format:	PBQF
Description:	Composes `q1, q2` and returns an equivalent reduced form.

Name:	`GEN bqf_comp_typecheck`
Input:	`GEN q1, GEN q2, int tored, long prec`
Input format:	PBQFs `q1, q2` of the same discriminant, `tored=0, 1`, `prec` the precision
Output format:	PBQF
Description:	Checks that `q1, q2` have the same discriminant, and returns their composition. If `tored=1` the form is reduced, otherwise it is not.

Name:	`GEN bqf_idelt`
Input:	`GEN D`
Input format:	Discriminant `D`
Output format:	BQF
Description:	Returns the identity element of discriminant `D`.

Name:	`GEN bqf_ncgp`
Input:	`GEN D, long prec`
Input format:	Discriminant `D`, `prec` the precision
Output format:	`[n, orders, forms]`
Description:	Computes and returns the narrow class group associated to `D`. `n` is the order of the group, `orders=[d1, d2, ..., dk]` where $d_1 \mid d_2 \mid \cdots \mid d_k$ and the group is isomorphic to $\prod_{i=1}^{k} \frac{\mathbb{Z}}{d_i\mathbb{Z}}$, and `forms` is the length `k` vector of PBQFs corresponding to the decomposition (so `forms[i]` has order `di`).

Name:	`GEN bqf_ncgp_lexic`
Input:	`GEN D, long prec`
Input format:	Discriminant `D`, `prec` the precision
Output format:	`[n, orders, forms]`
Description:	Computes and returns the narrow class group associated to `D`. The output is the same as `bqf_ncgp`, except the third output is now a lexicographical listing of representatives of all equivalence classes of forms of discriminant `D` (instead of the generators).

Name:	`GEN bqf_pow`
Input:	`GEN q, GEN n`
Input format:	PBQF `q`, integer `n`
Output format:	PBQF
Description:	Returns q^n.

Name:	`GEN bqf_pow_red`
Input:	`GEN q, GEN n, GEN rootD, int Dsign`
Input format:	PBQF `q` of discriminant D, integer `n`, `rootD` the positive square root of D if $D > 0$ (and anything if $D < 0$), `Dsign` the sign of D
Output format:	PBQF
Description:	Returns a reduction of q^n.

Name:	`GEN bqf_pow_typecheck`
Input:	`GEN q, GEN n, int tored, long prec`
Input format:	PBQF `q`, integer `n`, `tored=0, 1`, `prec` the precision
Output format:	PBQF
Description:	Checks that `q` is a PBQF and `n` is an integer, and returns a form equivalent to q^n. If `tored=1` the form is reduced, otherwise it is not necessarily.

Name:	`GEN bqf_square`
Input:	`GEN q`
Input format:	PBQF `q`
Output format:	PBQF
Description:	Returns q^2.

Name:	`GEN bqf_square_red`
Input:	`GEN q, GEN rootD, int Dsign`
Input format:	PBQF `q` of discriminant D, `rootD` the positive square root of D if $D > 0$ (and anything if $D < 0$), `Dsign` the sign of D
Output format:	PBQF
Description:	Returns a reduction of q^2.

Name:	`GEN bqf_square_typecheck`
Input:	`GEN q, int tored, long prec`
Input format:	PBQF `q`, `tored=0, 1`, `prec` the precision
Output format:	PBQF
Description:	Checks `q` is a PBQF, and returns a form equivalent to q^2. If `tored=1` the form is reduced, otherwise it is not necessarily.

Name:	`GEN bqf_ncgp_nonfundnarrow`
Input:	`GEN cgp, GEN D, GEN rootD`
Input format:	`cgp=quadclassunit0(D, 0, NULL, prec)`, `D` a positive (typically non-fundamental) discriminant with norm of the fundamental unit being `1`, `rootD` the positive square root of `D`
Output format:	`[n, orders, forms]`
Description:	With the described conditions, the narrow class group is twice the size of the class group. Since `quadclassunit0` computes the class group, this method modifies the output to computing the full narrow class group, and returns it in the same format as `bqf_ncgp`.

3.6 Representation of integers by forms - description tables

This section deals with questions of representing integers by quadratic forms. The three main problems we solve are

- Find all integral solutions (X, Y) to $AX^2 + BXY + CY^2 = n$ (`bqf_reps`);
- Find all integral solutions (X, Y) to $AX^2 + BXY + CY^2 + DX + EY = n$ (`bqf_bigreps`);
- Find all integral solutions (X, Y, Z) to $AX^2 + BY^2 + CZ^2 + DXY + EXZ + FYZ = n_1$ and $UX + VY + WZ = n_2$ (`bqf_linearsolve`).

