Normaliz 3.10.0

Winfried Bruns Max Horn Ulrich von der Ohe

Former Normaliz 3 team members: Tim Römer, Richard Sieg and Christof Söger

Normaliz 2 team member: Bogdan Ichim

https://normaliz.uos.de https://github.com/Normaliz

mailto:normaliz@uos.de

https://hub.docker.com/u/normaliz/dashboard/

https://mybinder.org/v2/gh/Normaliz/NormalizJupyter/master

Short reference: NmzShortRef.pdf

Contents

A. Mathematical background and terminology

For a coherent and thorough treatment of the mathematical background we refer the reader to [10].

A.1. Polyhedra, polytopes and cones

An *affine halfspace* of $\mathbb{R}^d$ is a subset given as

$$H_\lambda^+ = \{x : \lambda(x) \geq 0\},$$

where λ is an affine form, i.e., a non-constant map $\lambda : \mathbb{R}^d \to \mathbb{R}$, $\lambda(x) = \alpha_1 x_1 + \cdots + \alpha_d x_d + \beta$ with $\alpha_1, \ldots, \alpha_d, \beta \in \mathbb{R}$. If $\beta = 0$ and λ is therefore linear, then the halfspace is called *linear*. The halfspace is *rational* if λ is *rational*, i.e., has rational coordinates. If λ is rational, we can assume that it is even *integral*, i.e., has integral coordinates, and, moreover, that these are coprime. Then λ is uniquely determined by H_λ^+. Such integral forms are called *primitive*, and the same terminology applies to vectors.

Definition 1. A (rational) *polyhedron* P is the intersection of finitely many (rational) halfspaces. If it is bounded, then it is called a *polytope*. If all the halfspaces are linear, then P is a *cone*.

The *dimension* of P is the dimension of the smallest affine subspace $\mathrm{aff}(P)$ containing P.

A support hyperplane of P is an affine hyperplane H that intersects P, but only in such a way that H is contained in one of the two halfspaces determined by H. The intersection $H \cap P$ is called a *face* of P. It is a polyhedron (polytope, cone) itself. Faces of dimension 0 are called *vertices*, those of dimension 1 are called *edges* (in the case of cones *extreme rays*), and those of dimension $\dim(P) - 1$ are *facets*.

When we speak of *the* support hyperplanes of P, then we mean those intersecting P in a facet. Their halfspaces containing P cut out P from $\mathrm{aff}(P)$. If $\dim(P) = d$, then they are uniquely determined (up to a positive scalar).

The constraints by which Normaliz describes polyhedra are

(1) linear equations for $\mathrm{aff}(P)$ and
(2) linear inequalities (simply called support hyperplanes) cutting out P from $\mathrm{aff}(P)$.

In other words, the constraints are given by a linear system of equations and inequalities, and a polyhedron is nothing else than the solution set of a linear system of inequalities and equations. It can always be represented in the form

$$Ax \geq b, \qquad A \in \mathbb{R}^{m \times d}, b \in \mathbb{R}^m,$$

if we replace an equation by two inequalities.

A.2. Cones

The definition describes a cone by constraints. One can equivalently describe it by generators:

Theorem 2 (Minkowski-Weyl)**.** *The following are equivalent for $C \subset \mathbb{R}^d$;*

1. *C is a (rational) cone;*
2. *there exist finitely many (rational) vectors $x_1, \dots, x_n$ such that*

$$C = \{a_1x_1 + \cdots + a_nx_n : a_1, \dots, a_n \in \mathbb{R}_+\}.$$

By $\mathbb{R}_+$ we denote the set of nonnegative real numbers; $\mathbb{Q}_+$ and $\mathbb{Z}_+$ are defined in the same way.

The conversion between the description by constraints and that by generators is one of the basic tasks of Normaliz. It uses the *Fourier-Motzkin elimination*.

Let C_0 be the set of those $x \in C$ for which $-x \in C$ as well. It is the largest vector subspace contained in C. A cone is *pointed* if $C_0 = 0$. If a rational cone is pointed, then it has uniquely determined *extreme integral generators*. These are the primitive integral vectors spanning the extreme rays. These can also be defined with respect to a sublattice L of $\mathbb{Z}^d$, provided C is contained in $\mathbb{R}L$. If a cone is not pointed, then Normaliz computes the extreme rays of the pointed C/C_0 and lifts them to C. (Therefore they are only unique modulo C_0.)

The *dual cone* C^* is given by

$$C^* = \{\lambda \in (\mathbb{R}^d)^* : \lambda(x) \geq 0 \text{ for all } x \in C\}.$$

Under the identification $\mathbb{R}^d = (\mathbb{R}^d)^{**}$ one has $C^{**} = C$. Then one has

$$\dim C_0 + \dim C^* = d.$$

In particular, C is pointed if and only if C^* is full dimensional, and this is the criterion for pointedness used by Normaliz. Linear forms $\lambda_1, \dots, \lambda_n$ generate C^* if and only if C is the intersection of the halfspaces $H^+_{\lambda_i}$. Therefore the conversion from constraints to generators and its converse are the same task, except for the exchange of $\mathbb{R}^d$ and its dual space.

A.3. Polyhedra

In order to transfer the Minkowski-Weyl theorem to polyhedra it is useful to homogenize coordinates by embedding $\mathbb{R}^d$ as a hyperplane in $\mathbb{R}^{d+1}$, namely via

$$\kappa : \mathbb{R}^d \to \mathbb{R}^{d+1}, \qquad \kappa(x) = (x, 1).$$

If P is a (rational) polyhedron, then the closure of the union of the rays from 0 through the points of $\kappa(P)$ is a (rational) cone $C(P)$, called the *cone over* P. The intersection $C(P) \cap (\mathbb{R}^d \times \{0\})$ can be identified with the *recession* (or tail) *cone*

$$\operatorname{rec}(P) = \{x \in \mathbb{R}^d : y + x \in P \text{ for all } y \in P\}.$$

It is the cone of unbounded directions in P. The recession cone is pointed if and only if P has at least one bounded face, and this is the case if and only if it has a vertex.

The theorem of Minkowski-Weyl can then be generalized as follows:

Theorem 3 (Motzkin)**.** *The following are equivalent for a subset $P \neq \emptyset$ of $\mathbb{R}^d$:*

1. *P is a (rational) polyhedron;*
2. *$P = Q + C$ where Q is a (rational) polytope and C is a (rational) cone.*

If P has a vertex, then the smallest choice for Q is the convex hull of its vertices, and $C = \mathrm{rec}(P)$ is uniquely determined.

The *convex hull* of a subset $X \in \mathbb{R}^d$ is

$$\mathrm{conv}(X) = \{a_1x_1 + \cdots + a_nx_n : n \geq 1, x_1, \ldots, x_n \in X, a_1, \ldots, a_n \in \mathbb{R}_+, a_1 + \cdots + a_n = 1\}.$$

Clearly, P is a polytope if and only if $\mathrm{rec}(P) = \{0\}$, and the specialization to this case one obtains Minkowski's theorem: a subset P of $\mathbb{R}^d$ is a polytope if and only if it is the convex hull of a finite set. A *lattice polytope* is distinguished by having integral points as vertices.

Normaliz computes the recession cone and the polytope Q if P is defined by constraints. Conversely it finds the constraints if the vertices of Q and the generators of C are specified.

Suppose that P is given by a system

$$Ax \geq b, \qquad A \in \mathbb{R}^{m\times d},\ b \in \mathbb{R}^m,$$

of linear inequalities (equations are replaced by two inequalities). Then $C(P)$ is defined by the *homogenized system*

$$Ax - x_{d+1}b \geq 0$$

whereas the $\mathrm{rec}(P)$ is given by the *associated homogeneous system*

$$Ax \geq 0.$$

It is of course possible that P is empty if it is given by constraints since inhomogeneous systems of linear equations and inequalities may be unsolvable. By abuse of language we call the solution set of the associated homogeneous system the recession cone of the system.

Via the concept of dehomogenization, Normaliz allows for a more general approach. The *dehomogenization* is a linear form δ on $\mathbb{R}^{d+1}$. For a cone $\widetilde{C}$ in $\mathbb{R}^{d+1}$ and a dehomogenization δ, Normaliz computes the polyhedron $P = \{x \in \widetilde{C} : \delta(x) = 1\}$ and the recession cone $C = \{x \in \widetilde{C} : \delta(x) = 0\}$. In particular, this allows other choices of the homogenizing coordinate. (Often one chooses x_0, the first coordinate then.)

In the language of projective geometry, $\delta(x) = 0$ defines the hyperplane at infinity.

A.4. Affine monoids

An *affine monoid* M is a finitely generated submonoid of $\mathbb{Z}^d$ for some $d \geq 0$. This means: $0 \in M$, $M+M \subset M$, and there exist $x_1, \ldots, x_n$ such that

$$M = \{a_1x_1 + \cdots + a_nx_n : a_1, \ldots, a_n \in \mathbb{Z}_+\}.$$

We say that $x_1, \ldots, x_n$ is a *system of generators* of M. A monoid M is positive if $x \in M$ and $-x \in M$ implies $x = 0$. An element x in a positive monoid M is called *irreducible* if it has no decomposition $x = y + z$ with $y, z \in M$, $y, z \neq 0$. The *rank* of M is the rank of the subgroup $\mathrm{gp}(M)$ of $\mathbb{Z}^d$ generated by M. (Subgroups of $\mathbb{Z}^d$ are also called sublattices.) For certain aspects of monoid theory it is very useful (or even necessary) to introduce coefficients from a field K (or a more general commutative ring) and consider the monoid algebra $K[M]$.

Theorem 4 (van der Corput)**.** *Every positive affine monoid M has a unique minimal system of generators, given by its irreducible elements.*

We call the minimal system of generators the *Hilbert basis* of M. Normaliz computes Hilbert bases of a special type of affine monoid:

Theorem 5 (Gordan's lemma)**.** *Let $C \subset \mathbb{R}^d$ be a (pointed) rational cone and let $L \subset \mathbb{Z}^d$ be a sublattice. Then $C \cap L$ is a (positive) affine monoid.*

The monoids $M = C \cap L$ of the theorem have the pleasant property that the group of units M_0 (i.e., elements whose inverse also belongs to M) splits off as a direct summand. Therefore M/M_0 is a well-defied affine monoid. If M is not positive, then Normaliz computes a Hilbert basis of M/M_0 and lifts it to M.

Let $M \subset \mathbb{Z}^d$ be an affine monoid, and let $N \supset M$ be an overmonoid (not necessarily affine), for example a sublattice L of $\mathbb{Z}^d$ containing M.

Definition 6. The *integral closure* (or *saturation*) of M in N is the set

$$\widehat{M}_N = \{x \in N : kx \in M \text{ for some } k \in \mathbb{Z}, k > 0\}.$$

If $\widehat{M}_N = M$, one calls M *integrally closed* in N.

The integral closure $\overline{M}$ of M in $\mathrm{gp}(M)$ is its *normalization*. M is *normal* if $\overline{M} = M$.

The integral closure has a geometric description:

Theorem 7.

$$\widehat{M}_N = \mathrm{cone}(M) \cap N.$$

Combining the theorems, we can say that Normaliz computes integral closures of affine monoids in lattices, and the integral closures are themselves affine monoids as well. (More generally, $\widehat{M}_N$ is affine if M and N are affine.)

In order to specify the intersection $C \cap L$ by constraints we need a system of homogeneous inequalities for C. Every sublattice of $\mathbb{Z}^d$ can be written as the solution set of a combined system of homogeneous linear diophantine equations and a homogeneous system of congruences (this follows from the elementary divisor theorem). Thus $C \cap L$ is the solution set of a homogeneous linear diophantine system of inequalities, equations and congruences. Conversely, the solution set of every such system is a monoid of type $C \cap L$.

In the situation of Theorem 7, if $\mathrm{gp}(N)$ has finite rank as a $\mathrm{gp}(M)$-module, $\widehat{M}_N$ is even a finitely generated module over M. I.e., there exist finitely many elements $y_1, \ldots, y_m \in \widehat{M}_N$ such that $\widehat{M}_N = \bigcup_{i=1}^m M + y_i$. Normaliz computes a minimal system $y_1, \ldots, y_m$ and lists the nonzero y_i as a system of module generators of $\widehat{M}_N$ modulo M. We must introduce coefficients to make this precise: Normaliz computes a minimal system of generators of the $K[M]$-module $K[\widehat{M}_N]/K[M]$.

A.5. Lattice points in polyhedra

Let $P \subset \mathbb{R}^d$ be a rational polyhedron and $L \subset \mathbb{Z}^d$ be an *affine sublattice*, i.e., a subset $w + L_0$ where $w \in \mathbb{Z}^d$ and $L_0 \subset \mathbb{Z}^d$ is a sublattice. In order to investigate (and compute) $P \cap L$ one again uses homogenization: P is extended to $C(P)$ and L is extended to $\mathscr{L} = L_0 + \mathbb{Z}(w, 1)$. Then one computes $C(P) \cap \mathscr{L}$. Via this "bridge" one obtains the following inhomogeneous version of Gordan's lemma:

Theorem 8. *Let P be a rational polyhedron with vertices and $L = w + L_0$ an affine lattice as above. Set $\mathrm{rec}_L(P) = \mathrm{rec}(P) \cap L_0$. Then there exist $x_1, \ldots, x_m \in P \cap L$ such that*

$$P \cap L = \{(x_1 + \mathrm{rec}_L(P)) \cap \cdots \cap (x_m + \mathrm{rec}_L(P))\}.$$

If the union is irredundant, then $x_1, \ldots, x_m$ are uniquely determined.

The Hilbert basis of $\mathrm{rec}_L(P)$ is given by $\{x : (x, 0) \in \mathrm{Hilb}(C(P) \cap \mathscr{L})\}$ and the minimal system of generators can also be read off the Hilbert basis of $C(P) \cap \mathscr{L}$: it is given by those x for which $(x, 1)$ belongs to $\mathrm{Hilb}(C(P) \cap \mathscr{L})$. (Normaliz computes the Hilbert basis of $C(P) \cap L$ only at "levels" 0 and 1.)

We call $\mathrm{rec}_L(P)$ the *recession monoid* of P with respect to L (or L_0). It is justified to call $P \cap L$ a *module* over $\mathrm{rec}_L(P)$. In the light of the theorem, it is a finitely generated module, and it has a unique minimal system of generators.

After the introduction of coefficients from a field K, $\mathrm{rec}_L(P)$ is turned into an affine monoid algebra, and $N = P \cap L$ into a finitely generated torsionfree module over it. As such it has a well-defined *module rank* $\mathrm{mrank}(N)$, which is computed by Normaliz via the following combinatorial description: Let $x_1, \ldots, x_m$ be a system of generators of N as above; then $\mathrm{mrank}(N)$ is the cardinality of the set of residue classes of $x_1, \ldots, x_m$ modulo $\mathrm{rec}_L(P)$.

Clearly, to model $P \cap L$ we need linear diophantine systems of inequalities, equations and congruences which now will be inhomogeneous in general. Conversely, the set of solutions of such a system is of type $P \cap L$.

A.6. Hilbert series and multiplicity

Normaliz can compute the Hilbert series and the Hilbert (quasi)polynomial of a graded monoid. A *grading* of a monoid M is simply a homomorphism $\deg : M \to \mathbb{Z}^g$ where $\mathbb{Z}^g$ contains the degrees. The *Hilbert series* of M with respect to the grading is the formal Laurent series

$$H(t) = \sum_{u\in\mathbb{Z}^g} \#\{x \in M : \deg x = u\} t_1^{u_1} \cdots t_g^{u_g} = \sum_{x\in M} t^{\deg x},$$

provided all sets $\{x \in M : \deg x = u\}$ are finite. At the moment, Normaliz can only handle the case $g = 1$, and therefore we restrict ourselves to this case. We assume in the following that $\deg x > 0$ for all nonzero $x \in M$ and that there exists an $x \in \mathrm{gp}(M)$ such that $\deg x = 1$. (Normaliz always rescales the grading accordingly – as long as no module N is involved.) In the case of a nonpositive monoid, these conditions must hold for M/M_0, and its Hilbert series is considered as the Hilbert series of M.

The basic fact about $H(t)$ in the $\mathbb{Z}$-graded case is that it is the Laurent expansion of a rational function at the origin:

Theorem 9 (Hilbert, Serre; Ehrhart)**.** *Suppose that M is a normal positive affine monoid. Then*

$$H(t) = \frac{R(t)}{(1-t^e)^r}, \qquad R(t) \in \mathbb{Z}[t],$$

where r is the rank of M and e is the least common multiple of the degrees of the extreme integral generators of $\mathrm{cone}(M)$*. As a rational function, $H(t)$ has negative degree.*

The statement about the rationality of $H(t)$ holds under much more general hypotheses.

