Normaliz 3.10.1

Winfried Bruns Max Horn

Former Normaliz 3 team members: Tim Römer, Richard Sieg,
Christof Söger nd Ulrich von der Ohe

Normaliz 2 team member: Bogdan Ichim

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

mailto:normaliz@uos.de

https://hub.docker.com/r/normaliz/normaliz/

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

Short reference: NmzShortRef.pdf

Contents

1. Introduction

1.1. The objectives of Normaliz

The program Normaliz is a tool for computing the Hilbert bases and enumerative data of rational cones and, more generally, sets of lattice points in rational polyhedra. Moreover, Normaliz computes algebraic polyhedra, i.e., polyhedra defined over algebraic number fields embedded into $\mathbb{R}$.

Since version 3.10.0 Normaliz can also compute data of general affine monoids: their minimal systems of generators, Hilbert series, Markov and Gröbner bases of defining ideals and some local data, such as the singular locus.

The mathematical background and the terminology of this manual are explained in Appendix A. For a thorough treatment of the mathematics involved we refer the reader to [11]. The terminology follows [11]. For algorithms of Normaliz see [8], [9], [12], [13], [14], [15], [17], [18], and [19]. Some new developments are briefly explained in this manual.

Both polyhedra and lattices can be given by

(1) systems of generators and/or
(2) constraints.

Since version 3.1, cones need not be pointed and polyhedra need not have vertices, but are allowed to contain a positive-dimensional affine subspace.

Affine monoids can be defined by generators and by toric ideals, in other words, by binomial equations.

In order to describe a rational polyhedron by *generators*, one specifies a finite set of vertices $x_1,\dots,x_n \in \mathbb{Q}^d$ and a set $y_1,\dots,y_m \in \mathbb{Z}^d$ generating a rational cone C. The polyhedron defined by these generators is

$$P = \operatorname{conv}(x_1,\dots,x_n) + C, \qquad C = \mathbb{R}_+ y_1 + \cdots + \mathbb{R}_+ y_m.$$

An affine lattice defined by generators is a subset of $\mathbb{Z}^d$ given as

$$L = w + L_0, \qquad L_0 = \mathbb{Z} z_1 + \cdots + \mathbb{Z} z_r, \qquad w, z_1, \dots, z_r \in \mathbb{Z}^d.$$

Constraints defining a polyhedron are affine-linear inequalities with integral coefficients, and the constraints for an affine lattice are affine-linear diophantine equations and congruences. The conversion between generators and constraints is an important task of Normaliz.

The first main goal of Normaliz is to compute a system of generators for

$$P \cap L.$$

The minimal system of generators of the monoid $M = C \cap L_0$ is the Hilbert basis $\operatorname{Hilb}(M)$ of M. The homogeneous case, in which $P = C$ and $L = L_0$, is undoubtedly the most important one, and in this case $\operatorname{Hilb}(M)$ is the system of generators to be computed. In the general case

the system of generators consists of $\mathrm{Hilb}(M)$ and finitely many points $u_1, \ldots, u_s \in P \cap L$ such that

$$P \cap L = \bigcup_{j=1}^{s} u_j + M.$$

The second main goal are enumerative data that depend on a grading of the ambient lattice. Normaliz computes the Hilbert series and the Hilbert quasipolynomial of the monoid or set of lattice points in a polyhedron. In combinatorial terminology: Normaliz computes Ehrhart series and quasipolynomials of rational polyhedra. Normaliz also computes weighted Ehrhart series and Lebesgue integrals of polynomials over rational polytopes.

For algebraic polyhedra Normaliz realizes the computation goals above that make sense without the finite generation of the monoid of lattice points in a cone: convex hull and vertex enumeration for all algebraic polyhedra, and, for polytopes, lattice points, volumes and triangulations.

The computation goals of Normaliz can be set by the user. In particular, they can be restricted to subtasks, such as the lattice points in a polytope or the leading coefficient of the Hilbert (quasi)polynomial.

Performance data of Normaliz can be found in [9], [14], [15] and [16].

Acknowledgment. In 2013–2016 the development of Normaliz has been supported by the DFG SPP 1489 "Algorithmische und experimentelle Methoden in Algebra, Geometrie und Zahlentheorie". From November 2020 to October 2021 Normaliz is supported by the DFG project "Normaliz: development and long term sustainability".

1.2. Platforms, implementation and access from other systems

Executables for Normaliz are provided for Mac OS, Linux and MS Windows. If the executables prepared cannot be run on your system, then you can compile Normaliz yourself (see Section 12). The statically linked Linux binaries provided by us can be run in the Linux subsystem of MS Windows 10. A Docker image of Normaliz is available.

Normaliz is written in C++, and should be compilable on every system that has a GCC compatible compiler. It uses the standard package GMP (see Section 12). The parallelization is based on OpenMP. CoCoALib [2] and Flint [28] are optional packages. The computation of algebraic polytopes is based on e-antic [23], antic [27] and arb [30].

Normaliz consists of two parts: the front end "normaliz" for input and output and the C++ library "libnormaliz" that does the computations.

Normaliz can be accessed from the interactive general purpose system PYTHON via the interface PYNORMALIZ written by Sebastian Gutsche with contributions by Justin Shenk and Richard Sieg.

Normaliz can also be accessed from the following systems:

- SINGULAR via the library `normaliz.lib`,
- MACAULAY2 via the package `Normaliz.m2`,

- CoCoA via an external library and libnormaliz,
- GAP via the GAP package NormalizInterface [25] which uses libnormaliz,
- polymake (thanks to the polymake team),
- SageMath via PyNormaliz.

The Singular and Macaulay2 interfaces are contained in the Normaliz distribution. At present, their functionality is limited to Normaliz 2.10. Nevertheless they profit from newer versions.

Furthermore, Normaliz is used by B. Burton's system Regina and in SecDec by S. Borowka et al.

Normaliz does not have its own interactive shell. We recommend the access via PyNormaliz, GAP or SageMath for interactive use. PyNormaliz is documented in Appendix E.

1.3. Major changes relative to version 3.8.0

In 3.8.0:

(1) Computation of automorphism groups.
(2) Computation goal Incidence added.
(3) SCIP removed.

3.8.1 fixes a bug in the face lattice computation.

In 3.8.2:

(1) Replacement of `boost::dynamic_bitset` by own class.
(2) Improvements in the convex hull algorithm.
(3) Further tests.
(4) Improvements of build infrastructure.

3.8.3 is a technical release.

In 3.8.4:

(1) Bug fixes.
(2) Further improvements of the build infrastructure and additional tests.
(3) Extended use of precomputed data.
(4) New package layout for the releases.

In 3.8.5:

(1) Bug fixes.
(2) Improvements in several algorithms.
(3) Substantial improvement in the computation of integer hulls.
(4) Refined triangulations added.

3.8.6 is a technical prerelease.

In 3.8.7:

(1) Addition of computation goals IsEmptySemiopen and CoveringFace
(2) Source file structure changed

(3) Improvement in finding preexisting dependencies

In 3.8.8:

(1) Dual versions of face lattice, f-vector and incidence
(2) Rational lattices

In 3.8.9:

(1) `ExtremeRaysFloat` introduced
(2) `TriangulationGenerators` replace `Generators`
(3) Improved stability for interactive use

In 3.8.10:

(1) Management of triangulations and related decompositions completely revised.
(2) Documentation of PyNormaliz added.
(3) SHA256 hash values for certain data.

In 3.9.0:

(1) Volume and integral computation by signed decomposition.
(2) Variant `tExploitIsosMult` added to volume by descent.
(3) `AmbientAutomorphisms` and `InputAutomorphisms` added.
(4) `PlacingTriangulations` and `PullingTriangulation` added.
(5) e-antic updated to version 1.0.1.

In 3.9.1:

(1) Better handling of distributed computation.
(2) Python 2 no longer supported.

In 3.9.2:

(1) Compilation for MS Windows under MSYS; MPIR no longer forced under Windows.
(2) Bug fixes and improvements.
(3) Extension of sparse vectors to ranges of indices and `unit_matrix` as an input type.
(4) Output of an input fie with precomputed data.
(5) libnormaliz function that constructs a cone from an input file.

In 3.9.3:

(1) Bug fixes.
(2) Compilation for MS Windows under MSYS with all optional packages.
(3) Option NoHilbertBasisOutput added.
(4) Short reference for Normaliz added.
(5) `normaliz.lib` for Singular updated.

n 3.9.4:

(1) Polynomial constraints for lattice points.
(2) Coarse project-and-lift for positive systems.
(3) Patching variant for coarse project-and-lift .
(4) Input directive `convert_equations`.

In 3.10.0:

(1) Improvements in patching algorithm with polynomoial equations. item Input types `monoid`, `lattice_ideal` (changed), `toric_ideal`, `normal_toric_ideal`.
(2) Markov and Gröbnber bases of lattice ideals.
(3) Hilbert series for all positive affine monoids.
(4) Computation of the singular locus.

In 3.10.1:

(1) Weight vector for Gröbner bases of lattice ideals.
(2) Substantial improvements in the patching variant of project-and-lift.
(3) Time bound can be set (so far only in project-and-lift)
(4) Option NoOutputOnInterrupt.

See the file `CHANGELOG` in the Normaliz directory for more information on the history of Normaliz.

1.4. Future extensions

(1) Exploitation of automorphism groups,
(2) addition of linear programming methods,
(3) multigraded Hilbert series,
(4) access from further systems,
(5) heterogeneous parallelization,

1.5. Download and installation

In order to install Normaliz you should have a look at

`https://normaliz.uos.de/download/`.

It guides you to our GitHub repository

`https://github.com/Normaliz/Normaliz/releases`.

There you will also find binary releases for Linux, Mac OS and MS Windows. Unzip the package for your system in a directory of your choice. In it, a directory `normaliz-3.10.1` (called Normaliz directory in the following) is created with several subdirectories.

Another source for the executables of all three systems is the package manager Conda. See

`https://github.com/conda-forge/normaliz-feedstock`

An alternative to the (system dependent) executable is the

Docker image `normaliz/normaliz`

that is automatically downloaded from the Docker repository if you ask for it. (In the Docker container, the Normaliz directory is called `Normaliz`, independently of the version number.)

See Section 11 for more details on the distribution and the Docker image.

A source package is available as well. See Section 12 if you want to compile Normaliz yourself.

1.6. Exploring Normaliz online

You can explore Normaliz online at

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

(may take a while.) The button "New" offers you a terminal. Choose it, and you will be in a Docker container based on the Normaliz Docker image. Your username is `norm`, and Normaliz is contained in the subdirectory `Normaliz` of your home directory. Moreover, it is installed, and can be invoked by the command `normaliz` from anywhere. Just type

```
normaliz -c Normaliz/example/rational
```

to run a small computation. You van also have a Python shell and run PyNormaliz or study the tutorial of PyNormaliz (a Jupyter notebook).

It is possible to upload and download files, but please refrain from using Binder as a platform for heavy computations.

2. Discrete convex geometry by examples

2.1. Terminology

For the precise interpretation of parts of the Normaliz output some terminology is necessary, but this section can be skipped at first reading, and the user can come back to it when it becomes necessary. We will give less formal descriptions along the way. The following applies to rational polyhedra. Algebraic polyhedra are discussed in Section 8.

As pointed out in the introduction, Normaliz "computes" intersections $P \cap L$ where P is a rational polyhedron in $\mathbb{R}^d$ and L is an affine sublattice of $\mathbb{Z}^d$. It proceeds as follows:

(1) If the input is inhomogeneous, then it is homogenized by introducing a homogenizing coordinate: the polyhedron P is replaced by the cone $C(P)$: it is the closure of $\mathbb{R}_+(P \times \{1\})$ in $\mathbb{R}^{d+1}$. Similarly L is replaced by $\widetilde{L} = \mathbb{Z}(L \times \{1\})$. In the homogeneous case in which P is a cone and L is a subgroup of $\mathbb{Z}^d$, we set $C(P) = P$ and $\widetilde{L} = L$.

(2) The computations take place in the *efficient lattice*

$$\mathbb{E} = \widetilde{L} \cap \mathbb{R}C(P).$$

where $\mathbb{R}C(P)$ is the linear subspace generated by $C(P)$. The internal coordinates are chosen with respect to a basis of $\mathbb{E}$. The *efficient cone* is

$$\mathbb{C} = \mathbb{R}_+(C(P) \cap \mathbb{E}).$$

(3) Inhomogeneous computations are truncated using the dehomogenization (defined implicitly or explicitly).

(4) The final step is the conversion to the original coordinates. Note that we must use the coordinates of $\mathbb{R}^{d+1}$ if homogenization has been necessary, simply because some output vectors may be non-integral otherwise.

Normaliz computes inequalities, equations and congruences defining $\mathbb{E}$ and $\mathbb{C}$. The output contains only those constraints that are really needed. They must always be used jointly: the equations and congruences define $\mathbb{E}$, and the equations and inequalities define $\mathbb{C}$. Altogether they define the monoid $M = \mathbb{C} \cap \mathbb{E}$. In the homogeneous case this is the monoid to be computed. In the inhomogeneous case we must intersect M with the dehomogenizing hyperplane to obtain $P \cap L$.

In this section, only pointed cones (and polyhedra with vertices) will be discussed. Nonpointed cones will be addressed in Section 7.13.

2.2. Practical preparations

You may find it comfortable to run Normaliz via the GUI jNormaliz [4]. In the Normaliz directory open jNormaliz by clicking `jNormaliz.jar` in the appropriate way. (We assume that Java is installed on your machine.) In the jNormaliz file dialogue choose one of the input files

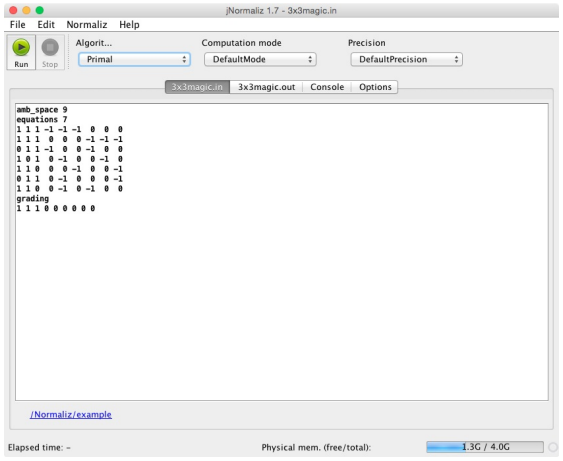


Figure 1: jNormaliz

in the subdirectory `example`, say `small.in`, and press `Run`. In the console window you can watch Normaliz at work. Finally inspect the output window for the results.

The menus and dialogues of jNormaliz are self explanatory, but you can also consult the documentation [4] via the help menu.

Remark The jNormaliz drop down menus do presently not cover all options of Normaliz. But since all computation goals and algorithmic variants can be set in the input file, there is no real restriction in using jNormaliz. The only option not reachable by jNormaliz is the output directory (see Section 6.6).

Moreover, one can, and often will, run Normaliz from the command line. This is fully explained in Section 6. At this point it is enough to call Normaliz by typing

```
normaliz -c <project>
```

where `<project>` denotes for the project to be computed. Normaliz will load the file `<project>.in`. The option `-c` makes Normaliz to write a progress report on the terminal. Normaliz writes its results to `<project>.out`.

Note that you may have to prefix `normaliz` by a path name, and `<project>` must contain a path to the input file if it is not in the current directory. Suppose the Normaliz directory is the current directory and we are using a Linux or Mac system. Then

```
./normaliz -c example/small
```

will run `small.in` from the directory `example`. On Windows we must change this to

```
.\normaliz -c example\small
```

The commands given above will run Normaliz with the at most 8 parallel threads. For the very small examples in this tutorial you may want to add `-x=1` to suppress parallelization. For large examples, you can increase the number of parallel threads by `-x=<N>` where `<N>` is the number

of threads that you want to suggest. See Section 6.3.

As long as you don't specify a computation goal on the command line or in the input file, Normaliz will use the *default computation goals*:

```
HilbertBasis
HilbertSeries
ClassGroup
```

The computation of the Hilbert series requires the explicit or implicit definition of a grading. Normaliz does only complain that a computation goal cannot be reached if the goal has been set explicitly. For example, if you say `HilbertSeries` and there is no grading, an exception will be thrown and Normaliz terminates, but an output file with the already computed data will be written.

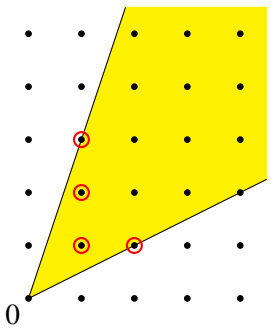
Note that the spacing in the output files may have changed over time and that not all these changes may have made their way into this manual.

Normaliz will always print the results that are obtained on the way to the computation goals and do not require extra effort.

Appendix B helps you to read the console output that you have demanded by the option `-c`.

2.3. A cone in dimension 2

We want to investigate the cone $C = \mathbb{R}_+(2,1) + \mathbb{R}_+(1,3) \subset \mathbb{R}^2$:



This cone is defined in the input file `2cone.in`:

```
amb_space 2
cone 2
1 3
2 1
```

The input tells Normaliz that the ambient space is $\mathbb{R}^2$, and then a cone with 2 generators is defined, namely the cone C from above.

The figure indicates the Hilbert basis, and this is our first computation goal.

If you prefer to consider the columns of a matrix as input vectors (or have a matrix in this format from another system) you can use the input

```
amb_space 2
cone transpose 2
1 2
3 1
```

Note that the number `2` following `transpose` is now the number of *columns*. Later on we will also show the use of formatted matrices.

2.3.1. The Hilbert basis

In order to compute the Hilbert basis, we run Normaliz from jNormaliz or by

```
./normaliz -c example/2cone
```

and inspect the output file:

```
4 Hilbert basis elements
2 extreme rays
2 support hyperplanes
```

Self explanatory so far.

```
embedding dimension = 2
rank = 2 (maximal)
external index = 1
internal index = 5
original monoid is not integrally closed  in chosen lattice
```

The embedding dimension is the dimension of the space in which the computation is done. The rank is the rank of the lattice $\mathbb{E}$ (notation as in Section 2.1). In fact, in our example $\mathbb{E} = \mathbb{Z}^2$, and therefore has rank 2.

For subgroups $G \subset U \subset \mathbb{Z}^d$ we denote the order of the torsion subgroup of U/G by the *index* of G in U. The *external index* is the index of the lattice $\mathbb{E}$ in $\mathbb{Z}^d$. In our case $\mathbb{E} = \mathbb{Z}^d$, and therefore the external index is 1. Note: the external index is 1 exactly when $\mathbb{E}$ is a direct summand of $\mathbb{Z}^d$.

For this example and many others the *original monoid* is well defined: the generators of the cone used as input are contained in $\mathbb{E}$. (This need not be the case if $\mathbb{E}$ is a proper sublattice of $\mathbb{Z}^d$, and we let the original monoid be undefined in inhomogeneous computations.) Let G be the subgroup generated by the original monoid. The *internal index* is the index of G in $\mathbb{E}$.

The original monoid is *integrally closed* if and only if the it contains the Hilbert basis, and this is evidently false for our example. We go on.

```
size of triangulation   = 1
resulting sum of |det|s = 5
```

The primal algorithm of Normaliz relies on a (partial) triangulation. In our case the triangulation consists of a single simplicial cone, and (the absolute value of) its determinant is 5.

```
No implicit grading found
```

If you do not define a grading explicitly, Normaliz tries to find one itself: the grading is defined if and only if there is a linear form γ on $\mathbb{E}$ under which all extreme rays of the efficient cone $\mathbb{C}$ have value 1, and if so, γ is the implicit grading. Such does not exist in our case.

The last information before we come to the vector lists:

```
rank of class group = 0
finite cyclic summands:
5: 1
```

The class group of the monoid M has rank 0, in other words, it is finite. It has one finite cyclic summand of order 5.

This is the first instance of a multiset of integers displayed as a sequence of pairs

```
<n>: <m>
```

Such an entry says: the multiset contains the number `<n>` with multiplicity `<m>`.

Now we look at the vector lists (typeset in two columns to save space):

```
4 Hilbert basis elements:       2 extreme rays:
1 1                             1 3
1 2                             2 1
1 3
2 1                             2 support hyperplanes:
                                -1  2
                                 3 -1
```

The support hyperplanes are given by the linear forms (or inner normal vectors):

$$\begin{aligned} -x_1 + 2x_2 &\geq 0, \\ 3x_1 - x_2 &\geq 0. \end{aligned}$$

If the order is not fixed for some reason, Normaliz sorts vector lists as follows: (1) by degree if a grading exists and the application makes sense, (2) lexicographically.

2.3.2. The cone by inequalities

Instead by generators, we can define the cone by the inequalities just computed (`2cone_ineq.in`). We use this example to show the input of a formatted matrix:

```
amb_space auto
inequalities
[[-1  2] [3 -1]]
```

A matrix of input type `inequalities` contains *homogeneous* inequalities. Normaliz can determine the dimension of the ambient space from the formatted matrix. Therefore we can declare the ambient space as being "auto determined" (but `amb_space 2` is not forbidden).

We get the same result as with `2cone.in` except that the data depending on the original monoid cannot be computed: the internal index and the information on the original monoid are missing since there is no original monoid.

2.3.3. The interior

Now we want to compute the lattice points in the interior of our cone. If the cone C is given by the inequalities $\lambda_i(x) \geq 0$ (within $\mathrm{aff}(C)$), then the interior is given by the inequalities $\lambda_i(x) > 0$. Since we are interested in lattice points, we work with the inequalities $\lambda_i(x) \geq 1$.

The input file `2cone_int.in` says

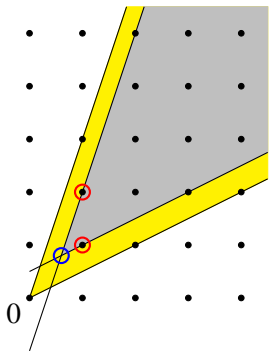
```
amb_space 2
strict_inequalities 2
-1  2
3 -1
```

The strict inequalities encode the conditions

$$-x_1 + 2x_2 \geq 1,$$
$$3x_1 - x_2 \geq 1.$$

This is our first example of inhomogeneous input.

Note that the strict inequalities do not define the interior of the cone as a point set. They define a (closed) polyhedron with the same lattice points as the interior.



Alternatively we could use the following two equivalent input files, in a more intuitive notation:

```
amb_space 2
constraints 2
-1  2 > 0
3 -1 > 0
```

```
amb_space 2
constraints 2
-1  2 >= 1
3 -1 >= 1
```

There is an even more intuitive way to type the input file using symbolic constraints that we will introduce in Section 2.6.2.

Normaliz homogenizes inhomogeneous computations by introducing an auxiliary homogenizing coordinate x_{d+1}. The polyhedron is obtained by intersecting the homogenized cone with the hyperplane $x_{d+1} = 1$. The recession cone is the intersection with the hyperplane $x_{d+1} = 0$. The recession monoid is the monoid of lattice points in the recession cone, and the set of lattice points in the polyhedron is represented by its system of module generators over the recession monoid.

Note that the homogenizing coordinate serves as the denominator for rational vectors. In our example the recession cone is our old friend that we have already computed, and therefore we need not comment on it.

```
2 module generators
4 Hilbert basis elements of recession monoid
1 vertices of polyhedron
2 extreme rays of recession cone
3 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 2
```

The only surprise may be the embedding dimension: Normaliz always takes the dimension of the space in which the computation is done. It is the number of components of the output vectors. Because of the homogenization it has increased by 1.

```
size of triangulation   = 1
resulting sum of |det|s = 25
```

In this case the homogenized cone has stayed simplicial, but the determinant has changed.

```
dehomogenization:
0 0 1
```

The dehomogenization is the linear form δ on the homogenized space that defines the hyperplanes from which we get the polyhedron and the recession cone by the equations $\delta(x) = 1$ and $\delta(x) = 0$, respectively. It is listed since one can also work with a user defined dehomogenization.

```
module rank = 1
```

This is the rank of the module of lattice points in the polyhedron over the recession monoid.

In our case the module is an ideal, and so the rank is 1.

The output of inhomogeneous computations is always given in homogenized form. The last coordinate is the value of the dehomogenization on the listed vectors, 1 on the module generators, 0 on the vectors in the recession monoid:

```
2 module generators:          4 Hilbert basis elements of recession monoid:
1 1 1                         1 1 0
1 2 1                         1 2 0
                              1 3 0
                              2 1 0
```

The module generators are $(1,1)$ and $(1,2)$.

```
1 vertices of polyhedron:
3 4 5
```

Indeed, the polyhedron has a single vertex, namely $(3/5,4/5)$.

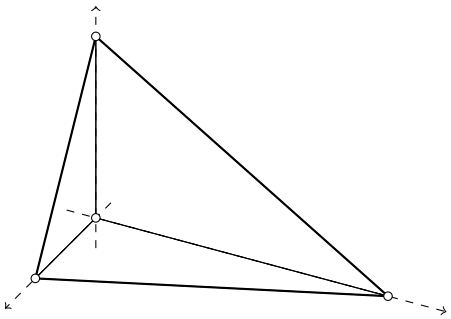
```
2 extreme rays of recession cone:     3 support hyperplanes of polyhedron (homogenized):
1 3 0                                 -1  2 -1
2 1 0                                  0  0  1
                                       3 -1 -1
```

Two support hyperplanes are exactly those that we have used to define the polyhedron – and it has only 2. But Normaliz always outputs the support hyperplanes that are needed for the cone that one obtains by homogenizing the polyhedron, as indicated by "homogenized". The homogenizing variable is always ≥ 0. In this case the support hyperplane $(0,0,1)$ is essential for the description of the cone. Note that it need not always appear.

2.4. A lattice polytope

The file `polytope.in` contains

```
amb_space 4
polytope 4
0 0 0
2 0 0
0 3 0
0 0 5
```



This is a good place to mention that Normaliz also accepts matrices (and vectors) in sparse format:

```
amb_space 4
polytope 4 sparse
;
1:2;
```

```
2:3;
3:5;
```

Each input row, concluded by `;`, lists the indices and the corresponding nonzero values in that row of the matrix.

The Ehrhart monoid of the integral polytope with the 4 vertices

$$(0,0,0)\,,\quad (2,0,0)\,,\quad (0,3,0)\quad \text{and}\quad (0,0,5)$$

in $\mathbb{R}^3$ is to be computed. The generators of the Ehrhart monoid are obtained by attaching a further coordinate 1 to the vertices, and this explains `amb_space 4`. In fact, the input type `polytope` is not only a convenient version of

```
cone 4
0 0 0 1
2 0 0 1
0 3 0 1
0 0 5 1
```

It also sets the he grading to be the last coordinate. See 4.12 below for general information on gradings.

Running `normaliz` produces the file `polytope.out`:

```
19 Hilbert basis elements
18 lattice points in polytope (Hilbert basis elements of degree 1)
4 extreme rays
4 support hyperplanes

embedding dimension = 4
rank = 4 (maximal)
external index = 1
internal index = 30
original monoid is not integrally closed in chosen lattice
```

Perhaps a surprise: the lattice points of the polytope do not yield all Hilbert basis elements.

```
size of triangulation   = 1
resulting sum of |det|s = 30
```

Nothing really new so far. The grading appears in the output file:

```
grading:
0 0 0 1

degrees of extreme rays:
1: 4
```

Again we encounter the notation `<n>: <m>`: we have 4 extreme rays, all of degree 1.

```
Hilbert basis elements are not of degree 1
```

We knew this already: the polytope is not integrally closed as defined in [11]. Now we see the enumerative data defined by the grading:

```
multiplicity = 30

Hilbert series:
1 14 15
denominator with 4 factors:
1: 4

degree of Hilbert Series as rational function = -2

Hilbert polynomial:
1 4 8 5
with common denominator = 1
```

The polytope has $\mathbb{Z}^3$-normalized volume 30 as indicated by the multiplicity (see Section 7.1.1 for a discussion of volumes and multiplicities). The Hilbert (or Ehrhart) function counts the lattice points in kP, $k \in \mathbb{Z}_+$. The corresponding generating function is a rational function $H(t)$. For our polytope it is

$$\frac{1+14t+15t^2}{(1-t)^4}.$$

The denominator is given in multiset notation: `1: 4` say that the factor $(1-t^1)$ occurs with multiplicity 4.

The Ehrhart polynomial (again we use a more general term in the output file) of the polytope is

$$p(k) = 1+4k+8k^2+5k^3\,.$$

In our case it has integral coefficients, a rare exception. Therefore one usually needs a denominator.

Everything that follows has already been explained.

```
rank of class group = 0
finite cyclic summands:
30: 1

**********************************************************************

18 lattice points in polytope (Hilbert basis elements of degree 1):
0 0 0 1
...
2 0 0 1
```

```
1 further Hilbert basis elements of higher degree:
1 2 4 2

4 extreme rays:                 4 support hyperplanes:
0 0 0 1                         -15 -10 -6 30
0 0 5 1                         0   0  1  0
0 3 0 1                         0   1  0  0
2 0 0 1                         1   0  0  0
```

The support hyperplanes give us a description of the polytope by inequalities: it is the solution of the system of the 4 inequalities

$$x_3 \geq 0\,, \quad x_2 \geq 0\,, \quad x_1 \geq 0 \quad \text{and} \quad 15x_1 + 10x_2 + 6x_3 \leq 30\,.$$

2.4.1. Only the lattice points

Suppose we want to compute only the lattice points in our polytope. In the language of graded monoids these are the degree 1 elements, and so we add `Deg1Elements` to our input file (`polytope_deg1.in`):

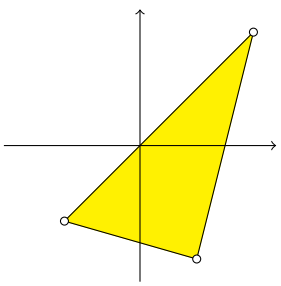
```
amb_space 4
polytope 4
0 0 0
2 0 0
0 3 0
0 0 5
Deg1Elements
/* This is our first explicit computation goal*/
```

We have used this opportunity to include a comment in the input file. The computation of lattice points in a polytope will be taken up again in Sections 2.13 and 7.2.

We lose all information on the Hilbert series, and from the Hilbert basis we only retain the degree 1 elements.

2.5. A rational polytope

The type `polytope` can (now) be used for rational polytopes as well.



We want to investigate the Ehrhart series of the triangle P with vertices

$$(1/2,1/2),\ (-1/3,-1/3),\ (1/4,-1/2).$$

For this example the procedure above yields the input file `rational.in`:

```
amb_space 3
polytope 3
1/2 1/2
-1/3 -1/3
1/4 -1/2
HilbertSeries
```

From the output file we only list the data of the Ehrhart series.

```
multiplicity = 5/8
multiplicity (float) = 0.625

Hilbert series:
1 0 0 3 2 -1 2 2 1 1 1 1 2
denominator with 3 factors:
1: 1  2: 1  12: 1

degree of Hilbert Series as rational function = -3

Hilbert series with cyclotomic denominator:
-1 -1 -1 -3 -4 -3 -2
cyclotomic denominator:
1: 3  2: 2  3: 1  4: 1

Hilbert quasi-polynomial of period 12:
0:   48 28 15                        7:   23 22 15
1:   11 22 15                        8:   16 28 15
2:  -20 28 15                        9:   27 22 15
3:   39 22 15                        10:   -4 28 15
4:   32 28 15                        11:    7 22 15
5:   -5 22 15                        with common denominator = 48
6:   12 28 15
```

The multiplicity is a rational number. Since in dimension 2 the normalized area (of full-dimensional polytopes) is twice the Euclidean area, we see that P has Euclidean area $5/16$. If the multiplicity is not integral, we also print it in floating point format, This is certainly superfluous for a fraction like $5/8$, but very handy if the numerator and the denominator have many digits.

Unlike in the case of a lattice polytope, there is no canonical choice of the denominator of the Ehrhart series. Normaliz gives it in 2 forms. In the first form the numerator polynomial is

$$1+3t^3+2t^4-t^5+2t^6+2t^7+t^8+t^9+t^{10}+t^{11}+2t^{12}$$

and the denominator is

$$(1-t)(1-t^2)(1-t^{12}).$$

As a rational function, $H(t)$ has degree -3. This implies that $3P$ is the smallest integral multiple of P that contains a lattice point in its interior.

Normaliz gives also a representation as a quotient of coprime polynomials with the denominator factored into cyclotomic polynomials. In this case we have

$$H(t) = -\frac{1+t+t^2+t^3+4t^4+3t^5+2t^6}{\zeta_1^3\zeta_2^2\zeta_3\zeta_4}$$

where ζ_i is the i-th cyclotomic polynomial ($\zeta_1 = t-1$, $\zeta_2 = t+1$, $\zeta_3 = t^2+t+1$, $\zeta_4 = t^2+1$).

Normaliz transforms the representation with cyclotomic denominator into one with denominator of type $(1-t^{e_1})\cdots(1-t^{e_r})$, $r = \text{rank}$, by choosing e_r as the least common multiple of all the orders of the cyclotomic polynomials appearing, e_{r-1} as the lcm of those orders that have multiplicity ≥ 2 etc.

There are other ways to form a suitable denominator with 3 factors $1-t^e$, for example $g(t) = (1-t^2)(1-t^3)(1-t^4) = -\zeta_1^3\zeta_2^2\zeta_3\zeta_4$. Of course, $g(t)$ is the optimal choice in this case. However, P is a simplex, and in general such optimal choice may not exist. We will explain the reason for our standardization below.

Let $p(k)$ be the number of lattice points in kP. Then $p(k)$ is a quasipolynomial:

$$p(k) = p_0(k) + p_1(k)k + \cdots + p_{r-1}(k)k^{r-1},$$

where the coefficients depend on k, but only to the extent that they are periodic of a certain period $\pi \in \mathbb{N}$. In our case $\pi = 12$ (the lcm of the orders of the cyclotomic polynomials).

The table giving the quasipolynomial is to be read as follows: The first column denotes the residue class j modulo the period and the corresponding line lists the coefficients $p_i(j)$ in ascending order of i, multiplied by the common denominator. So

$$p(k) = 1 + \frac{7}{12}k + \frac{5}{16}k^2, \qquad k \equiv 0 \pmod{12},$$

etc. The leading coefficient is the same for all residue classes and equals the Euclidean volume (in this case).

Our choice of denominator for the Hilbert series is motivated by the following fact: e_i is the common period of the coefficients $p_{r-i}, \ldots, p_{r-1}$. The user should prove this fact or at least verify it by several examples.

Especially in the case of a simplex the representation of the Hilbert series shown so far may not be the expected one. In fact, there is a representation in which the exponents of t in the denominator are the degrees of the integral extreme generators. So one would expect the denominator to be $(1-t^2)(1-t^3)(1-t^4)$ in our case. The generalization to the nonsimplicial case uses the degrees of a homogeneous system of parameters (see [11, p. 200]). Normaliz can compute such a denominator if the computation goal `HSOP` is set (`rationalHSOP.in`):

```
Hilbert series (HSOP):
1 1 1 3 4 3 2
denominator with 3 factors:
2: 1  3: 1  4: 1
```

Note that the degrees of the elements in a homogeneous system of parameters are by no means unique and that there is no optimal choice in general. To find a suitable sequence of degrees Normaliz must compute the face lattice of the cone to some extent. Therefore be careful not to ask for `HSOP` if the cone has many support hyperplanes.

2.5.1. The series with `vertices`?

It is tempting to define the polytope by the input type `vertices`. This choice makes the computation inhomogeneous, a mode that is mainly meant for (potentially) unbounded polyhedra. But it can be used for polytopes as well, and with this input type you can compute all of the data that we have seen above. You must ask for the `EhrhartSeries` instead of the `HilbertSeries`. The file `rational_inhom.in` is

```
amb_space 2
vertices 3
1/2 1/2 1
-1/3 -1/3 1
1/4 -1/2 1
EhrhartSeries
```

Nevertheless, there is also use for `HilbertSeries` in the inhomogeneous case. But then the grading must be defined on the affine space of the polytope (and not on the cone over the polytope). See Sections 7.1 and 7.10.2.

2.5.2. The rational polytope by inequalities

We extract the support hyperplanes of our polytope from the output file and use them as input (`poly_ineq.in`):

```
amb_space 3
inequalities 3
-8  2 3
1 -1 0
2  7 3
grading
unit_vector 3
HilbertSeries
```

At this point we have to help Normaliz because it has no way to guess that we want to investigate the polytope defined by the inequalities and the choice $x_3 = 1$. This is achieved by the

specification of the grading that maps every vector to its third coordinate.

This is the first time that we used the shortcut `unit_vector <n>` which represents the n-th unit vector $e_n \in \mathbb{R}^d$ and is only allowed for input types which require a single vector.

These data tell us that the polytope, as a subset of $\mathbb{R}^2$, is defined by the inequalities

$$\begin{aligned} -8x_1 + 2x_2 + 3 &\geq 0, \\ x_1 - x_2 + 0 &\geq 0, \\ 2x_1 + 7x_2 + 3 &\geq 0. \end{aligned}$$

These inequalities are inhomogeneous, but we are using the homogeneous input type `inequalities` which amounts to introducing the grading variable x_3 as explained above.

The inequalities as written above look somewhat artificial. It is certainly more natural to write them in the form

$$\begin{aligned} 8x_1 - 2x_2 &\leq 3, \\ x_1 - x_2 &\geq 0, \\ 2x_1 + 7x_2 &\geq -3. \end{aligned}$$

and for the direct transformation into Normaliz input we have introduced the type `hom_constraints`. The prefix `hom` indicates that we want homogeneous inequalities whereas plain `constraints` that we have already seen in Section 2.3.3 gives inhomogeneous inequalities. The file `poly_hom_const.in` contains

```
amb_space 3
hom_constraints 3
8 -2 <= 3
1 -1 >= 0
2 7  >= -3
grading
unit_vector 3
HilbertSeries
```

You can of course also switch to inhomogeneous input using `inhom_inequalities` or `constraints` in the same way as `polytope` can be replaced by `vertices`.