The general solution descriptions have a lot of cases, so we put the descriptions in Tables 1-3, and refer to the tables in the method descriptions.

For `bqf_reps`, let $q = [A, B, C]$ and let $d = B^2 - 4AC$. If there are no solutions the method will return `gen_0`, and otherwise it will return a vector `v`, where

$$v = [[\text{type}, v_{extra}], v_1, v_2, \ldots, v_k].$$

The types are are

-1=all, 0=finite, 1=positive, 2=linear.

Each (family of) solution(s) is given by a v_i, possibly with reference to the extra data. In this table we will only describe **half** of all solutions: we are only taking one of (X, Y) and $(-X, -Y)$. If you want all solutions without this restriction, you just have to add in these negatives.

Table 1: General solution for `bqf_reps`

Type	Conditions to appear	v_{extra}	v_i format	General solution
-1	$q = 0$, $n = 0$	-	-	X, Y are any integers
0	$d < 0$	-	$[x_i, y_i]$	$X = x_i$ and $Y = y_i$
	$d = \square > 0$,[a] $n \neq 0$			
	$d = \boxtimes$,[a] $n = 0$			
1	$d = \boxtimes > 0$, $n \neq 0$	M [b]	$[x_i, y_i]$	$\binom{X}{Y} = M^j \binom{x_i}{y_i}$ for $j \in \mathbb{Z}$
2	$d = 0$, $n \neq 0$	-	$[[s_i, t_i], [x_i, y_i]]$	$X = x_i + s_iU$, $Y = y_i + t_iU$ for $U \in \mathbb{Z}$
	$d = \square > 0$, $n = 0$			

[a] $\square$ means square, and $\boxtimes$ means non-square.
[b] $M \in \mathrm{SL}(2, \mathbb{Z})$

For `bqf_bigreps`, let $q = [A, B, C, D, E]$ and let $d = B^2 - 4AC$. If there are no solutions the method will return `gen_0`, and otherwise it will return a vector `v`, where

$$v = [[\text{type}, v_{extra}], v_1, v_2, \ldots, v_k].$$

The types are are

-2=quadratic, -1=all, 0=finite, 1=positive, 2=linear.

Each (family of) solution(s) is given by a v_i, possibly with reference to the extra data.

Table 2: General solution for `bqf_bigreps`

Type	Conditions to appear	v_{extra}	v_i format	General solution
-2	$d = 0$ and condition [a]	-	$[[a_i, b_i, c_i],$ $[e_i, f_i, g_i]]$	$X = a_iU^2 + b_iU + c_i$ and $Y = e_iU^2 + f_i + g_i$ for $U \in \mathbb{Z}$
-1	$q = 0$, $n = 0$	-	-	X, Y are any integers
0	$d < 0$	-	$[x_i, y_i]$	$X = x_i$ and $Y = y_i$
	$d = \square > 0$,[b] some cases[c]			
1	$d = \boxtimes > 0$, $n \neq 0$	$M, [s_1, s_2]$ [d]	$[x_i, y_i]$ [d]	$\binom{X}{Y} = M^j \binom{x_i}{y_i} + \binom{s_1}{s_2}$ for $j \in \mathbb{Z}$
2	$d = \square > 0$,[b] some cases[c]	-	$[[s_i, t_i], [x_i, y_i]]$	$x = x_i + s_iU$, $y = y_i + t_iU$ for $U \in \mathbb{Z}$
	$d = 0$, and condition [e]			

[a] At least one of $A, B, C \neq 0$ and at least one of $D, E \neq 0$.
[b] $\square$ means square, and $\boxtimes$ means non-square.
[c] "Some cases" refers to if the translated equation has $n = 0$ or not.
[d] $M \in \mathrm{SL}(2, \mathbb{Z})$ and s_1, s_2 are *rational*; they need not be integral. Same for x_i, y_i.
[e] $A = B = C = 0$ or $D = E = 0$. In this case, $s_i = s_j$ and $t_i = t_j$ for all i, j in fact.

For `bqf_linearsolve`, let $q = [A, B, C, D, E, F]$, and let $\text{lin} = [U, V, W]$. If there are no solutions the method will return `gen_0`, and otherwise it will return a vector `v`, where

$$v = [[\text{type}, v_{extra}], v_1, v_2, \ldots, v_k].$$

The types are are

-2=quadratic, -1=plane, 0=finite, 1=positive, 2=linear.