Usually one can find denominators for $H(t)$ of much lower degree than that in the theorem, and Normaliz tries to give a more economical presentation of $H(t)$ as a quotient of two polynomials. One should note that it is not clear what the most natural presentation of $H(t)$ is in general (when $e > 1$). We discuss this problem in [14, Section 4]. The examples 2.5 and 2.6.2, may serve as an illustration.

A rational cone C and a grading together define the rational polytope $Q = C \cap A_1$ where $A_1 = \{x : \deg x = 1\}$. In this sense the Hilbert series is nothing but the Ehrhart series of Q. The following description of the Hilbert function $H(M,k) = \#\{x \in M : \deg x = k\}$ is equivalent to the previous theorem:

Theorem 10. *There exists a quasipolynomial q with rational coefficients, degree* $\mathrm{rank}\, M - 1$ *and period π dividing e such that $H(M,k) = q(k)$ for all $q \geq 0$.*

The statement about the quasipolynomial means that there exist polynomials $q^{(j)}$, $j = 0,\ldots,\pi-1$, of degree $\mathrm{rank}\, M - 1$ such that

$$q(k) = q^{(j)}(k), \qquad j \equiv k \quad (\pi),$$

and

$$q^{(j)}(k) = q_0^{(j)} + q_1^{(j)}k + \cdots + q_{r-1}^{(j)}k^{r-1}, \qquad r = \operatorname{rank} M,$$

with coefficients $q_i^{(j)} \in \mathbb{Q}$. It is not hard to show that in the case of affine monoids all components have the same degree $r-1$ and the same leading coefficient:

$$q_{r-1} = \frac{\operatorname{vol}(Q)}{(r-1)!},$$

where vol is the lattice normalized volume of Q (a lattice simplex of smallest possible volume has volume 1). The *multiplicity* of M, denoted by $e(M)$ is $(r-1)!q_{r-1} = \operatorname{vol}(Q)$.

Suppose now that P is a rational polyhedron in $\mathbb{R}^d$, $L \subset \mathbb{Z}^d$ is an affine lattice, and we consider $N = P \cap L$ as a module over $M = \operatorname{rec}_L(P)$. Then we must give up the condition that deg takes the value 1 on $\operatorname{gp}(M)$ (see Section **??** for an example). But the Hilbert series

$$H_N(t) = \sum_{x \in N} t^{\deg x}$$

is well-defined, and the qualitative statement above about rationality remain valid. However, in general the quasipolynomial gives the correct value of the Hilbert function only for $k > r$ where r is the degree of the Hilbert series as a rational function. The multiplicity of N is given by

$$e(N) = \operatorname{mrank}(N)e(M).$$

where $\operatorname{mrank}(M)$ is the module rank of M.

Since N may have generators in negative degrees, Normaliz shifts the degrees into $\mathbb{Z}_+$ by subtracting a constant, called the *shift*. (The shift may also be positive.)

Above the multiplicity of M was defined under the assumption that $\operatorname{gp}(M)$ contains an element of degree 1. In the homogeneous situation where no module N comes into play, Normaliz achieves this extra condition by dividing the grading by the *grading denominator* so that we are effectively in the situation considered above, except in two situations: (i) the use of the grading denominator is blocked; (ii) when a module N is considered, it can easily happen that the grading restricted to the recession monoid M has a denominator $g > 1$, but there occur degrees in N that are not divisible by g. Let $\deg' = \deg/g$ and let $e'(M)$ be the multiplicity of M with respect to $\deg'$. Then

$$e(M) = \frac{e'(M)}{g^{r-1}}.$$

With this definition, $e(M)$ has the expected property as a dimension normed leading coefficient of the Hilbert quasipolynomial: if $q^{(j)}$ is a *nonzero* component of the quasipolynomial of M, then its leading coefficient satisfies

$$q_{r-1}^{(j)} = \frac{e(M)}{(r-1)!}.$$

This follows immediately from the substitution $k \mapsto k/g$ in the Hilbert function when we pass from $\deg'$ to deg: $H(M,k) = H'(M,k/g)$ if g divides k and $H(M,k) = 0$ otherwise. Also the

interpretation as a volume is consistent: $e(M)$ is the lattice normalized volume of the polytope $C \cap \{x : \deg x = 1\}$ (whereas $e'(M)$ is the lattice normalized volume of $C \cap \{x : \deg x = g\}$).

For the interpretation of the multiplicity $e(N) = \mathrm{mrank}(N)e(M)$ one must first split the module N into a direct sum where each summand bundles the elements whose degrees belong to a fixed residue class modulo g. Let $N^0, \ldots, N^{g-1}$ be these summands. Then $e(N^k)$ is the dimension normed constant leading coefficient of the Hilbert quasipolynomial of N^k for each k, and $e(N) = \sum_k e(N^k)$.

A.7. The class group

A normal affine monoid M has a well-defined divisor class group. It is naturally isomorphic to the divisor class group of $K[M]$ where K is a field (or any unique factorization domain); see [10, Section 4.F], and especially [10, Corollary 4.56]. The class group classifies the divisorial ideals up to isomorphism. It can be computed from the standard embedding that sends an element x of $\mathrm{gp}(M)$ to the vector $\sigma(x)$ where σ is the collection of support forms $\sigma_1, \ldots, \sigma_s$ of M: $\mathrm{Cl}(M) = \mathbb{Z}^s/\sigma(\mathrm{gp}(M))$. Finding this quotient amounts to an application of the Smith normal form to the matrix of σ.

A.8. Affine monoids and their defining ideals

In addition to [10], the reader may want to consult De Loera, Hmmecke nd Köppe [21] and Sturmfels [35] for the discussion of Markov and Gröbner bases in our context.

As soon as one wants to study affine monoids by generators and relations, the pure combinatorial treatment becomes cumbersome, since there are no exact sequences without coefficcients. (In monoid theory relations are given by congruences; see [10].) Therefore we start from a field a field K and a K-subalgebra A of a Laurent polsnomial ring $K[X_1^p m1, \ldots, X_n^{\pm}1]$ that is generated by finitely many monomials$M_1, \ldots, M_m$, i.e., power products of indeterminates and their inverses. Often we identify the monomials with their exponent vectors, sweitching from multiplicative to additive notation and back. The exponent vectors generate an affine monoid, and the corresponding monomials are a K-basis of A.

To study A by its relations, one takes a polynomial ring $P = K[Y_1, \ldots, Y_m]$ and the surjective K-algebra homomorphism $\varphi : P \to A$induced by the substitution $Y_i \mapsto M_i$. The kernel I of φ is the *defining ideal* of A (with respect to the generating system $M_1, \ldots, M_m$). It is generated by *binomials* $Y^{v^+} - Y^{v^-}$ where v^+ and v^- are vectors with m nonnegative integer entries, and for such a vector $v = (v_1, \ldots, v_m)$ we have set $Y^v = Y_1^{v_1} \cdots Y_m^{v_m}$. Since all variables $Y_1, \ldots, Y_m$ are nonzerodivisors modulo I, we only need to consider binomials ssuch thatat most one of v_i^+ and v_i^- is nonzero in computing I. So we restict our use of the term "inomial" by assuming that both monmials in them do not have a common factor. This restriction has tremendous computational advantages: a binomial (in our restricted sense) can be represented by the difference vector $v^+ - v^-$.

Ideals like I are called *toric ideals*. Computing I means to find a *Markov basis* of I, i.e., a

binomial sysdtem of generators, and for efficiency we may want a minimal Markov basis. It is not unique in general, but at least its cardinality is unique if $M_1,\dots,M_m$ generate a positive affine monoid. The name "Markov basis" is motivated by applications in algebraic statistics. For certain computations, for example a Hilbert series, one even needs a Gröbner basis of I.

Toric ideals are generalized by *lattice ideals* J: J is generated by binomials and have the property that no indeterminate is a zeroduvisor modulo J.

Markov and Gröbner bases of toric and lattice are combinatorial invariants. They are independent of the choice of the field K.

Normaliz uses a reimplementation of the project-and-lift algorithm of Hemmecke and Malkin [28] for the computation of Markov bases, realized by them in 4ti2 [1]. The project-and-lift algorithm is also explained in [21].

A.9. Affine monoids from binomial ideals

Let U be a subgroup of $\mathbb{Z}^n$. Then the natural image M of $\mathbb{Z}^n_+ \subset \mathbb{Z}^n$ in the abelian group $G = \mathbb{Z}^n/U$ is a submonoid of G. In general, G is not torsionfree, and therefore M may not be an affine monoid. However, the image N of M in the lattice $L = G/\text{torsion}(G)$ is an affine monoid. Given U, Normaliz chooses an embedding $L \hookrightarrow \mathbb{Z}^r$, $r = n - \text{rank}\, U$, such that N becomes a submonoid of $\mathbb{Z}^r_+$. In general there is no canonical choice for such an embedding, but one can always find one, provided N has no invertible element except 0.

The typical starting point is an ideal $J \subset K[X_1,\dots,X_n]$ generated by binomials

$$X_1^{a_1}\cdots X_n^{a_n} - X_1^{b_1}\cdots X_n^{b_n}.$$

The image of $K[X_1,\dots,X_n]$ in the residue class ring of the Laurent polynomial ring $S = K[X_1^{\pm 1},\dots,X_n^{\pm 1}]$ modulo the ideal JS is exactly the monoid algebra $K[M]$ of the monoid M above if we let U be the subgroup of $\mathbb{Z}^n$ generated by the differences

$$(a_1,\dots,a_n) - (b_1,\dots,b_n).$$

Ideals of type JS are called lattice ideals if they are prime. Since Normaliz automatically passes to $G/\text{torsion}(G)$, it replaces JS by the smallest lattice ideal containing it.

A.10. Local properties of affine monoid algebras

B. Annotated console output

Somewhat outdated, but not much has changed in the shown computations since 3.2.0.

B.1. Primal mode

With

```
./normaliz -ch example/A443
```

we get the following terminal output.

```
                                                    \.....|
                  Normaliz 3.2.0                     \....|
                                                      \...|
    (C) The Normaliz Team, University of Osnabrueck    \..|
                  January  2017                         \.|
                                                         \|
************************************************************
Command line: -ch example/A443
Compute: HilbertBasis HilbertSeries
************************************************************
starting primal algorithm with full triangulation ...
Roughness 1
Generators sorted by degree and lexicographically
Generators per degree:
1: 48
```

Self explanatory so far (see Section 7.3 for the definition of roughness). Now the generators are inserted.

```
Start simplex 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 22 25 26 27 28 31 34
37 38 39 40 43 46
```

Normaliz starts by searching linearly independent generators with indices as small as possible. They span the start simplex in the triangulation. The remaining generators are inserted successively. (If a generator does not increase the cone spanned by the previous ones, it is not listed, but this does not happen for `A443`.)

```
gen=17, 39 hyp, 4 simpl
```

We have now reached a cone with 39 support hyperplanes and the triangulation has 4 simplices so far. We omit some generators until something interesting happens:

```
gen=35, 667 hyp, 85 pyr, 13977 simpl
```

In view of the number of simplices in the triangulation and the number of support hyperplanes, Normaliz has decided to build pyramids and to store them for later triangulation.

```
gen=36, 723 hyp, 234 pyr, 14025 simpl
...
gen=48, 4948 hyp, 3541 pyr, 14856 simpl
```

All generators have been processed now. Fortunately our cone is pointed:

```
Pointed since graded
```

```
Select extreme rays via comparison ... done.
```

Normaliz knows two methods for finding the extreme rays. Instead of "comparison" you may see "rank". Now the stored pyramids must be triangulated. They may produce not only simplices, but also pyramids of higher level, and indeed they do so:

```
**************************************************
level 0 pyramids remaining: 3541
**************************************************
**************************************************
all pyramids on level 0 done!
**************************************************
level 1 pyramids remaining: 5935
**************************************************
**************************************************
all pyramids on level 1 done!
**************************************************
level 2 pyramids remaining: 1567
**************************************************
1180 pyramids remaining on level 2, evaluating 2503294 simplices
```

At this point the preset size of the evaluation buffer for simplices has been exceeded. Normaliz stops the processing of pyramids, and empties the buffer by evaluating the simplices.

```
||||||||||||||||||||||||||||||||||||||||||||||||||
2503294 simplices, 0 HB candidates accumulated.
**************************************************
all pyramids on level 2 done!
**************************************************
level 3 pyramids remaining: 100
**************************************************
**************************************************
all pyramids on level 3 done!
```

This is a small computation, and the computation of pyramids goes level by level without the necessity to return to a lower level. But in larger examples the buffer for level $n+1$ may be filled before level n is finished. Then it becomes necessary to go back. Some simplices remaining in the buffer are now evaluated:

```
evaluating 150978 simplices
||||||||||||||||||||||||||||||||||||||||||||||||||
2654272 simplices, 0 HB candidates accumulated.
Adding 1 denominator classes... done.
```

Since our generators form the Hilbert basis, we do not collect any further candidates. If all generators are in degree 1, we have only one denominator class in the Hilbert series, but otherwise there may be many. The collection of the Hilbert series in denominator classes reduces the computations of common denominators to a minimum.

```
Total number of pyramids = 14137, among them simplicial 2994
```

Some statistics of the pyramid decomposition.

```
------------------------------------------------------------
transforming data... done.
```

Our computation is finished.

A typical pair of lines that you will see for other examples is

```
auto-reduce 539511 candidates, degrees <= 1 3 7
reducing 30 candidates by 73521 reducers
```

It tells you that Normaliz has found a list of 539511 new candidates for the Hilbert basis, and this list is reduced against itself (auto-reduce). Then the 30 old candidates are reduced against the 73521 survivors of the auto-reduction.

B.2. Dual mode

Now we give an example of a computation in dual mode. It is started by the command

```
./normaliz -cid example/5x5
```

The option `i` is used to suppress the `HSOP` in the input file. The console output:

```
                                                    \.....|
                  Normaliz 3.2.0                     \....|
                                                      \...|
    (C) The Normaliz Team, University of Osnabrueck    \..|
                  January  2017                         \.|
                                                         \|
************************************************************
Command line: -cid example/5x5
Compute: DualMode
No inequalities specified in constraint mode, using non-negative orthant.
************************************************************
```

Indeed, we have used only equations as the input.

```
************************************************************
computing Hilbert basis ...
==========================================================
cut with halfspace 1 ...
Final sizes: Pos 1 Neg 1 Neutral 0
```

The cone is cut out from the space of solutions of the system of equations (in this case) by successive intersections with halfspaces defined by the inequalities. After such an intersection

we have the positive half space, the "neutral" hyperplane and the negative half space. The final sizes given are the numbers of Hilbert basis elements strictly in the positive half space, strictly in the negative half space, and in the hyperplane. This pattern is repeated until all hyperplanes have been used.

```
================================================
cut with halfspace 2 ...
Final sizes: Pos 1 Neg 1 Neutral 1
```

We leave out some hyperplanes ...

```
================================================
cut with halfspace 20 ...
auto-reduce 1159 candidates, degrees <= 13 27
Final sizes: Pos 138 Neg 239 Neutral 1592
================================================
cut with halfspace 21 ...
Positive: 1027  Negative: 367
..............................................
Final sizes: Pos 1094 Neg 369 Neutral 1019
```

Sometimes reduction takes some time, and then Normaliz may issue a message on "auto-reduction" organized by degree (chosen for the algorithm, not defined by the given grading). The line of dots is printed is the computation of new Hilbert basis candidates takes time, and Normaliz wants to show you that it is not sleeping. Normaliz shows you the number of positive and negative partners that must be pared produce offspring.

```
================================================
cut with halfspace 25 ...
Positive: 1856  Negative: 653
..............................................
auto-reduce 1899 candidates, degrees <= 19 39
Final sizes: Pos 1976 Neg 688 Neutral 2852
```

All hyperplanes have been taken care of.

```
Find extreme rays
Find relevant support hyperplanes
```

Well, in connection with the equations, some hyperplanes become superfluous. In the output file Normaliz will list a minimal set of support hyperplanes that together with the equations define the cone.

```
Hilbert basis 4828
```

The number of Hilbert basis elements computed is the sum of the last positive and neutral numbers.

```
Find degree 1 elements
```

The input file contains a grading.

```
transforming data... done.
```

Our example is finished.

The computation of the new Hilbert basis after the intersection with the new hyperplane proceeds in rounds, and there can be many rounds ... (not in the above example). Then you can see terminal output like

```
Round 100
Round 200
Round 300
Round 400
Round 500
```

C. Normaliz 2 input syntax

A Normaliz 2 input file contains a sequence of matrices. Comments or options are not allowed in it. A matrix has the format

```
<m>
<n>
<x_1>
...
<x_m>
<type>
```

where `<m>` denotes the number of rows, `<n>` is the number of columns and `<x_1>\dots<x_n>` are the rows with `<n>` entries each. All matrix types of Normaliz 3 are allowed (with Normaliz 3), also `grading` and `dehomogenization`. These vectors must be encoded as matrices with 1 row.