2.6. Magic squares

Suppose that you are interested in the following type of "square"

x_1	x_2	x_3
x_4	x_5	x_6
x_7	x_8	x_9

and the problem is to find nonnegative values for $x_1, \ldots, x_9$ such that the 3 numbers in all rows, all columns, and both diagonals sum to the same constant $\mathscr{M}$. Sometimes such squares are

called *magic* and $\mathscr{M}$ is the *magic constant*. This leads to a linear system of equations

$$\begin{aligned}
x_1+x_2+x_3 &= x_4+x_5+x_6;\\
x_1+x_2+x_3 &= x_7+x_8+x_9;\\
x_1+x_2+x_3 &= x_1+x_4+x_7;\\
x_1+x_2+x_3 &= x_2+x_5+x_8;\\
x_1+x_2+x_3 &= x_3+x_6+x_9;\\
x_1+x_2+x_3 &= x_1+x_5+x_9;\\
x_1+x_2+x_3 &= x_3+x_5+x_7.
\end{aligned}$$

This system is encoded in the file `3x3magic.in`:

```
amb_space 9
equations 7
1 1 1 -1 -1 -1  0  0  0
1 1 1  0  0  0 -1 -1 -1
0 1 1 -1  0  0 -1  0  0
1 0 1  0 -1  0  0 -1  0
1 1 0  0  0 -1  0  0 -1
0 1 1  0 -1  0  0  0 -1
1 1 0  0 -1  0 -1  0  0
grading
sparse 1:1 2:1 3:1;
```

The input type `equations` represents *homogeneous* equations. The first equation reads

$$x_1+x_2+x_3-x_4-x_5-x_6=0,$$

and the other equations are to be interpreted analogously. The magic constant is a natural choice for the grading. It is given in sparse form, equivalent to the dense form

```
grading
1 1 1 0 0 0 0 0 0
```

It seems that we have forgotten to define the cone. This may indeed be the case, but doesn't matter: if there is no input type that defines a cone, Normaliz chooses the positive orthant, and this is exactly what we want in this case.

The output file contains the following:

```
5 Hilbert basis elements
5 lattice points in polytope (Hilbert basis elements of degree 1)
4 extreme rays
4 support hyperplanes

embedding dimension = 9
```

```
rank = 3
external index = 1

size of triangulation   = 2
resulting sum of |det|s = 4

grading:
1 1 1 0 0 0 0 0 0
with denominator = 3
```

The input degree is the magic constant. However, as the denominator 3 shows, the magic constant is always divisible by 3, and therefore the effective degree is $\mathscr{M}/3$. This degree is used for the multiplicity, the Hilbert series, and the Hilbert basis elements of degree 1, and other date depending on the degree.

By introducing the grading denominator, Normaliz has changed the grading defined by you, and you may not like this. There is a way out: add the option `NoGradingDenom`. We will discuss the consequences below.

```
degrees of extreme rays:
1: 4

Hilbert basis elements are of degree 1
```

This was not to be expected (and is no longer true for 4×4 squares).

```
multiplicity = 4

Hilbert series:
1 2 1
denominator with 3 factors:
1: 3

degree of Hilbert Series as rational function = -1

Hilbert polynomial:
1 2 2
with common denominator = 1
```

The Hilbert series is

$$\frac{1+2t+t^2}{(1-t)^3}.$$

The Hilbert polynomial is

$$P(k) = 1+2k+2k^2,$$

and after substituting $\mathscr{M}/3$ for k we obtain the number of magic squares of magic constant $\mathscr{M}$, provided 3 divides $\mathscr{M}$. (If $3 \nmid \mathscr{M}$, there is no magic square of magic constant $\mathscr{M}$.)

```
rank of class group = 1
finite cyclic summands:
2: 2
```

So the class group is $\mathbb{Z} \oplus (\mathbb{Z}/2\mathbb{Z})^2$.

```
5 lattice points in polytope (Hilbert basis elements of degree 1):
0 2 1 2 1 0 1 0 2
1 0 2 2 1 0 0 2 1
1 1 1 1 1 1 1 1 1
1 2 0 0 1 2 2 0 1
2 0 1 0 1 2 1 2 0

0 further Hilbert basis elements of higher degree:
```

The 5 elements of the Hilbert basis represent the magic squares

2	0	1
0	1	2
1	2	0

1	0	2
2	1	0
0	2	1

1	1	1
1	1	1
1	1	1

1	2	0
0	1	2
2	0	1

0	2	1
2	1	0
1	0	2

All other solutions are linear combinations of these squares with nonnegative integer coefficients. One of these 5 squares is clearly in the interior:

```
4 extreme rays:                 4 support hyperplanes:
0 2 1 2 1 0 1 0 2               -2 -1 0 0  4 0 0 0 0
1 0 2 2 1 0 0 2 1               0 -1 0 0  2 0 0 0 0
1 2 0 0 1 2 2 0 1               0  1 0 0  0 0 0 0 0
2 0 1 0 1 2 1 2 0               2  1 0 0 -2 0 0 0 0
```

These 4 support hyperplanes cut out the cone generated by the magic squares from the linear subspace they generate. Only one is reproduced as a sign inequality. This is due to the fact that the linear subspace has submaximal dimension and there is no unique lifting of linear forms to the full space.

```
6 equations:                3 basis elements of generated lattice:
1 0 0 0 0  1 -2 -1  1       1 0 -1 -2 0  2  1  0 -1
0 1 0 0 0  1 -2  0  0       0 1 -1 -1 0  1  1 -1  0
0 0 1 0 0  1 -1 -1  0       0 0  3  4 1 -2 -1  2  2
0 0 0 1 0 -1  2  0 -2
0 0 0 0 1 -1  1  0 -1
0 0 0 0 0  3 -4 -1  2
```

So one of our equations has turned out to be superfluous (why?). Note that also the equations are not reproduced exactly. Finally, Normaliz lists a basis of the efficient lattice $\mathbb{E}$ generated by the magic squares.

Note that the equations and the lattice basis are not uniquely determined. We transform their

matrices into reduced row echelon form to force unique output files.

2.6.1. Blocking the grading denominator

As mentioned above, one can block the grading denominator and force Normaliz to use the input grading. For the magic squares we augment the input file as follows (`3x3magicNGD.in`):

```
amb_space 9
equations 7
1 1 1 -1 -1 -1  0  0  0
...
1 1 0  0 -1  0 -1  0  0
grading
sparse 1:1 2:1 3:1;
NoGradingDenom
```

The consequences:

```
grading:
1 1 1 0 0 0 0 0 0

degrees of extreme rays:
3: 4

multiplicity = 4/9
multiplicity (float) = 0.444444444444

Hilbert series:
1 0 0 2 0 0 1
denominator with 3 factors:
3: 3

degree of Hilbert Series as rational function = -3

The numerator of the Hilbert series is symmetric.

Hilbert series with cyclotomic denominator:
-1 0 0 -2 0 0 -1
cyclotomic denominator:
1: 3  3: 3

Hilbert quasi-polynomial of period 3:
0:  9 6 2
1:  0 0 0
2:  0 0 0
with common denominator = 9
```

```
rank of class group = 1
finite cyclic summands:
2: 2

**************************************************************************

0 lattice points in polytope (Hilbert basis elements of degree 1):
```

It is easy to relate the data with the grading denominator to those without. You must decide yourself what you prefer. One aspect is whether one prefers intrinsic data (with grading denominator) to extrinsic ones that depend on the embedding (without the grading denominator). We will discuss the topic again in Section 7.1.

2.6.2. With even corners

We change our definition of magic square by requiring that the entries in the 4 corners are all even. Then we have to augment the input file by the following (`3x3magiceven.in`):

```
congruences 4 sparse
1:1 10:2;
3:1 10:2;
7:1 10:2;
9:1 10:2;
```

This sparse form is equivalent to the dense form

```
congruences 4
1 0 0 0 0 0 0 0 0 2
0 0 1 0 0 0 0 0 0 2
0 0 0 0 0 0 1 0 0 2
0 0 0 0 0 0 0 0 1 2
```

The first 9 entries in each row represent the coefficients of the coordinates in the homogeneous congruences, and the last is the modulus:

$$x_1 \equiv 0 \mod 2$$

is the first congruence etc.

We could also define these congruences as symbolic constraints:

```
constraints 4 symbolic
x[1] ~ 0(2);
x[3] ~ 0(2);
x[7] ~ 0(2);
x[9] ~ 0(2);
```

The output changes accordingly:

```
9 Hilbert basis elements
0 lattice points in polytope (Hilbert basis elements of degree 1)
4 extreme rays
4 support hyperplanes

embedding dimension = 9
rank = 3
external index = 4

size of triangulation   = 2
resulting sum of |det|s = 8

grading:
1 1 1 0 0 0 0 0 0
with denominator = 3

degrees of extreme rays:
2: 4

multiplicity = 1

Hilbert series:
1 -1 3 1
denominator with 3 factors:
1: 1  2: 2

degree of Hilbert Series as rational function = -2

Hilbert series with cyclotomic denominator:
-1 1 -3 -1
cyclotomic denominator:
1: 3  2: 2

Hilbert quasi-polynomial of period 2:
0:   2 2 1
1:  -1 0 1
with common denominator = 2
```

After the extensive discussion in Section 2.5 it should be easy for you to write down the Hilbert series and the Hilbert quasipolynomial. (But keep in mind that the grading has a denominator.)

```
rank of class group = 1
finite cyclic summands:
4: 2

**************************************************************************
```

```
0 lattice points in polytope (Hilbert basis elements of degree 1):

9 further Hilbert basis elements of higher degree:
...

4 extreme rays:
0 4 2 4 2 0 2 0 4
2 0 4 4 2 0 0 4 2
2 4 0 0 2 4 4 0 2
4 0 2 0 2 4 2 4 0
```

We have listed the extreme rays since they have changed after the introduction of the congruences, although the cone has not changed. The reason is that Normaliz always chooses the extreme rays from the efficient lattice $\mathbb{E}$.

```
4 support hyperplanes:
...

6 equations:
...                              3 basis elements of generated lattice:
                                   1 0 -1 -2 0  2  1  0 -1
2 congruences:                     0 1 -1 -1 0  1  1 -1  0
1 0 0 0 0 0 0 0 0 2                0 0  3  4 1 -2 -1  2  2
0 1 0 0 1 0 0 0 0 2
```

The rank of the lattice has of course not changed, but after the introduction of the congruences the basis has changed.

2.6.3. The lattice as input

It is possible to define the lattice by generators. We demonstrate this for the magic squares with even corners. The lattice has just been computed (`3x3magiceven_lat.in`):

```
amb_space 9
lattice 3
0  1 2 3 1 -1 0  1 2
2 -1 2 1 1  1 0  3 0
0  3 0 1 1  1 2 -1 2
grading
1 1 1 0 0 0 0 0 0
```

It produces the same output as the version starting from equations and congruences.

`lattice` has a variant that takes the saturation of the sublattice generated by the input vectors (`3x3magic_sat.in`):

```
amb_space 9
```

```
saturation 3
0  1 2 3 1 -1 0  1 2
2 -1 2 1 1  1 0  3 0
0  3 0 1 1  1 2 -1 2
grading
1 1 1 0 0 0 0 0 0
```

Clearly, we remove the congruences by this choice and arrive at the output of `3x3magic.in`.

2.7. Decomposition in a numerical semigroup

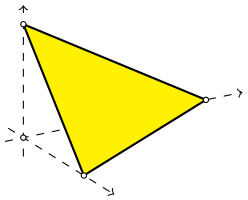
Let $S = \langle 6, 10, 15 \rangle$, the numerical semigroup generated by $6, 10, 15$. How can 97 be written as a sum in the generators?

In other words: we want to find all nonnegative integral solutions to the equation

$$6x_1 + 10x_2 + 15x_3 = 97.$$

Input (`NumSemi.in`):

```
amb_space 3
constraints 1 symbolic
6x[1] + 10x[2] + 15x[3] = 97;
```



The equation cuts out a triangle from the positive orthant.

The set of solutions is a module over the monoid M of solutions of the homogeneous equation $6x_1 + 10x_2 + 15x_3 = 0$. So $M = 0$ in this case.

```
6 lattice points in polytope (module generators):
2 1 5 1
2 4 3 1
2 7 1 1
7 1 3 1
7 4 1 1
12 1 1 1

0 Hilbert basis elements of recession monoid:
```

The last line is as expected, and the 6 lattice points (or module generators) are the goal of the computation.

Normaliz is smart enough to recognize that it must compute the lattice points in a polygon, and does exactly this. You can recognize it in the console output: Normaliz 3.10.1 has used the project-and-lift algorithm. We will discuss it further in Section 2.13 and Section 7.2.1.

For those who like to play: add the option `--NoProjection` to the command line. Then the terminal output will change; Normaliz computes the lattice points as a truncated Hilbert basis via a triangulation (only one simplicial cone in this case).

2.8. A job for the dual algorithm

We increase the size of the magic squares to 5×5. Normaliz can do the same computation as for 3×3 squares, but this will take some minutes. Suppose we are only interested in the Hilbert basis, we should use the dual algorithm for this example. (The dual algorithm goes back to Pottier [34].) The input file is `5x5dual.in`:

```
amb_space 25
equations 11
1 1 1 1 1 -1 -1 -1 -1 -1  0  0  0  0  0  0  0  0  0  0  0  0  0
...
1 1 1 1 0  0  0  0 -1  0  0  0 -1  0  0  0 -1  0  0  0 -1  0  0  0  0
grading
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
HilbertBasis
```

The input file does not say anything about the dual algorithm mentioned in the section title. With this input it is chosen automatically. See Section 7.5 for a discussion of when this happens. But you can insist on the dual algorithm by adding `DualMode` to the input (or `-d` to the command line). Or, if you want to compare it to the primal algorithm add `PrimalMode` (or `-P` to the command line).

The Hilbert basis contains 4828 elements, too many to be listed here.

With the file `5x5.in` you can compute the Hilbert basis and the Hilbert series, and the latter with HSOP:

```
Hilbert series (HSOP):
1 15 356 4692 36324 198467 ... 198467 36324 4692 356 15 1
denominator with 15 factors:
1: 5  2: 3  6: 2  12: 1  60: 2  420: 1  1260: 1

degree of Hilbert Series as rational function = -5

The numerator of the Hilbert Series is symmetric.
```

In view of the length of the numerator of the Hilbert series it may be difficult to observe the symmetry. So Normaliz does it for you. The symmetry shows that the monoid is Gorenstein, but if you are only interested in the Gorenstein property, there is a much faster way to check it (see Section 7.7).

The size 6×6 is out of reach for the Hilbert series, but the Hilbert basis can be computed (in the automatically chosen dual mode). It takes some hours.

2.9. A dull polyhedron

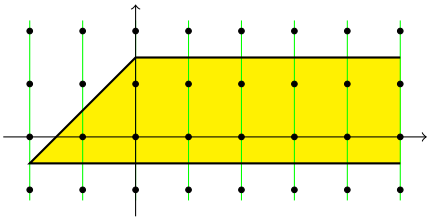
We want to compute the polyhedron defined by the inequalities

$$\begin{aligned}\xi_2 &\geq -1/2,\\ \xi_2 &\leq 3/2,\\ \xi_2 &\leq \xi_1 + 3/2.\end{aligned}$$

They are contained in the input file `InhomIneq.in`:

```
amb_space 2
constraints 3
0 1 >= -1/2
0 1 <=  3/2
-1 1 <=  3/2
grading
unit_vector 1
FVector
```

The grading says that we want to count points by the first coordinate, namely along the green lines:



It yields the output

```
2 module generators
1 Hilbert basis elements of recession monoid
2 vertices of polyhedron
1 extreme rays of recession cone
3 support hyperplanes of polyhedron (homogenized)

f-vector:
1 2 3 1
```

The empty set is the intersection of all facets, and this gives the first entry 1. Then we have 2 vertices, 3 edges, and finally the full polyhedron.

The out put continues:

```
embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 1
```

```
size of triangulation   = 1
resulting sum of |det|s = 8

dehomogenization:
0 0 1

grading:
1 0 0
```

The interpretation of the grading requires some care in the inhomogeneous case. We have extended the input grading vector by an entry 0 to match the embedding dimension. For the computation of the degrees of *lattice points* in the ambient space you can either use only the first 2 coordinates or take the full scalar product of the point in homogenized coordinates and the extended grading vector.

```
module rank = 2
multiplicity = 2
```

The module rank is 2 in this case since we have two "layers" in the solution module that are parallel to the recession monoid. This is of course also reflected in the Hilbert series.

```
Hilbert series:
1 1
denominator with 1 factors:
1: 1

shift = -1
```

We haven't seen a shift yet. It is always printed (necessarily) if the Hilbert series does not start in degree 0. In our case it starts in degree -1 as indicated by the shift -1. We thus get the Hilbert series

$$t^{-1}\,\frac{t+t}{1-t} = \frac{t^{-1}+1}{1-t}.$$

Note: We used the opposite convention for the shift in Normaliz 2.

Note that the Hilbert (quasi)polynomial is always computed for the unshifted monoid defined by the input data. (This was different in previous versions of Normaliz.)

```
degree of Hilbert Series as rational function = -1

Hilbert polynomial:
2
with common denominator = 1

******************************************************************************

2 module generators:
-1 0 1
```

```
0 1 1

1 Hilbert basis elements of recession monoid:
1 0 0

2 vertices of polyhedron:
-4 -1 2
0  3 2

1 extreme rays of recession cone:
1 0 0

3 support hyperplanes of polyhedron (homogenized):
0 -2 3
0  2 1
2 -2 3
```

The dual algorithm that was used in Section 2.8 can also be applied to inhomogeneous computations. We would of course loose the Hilbert series. In certain cases it may be preferable to suppress the computation of the vertices of the polyhedron if you are only interested in the integer points; see Section 5.6.

2.9.1. Defining it by generators

If the polyhedron is given by its vertices and the recession cone, we can define it by these data (`InhomIneq_gen.in`):

```
amb_space 2
vertices 2
-4 -1 2
0  3 2
cone 1
1 0
grading
unit_vector 1
```

The output is identical to the version starting from the inequalities.

2.10. The Condorcet paradox

In social choice elections each of the k voters picks a linear preference order of the n candidates. There are $n!$ such orders. The election result is the vector $(x_1,\dots,x_N)$, $N=n!$, in which x_i is the number of voters that have chosen the i-th preference order in, say, lexicographic enumeration of these orders. (Thus $x_1+\cdots+x_N=k$.) In the following we assume the *impartial anonymous culture* according to which every election result has the same probability if the

number of voters is fixed.

We say that candidate A *beats* candidate B if the majority of the voters prefers A to B. As the Marquis de *Condorcet* (and others) observed, "beats" is not transitive, and an election may exhibit the *Condorcet paradox*: there is no Condorcet winner. (See [19] and the references given there for more information.)

We want to find the probability for $k \to \infty$ that there is a Condorcet winner for $n = 4$ candidates. The event that A is the Condorcet winner can be expressed by linear inequalities on the election outcome (a point in 24-space). The wanted probability is the lattice normalized volume of the polytope cut out by the inequalities at $k = 1$. The file `Condorcet.in`:

```
amb_space 24
inequalities 3
1 1 1 1 1 1 -1 -1 -1 -1 -1 -1   1  1 -1 -1  1 -1   1  1 -1 -1  1 -1
1 1 1 1 1 1  1  1 -1 -1  1 -1  -1 -1 -1 -1 -1 -1   1  1  1 -1 -1 -1
1 1 1 1 1 1  1  1  1 -1 -1 -1   1  1  1 -1 -1 -1  -1 -1 -1 -1 -1 -1
nonnegative
total_degree
Multiplicity
```

The first inequality expresses that A beats B, the second and the third say that A beats C and D. (So far we do not exclude ties, and they need not be excluded for probabilities as $k \to \infty$.)

In addition to these inequalities we must restrict all variables to nonnegative values, and this is achieved by adding the attribute `nonnegative`. The grading is set by `total_degree`. It replaces the grading vector with 24 entries 1. Finally `Multiplicity` sets the computation goal.

From the output file we only mention the quantity we are out for:

```
multiplicity = 1717/8192
multiplicity (float) = 0.209594726562
```

Since there are 4 candidates, the probability for the existence of a Condorcet winner is $1717/2048 = 0.209595$.

We can refine the information on the Condorcet paradox by computing the Hilbert series. Either we delete `Multiplicity` from the input file or, better, we add `--HilbertSeries` (or simply `-q`) on the command line. The result:

```
Hilbert series:
1 5 133 363 4581 8655 69821 100915 ... 12346 890 481 15 6
denominator with 24 factors:
1: 1  2: 14  4: 9

degree of Hilbert Series as rational function = -25
```

If your executable of Normaliz was built with CoCoALib (see Section 12), for example the executables for Linux or Mac OS from our distribution or in the Docker image, it uses sym-

metrization for the computation of the Hilbert series. If not, then simply disregard any remark on symmetrization. Everything runs very quickly also without it.

If symmetrization has been used, you will also find a file `Condorcet.symm.out` in your directory. It contains the data computed for the symmetrization. You need not care at this point. We take continue the discussion of symmetrization in Section 7.8.

2.10.1. Excluding ties

Now we are more ambitious and want to compute the Hilbert series for the Condorcet paradox, or more precisely, the number of election outcomes having A as the Condorcet winner depending on the number k of voters. Moreover, as it is customary in social choice theory, we want to exclude ties. The input file changes to `CondorcetSemi.in`:

```
amb_space 24
excluded_faces 3
1 1 1 1 1 1 -1 -1 -1 -1 -1 -1   1  1 -1 -1  1 -1   1  1 -1 -1  1 -1
1 1 1 1 1 1  1  1 -1 -1  1 -1  -1 -1 -1 -1 -1 -1   1  1  1 -1 -1 -1
1 1 1 1 1 1  1  1  1 -1 -1 -1   1  1  1 -1 -1 -1  -1 -1 -1 -1 -1 -1
nonnegative
total_degree
HilbertSeries
```

We could omit `HilbertSeries`, and the computation would include the Hilbert basis. The type `excluded_faces` only affects the Hilbert series. In every other respect it is equivalent to `inequalities`.

From the file `CondorcetSemi.out` we only display the Hilbert series:

```
Hilbert series:
6 15 481 890 12346 ... 100915 69821 8655 4581 363 133 5 1
denominator with 24 factors:
1: 1  2: 14  4: 9

shift = 1

degree of Hilbert Series as rational function = -24
```

Surprisingly, this looks like the Hilbert series in the previous section read backwards, roughly speaking. This is true, and one can explain it as we will see below.

It is justified to ask why we don't use `strict_inequalities` instead of `excluded_faces`. It does of course give the same Hilbert series. However, Normaliz cannot (yet) apply symmetrization in inhomogeneous computations. Moreover, the algorithmic approach is different, and according to our experience `excluded_faces` is more efficient, independently of symmetrization.

See Section 7.20 for more information on `excluded_faces`.

2.10.2. At least one vote for every preference order

Suppose we are only interested in elections in which every preference order is chosen by at least one voter. This can be modeled as follows (`Condorcet_one.in`):

```
amb_space 24
inequalities 3
1 1 1 1 1 1 -1 -1 -1 -1 -1 -1   1  1 -1 -1  1 -1   1  1 -1 -1  1 -1
1 1 1 1 1 1  1  1 -1 -1  1 -1  -1 -1 -1 -1 -1 -1   1  1  1 -1 -1 -1
1 1 1 1 1 1  1  1  1 -1 -1 -1   1  1  1 -1 -1 -1  -1 -1 -1 -1 -1 -1
strict_signs
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
total_degree
HilbertSeries
```

The entry 1 at position i of the vector `strict_signs` imposes the inequality $x_i \geq 1$. A -1 would impose the inequality $x_i \leq -1$, and the entry 0 imposes no condition on the i-th coordinate.

```
Hilbert series:
1 5 133 363 4581 8655 69821 100915 ... 12346 890 481 15 6
denominator with 24 factors:
1: 1  2: 14  4: 9

shift = 24

degree of Hilbert Series as rational function = -1
```

Again we encounter (almost) the Hilbert series of the Condorcet paradox (without side conditions). It is time to explain this coincidence. Let C be the Condorcet cone defined by the nonstrict inequalities, M the monoid of lattice points in it, $I_1 \subset M$ the ideal of lattice points avoiding the 3 facets defined by ties, I_2 the ideal of lattice points with strictly positive coordinates, and finally I_3 the ideal of lattice points in the interior of C. Moreover, let $\mathbb{1} \in \mathbb{Z}^{24}$ be the vector with all entries 1.

Since $\mathbb{1}$ lies in the three facets defining the ties, it follows that $I_2 = M + \mathbb{1}$. This explains why we obtain the Hilbert series of I_2 by multiplying the Hilbert series of M by t^{24}, as just observed. Generalized Ehrhart reciprocity (see [11, Theorem 6.70]) then explains the Hilbert series of I_1 that we observed in the previous section. Finally, the Hilbert series of I_3 that we don't have displayed is obtained from that of M by "ordinary" Ehrhart reciprocity. But we can also obtain I_1 from I_3: $I_1 = I_3 - \mathbb{1}$, and generalized reciprocity follows from ordinary reciprocity in this very special case. (Also see [16].)

The essential point in these arguments (apart from reciprocity) is that $\mathbb{1}$ lies in all support hyperplanes of C except the coordinate hyperplanes.

You can easily compute the Hilbert series of I_3 by making all inequalities strict.

As the terminal output shows, symmetrization has not been applied for the reason mentioned above: `strict_signs` is an inhomogeneous input type. It would of course be possible to

encode the strict signs as `excluded_faces`. Then the sparse format of matrices is very handy:

```
excluded_faces 24
1:1;
1:2;
...
1:24;
```

This is a shorthand for the unit matrix.

2.10.3. The f-vector with codimension bound

Suppose we are interested in the f-vector of the cone defined by `Condorcet.in`. In view of the rather high dimension the face lattice must be expected to be extremely large, but computing the f-vector to codimension 4 should be no problem. (See [10] for the Normaliz face lattice algorithm.) Indeed it is not. We use `CondorcetFV.in`:

```
...
FVector
face_codim_bound 4
```

Then we find in the output file:

```
f-vector (possibly truncated):
17550 2925 351 27 1
```

Note that the face numbers are listed by descending codimension or, equivalently, by increasing dimension. The leftmost number is the number of faces in the highest codimension that has been computed. So we have 17550 codimension 4 faces.

2.11. Testing normality

We want to test the monoid $A_{4\times4\times3}$ defined by $4\times4\times3$ contingency tables for normality (see [12] for the background). The input file is `A443.in`:

```
amb_space 40
cone_and_lattice 48
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
HilbertBasis
```

Why `cone_and_lattice`? Well, we want to find out whether the monoid is normal, i.e., whether $M = C(M) \cap \mathrm{gp}(M)$. If M is even integrally closed in $\mathbb{Z}^{24}$, then it is certainly integrally closed in the evidently smaller lattice $\mathrm{gp}(M)$, but the converse does not hold in general, and therefore we work with the lattice generated by the monoid generators.

It turns out that the monoid is indeed normal:

```
original monoid is integrally closed in chosen lattice
```

Actually the output file reveals that M is even integrally closed in $\mathbb{Z}^{24}$: the external index is 1, and therefore $\mathrm{gp}(M)$ is integrally closed in $\mathbb{Z}^{24}$.

The output file also shows that there is a grading on $\mathbb{Z}^{24}$ under which all our generators have degree 1. We could have seen this ourselves: Every generator has exactly one entry 1 in the first 16 coordinates. (This is clear from the construction of M.)

A noteworthy detail from the output file:

```
size of partial triangulation   = 48
```

It shows that Normaliz uses only a partial triangulation in Hilbert basis computations; see [12].

It is no problem to compute the Hilbert series as well if you are interested in it. Simply add `-q` to the command line or remove `HilbertBasis` from the input file. Then a full triangulation is needed (size $2,654,272$).

Similar examples are `A543`, `A553` and `A643`. The latter is not normal, as we will see below. Even on a standard PC or laptop, the Hilbert basis computation does not take very long because Normaliz uses only a partial triangulation. The Hilbert series can still be determined, but the computation time will grow considerably since the it requires a full triangulation. See [15] for timings.

2.11.1. Computing just a witness

If the Hilbert basis is large and there are many support hyperplanes, memory can become an issue for Normaliz, as well as computation time. Often one is only interested in deciding whether the given monoid is integrally closed (or normal). In the negative case it is enough to find a single element that is not in the original monoid – a witness disproving integral closedness. As soon as such a witness is found, Normaliz stops the Hilbert basis computation (but will continue to compute other data if they are asked for). We look at the example `A643.in` (for which the full Hilbert basis is not really a problem):

```
amb_space 54
cone_and_lattice 72
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 ...
...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 ...
WitnessNotIntegrallyClosed
```

Don't add `HilbertBasis` because it will overrule `IsIntegrallyClosed`!

The output:

```
72 extreme rays
153858 support hyperplanes
```

```
embedding dimension = 54
rank = 42
external index = 1
internal index = 1
original monoid is not integrally closed in chosen lattice
witness for not being integrally closed:
0 0 1 0 1 1 1 1 0 0 1 0 0 1 0 1 0 1 1 0 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1 ...

grading:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 ...

degrees of extreme rays:
1: 72

**********************************************************************

72 extreme rays:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 ...
...
```

If you repeat such a computation, you may very well get a different witness if several parallel threads find witnesses. Only one of them is delivered.

If you just want to check integral closedness as quickly as possible, replace `WitnessNotIntegrallyClosed` by `IsIntegrallyClosed`. Normaliz first checks some necessary conditions. If they are satisfied, the calculation of the Hilbert basis is started. If it finds a witness for not being integrally closed, the witness is displayed in the output.

2.12. Convex hull computation/vertex enumeration

Normaliz computes convex hulls as should be very clear by now, and the only purpose of this section is to emphasize that Normaliz can be restricted to this task by setting an explicit computation goal. By convex hull computation we mean the determination of the support hyperplanes of a polyhedron is given by generators (or vertices). The converse operation is vertex enumeration. Both amount to the dualization of a cone, and can therefore be done by the same algorithm.

As an example we take the input file `cyclicpolytope30-15.in`, the cyclic polytope of dimension 15 with 30 vertices (suggested by D. Avis and Ch. Jordan):

```
/* cyclic polytope of dimension 15 with 30 vertices */
amb_space 16
polytope 30
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768
...
30 900 27000 810000 ... 478296900000000000000 1434890700000000000000
```

```
SupportHyperplanes
```

Already the entries of the vertices show that the computation cannot be done in 64 bit arithmetic. But you need not be worried. Just start Normaliz as usual. It will simply switch to infinite precision by itself, as shown by the terminal output (use the option `-c` or `--Verbose`).

```
\.....|
Normaliz 3.2.0                  \....|
\...|
(C) The Normaliz Team, University of Osnabrueck   \..|
January  2017                       \.|
\|
************************************************************
Compute: SupportHyperplanes
Could not convert 15181127029874798299.
Arithmetic Overflow detected, try a bigger integer type!
Restarting with a bigger type.
************************************************************
starting primal algorithm (only support hyperplanes) ...
Generators sorted lexicographically
Start simplex 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
gen=17, 72 hyp
gen=18, 240 hyp
gen=19, 660 hyp
gen=20, 1584 hyp
gen=21, 3432 hyp
gen=22, 6864 hyp
gen=23, 12870 hyp
gen=24, 22880 hyp
gen=25, 38896 hyp
gen=26, 63648 hyp
gen=27, 100776 hyp
gen=28, 155040 hyp
gen=29, 232560 hyp
gen=30, 341088 hyp
Pointed since graded
Select extreme rays via comparison ... done.
------------------------------------------------------------
transforming data... done.
```

Have a look at the output file if you are not afraid of 341088 linear forms.

If you have looked closely at the terminal output above, you should have stumbled on the lines

```
Could not convert 15181127029874798299.
Arithmetic Overflow detected, try a bigger integer type!
```

They show that Normaliz has tried the computation in 64 bit integers, but encountered a num-

ber that is too large for this precision. It has automatically switched to infinite precision. (See Section 5.3 for more information on integer types.)

2.13. Lattice points in a polytope and its Euclidean volume

The computation of lattice points in a polytope can be viewed as a truncated Hilbert basis computation, and we have seen in preceding examples. But Normaliz can be restricted to their computation, with homogeneous as well as with inhomogeneous input. Let us look at `ChF_8_1024.in`:

```
amb_space 8
constraints 16
0.10976576 0.2153132834 ... 0.04282847494 >= -1/2
...
0.10976576 -0.2153132834 ... -0.04282847494 >= -1/2
0.10976576 0.2153132834 ... 0.04282847494 <= 1/2
0.10976576 -0.2153132834 ...-0.04282847494 <= 1/2
LatticePoints
ProjectionFloat
```

This example comes from numerical analysis; see Ch. Kacwin, J. Oettershagen and T. Ullrich, On the orthogonality of the Chebyshev-Frolov lattice and applications, Monatsh. Math. 184 (2017), 425–441). Its origin explains the decimal fractions in the input. Normaliz converts them immediately into ordinary fractions of type numerator/denominator, and then makes the input integral as usual.

In the output file you can see to what integer vectors Normaliz has converted the inequalities of the input file:

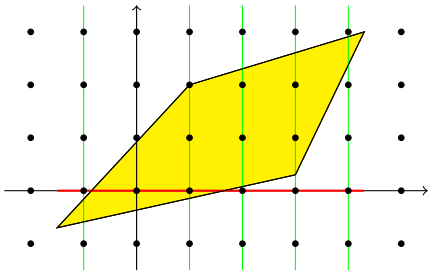
```
16 support hyperplanes of polyhedron (homogenized):
5488288000  10765664170  ...  2141423747 25000000000
...
-5488288000  10765664170 ...   2141423747 25000000000
```

The option `ProjectionFloat` indicates that we want to compute the lattice points in the polytope defined by the inequalities and that we want to use the floating point variant of the project-and-lift algorithm; `Projection` would make Normaliz use its ordinary arithmetic in this algorithm. For our example the difference in time is not really significant, but when you try `VdM_16_1048576.in`, it becomes very noticeable. Let us have a look at the relevant part of then terminal output:

```
Polyhedron is parallelotope
Computing lattice points by project-and-lift
LLL based on support hyperplanes
Projection
embdim 9 inequalities 16
...
```

```
embdim 6 inequalities 140
...
embdim 2 inequalities 2
embdim 1 inequalities 0
Lifting
Lifting to dimension 2
Lifting to dimension 3
...
Lifting to dimension 8
Lifting to dimension 9
embdim 2 LatticePoints 5
embdim 3 LatticePoints 21
...
embdim 8 LatticePoints 907
embdim 9 LatticePoints 1067
Project-and-lift complete
```

We start with embedding dimension 9 since we need a homogenizing coordinate in inhomogeneous computations. Then the polytope is successively projected onto a coordinate hyperplane until we reach a line segment given by 2 inequalities. In the second part Normaliz lifts the lattice points back through all projections. The following figure illustrates the procedure for a polygon that is projected to a line segment.



The green lines show the fibers over the lattice points in the (red) line segment. Note that not every lattice point in the projection must be liftable to a lattice point in the next higher dimension.

In `ChF_8_1024.out` we see

```
1067 lattice points in polytope (module generators):
-4  0  0  0  0  0  0  0 1
-3  0  0  0 -1  0  0  0 1
-3  0  0  0  0  0  0  0 1
...
3  0  0  0  0  0  0  0 1
3  0  0  0  1  0  0  0 1
4  0  0  0  0  0  0  0 1
```

Normaliz finds out that our polytope is in fact a parallelotope. This allows Normaliz to suppress the computation of its vertices. We are not interested in them, and they look frightening

when written as ordinary fractions (computed with the additional option `SupportHyperplanes`). This is only the first vertex, the denominator is the number in the last row:

```
256 vertices of polyhedron:
-7831972155307708173239167258085974255845869779051329651906336771582421875
-2560494334732147696394408175864650673712115229853232268085759500000000000
241193292411744825003604124168323708374286000514244717129567484500000000
-2170682283899852950367663781367299946065844697990214478942400250000000000
184601354007762175056223233356965155155965920765943807476092280050000000
-1450403531662801634587765586956338287943865886737024582718631750000000000
999055328718773316303519268629091038893656784654239444024061220000000000
-5093139905224682158163668274274288315089017971888102494350624500000000000
229248633580316965731682361560246162542228357108960340867209201212984250 6
...
```

Not all polytopes are parallelotopes, and in most cases Normaliz must compute the vertices or extreme rays as an auxiliary step, even if we are not interested in them. You can always add the option

`NoExtRaysOutput`

if you want to suppress their output. (The numerical information on the number of extreme rays etc. will however be included in the output file if it is available.) Similarly one can suppress the output of support hyperplanes by

`NoSuppHypsOutput`

On the other hand, the information provided by the vertices or support hyperplanes may be important. Instead of the unreadable integer output shown above, you can ask for

`VerticesFloat`

Then the vertices of polyhedra are printed in floating point format:

```
256 vertices of polyhedron:
-3.41637   -1.11691     1.0521  ... 0.435796  -0.222167          1
-3.41637  -0.946868   0.435796  ...  -1.0521   0.632677          1
...
```

Note that they can only be printed if a polyhedron is defined. This is always the case in inhomogeneous computations, but in the homogeneous case a grading is necessary. There is also a variant `ExtremeRaysFloat`.

Similarly we can get the support hyperplanes in floating point format (they are only defined up to a positive scalar multiple) by

`SuppHypsFloat`

resulting in

```
16 support hyperplanes of polyhedron (homogenized):
-0.219532  -0.430627  -0.405641  ...  -0.168022 -0.0856569          1
-0.219532  -0.365068  -0.168022  ...   0.405641    0.24393          1
```

```
...
```

By its construction, our polytope should have Euclidean volume 1024. We can confirm this number by computing the volume, using the option

`Volume, -V`

We get

```
volume (normalized) = 205078125000...00/49670537275735342575...58763
volume (normalized, float) =41287680.0308
volume (Euclidean) = 1024.00000076
```

The result makes us happy, despite of the small inaccuracy of the floating point computation on which the Euclidean volume is based. See Section 7.1.1 for a discussion of volumes and multiplicities.