Each (family of) solution(s) is given by a v_i, possibly with reference to the extra data.

Table 3: General solution for `bqf_linearsolve`

Type	v_{extra}	v_i format	General solution
-2	-	$[[x_1, x_2, x_3], [y_1, y_2, y_3], [z_1, z_2, z_3]]$	$X = x_1U^2 + x_2U + x_3$, $Y = y_1U^2 + y_2U + y_3$, $Z = z_1U^2 + z_2U + z_3$, for $U \in \mathbb{Z}$
-1	-	$[[a_1, a_2, a_3], [b_1, b_2, b_3], [c_1, c_2, c_3]]$ [a]	$X = a_1U + b_1V + c_1$ $Y = a_2U + b_2V + c_2$, $Z = a_3U + b_3V + c_3$, for $U, V \in \mathbb{Z}$
0	-	$[a_i, b_i, c_i]$	$X = a_i$, $Y = b_i$, and $Z = c_i$
1	$M, [s_1, s_2, s_3]$ [b]	$[a_i, b_i, c_i]$ [b]	$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M^j \begin{pmatrix} a_i \\ b_i \\ c_i \end{pmatrix} + \begin{pmatrix} s_1 \\ s_2 \\ s_3 \end{pmatrix}$ for $j \in \mathbb{Z}$
2	-	$[[a_1, a_2, a_3], [b_1, b_2, b_3]]$	$X = a_1U + b_1$, $Y = a_2U + b_2$, $Z = a_3U + b_3$, for $U \in \mathbb{Z}$

[a] In fact, $i = 1$ necessarily (there is one plane only).

[b] $M \in \mathrm{SL}(3, \mathbb{Z})$ and s_1, s_2, s_3 are *rational*; they need not be integral. Same for a_i, b_i, c_i.

3.7 Representation of integers by forms - methods

Name:	`GEN bqf_bigreps`
Input:	`GEN q, GEN n, long prec`
Input format:	`q=[A, B, C, D, E]` length 5 integer vector, `n` integer, `prec` the precision
Output format:	`gen_0` or `v=[[type, data], sol1, ...]`
Description:	Solves $AX^2+BXY+CY^2+DX+EY = n$, and returns ALL solutions. If no solutions returns `gen_0`; otherwise `v[1][1]` gives the format of the general solution in Table 2.

Name:	`GEN bqf_bigreps_typecheck`
Input:	`GEN q, GEN n, long prec`
Input format:	`q` length 5 integer vector, `n` integer, `prec` the precision
Output format:	`gen_0` or `v=[[type, data], sol1, ...]`
Description:	Checks that `q, n` have the correct type, and returns `bqf_bigreps(q, n, prec)`.

Name:	`GEN bqf_linearsolve`
Input:	`GEN q, GEN n1, GEN lin, GEN n2, long prec`
Input format:	`q=[A, B, C, D, E, F]` length 6 integer vector, `n1` an integer, `lin=[U, V, W]` length 3 integer vector, `n2` an integer, `prec` the precision
Output format:	`gen_0` or `v=[[type, data], sol1, ...]`
Description:	Solves $AX^2+BY^2+CZ^2+DXY+EXY+FYZ = n1$ and $UX+VY+WZ = n2$, and returns ALL solutions. If no solutions returns `gen_0`; otherwise `v[1][1]` gives the format of the general solution in Table 3.

Name:	`GEN bqf_linearsolve_typecheck`
Input:	`GEN q, GEN n1, GEN lin, GEN n2, long prec`
Input format:	`q` length 6 integer vector, `n1` an integer, `lin` length 3 integer vector, `n2` an integer, `prec` the precision
Output format:	`gen_0` or `v=[[type, data], sol1, ...]`
Description:	Checks that `q, n1, lin, n2` have the correct type, and returns `bqf_linearsolve(q, n1, lin, n2)`.

Name:	`GEN bqf_reps`
Input:	`GEN q, GEN n, int proper, int half, long prec`
Input format:	`q=[A, B, C]` length 3 integer vector, `n` integer, `proper=0, 1`, `half=0, 1`, `prec` the precision
Output format:	`gen_0` or `v=[[type, data], sol1, ...]`
Description:	Solves $AX^2 + BXY + CY^2 = n$, and returns ALL solutions. If no solutions returns `gen_0`; otherwise `v[1][1]` gives the format of the general solution in Table 1. If `proper=1` and the form is indefinite/definite, we only output solutions with $\gcd(x, y) = 1$ (otherwise, no restriction). If `half=1`, only outputs one of (the families corresponding to) (x, y) and $(-x, -y)$, and if `half=0` outputs both.