Note that algebraic polyhedra cannot be defined by input files in this format.

The optional output files with suffix `cst` are still in this format. Just create one and inspect it.

D. libnormaliz

The kernel of Normaliz is the C++ class library `libnormaliz`. It implements all the classes that are necessary for the computations. The central class is `Cone`. It realizes the communication with the calling program and starts the computations most of which are implemented in other classes. In the following we describe the class `Cone`; other classes of `libnormaliz` may follow in the future.

Of course, Normaliz itself is the prime example for the use of `libnormaliz`, but it is rather complicated because of the input and output it must handle. Therefore we have a added a simple example program at the end of this introduction.

libnormaliz defines its own name space. In the following we assume that

```
using namespace std;
using namespace libnormaliz;
```

have been declared. It is clear that opening these name spaces is dangerous. In this documentation we only do it to avoid constant repetition of std:: and libnormaliz::

D.1. The master header file

```
#include "libnormaliz/libnormaliz.h"
```

reads all installed header files of libnormaliz.

D.2. Optional packages and configuration

The file

```
#include "libnormaliz/lnmz_config.h"
```

is created and installed when Normaliz is built by the autotools scripts. It (un)defines the preprocessor variables that indicate the optional packages used in the build process. These are

```
ENFNORMALIZ   NMZ_NAUTY   NMZ_FLINT   NMZ_COCOA
```

with obvious interpretations (ENFNORMALIZ stands for e-antic).

D.3. Integer type as a template parameter

A cone can be constructed for two integer types, long long and mpz_class. (Also long is possible, but we disregard it in the following, since one should make sure that the integer type has at least 64 bits.) It is reasonable to choose mpz_class since the main computations will then be tried with long long and restarted with mpz_class if long long cannot store the results. This internal change of integer type is not possible if the cone is constructed for long long. (Nevertheless, the linear algebra routines can use mpz_class locally if intermediate results exceed long long; have a look into matrix.cpp.)

Internally the template parameter is called Integer. In the following we assume that the integer type has been fixed as follows:

```
typedef mpz_class Integer;
```

The internal passage from mpz_class to long long can be suppressed by

```
MyCone.deactivateChangeOfPrecision();
```

where we assume that MyCone has been constructed as described in the next section.

D.3.1. Alternative integer types

It is possible to use libnormaliz with other integer types than `mpz_class`, `long long`, `long` or `renf_elem_class` but we have tested only these types.

If you want to use other types, you probably have to implement some conversion functions which you can find in `integer.h` and `integer.cpp`. Namely the functions

```
bool libnormaliz::try_convert(TypeA, TypeB);
// converts TypeB to TypeA, returns false if not possible
```

where one type is your type and the other is `long`, `long long`, `mpz_class` and `nmz_float`. Additionally, if your type uses infinite precision (for example, it is some wrapper for GMP), you must also implement

```
template<> inline bool libnormaliz::using_GMP<YourType>() { return true; }
```

D.3.2. Decimal fractions and floating point numbers

libnormaliz has a type `nmz_float` (presently set to `double`) that allows the construction of cones from floating point data. These are are first converted into `mpq_class` by using the GMP constructor of `mpq_class`, and then denominators are cleared. (The input routine of Normaliz goes another way by reading the floating point input as decimal fractions.)

D.4. Construction of a cone

The construction requires the specification of input data consisting of one or more matrices and the input types they represent. In addition there is a constructor that takes a Normaliz input file.

The term "matrix" stands for

```
vector<vector<number> >
```

where predefined choices of number are `long long`, `mpz_class`, `mpq_class` and `nmz_float` (the latter representing `double`).

The available input types (from `input_type.h`) are defined as follows:

```
namespace Type {
enum InputType {
//
// homogeneous generators
//
polytope,
rees_algebra,
subspace,
cone,
cone_and_lattice,
lattice,
    saturation,
rational_lattice,
```

```
monoid,
//
// inhomogeneous generators
//
vertices,
offset,
rational_offset,
//
// homogeneous constraints
//
inequalities,
signs,
equations,
congruences,
excluded_faces,
//
// inhomogeneous constraints
//
inhom_equations,
inhom_inequalities,
strict_inequalities,
strict_signs,
inhom_congruences,
inhom_excluded_faces,
//
// linearforms
//
grading,
dehomogenization,
    //
// lattice ideals and friends
//
lattice_ideal,
toric_ideal,
normal_toric_ideal,
//
// special
open_facets,
projection_coordinates,
//
// precomputed data
//
support_hyperplanes,
extreme_rays,
maximal_subspace,
generated_lattice,
```

```
hilbert_basis_rec_cone,
//
// deprecated
//
integral_closure,
normalization,
polyhedron,
...
};
} //end namespace Type
```

The input types are explained in Section 4. (There are further input types used for debugging and tests.)

In certain environments it is not possible to use the enumeration. Therefore we provide a function that converts a string into the corresponding input type:

```
Type::InputType to_type(const string& type_string)
```

The types `grading`, `dehomogenization`, `signs`, `strict_signs`, `offset`and `open_facets` must be encoded as matrices with a single row. We come back to this point below.

The simplest constructor has the syntax

```
Cone<Integer>::Cone(InputType input_type, const vector< vector<Integer> >& Input)
```

and can be used as in the following example:

```
vector<vector <Integer> > Data = ...
Type::InputType type = cone;
Cone<Integer> MyCone = Cone<Integer>(type, Data);
```

For two and three pairs of type and matrix there are the constructors

```
Cone<Integer>::Cone(InputType type1, const vector< vector<Integer> >& Input1,
InputType type2, const vector< vector<Integer> >& Input2)

Cone<Integer>::Cone(InputType type1, const vector< vector<Integer> >& Input1,
InputType type2, const vector< vector<Integer> >& Input2,
InputType type3, const vector< vector<Integer> >& Input3)
```

If you have to combine more than three matrices, you can define a

```
map <InputType, vector< vector<Integer> > >
```

and use the constructor with syntax

```
Cone<Integer>::Cone(const map< InputType,
vector< vector<Integer> > >& multi_input_data)
```

The four constructors also exist in a variant that uses the `libnormaliz` type `Matrix<Integer>` instead of `vector< vector<Integer> >` (see `cone.h`).

For the input of rational numbers we have all constructors also in variants that use `mpq_class` for the

input matrix, for example

```
Cone<Integer>::Cone(InputType input_type, const vector< vector<mpq_class> >& Input)
```

etc.

Similarly, for the input of decimal fractions and floating point numbers we have all constructors also in variants that use `nmz_float` for the input matrix, for example

```
Cone<Integer>::Cone(InputType input_type, const vector< vector<nmz_float> >& Input)
```

etc.

Note that `rational_lattice` and `rational_offset` can only be used if the input data are given in class `mpq_class` or `nmz_float`.

For convenience we provide the function

```
vector<vector<T> > to_matrix<Integer>(vector<T> v)
```

in `matrix.h`. It returns a matrix whose first row is `v`. A typical example:

```
size_t dim = ...
vector<vector <Integer> > Data = ...
Type::InputType type = cone;
vector<Integer> total_degree(dim,1);
Type::InputType grad = grading;
Cone<Integer> MyCone = Cone<Integer>(type, Data,grad,to_matrix(total_degree));
```

There is a default constructor for cones,

```
Cone<Integer>::Cone()
```

D.4.1. Construction from an input file

One can construct a cone also from a Normaliz input file by

```
Cone<Integer>::Cone(const string project)
```

The constructor reads the file `<project>.in`. All options in the file and numerical parameters are disregarded. The polynomials if present are however forwarded to the cone.

D.5. Setting and changing additional data

These data must be given to the cone before starting the computation if they are needed. The numerical parameters have default values, and the grading can be set when the cone is constructed.

D.5.1. Polynomials

The polynomial needed for integrals and weighted Ehrhart series must be passed to the cone after construction:

```
void Cone<Integer>::setPolynomial(string poly)
```

Like the grading it can be changed later on. Then the results depending on the previous polynomial will be deleted.

Similarly polynomial constraints can be set:

```
void Cone<Integer>::setPolynomialEquations(const vector<string>& poly_equs)
void Cone<Integer>::setPolynomialInequalities(const vector<string>& poly_inequs)
```

En bloc setting is also possible:

```
void Cone<Integer>::setPolyParams(const map<PolyParam::Param, vector<string>>& poly_params)
```

Have a look at `cone.cpp`. The single `polynomial` must be disguised as the only member of a vector.

D.5.2. Grading

If your computation needs a grading, you should include it into the construction of the cone. However, especially in interactive use via PyNormaliz or other interfaces, it can be useful to add the grading if it was forgotten or to change it later on. The following function allows this:

```
void Cone<Integer>::resetGrading(const vector<Integer>& grading)
```

Note that it deletes all previously computed results that depend on the grading.

D.5.3. Projection coordinates

Similarly to `resetGrading` we have

```
void Cone<Integer>::resetProjectionCoords(const vector<Integer>& lf)
```

The entries of `lf` must be 0 or 1.

D.5.4. Numerical parameters

Some computations can be controlled by numerical parameters. They can be given to the cone en bloc or individually.

To set them individually, you can use the following functions:

```
void Cone<Integer>::setExpansionDegree(long degree)
void Cone<Integer>::setNrCoeffQuasiPol(long nr_coeff)
void Cone<Integer>::setFaceCodimBound(long bound)
void Cone<Integer>::setAutomCodimBoundVectors(long bound) // not yet used
void Cone<Integer>::setDecimalDigits(long digits)
void Cone<Integer>::setBlocksizeHollowTri(long block_size)
void Cone<Integer>::setGBDegreeBound(const long degree_bound)
void Cone<Integer>::setGBMinDegree(const long min_degree)
```

There common default value of degree is -1, signaling no expansion, all coefficients or no codimension bound etc. One can reset the values after a computation. Then they will delete the computation results that depend on them.

To set them en bloc you can use

```
void Cone<Integer>::setNumericalParams(const map <NumParam::Param, long >& num_params)
```

where NumParam::Param refers to

```
namespace NumParam {
enum Param {
expansion_degree,
nr_coeff_quasipol,
face_codim_bound,
autom_codim_bound_vectors,  // not yet used
block_size_hollow_tri,
decimal_digits,
not_a_num_param
};
} //end namespace NumParam
```

(see libnormaliz/input_type.h).

D.6. Modifying a cone after construction

Within some boundaries it is possible to change an already constructed cone (and lattice). To this end one can use the functions

```
void Cone<Integer>::modifyCone(const map<InputType, vector<vector<Integer> > >&
                                                        multi_add_input_const)
void Cone<Integer>::modifyCone(InputType input_type, const vector< vector<Integer> >& Input)
```

Similar to the cone constructor, it has variations for vector< vector<mpq_class> > and vector< vector<nmz_float> > for cones that are not of renf_elem_class. There are also versions with Matrix<...> .

The following input types are allowed (to be prefixed by Type::)

```
cone        vertices          subspace
equations   inhom_equations   inequalities   inhom_inequalities
```

Modifying the current cone C by *additional* generators (first row) means to extend C. Modifying it by *additional* constraints (second row) restricts C.

It is allowed to issue several modifyCone(...) at any time, but there are some restrictions:

(1) The inhomogeneous types are only allowed if the cone was constructed with inhomogeneous input.

(2) Normaliz cannot fall back behind the coordinate transformation that has been reached at the time of additional input. This implies: (i) Additional generators must satisfy the equations valid at the

time of addition. (They are automatically adapted to the congruences if there should be any.) (ii) Additional linear inequalities must vanish on the maximal subspace at the time of addition.

(3) `modifyCone` cannot be used if the cone was created with `rational_lattice` or `rational_offset`.

(4) Between two `compute(...)` several `modifyCone` are allowed. But they must be of the same category, either the types in the first line above (generators) or those in the second (constraints).

The last restriction are necessary to avoid ambiguities. If the cone constructor is used with generators and constraints simultaneously, then the *intersection* of the cones defined by the constraints on one side and the generators on the other side is computed. (The same applies to lattice data.) In contrast, the later addition of generators always leads to an *extension* of the existing cone. And: if both constraints and generators are added between two `compute`, should we first extend and then restrict, or the other way round? The two operations do not commute.

For flexibility both support hyperplanes and extrene rays are computed before the modification.

It may happen that a previously computed (or provided) grading gives a negative value on an added generator. In this case the grading is reset. In the inhomogeneous case, if the dehomogenization should give a negative value, a `BadInputException` is thrown.

D.7. Computations in a cone

Before starting a computation in a (previously constructed) cone, one must decide what should be computed and in which way it should be computed. The computation goals and algorithmic variants (see Section 5) are defined as follows (`cone_property.h`):

```
namespace ConeProperty {
enum Enum {
// matrix valued
START_ENUM_RANGE(FIRST_MATRIX),
ExtremeRays,
VerticesOfPolyhedron,
SupportHyperplanes,
HilbertBasis,
ModuleGenerators,
Deg1Elements,
LatticePoints,
ModuleGeneratorsOverOriginalMonoid,
ExcludedFaces,
OriginalMonoidGenerators,
MaximalSubspace,
Equations,
Congruences,
GroebnerBasis,
MarkovBasis,
Representations,
END_ENUM_RANGE(LAST_MATRIX),

START_ENUM_RANGE(FIRST_MATRIX_FLOAT),
```

```
SuppHypsFloat,
VerticesFloat,
ExtremeRaysFloat,
END_ENUM_RANGE(LAST_MATRIX_FLOAT),

// vector valued
START_ENUM_RANGE(FIRST_VECTOR),
Grading,
Dehomogenization,
WitnessNotIntegrallyClosed,
GeneratorOfInterior,
CoveringFace,
AxesScaling,
END_ENUM_RANGE(LAST_VECTOR),

// integer valued
START_ENUM_RANGE(FIRST_INTEGER),
TriangulationDetSum,
ReesPrimaryMultiplicity,
GradingDenom,
UnitGroupIndex,
InternalIndex,
END_ENUM_RANGE(LAST_INTEGER),

START_ENUM_RANGE(FIRST_GMP_INTEGER),
ExternalIndex = FIRST_GMP_INTEGER,
END_ENUM_RANGE(LAST_GMP_INTEGER),

// rational valued
START_ENUM_RANGE(FIRST_RATIONAL),
Multiplicity,
Volume,
Integral,
VirtualMultiplicity,
END_ENUM_RANGE(LAST_RATIONAL),

// field valued
START_ENUM_RANGE(FIRST_FIELD_ELEM),
RenfVolume,
END_ENUM_RANGE(LAST_FIELD_ELEM),

// floating point valued
START_ENUM_RANGE(FIRST_FLOAT),
EuclideanVolume,
EuclideanIntegral,
END_ENUM_RANGE(LAST_FLOAT),
```

```
// dimensions and cardinalities
START_ENUM_RANGE(FIRST_MACHINE_INTEGER),
TriangulationSize,
NumberLatticePoints,
RecessionRank,
AffineDim,
ModuleRank,
Rank,
EmbeddingDim,
CodimSingularLocus,
END_ENUM_RANGE(LAST_MACHINE_INTEGER),

// boolean valued
START_ENUM_RANGE(FIRST_BOOLEAN),
IsPointed,
IsDeg1ExtremeRays,
IsDeg1HilbertBasis,
IsIntegrallyClosed,
IsSerreR1,
IsLatticeIdealToric,
IsReesPrimary,
IsInhomogeneous,
IsGorenstein,
IsEmptySemiOpen,
//
// checking properties of already computed data
// (cannot be used as a computation goal)
//
IsTriangulationNested,
IsTriangulationPartial,
END_ENUM_RANGE(LAST_BOOLEAN),

// complex structures
START_ENUM_RANGE(FIRST_COMPLEX_STRUCTURE),
Triangulation,
UnimodularTriangulation,
LatticePointTriangulation,
AllGeneratorsTriangulation,
PlacingTriangulation,
PullingTriangulation,
StanleyDec,
InclusionExclusionData,
IntegerHull,
ProjectCone,
ConeDecomposition,
```

```
//
Automorphisms,
AmbientAutomorphisms,
CombinatorialAutomorphisms,
RationalAutomorphisms,
EuclideanAutomorphisms,
InputAutomorphisms,
//
HilbertSeries,
HilbertQuasiPolynomial,
EhrhartSeries,
EhrhartQuasiPolynomial,
WeightedEhrhartSeries,
WeightedEhrhartQuasiPolynomial,
//
FaceLattice,
FVector,
Incidence,
DualFVector,
DualIncidence,
SingularLocus,
//
Sublattice,
//
ClassGroup,
END_ENUM_RANGE(LAST_COMPLEX_STRUCTURE),

//
// integer type for computations
//
START_ENUM_RANGE(FIRST_PROPERTY),
BigInt,
//
// algorithmic variants
//
DefaultMode,
Approximate,
BottomDecomposition,
NoBottomDec,
DualMode,
PrimalMode,
Projection,
ProjectionFloat,
NoProjection,
Symmetrize,
NoSymmetrization,
```

```
NoSubdivision,
NoNestedTri,  // synonym for NoSubdivision
KeepOrder,
HSOP,
NoPeriodBound,
NoLLL,
NoRelax,
Descent,
NoDescent,
NoGradingDenom,
GradingIsPositive,
ExploitAutomsVectors, // not yet implemented
ExploitIsosMult,
StrictIsoTypeCheck,
SignedDec,
NoSignedDec,
FixedPrecision,
//
Dynamic,
Static,
    //
WritePreComp,
// Gröbner Basis
Lex,
RevLex,
DegLex,
// ONLY FOR INTERNAL CONTROL
//
...
END_ENUM_RANGE(LAST_PROPERTY),

EnumSize // this has to be the last entry, to get the number of entries in the enum

}; // remember to change also the string conversion function if you change this enum
}
```

The class `ConeProperties` is based on this enumeration. Its instantiations are essentially boolean vectors that can be accessed via the names in the enumeration. Instantiations of the class are used to set computation goals and algorithmic variants and to check whether the goals have been reached. The distinction between computation goals and algorithmic variants is not completely strict. See Section 5 for implications between some `ConeProperties`.