2.14. The integer hull

The integer hull of a polyhedron P is the convex hull of the set of lattice points in P (despite of its name, it usually does not contain P). Normaliz computes by first finding the lattice points and then computing the convex hull. The computation of the integer hull is requested by the computation goal `IntegerHull`.

The computation is somewhat special since it creates a second cone (and lattice) C_{int}. In homogeneous computations the degree 1 vectors generate C_{int} by an input matrix of type `cone_and_lattice`. In inhomogeneous computations the module generators and the Hilbert basis of the recession cone are combined and generate C_{int}. Therefore the recession cone is reproduced, even if the polyhedron should not contain a lattice point.

The integer hull computation itself is always inhomogeneous. The output file for C_{int} is `<project>.IntHull.out`.

As a very simple example we take `rationalIH.in` (`rational.in` augmented by `IntegerHull`):

```
amb_space 3
cone 3
1 1 2
-1 -1 3
1 -2 4
grading
unit_vector 3
HilbertSeries
IntegerHull
```

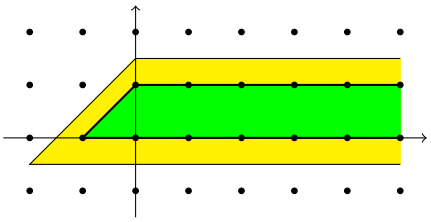
It is our rational polytope from Section 2.5. We know already that the origin is the only lattice point it contains. Nevertheless let us have a look at `rationalIH.IntHull.out`:

```
1 vertices of polyhedron
0 extreme rays of recession cone
```

```
1 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 0
rank of recession monoid = 0  (polyhedron is polytope)
internal index = 1


********************************************************************

1 vertices of polyhedron:
0 0 1

0 extreme rays of recession cone:

1 support hyperplanes of polyhedron (homogenized):
0 0 1

2 equations:
1 0 0
0 1 0

1 basis elements of generated lattice:
0 0 1
```

Since the lattice points in P are already known, the goal was to compute the constraints defining the integer hull. Note that all the constraints defining the integer hull can be different from those defining P. In this case the integer hull is cit out by the 2 equations.

As a second example we take the polyhedron of Section 2.9. The integer hull is the "green" polyhedron:



The input is `InhomIneqIH.in` (`InhomIneq.in` augmented by `IntegerHull`). The data of the integer hull are found in `InhomIneqIH.IntHull.out`:

```
...
2 vertices of polyhedron:
-1 0 1
0 1 1
```

```
1 extreme rays of recession cone:
1 0 0

3 support hyperplanes of polyhedron (homogenized):
0 -1 1
0  1 0
1 -1 1
```

2.15. Inhomogeneous congruences

We want to compute the nonnegative solutions of the simultaneous inhomogeneous congruences

$$x_1 + 2x_2 \equiv 3 \quad (7),$$
$$2x_1 + 2x_2 \equiv 4 \quad (13)$$

in two variables. The input file `InhomCong.in` is

```
amb_space 2
constraints 2 symbolic
x[1] + 2x[2] ~ 3 (7);
2x[1] +  2x[2] ~ 4 (13);
```

This is an example of input of symbolic constraints. We use ~ as the best ASCII character for representing the congruence sign $\equiv$.

Alternatively one can use a matrix in the input As for which we must move the right hand side over to the left.

```
amb_space 2
inhom_congruences 2
1 2 -3 7
2 2 -4 13
```

It is certainly harder to read.

The first vector list in the output:

```
3 module generators:
0 54 1
1  1 1
80  0 1
```

Easy to check: if $(1,1)$ is a solution, then it must generate the module of solutions together with the generators of the intersections with the coordinate axes. Perhaps more difficult to find:

```
6 Hilbert basis elements of recession monoid:
0 91 0
```

```
1 38 0
3 23 0                                          1 vertices of polyhedron:
5  8 0                                          0 0 91
12  1 0
91  0 0
```

Strange, why is $(0,0,1)$, representing the origin in $\mathbb{R}^2$, not listed as a vertex as well? Well the vertex shown represents an extreme ray in the lattice $\mathbb{E}$, and $(0,0,1)$ does not belong to $\mathbb{E}$.

```
2 extreme rays of recession cone:
0 91 0
91  0 0

3 support hyperplanes of polyhedron (homogenized)
0 0 1
0 1 0
1 0 0

1 congruences:
58 32 1 91
```

Normaliz has simplified the system of congruences to a single one.

```
3 basis elements of generated lattice:
1 0  33
0 1 -32
0 0  91
```

Again, don't forget that Normaliz prints a basis of the efficient lattice $\mathbb{E}$.

2.15.1. Lattice and offset

The set of solutions to the inhomogeneous system is an affine lattice in $\mathbb{R}^2$. The lattice basis of $\mathbb{E}$ above does not immediately let us write down the set of solutions in the form $w+L_0$ with a subgroup L_0, but we can easily transform the basis of $\mathbb{E}$: $(1,1,1)$ is in $\mathbb{E}$ and we use it to reduce the third column of the other two basis elements to 0. Try the file `InhomCongLat.in`:

```
amb_space 2
offset
1 1
lattice 2
5 8
-12 -1
```

2.15.2. Variation of the signs

Suppose we want to solve the system of congruences under the condition that both variables are negative (`InhomCongSigns.in`):

```
amb_space 2
inhom_congruences 2
1 2 -3 7
2 2 -4 13
signs
-1 -1
```

The two entries of the sign vector impose the sign conditions $x_1 \leq 0$ and $x_2 \leq 0$.

From the output we see that the module generators are more complicated now:

```
4 module generators:
-11   0 1
-4  -7 1
-2 -22 1
0 -37 1
```

The Hilbert basis of the recession monoid is simply that of the nonnegative case multiplied by -1.

2.16. Integral closure and Rees algebra of a monomial ideal

Next, let us discuss the example `MonIdeal.in` (typeset in two columns):

```
amb_space 5
rees_algebra 9
1 2 1 2                 1 0 3 4
3 1 1 3                 5 1 0 1
2 5 1 0                 2 4 1 5
0 2 4 3                 2 2 2 4
0 2 3 4
```

The input vectors are the exponent vectors of a monomial ideal I in the ring $K[X_1,X_2,X_3,X_4]$. We want to compute the normalization of the Rees algebra of the ideal. In particular we can extract from it the integral closure of the ideal. Since we must introduce an extra variable T, we have `amb_space 5`.

In the Hilbert basis we see the exponent vectors of the X_i, namely the unit vectors with last component 0. The vectors with last component 1 represent the integral closure $\bar{I}$ of the ideal. There is a vector with last component 2, showing that the integral closure of I^2 is larger than $\bar{I}^2$.

```
16 Hilbert basis elements:
0 0 0 1 0
```

```
...
5 1 0 1 1
6 5 2 2 2

11 generators of integral closure of the ideal:
0 2 3 4
...
5 1 0 1
```

The output of the generators of $\bar{I}$ is the only place where we suppress the homogenizing variable for "historic" reasons. If we extract the vectors with last component 1 from the extreme rays, then we obtain the smallest monomial ideal that has the same integral closure as I.

```
10 extreme rays:
0 0 0 1 0
...
5 1 0 1 1
```

The support hyperplanes which are not just sign conditions describe primary decompositions of all the ideals $\overline{I^k}$ by valuation ideals. It is not hard to see that none of them can be omitted for large k (for example, see: W. Bruns and G. Restuccia, Canonical modules of Rees algebras. J. Pure Appl. Algebra 201, 189–203 (2005)).

```
23 support hyperplanes:
0 0 0  0   1
0 ...
6 0 1  3 -13
```

2.16.1. Only the integral closure of the ideal

If only the integral closure of the ideal is to be computed, one can choose the input as follows (`IntClMonId.in`):

```
amb_space 4
vertices 9
1 2 1 2 1
...
2 2 2 4 1
cone 4
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

The generators of the integral closure appear as module generators in the output and the generators of the smallest monomial ideal with this integral closure are the vertices of the polyhedron.

3. Affine monid algebras and binomial ideals by examples

The role of binomials in the computation of affine monoids and their algebras is briefly explained in Sections A.8 and A.9. We assume that the user is familiar with them.

3.1. Computations for affine monids

3.1.1. Input and default computation goals

Affine monoids are given to Normaliz by the input type

monoid

as in monoid.in:

```
amb_space 3
monoid 6
1 0 0
2 3 5
0 0 1
1 1 2
0 1 3
3 1 0
/* grading 1 -2 1*/
/*HilbertSeries*/
/*GroebnerBasis*/
/*Lex*/
/*MarkovBasis*/
/*gb_degree_bound 11*/
/*gb_min_degree 9*/
/*Multiplicity*/
/*SingularLocus*/
/*CodimSingularLocus*/
/*IsSerreR1*/
```

Positivity of the monoid does *not* mean that all components of the input vectors are nonnegative. It only means that $x = 0$ if both x and $-x$ belong to it.

Let us translate this example into multiplicative notation. We have binomials in $K[X_1, X_2, X_3]$, namely

$$M_1 = X_1, \qquad M_2 = X_1^2 X_2^3 X_3^5, \ldots, \qquad M_6 = X_1^3 X_2.$$

In the output file we see

```
...
original monoid is not integrally closed in chosen lattice
...
```

```
6 Hilbert basis elements:
0 0 1
1 0 0
0 1 3
1 1 2
3 1 0
2 3 5

5 support hyperplanes:
0   0 1
0   1 0
1   0 0
2  -3 1
5 -15 7
```

The support hyperplanes re those of the cone generated by the monoid. They are used in auxiliary computations, for example in finding the Hilbert basis, i.e., the unique minimal system of generators of our monoid. In this case the input vectors are all in the Hilbert basis, but thins need not be the case. The Hilbert basis is ordered by degree and lexicographically within each degree. In fact, we have a grading

```
grading:
1 1 1
```

For the default choice of the grading we start from the standard grading on the ambient lattice. Then the grading, whether the default choice or an explicit grading in the input, is divided by the greatest common divisor of he degrees of the generators. In the context of monomial algebras it is the most natural choice. The division by the gcd can be suppressed by `NoGradingDenom`. In our example the gcd is 1.

Our monoid actually has another grading, in which all generators have degree 1: `grading 1 -2 1` in the input file. Activate it and study the changes.

Note: The input type `monoid` is close to `cone_and_lattice` if the monoid is normal.But there are two differences in the default choices: (1) The default computation goals and (2) the default grading. In fact, for `monoid` is is derived from the standard grading on the ambient lattice, whereas or `cone_and_lattice` it gives degree 1 to the extreme integral generators, provided this is possible.

The input type `monoid` allows fewer computationn goals nd options than `cone_and_lattice`. These are

`HilbertBasis`	`HilbertSeries`	`IsIntegrallyClosed`	`Multiplicity`
`Grading`	`IsSerreR1`	`HilbertQuasiPolynomial`	`Automorphisms`
`MarkovBasis`	`Representations`	`AmbientAutomorphisms`	`SingularLocus`
`GroebnerBasis`	`CodimSingularLocus`	`InputAutomorphisms`	`Revlex`
`Lex`	`DegLex`		

Automorphism groups of monoids are discussed in Secion 7.22.7.

3.1.2. Markov and Gröbner bases, Representations

The purpose of the computations in this section is to understand the defining ideal of the subalgebra A of $K[X_1,X_2,X_3]$ generated by our binomials $M_1,\dots,M_6$ introduced above. To this end we activate

`MarkovBasis`

in `monoid.in`, the Markov basis is computed and returned in the file with suffix

`mrk` file containing the Markov basis

In our case `monoid.mrk`:

```
7
6
1 0 -1 -1 1 0
-2 0 2 -1 0 1
-1 0 1 -2 1 1
2 1 0 -1 -1 -1
0 0 0 -3 2 1
1 1 1 -3 0 0
0 1 2 -2 -1 0
```

Each column corresponds to an input vector, and the rows aee indeed relations: the scalar product of a row listed in the Markov basis and a column of the matrix `monoid` is 0. The binomials in $P = K[Y_1,\dots,Y_6]$ corresponding to the rows in the Markov basis form a system of generators of the binomial ideal defining our monoid algebra as a residue class ring of P. The binomials are

$$b_1 = Y_1Y_5 - Y_3Y_4, \qquad b_2 = Y_3^2Y_6 - Y_1^2Y_3, \dots, \qquad b_7 = Y_2Y_3^2 - Y_4^2Y_5.$$

and indeed the binomials vanish if we substitute M_i for Y_i, $i = 1,\dots,7$. That they generate the defining ideal is claimed by Normaliz.

For easier reference the input matrix is mirrored in the file with suffix

`ogn` file with the original generators

in our case `monoid.ogn`:

```
6
3
1 0 0
2 3 5
0 0 1
1 1 2
0 1 3
3 1 0
```

In order to compute a Gröbner basis of our binomial ideal, we activate

`GroebnerBasis`

and get the output file with suffix

grb containing the Gröbner basis.

For the Gröbner basis one has to choose a monomial order. The default choice is degree reverse lexicographic" In our case it yields

```
8
6
-1 0 1 1 -1 0
0 0 0 3 -2 -1
1 0 -1 2 -1 -1
2 0 -2 1 0 -1
0 1 2 -2 -1 0
1 1 1 0 -2 -1
2 1 0 -1 -1 -1
3 0 -3 0 1 -1
```

More precisely: the indeterminates in the polynomial ring housing the binomials are ordered $Y_1 > \ldots Y_6$ and we take the degree reverse lexicographic extension, where' degree" means the total standard degree on the polynomial ring P. The file with suffix `ogn` is also created for the Gröbner basis. There is no output of the (minimal) Markov basis, unless you ask for it explicitly.

Despite of being the default choice, the degree reverse lexicographic order is in the list of pertaining computation goals:

RevLex degree reverse lexicographic order (with respect to the standard grading on the polynomial ring)

Lex lexicographic order

DegLex gegree lexicographic order (with the same degree as RevLex)

Activate also `Lex` in our example and see what changes. `DegLex` is taken with respect to the total standard degree as well, and makes no difference in our case, since the generating binomials are homogeneous in this grading. It is also possoible to apply a weight vector specified by

gb_weight

Its length is the number of variables of the binomial ideal. For the input type `monoid` it is the number of input generators of the monoid. (For `lattice_ideal` and `toric_ideal` it is `amb_space; see below.`) Monomials of egual weight must be distinguished by a monomial order that serves as a tie bgreaker. The choices are `RevLex` (default) or `Lex`. (`DegLex` has a fixed weight namely the standard grading.) Note that the entries of the weightb vector musat be positive if `RevLex` is the tie breaker, and nonnnegative for `Lex`. An example (`monoid_weight.in`):

```
amb_space 3
monoid 6
1 0 0
...
3 1 0
```

```
GroebnerBasis
gb_weight
1 2 3 4 5 6
```

and the Gröbner basis is

```
9
6
-1 0 1 1 -1 0
-1 -1 -1 3 0 0
-2 0 2 -1 0 1
0 -1 -2 2 1 0
-1 0 1 -2 1 1
-2 -1 0 1 1 1
1 -1 -3 1 2 0
-1 -1 -1 0 2 1
2 -1 -4 0 3 0
```

Warning: Don't confuse `grading` and `gb_weight`. The binoimials must be homogeneous polynomials for the grading so that the grading has no effect on picking the initial monomial. Therefore a grading (or homogeneous weight) has no effect on the Grobner basis compoutation.

A grading of the monomial algebra induces a grading on the binomials in its defining ideal such that the latter are homogeneous polynomials. With respect to this grading the output of Markov and Gröbner bases can be restricted:

`gb_degree_bound <n>` sets upper degree bound `<n>` for binomials,

`gb_min_degree <n>` sets lower degree bound `<n>` for binomials.

There is one more computation goal for monoids that complements `HilbertBasis` (switched on by default):

`Representations` representations of reducible elements in `monoid` in terms of the Hilbert basis

The outpur is a list of binomials in the file with suffix

`rep` representations of reducible elements in terms of the Hilbert basis.

Also the file with suffix `ogn` is written.

As a simple example we consider `representations.in`

```
amb_space 3
monoid 8
1 0 0
2 3 5
0 1 1
0 0 1
1 1 2
0 1 3
```

```
3 1 0
1 2 5
Representations
/*BinomialsPacked*/
```

with

```
4 Hilbert basis elements:
0 0 1
1 0 0
0 1 1
3 1 0
```

and representations of the other 4 elements in the input:

```
4
8
-1 0 -1 -1 1 0 0 0
0 0 -1 -2 0 1 0 0
-1 0 -2 -3 0 0 0 1
-2 1 -3 -2 0 0 0 0
```

The entries 1 in each row mark the reducible elements and the row should be read as a binomial vanishing on the input vectors (or monomials).

If you want to see computations that take longer than our toy example so far, run `A443monoid.in` and `Kwak80.in`.

3.1.3. Hilbert series and multiplicity

If we activate both (!) `HilbertSeries` and `Multiplicity` in `monoid.in`, the result is

```
multiplicity = 19/40
multiplicity (float) = 0.475

Hilbert series:
1 1 0 0 3 2 -2 -1 6 2 -4 0 6 1 -3 1 4 0 0 1 1
denominator with 3 factors:
1:1  2:1  20:1
...
```

followed by the representation with cyclotomic denominator and the Hilbert quasipolynomial.Activate `grading 1 -2 1` and observe the changes.

3.1.4. Binomial ideals from cone input

Defining binomial ideals can be computed not only for monoids defined by the input type `monoid`, but also for the monoids that defined by other input types as intersections of cones and

lattices, for example `cone`, `cone_and_lattice`, `equations`, `inequalities` etc.In the case of generator input there are actually two monoids, the "original monoid" as discussed in Section 7.18, and its integral closure in the lattice defined by the input. So, if we ask for the Markov basis of the defining ideal, which monoid is taken? Answer: always the integral closure generated by its Hilbert basis, unless the property makes only sense for the original monoid: if we ask `IsIntegrallyClosed`, the answer is always 'yes" for the integral closure.

As an example we take `cone_latt_markov.in` (`monoid.in` with a different input type):

```
amb_space 3
cone_and_lattice 6
1 0 0
2 3 5
0 0 1
1 1 2
0 1 3
3 1 0
MarkovBasis
SingularLocus
```

The output file contains

```
original monoid is not integrally closed in chosen lattice
...
codim singular locus = 2
18 Markov basis elements

******************************************************************************

9 lattice points in polytope (Hilbert basis elements of degree 1):
0 0 1
0 1 3
1 0 0
1 1 2
1 2 4
2 1 1
2 2 3
2 3 5
3 1 0

0 further Hilbert basis elements of higher degree:
```

The Markov basis is contained in `cone_latt_markov.mrk`:

```
18
9
-1 0 1 0 0 1 0 0 -1
0 0 0 -1 0 2 0 0 -1
...
```

```
0 -1 0 0 2 0 0 -1 0
```

The columns correspond to the Hilbert basis elements in the order they are listed above. For completeness the file with suffix `ogn` is written also in this case. It contains the Hilbert basis as listed in the `out` file.

The singular locus has codimension 2, the minimum for a normal monoid (algebra). The singular locus is stored in `cone_latt_markov.sng`.

We could equally well start from the inequalities defining the integral closure (the generators above generate the lattice $\mathbb{R}^3$) in `cone_latt_markov_supp.in`,

```
amb_space 3
inequalities 5
0   0 1
0   1 0
1   0 0
2  -3 1
5 -15 7
MarkovBasis
SingularLocus
```

with the same result as above, except that there is no originl monoid.

Sinc ethe number of monoid generaors is impossible to predict for cone input, it is not possible to givea weight vevctor.

3.2. Monoids from binomials

As an example, we consider the binomial ideal generated by

$$Y_1^2Y_2 - Y_4Y_5Y_6,\ \ Y_1Y_4^2 - Y_3Y_5Y_6,\ \ Y_1Y_2Y_3 - Y_5^2Y_6.$$

in the polynomial ring $P = K[Y_1,\dots,Y_6]$. We want to find an embedding of the toric ring it defines. When we say "defines", then we do not claim that the residue ring P/I is a toric ring. But there is a unique smallest binomial ideal $J \supset I$ with this property, and Normaliz finds the monoid and, if wanted, also a Markov (or Gröbner) basis of J. A priori $R = P/J$ is only defined as a residue class ring. It doesn't have a "canonical" embedding into another polynomial ring, but Normaliz computes such an embedding if the monoid underlying R/J is positive. As pointed out already, non-positive affine monoids an only be computed by Normaliz if they are normal.

3.2.1. Affine monoids from binomial ideals

The input type that asks for a toric ring from binomial input is

`toric_ideal`

The input vectors are obtained as the differences of the two exponent vectors in the binomials. So the input ideal `toric_ideal.in` for ur 3 binomials is

```
amb_space 6
toric_ideal 3
2 1  0 -1 -1  -1
1 0 -1  2 -1  -1
1 1  1  0 -2  -1
/* total_dgegree */
```

In order to avoid special input rules for this case in which our object is not defined as a subset of an ambient space, but as a quotient of type *generators/relations*, we abuse the name `amb_space`: it determines the space in which the input vectors live.

It is possible to define a `grading`. It must give positive degree to the unit vectors of the ambient space and degree 0 to the vectors representing the binomials so that the latter become homogeneous polynomials with respect to this grading.

In the output we get

```
6 original generators:
1 0 0
2 3 5
0 0 1
1 1 2
0 1 3
3 1 0
```

namely the residue classes of the indeterminates realized in an embedding. Test the binomials on the original generators! We know this monoid already from `monid.in`, and you can try the other computation goals discussed for the latter.

We see

```
grading:
1 1 1
```

So Normaliz uses the standard grading on the ambient polynomial ring into which R/J has been embedded. This is the default choice, as it is for the input type `monolid`. Our toric ring actually has its own standard tgrading: activate `total_degree` in the input file and look at the output. In fact, the binomials above are homogeneous in the standard gading on P, and `total_degree` sets this grading.

The generators are repeated (in this case) in a different order,m as we know already:

```
6 Hilbert basis elements:
0 0 1
1 0 0
0 1 3
1 1 2
3 1 0
```

```
2 3 5
```

Now they are sorted by degree and then lexicographically, as we always sort Hilbert bases.

As a trivial example in which the Hilbert basis does not simply repeat the original generators in a different order, compute `lin_bin.in`:

```
amb_space 2
toric_ideal 1
1 -1
```

The output contains

```
2 original generators:
1
1

1 Hilbert basis elements:
1
```

Normaliz transforms `toric_ideal` into `monoid`. Therefore onlyx the compuation goals of `monoid` are allowed with the exception of `AmbientAutomorphisms` and `InputAutomorphisms`.

A weight vector for a Gröbner basis computation can be given. Its number of components is `amb_space`. Eyample (`toric_ideal_grb`):

```
amb_space 6
toric_ideal 3
2 1  0 -1 -1  -1
1 0 -1  2 -1  -1
1 1  1  0 -2  -1
total_degree
GroebnerBasis
gb_weight
1 2 3 4 5 6
Lex
```

and the Gröbner basis is the same as for `monoid_weight.in`

3.2.2. Normalization of monoids from binomials

One go can a step further, using the input type

`normal_toric_ideal`

It asks for the *normalization* of the toric ring defined by the binomials. In `normal_toric_ideal.in` we take the same binomials as above:

```
amb_space 6
normal_toric_ideal 3
2 1  0 -1 -1  -1
```

```
1 0 -1  2 -1  -1
1 1  1  0 -2  -1
```

In the output file we find

```
6 original generators:
1 0 0
2 3 5
0 0 1
1 1 2
0 1 3
3 1 0

2 lattice points in polytope (Hilbert basis elements of degree 1):
0 0 1
1 0 0

7 further Hilbert basis elements of higher degree:
0 1 3
1 1 2
2 1 1
3 1 0
1 2 4
2 2 3
2 3 5
```

The “original generators" are the same as above, as they should be. Also the default grading is the same, and the default computation goals are identical as well. But the Hilbert series, Markov basis, Gröbner basis etc. are computed for the normalization, as the user can see by playing with the commented out computation goals. A weight vector for the Gröbner basis is not allowed since the generators of the normalization are not predictable.

Note: Until version 3.9.4 the input type `normal_toric_ideal` was called `lattice_ideal`, which has a different meaning now and is discussed in the next subsection.

`normal_toric_ideal` is transformed into `cone_and_lattice`. Thus all computation goals and options of `cone_and_lattice` can be used with the exception of `AmbientAutomorphisms` and `InputAutomorphisms`.

3.3. Lattice ideals

A lattice ideal I in a polynomial ring P is a binomial ideal modulo which all monomials are nonzerodivisors. This implies that P/J is a monoid ring whose underlying monoid is the natural image of the monoid of monomials in P. Moreover, it is a cancellative monoid, but not necessarily affine—the latter property requires torsion freeness additionally. The input type is

`lattice_ideal`

Normaliz tests whether the lattice ideral is toric and indicates it in the input file, but does not automatically treat the input like `toric_ideal` in the positive case.

A soimple example of a non-toric lattice ideal is `non_toric.in`

```
amb_space 4
lattice_ideal 4
2 -2 0 0
1 1 -1 -1
2 -1 1 -2
-1 -1 1 -1
/*GroebnerBasis
DegLex*/
```

The default computation goal is `MarkovBsis`. In our case the result is

```
4
4
-4 -1 1 0
-3 0 0 1
-2 2 0 0
6 0 0 0
```

Attention: the last binomial is $x_1^6 - 1$ so that the residue class of x_1 is a torsion element in the monoid of residue classes.

For internal reasons and the exchange of data with external programs we can ask

`IsLatticeIdealToric`

In addition to `MarkovBasis` nd `IsLatticeIdealToric` there is only one more allowed computation goal, namely `GroebnerfBasis`. There are no monoid generators for `lattice_ideal` (not even if the idfeal is toric).

Note: In versions until 3.9.4 `lattice_ideal` had the meaning of `normal_toric_ideal`.

4. The input file

The input file `<project>.in` consists of items. There are several types of them:

(1) definition of the ambient space,
(2) matrices with integer or rational entries (depending on the type),
(3) vectors with integer entries,
(4) constraints in tabular or symbolic format,
(5) a polynomial,
(6) computation goals and algorithmic variants,
(7) numerical parameters,
(8) number field definition,
(9) comments.

An item cannot include another item. In particular, comments can only be inserted between other items, but not within another item. Matrices and vectors can have two different formats, plain and formatted.

Matrices and vectors are classified by the following attributes:

(1) generators, constraints, accessory,
(2) cone/polyhedron, (affine) lattice,
(3) homogeneous, inhomogeneous.

The line structure is irrelevant for the interpretation of the input, but it is advisable to use it for the readability of the input file.

The input syntax of Normaliz 2 can still be used. It is explained in Appendix C.

4.1. Input items

4.1.1. The ambient space and lattice

The ambient space is specified as follows:

```
amb_space <d>
```

where `<d>` stands for the dimension d of the ambient vector space $\mathbb{R}^d$ in which the geometric objects live. The *ambient lattice* $\mathbb{A}$ is set to $\mathbb{Z}^d$.

Alternatively one can define the ambient space implicitly by

```
amb_space auto
```

In this case the dimension of the ambient space is determined by Normaliz from the first formatted vector or matrix in the input file. It is clear that any input item that requires the knowledge of the dimension can only follow the first formatted vector or matrix.

In the following the letter d will always denote the dimension set with `amb_space`.

An example:

```
amb_space 5
```

indicates that polyhedra and lattices are subobjects of $\mathbb{R}^5$. The ambient lattice is $\mathbb{Z}^5$.

The first non-comment input item must specify the ambient space.

4.1.2. Plain vectors

A plain vector is built as follows:

```
<T>
<x>
```

<T> denotes the type and <x> is the vector itself. The number of components is determined by the type of the vector and the dimension of the ambient space. At present, all vectors have length d.

Example:

```
grading
1 0 0
```

Normaliz allows also the input of sparse vectors. Sparse input is signalized by the key word `sparse` as the first entry. It is followed by entries of type `<col>:<val>` where `<col>` denotes the column and `<val>` the value in that column. (The unspecified columns have entry 0.) A sparse vector is terminated by the character `;` . Example:

```
grading
sparse 1:1;
```

One can also set a range of entries in sparse vectors like in

```
grading
sparse 1:1 3..5:-1 7:1;
```

which produces the vector $(1,0,-1,-1,-1,0,1,0\ldots,0)$.

For unit vectors vectors there exists a shortcuts. Example:

```
total_degree
unit_vector 25
```

4.1.3. Formatted vectors

A formatted vector is built as follows:

```
<T>
[ <x> ]
```

where <T> denotes the type and <x> is the vector itself. The components can be separated by white space, commas or semicolons. An example showing all possibilities (not recommended):

```
grading
[1,0; 0 5]
```

4.1.4. Plain matrices

A plain matrix is built as follows:

```
<T> <m>
<x_1>
...
<x_m>
```

Here <T> denotes the type of the matrix, <m> the number of rows, and <x_1>,...,<x_m> are the rows. Some types allow rational and floating point matrix entries, others are restricted to integers; see Sections 4.1.9 and 4.1.10.

The number of columns is implicitly defined by the dimension of the ambient space and the type of the matrix. Example (with amb_space 3):

```
cone 3
1/3 2 3
4 5 6
11 12/7 13/21
```

Normaliz allows the input of matrices in transposed form:

```
<T> transpose <n>
<x_1>
...
<x_m>
```

Note that <n> is now the number of *columns* of the matrix that follows it (assumed to be the number of input vectors). The number of rows is determined by the dimension of the ambient space and the type of the matrix. Example:

```
cone transpose 3
1  0   3/2
0 1/9   4
```

is equivalent to

```
cone 3
1  0
0 1/9
3/2 4
```

Like vectors, matrices have a sparse input variant, again signalized by the key word `sparse`. The rows are sparse vectors with entries `<col>:<val>`, and each row is concluded by the character `;`. Also here one can set a range of coordinates to the same value:

Example:

```
inequalities 2 sparse
1:1 2:-1;
3-5:-1;
```

chooses the 3×3 unit matrix as a matrix of type `inequalities`. Note that also in case of transposed matrices, sparse entry is row by row.

Matrices may have zero rows. Such empty matrices like

```
inhom_inequalities 0
```

can be used to make the input inhomogeneous (Section 4.1.16) or to avoid the automatic choice of the positive orthant in certain cases (Section 4.1.17). (The empty `inhom_inequalities` have both effects simultaneously.) Apart from these effects, empty matrices have no influence on the computation.

4.1.5. Formatted matrices

A formatted matrix is built as follows:

```
<T>
[ [<x_1>]
...
[<x_m>] ]
```

Here `<T>` denotes the type of the matrix and `<x_1>`,...,`<x_m>` are vectors. Legal separators are white space, commas and semicolons. An example showing all possibilities (not really recommended):

```
cone [
[ 2 1][3/7 4];
[0 1],
[9 10] [11 12/13]
]
```

Similarly as plain matrices, formatted matrices can be given in transposed form, and they can be empty.

4.1.6. Constraints in tabular format

This input type is somewhat closer to standard notation than the encoding of constraints in matrices. The general type of equations and inequalities is

```
<x> <rel> <rhs>;
```

where <x> denotes a vector of length d, <rel> is one of the relations =, <=, >=, <, > and <rhs> is a number.

Congruences have the form

```
<x> ~ <int> (<mod>);
```

where <mod> is a nonzero integer.

Examples:

```
1/2 -2 >= 0.5
1 -1/7 = 0
-1 1 ~ 7 (9)
```

Note: all numbers and relation signs must be separated by white space.

4.1.7. Constraints in symbolic format

This input type is even closer to standard notation than the encoding of constraints in matrices or in tabular format. It is especially useful if the constraints are sparse. Instead of assigning a value to a coordinate via its position in a vector, it uses coordinates named x[<n>] where <n> is the index of the coordinate. The index is counted from 1.

The general type of equations and inequalities is

```
<lhs> <rel> <rhs>;
```

where <lhs> and <rhs> denote affine linear function of the x<n> with rational coefficients. As above, <rel> is one of the relations =, <=, >=, <, >. (Both <lhs> and <rhs> must be nonempty.) Note the terminating semicolon.

Congruences have the form

```
<lhs> ~ <rhs> (<mod>);
```

where <mod> is a nonzero integer and <lhs> and <rhs> are affine linear functions with integer coefficients.

Examples:

```
1/3x[1] >= 2x[2] + 5;
x[1]+1=1/4x[2] ;
-x[1] + x[2] ~ 7 (9);
```

There is no need to insert white space for separation, but it may be inserted anywhere where it does not disrupt numbers or relation signs.

4.1.8. Polynomials

For the computation of weighted Ehrhart series and integrals Normaliz needs the input of a polynomial with rational coefficients. Moreover, one can apply polynomial constraints to lattice points in polytopes. A polynomial is first read as a string. For the computation the string is converted by the input function of CoCoALib [2]. Therefore any string representing a valid CoCoA expression is allowed. However, the names of the indeterminates are fixed: `x[1]`,...,`x[<N>` where `<N>]` is the value of `amb_space`. The polynomial must be concluded by a semicolon.

Example:

```
(x[1]+1)*(x[1]+2)*(x[1]+3)*(x[1]+4)*(x[1]+5)*
(x[2]+1)*(x[3]+1)*(x[4]+1)*(x[5]+1)*(x[6]+1)*(x[7]+1)*
(x[8]+1)*(x[8]+2)*(x[8]+3)*(x[8]+4)*(x[8]+5)*1/14400;

(x[1]*x[2]*x[3]*x[4])^2*(x[1]-x[2])^2*(x[1]-x[3])^2*
(x[1]-x[4])^2*(x[2]-x[3])^2*(x[2]-x[4])^2*(x[3]-x[4])^2;
```

4.1.9. Rational numbers

Rational numbers are allowed in input matrices, but not in all. They are *not* allowed in vectors and in matrices containing lattice generators and in congruences, namely in

`lattice`	`cone_and_lattice`	`offset`	`open_facets`
`congruences`	`inhom_congruences`	`rees_algebra`	`lattice_ideal`
`grading`	`dehomogenization`	`signs`	`strict_signs`

They are allowed in `saturation` since it defines the intersection of the vector space generated by the rows of the matrix with the integral lattice.

Avoid negative numbers as denominators.

Normaliz first reduces the input numbers to lowest terms. Then each row of a matrix is multiplied by the least common multiple of the denominators of its entries. In all applications in which the original monoid generators play a role, one should use only integers in input matrices to avoid any ambiguity.

4.1.10. Decimal fractions and floating point numbers

Normaliz accepts decimal fractions and floating point numbers in its input files. These are precisely converted to ordinary fractions (or integers). Examples:

```
1.1 --> 11/10    0.5 --> 1/2    -.1e1 --> -1
```

It is not allowed to combine an ordinary fraction and a decimal fraction in the same number. In other words, expressions like `1.0/2` are not allowed.

4.1.11. Numbers in algebraic extensions of $\mathbb{Q}$

Their format is explained in Section 8.1 together with the definition of number fields.

4.1.12. Numerical parameters

Their input has the form

```
<parameter> <n>
```

where <n> is the value assigned to <parameter>.

4.1.13. Computation goals and algorithmic variants

These are single or compound words, such as

```
HilbertBasis
Multiplicity
```

The file can contain several computation goals, as in this example.

4.1.14. Comments

A comment has the form

```
/* <text> */
```

where <text> stands for the text of the comment. It can have arbitrary length and stretch over several lines. Example:

```
/* This is a comment
*/
```

Comments are only allowed at places where also a new keyword would be allowed, especially not between the entries of a matrix or a vector. Comments can not be nested.

4.1.15. Restrictions

Input items can be combined quite freely, but there are some restrictions:

(1) The types

`cone, cone_and_lattice, polytope, rees_algebra`

exclude each other mutually.

(2) The input type `subspace` excludes `polytope` and `rees_algebra`.

(3) The types

`lattice`, `saturation`, `cone_and_lattice`

exclude each other mutually.

(4) `polytope` can not be combined with `grading`.
(5) The only type that can be combined with `lattice_ideal` is `grading`.
(6) The following types cannot be combined with inhomogeneous types or `dehomogenization`:
`polytope, rees_algebra, excluded_faces`
(7) The following types cannot be combined with inhomogeneous types:
`dehomogenization`
(8) Special restrictions apply for the input type `open_facets`; see Section 4.15.
(9) Special rules apply if precomputed data are used. See Section 7.23.
(10) For restrictions that apply to algebraic polyhedra see Section 8. Similar restrictions apply if the input types `rational_lattice` and `rational_offset` are used (see Section 7.21).
(11) The input types `monoid`, `toric_ideal`, `normal_toric_ideal` and `lattice_ideal` alloow only `grading` as a further input type.

A non-restriction: the same type can appear several times. This is useful if one wants to combine different formats, for example