Name:	`GEN bqf_reps_typecheck`
Input:	`GEN q, GEN n, int proper, int half, long prec`
Input format:	`q` length 3 integer vector, `n` integer, `proper=0, 1`, `half=0, 1`, `prec` the precision
Output format:	`gen_0` or `v=[[type, data], sol1, ...]`
Description:	Checks that `q, n` have the correct type, and returns `bqf_reps(q, n, proper, half, prec)`.

Name:	`GEN dbqf_reps`
Input:	`GEN qred, GEN D, GEN n, int proper, int half`
Input format:	`qred=[q',M]` with `q'` a reduced PDBQF of discriminant `D` and `M` the transition matrix, `n` integer, `proper=0, 1`, `half=0, 1`
Output format:	gen_0 or `v=[[0], sol1, ...]`
Description:	Sub-method solving `bqf_reps` in the definite case. Useful when you want to call `bqf_reps` on the same `q` many times.

Name:	`GEN ibqf_reps`
Input:	`GEN qorb, GEN qautom, GEN D, GEN rootD, GEN n, int proper, int half`
Input format:	For `q` a PIBQF, `qorb` is the output of `iqbf_redorbit_posonly_tmat(q)` sorted with `bqf_compare_tmat`, `qautom` the automorph of `q` of discriminant `D`, `rootD` the positive real square root of `D`, `n` an integer, `proper=0, 1`, `half=0, 1`
Output format:	gen_0 or `[[0], [0, 0]]` or `v=[[1, M], sol1, ...]`
Description:	Sub-method solving `bqf_reps` in the indefinite case. Useful when you want to call `bqf_reps` on the same `q` many times.

Name:	`GEN sbqf_reps`
Input:	`GEN q, GEN D, GEN rootD, GEN n, int half`
Input format:	`q` a primitive BQF, of positive square discriminant `D`, `rootD` the positive (integer) square root of `D`, `n` an integer, `half=0, 1`
Output format:	gen_0 or `v=[[0/2], sol1, ...]`
Description:	Sub-method solving `bqf_reps` in the positive square discriminant case. Very minimal savings over `bqf_reps`.

Name:	`GEN zbqf_reps`
Input:	`GEN A, GEN B, GEN n, int half`
Input format:	`q` a primitive non-trivial BQF of discriminant zero expressed as $(AX+BY)^2$ with `A, B` coprime, `n` integer, `half=0, 1`
Output format:	gen_0 or `v=[[2], sol1, ...]`
Description:	Sub-method solving `bqf_reps` in the discriminant zero case. Useful when you want to call `bqf_reps` on the same `q` many times.

Name:	`GEN zbqf_bigreps`
Input:	`GEN q, GEN n`
Input format:	`q` length 5 primitive integral and non-trivial with `q[1], q[3]>0`, `n` integer
Output format:	gen_0 or `v=[[+/-2], sol1, ....]`
Description:	Sub-method solving `bqf_bigreps` in the discriminant zero case. Very minimal savings over `bqf_bigreps`.

In the following three methods, we have first primitivized `q` (length 5 integer vector of non-zero discriminant `d`), and computed `a, b` for which the substitutions $X=\frac{x+a}{d}$ and $Y=\frac{y+b}{d}$ yield homogenous

BQFs of discriminant `d`.

Name:	`GEN bqf_bigreps_createrveсfin`
Input:	`GEN newsols, GEN a, GEN b, GEN disc`
Input format:	`newsols` the output of `bqf_reps` applied to our translated form, `a, b` the integers used to translate, `disc` the discriminant
Output format:	gen_0 or `v=[[0], sol1, sol2, ...]`
Description:	Takes the solutions to the translated form of type 0=finite and picks out only the ones which return to being integral.

Name:	`GEN bqf_bigreps_createrveсpos`
Input:	`GEN newsols, GEN a, GEN b, GEN disc`
Input format:	`newsols` the output of `bqf_reps` applied to our translated form, `a, b` the integers used to translate, `disc` the discriminant
Output format:	gen_0 or `v=[[1, M, [s1, s2]], sol1, sol2, ...]`
Description:	Takes the solutions to the translated form of type 1=positive and picks out only the ones which return to being integral.

Name:	`GEN bqf_bigreps_createrveсlin`
Input:	`GEN newsols, GEN a, GEN b, GEN disc`
Input format:	`newsols` the output of `bqf_reps` applied to our translated form, `a, b` the integers used to translate, `disc` the discriminant
Output format:	gen_0 or `v=[[2], sol1, sol2, ...]`
Description:	Takes the solutions to the translated form of type 2=linear and picks out only the ones which return to being integral.