There exist versions of `compute` for up to 3 cone properties:

```
ConeProperties Cone<Integer>::compute(ConeProperty::Enum cp)

ConeProperties Cone<Integer>::compute(ConeProperty::Enum cp1,
                   ConeProperty::Enum cp2)
```

```
ConeProperties Cone<Integer>::compute(ConeProperty::Enum cp1,
                    ConeProperty::Enum cp2, ConeProperty::Enum cp3)
```

An example:

```
MyCone.compute(ConeProperty::HilbertBasis, ConeProperty::Multiplicity)
```

If you want to specify more than 3 cone properties, you can define an instance of `ConeProperties` yourself and call

```
ConeProperties Cone<Integer>::compute(ConeProperties ToCompute)
```

An example:

```
ConeProperties Wanted;
Wanted.set(ConeProperty::Triangulation, ConeProperty::HilbertBasis);
MyCone.compute(Wanted);
```

All `get...` functions that are listed in the next section, try to compute the data asked for if they have not yet been computed. Unless you are interested a single result, we recommend to use `compute` since the data asked for can then be computed in a single run. For example, if the Hilbert basis and the multiplicity are wanted, then it would be a bad idea to call `getHilbertBasis` and `getMultiplicity` consecutively. More importantly, however, there is no choice of an algorithmic variant if you use `get...` without `compute` beforehand.

It is possible that a computation goal is unreachable. If this can be recognized from the input, a `BadInputException` will be thrown. If it cannot be recognized from the input, and `DefaultMode` is not set, then `compute()` will throw a `NotComputableException` so that `compute()` cannot return a value. In the presence of `DefaultMode`, the returned `ConeProperties` are those that have not been computed.

Please inspect `cone_property.cpp` for the full list of methods implemented in the class `ConeProperties`. Here we only mention the constructors

```
ConeProperties::ConeProperties(ConeProperty::Enum p1)

ConeProperties::ConeProperties(ConeProperty::Enum p1, ConeProperty::Enum p2)

ConeProperties::ConeProperties(ConeProperty::Enum p1, ConeProperty::Enum p2,
                               ConeProperty::Enum p3)
```

and the functions

```
ConeProperties& ConeProperties::set(ConeProperty::Enum p1, bool value)

ConeProperties& ConeProperties::set(ConeProperty::Enum p1, ConeProperty::Enum p2)

bool ConeProperties::test(ConeProperty::Enum Property) const
```

A string can be converted to a cone property and conversely:

```
ConeProperty::Enum toConeProperty(const string&)
```

```
const string& toString(ConeProperty::Enum)
```

You can return the whole collection of reached computation goals via

```
const ConeProperties& Cone<Integer>::getIsComputed() const
```

D.8. Retrieving results

As remarked above, all `get...` functions that are listed below, try to compute the data asked for if they have not yet been computed. As also remarked above, it is often better to use `compute` first.

The functions that return a matrix encoded as `vector<vector<number> >` have variants that return a matrix encoded in the `libnormaliz` class `Matrix<number>`. These are not listed below; see `cone.h`.

Note that there are now functions that return results by type so that interfaces need not implement all the functions in this section. See D.8.27.

D.8.1. Checking computations

In order to check whether a computation goal has been reached, one can use

```
bool Cone<Integer>::isComputed(ConeProperty::Enum prop) const
```

for example

```
bool done=MyCone.isComputed(ConeProperty::HilbertBasis)
```

D.8.2. Rank, index and dimension

```
size_t Cone<Integer>::getEmbeddingDim()
size_t Cone<Integer>::getRank()
Integer Cone<Integer>::getInternalIndex()
Integer Cone<Integer>::getUnitGroupIndex()

size_t Cone<Integer>::getRecessionRank()
long Cone<Integer>::getAffineDim()
size_t Cone<Integer>::getModuleRank()
```

The *internal* index is only defined if original generators are defined. See Section D.8.16 for the external index.

The last three functions return values that are only well-defined after inhomogeneous computations.

D.8.3. Support hyperplanes and constraints

```
const vector< vector<Integer> >& Cone<Integer>::getSupportHyperplanes()
size_t Cone<Integer>::getNrSupportHyperplanes()
```

The first function returns the support hyperplanes of the (homogenized) cone. The second function returns the number of support hyperplanes. Similarly we have

```
const vector< vector<Integer> >& Cone<Integer>::getEquations()
size_t Cone<Integer>::getNrEquations()
const vector< vector<Integer> >& Cone<Integer>::getCongruences()
size_t Cone<Integer>::getNrCongruences()
```

Support hyperplanes can be returned in floating point format:

```
const vector< vector<nmz_float> >& Cone<Integer>::getSuppHypsFloat()
size_t Cone<Integer>::getNrSuppHypsFloat()
```

For these functions there also exist `Matrix`versions.

D.8.4. Extreme rays and vertices

```
const vector< vector<Integer> >& Cone<Integer>::getExtremeRays()
size_t Cone<Integer>::getNrExtremeRays()
const vector< vector<Integer> >& Cone<Integer>::getVerticesOfPolyhedron()
size_t Cone<Integer>::getNrVerticesOfPolyhedron()
```

In the inhomogeneous case the first function returns the extreme rays of the recession cone, and the second the vertices of the polyhedron. (Together they form the extreme rays of the homogenized cone.)

Vertices and extreme rays can be returned in floating point format:

```
const vector< vector<nmz_float> >& Cone<Integer>::getVerticesFloat()
const vector< vector<nmz_float> >& Cone<Integer>::getExtremeRaysFloat()
size_t Cone<Integer>::getNrVerticesFloat()
```

D.8.5. Generators

```
const vector< vector<Integer> >& Cone<Integer>::getOriginalMonoidGenerators()
size_t Cone<Integer>::getNrOriginalMonoidGenerators()
```

Note that original generators are not always defined. The system of generators of the cone that is used in the computations and its cardinality are returned by

```
const vector< vector<Integer> >& Cone<Integer>::getGenerators()
size_t Cone<Integer>::getNrGenerators()
```

D.8.6. Lattice points in polytopes and elements of degree 1

```
const vector< vector<Integer> >& Cone<Integer>::getDeg1Elements()
size_t Cone<Integer>::getNrDeg1Elements()
```

These functions apply to the homogeneous case. `getNrDeg1Elements()` returns the number of degree 1 elements if these have been computed and storedl, and if the degree 1 elements are not available, it forces their computation and storage, even if the number of these elements should be known from other computations.

In the inhomogeneous case replace `Deg1Elements` by `ModuleGenerators`; see below. (They are also computable in the unbounded case.) A uniform access is possible by

```
const vector< vector<Integer> >& Cone<Integer>::getLatticePoints()
size_t Cone<Integer>::getNrLatticePoints()
```

In addition, we have

```
size_t Cone<Integer>::getNumberLatticePoints()
```

There is an important difference between `getNrLatticePoints()` and `getNumberLatticePoints()`: the latter returns the number whenever it is known for some reason. If the number is not known, it forces only the counting of lattice points, not their storage.

D.8.7. Hilbert basis

In the nonpointed case we need the maximal linear subspace of the cone:

```
const vector< vector<Integer> >& Cone<Integer>::getMaximalSubspace()
size_t Cone<Integer>::getDimMaximalSubspace()
```

One of the prime results of Normaliz and its cardinality are returned by

```
const vector< vector<Integer> >& Cone<Integer>::getHilbertBasis()
size_t Cone<Integer>::getNrHilbertBasis()
```

Inhomogeneous case the functions refer to the the Hilbert basis of the recession cone. The module generators of the lattice points in the polyhedron are accessed by

```
const vector< vector<Integer> >& Cone<Integer>::getModuleGenerators()
size_t Cone<Integer>::getNrModuleGenerators()
```

If the original monoid is not integrally closed, you can ask for a witness:

```
vector<Integer> Cone<Integer>::getWitnessNotIntegrallyClosed()
```

D.8.8. Module generators over original monoid

```
const vector< vector<Integer> >&
               Cone<Integer>::getModuleGeneratorsOverOriginalMonoid()
size_t Cone<Integer>::getNrModuleGeneratorsOverOriginalMonoid()
```

D.8.9. Generator of the interior

If the monoid is Gorenstein, Normaliz computes the generator of the interior (the canonical module):

```
const vector<Integer>& Cone<Integer>::getGeneratorOfInterior()
```

Before asking for this vector, one should test `isGorenstein()`.

D.8.10. Grading and dehomogenization

```
vector<Integer> Cone<Integer>::getGrading()
Integer Cone<Integer>::getGradingDenom()
```

The second function returns the denominator of the grading.

```
vector<Integer> Cone<Integer>::getDehomogenization()
```

D.8.11. Enumerative data

```
mpq_class Cone<Integer>::getMultiplicity()
```

Don't forget that the multiplicity is measured for a rational, not necessarily integral polytope. Therefore it need not be an integer. The same applies to

```
mpq_class Cone<Integer>::getVolume()
nmz_float Cone<Integer>::getEuclideanVolume()
```

which can be computed for polytopes defined by homogeneous or inhomogeneous input. In the homogeneous case the volume is the multiplicity.

The Hilbert and Ehrhart series are stored in instances class `HilbertSeries`. They are retrieved by

```
const HilbertSeries& Cone<Integer>::getHilbertSeries()
const HilbertSeries& Cone<Integer>::getEhrhartSeries()
```

They contain several data fields that can be accessed as follows (see `hilbert_series.h`):

```
const vector<mpz_class>& HilbertSeries::getNum() const;
const map<long, denom_t>& HilbertSeries::getDenom() const;

const vector<mpz_class>& HilbertSeries::getCyclotomicNum() const;
const map<long, denom_t>& HilbertSeries::getCyclotomicDenom() const;

const vector<mpz_class>& HilbertSeries::getHSOPNum() const;
const map<long, denom_t>& HilbertSeries::getHSOPDenom() const;

long HilbertSeries::getDegreeAsRationalFunction() const;
long HilbertSeries::getShift() const;
```

```
bool HilbertSeries::isHilbertQuasiPolynomialComputed() const;
const vector< vector<mpz_class> >& HilbertSeries::getHilbertQuasiPolynomial() const;
long HilbertSeries::getPeriod() const;
mpz_class HilbertSeries::getHilbertQuasiPolynomialDenom() const;

vector<mpz_class> HilbertSeries::getExpansion() const;
```

The first six functions refer to three representations of the Hilbert series as a rational function in the variable t: the first has a denominator that is a product of polynomials $(1-t^g)^e$, the second has a denominator that is a product of cyclotomic polynomials. In the third case the denominator is determined by the degrees of a homogeneous system of parameters (see Section 2.5). In all cases the numerators are given by their coefficient vectors, and the denominators are lists of pairs (g,e) where in the second case g is the order of the cyclotomic polynomial.

If you have already computed the Hilbert series without HSOP and you want it with HSOP afterwards, the Hilbert series will simply be transformed, but Normaliz must compute the degrees for the denominator, and this may be a nontrivial computation.

The degree as a rational function is of course independent of the chosen representation, but may be negative, as well as the shift that indicates with which power of t the numerator tarts. Since the denominator has a nonzero constant term in all cases, this is exactly the smallest degree in which the Hilbert function has a nonzero value.

The Hilbert quasipolynomial is represented by a vector whose length is the period and whose entries are itself vectors that represent the coefficients of the individual polynomials corresponding to the residue classes modulo the period. These integers must be divided by the common denominator that is returned by the last function.

For the input type `rees_algebra` we provide

```
Integer Cone<Integer>::getReesPrimaryMultiplicity()
```

D.8.12. Weighted Ehrhart series and integrals

The weighted Ehrhart series can be accessed by

```
const pair<HilbertSeries, mpz_class>& Cone<Integer>::getWeightedEhrhartSeries()
```

The second component of the pair is the denominator of the coefficients in the series numerator. Its introduction was necessary since we wanted to keep integral coefficients for the numerator of a Hilbert series. The numerator and the denominator of the first component of type `HilbertSeries` can be accessed as usual, but one *must not forget the denominator of the numerator coefficients*, the second component of the return value. There is a second way to access these data; see below.

The virtual multiplicity and the integral, respectively, are got by

```
mpq_class Cone<Integer>::getVirtualMultiplicity()
mpq_class Cone<Integer>::getIntegral()
nmz_float Cone<Integer>::getEuclideanIntegral()
```

Actually the cone saves these data in a special container of class `IntegrationData` (defined in `Hilbert_series.h`).

It is accessed by

```
const IntegrationData& Cone<Integer>::getIntData()
```

The three get functions above are only shortcuts for the access via `getIntData()`:

```
string IntegrationData::getPolynomial() const
long IntegrationData::getDegreeOfPolynomial() const
bool IntegrationData::isPolynomialHomogeneous() const

const vector<mpz_class>& IntegrationData::getNum_ZZ() const
mpz_class IntegrationData::getNumeratorCommonDenom() const
const map<long, denom_t>& IntegrationData::getDenom() const

const vector<mpz_class>& IntegrationData::getCyclotomicNum_ZZ() const
const map<long, denom_t>& IntegrationData::getCyclotomicDenom() const

bool IntegrationData::isWeightedEhrhartQuasiPolynomialComputed() const
void IntegrationData::computeWeightedEhrhartQuasiPolynomial()
const vector< vector<mpz_class> >& IntegrationData::getWeightedEhrhartQuasiPolynomial()
mpz_class IntegrationData::getWeightedEhrhartQuasiPolynomialDenom() const

vector<mpz_class> IntegrationData::getExpansion() const

mpq_class IntegrationData::getVirtualMultiplicity() const
mpq_class IntegrationData::getIntegral() const
```

The first three functions refer to the polynomial defining the integral or weighted Ehrhart series. The function `getNumeratorCommonDenom()` returns the integer by which the coefficients of the numerator of the series must be divided.

The computation of these data is controlled by the corresponding `ConeProperty`. The expansion is always computed on-the-fly. Its values must be divided by the same number as the coefficients of the numerator.

D.8.13. Triangulation and disjoint decomposition

The last triangulation that has been explicitly computed is returned by

```
const pair<vector<SHORTSIMPLEX<Integer> >, Matrix<Integer> >&
                                                Cone<Integer>::getTriangulation()
```

If no triangulation has been computed yet, the basic triangulation is returned.

The `Matrix<Integer>` contains (a superset of) the vectors that generate the simplicial cones in the triangulation. The simplicial cones are represented by the `<vector<SHORTSIMPLEX<Integer> >`:

```
struct SHORTSIMPLEX {
vector<key_t> key;      // full key of simplex
Integer height;         // height of last vertex over opposite facet, used in Full_Cone
```

```
Integer vol;            // volume if computed, 0 else
Integer mult;           // used for renf_elem_class in Full_Cone
vector<bool> Excluded;  // for disjoint decomposition of cone
                        // true in position i indicates that the facet
                        // opposite of generator i must be excluded
};
```

The `key` specifies the generators of the simplicial cone by their row indices in the matrix (counted from 0). The component `vol` is the (absolute value) of their determinant, and `Excluded` is only set if `ConeDecomposition` was asked for.