```
inequalities 2 sparse
1:1;
1:1 3:-1;
inequalities 2
1 1 0 1
1 -1 -1 0
```

4.1.16. Homogeneous and inhomogeneous input

Apart from the restrictions listed in the previous section, homogeneous and inhomogeneous types can be combined as well as generators and constraints. A single inhomogeneous type or `dehomogenization` in the input triggers an inhomogeneous computation. The input item of inhomogeneous type may be an empty matrix.

4.1.17. Default values

If there is no lattice defining item, Normaliz (virtually) inserts the the unit matrix as an input item of type `lattice`. If there is no cone defining item, the unit matrix is (additionally) inserted as an input item of type `cone`.

If the input is inhomogeneous, then Normaliz provides default values for vertices and the offset as follows:

(1) If there is an input matrix type `lattice`, but no `offset`, then the offset 0 is inserted.
(2) If there is an input matrix of type cone, but no `vertices`, then the vertex 0 is inserted.

An important point. If the input does not contain any cone generators or inequalities, Normaliz automatically assumes that you want to compute in the positive orthant. In order to

avoid this choice you can add an empty matrix of `inequalities`, `inhom_inequalities` or `strict_inequalities`. This will not affect the results, but avoid the sign restriction.

4.1.18. Normaliz takes intersections

The input may contain several cone defining items and several lattice defining items. We consider homogeneous input for simplicity. Inhomogeneous input is made homogeneous anyway.

One can subdivide the input items defining cones and lattices as follows:

1. cone generators: together they generate a cone C_1;
2. cone constraints, namely inequalities and equations: they define the cone C_2;
3. lattice generators: they generate the sublattice L_1 and the vector subspace $U_1 = \mathbb{R}L1$;
4. lattice constraints, namely equations and congruences: they define the sublattice L_2 and the vector subspace $U_2 = \mathbb{R}L_2$.

The cone defined by all these data is C=$C_1 \cap C_2 \cap U_1 \cap U_2$. The lattice defined by them is $\mathbb{R}C \cap L_1 \cap L_2$.

4.2. Homogeneous generators

4.2.1. Cones

The main type is `cone`. The other two types are added for special computations.

`cone` is a matrix with d columns. Every row represents a vector, and they define the cone generated by them. Section 2.3, `2cone.in`

`subspace` is a matrix with d columns. The linear subspace generated by the rows is added to the cone. Section 7.13.4.

`polytope` is a matrix with $d-1$ columns. It is internally converted to `cone` extending each row by an entry 1. Section 2.4, `polytope.in`. This input type automatically sets `NoGradingDenom` and defines the grading $(0,\dots,0,1)$. Not allowed in combination with inhomogeneous types.

`rees_algebra` is a matrix with $d-1$ columns. It is internally converted to type `cone` in two steps: (i) each row is extended by an entry 1 to length d. (ii) The first $d-1$ unit vectors of length d are appended. Section 2.16, `MonIdeal.in`. Not allowed in combination with inhomogeneous types.

`extreme_rays` is a matrix with d columns. See Section 7.23 for its use.

`maximal_subspace` is a matrix with d columns. See Section 7.23 for its use.

Moreover, it is possible to define a cone and a lattice by the same matrix:

`cone_and_lattice` The vectors of the matrix with d columns define both a cone and a lattice. Section 2.11, `A443.in`.

If `subspace` is used in combination with `cone_and_lattice`, then the sublattice generated by its rows is added to the lattice generated by `cone_and_lattice`.

The Normaliz 2 types `integral_closure` and `normalization` can still be used. They are synonyms for `cone` and `cone_and_lattice`, respectively.

4.2.2. Lattices

There are 5 types. With the exception of `rational_lattice` and `saturation` their entries are integers.

`lattice` is a matrix with d columns. Every row represents a vector, and they define the lattice generated by them. Section 2.6.3, `3x3magiceven_lat.in`.

`rational_lattice` is a matrix with d columns. Its entries can be fractions. Every row represents a vector, and they define the sublattice of $\mathbb{Q}^d$ generated by them. See Section 7.21, `ratlat_2.in`.

`saturation` is a matrix with d columns. Every row represents a vector, and they define the lattice $U \cap \mathbb{Z}^d$ where U is the subspace generated by them. Section 2.6.3, `3x3magic_sat.in`. (If the vectors are integral, then $U \cap \mathbb{Z}^d$ is the saturation of the lattice generated by them.)

`cone_and_lattice` See Section 4.2.1.

`generated_lattice` is a matrix with d columns. See Section 7.23 for its use.

`hilbert_basis_rec_cone` is a matrix with d columns. It contains the precomputed Hilbert basis of the recession cone. See Section 7.23.3.

4.2.3. Affine monoids

`monoid` is a matrix with d columns. Every row represents a vector, and they generate a szb-monoid of $\mathbb{Z}$. See Section **??**, `monoid.in`, `A443monoid.in`.

4.3. Homogeneous Constraints

The coefficients ξ_i of the constraints are rational numbers unless indicated otherwise.

4.3.1. Cones

`inequalities` is a matrix with d columns. Every row $(\xi_1, \ldots, \xi_d)$ represents a homogeneous inequality

$$\xi_1 x_1 + \cdots + \xi_d x_d \geq 0$$

for the vectors $(x_1, \ldots, x_d) \in \mathbb{R}^d$. Sections 2.3.2, 2.5.2, `2cone_ineq.in`, `poly_ineq.in`

`equations` is a matrix with d columns. Every row $(\xi_1, \ldots, \xi_d)$ represents an equation

$$\xi_1 x_1 + \cdots + \xi_d x_d = 0$$

for the vectors $(x_1, \ldots, x_d) \in \mathbb{R}^d$. Section 2.6, `3x3magic.in`

signs is a vector with d entries in $\{-1,0,1\}$. It stands for a matrix of type `inequalities` composed of the sign inequalities $x_i \geq 0$ for the entry 1 at the i-th component and the inequality $x_i \leq 0$ for the entry -1. The entry 0 does not impose an inequality. See Section 2.15.2, `InhomCongSigns.in`.

excluded_faces is a matrix with d columns. Every row $(\xi_1,\ldots,\xi_d)$ represents an inequality

$$\xi_1 x_1 + \cdots + \xi_d x_d > 0$$

for the vectors $(x_1,\ldots,x_d) \in \mathbb{R}^d$. It is considered as a homogeneous input type though it defines inhomogeneous inequalities. The faces of the cone excluded by the inequalities are excluded from the Hilbert series computation, but `excluded_faces` behave like `inequalities` in almost every other respect. Section 2.10.1, `CondorcetSemi.in`. Also see Section 7.20.

support_hyperplanes is a matrix with d columns. See Section 7.23.

A useful shortcut:

nonnegative inserts the sign inequalities $x_i \geq 0$ for all coordinates. See `Condorcet.in`.

4.3.2. Lattices

congruences is a matrix with $d+1$ columns. Each row $(\xi_1,\ldots,\xi_d,c)$ represents a congruence

$$\xi_1 z_1 + \cdots + \xi_d z_d \equiv 0 \mod c, \qquad \xi_i, c \in \mathbb{Z},$$

for the elements $(z_1,\ldots,z_d) \in \mathbb{Z}^d$. Section 2.6.2, `3x3magiceven.in`.

4.4. Inhomogeneous generators

4.4.1. Polyhedra

vertices is a matrix with $d+1$ columns. Each row $(p_1,\ldots,p_d,q)$, $q>0$, specifies a generator of a polyhedron (not necessarily a vertex), namely

$$v_i = \left(\frac{p_1}{q},\ldots,\frac{p_n}{q}\right), \qquad p_i \in \mathbb{Q}, q \in \mathbb{Q}_{>0},$$

Section 2.9.1, `InhomIneq_gen.in`

Note: `vertices` and `cone` together define a polyhedron. If `vertices` is present in the input, then the default choice for `cone` is the empty matrix.

The format of `vertices` was introduced when Normaliz only accepted integer numbers in its input. There is no need for an extra denominator anymore, but for backward compatibility the format has not been changed.

The Normaliz 2 input type `polyhedron` can still be used.

4.4.2. Affine lattices

offset is a vector with d integer entries. It defines the origin of the affine lattice. Section 2.15.1, InhomCongLat.in.

rational_offset is a vector with d rational entries. It defines the origin of the rational affine lattice. Section 7.21, ratlat_2.in.

Note: offset and lattice (or saturation) together define an affine lattice. If offset is present in the input, then the default choice for lattice is the empty matrix.

4.5. Inhomogeneous constraints

4.5.1. Polyhedra

inhom_inequalities is a matrix with $d+1$ columns. We consider inequalities

$$\xi_1 x_1 + \cdots + \xi_d x_d \geq \eta,$$

rewritten as

$$\xi_1 x_1 + \cdots + \xi_d x_d + (-\eta) \geq 0$$

and then represented by the input vectors

$$(\xi_1, \ldots, \xi_d, -\eta).$$

Section 2.9, InhomIneq.in.

inhom_equations is a matrix with $d+1$ columns. We consider equations

$$\xi_1 x_1 + \cdots + \xi_d x_d = \eta,$$

rewritten as

$$\xi_1 x_1 + \cdots + \xi_d x_d + (-\eta) = 0$$

and then represented by the input vectors

$$(\xi_1, \ldots, \xi_d, -\eta).$$

See Section 2.7NumSemi.in.

strict_inequalities is a matrix with d columns. We consider inequalities

$$\xi_1 x_1 + \cdots + \xi_d x_d \geq 1,$$

represented by the input vectors

$$(\xi_1, \ldots, \xi_d).$$

Section 2.3.3, 2cone_int.in.

strict_signs is a vector with d components in $\{-1, 0, 1\}$. It is the "strict" counterpart to signs. An entry 1 in component i represents the inequality $x_i > 0$, an entry -1 the opposite inequality, whereas 0 imposes no condition on x_i. Section 2.10.2, Condorcet_one.in

inhom_excluded_faces is a matrix with $d+1$ columns. Every row $(\xi_1,\dots,\xi_d,-\eta)$ represents an inequality

$$\xi_1 x_1 + \cdots + \xi_d x_d > \eta$$

for the vectors $(x_1,\dots,x_d) \in \mathbb{R}^d$. The faces of the polyhedron excluded by the inequalities are excluded from the Hilbert and Ehrhart series series computation, but `inhom_excluded_faces` behave like `inhom_inequalities` in almost every other respect. See Section 7.20.

4.5.2. Affine lattices

inhom_congruences We consider a matrix with $d+2$ columns. Each row $(\xi_1,\dots,\xi_d,-\eta,c)$ represents a congruence

$$\xi_1 z_1 + \cdots + \xi_d z_d \equiv \eta \mod c, \qquad \xi_i,\eta,c \in \mathbb{Z},$$

for the elements $(z_1,\dots,z_d) \in \mathbb{Z}^d$. Section 2.15, `InhomCongSigns.in`.

4.6. Tabular constraints

constraints <n> allows the input of `<n>` equations, inequalities and congruences in a format that is close to standard notation. As for matrix types the keyword `constraints` is followed by the number of constraints. The syntax of tabular constraints has been described in Section 4.2.1. If $(\xi_1,\dots,\xi_d)$ is the vector on the left hand side and η the number on the right hand side, then the constraint defines the set of vectors $(x_1,\dots,x_d)$ such that the relation

$$\xi_1 x_1 + \cdots + \xi_d x_d \ \texttt{rel}\ \eta$$

is satisfied, where `rel` can take the values $=,\le,\ge,<,>$ with the represented by input strings `=,<=,>=,<,>`, respectively.

Tabular constraints cannot be used for `excluded_faces` or `inhom_excluded_faces`.

A further choice for `rel` is `~`. It represents a congruence $\equiv$ and requires the additional input of a modulus: the right hand side becomes $\eta(c)$. It represents the congruence

$$\xi_1 x_1 + \dots \xi_d x_d \equiv \eta \pmod c.$$

Sections 2.3.3, `2cone_int.in`, 2.6.2, `3x3magiceven.in`, 2.9, `InhomIneq.in`.

A right hand side $\neq 0$ makes the input inhomogeneous, as well as the relations $<$ and $>$. Strict inequalities are always understood as conditions for integers. So

$$\xi_1 x_1 + \cdots + \xi_d x_d < \eta$$

is interpreted as

$$\xi_1 x_1 + \dots \xi_d x_d \le \eta - 1,$$

4.6.1. Forced homogeneity

It is often more natural to write constraints in inhomogeneous form, even when one wants the computation to be homogeneous. The type `constraints` does not allow this. Therefore we have introduced

`hom_constraints` for the input of equations, non-strict inequalities and congruences in the same format as `constraints`, except that these constraints are meant to be for a homogeneous computation. It is clear that the left hand side has only $d-1$ entries now. See Section 2.5.2, `poly_hom_const.in`.

4.7. Symbolic constraints

The input syntax is

`constraints <n> symbolic` where `<n>` is the number of constraints in symbolic form that follow.

The constraints have the form described in Section 4.1.7. Note that every symbolic constraint (including the last) must be terminated by a semicolon.

See Sections 2.7, `NumSemi.in`, 2.15, `InhomCong.in`.

The interpretation of homogeneity follows the same rules as for tabular constraints. The variant `hom_constraints` is allowed and works as for tabular constraints.

4.8. Converting equations to inequalities

The directive

`convert_equations`

tells Normaliz to replace eevery equation in the input file by a pair of equations. It is meant for very large computations of lattice points by project-and-lift since it avoids a coordinate transformation that is not needed. The default choice of `nonnegative` in the absence of explicit inequalities is left untouched by `convert_equationst`. See `pet.in` and `baby.in` (for which the coordinate transformation would be harmless).

4.9. Polynomial constraints

Normaliz can apply polynomial constraints to lattice points in polytopes. The input syntax is

`polynomial_equations <n>`
`polynomial_inequalities <n>`

where `<n>` is the number of polynomials that follow. The equations defined by a polynomial f is always given by $f(x) = 0$, and the inequality is $f(x) \geq 0$. Therefore no relation signs or "right hand sides" are allowed.Don't forget to conclude every polynomial by a semicolon.

See `pet.in`, `baby.in` and Section 7.2.5.

4.10. Binomial ideals

There are three types of input for binomial ideals. The rows of the matrices coming with these input types represent binomials. The representation nof binomials by vecors is discussed at the beginjing of Section n3. The mathematics of these input types is discussed in Section **??**.

The input types differ in the object computed frtom them.

`lattice_ideal` is an integer matrix with d columns. The object computed from the binomials in it is the smallest lattice ideal containing them. Section 3.3, `non_toric.in`.

`toric_ideal` is an integer matrix with d columns. The object computed from the binomials in it is the smallest toric ideal containing them and the toric ring whose defining ideal the latter is. Section 3.2.1, `toric_ideal.in`.

`normal_toric_ideal` is an intehger matrix with d columns. The object computed from the binomials in it is the the normalization of the toric ring it defines. Section 3.2.2, `normal_toric_ideal.in`.

4.11. Unit vectors and unit matrix

A grading or a dehomogenization is often given by a unit vector:

`unit_vector <n>` represents the n-th unit vector in $\mathbb{R}^d$ where n is the number given by `<n>`.

This shortcut cannot be used as a row of a matrix. It can be used whenever a single vector is asked for, namely after `grading`, `dehomogenization`, `signs` and `strict_signs`. See Section 2.5, `rational.in`.

The unit matrix can be given to every input type that expects a matrix:

`unit_matrix`

Example:

```
cone unit_matrix
```

The number of rows is defined by `amb_space` and the type of the matrix, as usual.

4.12. Grading

This type is accessory. A $\mathbb{Z}$-valued grading can be specified in two ways:

(1) *explicitly* by including a grading in the input, or

(2) *implicitly*. In this case Normaliz checks whether the extreme integral generators of the monoid lie in an (affine) hyperplane A given by an equation $\lambda(x) = 1$ with a $\mathbb{Z}$-linear form λ. If so, then λ is used as the grading.

Implicit gradings are only possible for homogeneous computations.

If the attempt to find an implicit grading causes an arithmetic overflow and `verbose` has been set (say, by the option `-c`), then Normaliz issues the warning

```
Giving up the check for a grading
```

If you really need this check, rerun Normaliz with a bigger integer type.

Explicit definition of a grading:

`grading` is a vector of length d representing the linear form that gives the grading. Section 2.5, `rational.in`.

`total_degree` represents a vector of length d with all entries equal to 1. Section 2.10, `Condorcet.in`.

Before Normaliz can apply the degree, it must be restricted to the effective lattice $\mathbb{E}$. Even if the entries of the grading vector are coprime, it often happens that all degrees of vectors in $\mathbb{E}$ are divisible by a greatest common divisor $g > 1$. Then g is extracted from the degrees, and it will appear as `denominator` in the output file.

Normaliz checks whether all generators of the (recession) monoid have positive degree (after passage to the quotient modulo the unit group in the nonpointed case). Vertices of polyhedra may have degrees ≤ 0.

4.12.1. With binomial ideal input

In this case the unit vectors correspond to generators of the monoid. Therefore the degrees assigned to them must be positive. Moreover, the vectors in the input represent binomial relations, and these must be homogeneous. In other words, both monomials in a binomial must have the same degree. This amounts to the condition that the input vectors have degree 0. Normaliz checks this condition.

4.13. Dehomogenization

Like `grading` this is an accessory type.

Inhomogeneous input for objects in $\mathbb{R}^d$ is homogenized by an additional coordinate and then computed in $\mathbb{R}^{d+1}$, but with the additional condition $x_{d+1} \geq 0$, and then dehomogenizing all results: the substitution $x_{d+1} = 1$ acts as the *dehomogenization*, and the inhomogeneous input types implicitly choose this dehomogenization.

Like the grading, one can define the dehomogenization explicitly:

`dehomogenization` is a vector of length d representing the linear form δ.

The dehomogenization can be any linear form δ satisfying the condition $\delta(x) \geq 0$ on the cone that is truncated. (In combination with constraints, the condition $\delta(x) \geq 0$ is automatically satisfied since δ is added to the constraints.)

The input type `dehomogenization` can only be combined with homogeneous input types, but makes the computation inhomogeneous, resulting in inhomogeneous output. The polyhedron

computed is the intersection of the cone $\mathbb{C}$ (and the lattice $\mathbb{E}$) with the hyperplane given by $\delta(x) = 1$, and the recession cone is $\mathbb{C} \cap \{x : \delta(x) = 0\}$.

A potential application is the adaptation of other input formats to Normaliz. The output must then be interpreted accordingly.

Section 7.11, `dehomogenization.in`.

4.14. Weight vector for Gröbner bases

For the computation of Gröbner bases one can specify a weight vedtor by

`gb_weight`

It is a vecor with nonnegative entries for `Lex` as a tiebrteaker and positive entries for `RevLeX` (default choice). The length derpends on the type of input. See Section **??** for a discussion and examples.

4.15. Open facets

The input type `open_facets` is similar to `strict_inequalities`. However, it allows to apply strict inequalities that are not yet known. This makes only sense for simplicial polyhedra where a facet can be identified by the generator that does *not* lie in it.

`open_facets` is a vector with entries $\in \{0,1\}$.

The restrictions for the use of open facets are the following:

(1) Only the input types `cone`, `vertices` and `grading` can appear together with `open_facets`.
(2) The vectors in `cone` are linearly independent.
(3) There is at most one vertex.

The number of vectors in `cone` may be smaller than d, but `open_facets` must have d entries.

`open_facets` make the computation inhomogeneous. They are interpreted as follows. Let v be the vertex—if there are no `vertices`, then v is the origin. The shifted $C' = v + C$ is cut out by affine-linear inequalities $\lambda_i(x) \geq 0$ with coprime integer coefficients. We number these in such a way that $\lambda_i(v + c_i) \neq 0$ for the generators c_i of C (in the input order), $i = 1, \ldots, n$. Then all subsequent computations are applied to the shifted cone $C'' = v' + C$ defined by the inequalities

$$\lambda_i(x) \geq u_i$$

where the vector $(u_1, \ldots, u_d)$ is given by `open_facets`. (If $\dim C < d$, then the entries u_j with $j > \dim C$ are ignored.)

That 1 indicates "open" is in accordance with its use for the disjoint decomposition; see Section 7.14.2. Section 7.19 discusses an example.

4.16. Coordinates for projection

The coordinates of a projection of the cone can be chosen by

projection_coordinates It is a 0-1 vector of length d.

The entries 1 mark the coordinates of the image of the projection. The other coordinates give the kernel of the projection. See Section 7.12 for an example.

4.17. Numerical parameters

Certain numerical parameters used by Normaliz can (only) be set in the input file.

4.17.1. Degree bound for series expansion

It can be set by

expansion_degree <n>

where <n> is the number of coefficients to be computed and printed. See Section 7.10.

4.17.2. Number of significant coefficients of the quasipolynomial

It can be set by

nr_coeff_quasipol <n>

where <n> is the number of highest coefficients to be printed. See Section 7.10.3.

4.17.3. Codimension bound for the face lattice

It can be set by

face_codim_bound <n>

where <n> is the bound for the codimension of the faces to be computed.

4.17.4. Degree bounds for Markov and Gröbbner bases

gb_degree_bound <n> sets the upper bound <n> for Markov and Gröbner bases,
gb_min_degree <n> sets the lower bound <n> for Markov and Gröbner bases.

4.17.5. Number of digits for fixed precision

The computation of vilumes by signed decomposition can be done with a fixed precision. It is et by

decimal_digits <n>

where <n> sets the precision to 10^{-n}.

4.17.6. Block size for distributed computation

See Appendix F.1 for an explanation. It is set by

block_size_hollow_tri <n>

4.18. Pointedness

Since version 3.1 Normaliz can also compute nonpointed cones and polyhedra without vertices.

4.19. The zero cone

The zero cone with an empty Hilbert basis is a legitimate object for Normaliz. Nevertheless a warning message is issued if the zero cone is encountered.

5. Computation goals and algorithmic variants

The library `libnormaliz` contains a class `ConeProperties` that collects computation goals, algorithmic variants and additional data that are used to control the work flow in `libnormaliz` as well as the communication with other programs. The latter are not important for the Normaliz user, but are listed as a reference for `libnormaliz`. See Appendix D for a description of `libnormaliz`.

All computation goals and algorithmic variants can be communicated to Normaliz in two ways:

(1) in the input file, for example `HilbertBasis`,
(2) via a verbatim command line option, for example `--HilbertBasis`.

For the most important choices there are single letter command line options, for example `-N` for `HilbertBasis`. The single letter options ensure backward compatibility to Normaliz 2. In jNormaliz they are also accessible via their full names.

Some computation goals apply only to homogeneous computations, and some others make sense only for inhomogeneous computations.

Some single letter command line options combine two or more computation goals, and some algorithmic variants imply computation goals.

There are restrictions for algebraic polyhedra. See Section 8.3.

5.1. Default choices and basic rules

If several computation goals are set, all of them are pursued. In particular, computation goals in the input file and on the command line are accumulated. But

`--ignore, -i` on the command line switches off the computation goals and algorithmic variants set in the input file.

The default computation goal is set if neither the input file nor the command line contains a computation goal or an algorithmic variant that implies a computation goal. The deafault computatin goal depends on the input type.

- Except the input of a monoid or binomial ideal it is
  `SupportHyperplanes` + `HilbertBasis` + `HilbertSeries` .
  In the homogeneous case, `ClassGroup` is included as well.
- For `monoid`, `toric_ideal` and `normal_toric _ideal` it is `HilbertBasis` + `IsIntegrallyClosed` for the momnoid derived from them.
- For the input type `lattice_ideal` it is `MarkovBasis`.

If set explicitly in the input file or on the command line the following adds these computation goals:

`DefaultMode`

`DefaultMode` can be set explicitly in addition to other computation goals. If it is set, implicitly

or explicitly, Normaliz will not complain about unreachable computation goals.

5.2. Computation goals

Almost always the computation goals set explicitly or by default require the comoputation of auxiliary data that themselves can be asked for by explicit computation goals. In most cases the results of these computations appear in the output. In case of doubt set exlicit computation goals.

5.2.1. Lattice data

`Sublattice, -S` (upper case S) asks Normaliz to compute the coordinate transformation to and from the efficient sublattice.

5.2.2. Support hyperplanes and extreme rays

`SupportHyperplanes, -s` triggers the computation of support hyperplanes and extreme rays.

Normaliz tries to find a grading in the homogeneous case.

`VerticesFloat` converts the format of the vertices to floating point. It implies `SupportHyperplanes`.

`SuppHypsFloat` converts the format of the support hyperplanes to floating point. It implies `SupportHyperplanes`.

`ExtremeRaysFloat` does the same for the extreme rays.

Note that `VerticesFloat` and `SuppHypsFloat` are not pure output options. They are computation goals, and therefore break implicit `DefaultMode`.

`ProjectCone` Normaliz projects the cone defined by the input data onto a subspace generated by selected coordinate vectors and computes the image with the goal `SupportHyperplanes`.

5.2.3. Hilbert basis and lattice points

`HilbertBasis, -N` triggers the computation of the Hilbert basis. In inhomogeneous computations it asks for the Hilbert basis of the recession monoid *and* the module generators.

`WitnessNotIntegrallyClosed, -w` With this option, Normaliz stops the Hilbert basis computation as soon it has found a witness confirming that the original monoid is not integrally closed.

`Deg1Elements, -1` restricts the computation to the degree 1 elements of the Hilbert basis in homogeneous computations (where it requires the presence of a grading).

`LatticePoints` is identical to `Deg1Elements` in the homogeneous case, but implies `NoGradingDenom`. In inhomogeneous computations it is a synonym for `HilbertBasis`.

ModuleGeneratorsOverOriginalMonoid, -M computes a minimal system of generators of the integral closure over the original monoid (see Section 7.18). Requires the existence of original monoid generators.

The boolean valued computation goal `IsIntegrallyClosed` is also related to the Hilbert basis; see Section 5.2.15.

5.2.4. Enumerative data

The computation goals in this section require a grading. They include `SupportHyperplanes`.

HilbertSeries,-q triggers the computation of the Hilbert series.

EhrhartSeries computes the Ehrhart series of a polytope, regardless of whether it is defined by homogeneous or inhomogeneous input. In the homogeneous case it is equivalent to `HilbertSeries + NoGradingDenom`, but not in the inhomogeneous case. See the discussion in Section 7.1. Can be combined with `HSOP`.

Multiplicity, -v restricts the computation to the multiplicity.

Volume, -V computes the lattice normalized and the Euclidean volume of a polytope given by homogeneous or inhomogeneous input (implies `Multiplicity` in the homogeneous case, but also sets `NoGradingDenom`).

HSOP lets Normaliz compute the degrees in a homogeneous system of parameters and the induced representation of the Hilbert or Ehrhart series series. Note that `HSOP` does not imply `HilbertSeries` or `EhrhartSeries`.

NoPeriodBound This option removes the period bound that Normaliz sets for the computation of the Hilbert quasipolynomial (presently 10^6).

NumberLatticePoints finds the number of lattice points. They are not stored.

5.2.5. Combined computation goals

Can only be set by single letter command line options:

-n `HilbertBasis + Multiplicity`

-h `HilbertBasis + HilbertSeries`

-p `Deg1Elements + HilbertSeries`

5.2.6. The class group

ClassGroup, -C is self explanatory, includes `SupportHyperplanes`. Not allowed in inhomogeneous computations.

5.2.7. Integer hull

IntegerHull, -H computes the integer hull of a polyhedron. Implies the computation of the lattice points in it.

More precisely: in homogeneous computations it implies `Deg1Elements`, in inhomogeneous computations it implies `HilbertBasis`. See Section 2.14.

5.2.8. Triangulation and Stanley decomposition

`Triangulation, -T` makes Normaliz compute, store and export the full triangulation.

`ConeDecomposition, -D` Normaliz computes a disjoint decomposition of the cone into semi-open simplicial cones. Implies `Triangulation`.

`TriangulationSize, -t` makes Normaliz count the simplicial cones in the full triangulation.

`TriangulationDetSum` makes Normaliz additionally sum the absolute values of their determinants.

`StanleyDec, -y` makes Normaliz compute, store and export the Stanley decomposition.

`AllGeneratorsTriangulation` makes Normaliz compute and store a triangulation that uses all generators.

`LatticePointTriangulation` makes Normaliz compute and store a triangulation that uses all lattice points in a polytope.

`UnimodularTriangulation` makes Normaliz compute and store a unimodular triangulation.

The triangulation and the Stanley decomposition are treated separately since they can become very large and may exhaust memory if they must be stored for output.

Note that these decompositions cannot be computed for a polyhedron that is unbounded (modulo its maximal subspace). However, they are allowed for polytopes defined by inhomogeneous input. `UnimodularTriangulation` is only allowed in homogeneous computations and is excluded for algebraic polyhedra.

The following triangulations are defined by the order of the generators. See SEctions 7.15.5 and 7.15.6.

`PlacingTriangulation`

`PullingTriangulation`

5.2.9. Face structure

The f-vector of a polyhedron is computed by

`FVector`

The set of faces of a polyhedron is computed by

`FaceLattice`

Like the triangulation or Stanley decomposition the face lattice can become very large, and it is already computed with `FVector`. `FaceLattice` writes an extra output file. The details of its representation in the extra output file are discussed in Section 7.17.

The face lattice computation is based on the incidence vectors of the facets. It is possible to retrieve this matrix (independently of `FVector` or `FaceLattice`) via the computation goal

`Incidence`

Section 7.17 as well. See it also for the dual versions

DualFVector
DualFaceLattice
DualIncidence

5.2.10. Semiopen polyhedra

IsEmptySemiopen

asks for the emptiness of a semiopen polyhedron. See Section 7.20.

5.2.11. Automorphism groups

Automorphism groups are defined in Section 7.22.

Automorphisms computes the integral automorphisms of rational polyhedra and the algebraic automorphisms of algebraic polytopes.

RationalAutomorphisms computes the rational automorphisms of rational polytopes.

EuclideanAutomorphisms computes the euclidean automorphisms of rational and algebraic polytopes.

CombinatorialAutomorphisms computes ate combinatorial automorphisms of polyhedra.

AmbientAutomorphisms computes automorphisms induce by permutations of coordinates of the ambient space.

InputAutomorphisms computes taional (or algebraic) automorphisms based solely on the input and initial coordinate transformations.

5.2.12. Weighted Ehrhart series and integrals

WeightedEhrhartSeries, -E makes Normaliz compute a generalized Ehrhart series.

VirtualMultiplicity, -L makes Normaliz compute the virtual multiplicity of a weighted Ehrhart series.

Integral, -I makes Normaliz compute an integral over a polytope. Implies `NoGradingDenom`.

These computation goals require a homogeneous computation.

Don't confuse these options with symmetrization. The latter symmetrizes (if possible) the given data and uses `-E` or `-L` internally on the symmetrized object. The options `-E,-I,-L` ask for the input of a polynomial. See Section 4.1.8.

5.2.13. Markov and Gröbner bases

They are discussed in Section 3.

MarkovBasis computes a system of generators for a toric ideal defining a monoid or a lattice ideal.

GroebnerBasis computes a system of generators for such ideals.

Representations compites the represenztation of the reducible elements in a generating system of an affine monoid by the Hilbert basis.

Lex sets the lexicographic monomial order for Gröbner bases,

RevLex sets the degree reverse lexicographic order,

DegLex sets the degree lexicographic order.

5.2.14. Local structure

SingularLocus comoputes the singular locus of an affine monoid (algebra),

CodimSingularLocus computes its codimension.

5.2.15. Boolean valued computation goals

They tell Normaliz to find out the answers to the questions they ask. Two of them are more important than the others since they may influence the course of the computations:

IsIntegrallyClosed : is the original monoid integrally closed? Normaliz stops the Hilbert basis computation as soon as it can decide whether the original monoid contains the Hilbert basis (see Section 2.11.1). Normaliz tries to find the answer as quickly as possible. This may include the computation of a witness, but not necessarily. If you need a witness, use `WitnessNotIntegrallyClosed, -w`.

IdsSerreR1 checks the Serre property (R_1) for ffine monoids (automatically satisfied by normal monoids).

IsPointed : is the efficient cone $\mathbb{C}$ pointed? This computation goal is sometimes useful to give Normaliz a hint that a nonpointed cone is to be expected. See Section 7.13.3.

For the following we only need the support hyperplanes and the lattice:

IsGorenstein, -G : is the monoid of lattice points Gorenstein? In addition to answering this question, Normaliz also computes the generator of the interior of the monoid (the canonical module) if the monoid is Gorenstein. (Only in homogeneous computations.)

The remaining ones:

IsDeg1ExtremeRays : do the extreme rays have degree 1? (Only in homogeneous computations.)

IsDeg1HilbertBasis : do the Hilbert basis elements have degree 1? (Only in homogeneous computations.)

IsReesPrimary : for the input type `rees_algebra`, is the monomial ideal primary to the irrelevant maximal ideal?

IsLatticeIdealToric asks whether the `lattice ideal` in the input is actually toric

The last three computation goals are not really useful for Normaliz since they will be answered automatically. Note that they may trigger extensive computations.

5.3. Integer type

There is no need to worry about the integer type chosen by Normaliz. All preparatory computations use infinite precision. The main computation is then tried with 64 bit integers. If it fails, it will be restarted with infinite precision.

Infinite precision does not mean that overflows are completely impossible. In fact, Normaliz requires numbers of type "degree" fit the type `long` (typically 64 bit on 64 bit systems). If an overflow occurs in the computation of such a number, it cannot be remedied.

The amount of computations done with infinite precision is usually very small, but the transformation of the computation results from 64 bit integers to infinite precision may take some time. If you need the highest possible speed, you can suppress infinite precision completely by

`LongLong`

With this option, Normaliz cannot restart a failed computation.

On the other hand, the 64 bit attempt can be bypassed by

`BigInt, -B`

Note that Normaliz tries to avoid overflows by intermediate results (even if `LongLong` is set). If such overflow should happen, the computation is repeated locally with infinite precision. (The number of such GMP transitions is shown in the terminal output.) If a final result is too large, Normaliz must restart the computation globally.

`LongLong` is not a cone property.

Caveat. The overflow check of Normaliz is not an absolute guarantee. The probability that it fails is microscopically small, but failure is not totally excluded. Very critical computations for which one has no other confirmation should be redone in `BigInt`.

5.4. The choice of algorithmic variants

For its main computation goals Normaliz has algorithmic variants. It tries to choose the variant that seems best for the given input data. This automatic choice may however be a bad one. Therefore the user can completely control which algorithmic variant is used.

5.4.1. Primal vs. dual

For the computation of Hilbert bases Normaliz has two algorithms, the primal algorithm that is based on triangulations, and the dual algorithm that is of type "pair completion". We have seen both in Section 2. Roughly speaking, the primal algorithm is the first choice for generator input, and the dual algorithm is usually better for constraints input. The choice also applies to the computation of degree 1 elements. However, for them the default choice is project-and-lift (well, almost always). See Section 7.2.1. The conditions under which the dual algorithm is chosen are specified in Section 7.5.

The choice of the algorithm can be fixed or blocked:

DualMode, -d activates the dual algorithm for the computation of the Hilbert basis and degree 1 elements. Includes `HilbertBasis`, unless `Deg1Elements` is set. It overrules `IsIntegrallyClosed`.

PrimalMode, -P blocks the use of the dual algorithm.

The automatic choice can of course fail. See Section 7.5 for an example for which it is bad.

5.4.2. Lattice points in polytopes

For this task Normaliz has several methods. They are discussed in Section 7.2. The default choice is the project-and-lift algorithm. It can be chosen explicitly:

Projection, -j

NoProjection blocks it.

Alternative choices are

ProjectionFloat, -J , project-and-lift with floating point arithmetic,

PrimalMode, -P , triangulation based method,

Approximate, -r , approximation of rational polytopes followed by triangulation and

DualMode, -d , dual algorithm.

Note: none of these algorithmic variants implies the computation of the lattice points. They must be asked for by a computation goal.

The following options modify `Projection` and `ProjectionFloat`:

NoLLL blocks the use of LLL reduced coordinates,

NoRelax blocks relaxation.

Both LLL and relaxation are switched on by default. See Section 7.2.3.

For positive systems (see Section 7.2.4) Normaliz chooses "coarse projection", and it may use a patching variant of project-and-lift. These choices can be blocked by

NoCoarseProjection

NoPatching

Moreover, there are further options by which the order, in which the "patches" are processed, can be influenced. See Section 7.2.6.

5.4.3. Bottom decomposition and order

Bottom decomposition is a way to produce an optimal triangulation for a given set of generators. It is discussed in Section 7.3. The criterion for its automatic choice is explained there. It can be forced or blocked:

BottomDecomposition, -b tells Normaliz to use bottom decomposition in the primal algorithm.

NoBottomDec, -o forbids Normaliz to use bottom decomposition in the primal algorithm, even if it would otherwise be chosen because of large roughness (see Section 7.3).

An option to be mentioned in this context is

KeepOrder, -k forces Normaliz to insert the generators (for generator input) or the inequalities (for constraint input) in the input order. This option is useful if the input has been produced in a systematic order that would be destroyed by the degree-lexicogrpahic order applied by Normaliz. Also blocks `BottomDecomposition`.

5.4.4. Multiplicity, volume and integrals

For the computation of multiplicities Normaliz offers has three main algorithms:

(1) the computation and evaluation of a full triangulation,
(2) descent in the face lattice,
(3) signed decomposition.

These are described in more detail in Section 7.6. Moreover, one can use symmetrization (see below), and (2) has a variant using isomorphism types.

Normaliz tries them by default in the order signed decoposition, descent, symmetritation and uses the first for which the default conditions are satisfied (as long as there is no need to compute a full triangulation for other reasons). The last resort is (1).

The options asking explicitly for an algorithm or excluding it are

Descent, -F

NoDescent

SignedDec

NoSignedDec

The variant using isomorphism types can be activated by

Descent ExploitIsosMult

You van ask for

StrictTypeChecking

if you don't btrust SHA256 hash values. See Section 7.6.3.

Another option to be mentioned in this context is

FixedPrecision

It can be applied if the multiplicity is computed by signed decomposition. See Section 7.6.5

For integrals one can chose either the standard triangulation or signed decomposition. In the latter case `FixedPrecision` is also available.

If one wants to compute multiplicities (or volumes) with signed decomposition, it is possible to split the most time consuming part of the computation into blocks that can be processed independently of each other. It can be asked for by the option

DistributedComp

Distributed computation is described in Appendix F.1.

5.4.5. Symmetrization

In rare cases Normaliz can use symmetrization in the computation of multiplicities or Hilbert series. If applicable, this is a very strong tool. We have mentioned it in Section 2.10 and will discuss it in Section 7.8. It will be chosen automatically, but can also be forced or blocked:

`Symmetrize, -Y` lets Normaliz compute the multiplicity and/or the Hilbert series via symmetrization (or just compute the symmetrized cone).

`NoSymmetrization` blocks symmetrization.

The integration involved in symmetrization can be done by signed decomposition.

5.4.6. Subdivision of simplicial cones

Subdivision requires enlarging the set of generators and can lead to a nested triangulation (see Sections 7.4 and 7.14.1). The subdivision can be blocked by

`NoSubdivision`

5.4.7. Options for the grading

By setting

`NoGradingDenom`

you can force Normaliz not to change the original grading if it would otherwise divide it by the grading denominator. It is implied by several computation goals for polytopes. See Section 7.1.

`NoGradingDenom` *is set automatically inn inhomogeneous computations.*

By

`GradingIsPositive`

the user guarantees that the grading is positive. This option can be useful in rare cases if Normaliz would otherwise compute extreme rays only to check the positivity of the grading.

5.5. Control of computations and communication with interfaces

In addition to the computation goals in Section 5.2, the following elements of `ConeProperties` control the work flow in `libnormaliz` and can be used by programs calling Normaliz to ensure the availability of the data that are controlled by them.

`OriginalMonoidGenerators` controls the generators of the original monoid.

`ModuleGenerators` controls the module generators in inhomogeneous computation.

`ExtremeRays` controls the extreme rays.

VerticesOfPolyhedron controls the vertices of the polyhedron in the inhomogeneous case.

MaximalSubspace controls the maximal linear subspace of the (homogenized) cone.

EmbeddingDim controls the embedding dimension.

Rank controls the rank.

RecessionRank controls the rank of the recession monoid in inhomogeneous computations.

AffineDim controls the affine dimension of the polyhedron in inhomogeneous computations.

ModuleRank in inhomogeneous computations it controls the rank of the module of lattice points in the polyhedron as a module over the recession monoid.

ExcludedFaces controls the excluded faces.

InclusionExclusionData controls data derived from the excluded faces.

Grading controls the grading.

GradingDenom controls its denominator.

Dehomogenization controls the dehomogenization.

ReesPrimaryMultiplicity controls the multiplicity of a monomial ideal, provided it is primary to the maximal ideal generated by the indeterminates. Used only with the input type `rees_algebra`.

EuclideanVolume controls the Euclidean volume.

GeneratorOfInterior controls the generator of the interior if the monoid is Gorenstein.

CoveringFace asks for an excluded face making the semiopen polyhedron empty.

Equations controls the equations.

Congruences controls the congruences.

ExternalIndex controls the external index.

InternalIndex controls the internal index.

UnitGroupIndex controls the unit group index.

IsInhomogeneous controls the inhomogeneous case.

HilbertQuasiPolynomial controls the Hilbert quasipolynomial.

EhrhartQuasiPolynomial controls the Ehrhart quasipolynomial.

WeightedEhrhartQuasiPolynomial controls the weighted Ehrhart quasipolynomial.

IsTriangulationNested controls the indicator of this property.

IsTriangulationPartial similar.

5.6. Rational and integer solutions in the inhomogeneous case

The integer solutions of a homogeneous diophantine system generate the rational solutions as well: every rational solution has a multiple that is an integer solution. Therefore the rational solutions do not need an extra computation. If you prefer geometric language: a rational cone is generated by its lattice points.

This is no longer true in the inhomogeneous case where the computation of the rational solutions is an extra task for Normaliz. This extra step is inevitable for the primal algorithm, but not for the dual algorithm. In general, the computation of the rational solutions is much faster than the computation of the integral solutions, but this by no means always the case.

Therefore we have decoupled the two computations if the dual algorithm is applied to inhomogeneous systems or to the computation of degree 1 points in the homogeneous case. The combinations

DualMode HilbertBasis, -dN

DualMode Deg1Elements, -d1

DualMode ModuleGenerators

DualMode LatticePoints

do not imply the computation goal `SupportHyperplanes` (and not even `Sublattice`) which would trigger the computation of the rational solutions (geometrically: the vertices of the polyhedron). If you want to compute them, you must add one of

SupportHyperplanes, -s

ExtremeRays

VerticesOfPolyhedron

The last choice is only possible in the inhomogeneous case. Another possibility in the inhomogeneous case is is to use `DualMode` without a restriction.

If `Projection` or `ProjectionFloat` is used for parallelotopes defined by inequalities, then Normaliz does not compute the vertices, unless asked for by one of the three computation goals just mentioned or the extreme rays are needed for some other computation. The same holds if the volume of a parallelotope is computed.

6. Running Normaliz

The standard form for calling Normaliz is