Name:	`GEN bqf_reps_all`
Input:	`GEN n`
Input format:	`n` integer
Output format:	gen_0 or `[[-1]]`
Description:	Solves `bqf_reps` for `q=0`.

Name:	`GEN bqf_reps_creatervec`
Input:	`glist *sols, glist *scale, llist *nsolslist, long *totnsols, long *count, int half`
Input format:	See C code
Output format:	Vector
Description:	This creates the return vector given the computed list of solutions to `bqf_reps`. This does not initialize the first component (the type), and is only useful internally to `bqf_reps`. The method is not stack clean, but the return is suitable for gerepileupto.

Name:	GEN bqf_reps_createrveс_proper
Input:	`glist *sols, long nsols, int half`
Input format:	See C code
Output format:	Vector
Description:	This creates the return vector given the computed list of solutions to `bqf_reps`. This does not initialize the first component (the type), and is only useful internally to `bqf_reps`. It differs to `bqf_reps_creatervec` in that it deals with the proper solutions only case, and is more efficient in this case. The method is not stack clean, but the return is suitable for gerepile-upto.

Name:	GEN bqf_reps_makeprimitive
Input:	`GEN q, GEN *n`
Input format:	BQF `q`, pointer to integer `n`
Output format:	gen_0 or primitive BQF
Description:	We divide through `q` and `n` by $\gcd(q)$, update `n`, and return the new `q`. If `n` is no longer an integer (hence no solutions to `bqf_reps`), we return NULL. This clutters the stack.

Name:	GEN bqf_reps_trivial
Input:	`void`
Input format:	-
Output format:	`[[0], [0, 0]]`
Description:	Returns the trivial solution set.

Name:	void bqf_reps_updatesolutions
Input:	`glist **sols, long *nsols, GEN *a, GEN *b`
Input format:	See C code
Output format:	-
Description:	This method adds a new solution to the `glist` of solutions, and updates the relevant fields. This does not clutter the stack, but is NOT gerepile safe as the vector is created after the components. This is OK for the internal purposes of `bqf_reps` as the conversion from `glist` to vector includes a copying.

Name:	void dbqf_reps_proper
Input:	`GEN qred, GEN D, GEN n, glist **sols, long *nsols, GEN f,` `int *terminate`
Input format:	See C code
Output format:	-
Description:	This solves the proper representation case of `dbqf_reps`, updating the solution `glist`. Internal function for `dbqf_reps`.

Name:	`void ibqf_reps_proper`
Input:	`GEN qorb, GEN D, GEN rootD, GEN n, glist **sols, long *nsols, GEN f, int *terminate`
Input format:	See C code
Output format:	-
Description:	This solves the proper representation case of `ibqf_reps`, updating the solution `glist`. Internal function for `dbqf_reps`.

The following 5 methods all deal with `bqf_linearsolve`. We take a `3x3` matrix `M` with inverse `Minv` such that the top row is equal to `lin`, and substitute in `[x;y;z]=M*[X;Y;Z]`. This new equation has solutions `x=n2` and `y, z` described by `yzsols`. The methods bump this back to solutions for `X, Y, Z`, depending on the nature of the `y, z` solutions.

Name:	`GEN bqf_linearsolve_zall`
Input:	`GEN yzsols, GEN n2, GEN Minv`
Input format:	As above
Output format:	`[[-1], [[a1, a2, a3], [b1, b2, b3], [c1, c2, c3]]]`
Description:	As above, where the type of `yzsols` is `-1`, i.e. anything.

Name:	`GEN bqf_linearsolve_zfin`
Input:	`GEN yzsols, GEN n2, GEN Minv`
Input format:	As above
Output format:	`[[0], sol1, ...]`
Description:	As above, where the type of `yzsols` is `0`, i.e. finite.

Name:	`GEN bqf_linearsolve_zlin`
Input:	`GEN yzsols, GEN n2, GEN Minv`
Input format:	As above
Output format:	`[[2], sol1, ...]`
Description:	As above, where the type of `yzsols` is `2`, i.e. linear.

Name:	`GEN bqf_linearsolve_zpos`
Input:	`GEN yzsols, GEN n2, GEN Minv, GEN M`
Input format:	As above
Output format:	`[[1, M, [s1, s2, s3]], sol1, ...]`
Description:	As above, where the type of `yzsols` is `1`, i.e. positive.