For the refined triangulations one uses

```
const pair<vector<SHORTSIMPLEX<Integer> >, Matrix<Integer> >&
                    Cone<Integer>::getTriangulation(ConeProperty::Enum quality)
```

In which the parameter specifies the type of triangulation that is to be computed:

```
ConeProperty::Triangulation
ConeProperty::AllGeneratorsTriangulation
ConeProperty::LatticePointTriangulation
ConeProperty::UnimodularTriangulation
```

where the first choice returns the basic triangulation.

```
const pair<vector<SHORTSIMPLEX<Integer> >, Matrix<Integer> >&
                                          Cone<Integer>::getConeDecomposition()
```

has the same effect as `getTriangulation(ConeProperty::Triangulation)`, except that the components `Excluded` are definitely set.

Additional information on the possibly nested and /or partial triangulation that has been used for the computation in primal ode can be retrieved by

```
size_t Cone<Integer>::getTriangulationSize()
Integer Cone<Integer>::getTriangulationDetSum()
```

D.8.14. Stanley decomposition

The Stanley decomposition is stored in a list whose entries correspond to the simplicial cones in the triangulation:

```
const pair<list<STANLEYDATA<Integer> >, Matrix<Integer> > &  Cone<Integer>::getStanleyDec()
```

The `Matrix<Integer>` has the same meaning as for triangulations. `STANLEYDATA` defined as follows:

```
struct STANLEYDATA {
vector<key_t> key;
Matrix<Integer> offsets;
};
```

The key has the same interpretation as for the triangulation, namely as the vector of indices of the generators of the simplicial cone (counted from 0). The matrix contains the coordinate vectors of the offsets of the components of the decomposition that belong to the simplicial cone defined by the key. See Section 7.16 for the interpretation. The format of the matrix can be accessed by the following functions of class `Matrix<Integer>`:

```
size_t nr_of_rows() const
size_t nr_of_columns() const
```

The entries are accessed in the same way as those of `vector<vector<Integer> >`.

D.8.15. Scaling of axes

If `rational_lattice` or `rational_offset` are in the input for the cone, then the vector giving scaling of axes can be retrieved by

```
vector<Integer> Cone<Integer>::getAxesScaling()
```

The cone property `AxesScaling` cannot be used as a computation goal, but one can ask for its computation as usual.|

D.8.16. Coordinate transformation

The coordinate transformation from the ambient lattice to the sublattice generated by the Hilbert basis (whether it has been computed or not) can be returned as follows:

```
const Sublattice_Representation<Integer>& Cone<Integer>::getSublattice()
```

For algebraic polyhedra it defines the subspace generated by the (homogenized) cone.

An object of type `Sublattice_Representation` models a sequence of $\mathbb{Z}$-homomorphisms

$$\mathbb{Z}^r \xrightarrow{\varphi} \mathbb{Z}^n \xrightarrow{\pi} \mathbb{Z}^r$$

with the following property: there exists $c \in \mathbb{Z}$, $c \neq 0$, such that $\pi \circ \varphi = c \cdot \mathrm{id}_{\mathbb{Z}^r}$. In particular φ is injective. One should view the two maps as a pair of coordinate transformations: φ is determined by a choice of basis in the sublattice $U = \varphi(\mathbb{Z}^r)$, and it allows us to transfer vectors from $U \cong \mathbb{Z}^r$ to the ambient lattice $\mathbb{Z}^n$. The map π is used to realize vectors from U as linear combinations of the given basis of $U \cong \mathbb{Z}^r$: after the application of π one divides by c. (If U is a direct summand of $\mathbb{Z}^n$, one can choose $c = 1$, and conversely.) Normaliz considers vectors as rows of matrices. Therefore φ is given as an $r \times n$-matrix and π is given as an $n \times r$ matrix.

The data just described can be accessed as follows (`sublattice_representation.h`). For space reasons we omit the class specification `Sublattice_Representation<Integer>::`

```
const vector<vector<Integer> >& getEmbedding() const
const vector<vector<Integer> >& getProjection() const
Integer getAnnihilator() const
```

Here "Embedding" refers to φ and "Projection" to π (though π is not always surjective). The "Annihilator" is the number c above. (It annihilates $\mathbb{Z}^r$ modulo $\pi(\mathbb{Z}^n)$.)

The numbers n and r are accessed in this order by

```
size_t getDim() const
size_t getRank() const
```

The external index, namely the order of the torsion subgroup of $\mathbb{Z}^n/U$, is returned by

```
mpz_class getExternalIndex() const
```

Very often φ and ψ are identity maps, and this property can be tested by

```
bool IsIdentity() const
```

The constraints computed by Normaliz are "hidden" in the sublattice representation. They van be accessed by

```
const vector<vector<Integer> >& getEquations() const
const vector<vector<Integer> >& getCongruences() const
```

But see Section D.8.3 above for a more direct access.

D.8.17. Coordinate transformations for precomputed data

For precomputed data we need `Type::generated_lattice` and `Type::maximal_subspace`, should they be nontrivial. The maximal subspace is retrieved by

```
getMaximalSubspace()
```

mentioned already in Section D.8.7. The generated lattice (subspace in the algebraic case) is accessed by

```
getSublattice().getEmbedding()
```

introduced in Section D.8.16.

D.8.18. Automorphism groups

The automorphism group is accessed by

```
const AutomorphismGroup<Integer>& Cone<Integer>::getAutomorphismGroup();
```

independently of the type of the automorphism group (see below). Only one type of automorphism group can be computed in a run of `compute(...)` and this type is stored.

Contrary to other get functions, `getAutomorphismGroup()` does not trigger a computation since it is unclear what quality of automorphisms is asked for. If no automorphism group has been computed, a `BadInputException` is thrown.

Additionally we have

```
const AutomorphismGroup<Integer>&
            Cone<Integer>::getAutomorphismGroup(ConeProperty::Enum quality)
```

in which the quality can be specified. If the automorphism group has already been computed with a different quality, then it is recomputed.

If the automorphism group has been computed by those options that use extreme rays and support hyperplanes, i.e., all except `AmbientAutomorphisms` and `InputAutomorphisms`, then the action of the group is recorded in

```
mpz_class  getOrder() const;
const vector<vector<key_t> >&  getVerticesPerms() const
const vector<vector<key_t> > getExtremeRaysPerms() const
const vector<vector<key_t> > getSupportHyperplanesPrms() const

const vector<vector<key_t> > getVerticesOrbits() const
const vector<vector<key_t> > geExtremeRaystOrbits() const
const vector<vector<key_t> > getSupportHyperplanesOrbits() const
```

AQll these functions and the following ones belong to the class `AutomorphismGroup<Integer>`.

"Perms" is a shorthand for "permutations", and each generator of the automorphism group is represented by the permutation of the extreme rays that it induces. In the permutations, objects are counted from 0. The reference order of the vectors is the same as in the output files. The entry `[i][j]`| is the index of the object to which the j-th object is mapped by the i-th generator of the automorphism group.

The orbits are listed one by one: each `vector<key_t>` contains the indices that form an orbit, and the collection of orbits is given by the outer vector.

The action of `AmbientAutomorphisms` and `InputAutomorphisms` is documented in

```
const vector<vector<key_t> >& getGensPerms() const;
const vector<vector<key_t> >& getGensOrbits() const;
const vector<vector<key_t> >& getLinFormsPerms() const;
const vector<vector<key_t> >& getLinFormsOrbits() const;
```

where the ' Gens" are the input vectors representing generators of the primal cone or inequalities, given by linear forms generating the dual cone. "LinForms" are defined only for `AmbientAutomorphisms`, and they represent the coordinate linear forms. The generators from which the group has been computed are returned by

```
const Matrix<Integer>& getGens() const;
```

The qualities of the automorphisms is returned by

```
set<AutomParam::Quality> getQualities() const;
```

and the qualities are given by

```
namespace AutomParam {
enum Quality {
combinatorial,
rational,
integral,
euclidean,
ambient_gen,
```

```
ambient_ineq,
input_gen,
input_ineq,
algebraic,
graded // not used at present
};
...
```

Input and ambient automorphisms appear twice since Normaliz records what type of input is used for the computation, and this information is shown in the output files.

Another access is given by

```
string getQualitiesString()
```

and

```
string quality_to_string(AutomParam::Quality quality)
```

does a single conversion.

Moreover, you can ask

```
bool IsIntegralityChecked() const;
bool IsIntegral() const;
```

If you are interested in cycle decompositions, you can use

```
vector<vector<key_t> > cycle_decomposition(vector<key_t> perm, bool with_fixed_points)
```

where `with_fixed_points` decides whether cycles of length 1 are produced.

D.8.19. Class group

```
vector<Integer> Cone<Integer>::getClassGroup()
```

The return value is to be interpreted as follows: The entry for index 0 is the rank of the class group. The remaining entries contain the orders of the summands in a direct sum decomposition of the torsion subgroup.

D.8.20. Face lattice and f-vector

```
vector<size_t> Cone<Integer>::getFVector()
const map<dynamic_bitset,int>& Cone<Integer>::getFaceLattice()
```

Each element of the set represents a face F: the `int` is its codimension, and the `vector<bool>` v represents the facets containing F: $v[i] = 1$, if and only if the facet given by the i-th row of `getSupportHyperplanes()` contains F. (See Section 7.17.)

The incidence matrix can be accessed by

```
const vector<dynamic_bitset>& Cone<Integer>::getIncidence()
```

These functions have dual versions:

```
vector<size_t> Cone<Integer>::getDualFVector()
const map<dynamic_bitset,int>& Cone<Integer>::getDualFaceLattice()
const vector<dynamic_bitset>& Cone<Integer>::getDualIncidence()
```

D.8.21. Local properties

They are properties of localizations. So far we have

```
const map<dynamic_bitset, int>& Cone<Integer>::getSingularLocus()
size_t Cone<Integer>::getCodimSingularLocus()
bool Cone<Integer>::isSerreR1()
```

D.8.22. Markov abd Grobner bases, representations

```
const  vector<vector<Integer> >& Cone<Integer>::getMarkovBasis()
const  vector<vector<Integer> >& Cone<Integer>::getGroebnerBasis()
const  vector<vector<Integer> >& Cone<Integer>::getRepresentations()
```

For all three there are also the usual variants with `Matrix` and `Nr`. Note that they reurn only those binomials for Markov and Gröbner bases that satisfy the degree bounds set by `gb_degree_bound` and `gb_min_degree`.

D.8.23. Integer hull

For the computation of the integer hull an auxiliary cone is constructed. A reference to it is returned by

```
Cone<Integer>& Cone<Integer>::getIntegerHullCone() const
```

For example, the support hyperplanes of the integer hull can be accessed by

```
MyCone.getIntegerHullCone().getSupportHyperplanes()
```

D.8.24. Projection of the cone

Like the integer hull, the image of the projection is contained in an auxiliary cone that can be accessed by

```
Cone<Integer>& Cone<Integer>::getProjectCone() const
```

It contains constraints and extreme rays of the projection.

D.8.25. Excluded faces

Before using the excluded faces Normaliz makes the collection irredundant by discarding those that are contained in others. The irredundant collection (given by hyperplanes that intersect the cone in the faces) and its cardinality are returned by

```
const vector< vector<Integer> >& Cone<Integer>::getExcludedFaces()
size_t Cone<Integer>::getNrExcludedFaces()
```

For the computation of the Hilbert series the all intersections of the excluded faces are computed, and for each resulting face the weight with which it must be counted is computed. These data can be accessed by

```
const vector< pair<vector<key_t>,long> >& Cone<Integer>::getInclusionExclusionData()
```

The first component of each pair contains the indices of the generators (counted from 0) that lie in the face and the second component is the weight.

The emptiness of semiopen polyhedra can be tested by

```
bool Cone<Integer>::isEmptySemiOpen()
```

If the answer is positive, an excluded face making the semiopen polyhedron empty is returned by

```
vector<Integer> Cone<Integer>::getCoveringFace()
```

D.8.26. Boolean valued results

All the "questions" to the cone that can be asked by the boolean valued functions in this section start a computation if the answer is not yet known.

The first, the question

```
bool Cone<Integer>::isIntegrallyClosed()
```

does not trigger a computation of the full Hilbert basis. The computation stops as soon as the answer can be given, and this is the case when an element in the integral closure has been found that is not in the original monoid. Such an element is retrieved by

```
vector<Integer> Cone<Integer>::getWitnessNotIntegrallyClosed()
```

As discussed in Section 7.13.3 it can sometimes be useful to ask

```
bool Cone<Integer>::isPointed()
```

before a more complex computation is started.

The Gorenstein property can be tested with

```
bool Cone<Integer>::isGorenstein()
```

If the answer is positive, Normaliz computes the generator of the interior of the monoid. Also see D.8.9.

The next two functions answer the question whether the Hilbert basis or at least the extreme rays live

in degree 1.

```
bool Cone<Integer>::isDeg1ExtremeRays()
bool Cone<Integer>::isDeg1HilbertBasis()
```

Finally we have

```
bool Cone<Integer>::isInhomogeneous()
bool Cone<Integer>::isReesPrimary()
```

`isReesPrimary()` checks whether the ideal defining the Rees algebra is primary to the irrelevant maximal ideal.

D.8.27. Results by type

It is also possible to access (and compute if necessary) the output data of Normaliz by functions that only depend on the C++ type of the data:

```
const Matrix<Integer>& getMatrixConePropertyMatrix(ConeProperty::Enum property);
const vector< vector<Integer> >& getMatrixConeProperty(ConeProperty::Enum property);
const Matrix<nmz_float>& getFloatMatrixConePropertyMatrix(ConeProperty::Enum property);
const vector< vector<nmz_float> >& getFloatMatrixConeProperty(ConeProperty::Enum property);
vector<Integer> getVectorConeProperty(ConeProperty::Enum property);
Integer getIntegerConeProperty(ConeProperty::Enum property);
mpz_class getGMPIntegerConeProperty(ConeProperty::Enum property);
mpq_class getRationalConeProperty(ConeProperty::Enum property);
renf_elem_class getFieldElemConeProperty(ConeProperty::Enum property);
nmz_float getFloatConeProperty(ConeProperty::Enum property);
size_t getMachineIntegerConeProperty(ConeProperty::Enum property);
bool getBooleanConeProperty(ConeProperty::Enum property);
```

For example, `getMatrixConeProperty(ConeProperty::HilbertBasis)` will return the Hilbert basis as a `const vector< vector<Integer> >&`.

These functions make it easier to write interfaces to Normaliz since they need not to introduce new functions for results that have one of the types listed above.

It is clear that the complex results can only be accessed via their specialized "get" functions.

D.9. Algebraic polyhedra

Cones over algebraic number fields are constructed by

```
Cone<renf_elem_class>(...)
```

where `...` stands for all the variants that have been discussed in Section D.4, except that all matrices must be of type `vector<vector<renf_elem_class> >` or `Matrix<renf_elem_class>`. `Cone<renf_elem_class>(...)` is predefined in `libnormaliz`.

Note that not all integer, rational or float input types are allowed; see Section 8.

After the construction of the cone you must use

```
void Cone<renf_elem_class>::setRenf(renf_class* renf)
```

It is necessary to forward the information about the number field to derived cones. In the other direction:

```
renf_class* Cone<renf_elem_class>::getRenf()
```

Since version 1.0.0 the `renf_class*` is administrated through a `std::shared_ptr<const renf_class>`. It is returned by

```
const std::shared_ptr<const renf_class> Cone<Integer>::getRenfSharedPtr()
```

One can retrieve the minimal polynomial and the embedding by

```
vector<string> Cone<renf_elem_class>::getRenfData()
```

The name of the field generator is returned by

```
string Cone<renf_elem_class>::getRenfGenerator()
```

The computation follows the same rules that have been explained above, again with some restriction of the computation goals that can be reached. Again see Section 8.

In return values `Integer` must be specialized to `renf_elem_class`. A special return value is the volume that in general is no longer of type `mpq_class`. It is retrieved by

```
renf_elem_class Cone<renf_elem_class>::getRenfVolume()
```

The number field must be defined outside of libnormaliz. Have a look at `source/normaliz.cpp` and `source/input.in` to see the details.

The integer hull cone is of type `libnormaliz::Cone<renf_elem_class>`.

Remark: In the code, the template `Integer` does no longer stand for a truly integer type, but also for `renf_elem_class`, and thus for elements from a field.

D.10. Reusing previous computation results

To some extent it is possible to exploit the results of a previous computation after the modification of a cone (see Section D.6). This is controlled by

```
ConeProperty::Dynamic
ConeProperty::Static
```

where `Dynamic` activates this feature and `Static` deactivates it.

At present only results of previous convex hull computations or vertex enumerations can be reused. Restrictions:

(1) The coordinate transformation that had been reached before the previous computation must have remained unchanged. Note that a change may have happened as a consequence the previous computation. For example, the addition of inequalities can reduce the dimension.

(2) If a convex hull computation simultaneously creates a triangulation, then it must start from

scratch.