```
normaliz [options] <project>
```

where `<project>` is the name of the project, and the corresponding input file is `<project>.in`. Note that `normaliz` may require to be prefixed by a path name, and the same applies to `<project>`. A typical example on a Linux or Mac system:

```
./normaliz --verbose -x=5 example/big
```

that for MS Windows must be converted to

```
.\normaliz --verbose -x=5 example\big
```

Normaliz uses the standard conventions for calls from the command line:

(1) the order of the arguments on the command line is arbitrary.
(2) Single letter options are prefixed by the character `-` and can be grouped into one string.
(3) Verbatim options are prefixed by the characters `--`.

The options for computation goals and algorithmic variants have been described in Section 5. In this section the remaining options for the control of execution and output are discussed, together with some basic rules for the use of the options.

6.1. Basic rules

The options for computation goals and algorithms variants have been explained in Section 5. The options that control the execution and the amount of output will be explained in the following. Basic rules for the use of options:

1. If no `<project>` is given, the program will terminate.
2. The option `-x` differs from the other ones: `<T>` in `-x=<T>` represents a positive number assigned to `-x`; see Section 6.3.
3. Similarly the option `--OutputDir=<outdir>` sets the output directory; see 6.6.
4. Normaliz will look for `<project>.in` as input file.

   If you inadvertently typed `rafa2416.in` as the project name, then Normaliz will first look for `rafa2416.in.in` as the input file. If this file doesn't exist, `rafa2416.in` will be loaded.
5. The options can be given in arbitrary order. All options, including those in the input file, are accumulated, and syntactically there is no mutual exclusion. However, some options may block others during the computation. For example, `KeepOrder` blocks `BottomDecomposition`.
6. If Normaliz cannot perform a computation explicitly asked for by the user, it will terminate. Typically this happens if no grading is given although it is necessary.
7. In the options include `DefaultMode`, Normaliz does not complain about missing data (anymore). It will simply omit those computations that are impossible.
8. If a certain type of computation is not asked for explicitly, but can painlessly be produced as a side effect, Normaliz will compute it. For example, as soon as a grading is present and the Hilbert basis is computed, the degree 1 elements of the Hilbert basis are selected from it.

6.2. Info about Normaliz

`--help, -?` displays a help screen listing the Normaliz options.
`--version` displays information about the Normaliz executable.

6.3. Control of execution

The options that control the execution are:

`--verbose, -c` activates the verbose ("console") behavior of Normaliz in which Normaliz writes additional information about its current activities to the standard output.

`-x=<T>` Here `<T>` stands for a positive integer limiting the number of threads that Normaliz is allowed access on your system. The default value is 8. (Your operating system may set a lower limit).

`-x=0` switches off the limit set by Normaliz.

If you want to run Normaliz in a strictly serial mode, choose `-x=1`.

`parallel_threads <T>` can be used in the input file instead.

The number of threads can also be controlled by the environment variable `OMP_NUM_THREADS`. See Section 10.1 for further discussion.

If there ar5e many polynomials in the input it can be difficult to find an error in them. As a help ion such cases one can say

`list_polynomials`

The last polynomial listed has caused the error.

6.4. Interruption

During a computation `normaliz` can be interrupted by pressing Ctrl-C on the keyboard. If this happens, Normaliz will stop the current computation and write the already computed data to the output file(s). By

`NoOutputOnInterrupt`

the output of data can be blocked. Can be set in the input file or on the command line as a long option.

If Ctrl-C is pressed during the output phase, Normaliz is stopped immediately.

6.5. Time bound

In order to set a time bound for the execution of Normaliz one creates a file

`normaliz.time`

in the working directory. It contains a single floating number that bounds the wall clock time of Normaliz. At present it is only implemented in the project-and-lift algorithm for lattice points.

6.6. Control of output files

In the default setting Normaliz writes only the output file `<project>.out` (and the files produced by `Triangulation`, `StanleyDec` and `FaceLattice`). The amount of output files can be increased as follows:

`--files, -f` Normaliz writes the additional output files with suffixes `gen`, `cst`, and `inv`, provided the data of these files have been computed.

`--all-files, -a` includes `Files`, Normaliz writes all available output files (except `typ` and those that are automatically written by computation goals).

`--<suffix>` chooses the output file with suffix `<suffix>`.

For the list of potential output files, their suffixes and their interpretation see Section 9. There are several options `--<suffix>`.

If the computation goal `IntegerHull` is set, Normaliz computes a second cone and lattice. The output is contained in `<project>.IntHull.out`. The options for the output of `<project>` are applied to `<project>.IntHull` as well. There is no way to control the output of the two computations individually.

Similarly, if symmetrization has been used, Normaliz writes the file `<project>.symm.out`. It contains the data of the symmetrized cone.

Sometimes one wants the output to be written to another directory. The output directory can be set by

`--OutputDir=<outdir>` . The path `<outdir>` is an absolute path or a path relative to the current directory (which is not necessarily the directory of `<project>.in`.)

Note that all output files will be written to the chosen directory. It must be created before Normaliz is started.

Extreme rays and vertices may have very long integer coordinates. One can suppress their output by

`NoExtRaysOutput`

For similar reasons one may want to suppress the output of support hyperplanes, namely by

`NoSuppHypsOutput`

Similarly,

`NoHilbertBasisOutput`

supprsesses thze output of Hilbert bases and latticec points. An even more drastic option is

`NoMatricesOutput`

It suppresses all output after the "preamble". It is useful in testing large examples where the numbers of extreme rays, lattice points etc. are usually a good criterion for correctness.

`NoExtRaysOutput`, `NoSuppHypsOutput`and `NoMatricesOutput` are not cone properties.

`BinomialsPacked` chooses a packed format for files containing binomials. See Section 7.25.

6.7. Ignoring the options in the input file

Since Normaliz accumulates options, one cannot get rid of settings in the input file by command line options unless one uses

`--ignore, -i` This option disables all options in the input file.

7. Advanced topics

7.1. Computations with a polytope

In many cases the starting point of a computation is a polytope, i.e., a bounded polyhedron – and not a cone or monoid. Normaliz offers two types of input for polytopes that allow almost the same computations, namely

(1) *homogeneous* input type for which the polytope is the intersection of a cone with a hyperplane defined by the grading (automatically bounded): $P = \{x \in C : \deg x = 1\}$.
(2) *inhomogeneous* input defining a polytope (and not an unbounded polyhedron).

A problem that can arise with homogeneous input is the appearance of a grading enumerator $g > 1$. In this case the polytope P defined by the input grading is replaced by gP. This may be undesirable and can therefore be blocked by `NoGradingDenom`. Note: a grading denominator $g > 1$ can only appear if the affine space spanned by the polytope does not contain a lattice point. This is a rare situation, but nevertheless you may want to safeguard against it.

Computation goals whose names have a "polytopal touch" (as opposed to "algebraic touch") set `NoGradingDenom` automatically. These computation goals are also to be used with inhomogeneous input; see the following table. The homogeneous input type `polytope` sets `NoGradingDenom` as well.

In the following table L is the lattice of reference defined by the input data.

desired data	inhom input or hom input blocking grading denominator	hom input allowing grading denominator
lattice points	`LatticePoints`	`Deg1Elements`
number of lattice points	`NumberLatticePoints`	—
convex hull of lattice points	`IntegerHull`	—
generating function of $k \mapsto \#(kP \cap L)$	`EhrhartSeries`	`HilbertSeries`
volume or multiplicity	`Volume`	`Multiplicity`
integral	`Integral`	—

Note that `HilbertSeries` and `Multiplicity` make also sense with inhomogeneous input, but they refer to a different counting function, namely

$$k \mapsto \#(x \in P \cap L, \deg x = k).$$

Even if P is a polytope, this function has applications; see Section 7.10.2. Note that inhomogeneous input sets `NoGradingDenom`.

7.1.1. Lattice normalized and Euclidean volume

As just mentioned, for polytopes defined by homogeneous input Normaliz has two computation goals, `Multiplicity`, `-v`, and `Volume, -V`, that are almost identical: `Volume` = `Multiplicity` + `NoGradingDenom`. Both compute the lattice normalized volume; moreover, `Volume` additionally computes the Euclidean volume and can also be used with inhomogeneous input, for which `Multiplicity` has a different meaning. (For the algebraic origin of `Multiplicity` see Appendix A.6.)

In the following we want to clarify the notion of *lattice normalized volume*.

(1) Let $P \subset \mathbb{R}^d$ be a polytope of dimension r and let A be the affine subspace spanned by P. Then the Euclidean volume $\mathrm{vol}_{\mathrm{eucl}}(P)$ of P is computed with respect to the r-dimensional Lebesgue measure in which an r-dimensional cube in A of edge length 1 has measure 1.

(2) For the lattice normalized volume we need a lattice L of reference. We assume that $\mathrm{aff}(P) \subset \mathrm{aff}(L)$. (It would be enough to have this inclusion after a parallel translation of $\mathrm{aff}(P)$.) Choosing the origin in L, one can assume that $\mathrm{aff}(L)$ is a vector subspace of $\mathbb{R}^d$ so that we can identify it with $\mathbb{R}^d$ after changing d if necessary. After a coordinate transformation we can further assume that $L = \mathbb{Z}^d$ (in general this is not an orthogonal change of coordinates!). To continue we need that $\mathrm{aff}(P)$ is a rational subspace. There exists $k \in \mathbb{N}$ such that $k\,\mathrm{aff}(P)$ contains a lattice simplex. The lattice normalized volume vol_L of kP is then given by the Lebesgue measure on $k\,\mathrm{aff}(P)$ in which the smallest possible lattice simplex in $k\,\mathrm{aff}(P)$ has volume 1. Finally we set $\mathrm{vol}_L(P) = \mathrm{vol}_L(kP)/k^r$ where $r = \dim(P)$.

If P is a full-dimensional polytope in $\mathbb{R}^d$ and $L = \mathbb{Z}^d$, then $\mathrm{vol}_L(P) = d!\,\mathrm{vol}_{\mathrm{eucl}}(P)$, but in general the correction factor is $cr!$ with c depending on $\mathrm{aff}(P)$: the line segment in $\mathbb{R}^2$ connecting $(1,0)$ and $(0,1)$ has euclidean length $\sqrt{2}$, but lattice normalized volume 1. As this simple example shows, c can be irrational.

7.1.2. Developer's choice: homogeneous input

Our recommendation: if you have the choice between homogeneous and inhomogeneous input, go homogeneous (with `NoGradingDenom` if necessary). You do not lose any computation goal and can only gain efficiency.

7.2. Lattice points in polytopes once more

Normaliz has three main algorithms for the computation of lattice points of which two have two variants each:

(1) the project-and-lift algorithm (`Projection, -j`),
(2) its variant using floating point arithmetic (`ProjectionFloat, -J`),
(3) the triangulation based Normaliz primal algorithm specialized to lattice points (`PrimalMode, -P`),
(4) its variant using approximation of rational polytopes (`Approximate, -r`),

(5) the dual algorithm specialized to lattice points (`DualMode, -d`).

The options `Projection`, `ProjectionFloat` and `Approximate` do not imply a computation goal. Since `PrimalMode` can also be used for the computation of Hilbert series and Hilbert bases, one must add the computation goal to it. In the homogeneous case one must add the computation goal also to `DualMode`.

Remark. The triangulation based primal algorithm and the dual algorithm do not depend on the embedding of the computed objects into the ambient space since they use only data that are invariant under coordinate transformations. This is not true for project-and-lift and the approximation discussed below. Often `Projection` and `ProjectionFloat` (and in certain cases also `PrimalMode`) profit significantly from LLL reduced coordinates (since version 3.4.1). We discuss this feature in Section 7.2.3.

We recommend the reader to experiment with the following input files:

- `5x5.in`
- `6x6.in`
- `max_polytope_cand.in`
- `hickerson-18.in`
- `knapsack_11_60.in`
- `ChF_2_64.in`
- `ChF_8_1024.in`
- `VdM_16_1048576.in` (may take some time)
- `pedro2.in`
- `pet.in`
- `baby.in`

In certain cases you must use `-i` on the command line to override the options in the input file if you want to try other options.

`max_polytope_cand.in` came up in connection with the paper "Quantum jumps of normal polytopes" by W. Bruns, J. Gubeladze and M. Michałek, Discrete Comput. Geom. 56 (2016), no. 1, 181–215. `hickerson-18.in` is taken from the LattE distribution [5]. `pedro2.in` was suggested by P. Garcia-Sanchez.

The files `ChF*.in` and `VdM*.in` are taken from the paper "On the orthogonality of the Chebyshev-Frolov lattice and applications" by Ch. Kacwin, J. Oettershagen and T. Ullrich (Monatsh. Math. 184 (2017), 425–441). The file `VdM_16_1048576.in` is based on the linear map given directly by the Vandermonde matrix. A major point of the paper is a coordinate transformation that simplifies computations significantly, and the files `ChF*.in` are based on it.

The files `pet.in` and `baby.in` have been created in connection with the work on fusion rings. See [3].

7.2.1. Project-and-lift

We have explained the project-and-lift algorithm in Section 2.13. This algorithm is very robust arithmetically since it needs not compute determinants or solve systems of linear equations.

Moreover, the project-and-lift algorithm itself does not use the vertices of the polytope explicitly and only computes lattice points in P and its successive projections. Therefore it is rather insensitive against rational vertices with large denominators. (To get started it must usually compute the vertices of the input polytope; an exception are parallelotopes, as mentioned in Section 2.13 and discussed below.) Project-and-lift is not done by default if the number of support hyperplanes exceeds that of the number of extreme rays by a factor > 100.

The option for project-and-lift is

`Projection, -j`

There are two complications that may slow it down unexpectedly: (i) the projections may have large numbers of support hyperplanes, as seen in the example `VdM_16_1048576.in` (it uses floating point arithmetic in the lifting part):

```
Projection
embdim 17 inequalities 32
embdim 16 inequalities 240
...
embdim 11 inequalities 22880
embdim 10 inequalities 25740
embdim 9 inequalities 22880
...
embdim 3 inequalities 32
embdim 2 inequalities 2
```

(ii) The projections may have many lattice points that cannot be lifted to the top. As an example we look at the terminal output of `pedro2.in`:

```
embdim 2 LatticePoints 40
embdim 3 LatticePoints 575
embdim 4 LatticePoints 6698
embdim 5 LatticePoints 6698
embdim 6 LatticePoints 2
```

Despite of these potential problems, `Projection` is the default choice of Normaliz for the computation of lattice points (if not combined with Hilbert series or Hilbert basis). If you do not want to use it, you must either choose another method explicitly or switch it off by `NoProjection`. Especially for lattice polytopes with few extreme rays, but many support hyperplanes the triangulation base algorithm is often the better choice.

Parallelotopes. Lattice points in parallelotopes that are defined by inequalities, like those in the input files `VdM*.in`, can be computed without any knowledge of the vertices. In fact, for them it is favorable to present a face F by the list of facets whose intersection F is (and not by the list of the $2^{\dim F}$ vertices of F!). Parallelotopes are not only simple polytopes. It is important that two faces do not intersect if and only if they are contained in parallel facets, and this is easy to control. Normaliz recognizes parallelotopes by itself, and suppresses the computation of the vertices unless asked to compute them.

7.2.2. Project-and-lift with floating point arithmetic

Especially the input of floating point numbers often forces Normaliz into GMP arithmetic. Since GMP arithmetic is slow (compared to arithmetic with machine integers or floating point numbers), Normaliz has a floating point variant of the project-and-lift algorithm. (Such an algorithm makes no sense for Hilbert bases or Hilbert series.) It behaves very well, even in computations for lower dimensional polytopes. We have not found a single deviation from the results with GMP arithmetic in our examples. Nevertheless, the projection phase is done in the in integer arithmetic, and only the lifting uses floating point.

The option for the floating point variant of project-and-lift is

`ProjectionFloat, -J`

If you want a clear demonstration of the difference between `Projection` and `ProjectionFloat`, try `VdM_16_1048576.in`.

The use of `ProjectionFloat` or any other algorithmic variant is independent of the input type. The coordinates of the lattice points computed by `ProjectionFloat` are assumed to be at most 64 bits wide, independently of the surrounding integer type. If this condition should not be satisfied in your application, you must use `Projection` instead.

7.2.3. LLL reduced coordinates and relaxation

The project-and-lift algorithm depends very much on the embedding of the polytope in the ambient space. We use LLL reduction to find coordinate transformations of the ambient space in which the vertices of the polytope have small coordinates so that the successive projections have few lattice points. Roughly speaking, LLL reduced coordinates are computed as follows. We form a matrix A whose *rows* are the vertices or the support hyperplanes of the polytope, depending on the situation. Suppose A has d columns; A need not have integral entries, but it must have rank d. Then we apply LLL reduction to the lattice generated by the *columns* of A. This amounts to finding a matrix $U \in \mathrm{GL}(d,\mathbb{Z})$ such that the columns of AU are short vectors (in the Euclidean norm). The matrix U and its inverse then define the coordinate transformations forth and back.

Often LLL reduction has a stunning effect. We have a look at the terminal output of `pedro2.in` run with `-P`. The left column shows the present version, the right one is produced by Normaliz 3.4.0:

```
embdim 2 LatticePoints 2              embdim 2 LatticePoints 40
embdim 3 LatticePoints 2              embdim 3 LatticePoints 672
embdim 4 LatticePoints 2              embdim 4 LatticePoints 6698
embdim 5 LatticePoints 3              embdim 5 LatticePoints 82616047
embdim 6 LatticePoints 2              embdim 6 LatticePoints 2
```

We have no example for which LLL increases the computation time. Though its application not seem to be a real disadvantage, it can be switched off for `Projection` and `ProjectionFloat` by

NoLLL

Without LLL certain computations are simply impossible – just try `VdM_16_1048576` with `NoLLL`. (This option is used internally to avoid a repetition of LLL computations.)

We use the original LLL original algorithm with the factor 0.9.

Another aspect of the implementation that must be mentioned is the relaxation of inequalities: for the intermediate lifting of lattice points Normaliz uses at most 1000 (carefully chosen) inequalities. Some additional intermediate lattice points are acceptable if the evaluation of inequalities is reduced by a substantial factor. On the left we see `VdM_16_1048576` with relaxation, on the right without:

```
...                                        ...
embdim 6 LatticePoints 2653                 embdim 6 LatticePoints 2297
...                                        ...
embdim 10 LatticePoints 431039              embdim 10 LatticePoints 128385
embdim 11 LatticePoints 1031859             embdim 11 LatticePoints 277859
embdim 12 LatticePoints 2016708             embdim 12 LatticePoints 511507
embdim 13 LatticePoints 2307669             embdim 13 LatticePoints 806301
...                                        ...
```

No surprise that relaxation increases the number of intermediate lattice points, but it reduces the computation time by about a factor 2.

It is of course not impossible that relaxation exhausts RAM or extends the computation time. Therefore one can switch it off by

NoRelax

7.2.4. Positive systems, coarse project-and-lift and patching

We call a system of linear equations and inequalities *positive* if the inequalities contain the sign inequalities $x_i \geq 0$ for all coordinates and inequalities (including inequalities implied by equations) for all i of type

$$\xi_1 x_1 + \cdots + \xi_{i-1} x_{i-1} + \xi_i x_i + \xi_{i+1} x_{i+1} + \cdots + \xi_d x_d \leq \eta \tag{1}$$

where $\xi_j \geq 0$ for all j and $\xi_i > 0$. This allows one to find an a priori upper bound for every coordinate x_i. For homogeneous input this must be modified a little: the grading must be a coordinate function, this coordinate is omitted on the left hand side, and η is the coefficient of the coordinate that represents the grading.

For positive systems it is a priori clear that the polyhedron defined by the linear constraints is a polytope, and there is no need to check that by computing the vertices. Moreover, we can base a relaxed version of project-and-lift on the nonnegativity inequalities and the upper bounds 1. This variant often has a tremendous advantage: run `pet.in` as it is or with the option

NoCoarseProjection

The computation times are < 1 sec and (most likely) $\sim \infty$.

Under certain conditions coarse project-and-lift is modified. Instead of proceeding from one coordinate to the next, "local systems" are used to extend the range of coordinates by 'patching". It is a particularly fast method if it applies. Our standard example `pet.in` is automatically run in this mode, unless you forbid it by

`NoPatching`

To see the effect, try `baby.in` as it is and with `NoPatching`.

The patching algorithm and the use of polynomial equations was developed for computations of fusion rings reported in [3] and extensions.

Note that the patching algorithm allows distributed computation (se Section F.2)

7.2.5. Polynomial constraints for lattice points

The lattice points in polytopes computed by Normaliz can be further constrained by polynomial equations an d inequalities (see Section 4.9). For all algorithms different from project-and-lift the lattice point must be computed completely before the polynomial constraints can be used. It is a significant advantage of project-and-lift and its variants that every polynomial constraint can be applied as soon as all coordinates appearing in it with a nonzero coefficient have been collected. This helps enormously to stop the extension of coordinates.

To avoid ambiguities, only computation goals for lattice points in polytopes and convex hull data can be computed in the presence of polynomial constraints.

`pet.in`, `baby.in` and `poly_equ.in` are examples with polynomial constraints.

7.2.6. The patching order

If polynmomial equations are present, then Normaliz tries to use them with as few coordinates as possible and schedules the patching algorithm accordingly. However, this can have a negative effect if it makes 'bad" patches used too early. Also congruences, whether explicitly given in the input or derived by Normaliz from linear equations, can influence the insertion order of the patches.

There are some options that can modify the default behavior:

`UseWeightsPatching` gives a "weight" to the coordinates so that "'bad" patches are biased negatively.

`LinearOrderPatches` disables the scheduling by polynomial equations completely. It can be modified by `UseWeightsPatching`.

Moreover it is possible to use another type of constraints:

`CongOrderPatches` bases the scheduling on congruences derived from the linear equations in the input. (Can also be modified by `UseWeightsPatching`).

Finally it is possible to define the patching order by hand. To this end one uses an extra input file

`<project>.order.patches`

It contains a sequence of numbers: the first is the number <n>of patches that follow, and then the first <n> patches are inserted in this order. The remaining ones are inserted linearly.

In very rare cases it could be useful to use

MinimizePolyEquations

The minimization can take very long. It is only applicable if the polynomials have degree ≤ 2.

These options must be used with care and may require experimentation. For a first encounter you can try them on the three examples mentioned above.

7.2.7. The triangulation based primal algorithm

With this algorithm, Normaliz computes a partial triangulation as it does for the computation of Hilbert bases (in primal mode) for the cone over the polytope. Then it computes the lattice points in each of the subpolytopes defined by the simplicial subcones in the triangulation. The difference to the Hilbert basis calculation is that all points that do not lie in our polytope P can be discarded right away and that no reduction is necessary.

The complications that can arise are (i) a large triangulation or (ii) large determinants of the simplicial cones. Normaliz tries to keep the triangulations small by restricting itself to a partial triangulation, but often there is nothing one can do. Normaliz deals with large determinants by applying project-and-lift to the simplicial subcones with large determinants. We can see this by looking at the terminal output of `max_polytope_cand.in`, computed with `-cP -x=1`:

```
...
evaluating 49 simplices
|||||||||||||||||||||||||||||||||||||||||||||||||||
49 simplices, 819 deg1 vectors accumulated.
47 large simplices stored
Evaluating 47 large simplices
Large simplex 1 / 47
************************************************************
starting primal algorithm (only support hyperplanes) ...
Generators sorted lexicographically
Start simplex 1 2 3 4 5
Pointed since graded
Select extreme rays via comparison ... done.
------------------------------------------------------------
transforming data... done.
Computing lattice points by project-and-lift
Projection
embdim 6 inequalities 7
...
embdim 2 inequalities 2
Lifting
embdim 2 Deg1Elements 9
...
```

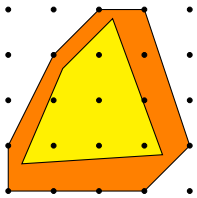
```
embdim 6 Deg1Elements 5286
Project-and-lift complete
...
```

After finishing the 49 "small" simplicial cones, Normaliz takes on the 47 "large" simplicial cones, and does them by project-and-lift (including LLL). Therefore one can say that Normaliz takes a hybrid approach to lattice points in primal mode.

An inherent weakness of the triangulation based algorithm is that its efficiency drops with $d!$ where d is the dimension because the proportion of lattice points in P of all points generated by the algorithm must be expected to be $1/d!$ (as long as small simplicial cones are evaluated). To some extent this is compensated by the extremely fast generation of the candidates.

7.2.8. Lattice points by approximation

Large determinants come up easily for rational polytopes P whose vertices have large denominators. In previous versions, Normaliz fought against large determinants coming from rational vertices by finding an integral polytope Q containing P, computing the lattice points in Q and then sieving out those that are in $Q \setminus P$:



This approach is still possible. It is requested by the option

`Approximate, -r`

This is often a good choice, especially in low dimension.

It is not advisable to use approximation for polytopes with a large number of vertices since it must be expected that the approximation multiplies the number of vertices by $\dim P + 1$ so that it may become difficult to compute the triangulation.

Approximation requires that the grading denominator is equal to 1. If this condition is not satisfied, primal mode is used.

7.2.9. Lattice points by the dual algorithm

Often the dual algorithm is extremely fast. But it can also degenerate terribly. It is very fast for `6x6.in` run with `-d1`. The primal algorithm or approximation fail miserably. (`-1`, the default choice project-and-lift, is also quite good. The difference is that `-d1` does not compute the vertices that in this case are necessary for the preparation of project-and-lift.)

On the other hand, the dual algorithm is hopeless already for the 2-dimensional parallelotope `ChF_2_64.in`. Try it. It is clear that complicated arithmetic is dangerous for the dual algorithm.

(The dual algorithm successively computes the lattice points correctly for all intermediate polyhedra, defined as intersections of the half spaces that have been processed so far. The intermediate polyhedra can be much more difficult than the final polytope, as in this case.)

In certain cases (see Section 7.5) Normaliz will try the dual algorithm if you forbid project-and-lift by `NoProjection`.

7.2.10. Counting lattice points

In some applications one is not interested in the lattice points, but only in their number. In this case you can set the computation goal

`NumberLatticePoints`

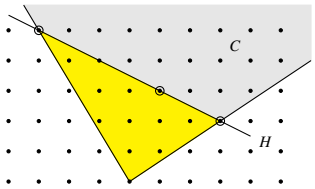
The main advantage is that it does not store the lattice points and therefore cannot fail because of lack of memory if their number becomes very large. In the inhomogeneous case `NumberLatticePoints` can be combined with `HilbertSeries`. Then the lattice points are counted by degree. See Section 7.10.2 for an application.

`NumberLatticePoints` uses project-and-lift (with floating point if `ProjectionFloat` is set). Therefore don't block it. If the number of lattice points is so large that memory becomes a problem, then the primal and the dual algorithm will most likely not be able to compute them.

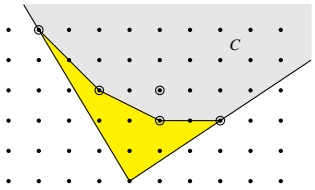
7.3. The bottom decomposition

The triangulation size and the determinant sum of the triangulation are critical size parameters in Normaliz computations. Normaliz always tries to order the generators in such a way that the determinant sum is close to the minimum, and on the whole this works out well. The use of the bottom decomposition by `BottomDecomposition, -b` enables Normaliz to compute a triangulation with the optimal determinant sum for the given set of generators, as we will explain in the following.

The determinant sum is independent of the order of the generators of the cone C if they lie in a hyperplane H. Then the determinant sum is exactly the normalized volume of the polytope spanned by 0 and $C \cap H$. The triangulation itself depends on the order, but the determinant sum is constant.



This observation helps to find a triangulation with minimal determinant sum in the general case. We look at the *bottom* (the union of the compact faces) of the polyhedron generated by $x_1, \ldots, x_n$ as vertices and C as recession cone, and take the volume underneath the bottom:



With the option `BottomDecomposition`, `-b`, Normaliz computes a triangulation that respects the bottom facets. This yields the optimal determinant sum for the given generators. If one can compute the Hilbert basis by the dual algorithm, it can be used as input, and then one obtains the absolute bottom of the cone, namely the compact facets of the convex hull of all nonzero lattice points.

Normaliz does not always use the bottom decomposition by default since its computation requires some time and administrative overhead. However, as soon as the input "profile" is considered to be "rough" it is invoked. The measure of roughness is the ratio between the maximum degree (or L_1 norm without a grading) and the minimum. A ratio ≥ 10 activates the bottom decomposition.

If you have the impression that the bottom decomposition slows down your computation, you can suppress it by `NoBottomDec`, `-o`.

The bottom decomposition is part of the subdivision of large simplicial cones discussed in the next section.

The example `strictBorda.in` belongs to social choice theory like `Condorcet.in` (see Section 2.10), `PluralityVsCutoff.in` and `CondEffPlur.in`. The last two profit enormously from symmetrization (see Section 7.8), but `strictBorda.in` does not. Therefore we must compute the Hilbert series for a monoid in dimension 24 whose cone has 6363 extreme rays. It demonstrates the substantial gain that can be reached by bottom decomposition. Since the roughness is large enough, Normaliz chooses bottom decomposition automatically, unless we block it.

algorithm	triangulation size	determinant sum
bottom decomposition	30,399,162,846	75,933,588,203
standard order of extreme rays, `-o`	119,787,935,829	401,249,361,966

7.4. Subdivision of large simplicial cones

Especially in computations with rational polytopes one encounters very large determinants that can keep the Normaliz primal algorithm from terminating in reasonable time. As an example we take `hickerson-18.in` from the LattE distribution [5]. It is simplicial and the complexity is totally determined by the large determinant $\approx 4.17 \times 10^{14}$ (computed with `-v`).

If we are just interested in the degree 1 points, Normaliz uses the project-and-lift method of Section 7.2.1 and finds 44 degree 1 points in the blink of an eye. If we use these points together with the extreme rays of the simplicial cone, then the determinant sum decreases to $\approx 1.3 \times 10^{12}$, and the computation of the Hilbert basis and the Hilbert series is in reach. But it is better to pursue the idea of subdividing large simplicial cones systematically. Normaliz

uses its own algorithm for finding optimal subdivision points, based on project-and-lift(and LLL reduced coordinates).

Normaliz tries to subdivide a simplicial cone if it has determinant $\geq 10^8$ or 10^7 if the Hilbert basis is computed. Both methods are used recursively via stellar subdivision until simplicial cones with determinant $< 10^6$ have been reached or no further improvement is possible. All subdivision points are then collected, and the start simplicial cone is subdivided with bottom decomposition, which in general leads to substantial further improvement.

The following table contains some performance data for subdivisions based on the Normaliz method (default mode, parallelization with 8 threads).

	`hickerson-16`	`hickerson-18`	`knapsack_11_60`
simplex volume	9.83×10^7	4.17×10^{14}	2.8×10^{14}
stellar determinant sum	3.93×10^6	9.07×10^8	1.15×10^8
volume under bottom	8.10×10^5	3.86×10^7	2.02×10^7
volume used	3.93×10^6	6.56×10^7	2.61×10^7
runtime without subdivision	2.8 s	$>$ 12 d	$>$ 8 d
runtime with subdivision	0.4 s	24 s	5.1 s

A good nonsimplicial example showing the subdivision at work is `hickerson-18plus1.in` with option `-q`.

Note: After subdivision the decomposition of the cone may no longer be a triangulation in the strict sense, but a decomposition that we call a *nested triangulation*; see Section 7.14.1. If the creation of a nested triangulation must be blocked, one uses the option `NoSubdivision`. Inevitably it blocks the subdivision of large simplicial cones.

Remark The bounds mentioned above work well up to dimension ≈ 10. For a fixed determinant, the probability for finding a subdivision point decreases rapidly.

7.5. Primal vs. dual – division of labor

As already mentioned several times, Normaliz has two main algorithms for the computation of Hilbert bases, the primal algorithm and the dual algorithm. It is in general very hard to decide beforehand which of the two is better for a specific example. Nevertheless Normaliz tries to guess, unless `PrimalMode`, `-P` or `DualMode`, `-d` is explicitly chosen by the user. In first approximation one can say that the dual algorithm is chosen if the computation is based on constraints and the number of inequalities is neither too small nor too large. Normaliz chooses the dual algorithm if at the start of the Hilbert basis computation the cone is defined by s inequalities such that

$$r + \frac{50}{r} \leq s \leq 2e$$

where r is the rank of the monoid to be computed and e is the dimension of the space in which the data are embedded. These conditions are typically fulfilled for diophantine systems

of equations whose nonnegative solutions are asked for. In the case of very few or many hyperplanes Normaliz prefers the primal algorithm. While this combinatorial condition is the only criterion for Normaliz, it depends also on the arithmetic of the example what algorithm is better. At present Normaliz makes no attempt to measure it in some way.

When both Hilbert basis and Hilbert series are to be computed, the best solution can be the combination of both algorithms. We recommend `2equations.in` as a demonstration example which combines the algorithmic variant `DualMode` and the computation goal `HilbertSeries`:

```
amb_space 9
equations 2
1 6 -7 -18 25 -36 6 8 -9
7 -13 15 6 -9 -8 11 12 -2
total_degree
DualMode
HilbertSeries
```

As you will see, the subdivision of large simplicial cones is very useful for such computations.

Compare `2equations.in` and `2equations_default.in` for an impression on the relation between the algorithms.

7.6. Various volume versions

Normaliz offers various algorithms for the volume of a polytope. They all compute the lattice normalized volume, and additionally convert it to the Euclidean volume. There are 3 basic algorithms:

(1) the *primal* volume algorithm: Normaliz computes a lexicograhic triangulation, and finds the volume as the sum of the volumes of the simplices in the triangulation;
(2) volume by *descent in the face lattice*: there is a reverse lexicographic triangulation in the background, but it is not computed explicitly;
(3) volume by *signed decomposition*: Normaliz computes a triangulation of the dual cone and converts it into a signed decomposition of the polytope.

For algebraic polytopes only (1) is implemented at present. But (3) could be extended to them, whereas (2) is not suitable.

By rule of thumb one can say that the best choice is

(1) if the polytope has few vertices, but potentially many facets;
(2) if the number of vertices and the number of facets are of the same order of magnitude;
(3) if there are *very* few facets and many vertices.

Normaliz tries to choose the optimal algorithm by default. We will illustrate this recommendation by examples below.

There are variants:

(a) exploitation of isomorphism types of faces in the descent algorithm;

(b) symmetrization (explained in Section 7.8).

In volume computations that are not part of a Hilbert series computation Normaliz checks the default conditions of the algorithms in the order

$$\text{signed decomposition} \to \text{descent} \to \text{symmetrization}$$

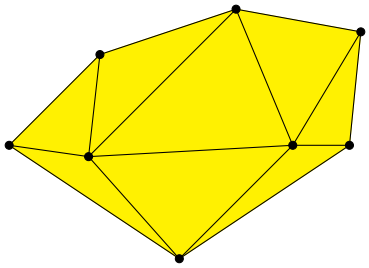
If the default conditions are not satisfied for any of them, the primal triangulation algorithm is used. These decisions must often be made on the basis of partial information. For example, the really critical parameter for descent is the number of non-simplicial facets. Therefore it can be useful to ask for a certain variant explicitly or to exclude the others. The exploitation of isomorphism types must always be asked for explicitly by the user.

Normaliz recognizes parallelotopes and applies an extremely fast method for their volumes.

We compare computation times for some significant examples in Section 7.6.6. Normaliz always computes multiplicities of monoids, but we simply talk of volumes in this section.

7.6.1. The primal volume algorithm

It has been used many times in the examples of this manual, and mathematically there is nothing to say: if a polytope P is decomposed into simplices with non-overlapping interiors, then its volume is the sum of the volumes of the simplices forming the decomposition.

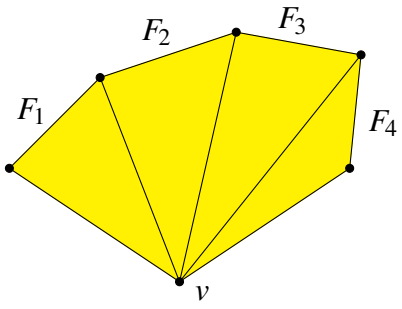


7.6.2. Volume by descent in the face lattice

The idea is to exploit the formula

$$\text{mult}(P) = \sum_i \text{ht}_{F_i}(v)\,\text{mult}(F_i)/\deg(v).$$

recursively where v is a vertex of the polytope P with as few opposite facets F_i as possible, and $\text{ht}_{F_i}(v)$ is the lattice height of v over F_i. The formula is illustrated by the figure:



The recursive application results in building a subset $\mathscr{F}$ of the face lattice so that for each face $F \in \mathscr{F}$ to which the formula is applied all facets of F that are opposite to the selected vertex are contained in $\mathscr{F}$. However, if a face is simplicial, its multiplicity is computed by the standard determinant formula. The algorithm is implemented in such a way that all data are collected in the descent and no backtracking is necessary. The RAM usage is essentially determined by the two largest layers. For a detailed discussion we refer the reader to [14]. However, meanwhile many examples disussed in [14] can be computed much faster by signed decomposition, which is discussed below.

You can force this algorithm is by

`Descent, -F`

and block it by

`NoDescent`

Note that `Descent` does *not* imply `Multiplicity` or `Volume`. (We cannot exclude that in the future descent is used also for other computations.)

As an example we have a look at `lo6` and show part of its terminal output. We look at this example again when we discuss the variant that exploits isomorphism types.