Name:	`GEN bqf_linearsolve_zquad`
Input:	`GEN yzsols, GEN n2, GEN Minv`
Input format:	As above
Output format:	`[[-2], sol1, ...]`
Description:	As above, where the type of `yzsols` is `-2`, i.e. quadratic.

3.8 Checking GP inputs

Most methods in the library are easily breakable by having bad inputs. When working in GP, we do not want to cause segmentation faults, so we define wrapper functions to check the inputs. The methods in this section are useful in making these wrapper functions.

Name:	`void bqf_check`
Input:	`GEN q`
Input format:	-
Output format:	-
Description:	Checks that `q` is a BQF, and produces an error if not. Useful for making sure that GP inputs do not break our PARI functions.

Name:	`GEN bqf_checkdisc`
Input:	`GEN q`
Input format:	-
Output format:	Integer
Description:	Checks that `q` is a BQF with non-square discriminant and produces an error if not, where we return the discriminant of `q` if it passes. Useful for making sure that GP inputs do not break our PARI functions.

Name:	`void intmatrix_check`
Input:	`GEN mtx`
Input format:	-
Output format:	-
Description:	Checks that `mtx` is a `2x2` integral matrix, and produces an error if not. Useful for making sure that GP inputs do not break our PARI functions.

4 Method declarations

Methods in this section are divided into subsections by the files, and into subsubsections by their general function. They will appear approximately alphabetically in each subsubsection, with the static methods always appearing at the bottom. Clicking on a method name will bring you to its full description in the previous sections.

4.1 c_base

4.1.1 Complex geometry

`GEN`	`crossratio`	`GEN a, GEN b, GEN c, GEN d`
`GEN`	`mat_eval`	`GEN M, GEN x`
`GEN`	`mat_eval_typecheck`	`GEN M, GEN x`

4.1.2 Infinity

`GEN`	`addoo`	`GEN a, GEN b`

GEN	divoo	GEN a, GEN b

4.1.3 Integer vectors

GEN	ZV_copy	GEN v
GEN	ZV_Z_divexact	GEN v, GEN y
GEN	ZV_Z_mul	GEN v, GEN x
int	ZV_equal	GEN v1, GEN v2

4.1.4 Linear equations and matrices

GEN	lin_intsolve	GEN A, GEN B, GEN n
GEN	lin_intsolve_typecheck	GEN A, GEN B, GEN n
GEN	mat3_complete	GEN A, GEN B, GEN C
GEN	mat3_complete_typecheck	GEN A, GEN B, GEN C

4.1.5 Lists

void	clist_free	clist *l, long length
void	clist_putafter	clist **head_ref, GEN new_data
void	clist_putbefore	clist **head_ref, GEN new_data
GEN	clist_togvec	clist *l, long length, int dir
void	glist_free	glist *l
GEN	glist_pop	glist **head_ref
void	glist_putstart	glist **head_ref, GEN new_data
GEN	glist_togvec	glist *l, long length, int dir
void	llist_free	llist *l
long	llist_pop	llist **head_ref
void	llist_putstart	llist **head_ref, long new_data
GEN	llist_togvec	llist *l, long length, int dir
GEN	llist_tovecsmall	llist *l, long length, int dir

4.1.6 Solving equations modulo n

GEN	sqmod	GEN x, GEN n, GEN fact
GEN	sqmod_typecheck	GEN x, GEN n
GEN	sqmod_ppower	GEN x, GEN p, long n, GEN p2n, int iscoprime

4.1.7 Time

void	printtime	void
char*	returntime	void

4.2 c_bqf

4.2.1 Discriminant methods

GEN	disclist	GEN D1, GEN D2, int fund, GEN cop
GEN	discprimeindex	GEN D, GEN facs
GEN	discprimeindex_typecheck	GEN D
GEN	fdisc	GEN D
GEN	fdisc_typecheck	GEN D
int	isdisc	GEN D
GEN	pell	GEN D
GEN	pell_typecheck	GEN D
GEN	posreg	GEN D, long prec
GEN	posreg_typecheck	GEN D, long prec
GEN	quadroot	GEN D
GEN	quadroot_typecheck	GEN D