An example for the use of `ConeProperty::Dynamic` is given in `source/dynamic/dynamic.cpp`. It is compiled automatically by the `autotools` scripts, and can also be compiled in `source` by `make -f Makefile.classic dynamic`.

D.11. Control of execution

D.11.1. Exceptions

All exceptions that are thrown in `libnormaliz` are derived from the abstract class `NormalizException` that itself is derived from `std::exception`:

```
class NormalizException: public std::exception
```

The following exceptions must be caught by the calling program:

```
class ArithmeticException: public NormalizException
class BadInputException: public NormalizException
class NotComputableException: public NormalizException
class FatalException: public NormalizException
class NmzCoCoAException: public NormalizException
class InterruptException: public NormalizException
```

The `ArithmeticException` leaves `libnormaliz` if a nonrecoverable overflow occurs (it is also used internally for the change of integer type). This should not happen for cones of integer type `mpz_class`, unless it is caused by the attempt to create a data structure of illegal size or by a bug in the program. The `BadInputException` is thrown whenever the input is inconsistent; the reasons for this are manifold. The `NotComputableException` is thrown if a computation goal cannot be reached. The `FatalException` should never appear. It covers error situations that can only be caused by a bug in the program. At many places `libnormaliz` has `assert` verifications built in that serve the same purpose.

There are two more exceptions for the communication within `libnormaliz` that should not leave it:

```
class NonpointedException: public NormalizException
class NotIntegrallyClosedException: public NormalizException
```

The `InterruptException` is discussed in the next section.

D.11.2. Interruption

In order to find out if the user wants to interrupt the program, the functions in `libnormaliz` test the value of the global variable

```
volatile sig_atomic_t nmz_interrupted
```

If it is found to be `true`, an `InterruptException` is thrown. This interrupt leaves `libnormaliz`, so that the calling program can process it. The `Cone` still exists, and the data computed in it can still be accessed. Moreover, `compute` can again be applied to it.

The calling program must take care to catch the signal caused by Ctrl-C and to set `nmz_interrupted=1`.

D.11.3. Inner parallelization

By default the cone constructor sets the maximal number of parallel threads to 8, unless the system has set a lower limit. You can change this value by

```
long set_thread_limit(long t)
```

The function returns the previous value.

`set_thread_limit(0)` raises the limit set by libnormaliz to ∞.

D.11.4. Outer parallelization

The libnormaliz functions can be called by programs that are parallelized via OpenMP themselves. The functions in libnormaliz switch off nested parallelization.

As a test program you can compile and run `outerpar` in `source/outerpar`. Compile it by For the compilation of `maxsimplex.cpp` use

```
make -f Makefile.classic outerpar
```

in `source`.

D.11.5. Control of terminal output

By using

```
bool setVerboseDefault(bool v)
```

one can control the verbose output of `libnormaliz`. The default value is `false`. This is a global setting that effects all cones constructed afterwards. However, for every cone one can set an individual value of `verbose` by

```
bool Cone<Integer>::setVerbose(bool v)
```

Both functions return the previous value.In order to `setVerbose` for a cone, it must have already beren constructed, and during construction the global verbose determines terminalk output. The construction phase does some precomputations, and they may issue some inwanted terminal output. In order to suppress it, one can use

```
void suppressNextConstructorVerbose()
```

It sets the value of `verbose` to `false` for the next cone constructed.

The default values of verbose output and error output are `std::cout` and `std::cerr`. These values can be changed by

```
void setVerboseOutput(std::ostream&)
void setErrorOutput(std::ostream&)
```

D.11.6. Printing the cone

The function

```
void Cone<Integer>::write_cone_output(const string& output_file)
```

writes the standard `out` file using the content of `output_file` instead of the standard `<project>`. It is meant as a tool for debugging libraries. It is not possible to write any file with a suffix different from `out`.

We also have

```
void Cone<Integer>::write_precomp_for_input(const string& output_file)
```

It writes an input file with precomputed data (see Section 9.4). writes the file with suffix `precomp.in` file using the content of `output_file` instead of the standard `<project>`.

D.12. A simple program

The example program is a simplified version of the program on which the experiments for the paper "Quantum jumps of normal polytopes" by W. Bruns, J. Gubeladze and M. Michałek, Discrete Comput. Geom. 56 (2016), no. 1, 181–215, are based. Its goal is to find a maximal normal lattice polytope P in the following sense: there is no normal lattice polytope $Q \supset P$ that has exactly one more lattice point than P. "Normal" means in this context that the Hilbert basis of the cone over P is given by the lattice points of P, considered as degree 1 elements in the cone.

The program generates normal lattice simplices and checks them for maximality. The dimension is set in the program, as well as the bound for the random coordinates of the vertices.

Let us have a look at `source/maxsimplex/maxsimplex.cpp`. First the more or less standard preamble:

```
#include <cstdlib>
#include <vector>
#include <fstream>
#include <omp.h>
using namespace std;

#include "libnormaliz/libnormaliz.h"
```

Since we want to perform a high speed experiment which is not expected to be arithmetically demanding, we choose 64 bit integers:

```
typedef long long Integer;
```

The first routine finds a random normal simplex of dimension `dim`. The coordinates of the vertices are integers between 0 and `bound`. We are optimistic that such a simplex can be found, and this is indeed no problem in dimension 4 or 5.

```
Cone<Integer> rand_simplex(size_t dim, long bound){

    vector<vector<Integer> > vertices(dim+1,vector<Integer> (dim));
    while(true){ // an eternal loop ...
```

```
        for(size_t i=0;i<=dim;++i){
            for(size_t j=0;j<dim;++j)
                vertices[i][j]=rand()%(bound+1);
        }

        Cone<Integer> Simplex(Type::polytope,vertices);
        // we must check the rank and normality
        if(Simplex.getRank()==dim+1 && Simplex.isDeg1HilbertBasis())
            return Simplex;
    }
    vector<vector<Integer> > dummy_gen(1,vector<Integer>(1,1));
    // to make the compiler happy
    return Cone<Integer>(Type::cone,dummy_gen);
}
```

We are looking for a normal polytope $Q \supset P$ with exactly one more lattice point. The potential extra lattice points z are contained in the matrix `jump_cands`. There are two obstructions for $Q = \mathrm{conv}(P,z)$ to be tested: (i) z is the only extra lattice point and (ii) Q is normal. It makes sense to test them in this order since most of the time condition (i) is already violated and it is much faster to test.

```
bool exists_jump_over(Cone<Integer>& Polytope,
                      const vector<vector<Integer> >& jump_cands){

    vector<vector<Integer> > test_polytope=Polytope.getExtremeRays();
    test_polytope.resize(test_polytope.size()+1);
    for(size_t i=0;i<jump_cands.size();++i){
        test_polytope[test_polytope.size()-1]=jump_cands[i];
        Cone<Integer> TestCone(Type::cone,test_polytope);
        if(TestCone.getNrDeg1Elements()!=Polytope.getNrDeg1Elements()+1)
            continue;
        if(TestCone.isDeg1HilbertBasis())
            return true;
    }
    return false;
}
```

In order to make the (final) list of candidates z as above we must compute the widths of P over its support hyperplanes.

```
vector<Integer> lattice_widths(Cone<Integer>& Polytope){

    if(!Polytope.isDeg1ExtremeRays()){
        cerr<< "Cone in lattice_widths is not defined by lattice polytope"<< endl;
        exit(1);
    }
    vector<Integer> widths(Polytope.getNrExtremeRays(),0);
    for(size_t i=0;i<Polytope.getNrSupportHyperplanes();++i){
        for(size_t j=0;j<Polytope.getNrExtremeRays();++j){
```

```
            // v_scalar_product is a useful function from vector_operations.h
            Integer test=v_scalar_product(Polytope.getSupportHyperplanes()[i],
            Polytope.getExtremeRays()[j]);
            if(test>widths[i])
                widths[i]=test;
        }
    }
    return widths;
}
```

```
int main(int argc, char* argv[]){

    time_t ticks;
    srand(time(&ticks));
    cout << "Seed " <<ticks << endl;  // we may want to reproduce the run

    size_t polytope_dim=4;
    size_t cone_dim=polytope_dim+1;
    long bound=6;
    vector<Integer> grading(cone_dim,0);
           // at some points we need the explicit grading
    grading[polytope_dim]=1;

    size_t nr_simplex=0; // for the progress report
```

Since the computations are rather small, we suppress parallelization (except for one step below).

```
    while(true){

#ifdef _OPENMP
        omp_set_num_threads(1);
#endif
    Cone<Integer> Candidate=rand_simplex(polytope_dim,bound);
    nr_simplex++;
    if(nr_simplex%1000 ==0)
    cout << "simplex " << nr_simplex << endl;
```

Maximality is tested in 3 steps. Most often there exists a lattice point z of height 1 over P. If so, then $\mathrm{conv}(P,z)$ contains only z as an extra lattice point and it is automatically normal. In order to find such a point we must move the support hyperplanes outward by lattice distance 1.

```
    vector<vector<Integer> > supp_hyps_moved=Candidate.getSupportHyperplanes();
    for(size_t i=0;i<supp_hyps_moved.size();++i)
        supp_hyps_moved[i][polytope_dim]+=1;
    Cone<Integer> Candidate1(Type::inequalities,supp_hyps_moved,
    Type::grading,to_matrix(grading));
    if(Candidate1.getNrDeg1Elements()>Candidate.getNrDeg1Elements())
```

```
        continue;                       // there exists a point of height 1
```

Among the polytopes that have survived the height 1 test, most nevertheless have suitable points z close to them, and it makes sense not to use the maximum possible height immediately. Note that we must now test normality explicitly.

```
    cout << "No ht 1 jump"<< " #latt " << Candidate.getNrDeg1Elements() << endl;
    // move the hyperplanes further outward
    for(size_t i=0;i<supp_hyps_moved.size();++i)
        supp_hyps_moved[i][polytope_dim]+=polytope_dim;
    Cone<Integer> Candidate2(Type::inequalities,supp_hyps_moved,
                             Type::grading,to_matrix(grading));
    cout << "Testing " << Candidate2.getNrDeg1Elements()
         << " jump candidates" << endl; // including the lattice points in P
    if(exists_jump_over(Candidate,Candidate2.getDeg1Elements()))
            continue;
```

Now we can be optimistic that a maximal polytope P has been found, and we test all candidates z that satisfy the maximum possible bound on their lattice distance to P.

```
    cout << "No ht <= 1+dim jump" << endl;
    vector<Integer> widths=lattice_widths(Candidate);
    for(size_t i=0;i<supp_hyps_moved.size();++i)
            supp_hyps_moved[i][polytope_dim]+=
                            -polytope_dim+(widths[i])*(polytope_dim-2);
```

The computation may become arithmetically critical at this point. Therefore we use `mpz_class` for our cone. The conversion to and from `mpz_class` is done by routines contained in `convert.h`.

```
    vector<vector<mpz_class> > mpz_supp_hyps;
    convert(mpz_supp_hyps,supp_hyps_moved);
    vector<mpz_class> mpz_grading=convertTo<vector<mpz_class> >(grading);
```

The computation may need some time now. Therefore we allow a little bit of parallelization.

```
#ifdef _OPENMP
        omp_set_num_threads(4);
#endif
```

Since P doesn't have many vertices (even if we use these routines for more general polytopes than simplices), we don't expect too many vertices for the enlarged polytope. In this situation it makes sense to set the algorithmic variant `Approximate`.

```
    Cone<mpz_class> Candidate3(Type::inequalities,mpz_supp_hyps,
                               Type::grading,to_matrix(mpz_grading));
    Candidate3.compute(ConeProperty::Deg1Elements,ConeProperty::Approximate);
    vector<vector<Integer> > jumps_cand; // for conversion from mpz_class
    convert(jumps_cand,Candidate3.getDeg1Elements());
    cout << "Testing " << jumps_cand.size() << " jump candidates" << endl;
    if(exists_jump_over(Candidate, jumps_cand))
```

```
        continue;
```

Success!

```
    cout << "Maximal simplex found" << endl;
    cout << "Vertices" << endl;
    Candidate.getExtremeRaysMatrix().pretty_print(cout); // a goody from matrix.h
    cout << "Number of lattice points = " << Candidate.getNrDeg1Elements();
    cout << " Multiplicity = " << Candidate.getMultiplicity() << endl;

    } // end while
} // end main
```

For the compilation of `maxsimplex.cpp` use

```
make -f Makefile.classic maxsimplex
```

in `source`. Running the program needs a little bit of patience. However, within a few hours a maximal simplex should have emerged. From a log file:

```
simplex 143000
No ht 1 jump #latt 9
Testing 22 jump candidates
No ht 1 jump #latt 10
Testing 30 jump candidates
No ht 1 jump #latt 29
Testing 39 jump candidates
No ht <= 1+dim jump
Testing 173339 jump candidates
Maximal simplex found
Vertices
1 3 5 3 1
2 3 0 3 1
3 0 5 5 1
5 2 2 1 1
6 5 6 2 1
Number of lattice points = 29 Multiplicity = 275
```

E. Normaliz interactive: PyNormaliz

PyNormaliz serves three purposes:

- It is the bridge from Normaliz to SageMath.
- It provides an interactive access to Normaliz from a Python command line.
- It is a flexible environment for the exploration of Normaliz.

In the following we describe the use of PyNormaliz from a Python command line and document the basic functions that allow the access from SageMath.

For a brief introduction please consult the PyNormaliz tutorial at `https://nbviewer.jupyter.org/github/Normaliz/PyNormaliz/blob/main/doc/PyNormaliz_Tutorial.ipynb`.

You can also open the tutorial for PyNormaliz interactively on `https://mybinder.org` following the link `https://mybinder.org/v2/gh/Normaliz/NormalizJupyter/master`.

E.1. Installation

The PyNormaliz install script assumes that you have executed the

`install_normaliz_with_eantic.sh`

script. (It is however possible to install PyNormaliz with fewer optional packages.) In the following we assume that PyNormaliz resides in the subdirectory `PyNormaliz` of the Normaliz directory. This automatically the case if you have downloaded a Normaliz source package. If you have obtained Normaliz or PyNormaliz in another way, make sure that our assumption is satisfied.

To install PyNormaliz navigate to the Normaliz directory and type

```
./install_pynormaliz.sh --user
```

The script detects your Python3 version, assuming the executable is in the `PATH`. Note that the installation stores the produced files in `~/.local`.

If you want to install PyNormaliz system wide, replace `--user` by `--sudo`. Then you will be asked for your root password. The following additional options are available for `install_pynormaliz.sh`:

- `--python3 <path>`: Path to a python3 executable.
- `--prefix <path>`: Path to the Normaliz install path

Depending on your setup, you might be able to install PyNormaliz via pip, typing

```
pip3 install PyNormaliz
```

at a command prompt.

The installation requires the `setuptools`. If you are missing them install them with `pip3`.

E.2. The high level interface by examples

PyNormaliz has a high level interface which allows a very intuitive use. We load PyNormaliz:

```
winfried@ryzen:~$ python3
Python 3.6.9 (default, Oct  8 2020, 12:12:24)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import PyNormaliz
>>> from PyNormaliz import *
```

E.2.1. Creating a cone

The only available class in PyNormaliz is `Cone`. As often in this manual, "cone" includes a lattice of reference, unless we are working in an algebraic number field. We come back to this case below. First we have to create a cone (and a lattice). We can use all input types that are allowed in Normaliz input files. They must be given as named parameters as in the following example:

```
>>> C = Cone(cone = [[1,3],[2,1]])
```

This is the example from Section 2.3. There can be several input matrices. The example shows us how Normaliz matrices are represented as Python types: each row is a `list`, and the matrix then is a `list` whose members are the lists representing the rows. Important: This encoding matches exactly the formatted matrices in Normaliz input files.

It is possible to use (decimal) fractions in the input, but they must be encoded as strings. Our cone from above could be defined by

```
>>> C = Cone(cone = [[1,"3.0"],[1,"1/2"]])
```

This creates a `Cone<mpz_class>` on the Normaliz side. One can also create a `Cone<long long>` by

```
>>> C = Cone(cone = [[1,"3.0"],[1,"1/2"]], CreateAsLongLong = True)
```

In the following `Cone` (with a capital C) is a class defined in `PyNormaliz.py`. An instance of this class contains an `NmzCone` which is the Python equivalent of a `Cone<Integer>` defined on the Normaliz side. The `NmzCone` in the `Cone C`, is referred to by `C.cone`. This is only important when one wants to access the low level interface.

One can create a cone from a Normaliz input file as follows:

```
C = Cone(file = "example/small")
```

It will read the file `small.in` in the directory `example` /relative to the current directory. `CreateAsLongLong = True` can be used.