```
Command line: -c ../example/lo6 -iv --Descent
Compute: Multiplicity Descent
...
Descent from dim 15, size 854
...............................................
Descent from dim 14, size 7859
...............................................
Descent from dim 13, size 37587
```

7.6.3. Descent exploiting isomorphism classes of faces

The descent algorithm computes a subset of the face lattice. We can reduce the size of this "descent system" if we identify faces in it that are isomorphic. In order to have a beneficial effect on computation time, the reduction must be substantial since the computation of isomorphism types is relatively slow. The polytope should at least have a large automorphism

group, but this alone is no guarantee for an acceleration. The exploitation of isomorphism types is asked for by

`Descent ExploitIsosMult`

It is better to ask for `Descent` explicitly, but `ExploitIsosMult` will be recognized if `Descent` is chosen by default.

This variant is only available if Normaliz has been built with nauty and hash-library. The latter is used to store the normal forms that take much memory by their SHA256 hash values. But you can insist on strict type checking by

`StrictIsoTypes`

We show a little bit of the terminal output for `lo6` for which this variant is particularly fast:

```
Command line: -c ../example/lo6 -iv --Descent --ExploitIsosMult
Compute: Multiplicity Descent ExploitIsosMult
...
Descent from dim 15, size 2
Descent from dim 14, size 232
Collecting isomorphism classes
...............................................
Iso types 5
Descent from dim 13, size 224
Collecting isomorphism classes
...
```

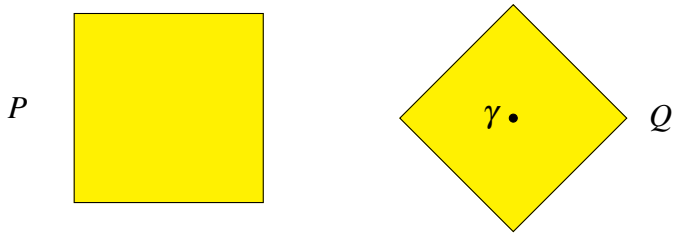
Compared to `Descent` without exploitation of isomorphism classes the reduction is indeed substantial, and is reflected drastically by the computation times.

Using isomorphism types opens descent for polytopes with many facets, but few isomorphism classes of them.

7.6.4. Volume by signed decomposition

This algorithm uses that a "generic" triangulation of the dual cone induces a "signed decomposition" of the primal polytope. More precisely: the indicator function of the primal polytope is the sum of the indicator functions of simplices with appropriate signs.

Let $P \subset \mathbb{R}^d$ be a polytope of dimension d (it is important that P is full-dimensional). We realize P as the intersection of a cone C with the hyperplane H defined by a grading γ: $H = \{x : \gamma(x) = 1\}$. The grading is an interior element of the dual cone $C^* = \{\lambda \in (\mathbb{R}^d)^* : \lambda(x) \geq 0 \text{ for all } x \in C\}$. In order to visualize the situation we take an auxiliary (irrelevant) cross-section Q of the dual cone:



Now suppose that we have a *generic* triangulation Δ of the dual cone where genericity is defined as follows: γ is not contained in any hyperplane that intersects any $\delta \in \Delta$ in a facet. Let $\delta \in \Delta$ be given, and denote the linear forms on $(\mathbb{R}^d)^*$ defining its facets by $\ell_1, \dots \ell_d \in (\mathbb{R}^d)^{**} = \mathbb{R}^d$. ($\ell_1, \dots \ell_d$ are the extreme rays of the dual of δ.) The hyperplanes defined by the vanishing of $\ell_1, \dots \ell_d$ decompose $(\mathbb{R}^d)^*$ into "orthants" that can be labeled by a sign vector $\sigma = (s_1, \dots, s_d) \in \{\pm 1\}^d$:

$$D(\delta, \sigma) = \{\alpha : (-1)^{s_i} \ell_i(\alpha) \geq 0\}.$$

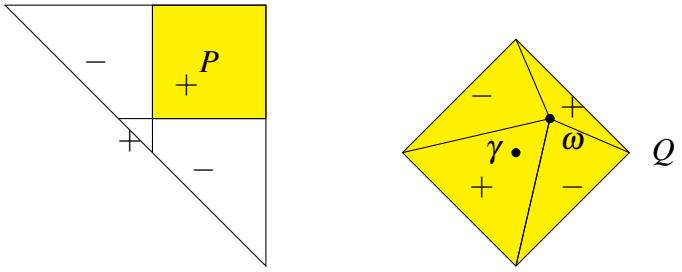
By the assumption on γ, there is *exactly one* sign vector σ such that γ lies in the interior of $D(\delta, \sigma)$. Consequently the hyperplane H intersects the dual $D(\delta, \sigma)^*$ in a polytope R_δ. (We identify $(\mathbb{R}^d)^{**}$ with $\mathbb{R}^d$.) Furthermore we set $e(\delta) = |\{i : s_i = -1\}|$.

Let ι_X denote the indicator function of a subset $X \subset \mathbb{R}^d$. Then

$$\iota_P(x) = \sum_{\delta \in \Delta} (-1)^{e(\delta)} \iota_{R_\delta}(x) \tag{2}$$

for all $x \in \mathbb{R}^d$ outside a union of finitely many hyperplanes. Since volume (lattice normalized or Euclidean) is additive on indicator functions this formula can be used for the computation of the volume of P. (We give a reference at the end of this section.)

In order to find a generic triangulation, Normaliz first computes a triangulation Δ_0 of C^* and saves the induced "hollow triangulation" that Δ_0 induces on the boundary of C^*. Then it finds a "generic" element $\omega \in C^*$ such that the "star" triangulation Δ of C^* in which every simplicial cone is the pyramid with apex ω and base in the hollow triangulation is generic.



Since ω almost inevitably has unpleasantly large coordinates, the polytopes R_δ have even worse rational vertices, and their volumes usually are rational numbers with very large numerators and denominators. This extreme arithmetical complexity limits the applicability of the signed decomposition.

Signed decomposition is asked for by

SignedDec

and blocked by

NoSignedDec

We show part of the terminal output for `strictBorda`:

```
...
Command line: -c ../example/strictBorda
Compute: Multiplicity
Working with dual cone
************************************************************
starting full cone computation
Starting primal algorithm with full triangulation ...
...
Computing  by signaed decomposition
Making hollow triangulation
...
Size of triangulation 100738
Size of hollow triangulation 324862
Trying to find geric vector
Trying to find generic linear combination of
164 107 65 125 116  66 ...  100
32 130 57 105 108 153 ...  139
...
Must increase coefficients
Trying to find generic linear combination of
270 228 347 407 399 280 ...167
227 362 305 135 354 272 ... 499

Generic with coeff 56 1
Computing multiplicity
Generic 15347 13130 19737 22927 ...9851
...

Mult (before ...) 1281727528...66511/25940255...784000000000
Mult (float) 4.94107527277e-05
```

The algorithm described in this section has been developed by Lawrence [31] in the language of linear programming, and [21] describes the floating point implementation in the package vinci [20]. We have learnt it from Filliman's paper [24], which contains a proof of equation (2). See also the references to older literature in [24].

Volume by signed decomposition allows distributed computing. See Appemdix F.1.

7.6.5. Fixed precision for signed decomposition

In very large computations the fractions that arise in the computation of volumes by signed decomposition can become gigantic (indeed, take gigabytes) so that their handling becomes impossible. Therefore Normaliz has a fixed precision option for volumes by signed decomposition. This means that the volumes of the simplices in the hollow triangulation are computed precisely as rational numbers, but are truncated to fixed precision before being added. The cone property to be used is

`FixedPrecision`

It defines the precision to be 10^{-100}. Then the precision of the final result is $\leq H * 10^{-100}$ where H is the number of simplices in the hollow triangulation. Therefore 10^{-100} should suffice for all computations that can be done at present.

If the default value of 100 is too large or too small it can be set by

`decimal_digits <n>`

in the input file.

We run `strictBorda_fixed_prec.in`:

```
amb_space 24
inequalities 9
...
Multiplicity
FixedPrecision
```

Then the terminal output ends by

```
Mult (before NoGradingDenom correction) 4941075272...6309/1000000...000000000
Mult (float) 4.94107527277e-05
```

and in the output file we find

```
multiplicity (fixed precision) = 4941075...1726309/100000000...00000000000
multiplicity (float) = 4.94107527277e-05
```

7.6.6. Comparing the algorithms

The computation times in the table were obtained on a compute server with a parallelization of 32 threads in order to save time for the big computations. The fast ones do not really profit from it. The optimal time is printed in bold face. If the default choice is different, it is indicted in italics.

	dim	#ext	#supp	signed dec	desc iso	descent	symm	symm sd	primal
`A553`	43	75	306955	–	**5:48 m**	–	–	–	*45:35 m*
`lo6`	16	720	910	–	**6.0 s**	2:16 m	–	–	*18:07 m*
`cross-24`	25	48	2^{24}	–	7:59 m	10:43 m	–	–	**7:55 m**
`CondEffPlur`	24	3928	30	**0.3 s**	2.5 s	0.9 s	6:28 m	31.3 s	41 h
`strictBorda`	24	6363	33	**2.0 s**	–	26.7 s	–	–	4:18 h

The decision for `lo6` is made without knowledge of the unexpectedly small number of support hyperplanes. This is a design decision of Normaliz: if the primal algorithm should apply, then it would be time consuming to compute the support hyperplanes beforehand. But in this case it is the wrong decision.

For `A553` it is unpredictable that descent with isomorphism types speeds up the computation of the volume – one would have at least to compute the automorphism group and see that the number of orbits of the support hyperplanes is really small.

One would expect that descent with isomomorphism types is very fast for `cross-24` since there is single orbit of support hyperplanes. But it takes time to find this out, and the primal algorithm is slightly faster.

`CondEffPlur` illustrates the evolution of volume computations in Normaliz. Though symmetrization is not the fastest choice for any of the examples in the table, it remains important since we have no better algorithm for the computation of the Hilbert series of `CondEffPlur` and related examples.

7.7. Checking the Gorenstein property

If the Hilbert series has been computed, one can immediately see whether the monoid computed by Normaliz is Gorenstein: this is the case if and only if the numerator is a symmetric polynomial, and Normaliz indicates that (see Section 2.8). However, there is a much more efficient way to check the Gorenstein property, which does not even require the existence of a grading: we must test whether the *dual* cone has degree 1 extreme rays. This amounts to checking the existence of an implicit grading on the dual cone.

This very efficient Gorenstein test is activated by the option `IsGorenstein`, equivalently `-G` on the command line. We take `5x5Gorenstein.in`:

```
amb_space 25
equations 11
1 1 1 1 1 -1 -1 -1 -1 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
...
1 1 1 1 0  0  0  0 -1  0  0  0 -1  0  0  0 -1  0  0  0 -1  0  0  0  0
IsGorenstein
```

In the output we see

```
Monoid is Gorenstein
Generator of interior
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

In fact, the Gorenstein property is (also) equivalent to the fact that the interior of our monoid is generated by a single element as an ideal, and this generator is computed if the monoid is Gorenstein. (It defines the grading under which the extreme rays of the dual cone have degree 1.)

If the monoid is not Gorenstein, Normaliz will print the corresponding message.

7.8. Symmetrization

Under certain conditions one can count lattice points in a cone C by mapping C to a cone C' of lower dimension and then counting each lattice point y in C' with the number of its lattice preimages. This approach works well if the number of preimages is given by a polynomial in the coordinates of y. Since C' has lower dimension, one can hope that its combinatorial structure is much simpler that that of C. One must of course pay a price: instead of counting each lattice point with the weight 1, one must count it with a polynomial weight. This amounts to a computation of a weighted Ehrhart series that we will discuss in Section 7.9. Similarly multiplicity can be computed as the virtual multiplicity of a polynomial after projection.

The availability of this approach depends on symmetries in the coordinates of C, and therefore we call it *symmetrization*. Normaliz tries symmetrization under the following condition: C is given by constraints (inequalities, equations, congruences, excluded faces) and the inequalities contain the sign conditions $x_i \geq 0$ for all coordinates x_i of C. (Coordinate hyperplanes may be among the excluded faces.) Then Normaliz groups coordinates that appear in all constraints and the grading (!) with the same coefficients, and, roughly speaking, replaces them by their sum. The number of preimages that one must count for the vector y of sums is then a product of binomial coefficients – a polynomial as desired. More precisely, if y_j, $j = 1,\dots,m$, is the sum of u_j variables x_i then

$$f(y) = \binom{u_1 + y_1 - 1}{u_1 - 1} \cdots \binom{u_m + y_m - 1}{u_m - 1}.$$

is the number of preimages of $(y_1,\dots,y_m)$. This approach to Hilbert series has been suggested by A. Schürmann [35].

Note that symmetrization requires an explicit grading. Moreover, it sets `NoGradingDenom`.

As an example we look again at the input for the Condorcet paradox:

```
amb_space 24
inequalities 3
1 1 1 1 1 1 -1 -1 -1 -1 -1 -1   1  1 -1 -1  1 -1   1  1 -1 -1  1 -1
1 1 1 1 1 1  1  1 -1 -1  1 -1  -1 -1 -1 -1 -1 -1   1  1  1 -1 -1 -1
1 1 1 1 1 1  1  1  1 -1 -1 -1   1  1  1 -1 -1 -1  -1 -1 -1 -1 -1 -1
nonnegative
total_degree
Multiplicity
```

The grading is completely symmetric, and it is immediately clear that the input is symmetric in the first 6 coordinates. But also the column of three entries -1 appears 6 times, and there are 6 more groups of 2 coordinates each (one group for each ± 1 pattern). With the suitable labeling, the number of preimages of$(y_1,\dots,y_8)$ is given by

$$f(y) = \binom{y_1+5}{5}(y_2+1)(y_3+1)(y_4+1)(y_5+1)(y_6+1)(y_7+1)\binom{y_8+5}{5}.$$

Normaliz finds the groups of variables that appear with the same sign pattern, creates the data

for the weighted Ehrhart series, and interprets it as the Hilbert series of the monoid defined by the input data.

However, there is a restriction. Since the polynomial arithmetic has its own complexity and Normaliz must do it in GMP integers, it makes no sense to apply symmetrization if the dimension does not drop by a reasonable amount. Therefore we require that

$$\dim C' \leq \frac{2}{3}\dim C).$$

If called with the option `-q`, Normaliz will try symmetrization, and also with `-v`, provided the multiplicity has not already been computed by the descent algorithm (see Section 7.6.2). If the inequality for $\dim C'$ is not satisfied, it will simply compute the Hilbert series or the multiplicity without symmetrization. (In default mode it of course tries symmetrization for the Hilbert series.)

Whenever Normaliz has used symmetrization, it writes the file `<project>.symm.out` that contains the data of the symmetrized object. In it you find the multiplicity of `<project>.out` as virtual multiplicity and the Hilbert series as weighted Ehrhart series.

If you use the option `Symmetrize`, then the behavior depends on the other options:

(1) If neither the `HilbertSeries` nor `Multiplicity` is to be computed, Normaliz writes only the output file `<project>.symm.out` computed with `SupportHyperplanes`.
(2) If one of these goals is to be computed, Normaliz will do the symmetrization, regardless of the dimension inequality above (and often this makes sense).

By doing step (1) first, the user gets useful information of what to expect by symmetrization. In a second run, one can add `HilbertSeries` or `Multiplicity` if (1) was satisfactory.

The Condorcet example is too small in order to demonstrate the power of symmetrization. A suitable example is `PluralityVsCutoff.in`:

```
winfried@ubuntu:~/Dropbox/git_normaliz/source$ time ./normaliz -c ../example/PluralityVsCutoff
                                                    \.....|
                    Normaliz 3.3.0                   \....|
                                                      \...|
     (C) The Normaliz Team, University of Osnabrueck   \..|
                      March  2017                       \.|
                                                         \|
************************************************************
Command line: -c ../example/PluralityVsCutoff
Compute: DefaultMode
Embedding dimension of symmetrized cone = 6
...
------------------------------------------------------------
transforming data... done.

real        0m2.655s
user        0m5.328s
sys        0m0.080s
```

The Hilbert series is computable without symmetrization, but you better make sure that there is no power failure for the next week if you try that. (The time above includes the Hilbert basis computed automatically in dual mode).

Another good example included in the distribution is `CondEffPlur.in`, but it takes some hours with symmetrization (instead of days without). For it, the dimension drops only from 24 to 13.

Symmetrization is a special type of computations with a polynomial weight, and therefore requires Normaliz to be built with CoCoALib.

In the computation of multiplicities via symmetrization Normaliz can use (implicitly or explicitly) signed decomposition, including fixed prrecision if asked for.

7.9. Computations with a polynomial weight

For a graded monoid M, which arises as the intersection $M = C \cap L$ of a rational cone$C$ and a lattice L, Normaliz computes the volume of the rational polytope

$$P = \{x \in \mathbb{R}_+M : \deg x = 1\},$$

called the multiplicity of M (for the given grading), the Hilbert series of M, and the quasipolynomial representing the Hilbert function. This Hilbert series of M is also called the Ehrhart series of P (with respect to L), and for the generalization introduced in this section we speak of Ehrhart series and functions.

The computations of these data can be understood as integrals of the constant polynomial $f = 1$, namely with respect to the counting measure defined by L for the Ehrhart function, and with respect to the (suitably normed) Lebesgue measure for the volume. Normaliz generalizes these computations to arbitrary polynomials f in n variables with rational coefficients. (Mathematically, there is no need to restrict oneself to rational coefficients for f.)

More precisely, set

$$E(f,k) = \sum_{x \in M, \deg x = k} f(x),$$

and call $E(f,\_)$ the *weighted Ehrhart function* for f. (With $f = 1$ we simply count lattice points.) The *weighted Ehrhart series* is the ordinary generating function

$$E_f(t) = \sum_{k=0}^{\infty} E(f,k)t^k.$$

It turns out that $E_f(t)$ is the power series expansion of a rational function at the origin, and can always be written in the form

$$E_f(t) = \frac{Q(t)}{(1-t^\ell)^{\operatorname{totdeg} f + \operatorname{rank} M}}, \qquad Q(t) \in \mathbb{Q}[t],\ \deg Q < \operatorname{totdeg} f + \operatorname{rank} M.$$

Here $\operatorname{totdeg} f$ is the total degree of the polynomial f, and ℓ is the least common multiple of the degrees of the extreme integral generators of M. See [19] for an elementary account, references and the algorithm used by Normaliz.

Note that `excluded_faces` is a homogeneous input type. For them the monoid M is replaced by the set

$$M' = C' \cap L$$

where $C' = C \setminus \mathscr{F}$ and $\mathscr{F}$ is the union of a set of faces (not necessarily facets) of C. What has been said above about the structure of the weighted Ehrhart series remains true. We discuss an example below.

It follows from the general theory of rational generating functions that there exists a quasipolynomial $q(k)$ with rational coefficients and of degree $\leq \operatorname{totdeg} f + \operatorname{rank} M - 1$ that evaluates to $E(f,k)$ for all $k \geq 0$.

Let $m = \operatorname{totdeg} f$ (we use this notation to distinguish the degree of the polynomial from the degree of lattice points) and f_m be the degree m homogeneous component of f. By letting k go to infinity and approximating f_m by a step function that is constant on the meshes of $\frac{1}{k}L$ (with respect to a fixed basis), one sees

$$q^{(j)}_{\operatorname{totdeg} f + \operatorname{rank} M - 1} = \int_P f_m \, d\lambda$$

where $d\lambda$ is the Lebesgue measure that takes value 1 on a basic mesh of $L \cap \mathbb{R}M$ in the hyperplane of degree 1 elements in $\mathbb{R}M$. In particular, the *virtual leading coefficient* $q^{(j)}_{\operatorname{totdeg} f + \operatorname{rank} M - 1}$ is constant and depends only on f_m. If the integral vanishes, the quasipolynomial q has smaller degree, and the true leading coefficient need not be constant. Following the terminology of commutative algebra and algebraic geometry, we call

$$(\operatorname{totdeg} f + \operatorname{rank} M - 1)! \cdot q_{\operatorname{totdeg} f + \operatorname{rank} M - 1}$$

the *virtual multiplicity* of M and f. It is an integer if f_m has integral coefficients and P is a lattice polytope.

The input format of polynomials has been discussed in Section 4.1.8.

The terminal output contains a factorization of the polynomial as well as some computation results. From the terminal output you may also recognize that Normaliz first computes the triangulation and the Stanley decomposition and then applies the algorithms for integrals and weighted Ehrhart series.

Remarks (1) Large computations with many parallel threads may require much memory due to the fact that very long polynomials must be stored. Another reason for large memory usage can be the precomputed triangulation or Stanley decomposition.

(2) You should think about the option `BottomDecomposition`. It will be applied to the symmetrized input. (Under suitable conditions it is applied automatically.)

(3) A priori it is not impossible that Normaliz replaces a given grading deg by $\deg / g$ where g is the grading denominator. If you want to exclude this possibility, set `NoGradingDenom`.

7.9.1. A weighted Ehrhart series

We discuss the Condorcet paradox again (and the last time), now starting from the symmetrized form. The file `Condorcet.symm.in` from the directory `example` contains the fol-

lowing:

```
amb_space 8
inequalities 3
1 -1 1 1 1 -1 -1 -1
1 1 -1 1 -1 1 -1 -1
1 1 1 -1 -1 -1 1 -1
nonnegative
total_degree
polynomial
1/120*1/120*(x[1]+5)*(x[1]+4)*(x[1]+3)*(x[1]+2)*(x[1]+1)*(x[2]+1)*
(x[3]+1)*(x[4]+1)*(x[5]+1)*(x[6]+1)*(x[7]+1)*(x[8]+5)*(x[8]+4)*
(x[8]+3)*(x[8]+2)*(x[8]+1);
```

We have seen this polynomial in Section 7.8 above.

From the Normaliz directory we start the computation by

```
./normaliz -cE example/Condorcet.symm
```

We could have used `--WeightedEhrhartSeries` instead of `-E` or put `WeightedEhrhartSeries` into the input file.

The file `Condorcet.symm.out` we find the information on the weighted Ehrhart series:

```
Weighted Ehrhart series:
1 5 133 363 ... 481 15 6
Common denominator of coefficients: 1
Series denominator with 24 factors:
1: 1  2: 14  4: 9

degree of weighted Ehrhart series as rational function = -25

Weighted Ehrhart series with cyclotomic denominator:
...
```

The only piece of data that we haven't seen already is the common denominator of coefficients. But since the polynomial has rational coefficients, we cannot any longer expect that the polynomial in the numerator of the series has integral coefficients. We list them as integers, but must then divide them by the denominator (which is1 in thus case since the weighted Ehrhart series is a Hilbert series in disguise). As usual, the representation with a denominator of cyclotomic polynomials follows.

And we have the quasipolynomial as usual:

```
Weighted Ehrhart quasi-polynomial of period 4:
0:   693959790182222163590774784000 20899225...000000 ... 56262656
1:   203475031022335179700809216000  7092764...648000 ... 56262656
2:   693308184929915219977568256000 20892455...168000 ... 56262656
```

```
3:   203475031022335179700809216000O  7092764...648000 ... 56262656
with common denominator: 693959790182222163590774784000O
```

The left most column indicates the residue class modulo the period, and the numbers in line k are the coefficients of the k-th polynomial after division by the common denominator. The list starts with $q_0^{(k)}$ and ends with (the constant) $q_{23}^{(k)}$. The interpretation of the remaining data is obvious:

```
Degree of (quasi)polynomial: 23

Expected degree: 23

Virtual multiplicity: 1717/8192
Virtual multiplicity (float) = 0.209594726562
```

Weighted Ehrhart series can be computed for polytopes defined by homogeneous or inhomogeneous input. Weighted Ehrhart series as a weighted variant of Hilbert series for unbounded polyhedra are not defined in Normaliz.

7.9.2. Virtual multiplicity

Instead of the option `-E` (or `--WeightedEhrhartSeries`) we use `-L` or (`--VirtualMultiplicity`). Then we can extract the virtual multiplicity from the output file.

The scope of computations is the same as for Weighted Ehrhart series.

7.9.3. An integral

In their paper *Multiplicities of classical varieties* (Proc. Lond. Math. Soc. 110 (2015), no. 4, 1033–1055) J. Jeffries, J. Montaño and M. Varbaro ask for the computation of the integral

$$\int\limits_{\substack{[0,1]^m\\ \sum x=t}} (x_1\cdots x_m)^{n-m} \prod_{1\le i<j\le m} (x_j-x_i)^2 \mathrm{d}\mu$$

taken over the intersection of the unit cube in $\mathbb{R}^m$ and the hyperplane of constant coordinate sum t. It is supposed that $t \le m \le n$. We compute the integral for $t = 2$, $m = 4$ and $n = 6$.

The polytope is specified in the input file `j462.in` (partially typeset in 2 columns):

```
amb_space 5          -1 0 0 0 1
inequalities 8       0 -1 0 0 1
1 0 0 0 0            0 0 -1 0 1
0 1 0 0 0            0 0 0 -1 1
0 0 1 0 0            equations 1
0 0 0 1 0            -1 -1 -1 -1 2
grading
```

```
unit_vector 5
polynomial
(x[1]*x[2]*x[3]*x[4])^2*(x[1]-x[2])^2*(x[1]-x[3])^2*
(x[1]-x[4])^2*(x[2]-x[3])^2*(x[2]-x[4])^2*(x[3]-x[4])^2;
```

The 8 inequalities describe the unit cube in $\mathbb{R}^4$ by the inequalities $0 \leq z_i \leq 1$ and the equation gives the hyperplane $z_1 + \cdots + z_4 = 2$ (we must use homogenized coordinates!). (Normaliz would find the grading itself.)

From the Normaliz directory the computation is called by

```
./normaliz -cI example/j462
```

where `-I` could be replaced by `--Integral`.

It produces the output in `j462.out` containing

```
integral  = 27773/29515186701000
integral  (float) = 9.40973210888e-10
```

As pointed out above, *Normaliz integrates with respect to the measure in which the basic lattice mesh has volume* 1. (this is $1/r!$ times the lattice normalized measure, $r = \dim P$.) In the full dimensional case that is just the standard Lebesgue measure. But in lower dimensional cases this often not the case, and therefore Normaliz also computes the integral with respect to this *Euclidean* measure:

```
integral (euclidean) = 1.88194642178e-09
```

Note that `Integral` automatically sets `NoGradingDenom` since the polytope must be fixed for integrals.

Note: integrals can be computed by signed decomposition, and Normaliz chooses this variant if it seems better. Nevertheless you can control it by `SignedDec` and `NoSignedDec`. Fixed precision set by `decimal_digits` is used for integrals as well.

7.10. Expansion of the Hilbert or weighted Ehrhart series

Normaliz can compute the expansion of the Hilbert function or the weighted Ehrhart function up to a given degree. To this end its expands the series. For the Hilbert function there is a second possibility by lattice point computation.

7.10.1. Series expansion

This is best explained by `CondorcetExpansion.in`:

```
amb_space 24
inequalities 3
1 1 1 1 1 1      -1 -1 -1 -1 -1 -1   1 1 -1 -1 1 -1     1 1 -1 -1 1 -1
```

```
1 1 1 1 1 1       1 1 -1 -1 1 -1    -1 -1 -1 -1 -1 -1   1 1 1 -1 -1 -1
1 1 1 1 1 1       1 1 1 -1 -1 -1     1 1 1 -1 -1 -1    -1 -1 -1 -1 -1 -1
nonnegative
total_degree
HilbertSeries
expansion_degree 50
```

By `expansion_degree 50` we tell Normaliz to compute the coefficients from degree 0 to degree 50 in the expansion of the Hilbert series. So the output contains

```
Expansion of Hilbert series
0: 1
1: 6
2: 153
3: 586
4: 7143
5: 21450
...
49: 817397314032054600
50: 1357391110355875044
```

If the shift is nonzero, it is automatically added to the degree so that the expansion always starts at the shift.

The expansion degree applies to the weighted Ehrhart series as well if it is computed.

There is nothing more to say, except that (in principle) there is another method, as discussed in the next section.

7.10.2. Counting lattice points by degree

As an example we look at `CondorcetRange.in`:

```
amb_space 24
inequalities 3
1 1 1 1 1 1      -1 -1 -1 -1 -1 -1   1 1 -1 -1 1 -1     1 1 -1 -1 1 -1
1 1 1 1 1 1       1 1 -1 -1 1 -1    -1 -1 -1 -1 -1 -1   1 1 1 -1 -1 -1
1 1 1 1 1 1       1 1 1 -1 -1 -1     1 1 1 -1 -1 -1    -1 -1 -1 -1 -1 -1
nonnegative
total_degree
constraints 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 <= 5
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 >= 3
Projection
NumberLatticePoints
HilbertSeries
expansion_degree 5
```

This input defines the polytope that is cut out from the cone (defined by the 3 inequalities) by the two inequalities that are defined as constraints (for clarity). These two inequalities mean that we want to compute the polytope of all points x in the cone satisfying the condition $3 \leq \deg x \leq 5$. We add `Projection` in conjunction with `NumebrLatticePoints` to keep Normaliz from choosing the primal algorithm, which would do the job as well, but much more slowly.

In the output we find

```
Hilbert series:
586 7143 21450
denominator with 0 factors:

shift = 3
```

Taking the shift into account, we see that there are 586 lattice points in degree 3, 7413 in degree 4 and 21450 in degree 5. But this becomes even more obvious by (the unnecessary) `expansion_degree 5`:

```
Expansion of Hilbert series
3: 586
4: 7143
5: 21450
```

With `NumberLatticePoints` the lattice points are not stored. Therefore very large numbers of lattice points can be computed. (But they must be produced, and the production process also needs some space, which however depends only on the dimension.)

7.10.3. Significant coefficients of the quasipolynomial

If the degree and simultaneously the period of the Hilbert or weighted Ehrhart quasipolynomial are large, the space needed to store it (usually with large coefficients) may exceed the available memory. Depending on the application, only a certain number of the coefficients may be significant. Therefore one can limit the number of highest coefficients that are stored and printed. We look at the input file `CondorcetN.in`:

```
amb_space 24
inequalities 3
1 1 1 1 1 1     -1 -1 -1 -1 -1 -1     1  1 -1 -1  1 -1     1  1 -1 -1  1 -1
1 1 1 1 1 1      1  1 -1 -1  1 -1    -1 -1 -1 -1 -1 -1     1  1  1 -1 -1 -1
1 1 1 1 1 1      1  1  1 -1 -1 -1     1  1  1 -1 -1 -1    -1 -1 -1 -1 -1 -1
nonnegative
total_degree
nr_coeff_quasipol 2
```

The output file shows the following information on the quasipolynomial:

```
Hilbert quasi-polynomial of period 4:
only 2 highest coefficients computed
```

```
their common period is 2
0:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 15982652919 56262656
1:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 15528493056 56262656
with common denominator = 693959790182222163590774784000 0
```

Normaliz computes and prints only as many components of the quasipolynomial as required by the common period of the printed coefficients. Coefficients outside the requested range are printed as 0.

The bound on the significant coefficients applies simultaneously to the Hilbert polynomial and the weighted Ehrhart quasipolynomial—usually one is interested in only one of them.

By default Normaliz computes the quasipolynomial only if the period does not exceed a preset bound, presently 10^6. If this bound is too small for your computation, you can remove it by the option

```
NoPeriodBound
```

7.11. Explicit dehomogenization

Inhomogeneous input for data in $\mathbb{R}^d$ is homogenized by an extra $(d+1)$-th coordinate. The dehomogenization sets the last coordinate equal to 1. Other systems may prefer the first coordinate. By choosing an explicit dehomogenization Normaliz can be adapted to such input. The file `dehomogenization.in`

```
amb_space 3
inequalities 2
-1 1 0
-1 0 1
dehomogenization
unit_vector 1
```

indicates that in this case the first variable is the homogenizing one. The output file

```
1 module generators
2 Hilbert basis elements of recession monoid
1 vertices of polyhedron
2 extreme rays of recession cone
3 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 2

size of triangulation   = 0
resulting sum of |det|s = 0

dehomogenization:
```

```
1 0 0


module rank = 1

************************************************************************

1 module generators:
1 1 1

2 Hilbert basis elements of recession monoid:
0 0 1
0 1 0

1 vertices of polyhedron:              3 support hyperplanes of ... (homogenized)
1 1 1                                  -1 0 1
                                       -1 1 0
2 extreme rays of recession cone:       1 0 0
0 0 1
0 1 0
```

shows that Normaliz does the computation in the same way as with implicit dehomogenization, except that now the first coordinate decides what is in the polyhedron and what belongs to the recession cone, roughly speaking.

Note that the dehomogenization need not be a coordinate. It can be any linear form that is nonnegative on the cone generators.

7.12. Projection of cones and polyhedra

Normaliz can not only compute projections (as has become visible in the discussion of project-and-float), but also export them if asked for by the computation goal

`ProjectCone`

As the computation goal says, only the cone is projected. Lattice data are not taken care of. The image of the projection is computed with the goals `SupportHyperplanes` and `ExtremeRays`, and the result is contained in an extra output file `<project>.ProjectCone.out`, similarly to the result of the integer hull computation. (All other computation goals are applied to the input cone.)

The image and the kernel a of the projection are complementary vector subspaces generated by unit vectors. Those spanning the image are selected by the entries 1 in the 0-1 vector `projection_coordinates` of the input file. As an example we take `small_proj.in`:

```
amb_space 6
cone 190
6 0 7 0 10 1
```

```
...
0 0 0 16 7 1
projection_coordinates
1 1 0 1 0  1
ProjectCone
```

As you can see from `small_proj.out`, almost nothing is computed for the input cone itself. (However, any further computation goal would change this.) The result of the projection is contained in `small_proj.ProjectCone.out`:

```
14 extreme rays
9 support hyperplanes

embedding dimension = 4
...
14 extreme rays:
 0  0  1 1
 0  0 17 1
...
11  0  5 1
11  0  6 1

9 support hyperplanes:
-1 -1 -1 20
...
 1  0  1 -1
```

An equivalent inhomogeneous input file is `small_proj_inhom.in`. Note that no computation goals are set for the projection – only support hyperplanes and extreme rays are computed (plus the automatically included data).

Polyhedra and polytopes are treated by Normaliz as intersections of cones and hyperplanes. The hyperplane is given by the grading in the homogeneous case and by the dehomogenization in the inhomogeneous case. For the projection of the polyhedron, the kernel of the projection must be parallel to this hyperplane. Normaliz checks this condition (automatically satisfied for inhomogeneous input) and transfers the grading or the dehomogenization, respectively, to the image. Therefore the image of the input polyhedron is indeed the polyhedron defined by the projection.

7.13. Nonpointed cones

Nonpointed cones and nonpositive monoids contain nontrivial invertible elements. The main effect is that certain data are no longer unique, or may even require a new definition. An important point to note is that cones always split off their unit groups as direct summands and the same holds for normal affine monoids. Since Normaliz computes only normal affine monoids, we can always pass to the quotient by the unit groups. Roughly speaking, all data

are computed for the pointed quotient and then lifted back to the original cone and monoid. It is inevitable that some data are no longer uniquely determined, but are unique only modulo the unit group, for example the Hilbert basis and the extreme rays. Also the multiplicity and the Hilbert series are computed for the pointed quotient. From the algebraic viewpoint this means to replace the field K of coefficients by the group ring L of the unit group, which is a Laurent polynomial ring over K: instead of K-vector space dimensions one considers ranks over L.