4.2.2 Basic methods for binary quadratic forms

GEN	bqf_automorph_typecheck	GEN q
int	bqf_compare	void *data, GEN q1, GEN q2
int	bqf_compare_tmat	void *data, GEN d1, GEN d2
GEN	bqf_disc	GEN q
GEN	bqf_disc_typecheck	GEN q
GEN	bqf_isequiv	GEN q1, GEN q2, GEN rootD, int Dsign, int tmat
GEN	bqf_isequiv_set	GEN q, GEN S, GEN rootD, int Dsign, int tmat
GEN	bqf_isequiv_typecheck	GEN q1, GEN q2, int tmat, long prec
int	bqf_isreduced	GEN q, int Dsign
int	bqf_isreduced_typecheck	GEN q
GEN	bqf_random	GEN maxc, int type, int primitive
GEN	bqf_random_D	GEN maxc, GEN D
GEN	bqf_red	GEN q, GEN rootD, int Dsign, int tmat
GEN	bqf_red_typecheck	GEN q, int tmat, long prec
GEN	bqf_roots	GEN q, GEN D, GEN w
GEN	bqf_roots_typecheck	GEN q
GEN	bqf_trans	GEN q, GEN M
GEN	bqf_trans_typecheck	GEN q, GEN M
GEN	bqf_transL	GEN q, GEN n
GEN	bqf_transR	GEN q, GEN n
GEN	bqf_transS	GEN q
GEN	bqf_trans_coprime	GEN q, GEN n
GEN	bqf_trans_coprime_typecheck	GEN q, GEN n
GEN	ideal_tobqf	GEN numf, GEN ideal

4.2.3 Basic methods, but specialized

GEN	dbqf_automorph	GEN q, GEN D
GEN	dbqf_isequiv	GEN q1, GEN q2
long	dbqf_isequiv_set	GEN q, GEN S
GEN	dbqf_isequiv_set_tmat	GEN q, GEN S
GEN	dbqf_isequiv_tmat	GEN q1, GEN q2
GEN	dbqf_red	GEN q
GEN	dbqf_red_tmat	GEN q
GEN	ibqf_automorph_D	GEN q, GEN D
GEN	ibqf_automorph_pell	GEN q, GEN qpell
GEN	ibqf_isequiv	GEN q1, GEN q2, GEN rootD
long	ibqf_isequiv_set_byq	GEN q, GEN S, GEN rootD
long	ibqf_isequiv_set_byq_presorted	GEN qredsorted, GEN S, GEN rootD
GEN	ibqf_isequiv_set_byq_tmat	GEN q, GEN S, GEN rootD
GEN	ibqf_isequiv_set_byq_tmat_presorted	GEN qredsorted, GEN S, GEN rootD
long	ibqf_isequiv_set_byS	GEN q, GEN S, GEN rootD
long	ibqf_isequiv_set_byS_presorted	GEN q, GEN Sreds, GEN perm, GEN rootD
GEN	ibqf_isequiv_set_byS_tmat	GEN q, GEN S, GEN rootD
GEN	ibqf_isequiv_set_byS_tmat_presorted	GEN q, GEN Sreds, GEN perm, GEN rootD
GEN	ibqf_isequiv_tmat	GEN q1, GEN q2, GEN rootD
GEN	ibqf_red	GEN q, GEN rootD
GEN	ibqf_red_tmat	GEN q, GEN rootD
GEN	ibqf_red_pos	GEN q, GEN rootD
GEN	ibqf_red_pos_tmat	GEN q, GEN rootD

4.2.4 Basic methods for indefinite quadratic forms

int	ibqf_isrecip	GEN q, GEN rootD
int	ibqf_isrecip_typecheck	GEN q, long prec
GEN	ibqf_leftnbr	GEN q, GEN rootD
GEN	ibqf_leftnbr_tmat	GEN q, GEN rootD
GEN	ibqf_leftnbr_typecheck	GEN q, int tmat, long prec
GEN	ibqf_leftnbr_update	GEN qvec, GEN rootD
GEN	ibqf_redorbit	GEN q, GEN rootD
GEN	ibqf_redorbit_tmat	GEN q, GEN rootD
GEN	ibqf_redorbit_posonly	GEN q, GEN rootD
GEN	ibqf_redorbit_posonly_tmat	GEN q, GEN rootD
GEN	ibqf_redorbit_typecheck	GEN q, int tmat, int posonly, long prec
GEN	ibqf_rightnbr	GEN q, GEN rootD
GEN	ibqf_rightnbr_tmat	GEN q, GEN rootD
GEN	ibqf_rightnbr_typecheck	GEN q, int tmat, long prec