For polynomial constraints one uses commands like

```
PolyEq = ["x[1] -x[2]^2", "x[2]*x[3] - 27"]
C.SetPolynomialEquations(PolyEQ)
```

The argument of `SetPolynomialEquations` is a list of strings in which each component represents a polynomial expression. See Sections 4.1.8 and 4.9. The equations are always of type $f(x) = 0$. Similarly, inequalities defined by

```
C.SetPolynomialInequalities(PolyEQ)
```

are vinterpreted as $f(x) \geq 0$.

Selected input types:

- Homogeneous generators: `polytope, subspace, cone, cone_and_lattice, lattice monoid`
- Inhomogeneous generators: `vertices`
- Homogeneous constraints: `inequalities, signs, equations, congruences`
- Inhomogeneous constraints: `inhom_equations, inhom_inequalities, inhom_congruences`
- Linear forms: `grading, dehomogenization`
- Lattice ideals and friends: `lattice_ideal, toric_ideal, normal_toric_ideal`

For explanantions and other input types se the Normaliz manual. The input type `constraints` can't be used in PyNormaliz. Also shortcuts like `nonnegative` or `total_degree` are not available.

E.2.2. Matrices, vectors and numbers

The matrix format of the input is of course also used in PyNormaliz results:

```
>>> C.HilbertBasis()
[[1, 1], [1, 2], [1, 3], [2, 1]]
```

PyNormaliz contains some functions that help reading complicated output. For matrices we can use

```
>>> print_matrix(C.HilbertBasis())
1 1
1 2
1 3
2 1
```

Similarly

```
>>> print_matrix(C.SupportHyperplanes())
-1  2
3 -1
```

Since our input defines an original monoid, we can ask for the module generators over it:

```
>>> print_matrix(C.ModuleGeneratorsOverOriginalMonoid())
0 0
1 1
1 2
2 2
2 3
```

Binomials are retrieved in the same way:

```
>>> print_matrix(C.MarkovBasis())
-1  2 -1 0
-3  1  0 1
```

```
-2 -1  1 1
```

In this connection note that you can set upper and lower bounds for the degrees in the output of Markov and Gröbner bases:

```
C.SetGBDegreeBound(3)
C.SetGBMinDegree(2)
```

If you want to set a monomial order for the Gröbner basis, you must use the `Compute` function:

```
C.Compute("GroebnerBasis", "Lex")
C.GroebnerBasis()
```

Some numerical invariants:

```
>>> C.Rank()
2
>>> C.EmbeddingDim()
2
>>> C.ExternalIndex()
1
>>> C.InternalIndex()
5
```

If we want to know whether a certain cone property has already been computed, we can ask for it:

```
>>> C.IsComputed("HilbertBasis")
True
```

The essential point is that this query does *not* force the computation if the property has not yet been computed. There are several more computation goals that come as matrices, vectors or numbers. We list all of them:

- Matrices: `ExtremeRays, VerticesOfPolyhedron, SupportHyperplanes, HilbertBasis, ModuleGenerators, Deg1Elements, LatticePoints, ModuleGeneratorsOverOriginalMonoid, ExcludedFaces, OriginalMonoidGenerators, MaximalSubspace, Equations, Congruences, GroebnerBasis, Representations`
- Matrices with floating point entries: `ExtremeRaysFloat, SuppHypsFloat, VerticesFloat`
- Vectors: `Grading, Dehomogenization, WitnessNotIntegrallyClosed, GeneratorOfInterior, CoveringFace, AxesScaling`
- Numbers: `TriangulationSize, NumberLatticePoints, RecessionRank, AffineDim, ModuleRank, Rank, EmbeddingDim, ExternalIndex, TriangulationDetSum, GradingDenom, UnitGroupIndex, InternalIndex,`

The numbers have several different representations on the Normaliz side. In Python they are all (long) integers.

E.2.3. Triangulations, automorphisms and face lattice

Some of the raw output is complicated:

```
>>> U = C.UnimodularTriangulation()
>>> U
[[[[1, 2], 1, []], [[2, 3], 1, []], [[0, 3], 1, []]], [[1, 3], [2, 1], [1, 1], [1, 2]]]
```

Taking a close look, we see two members of the outermost `list`. The second is an ordinary matrix, namely the matrix of the rays of the triangulation:

```
>>> print_matrix(U[1])
1 3
2 1
1 1
1 2
```

The first member is not a matrix, but close enough so that we can use `print_matrix`:

```
>>> print_matrix(U[0])
[1, 2] 1 []
[2, 3] 1 []
[0, 3] 1 []
```

In each line we find the information on a simplicial cone, first the list of the rays by their indices relative to the matrix of rays (counting rows from 0). The next is the determinant relative to a lattice basis (in our case the unit vectors). In a unimodular triangulation these determinants must of course be 1. The third component is the list of excluded faces if we have computed a disjoint decomposition. This is explained in Section 7.14.2.

To see an even more complicated data structure we ask for the combinatorial automorphisms:

```
>>> G = C.CombinatorialAutomorphisms()
>>> G
[2, Faase, False,  [[[1, 0]], [[0, 1]]], [[], []], [[[1, 0]], [[0, 1]]]]
```

There are 6 components on the outermost level. The first is the order of the group. The second amswers the question whether the integrality of the automorphisms has been checked. The answer is always "no" for compinatorial automorphisms, and therefore the third give the answer "no" to the question whether the automorphisms are integral.

The next three contain information on the

- extreme rays of the (recession) cone,
- the vertices of the polyhedron,
- he support hyperplane

in this order. In each of them we find

- the action of the group generators on the respective vectors,
- their orbits under the group.

In our case there are no vertices of the polyhedron (only defined for inhomogeneous input). This explains the empty `list`. Fortunately we can print the complicated result nicely with an explanation:

```
>>> print_automs(G)
order  2
```

```
permutations of  extreme rays of (recession) cone
0 :  [1, 0]
orbits of  extreme rays of (recession) cone
0 :  [0, 1]
permutations of  support hyperplanes
0 :  [1, 0]
orbits of  support hyperplanes
0 :  [0, 1]
```

It makes sense to have a look at Section 7.22. (Here we count from 0.)

`AmbientAutomorphisms` and `InputAutomorphisms` yield a slightly different result. The permutations and orbits in the third element of the outer list now refer to the input vectors. The fourth element gives data for thempty set, as does the fifth for `InputAutomorphisms` . For `AmbientAutomorphisms` it lists the permutation and oprbits of the coordinates of the ambient lattice. All this is folloowed by the input vectors for reference. A simple example:

```
>>> C = Cone(cone = [[0,1],[1,0]])
>>> C.AmbientAutomorphisms()
[2, True, True, [[[1, 0]], [[0, 1]]], [[], []], [[[1, 0]], [[0, 1]]], [[0, 1], [1, 0]]]
>>> print_automs(C.AmbientAutomorphisms())
order  2
automorphisms are integral
permutations of  input vectors
0 :  [1, 0]
orbits of  input vectors
0 :  [0, 1]
permutations of  coordinates
0 :  [1, 0]
orbits of  coordinates
0 :  [0, 1]
input vectors
0 1
1 0
```

Of course, we also want to know the face lattice:

```
>>> C.FaceLattice()
[[[0, 0], 0], [[1, 0], 1], [[0, 1], 1], [[1, 1], 2]]
```

Hard to read. Much better:

```
>>> print_matrix(C.FaceLattice())
[0, 0] 0
[1, 0] 1
[0, 1] 1
[1, 1] 2
```

So there are four faces. The `list` contains the support hyperplanes that meet in the face and the number is the codimension. The support hyperplanes are given by their row indices relative to the matrix of

support hyperplanes. Also see Section 7.17. The f-vector:

```
>>> C.FVector()
[1, 2, 1]
```

If you want to limit the codimension of the faces computed with `FaceLattice` or `FVector`, set the bound by

```
>>> SetFaceCodimBound(1)
```

Try it and ask for `FaceLattice` once more. If you want to get rid of a previously set bound:

```
>>> SetFaceCodimBound()
```

or take -1 as the argument.

We also have a printer for the Stanley decomposition:

```
>>> print_Stanley_dec(C.StanleyDec())
```

Try it.

The cone properties that fall into the categories discussed in this section are: `Triangulation`, `UnimodularTriangulation`, `LatticePointTriangulation`, `AllGeneratorsTriangulation`, `PlacingTriangulation`, `PullingTriangulation`, `StanleyDec`, `InclusionExclusionData`, `Automorphisms`, `CombinatorialAutomorphisms`, `RationalAutomorphisms`, `EuclideanAutomorphisms`, `AmbientAutomorphisms`, `InputAutomorphisms`, `FaceLattice`, `DualFaceLattice`, `FVector`, `DualFVector`, `Incidence`, `DualIncidence`.

E.2.4. Hilbert and other series

Now we turn to the Hilbert series.

```
>>> C.HilbertSeries()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/home/winfried/../PyNormaliz.py", line 403, in inner
return self._generic_getter(name, **kwargs)
File "/home/winfried/.../PyNormaliz.py", line 393, in _generic_getter
PyNormaliz_cpp.NmzCompute(self.cone, input_list)
PyNormaliz_cpp.NormalizError: Could not compute:
No grading specified and cannot find one. Cannot compute some requested properties!
```

Indeed, we forgot the grading. We could have added it at the time of construction

```
>>> C = Cone(cone = [[1,3],[2,1]], grading = [[1,2]])
```

where it must be given as a matrix with a single row. Or we can add it later:

```
>>> C.SetGrading([1,2])
```

(A similar function is `SetProjectionCoords`.) We check the grading:

```
>>> C.Grading()
```

```
[[1, 2], 1]
```

The number 1 following the vector is the grading denominator.

Now:

```
>>> C.HilbertSeries()
[[1, -1, 0, 1, 0, 0, 0, 1, 0, -1, ..., 0, 0, 0, 0, 1, -1, 1], [1, 28], 0]
```

For space reasons we have omitted some components in the first `list`, the numerator of the Hilbert series. The second gives the denominator, and the last is the shift. Much nicer:

```
>>> print_series(C.HilbertSeries())
(1 -  t +  t^3 +  t^7 -  t^9 +  t^10 +  t^12 -  t^13 +  t^14 +  t^19 +  t^24 -  t^25 +  t^26)
-------------------------------------------------------------------------------------------
(1 - t) (1 - t^28)
```

Options can be added as named parameters:

```
>>> print_series(C.HilbertSeries(HSOP = True))
(1 +  t^3 +  t^5 +  t^6 +  t^8)
-------------------------------
(1 - t^4) (1 - t^7)
```

This representation is much more natural in this case. Perhaps we want so see the Hilbert quasipolynomial:

```
>>> print_quasipol(C.HilbertQuasiPolynomial())
28 5
-5 5
...
10 5
5 5
divide all coefficients by  28
```

In this case it seems better to print the polynomials as vectors of coefficients.

If the quasipolynomial has a large period and high degree, you may want to restrict the information to only a few coefficients from the top:

```
SetNrCoeffQuasiPol(bound)
```

The bound -1 or `SetNrCoeffQuasiPol()`mean "all", in case you want to get rid of the previously set bound.

Normaliz can compute the values of the coefficients of the Hilbert series for you:

```
>>> C.HilbertSeriesExpansion(10)
[1, 0, 0, 1, 1, 1, 1, 2, 2, 1, 2]
```

For the weighted Ehrhart series we need a polynomial. Let's add it (can also be done in the constructor with `polynomial = <string>`):

```
>>> C.SetPolynomial("x[1]+x[2]")
```

```
True
```

Then

```
print_series(C.WeightedEhrhartSeries())
```

We don't show the result because it is too long for this manual.

The cone properties of this section: `HilbertSeries, HilbertQuasiPolynomial, EhrhartSeries, EhrhartQuasiPolynomial, WeightedEhrhartSeries, WeightedEhrhartQuasiPolynomial`

E.2.5. Multiplicity, volume and integral

The first time we see a fraction printed as such:

```
>>> C.Multiplicity()
'5/28'
```

Since Python has no built-in type for fractions, we print it as a string.

```
>>> C.EuclideanVolume()
'0.3993'
```

The decimal fractions is rounded to 4 decimals. If you need more precision, you can directly use the low level interface:

```
>>> NmzResult(C.cone,"EuclideanVolume")
0.39929785312496247
```

By default, the low level interface returns raw values. We use it once more:

```
>>> NmzResult(C.cone,"EuclideanIntegral")
0.2638217958147073
```

We have integrated our polynomial from above. In case we have forgotten it:

```
>>> C.Polynomial()
'x[1]+x[2]'
```

For computations with fixed precision one can specify the number of decimal digits:

```
>>> C.setDecimalDigits(50)
```

This function is hardly necessary, since the default value of `100` is almost always satisfactory.

The cone properties of this section: `Multiplicity, Volume, Integral, VirtualMultiplicity, EuclideanVolume, EuclideanIntegral, ReesPrimaryMultiplicity`

E.2.6. Integer hull and other cones as values

Let us define a nonintegral polytope (we vary the format of the numbers on purpose):

```
>>> R = Cone(vertices = [["-3/2", '7/5',1], [9,-15,4], ["7.0",8,3]])
>>> R.VerticesOfPolyhedron()
[[-15, 14, 10], [7, 8, 3], [9, -15, 4]]
```

The last component of each vector acts as the denominator of the first two, and we recognize the fractions in the input. Numerical invariants available with inhomogeneous input:

```
>>> R.AffineDim()
2
>>> R.RecessionRank()
0
>>> R.LatticePoints()
[[-1, 1, 1], [0, 0, 1], [0, 1, 1], [1, -2, 1], ...  [2, -1, 1], [2, 0, 1], [2, 1, 1], [2, 2, 1]]
>>> H = R.IntegerHull()
>>> H
<Normaliz Cone>
```

So we have computed a new cone, the cone over the polytope (in this case) spanned by the lattice points in the polytope with rational vertices `[[-15, 14, 10], [7, 8, 3], [9, -15, 4]]`.

```
>>> H.VerticesOfPolyhedron()
[[-1, 1, 1], [1, -2, 1], [1, 2, 1], [2, -3, 1], [2, 2, 1]]
```

The last component is 1 as it must be for lattice points of the polytope.

```
>>> print_matrix(H.SupportHyperplanes())
-1  0 2
 0 -1 2
 1 -2 3
 1  1 1
 3  2 1
```

The other computations that return a cone are `ProjectCone` and `SymmetrizedCone`.

E.2.7. Boolean values

We ask our cone `C` many questions:

```
>>> C.IsGorenstein()
False
>>> C.IsDeg1HilbertBasis()
False
>>> C.IsDeg1ExtremeRays()
False
>>> C.IsPointed()
True
>>> C.IsInhomogeneous()
False
```

```
>>> C.IsEmptySemiOpen()
...
PyNormaliz_cpp.NormalizError: ...: IsEmptySemiOpen can only be computed with excluded faces
>>> C.IsIntegrallyClosed()
False
>>>
>>> C.IsReesPrimary()
...
PyNormaliz_cpp.NormalizError: Could not compute: IsReesPrimary !
```

E.2.8. Algebraic polyhedra

For an algebraic polyhedron we must define the real embedded number field over which the polyhedron is living. This information is given in the cone constructor:

```
>>> A = Cone(number_field=[ "a^2-2", "a", "1.4+/-0.1" ],
             vertices = [["1/2a", "13/3",1], ["-3a^1",-6,2], [-6, "-1/2a-7",1]])
>>> print_matrix(A.VerticesOfPolyhedron())
    -6 -1/2*a-7 1
-3/2*a       -3 1
 1/2*a     13/3 1
>>> print_matrix(A.VerticesFloat())
-6.0000 -7.7071 1.0000
-2.1213 -3.0000 1.0000
 0.7071  4.3333 1.0000
>>> A.RenfVolume()
'-19*a+42'
>>> A.EuclideanVolume()
'7.5650'
>>> print_matrix(A.LatticePoints())
-5 -6 1
...
-1  1 1
0  3 1
>>> A.NumberFieldData()
('a^2 - 2', '[1.414213562373095048801688724209698078569671875376948073176679738 +/- 3.57e-64]')
>>> A.GetFieldGeneratorName()
'a'
```

The only point to notice is `RenfVolume` that we must use instead of `Volume` here. The number field data show you to what precision $\sqrt{2}$ had to be computed to make all decisions about positivity for our little polytope.

E.2.9. The collective compute command and algorithmic variants

So far we have asked Normaliz for a single cone property. It is also possible to bundle several computation goals and options in a single compute command:

```
>>> C.Compute("HilbertBasis", "HilbertSeries", "ClassGroup", "DualMode")
True
>>> C.IsComputed("ClassGroup")
True
>>> C.ClassGroup()
[0, 5]
```

which means that the class group is isomorphic to $\mathbb{Z}/(5)$. The first number 0 indicates that the class group has rank 0.