7.13.1. A nonpointed cone

As a very simple example we consider the right halfplane (`halfspace2.in`):

```
amb_space 2
inequalities 1
1 0
```

When run in default mode, it yields the following output:

```
1 Hilbert basis elements
1 lattice points in polytope (Hilbert basis elements of degree 1)
1 extreme rays
1 support hyperplanes

embedding dimension = 2
rank = 2 (maximal)
external index = 1
dimension of maximal subspace = 1

size of triangulation   = 1
resulting sum of |det|s = 1

grading:
1 0

degrees of extreme rays:
1: 1

Hilbert basis elements are of degree 1

multiplicity = 1

Hilbert series:
1
denominator with 1 factors:
1: 1

degree of Hilbert Series as rational function = -1
```

```
Hilbert polynomial:
1
with common denominator = 1

rank of class group = 0
class group is free

***********************************************************************

1 lattice points in polytope (Hilbert basis elements of degree 1):
1 0

0 further Hilbert basis elements of higher degree:

1 extreme rays:
1 0

1 basis elements of maximal subspace:
0 1

1 support hyperplanes:
1 0
```

In the preamble we learn that the cone contains a nontrivial subspace. In this case it is the vertical axis, and close to the end we see a basis of this subspace, namely $(0,1)$. This basis is always simultaneously a $\mathbb{Z}$-basis of the unit group of the monoid. The rest of the output is what we have gotten for the positive horizontal axis which in this case is a natural representative of the quotient modulo the maximal subspace, The quotient can always be embedded in the cone or monoid respectively, but there is no canonical choice. We could have gotten $(1,5)$ as the Hilbert basis as well.

Normaliz has found a grading. Of course it vanishes on the unit group, but is positive on the quotient monoid modulo the unit group.

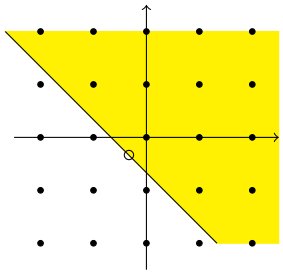
Note that the data of type "dimension" (embedding dimension, rank, rank of recession monoid in the inhomogeneous case, affine dimension of the polyhedron)) are measured before the passage to the quotient modulo the maximal subspace. The same is true for equations and congruences (which are trivial for the example above).

7.13.2. A polyhedron without vertices

We define the affine halfspace of the figure by `gen_inhom_nonpointed.in`:

```
amb_space 2
cone 3
1 -1
```

```
-1 1
0 1
vertices 1
-1 -1 3
```



It is clear that the "vertex" is not a vertex in the strict sense, but only gives a displacement of the cone. The output when run in default mode:

```
1 module generators
1 Hilbert basis elements of recession monoid
1 vertices of polyhedron
1 extreme rays of recession cone
2 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 2
internal index = 3
dimension of maximal subspace = 1

size of triangulation   = 1
resulting sum of |det|s = 3

dehomogenization:
0 0 1


module rank = 1

***********************************************************************

1 module generators:
0 0 1

1 Hilbert basis elements of recession monoid:
0 1 0

1 vertices of polyhedron:
```

```
0 -2 3

1 extreme rays of recession cone:
0 1 0

1 basis elements of maximal subspace:
1 -1 0

2 support hyperplanes of polyhedron (homogenized):
0 0 1
3 3 2
```

The "vertex" of the polyhedron shown is of course the lifted version of the vertex modulo the maximal subspace. It is not the input "vertex", but agrees with it up to a unit.

7.13.3. Checking pointedness first

Nonpointed cones will be an exception in Normaliz computations, and therefore Normaliz assumes that the (recession) cone it must compute is pointed. Only in rare circumstances it could be advisable to have this property checked first. There is no need to do so when the dual algorithm is used since it does not require the cone to be pointed. Moreover, if an explicit grading is given or a grading dependent computation is asked for, one cannot save time by checking the pointedness first.

The exceptional case is a computation, say of a Hilbert basis, by the primal algorithm in which the computation of the support hyperplanes needs very long time to be completed. If you are afraid this may happen, you can force Normaliz to compute the support hyperplanes right away by adding `IsPointed` to the computation goals. This is a disadvantage only if the cone is unexpectedly pointed.

7.13.4. Input of a subspace

If a linear subspace contained in the cone is known a priori, it can be given to Normaliz via the input type `subspace`. If Normaliz detects a `subspace`, it appends the rows of the matrix to the generators of the cone, and additionally the negative of the sum of the rows (since we must add the subspace as a cone). If `subspace` is combined with `cone_and_lattice`, then the rows of `subspace` are also appended to the generators of the lattice. It is not assumed that the vectors in `subspace` are linearly independent or generate the maximal linear subspace of the cone. A simple example (`subspace4.in`):

```
amb_space 4
cone 4
1 0 2 0
0 1 -2 1
0 0 0 1
```

```
0 0 0 -1
subspace 1
0 0 1 0
```

From the output:

```
2 lattice points in polytope (Hilbert basis elements of degree 1):
 0 1 0 0
 1 0 0 0

0 further Hilbert basis elements of higher degree:

2 extreme rays:
 0 1 0 0
 1 0 0 0

2 basis elements of maximal subspace:
 0 0 1 0
 0 0 0 1

2 support hyperplanes:
 0 1 0 0
 1 0 0 0
```

One should note that the maximal subspace is generated by the smallest face that contains all invertible elements. Therefore, in order to make all vectors in a face invertible, it is enough to put a single vector from the interior of the face into `subspace`.

7.13.5. Data relative to the original monoid

If original monoid generators are defined, there are two data related to them that must be read with care.

First of all, we consider the original monoid generators as being built from the vectors in `cone` or `cone_and_lattice` plus the vectors in `subspace` and additionally the negative of the sum of the latter (as pointed out above).

The test for "Original monoid is integrally closed" is correct – it returns `true` if and only if the original monoid as just defined indeed equals the computed integral closure. (There was a mistake in version 3.0.)

The "module generators over the original monoid" only refer to the *image* of the original monoid and the image of the integral closure *modulo the maximal subspace*. They do not take into account that the unit group of the integral closure may not be generated by the original generators. An example in which the lack of integral closedness is located in the unit group (`normface.in`):

```
amb_space 5
```

```
cone 4
0 0 0 1 1
1 0 0 1 1
0 1 0 1 1
0 0 1 1 1
subspace 4
0 0 0 0 1
1 0 0 0 1
0 1 0 0 1
1 1 2 0 1
```

From the output file:

```
...
dimension of maximal subspace = 4
original monoid is not integrally closed  in chosen lattice
unit group index = 2
...

1 lattice points in polytope (Hilbert basis elements of degree 1):
0 0 0 1 0
...
1 module generators over original monoid:
0 0 0 0 0
```

The original monoid is not integrally closed since the unit group of the integral closure is strictly larger than that of the original monoid: the extension has index 2, as indicated. The quotients modulo the unit groups are equal, as can be seen from the generator over the original monoid or the Hilbert basis (of the integral closure) that is contained in the original monoid.

7.14. Exporting the triangulation

The primal algorithm of Normaliz is based on "triangulations". What we call a "triangulation" here, is often only a collection of simplicial cones with properties that come close to those of a triangulation in the strict sense. Without being asked explicitly, Normaliz does not try to store and export its computational tool. In the file `<project>.out` you can sometimes see words "partial" and "nested". "Partial" means that only a subset of the cone has been triangulated, and "nested" is explained below. But if the user wants Normaliz to export a triangulation, then a triangulation in the strict sense is computed.

The option `Triangulation`, `-T` asks Normaliz to export a triangulation by writing the files `<project>.tgn` and `<project>.tri`:

`tgn` The file `tgn` contains a matrix of vectors (in the coordinates of $\mathbb{A}$) spanning the simplicial cones in the triangulation.

`tri` The file `tri` lists the simplicial subcones. There are two variants, depending on whether

ConeDecomposition had been set. Here we assume that ConeDecomposition is not computed. See Section 7.14.2 for the variant with ConeDecomposition.

The first line contains the number of simplicial cones in the triangulation, and the next line contains the number $m+1$ where $m = \operatorname{rank} \mathbb{E}$. Each of the following lines specifies a simplicial cone Δ: the first m numbers are the indices (with respect to the order in the file tgn) of those generators that span Δ, and the last entry is the multiplicity of Δ in $\mathbb{E}$, i.e., the absolute value of the determinant of the matrix of the spanning vectors (as elements of $\mathbb{E}$).

The following example is the 2-dimensional cross polytope with one excluded face (cross2.in). The excluded face is irrelevant for the triangulation.

```
amb_space 3
polytope 4
1 0
0 1
-1 0
0 -1
excluded_faces 1
-1 -1 1
Triangulation
StanleyDec
```

(The Stanley decomposition will be discussed in Section 7.16.) Its tgn and tri files are

```
tgn         tri
4           2
3           4
-1 0 1      1 2 3    2
0 -1 1      2 3 4    2
0 1 1
1 0 1
```

We see the 4 vertices $v_1, \ldots, v_4$ in homogenized coordinates in tgn and the 2 simplices (or the simplicial cones over them) in tri: both have multiplicity 2.

In addition to the files <project>.tgn and <project>.tri, also the file <object>.inv is written. It contains the data of the file <project>.out above the line of stars in a human and machine readable format.

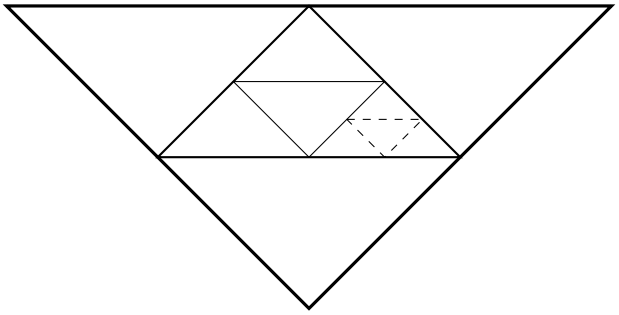
Note: Normaliz (now) allows the computation of triangulations for all input. In the homogeneous case it computes a triangulation of the (pointed quotient of the) cone C defined by the input. It can then be interpreted as a triangulation of a cross-section polytope if a grading is given. In the inhomogeneous case for which the input defines a polyhedron P, C is the cone over P. If P is a polytope, then a triangulation of C can again be identified with a triangulation of P. However, if P is unbounded, the the triangulation of C only induces a polyhedral decomposition of P into subpolyhedra whose compact faces are simplices.

7.14.1. Nested triangulations

We explain what we mean by a nested triangulation, even if it cannot be exported. If Normaliz has subdivided a simplicial cone of a triangulation of the cone C, the resulting decomposition of C may no longer be a triangulation in the strict sense. It is rather a *nested triangulation*, namely a map from a rooted tree to the set of full-dimensional subcones of C with the following properties:

(1) the root is mapped to C,
(2) every other node is mapped to a full dimensional simplicial subcone,
(3) the simplicial subcones corresponding to the branches at a node x form a triangulation of the simplicial cone corresponding to x.

The following figure shows a nested triangulation:

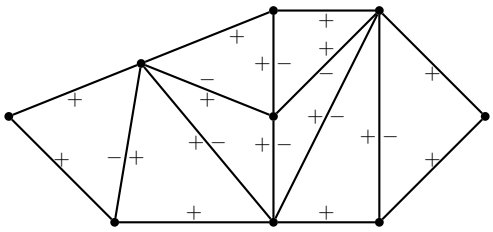


For the Normaliz computations, nested triangulations are as good as ordinary triangulations, but in other applications the difference may matter. With the option `-T`, Normaliz prints the leaves of the nested triangulation to the `tri` file. They constitute the simplicial cones that are finally evaluated by Normaliz. The subdivision can be blocked by `NoSubdivision`, independently of the computation goals.

The triangulation is always plain if `-T` is used, or if one of the refined triangulations below is computed.

7.14.2. Disjoint decomposition

Normaliz can export the disjoint decomposition of the cone that it has computed. This decomposition is always computed together with a full triangulation, unless only the multiplicity is asked for. It represents the cone as the disjoint union of semiopen simplicial subcones. The corresponding closed cones constitute the triangulation, and from each of them some facets are removed so that one obtains a disjoint decomposition. In the following figure, the facets separating the triangles are omitted in the triangle on the $-$ side.



If you want to access the disjoint decomposition, you must activate the computation goal `ConeDecomposition` or use the command line option is `-D`. As an example we compute `cross2.in` with the computation goal `ConeDecomposition`. The file `cross2.tri` now looks as follows:

```
2
7
1 2 3    2    0 0 0
2 3 4    2    0 0 1
```

As before the first line contains the size of the triangulation and the second is the number of entries of each row. The first 3 entries in each line are the indices of the extreme rays with respect to the `tgn` file and the fourth entry is the determinant. They are followed by a $0/1$ vector indicating the open facets in the order in which they are opposite to the extreme rays. If the corresponding entry is 1, the facet must be removed.

In our example all facets of the first simplicial cone are kept, and from the second simplicial cone the facet opposite to the third extreme ray (with index 4 relative to `tgn`) must be removed.

The disjoint decomposition which is the basis of all Hilbert series computations uses the algorithm suggested by Köppe and Verdoolaege [32].

7.15. Terrific triangulations

The basic triangulation computed by the Normaliz primal algorithm is a collection of simplicial cones each of which is generated by a subset of the generators of the cone C that is computed. Neither it is guaranteed that every generator of C appears as a generator of one of the simplicial cones, nor that every lattice point of a polytope participates in the triangulation. Moreover, there is no restriction on the determinants of the simplicial cones. Normaliz offers refined triangulations that satisfy the type of condition just mentioned. The refined triangulations start from the basic triangulation and refine it by iterated stellar subdivision. For background information we recommend [11], especially Chapter 2.

All these triangulations are "plain" and "full". Moreover, Normaliz can hold only a single triangulation. Therefore the refined triangulations exclude each other mutually.

The number of simplicial cones and the determinant sum appearing in the output file refer to the basic triangulation. The files `tri` and tgn contain the refined triangulation. It is not possible to derive a disjoint cone decomposition from a refined triangulation.

Warning. Refined triangulations can become very large. For example, for `small.in` the basic triangulation has 4580 simplicial cones, but the `LattcicePointTriangulation` has $739,303$

of them. For the unimodular triangulation the number rises to $49,713,917$, and the number of rays is $5,558,042$, whereas the number of lattice points is only $34,591$. You should use `LongLong` whenever possible.

In addition to the refined triangulations Normaliz offers *placing* and *pulling* triangulations which are defined combinatorially with respect to the order in which the generators are inserted.

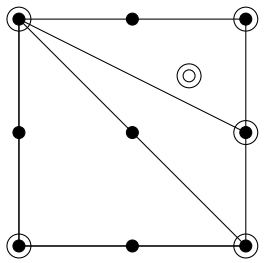
7.15.1. Just Triangulation

Our running example in the following is `square2.in` :

```
amb_space 3
cone 6
0 0 1
0 2 1
2 0 1
2 1 1
2 2 1
3 3 2
Triangulation
/* AllGeneratorsTriangulation */
/* LatticePointTriangulation */
/* UnimodularTriangulation */
/* PullingTriangulation */
```

The input file defines a square in the plane. For demonstration purposes we have added two generators to the first four that define the vertices of the square. The output is the basic triangulation:

```
tri                tgn
3                  6
4                  3
1 2 3    4         0 0 1
2 3 4    2         0 2 1
2 4 5    2         2 0 1
                   2 1 1
                   2 2 1
                   3 3 2
```



Normaliz sorts the generators lexicographically by default so that $(2,1,1)$ is inserted into cone building before $(2,2,1)$. If you add `KeepOrder` to the input, the basic triangulation will have only 2 triangles: the square is subdivided along its diagonal.

Note: The remark in Section 7.14 about the interpretation of general triangulations applies to the refined triangulations as well. The refined triangulations are computed for the cone over the polyhedron if the input is inhomogeneous. `LatticePointTriangulation` is only allowed if the input defines a polytope.

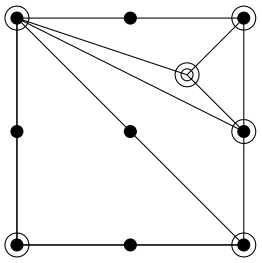
7.15.2. All generators triangulation

The option

`AllGeneratorsTriangulation`

asks for a triangulation such that all generators appear as rays in it. (It can be added to `Triangulation`, but can also be used alone.) For our example we get

```
tri                tgn
5                  6
4                  3
1 2 3    4         0 0 1
2 3 4    2         0 2 1
4 5 6    1         2 0 1
2 5 6    2         2 1 1
2 4 6    1         2 2 1
                   3 3 2
```



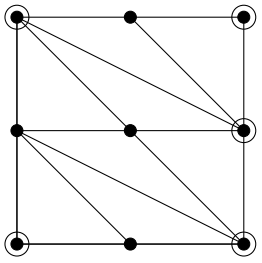
7.15.3. Lattice point triangulation

The option

`LatticePointTriangulation`

asks for a triangulation such that all lattice points of a polytope appear as vertices in it. (It can be added to `Triangulation`, but can also be used alone.) This option implies `LatticePoints` and, therefore, `NoGradingDenom`. For our example we get

```
tri                tgn
8                  10
4                  3
3 4 9    1         0 0 1
2 4 9    1         0 2 1
4 5 10    1        2 0 1
2 4 10    1        2 1 1
3 7 8    1         2 2 1
1 7 8    1         3 3 2
3 7 9    1         0 1 1
2 7 9    1         1 0 1
                   1 2 1
```



7.15.4. Unimodular triangulation

The option

`UnimodularTriangulation`

asks for a triangulation such that all generators appear as rays in it. (It can be added to

`Triangulation`, but can also be used alone.) The goal is a triangulation into simplicial cones of determinant 1. It implies `HilbertBasis` since all elements of the Hilbert basis must appear as rays in a unimodular triangulation, but in general further vectors must be used.

`UnimodularTriangulation` is not allowed in inhomogeneous computations or for algebraic polyhedra.

For our example above we get nothing new since lattice point triangulations of 2-dimensional lattice polytopes are automatically unimodular. We recommend to run `polytope.in` with the option `UnimodularTriangulation`.

7.15.5. Placing triangulation

This is very close to the basic triangulation that Normaliz computes, except that for the basic triangulation Normaliz takes the freedom to reorder the generators and to apply bottom decomnposition if it seems to be useful If you insist on

`PlacingTriangulation`

then these manipulations are excluded. The generators are inserted exactly in the order as Normaliz gets them. The triangulation is built incrementally: if the polytope (or cone) P is extended by the next generator x to form the polytope Q, then the triangulation is augmented by all simplices that arise as the convex (or conical) hulls of the new generators and the faces of the 'òld" triangulation that are in those facets of P which are *visible* from x. In particular this means that the new triangulation of Q is exactly the old of P if $x \in P$.

For our running example `PlacingTiangulations` gives the same result as `Triangulation`, and therefore we don't repeat the output.

Placing triangulations arise as *lexicographic* triangulations in the context of Gröbner bases of toric ideals; see Sturmfels [36, p. 67].

7.15.6. Pulling triangulation

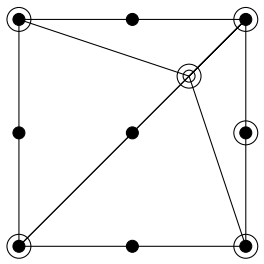
For the pulling triangulation,

`PullingTriangulation`

the generators are also inserted in the order given. However, the extension from P to Q follows are different rule: now the new triangulation is formed by taking the convex (or conical) hull of the new generator x and all faces of the 'òld" triangulation that are in those facets of P which are *invisible* from x and their collection replaces theòld triangulation – it is indeed a triangulation of Q. If $x \in P$, then the invisible facets of P are those that do not contain x. One consequence is that the last inserted generator is in all facets of the pulling triangulation.

For our running example we get

```
tri                tgn
4                  6
4                  3
1 2 6    6         0 0 1
1 3 6    6         0 2 1
2 5 6    2         2 0 1
3 5 6    2         2 1 1
                   2 2 1
                   3 3 2
```



Pulling triangulations arise as *reverse lexicographic* triangulations in the context of Gröbner bases of toric ideals; see Sturmfels [36, p. 67].

7.16. Exporting the Stanley decomposition

The computation goal `StanleyDec`, `-y` makes Normaliz write the files `<project>.tgn`, `<project>.dec` and `<project>.inv`. Stanley decomposition is contained in the file with the suffix `dec`. But this file also contains the inclusion/exclusion data if there are excluded faces:

(a) If there are any excluded faces, the file starts with the word `in_ex_data`. The next line contains the number of such data that follow. Each of these lines contains the data of a face and the coefficient with which the face is to be counted: the first number lists the number of generators that are contained in the face, followed by the indices of the generators relative to the `tgn` file and the last number is the coefficient.

(b) The second block (the first if there are no excluded faces) starts with the word `Stanley_dec`, followed by the number of simplicial cones in the triangulation.

For each simplicial cone Δ in the triangulation this file contains a block of data:

(i) a line listing the indices $i_1,\dots,i_m$ of the generators $v_{i_1},\dots,v_{i_m}$ relative to the order in `tgn` (as in `tri`, $m = \operatorname{rank}\mathbb{E}$);

(ii) a $\mu \times m$ matrix where μ is the multiplicity of Δ (see above).

In the notation of [15], each line lists an "offset" $x+\varepsilon(x)$ by its coordinates with respect to $v_{i_1},\dots,v_{i_m}$ as follows: if $(a_1,\dots,a_m)$ is the line of the matrix, then

$$x+\varepsilon(x) = \frac{1}{\mu}(a_1 v_{i_1} + \cdots + a_m v_{i_m}).$$

The `dec` file of the example `cross2.in` is

```
in_ex_data
1
2 3 4 -1
Stanley_dec
2
1 2 3        2 3 4
2            2
```

```
3           3
0 0 0       0 0 2
0 1 1       1 1 2
```

For reference: `cross2.tgn` is

```
4
3
-1 0 1
0 -1 1
0 1 1
1 0 1
```

There is 1 face in `in_ex_data` (namely the excluded one), it contains the 2 generators v_3 and v_4 and appears with multiplicity -1. The Stanley decomposition consists of 4 components of which each of the simplicial cone contains 2. The second offset in the second simplicial cone is

$$\frac{1}{2}(1v_2 + 1v_3 + 2v_4) = (1,0,2).$$

Another input file in `example` is `Stanleydec.in`.

Note: The computation and export of the Stanley decomposition in the inhomogeneous case is the same as that of triangulations: it is computed for the cone over the polyhedron.

7.17. Face lattice, f-vector and incidence matrix

In connection with "face", "lattice" means a partially ordered set with meet and join. Every face of a polyhedron is the intersection of the facets that contain it, and therefore Normaliz computes all intersections of facets, including the polyhedron itself and the empty set if the intersection of all facets should be empty.

The computation of the face lattice or just the f-vector might require very much memory. Therefore one should be careful if the dimension is large or there are many support hyperplanes.

The file `rationalFL.in` contains

```
amb_space 3
polytope 3
1/2 1/2
-1/3 -1/3
1/4 -1/2
HilbertSeries
FaceLattice
Incidence
```

representing a rational triangle. (Without `FaceLattice` it has been discussed in Section 2.5.) (`Incidence` is discussed below.)

Since the face lattice can be very large, it is returned as a separate file `<project>.fac`. For our example we get `rationalFL.fac`:

```
8
3

000 0
100 1
010 1
110 2
001 1
101 2
011 2
111 3
primal
```

The first line contains the number of faces, and the second the number of facets. The other lines list the faces F, encoded by a a 0-1-vector and an integer. The integer is the codimension of F. The 0-1-vector lists the facets containing F: the entry 1 at the i-th coordinate indicates that the i-th facet contains F.

The attribute `primal` indicates that we have computed the face lattice on the primal side. Dual face lattices will be introduced below.

The facets are counted as in the main output file `<project>.out`. (If you want them in a separate file, activate the output file `<project>.cst`.) In our case the support hyperplanes are:

```
-8 2 3
1 -1 0
2 7 3
```

So, for example, the face `011` is contained in the facets given by the linear forms $(1,-1,0)$ and $(2,7,3)$: it is the vertex $(1/2,1/2,1)$ (in homogeneous coordinates). The first face `000` is the intersection of the empty set of facets, namely the full triangle, and the last face `111` is the empty set.

Note that one can set a bound on the codimension of the faces that are to be computed. See Section 2.10.3.

One can retrieve the incidence matrix using the computation goal `Incidence`. It is printed to the file `<project>.inc`. The format of this files is illustrated by two examples. The first is `rationalFL` again, with its homogeneous input:

```
3
0
3

101
110
011
```

```
primal
```

The first line contains the number of support hyperplanes, the second the number of vertices of the polyhedron (0 for homogeneous input), and the third the number of extreme rays of the (recession) cone. The following lines list the incidence vectors of the facets. They are ordered in the same way as the support hyperplanes in the main output file. The incidence vector has entry 1 for an extreme ray (or) vertex) contained in the facet, and 0 otherwise. The extreme rays are ordered as in the main output file.

In the inhomogeneous case each line starts with the incidence for the vertices of the polyhedron, followed by the extreme rays of the recession cone. An example is `InhomIneqInc.inc`

```
3
2
1

01  1
10  1
11  0
primal
```

with its 2 vertices and 1 extreme ray of the recession cone.

7.17.1. Dual face lattice, f-vector and incidence matrix

Normaliz can also compute the face lattice of the dual cone. On the primal side this means that the face lattice is built bottom up and each face is represented by the extreme rays it contains. Since this is not possible for unbounded polyhedra, the dual versions are restricted to homogeneous input or inhomogeneous input defining polytopes. One application of the dual version is the computation of faces of low dimension which may be difficult to reach from the top if there are many facets. The numerical `face_codim_bound` now refers to the face codimension on the dual side. For example, if one wants to compute the edges of a polytope from the vertices, `face_codim_bound` must be set to 2 since the edges define codimension 2 faces of the dual polytope.

An example (`cube_3_dual_fac.in`):

```
amb_space 3
constraints 6 symbolic
x[1] >= 0;
x[2] >= 0;
x[3] >= 0;
x[1] <= 1;
x[2] <= 1;
x[3] <= 1;
DualFaceLattice
DualIncidence
```

```
face_codim_bound 2
```

In the output file we see

```
dual f-vector (possibly truncated):
12 8 1
```

which is the f-vector of the dual polytope (or cone) starting from codimension 2 and going up to codimension 0.

The dual face lattice up to codimension 2 is given by is given by

```
21
8

00000000 0
10000000 1
...
00000011 2
dual
```

Indeed, we have 21 faces in that range, and each face is specified by the vertices (or extreme rays) it contains. The attribute `dual` helps to recognize the dual situation.

The dual incidence matrix lists the support hyperplanes containing the vertices (or extreme rays):

```
8
0
6

000111
...
111000
dual
```

For the cube defined by inhomogeneous input we have 8 vertices of the polyhedron, 0 extreme rays of the recession cone and 6 facets.

Primal and dual versions of face lattice and incidence, respectively, are printed to the same file. Therefore only one of them is allowed.

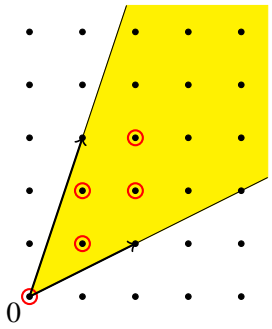
7.18. Module generators over the original monoid

Suppose that the original generators are well defined in the input. This is always the case when these consists just of a `cone` or a `cone_and_lattice`. Let M be the monoid generated by them. Then Normaliz computes the integral closure N of M in the effective lattice $\mathbb{E}$. It is often interesting to understand the difference set $N \setminus M$. After the introduction of a field K of coefficients, this amounts to understanding $K[N]$ as a $K[M]$-module. With the option

`ModuleGeneratorsOverOriginalMonoid, -M` Normaliz computes a minimal generating set T of this module. Combinatorially this means that we find an irreducible cover

$$N = \bigcup_{x \in T} x + M.$$

Note that $0 \in T$ since $M \subset N$.



As an example, we can run `2cone.in` with the option `-M` on the command line. This yields the output

```
...
4 Hilbert basis elements:
1 1
1 2                  5 module generators over original monoid:
1 3                   0 0
2 1                   1 1
                      1 2
2 extreme rays:       2 2
1 3                   2 3
2 1
```

In the nonpointed case Normaliz can only compute the module generators of N/N_0 over $M/(M \cap N_0)$ where N_0 is the unit group of N. If $M_0 \neq M_0$, this is not a system of generators of M over N.

7.18.1. An inhomogeneous example

Let us have a look at a very simple input file (`genmod_inhom2.in`):

```
amb_space 2
cone 2
0 3
2 0
vertices 1
0 0 1
ModuleGeneratorsOverOriginalMonoid
```

The cone is the positive orthant that we have turned into a polyhedron by adding the vertex $(0,0)$. The original monoid is generated by $(2,0)$ and $(0,3)$.

In addition to the original monoid M and its integral closure N we have a third object, namely the module P of lattice points in the polyhedron.We compute

1. the system of generators of P over N (the `module generators`) and
2. the system of generators of P over N (the `module generators over original monoid`).

We do not compute the system of generators of N over M (that we get in the homogeneous case).

The output:

```
1 module generators
2 Hilbert basis elements of recession monoid
1 vertices of polyhedron
2 extreme rays of recession cone
6 module generators over original monoid
3 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 2
internal index = 6

size of triangulation   = 1
resulting sum of |det|s = 6

dehomogenization:
0 0 1


module rank = 1

***********************************************************************

1 module generators:
 0 0 1

2 Hilbert basis elements of recession monoid:
 0 1 0
 1 0 0

1 vertices of polyhedron:
 0 0 1

2 extreme rays of recession cone:
 0 1 0
```

```
 1 0 0

6 module generators over original monoid:
 0 0 1
 0 1 1
 0 2 1
 1 0 1
 1 1 1
 1 2 1

3 support hyperplanes of polyhedron (homogenized):
 0 0 1
 0 1 0
 1 0 0
```

7.19. Lattice points in the fundamental parallelepiped

Let $u_1,\dots,u_n$ be linearly independent vectors in $\mathbb{Z}^d \subset \mathbb{R}^d$. They span a simplicial cone C. Moreover let U be the subgroup of $(\mathbb{R}^d,+)$ generated by $u_1,\dots,u_n$ and let $v \in \mathbb{R}^d$. We are interested in the shifted cone $C' = v+C$. We assume that C' contains a lattice point. This need not be true if $n < s$, but with our assumption we can also assume that $n = d$ after the restriction to the affine space spanned by C'. The *fundamental* parallelepiped of C (with respect to U) is

$$F = \mathrm{par}(u_1,\dots,u_d) = \{q_q u_1 + \cdots + q_d u_d : 0 \le q_i < 1\}.$$

Set $F' = v+F$. Then the translates $u+F'$, $u \in U$, tile $\mathbb{R}^d$; so F' is a fundamental domain for the action of U on $\mathbb{R}^d$ by translation, and we call it F' the *fundamental* parallelepiped of C' (with respect to U). Every point in $\mathbb{R}^d$ differs from exactly one point in F' by an element of U. This holds in particular for the lattice points.

One of the main basic tasks if Normaliz is the computation of the lattice points in F', especially in the case $v = 0$ (but not only). Looking back at the examples in Section 7.18, we see that we can in fact compute and export these lattice points via the computation goal `ModuleGeneratorsOverOriginalMonoid`.

Often however, an additional complication comes up: we must shift F' by an infinitesimally small vector in order to exclude certain facets of C'. This would be difficult in Normaliz without the input type `open_facets` (see Section 4.15). Recall that this is a 0-1-vector whose entries 1 indicate which facets must be avoided: if its i-th entry is 1, then the facet opposite to $v+u_i$ must be made "open".

The input file `no_open_facets.in` is

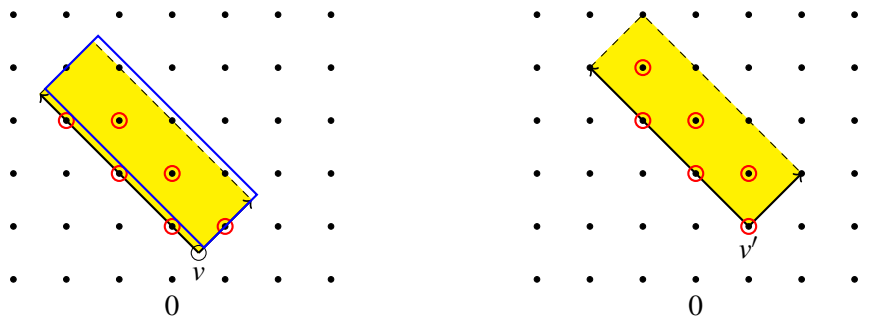
```
amb_space 2
cone 2
1 1
-3 3
```

```
vertices 1
1/2 1/2 1
ModuleGeneratorsOverOriginalMonoid
```

Then `no_open_facets.out` contains

```
6 module generators over original monoid:
-2 3 1
-1 2 1
-1 3 1
 0 1 1
 0 2 1
 1 1 1
```

These are 6 encircled points in the left figure.



Now we add

```
open_facets
1 0
```

to the input (to get `open_facets.in`). We have tried to indicate the infinitesimal shift by the blue rectangle in the left figure. The computation yields

```
6 module generators over original monoid:
-1 3 1
-1 4 1
0 2 1
0 3 1
1 1 1
1 2 1
```

which are the encircled lattice points in the right figure. It is explained in Section 4.15 how the new vector v' is computed.

Note that the lattice points are listed with the homogenizing coordinate 1. In fact, both `vertices` and `open_facets` make the computation inhomogeneous. If both are missing, then the lattice points are listed without the homogenizing coordinate. If you want a uniform format for the output, you can use the zero vector for `open_facets` or the origin as the vertex.

Both options change the result only to the extent that the homogenizing coordinate is added.

7.20. Semiopen polyhedra

A *semiopen polyhedron* P is a subset of $\mathbb{R}^d$ defined by system of inequalities $\lambda_i(x) \geq 0$, $i = 1,\dots,u$, and $\lambda_i(x) > 0$, $i = u+1,\dots,v$, where $\lambda_1,\dots,\lambda_v$ are affine linear forms. Normaliz can check whether P is empty and compute Hilbert/Ehrhart series if P is a semiopen polytope.

The inequalities $\lambda_i(x) > 0$, $i = u+1,\dots,v$, must be defined by `excluded_faces` in the homogeneous case and `inhom_excluded_faces` in the inhomogeneous case. (Don't use `strict_inequalities`; they have a different effect.) These input types can be combined with generators and other constraints.

Let $\overline{P}$ be the closed polyhedron defined by the inequalities $\lambda_i(x) \geq 0$, $i = 1,\dots,u$ and the "weak" inequalities $\lambda_i(x) \geq 0$, $i = u+1,\dots,v$. Then $\overline{P}$ is the topological closure of P, provided $P \neq \emptyset$. The main object for Normaliz is $\overline{P}$, but the computation is restricted to P for the following goals if `excluded_faces` or `inhom_excluded_faces` are present in the input:

```
HilbertSeries  EhrhartSeries  WeightedEhrhartSeries
      StanleyDecomposition   IsEmptySemiOpen
```

See Section 2.10.1 for a typical example of `HilbertSeries`. For all other computation goals `excluded_faces` and `inhom_excluded_faces` are simply ignored. Note that for lattice points in P the inequalities $\lambda_i(x) > 0$, $i = u+1,\dots,v$, can be replaced by $\lambda_i(x) \geq 1$ (if the λ_i have integral coefficients). Therefore lattice points in semiopen polyhedra can be computed as well. But they require a different input.

Note that Normaliz throws a `BadInputException` if you try to compute one the first four goals above for the empty set.

Let us have a look at two examples. In the first P is empty, in the second P is nonempty.

```
IsEmpty.in                      IsNonEmpty.in

amb_space 1                     amb_space 1
inequalities 1                  inequalities 1
1                               1
inhom_excluded_faces 1          inhom_excluded_faces 1
-1 0                            -1 1
IsEmptySemiOpen                 EhrhartSeries
                                IsEmptySemiOpen
```

The empty semiopen polytope is defined by the inequalities $\lambda_1(x) \geq 0$ and $\lambda_2(x) < 0$. In the second example the second inequality is replaced by $\lambda_2(x) < 1$.

The first output file:

```
1 vertices of polyhedron
0 extreme rays of recession cone
```

```
1 support hyperplanes of polyhedron (homogenized)

1 excluded faces

embedding dimension = 2
affine dimension of the polyhedron = 0
rank of recession monoid = 0 (polyhedron is polytope)

dehomogenization:
0 1

Semiopen polyhedron is empty
Covering face:
-1 0
...
```

We are informed that the semiopen polyhedron P is empty. Moreover, we see an excluded face that covers $\overline{P}$ and forces P to be empty. All other data refer to $\overline{P} = \{0\}$.

Now the output for the nonempty semiopen polytope:

```
2 vertices of polyhedron
0 extreme rays of recession cone
2 support hyperplanes of polyhedron (homogenized)

1 excluded faces

embedding dimension = 2
affine dimension of the polyhedron = 1 (maximal)
rank of recession monoid = 0 (polyhedron is polytope)

dehomogenization:
0 1

Ehrhart series:
1
denominator with 2 factors:
1: 2

shift = 1

degree of Ehrhart Series as rational function = -1

The numerator of the Ehrhart series is symmetric.

Ehrhart polynomial:
0 1
with common denominator = 1
```

```
Semiopen polyhedron is nonempty
```

Note that the Ehrhart series is computed for the interval $[0,1)$. All other data are computed for $[0,1]$.