GEN	ibqf_rightnbr_update	GEN qvec, GEN rootD
GEN	ibqf_river	GEN q, GEN rootD
GEN	ibqf_river_positions	GEN q, GEN rootD
GEN	ibqf_river_positions_forms	GEN q, GEN rootD
GEN	ibqf_river_typecheck	GEN q, long prec
GEN	ibqf_riverforms	GEN q, GEN rootD
GEN	ibqf_riverforms_typecheck	GEN q, long prec
GEN	ibqf_symmetricarc	GEN q, GEN D, GEN rootD, GEN qpell, long prec
GEN	ibqf_symmetricarc_typecheck	GEN q, long prec
GEN	ibqf_toriver	GEN q, GEN rootD
GEN	ibqf_toriver_tmat	GEN q, GEN rootD
GEN	mat_toibqf	GEN M
GEN	mat_toibqf_typecheck	GEN M

4.2.5 Class group and composition of forms

GEN	bqf_comp	GEN q1, GEN q2
GEN	bqf_comp_red	GEN q1, GEN q2, GEN rootD, int Dsign
GEN	bqf_comp_typecheck	GEN q1, GEN q2, int tored, long prec
GEN	bqf_idelt	GEN D
GEN	bqf_ncgp	GEN D, long prec
GEN	bqf_ncgp_lexic	GEN D, long prec
GEN	bqf_pow	GEN q, GEN n
GEN	bqf_pow_red	GEN q, GEN n, GEN rootD, int Dsign
GEN	bqf_pow_typecheck	GEN q, GEN n, int tored, long prec
GEN	bqf_square	GEN q
GEN	bqf_square_red	GEN q, GEN rootD, int Dsign
GEN	bqf_square_typecheck	GEN q, int tored, long prec
GEN	bqf_ncgp_nonfundnarrow	GEN cgp, GEN D, GEN rootD

4.2.6 Representation of integers by forms

GEN	bqf_bigreps	GEN q, GEN n, long prec
GEN	bqf_bigreps_typecheck	GEN q, GEN n, long prec
GEN	bqf_linearsolve	GEN q, GEN n1, GEN lin, GEN n2, long prec
GEN	bqf_linearsolve_typecheck	GEN q, GEN n1, GEN lin, GEN n2, long prec
GEN	bqf_reps	GEN q, GEN n, int proper, int half, long prec
GEN	bqf_reps_typecheck	GEN q, GEN n, int proper, int half, long prec
GEN	dbqf_reps	GEN qred, GEN D, GEN n, int proper, int half
GEN	ibqf_reps	GEN qorb, GEN qautom, GEN D, GEN rootD, GEN n, int proper, int half

GEN	sbqf_reps	GEN q, GEN D, GEN rootD, GEN n, int half
GEN	zbqf_reps	GEN A, GEN B, GEN n, int half
GEN	zbqf_bigreps	GEN q, GEN n
GEN	bqf_bigreps_creatervecfin	GEN newsols, GEN a, GEN b, GEN disc
GEN	bqf_bigreps_creatervecpos	GEN newsols, GEN a, GEN b, GEN disc
GEN	bqf_bigreps_createrveclin	GEN newsols, GEN a, GEN b, GEN disc
GEN	bqf_reps_all	GEN n
GEN	bqf_reps_creatervec	glist *sols, glist *scale, llist *nsolslist, long *totnsols, long *count, int half
GEN	bqf_reps_creatervec_proper	glist *sols, long nsols, int half
GEN	bqf_reps_makeprimitive	GEN q, GEN *n
GEN	bqf_reps_trivial	void
void	bqf_reps_updatesolutions	glist **sols, long *nsols, GEN *a, GEN *b
void	dbqf_reps_proper	GEN qred, GEN D, GEN n, glist **sols, long *nsols, GEN f, int *terminate
void	ibqf_reps_proper	GEN qorb, GEN D, GEN rootD, GEN n, glist **sols, long *nsols, GEN f, int *terminate
GEN	bqf_linearsolve_zall	GEN yzsols, GEN n2, GEN Minv
GEN	bqf_linearsolve_zfin	GEN yzsols, GEN n2, GEN Minv
GEN	bqf_linearsolve_zlin	GEN yzsols, GEN n2, GEN Minv
GEN	bqf_linearsolve_zpos	GEN yzsols, GEN n2, GEN Minv, GEN M
GEN	bqf_linearsolve_zquad	GEN yzsols, GEN n2, GEN Minv

4.2.7 Checking GP inputs

void	bqf_check	GEN q
GEN	bqf_checkdisc	GEN q
void	intmatrix_check	GEN mtx

References

[The20] The PARI Group, Univ. Bordeaux. *PARI/GP version 2.11.3*, 2020. available from http://pari.math.u-bordeaux.fr/.