The collective compute command not only allows you to sset several computation goals simultaneously. It allows you to specify algorithmic variants, like `DulaMode`. There is a whole collection of variants explained elsewhere in this manual:

`DefaultMode, Approximate, BottomDecomposition, NoBottomDec, DualMode, PrimalMode, Projection, ProjectionFloat, NoProjection, Symmetrize, NoSymmetrization, NoSubdivision, NoNestedTri, KeepOrder, HSOP, NoPeriodBound, NoLLL, NoRelax, Descent, NoDescent, NoGradingDenom, GradingIsPositive, ExploitAutomsVectors (not yet implemented), ExploitIsosMult, StrictIsoTypeCheck, SignedDec, NoSignedDec, FixedPrecision`

E.2.10. Miscellaneous functions

In order to get some information about what is going on in Normaliz, we can switch on the terminal output:

```
>>> C = Cone(cone = [[1,3],[2,1]], grading = [[1,2]])
>>> C.SetVerbose()
False
>>> C.HilbertBasis(DualMode = True)
Computing support hyperplanes for the dual mode:
************************************************************
starting full cone computation
Generators sorted lexicographically
Starting primal algorithm (only support hyperplanes) ...
Start simplex 1 2
Pointed since graded
Select extreme rays via comparison ... done.
------------------------------------------------------------
transforming data... done.
************************************************************
computing Hilbert basis ...
============================================
cut with halfspace 1 ...
Final sizes: Pos 1 Neg 1 Neutral 0
```

```
================================================
cut with halfspace 2 ...
Final sizes: Pos 3 Neg 3 Neutral 1
Hilbert basis 4
Find degree 1 elements
transforming data... done.
[[1, 1], [2, 1], [1, 2], [1, 3]]
```

The return value of `SetVerbose` is the *old value* of *verbose*. We had to redefine `C` to get of the already computed Hilbert basis. The very last line is our Hilbert basis.

If we want to see all data computed for `C`, call

```
>>> C.print_properties()
ExtremeRays:                        NumberLatticePoints:
[[2, 1], [1, 3]]                    0
SupportHyperplanes:                 Rank:
[[-1, 2], [3, -1]]                  2
HilbertBasis:                       EmbeddingDim:
[[1, 1], [2, 1], [1, 2], [1, 3]]    2
Deg1Elements:                       IsPointed:
[]                                  True
OriginalMonoidGenerators:           IsDeg1ExtremeRays:
[[1, 3], [2, 1]]                    False
MaximalSubspace:                    IsDeg1HilbertBasis:
[]                                  False
Grading:                            IsIntegrallyClosed:
[[1, 2], 1]                         False
GradingDenom:                       IsInhomogeneous:
1                                   False
UnitGroupIndex:                     Sublattice:
1                                   [[[1, 0], [0, 1]], [[1, 0], [0, 1]], 1]
InternalIndex:
```

Typeset in two columns. The last property we see is `Sublattice`. It consists of two matrices and a number. See Section D.8.16 for the interpretation.

Finally, we can write a Normaliz output file:

```
>>> C.WriteOutputFile("Wonderful")
True
```

Now you should find a file `Wonderful.out` in the current directory.

One can also write a file for the inout of precomputed data:

```
>>> C.WritePrecompData("Wonderful")
True
```

It creates the file `Wonderful.precomp.in`.

E.3. The low level interface

The low level interface is contained in `NormalizModule.cpp`. Its functions are listed in `PyNormaliz_cppMethods[]`. They allow the construction of an `NmzCone` (accompanied by a lattice), the computation in it, and give access to the computation results. The use of the low level interface is indirectly explained by the examples above. Therefore we keep the discussion short.

E.3.1. The main functions

For the construction one uses

```
NmzCone(**kwargs)
```

The keyword arguments `kwargs` transport Normaliz input types and the corresponding matrices in Python format. In addition we must use `number_field` for algebraic polyhedra. You can use `polynomial` for computations with a polynomial weight. (There is also an extra function for setting the polynomial; see below.) You can also ask for a `Cone<long long>` by adding `CreateAsLongLong = True`.

Once and for all: in the functions listed in the following that apply to a specific `NmzCone`, this `NmzCone` must be the first argument in `*args`.

Computations are started by

```
NmzCompute(*args)
```

The arguments list the computation goals and options as strings.

Access to the computation results is given by

```
NmzResult(*args, **kwargs)
```

There must be exactly two positional arguments. The first is the `NmzCone`, the second names the result to be returned, given as a string.

The `*kwargs` specify handlers, routines that format the raw results of output types that are not existent in Python or should be formatted for another reason. The potential handlers:

`RatHandler` defines the formatting of fractions.

`FloatHandler` defines the formatting of floating point numbers.

`NumberfieldElementHandler` defines the formatting of number field elements.

`VectorHandler` defines the formatting of vectors.

`MatrixHandler` defines the formatting of matrices.

The default handler for vectors and matrices is `list`, and there is not be much point in changing it. If you don't like lists, you can set `VectorHandler=tuple`, for example. But especially `RatHandler` and `NumberfieldElementHandler` are very useful since the raw versions are difficult to read. Examples of handlers can be found in `PyNormaliz.py`.

Note: When `NmzResult` is called, its first action is to reset the handlers to the raw format. Then the `kwargs` are evaluated. In other words: the values of the handlers are only applied to the current result, and not to future ones.

In the same way as the data access functions of Normaliz, `NmzResult` triggers the computation of the required result if it should not have been computed yet. Whether a result has been computed yet can be

checked by

```
NmzIsComputed(*args)
```

The second argument of exactly 2 is the result whose computation is to be checked, given as a string.

E.3.2. Additional input and modification of existing cones

These functions allow the input of data that cannot be passed through the cone constructor or modify a cone after construction. For example:

```
NmzSetGrading(cone, grading)
```

The grading is a vector encoded as a Python list. Similarly

```
NmzSetProjectionCoords(cone, coordinates)
```

where `coordinates` is a list with entries 0 or 1.

```
NmzSetPolynomial(cone, polynomial)
```

The polynomial is given as a string.

```
NmzSetNrCoeffQuasiPol(cone, number)
NmzSetFaceCodimBound(cone, number)
```

Do what the names say.

```
NmzModifyCone(cone, type, matrix)
```

This is the PyNormaliz version of the libnormaliz function modifyCone. Please have a look at Section D.6.

E.3.3. Additional data access

Some values cannot be returned as cone properties. For them we have additional access functions.

```
NmzGetPolynomial(cone)
```

returns the polynomial weight if one has been set.

The functions

```
NmzHilbertSeriesExpansion(cone, degree)
NmzEhrhartSeriesExpansion(cone, degree)
NmzWeightedEhrhartSeriesExpansion(cone, degree)
```

return the expansion of the named series up to the given degree as a list of numbers.

```
NmzIntegerHullCone(cone)
NmzProjectCone(cone)
NmzSymmetrizedCone(cone)
```

return NmzCone.

```
NmzGetRenfInfo(cone)
NmzFieldGenName(cone)
```

return the data defining the number field.

E.3.4. Miscellaneous functions

```
NmzSetVerbose(cone, value=True)
NmzSetVerboseDefault(value=True)
```

The first sets verbose to the specified value for cone, whereas the second sets it for all subsequently defined cones.

```
NmzConeCopy(cone)
```

returns a copy of cone.

```
NmzSetNumberOfNormalizThreads(number)
```

does what its name says. The previous number of threads is returned.

```
NmzWriteOutputFile(cone, project)
NmzWritePrecompData(cone, project)
```

The first writes a Normaliz output file whose name is the string project with the suffix .out, the second a file whose name is the string project with suffix precomp.in.

The functions

```
NmzHasEantic(cone)
NmzHasCoCoA(cone)
NmzHasFlint(cone)
NmzHasFlint(cone)
```

return True or False, depending on whether Normaliz has been built with the corresponding package.

```
NmzListConeProperties()
```

lists all cone properties in case you should have forgotten any of them.

```
error_out(PyObject* m)
```

writes an error message if something bad has happened.

E.3.5. Raw formats of numbers

All Normaliz integers are transformed to Python long integers, and floating point numbers are transformed to Python floats.

Numbers of type `mpq_class` are represented by a `list` with two components on the Python side, namely the numerator and the denominator.

An algebraic number is represented by a `list` whose members are rational numbers each of which is a `list` with two members. They are the coefficients of the polynomial representing the algebraic number.

F. Distributed computation for volume via signed decomposition

Normaliz offers a possibility to compute volumes via signed decomposition by distributing the task to several computers or nodes in a gigh performance cluster that run independently of each other. The principal approach:

(1) The first step is the computation of the hollow triangulation and the generic vector on a single machine. This step can require considerable time and memory.
(2) These data (and some more) are written to "hollow tri" files.
(3) The files are read by Normaliz with the `--Chunk` option that makes it compute the contribution to the volume that comes from a single data file ("chunk") and write this volume to a "mult" file.
(4) A final run of Normaliz with the `--AddChunks` option so that it reads all the "mult" files and adds the partial volumes.

This is certainly a robust and flexible approach to distributed computation. While the main purpose of distributed computation is a massive increase in parallelization, one should consider its use even if the computation is done on a single machine. It limits the loss of data caused by system crashes or similar interruptions to a small amount and allows easy repair. Another advantage is that the most time consuming step (3) needs very little RAM for a single "chunk", compared to step (1).

To make Normaliz write the data files and to stop once they have been written, one uses the cone property

`DistributedComp`

The size of the locks can be set by

`block_size_hollow_tri <n>`

to the input file where `<n>` is the number of simplices of the full triangulation that should go into a single output file. The default value chosen by `DistributedComp` is $500,000$.

The output files are

`<project>.hollow_tri.<n>.gz`

where `<n>` numbers these files consecutively, starting from 0. As usual, `<project>` is the name of the project. These files are gzipped to save disk space.

Moreover, there is a common data file:

`<project>.basic.data`

Each file of the hollow triangulation must be run by Normaliz with the option `--Chunk`. The input is read from `stdin` to which the gzipped file(s) must be decompressed and redirected or piped. The directory `source/chunk` contains `run_single.sh` hat can be used for this purpose:

```
time zcat $1.hollow_tri.$2.gz | ../normaliz --Chunk
```

where `$1` is the project name and `$2` is the number <n> from above. The OpenMP parallelization is set to 8 threads by this call, but one can add the option `-x=`<p> where <p> is the number of parallel threads to be used. Normaliz processes the single blocks with the fixed precision of 100 decimal digits. The path to normaliz (`../` above) must be adapted to your system.

On a cluster system one uses a script to start a job array where our number <n> serves as an index for the array. An example:

```
#SBATCH --job-name="CondEffPlur"
#SBATCH --comment="CondEffPlur"
#SBATCH --time=24:00:00
#SBATCH --ntasks=8
#SBATCH --threads-per-core=1
#SBATCH --mem=15000
#SBATCH --array=0-359%100

# each job will see a different ${SLURM_ARRAY_TASK_ID}
../run_single.sh CondEffPlur ${SLURM_ARRAY_TASK_ID}
```

In this example <n> runs from 0 to 359, and 100 jobs can be processed simultaneously.

Finally, execute

```
normaliz <project> --AddChunks
```

to sum the partial multiplicities in the files `<project>.mult.`<n> The result is written to the terminal and also to the file `<project>.total.mult`.

References

[1] 4ti2 team. 4ti2-A software package for algebraic, geometric and combinatorial problems on linear spaces. Available at https://github.com/4ti2/4ti2.

[2] J. Abbott, A. M. Bigatti and G. Lagorio, *CoCoA-5: a system for doing Computations in Commutative Algebra*. Available at http://cocoa.dima.unige.it.

[3] V. Almendra and B. Ichim, *jNormaliz 1.7*. Available at https://normaliz.uos.de.

[4] V. Baldoni, N. Berline, J. A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, M. Vergne and J. Wu, *A User's Guide for LattE integrale v1.7.2, 2013*. Software package LattE is available at https://www.math.ucdavis.edu/~latte/.

[5] D. Bremner, M. D. Sikirić, D. V. Pasechnik, Th. Rehn and A. Schürmann, *Computing symmetry groups of polyhedra*. LMS J. Comp. Math. 17 (2014), 565–581.

[6] St. Brumme, *Hash library*. Package available at https://create.stephan-brumme.com/.

[7] W. Bruns, *Automorphism groups and normal forms in Normaliz.* Res. Math. Sci. 9 (2022), no. 2, Paper No. 20, 15 pp.

[8] W. Bruns, *Polytope volume in Normaliz.* São Paulo J. Math. Sci. https://doi.org/10.1007/s40863-022-00317-9

[9] W. Bruns, P. Garcia-Sanchez, C. O'Neill and D. Wilburne, *Wilf's conjecture in fixed multiplicity*. Int. J. Algebra Comp. 30 (2020), 861–882.

[10] W. Bruns and J. Gubeladze, *Polytopes, rings, and K-theory*. Springer, 2009.

[11] W. Bruns, R. Hemmecke, B. Ichim, M. Köppe and C. Söger, *Challenging computations of Hilbert bases of cones associated with algebraic statistics*. Exp. Math. 20 (2011), 25–33.

[12] W. Bruns and B. Ichim, *Normaliz: algorithms for rational cones and affine monoids*. J. Algebra 324 (2010) 1098–1113.

[13] W. Bruns and B. Ichim, *Polytope volume by descent in the face lattice and applications in social choice*. Math. Prog. Comp. 113 (2020), 415–442.

[14] W. Bruns, B. Ichim and C. Söger, *The power of pyramid decomposition in Normaliz*. J. Symb. Comp. 74 (2016), 513–536.

[15] W. Bruns, B. Ichim and C. Söger, *Computations of volumes and Ehrhart series in four candidates elections*. Ann. Oper. Res. 280 (2019), 241–265.

[16] W. Bruns and R. Koch, *Computing the integral closure of an affine semigroup*. Univ. Iagell. Acta Math. 39 (2001), 59–70.

[17] W. Bruns, R. Sieg and C. Söger, *Normaliz 2013–2016*. In G. Böckle, W. Decker and G. Malle, editors, *Algorithmic and Experimental Methods in Algebra, Geometry, and Number Theory*, pages 123–146. Springer, 2018.

[18] W. Bruns and C. Söger, *The computation of weighted Ehrhart series in Normaliz*. J. Symb. Comp. 68 (2015), 75–86.

[19] B. Büeler and A. Enge, *Vinci.* Package available from https://www.math.u-bordeaux.fr/~aenge/

[20] B. Büeler, A. Enge, K. Fukuda, *Exact volume computation for polytopes: a practical study.* In: Polytopes - combinatorics and computation (Oberwolfach, 1997), pp. 131 – 154, DMV Sem. 29,

Birkhäuser, Basel, 2000.

[21] J. A. De Loera, R. Hemmecke and M. Köppe. Algebraic and geometric ideas in the theory of discrete optimization. MOS-SIAM Series on Optimization, 14. Society for Industrial and Applied Mathematics (SIAM), Philadelphia 2013.

[22] V. Delecroix, *embedded algebraic number fields (on top of antic)*, package available at `https://github.com/flatsurf/e-antic`.

[23] P. Filliman, *The volume of duals and sections of polytopes.* Mathematika 37 (1992), 67–80.

[24] S. Gutsche, M. Horn and C. Söger, *NormalizInterface for GAP*. Available at `https://github.com/gap-packages/NormalizInterface`.

[25] S. Gutsche and R. Sieg, *PyNormaliz - an interface to Normaliz from python*. Available at `https://github.com/Normaliz/PyNormaliz`.

[26] W. B. Hart, *Algebraic Number Theory In C*. Package available at `https://github.com/wbhart/antic`.

[27] W. B. Hart, F. Johansson and S. Pancratz, *FLINT: Fast Library for Number Theory*. Available at `https://flintlib.org`.

[28] Hemmecke and P. N. Malkin. Computing generating sets of lattice ideals and Markov bases of lattices. J. Symb. Comp. 44, 1463–1476 (2009).

[29] F. Johansson, *Arb - a C library for arbitrary-precision ball arithmetic*. Available at `https://arblib.org/`.

[30] J. Lawrence, *Polytope volume computation.* Math. Comp. 57 (1991), 259–271.

[31] M. Köppe and S. Verdoolaege, *Computing parametric rational generating functions with a primal Barvinok algorithm*. Electron. J. Comb. 15, No. 1, Research Paper R16, 19 p. (2008).

[32] B. D. McKay and A. Piperno, *Practical graph isomorphism, II*. J. Symbolic Comput. 60 (2014), 94–112.

[33] L. Pottier, *The Euclide algorithm in dimension n*. Research report, ISSAC 96, ACM Press 1996.

[34] A. Schürmann, *Exploiting polyhedral symmetries in social choice*. Social Choice and Welfare 40 (2013), 1097–1110.

[35] B. Sturmfels, *Gröbner baes and convex polytopes*. American Mathematical Society 1996.

Index of keywords