7.21. Rational lattices

It is sometimes desirable to work in a sublattice of $\mathbb{R}^d$ that is not contained in $\mathbb{Z}$. Such lattices can be defined by the input type `rational_lattice`. In the inhomogeneous case the origin can be moved by `rational_offset`. Note that a finitely generated $\mathbb{Z}$-submodule of $\mathbb{Q}^d$ is automatically discrete. An example input file (`ratlat_2.in`):

```
amb_space 2
vertices 3
0 0 1
0 1 1
1 0 1
rational_lattice 2
1/2 -1/3
1 1/2
rational_offset
1 0
EhrhartSeries
HSOP
```

Though the origin is shifted by an integral vector, `rational_offset` has to be used. Conversely, if `rational_offset` is in the input, the lattice can only be defined by `rational_lattice`.

Normaliz must return the results by integer vectors. Therefore it scales the coordinate axes of $\mathbb{Q}^d$ in such a way that the vectors given in `rational_lattice` and `rational_offset` become integral with respect to the scaled coordinate axes. The output:

```
3 lattice points in polytope (module generators)
0 Hilbert basis elements of recession monoid
3 vertices of polyhedron
0 extreme rays of recession cone
3 support hyperplanes of polyhedron (homogenized)

embedding dimension = 3
affine dimension of the polyhedron = 2 (maximal)
rank of recession monoid = 0 (polyhedron is polytope)

scaling of axes
2 6

dehomogenization:
```

```
0 0 1


module rank = 3

Ehrhart series (HSOP):
1 2 3 4 8 8 10 10 10 9 8 4 4 2 1
denominator with 3 factors:
1: 1  7: 2

degree of Ehrhart Series as rational function = -1

...1

Ehrhart quasi-polynomial of period 7:
0:   7 5 6
...
with common denominator = 7

**************************************************************************

3 lattice points in polytope (module generators):
0 4 1
1 2 1
2 0 1

0 Hilbert basis elements of recession monoid:

3 vertices of polyhedron:
0  0 7
0 42 7
2  0 1

0 extreme rays of recession cone:

3 support hyperplanes of polyhedron (homogenized):
-3 -1 6
0  1 0
1  0 0

1 congruences:
3 5 1 7

3 basis elements of generated  lattice:
1 0 -3
0 1  2
```

```
0 0  7
```

The vector following `scaling of axes` contains the inverses of the scaling factors of the basis elements of $\mathbb{Q}^d$. In the example above the first basis vector is divided by 2 and the second by 6. Thus the ambient lattice has changed from $\mathbb{Z}$ to $A = \mathbb{Z}(1/2,0) + \mathbb{Z}(0,1/6)$. We can see from the appearance of a congruence that the lattice $L = \mathbb{Z}(1/2,-1/3) + \mathbb{Z}(1,12)$ is strictly contained in A. If the rank were smaller than 2, equations would appear.

The 3 lattice points, in original coordinates, are $(0,2/3)$, $(1/2,1/3)$ and $(1,0)$. The last is our origin.

Since certain input types do not allow division of coordinates they are excluded by `rational_lattice` and `rational_offset`. See Section 8.2 for a list (with the inevitable changes).

7.22. Automorphism groups

The *rational automorphism group* $\mathrm{Aut}_{\mathbb{Q}}(P)$ of a rational polyhedron $P \subset \mathbb{R}^d$ is the group of all rational affine-linear transformations α of $\mathrm{aff}(P)$ that satisfy $\alpha(P) = P$. In general, this group is infinite. For example, if P is a cone of positive dimension, then $\mathrm{Aut}_{\mathbb{Q}}(P)$ contains the multiplicative group of positive rational numbers as a subgroup. At the other extreme, if P is a polytope, then $\mathrm{Aut}_{\mathbb{Q}}(P)$ is a finite group since every automorphism permutes the finitely many vertices of P and is uniquely determined by its values on them. Often one is interested in subgroups of $\mathrm{Aut}_{\mathbb{Q}}(P)$, for example the isometries in it or the automorphisms that permute the lattice points.

Normaliz computes only subgroups of $\mathrm{Aut}_{\mathbb{Q}}(P)$ that permute a given finite set G of "generators". This subgroup is denoted by $\mathrm{Aut}_{\mathbb{Q}}(P;G)$. Bremner et al. [6] have shown how to compute $\mathrm{Aut}_{\mathbb{Q}}(P;G)$ by reducing this task to finding the automorphisms of a weighted graph, and the latter task can be solved efficiently by nauty [33]. We use the same approach (with variations).

Every polyhedron defines its face lattice as a purely combinatorial object, and it makes also sense to consider the automorphisms of the face lattice that we call *combinatorial* automorphisms of P. All the automorphism groups that Normaliz computes are subgroups of the combinatorial automorphism group in a natural way. (This does not necessarily apply to `AmbientAutomorphisms` and `InputAutomorphisms`.)

The automorphism group is contained in the extra output file `<project>.aut`. Its contents are explained in the following.

Note:

(1) If a grading is defined, then Normaliz computes only automorphisms that preserve the grading.
(2) The automorphism groups of a nonpointed polyhedron (as far as they can be computed) are those of the quotient by the maximal subspace. (This does not necessarily apply to `AmbientAutomorphisms` and `InputAutomorphisms`.)
(3) Even if the automorphism groups of different types coincide for a polyhedron P, the output files can differ since some details of the algorithms depend on the type and may

yield different systems of generators for the same group.

(4) Only one type of automorphism group can be computed in a single run of Normaliz (or a call of the libnormaliz function `compute`). (This may change in then future.)

The examples below are very simple so that the results can be verified directly. The reader is advised to try some larger examples, say `lo6`, `bo5`, `A543`, `6x6`.

Normaliz can compute groups of automorphisms that only need the input and do not require the passage from extreme rays to facets or conversely. They are discussed in the last two subsections. Their main advantage is that they do not need extreme rays *and* facets, but only the input vectors. Therefore automorphism groups (with some restrictions) can be computed in cases in which the sheer number of extreme rays or facets prevent the computation of the more refinded versions.

The groups computed from 'raw" input must be interpreted with care. They are not necessarily intrinsic data of the polyhedron (and lattice) they represent. We will see an example in Section 7.22.6. If you run `normaliz` with an input file, then the raw automorphism groups are computed before any other data so that there is no ambiguity what is meant by "input". In interactive mode this may depend on the order of computations and in particular can change if the cone is modified after construction. For this type of automorphism group Normaliz saves the reference input with the automorphism group. It is printed into the `aut` file and can be retrieved from `libnormaliz`.

As it can be done with reasonable effort, Normaliz checks whether the computed group consists of integral automorphisms. The output files therefore contain one of the following alternatives:

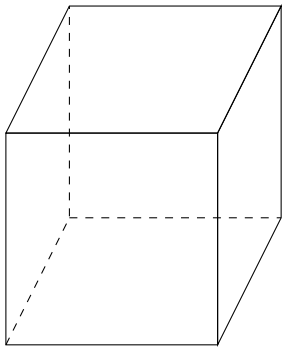
```
Automorphisms are integral
Automorphisms are not integral
Integrality not known
```

This information is always given, even if it is a priori known.

7.22.1. Euclidean automorphisms

Normaliz restricts the computation of euclidean automorphisms of a polyhedron P, i.e., rigid motions that map P onto itself, to polytopes P. We briefly discuss the problem for general polyhedra below. As a simple example we choose the cube of dimension 3 (`cube_3.in`):

```
amb_space 3
constraints 6 symbolic
x[1] >= 0;
x[2] >= 0;
x[3] >= 0;
x[1] <= 1;
x[2] <= 1;
x[3] <= 1;
EuclideanAutomorphisms
```



The file `cube_3.aut` contains the following:

```
Euclidean automorphism group of order 48
Integrality not known
************************************************************************
3 permutations of 8 vertices of polyhedron

Perm 1: 1 3 2 4 5 7 6 8
Perm 2: 1 2 5 6 3 4 7 8
Perm 3: 2 1 4 3 6 5 8 7

Cycle decompositions

Perm 1: (2 3) (6 7) --
Perm 2: (3 5) (4 6) --
Perm 3: (1 2) (3 4) (5 6) (7 8) --

1 orbits of vertices of polyhedron

Orbit 1 , length 8:  1 2 3 4 5 6 7 8

************************************************************************
3 permutations of 6 support hyperplanes

Perm 1: 1 3 2 5 4 6
Perm 2: 2 1 3 4 6 5
Perm 3: 1 2 4 3 5 6

Cycle decompositions

Perm 1: (2 3) (4 5) --
Perm 2: (1 2) (5 6) --
Perm 3: (3 4) --

1 orbits of support hyperplanes

Orbit 1 , length 6:  1 2 3 4 5 6
```

The automorphism group has order 48. The system of generators computed by nauty has 3 elements, listed as permutations of the extreme rays, and, in the second part, as permutations of the facets. `Perm 1: 1 3 2 4 5 7 6 8` says that the first permutation maps vertex 1 to itself, vertex 2 to vertex 3 etc. The reference order of the vertices is the one in which they are listed in `cube_3.out`:

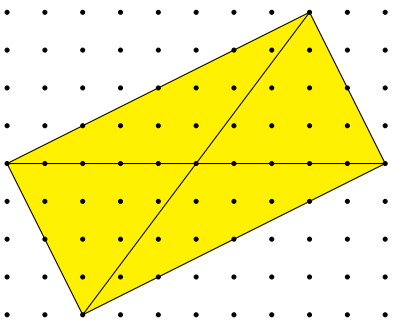
```
8 vertices of polyhedron:
0 0 0 1
0 0 1 1
```

```
0 1 0 1
0 1 1 1
1 0 0 1
1 0 1 1
1 1 0 1
1 1 1 1
```

The cycle decompositions show that all generators of the euclidean automorphism group have order 2. It is a good exercise to identify them geometrically.

Both the vertices and the facets form a single orbit for the 3-cube. As a simple example for which this not the case we take `pythagoras.in`:

```
amb_space 2
vertices 4
5 0 1
-5 0 1
3 4 1
-3 -4 1
EuclideanAutomorphisms
```



We get

```
Euclidean automorphism group of order 4
...
...
2 permutations of 4 support hyperplanes

Perm 1: 2 1 4 3
Perm 2: 1 3 2 4

Cycle decompositions

Perm 1: (1 2) (3 4) --
Perm 2: (2 3) --

1 orbits of support hyperplanes

Orbit 1 , length 4:  1 2 3 4
```

Clearly, this rectangle is not a square.

The euclidean automorphism group of a rational polyhedron with vertices, and in particular the euclidean automorphism group of a pointed cone, is finite and can be computed. For the cone it would be necessary to find points on the extreme rays that have distance 1 from the origin. In general this requires the extension of $\mathbb{Q}$ by square roots. In principle such extensions are accessible to Normaliz (see Section 8).

The euclidean automorphisms can only be computed if the input defines a polytope – it is not enough that the quotient by the maximal subspace does this.

7.22.2. Rational automorphisms

Also the computation of rational automorphism groups is restricted to polytopes in Normaliz. Let us take up the rectangle from `pythagoras.in` again, this time asking for rational automorphisms (`pythagoras_rat.in`):

```
amb_space 2
...
RationalAutomorphisms
```

Result:

```
Rational automorphism group of order 8
Automorphisms are not integral
********************************************************************************
2 permutations of 4 vertices of polyhedron

Perm 1: 1 3 2 4
Perm 2: 2 1 4 3
```

This is (hopefully) expected: as an object of rational linear geometry, every rectangle is isomorphic to a square whose automorphism group (in any reasonable sense) is of order 8, namely the dihedral group of this order.

7.22.3. Integral automorphisms

In general, euclidean and rational automorphisms do not map lattice points in polyhedra to lattice points. If we want to exploit automorphism groups in the computation of lattice points or enumerative invariants of polyhedra, we can only depend on integral automorphisms.

Consider a rational pointed cone $C \subset \mathbb{R}^d$. Let $L \subset \mathbb{Z}^d$ be a sublattice such that $L \cap \mathbb{Q}C$ spans $\mathbb{Q}C$ (the situation in which we compute Hilbert bases and Hilbert series). We define $\mathrm{Aut}_L(C)$ as the group of rational automorphisms of C that map L onto itself. On the one hand, such an automorphism must permute the Hilbert basis H of the monoid $C \cap L$. On the other hand, H generates the lattice L as a group, and therefore $\mathrm{Aut}_L(C) = \mathrm{Aut}_{\mathbb{Q}}(C;H)$. It follows that $\mathrm{Aut}_L(C)$ is a finite group, and that it can be computed as the group of rational automorphisms permuting a finite set of generators of C.

For a rational polyhedron P we pass to the cone $C(P)$ and the corresponding extension L' of L. Then $\mathrm{Aut}_L(P)$ is the subgroup of $\mathrm{Aut}_{L'}(C(P))$ of those automorphisms that map P onto itself. We simply speak of *integral* automorphisms, assuming that the lattice L is fixed.

If we had to always find the Hilbert basis first, then it would often be very hard to compute integral automorphism groups, and it would be impossible in the future to use the integral automorphisms in the computation of Hilbert bases. Fortunately one often gets away without

computing the Hilbert basis, and Normaliz only uses it as the last resort (as in the example below).

Again, let us consider our rectangle, but this time we compute the integral automorphisms (`pythagoras_int.in`).

```
amb_space 2
...
Automorphisms
```

Note that integral automorphisms are asked for by `Automorphisms` without an attribute since integral automorphisms are considered the standard choice for Normaliz.

Since an automorphism of a rectangle must permute the diagonals, and these have different numbers of lattice points, the integral automorphisms must fix them, and only the point reflection at the origin remains:

```
Integral automorphism group of order 2
Automorphisms are integral
**********************************************************************
1 permutations of 4 vertices of polyhedron

Perm 1: 4 3 2 1
...
```

Note that integral automorphisms in general depend on the choice of the reference lattice L. For our rectangle R, if we replace the full lattice $\mathbb{Z}^2$ by the sublattice L spanned by the vertices, then $\mathrm{Aut}_L(R)$ is simply the rational automorphism group of the polytope. You can test this by adding

```
lattice 4
5 0
-5 0
3 4
-3 -4
```

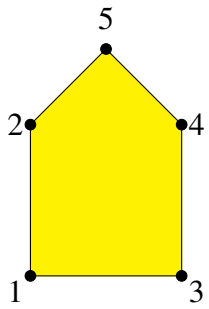
to the input file.

7.22.4. Combinatorial automorphisms

For polytopes the combinatorial automorphisms are those permutations of the vertices that induce an automorphism of the face lattice. For this property It is necessary and sufficient that they map facets to facets.

As an example we consider the input file `pentagon.in`:

```
amb_space 2
vertices 5
0 0
1 0
1 1
0.5 1.5
0 1
CombinatorialAutomorphisms
```



This is a polygon with 5 vertices. Result (shortened):

```
combinatorial automorphism group of order 10
Integrality not known
************************************************************************
2 permutations of 5 vertices of polyhedron

Perm 1: 1 3 2 5 4
Perm 2: 2 1 5 4 3

Cycle decompositions

Perm 1: (2 3) (4 5) --
Perm 2: (1 2) (3 5) --

1 orbits of vertices of polyhedron

Orbit 1 , length 5:  1 2 3 4 5
...
```

Clearly, every combinatorial automorphism is determined by the values of the two vertices of an edge, and we can freely choose the vertices of any of the five edges as values. So the combinatorial automorphisms group has order 10, and is in fact the dihedral group of this order. (All other automorphism groups of this pentagon have order 2.)

7.22.5. Ambient automorphisms

Roughly speaking, the ambient automorphisms are those permutations of the coordinates of the ambient space that permute the input vectors. They rae always defined for generator input and for input of inequalities (without an restriction of the lattice or subspace). These automorphisms are always integral and euclidean, but very often they are only a very small subgroup of the group of all integral/algebraic or euclidean automorphisms . The option for them is

`AmbientAutomorphisms`

As an example let us take the linear order polytope for S_6. If we run

```
./normaliz -c  example/lo6 -i --AmbientAutomorphisms
```

then the files `lo6.aut` starts with

```
Ambient(from generators) automorphism group of order 2 (possibly only approximation)
Automorphisms are integral
********************************************************************************
1 permutations of 720 input generators
...
```

The linear order polytope has 10080 integral automorphisms.

Note that permutations and orbits cannot be computed for factes if the input is by generators or for extreme rays if it is by inequalities since they are simply not known. However, Normaliz prints these data for the coordinates. In the cae of `lo6`

```
********************************************************************************
1 permutations of 16 Coordinates

Perm 1: 15 14 12 9 5 13 11 8 4 10 7 3 6 2 1 16

Cycle decompositions

Perm 1: (1 15) (2 14) (3 12) (4 9) (6 13) (7 11) --

10 orbits of Coordinates

Orbit 1 , length 2:  1 15
...
Orbit 10 , length 1:  16
```

Since the input vectors are not necessarily printed verbatim in the output file, they appear at the end of the `aut` file:

```
input generators

 1:  1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
 2:  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
...
```

7.22.6. Automorphisms from input

For the computation of the input automorphisms Notrmaliz applies the initial coordinate transformations to the input vectors and then computes their permutations that are given by rational (or algebraic) maps. The option is

`InputAutomorphisms`

With

```
./normaliz -c  example/lo6 -i --InputAutomorphisms
```

we get indeed the full automorphism group. The permutations of the facets are not computed:

```
Input(from generators) automorphism group of order 10080 (possibly only approximation)
Automorphisms are integral
**********************************************************************
3 permutations of 720 input generators
...
```

as you can check in the `aut` file. As for ambient automorphisms the input vectors are listed at the end.

Note: After the initial coordinate transformations, Normaliz reaches

(1) a full-dimensional primal cone if the input is by generators, or

(2) a full-dimensional dual cone if the input is by inequalities,

but not more. The passage to the full-dimensional pointed primal (or dual) cone is not possible at this point. Therefore the automorphisms computed from raw input do in general not map bijectively to automorphisms of the pointed full-dimensional quotient (or subcone).

Furthermore, in the inhomogeneous case it must be taken into account that Normaliz considers the inequality that makes the homogenizing variable nonnegative as part of the input. This is sometimes necessary to reach a full-dimensional dual cone after the initial coordinate transformation.

What has just been said is illustrated by `halfspace3inhom-input.in`:

```
amb_space 3
inhom_inequalities 3
1 0 0 1
0 0 1 0
0 0 -1 0
InputAutomorphisms
```

We expect an automorphism exchanging the second and the third inequality, and we get it:

```
Input(from inequalities) automorphism group of order 2 (possibly only approximation)
Integrality not known
**********************************************************************
1 permutations of 3 input inequalities

Perm 1: 1 3 2

Cycle decompositions

Perm 1: (2 3) --

2 orbits of input inequalities

Orbit 1 , length 1:  1
Orbit 2 , length 2:  2 3
```

```
************************************************************************
input inequalities

1:  1 0  0 1
2:  0 0  1 0
3:  0 0 -1 0
```

Compute the `Automorphisms` for this example!

7.22.7. Monoid automorphisms

With the computation goal `Automorphisms` Normaliz computes the automorphisms group of an affine monoisd. These the bijective additive maps from the monoid to itself, and are represen5ted by permutations of the Hilbert basis. An example (`monoid_auto.in`)

```
amb_space 2
monoid 4
2 0
1 1
0 2
0 2
grading
1 1
Automorphisms
```

The doubling of the last generator is non-intrinsic for the monoid structure and for the computation of the automorphism group we need intrinsic data:

```
3 Hilbert basis elements:
0 2
1 1
2 0
```

The automorphism group is

```
Monoid automorphism group of order 2 (possibly approximation if very large)
Automorphisms are integral
************************************************************************
1 permutations of 3 Hilbert basis elements

Perm 1: 3 2 1

Cycle decompositions

Perm 1: (1 3) --

2 orbits of Hilbert basis elements
```

```
Orbit 1 , length 2:  1 3
Orbit 2 , length 1:  2

********************************************************************
Hilbert basis elements

1:  0 2
2:  1 1
3:  2 0
```

Try `InputAutomorphism` and `AmbientAutomorphism` for the same input.

7.23. Precomputed data

The input of precomputed data can be useful if their computation takes long and they can be used again in subsequent computations. Normaliz takes their correctness for granted since there is no way of checking it without recomputation. Nevertheless some consistency checks are done.

We see the main use of precomputed data in interactive access when data had been stored from previous runs and can be made available again. These data allow the reconstruction of a cone (and lattice) and its subsequent modification via `modifyCone` without starting from scratch in the convex hull computation or vertex enumeration.

A file for future input of precomputed data can be asked for by the cone property `WritePreComp`. See Section 9.4.

7.23.1. Precomputed cones and coordinate transformations

Precomputed input of this type is given by the homogeneous input types `extreme_rays` and `support_hyperplanes`. (There is a third type `hilbert_basis_rec_cone`; see Section 7.23.3.) They can only be used *together*. Moreover, only the following types are allowed with them:

```
grading, dehomogenization, generated_lattice, maximal_subspace
```

This implies that data from inhomogeneous computations must be homogenized and then dehomogenized with explicit `dehomogenization` (see Section 7.23.2). For algebraic polyhedra `generated_lattice` represents a subspace without lattice structure.

Note that support hyperplanes and/or extreme rays do in general not define the object that Normaliz computes: the final pointed object of the computation lives in a subquotient U/W where U is a subspace (or sublattice) of the ambient space V and W is a subspace of U. Internally, this information is contained in two coordinate transformations. It is restored via

(1) `generated_lattice` for U if $U \neq V$,
(2) `maximal_subspace` for W if $W \neq 0$.

As an example we consider the input file `tame.in` which has transparent arithmetic:

```
amb_space 4
cone 1
1 0 0 0
subspace 1
0 1 0 0
congruences 1
1 0 0 0 2
```

In the output file `tame.out` we find

```
1 extreme rays:
2 0 0 0

1 basis elements of maximal subspace:
0 1 0 0

1 support hyperplanes:
1 0 0 0

...

2 basis elements of generated  lattice:
2 0 0 0
0 1 0 0
```

This information is transferred to `tame_prec.in` as

```
amb_space 4
extreme_rays 1
2 0 0 0
maximal_subspace 1
0 1 0 0
support_hyperplanes 1
1 0 0 0
generated_lattice 2
2 0 0 0
0 1 0 0
```

Running it reproduces the same output.

7.23.2. An inhomogeneous example

We use the input file `InhomIneq.in` already discussed in Section 2.9:

```
amb_space 2
constraints 3
```

```
0 1 >= -1/2
0 1 <=  3/2
-1 1 <=  3/2
grading
unit_vector 1
```

In the output file we find

```
dehomogenization:
0 0 1

grading:
1 0 0

...

2 vertices of polyhedron:
-4 -1 2
0  3 2

1 extreme rays of recession cone:
1 0 0

3 support hyperplanes of polyhedron (homogenized):
0 -2 3
0  2 1
2 -2 3
```

The coordinate transformations are trivial in this case. The translation into an input file with precomputed data is `InhomIneq_prec.in`:

```
amb_space 3
extreme_rays 3
-4 -1 2
0  3 2
1  0 0
support_hyperplanes 3
0 -2 3
0  2 1
2 -2 3
grading
1 0 0
dehomogenization
0 0 1
```

The vectors from the output can be copied. But there are two points to note:

(1) The change of `amb_space` from 2 to 3.

(2) The `extreme_rays` unite the vertices of the polyhedron and the extreme rays of the recession cone.

7.23.3. Precomputed Hilbert basis of the recession cone

In applications one may want to compute several polyhedra with the same recession cone. In these cases it is useful to add the Hilbert basis of the recession cone to the input. An example is `small_inhom_hbrc.in`:

```
amb_space 6
cone 190
6 0 7 0 10 1
...
vertices 4
0 0 0 0 0 0 1
1 2 3 4 5 6 2
-1 3 9 8 7 1 3
0 2 4 5 8 10 7
hilbert_basis_rec_cone 34591
0  0  0  1  6 1 0
0  0  0  1  7 1 0
...
```

As in the other cases with precomputed data, Normaliz must believe you and the precomputed Hilbert basis of the recession cone does not define the latter.

It requires inhomogeneous input. Note that it can only be used in the primal algorithm. In the dual algorithm it is useless and therefore ignored.

7.24. Singular locus

Certain data of affine monoid algebras are accedssuble by purely combinatorial methods. In particular this is true for the singular locus. (See SEction **??**.) It is computed by

`SingularLocus`

For `monoid.in` (see Section 3.1.1) the output file contains

```
codim singular locus = 1
```

and the file with suffix

`sng` contains the singular locus.

The format is the same as that for the face lattice:

```
2
5

00010 1
```

```
00001 1
```

The last integer in each row is the codimension of a minimal singular prime ideal and the face opposite to it is the intersection of the support hyperplanes with entry 1 in the 0-1-vector. The codimension 1 is not surprising for a nonnormal monoid.

If we are only interested in the codimenasion, we ask for

`CodimSingularLocus` computing the codimension of the singular locus

or only

`IsSerreR1` checking the Serre property (R_1)

which means that the codimension of the singular locus is ≥ 2. If this is all you want to know, then use `IsSerreR1` since it avoids the computation of the face lattice.

7.25. Packed format in the output of binomials

Output files of binomials are often very sparse, i.e., contain many entries 0. Reading them by computer algebra systems with a file interface like Singular or Macaulay2 can therefore be time consuming. To speed reading up, we have introduced a packed format for binomial output files.

`BinomialsPacked` activates the packed format for binimial output files.

For `representations.in` (discussed in Swction 3.1.2) it turns `representations.rep` into

```
-4
8
4 1 -1 3 -1 4 -1 5 1
3 3 -1 4 -2 6 1
4 1 -1 3 -2 4 -3 8 1
4 1 -2 2 1 3 -3 4 -2
```

The negative entry in the first line indicates the packed format. Its absolute value is the number of rows. Each row starts with the number vof nonzero entries, and each such entry is given by a pair (column, value).

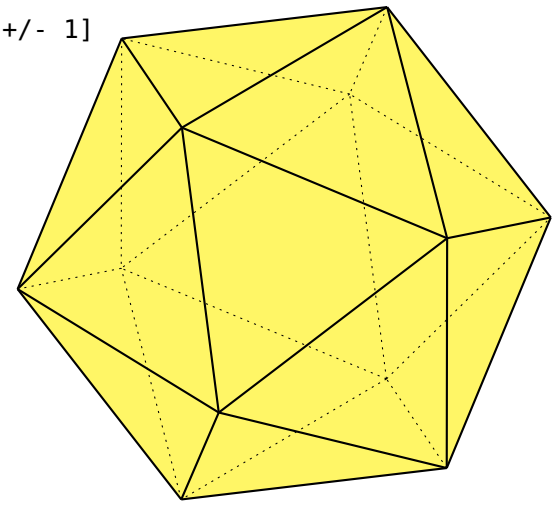
8. Algebraic polyhedra

Normaliz can use coefficients from real algebraic extensions of $\mathbb{Q}$. It is clear that the computations are then restricted to those that do not depend on finite generation of monoids. Whether algebraic coordinates are needed, is decided when Normaliz reads the input file and checks whether it defines an algebraic extension of $\mathbb{Q}$ embedded into $\mathbb{R}$.

8.1. An example

The icosahedron, one of the platonic solids, needs $\sqrt{5}$ for its coordinates. Via its vertices it can be defined as follows (`icosahedron-v.in`, picture by J.-Ph. Labbé):

```
amb_space 3
number_field min_poly (a^2 - 5) embedding [2 +/- 1]
vertices 12
0 2 (a + 1) 4
0 -2 (a + 1) 4
2 (a + 1) 0 4
...
(-a - 1) 0 -2 4
Volume
ModuleGenerators
FVector
EuclideanAutomorphisms
```



The second line specifies the extension $\mathbb{Q}[\sqrt{5}]$ of $\mathbb{Q}$ over which we want to define the icosahedron. In addition to the minimal polynomial (`min_poly` or `minpoly`)we have to give an interval from which the zero of the polynomial is to be picked. The square brackets are mandatory. There must be a *single* zero in that interval. The name of the root can be any single letter except `x` or `e`. The number field specification must follow `amb_space`. Otherwise Normaliz believes that you want to work over $\mathbb{Z}$.

Note that the entries of the input file that contain `a` must be enclosed in round brackets. You can enter any $\mathbb{Q}$-linear combination of powers of `a`. We allow $*$ between the coefficient and the power of `a`, but it need not appear. The character `^` indicates the exponent. It is mandatory. So

```
(a^3-2*a^2  +   4a-1/2)
(a+a-2a-10 + 10*a^0)
```

are legal numbers in the input. Instead of the delimiters `(...)` one can also use `"` and `'` on both sides so that

```
"a^3-2*a^2  +   4a-1/2"
```

```
'a+a-2a-10 + 10*a^0'
```

are also legal in matrices. However, in order to stick to standard conventions in mathematical notation, one must use `(...)` in symbolic constraints.

The result of the computation by `normaliz -c ../example/icosahedron-v` starts

```
Real embedded number field:
min_poly (a^2 - 5) embedding [2.23606797749978969...1835961152572 +/- 5.14e-54]
```

It indicates that the precision to which the root had to be computed in order to decide all the inequalities that have come up in the computation and to compute floating point approximations. Then we go on as usual:

```
1 lattice points in polytope
12 vertices of polyhedron
0 extreme rays of recession cone
20 support hyperplanes of polyhedron (homogenized)

f-vector:
1 12 30 20 1

embedding dimension = 4
affine dimension of the polyhedron = 3 (maximal)
rank of recession cone = 0 (polyhedron is polytope)

size of triangulation   = 18
resulting sum of |det|s = (5/2*a+15/2 ~ 13.090170)

dehomogenization:
0 0 0 1


volume (lattice normalized) = (5/2*a+15/2 ~ 13.090170)
volume (Euclidean) = 2.18169499062

Euclidean automorphism group has order 120
```

From the vertices below you can compute the radius of the sphere in which the icosahedron is inscribed and check that it is < 1. So no surprise:

```
1 lattice points in polytope:
0 0 0 1

12 vertices of polyhedron:
(-1/4*a-1/4 ~ -0.809017)                0       (-1/2 ~ -0.500000) 1
(-1/4*a-1/4 ~ -0.809017)                0         (1/2 ~ 0.500000) 1
...
(1/4*a+1/4 ~ 0.809017)                  0         (1/2 ~ 0.500000) 1
```

```
0 extreme rays of recession cone:

20 support hyperplanes of polyhedron (homogenized):

(-a+1 ~ -1.236068) (-2*a+4 ~ -0.472136)                  0 1
(-a+1 ~ -1.236068)   (2*a-4 ~ 0.472136)                  0 1
...
(a-1 ~ 1.2361)        (2*a-4 ~ 0.47214)                  0 1
```

Now every nonintegral number appears in round brackets together with its approximation as a decimal fraction.

The data of the integer hull cone are printed into a separate file as usual.

The order of the automorphism group of this regular polyhedron is exactly what we learn in geometry.

The matrices in the (optional) output file(s) can be used as input; see `perm7_d2_dual.in`. The input routine skips all characters from ~ when it reads a number.

For an example with precomputed data see `icosahedron_prec.in`.

8.2. Input

The following input types are NOT allowed for algebraic polytopes:

```
lattice           strict_inequalities     strict_signs   open_facets
cone_and_lattice  inhom_congruences       lattice_ideal  offset
congruences       hilbert_basis_rec_cone  rees_algebra   rational_lattice
rational_offset
```

The only other restriction is that decimal fractions and floating point numbers are not allowed in the input file. The input format for field coefficients is explained in the example above.

It may seem contradictory, but `saturation` are allowed. It must be interpreted as a generating set for a subspace that is intersected with all the objects defined by other input items.

With coordinates in number fields, Normaliz does not look for an implicit grading, but it can use an explicit grading for lattice point or volume computations in the homogeneous case. `NoGradingDenom` is set automatically. For inhomogeneous input a grading makes no sense in the number field case and is therefore forbidden.

While `polynomial_equations` and `polynomial_inequalities` are allowed for algebraic polytopes, the coefficients of the polynomials must be rational: This may change in the future.

8.3. Computations

The only (main) computation goals and algorithmic variants allowed are:

SupportHyperplanes	Sublattice	LatticePoints	LatticePointTriangulation
NumberLatticePoints	IntegerHull	VerticesFloat	TriangulationSize
Triangulation	ProjectCone	KeepOrder	ConeDecomposition
BottomDecomposition	SuppHypsFloat	NoBottomDec	TriangulationDetSum
GradingIsPositive	DefaultMode	IsPointed	EuclideanAutomorphisms
FVector	FaceLattice	Automorphisms	CombinatorialAutomorphisms
Incidence	Deg1Elements	IsEmptySemiOpen	AllGeneratorsTriangulation
SingleLatticePoint			

It may seem paradoxical that `Sublattice` appears here. As in the true lattice case, the `Sublattice Representation` is the coordinate transformation used by Normaliz. Over a field F there is no need for the annihilator c, and one simply has a pair of linear maps $F^r \to F^d \to F^r$ whose composition is the identity of F^r. Of course, congruences and external index make no sense anymore.

`Deg1Elements`, `LatticePoints` and `IntegerHull` are restricted to (bounded) polytopes since polyhedra in general lack the necessary finiteness properties. The lattice of reference is the full integral lattice.

`Automorphisms` is interpreted as *algebraic* automorphisms. They are defined in the same way as rational automorphisms of rational polytopes. One has only to replace the field of rational numbers by the number field defined for the polytope. `EuclideaAutomorphisms` and `CombinatorialAutomorphisms` have the usual meaning.

`Volume` is restricted to full-dimensional polytopes. In the homogeneous case the grading must have integer coprime coefficients.

The only algorithmic variants that appear concern the bottom decomposition. Implicit or explicit `DefaultMode` is interpreted as `SupportHyperplanes`.

Volumes are computed by triangulation and lattice points by project-and-lift.

For the control of computations and communication with interfaces the following are allowed:

Generators	ExtremeRays	VerticesOfPolyhedron	MaximalSubspace
RecessionRank	AffineDim	Rank	EmbeddingDim
IsInhomogeneous	RenfVolume	EuclideanVolume	ModuleGenerators
Dehomogenization	NoGradingDenom	Equations	

9. Optional output files

In addition nto the output file with suffix `out` several computation goals write their results into extra files. But these are *not* optional. They are explained together with the pertaining computationn goals, and Section 9.5 gives an overview.

The main purpose of the optional output files is to provide an interface to Normaliz by files that can more easily be parsed than the main output file.

When one of the options `Files, -f` or `AllFiles, -a` is activated, Normaliz writes additional optional output files whose names are of type `<project>.<type>`. Moreover one can select the optional output files individually on the command line. Most of these files contain matrices in a simple format:

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

where each row has `<n>` entries. Exceptions are the files with suffixes `cst`, `inv`, `esp`.

Note that the files are only written if they would contain at least one row.

As pointed out in Section 6.6, the optional output files for the integer hull are the same as for the original computation, as far as their content has been computed.

9.1. The homogeneous case

The option `-f` makes Normaliz write the following files:

`gen` contains the Hilbert basis. If you want to use this file as an input file and reproduce the computation results, then you must make it a matrix of type `cone_and_lattice` (and add the dehomogenization in the inhomogeneous case).

`cst` contains the constraints defining the cone and the lattice in the same format as they would appear in the input: matrices of types *constraints* following each other. Each matrix is concluded by the type of the constraints. Empty matrices are indicated by 0 as the number of rows. Therefore there will always be at least 3 matrices.

If a grading is defined, it will be appended. Therefore this file (with suffix `in`) as input for Normaliz will reproduce the Hilbert basis and all the other data computed, at least in principle.

In the case of number field coordinates this file must be transformed from Normaliz 2 input format to Normaliz 3 format by hand before it can be used for input.

`inv` contains all the information from the file `out` that is not contained in any of the other files.

If `-a` is activated, then the following files are written *additionally:*

`ext` contains the extreme rays of the cone.

ht1 contains the degree 1 elements of the Hilbert basis if a grading is defined.

egn,esp These contain the Hilbert basis and support hyperplanes in the coordinates with respect to a basis of $\mathbb{E}$. `esp` contains the grading and the dehomogenization in the coordinates of $\mathbb{E}$. Note that no equations for $\mathbb{C} \cap \mathbb{E}$ or congruences for $\mathbb{E}$ are necessary.

lat contains the basis of the lattice $\mathbb{E}$.

mod contains the module generators of the integral closure modulo the original monoid.

msp contains the basis of the maximal subspace.

In order to select one or more of these files individually, add an option of type `--<suffix>` to the command line where `<suffix>` can take the values

```
gen, cst, inv, ext, ht1, egn, esp, lat, mod, msp, typ
```

The type `typ` is not contained in `Files` or `AllFiles` since it can be extremely large. It is of the matrix format described above. It is the product of the matrices corresponding to `egn` and the transpose of `esp`. In other words, the linear forms representing the support hyperplanes of the cone C are evaluated on the Hilbert basis. The resulting matrix, with the generators corresponding to the rows and the support hyperplanes corresponding to the columns, is written to this file.

The suffix `typ` is motivated by the fact that the matrix in this file depends only on the isomorphism type of monoid generated by the Hilbert basis (up to row and column permutations). In the language of [11] it contains the *standard embedding*.

Note: the explicit choice of an optional output file does *not* imply a computation goal. Output files that would contain unknown data are simply not written without a warning or error message.

9.2. Modifications in the inhomogeneous case

The optional output files are a subset of those that can be produced in the homogeneous case. The main difference is that the generators of the solution module and the Hilbert basis of the recession monoid appear together in the file `gen`. They can be distinguished by evaluating the dehomogenization on them (simply the last component with inhomogeneous input), and the same applies to the vertices of the polyhedron and extreme rays of the recession cone. The file `cst` contains the constraints defining the polyhedron and the recession cone in conjunction with the dehomogenization, which is also contained in the `cst` file, following the constraints.

In the file with suffix `ext` the vertices of polyhedron are listed first, followed by the extreme rays of the recession cone.

With `-a` the files `egn` and `esp` are produced. These files contain `gen` and the support hyperplanes of the homogenized cone in the coordinates of $\mathbb{E}$, as well as the dehomogenization.

9.3. Algebraic polyhedra

Some entries in the inv file are listed as `integer`, even if they are not integer numbers. But all entries make sense as elements of the algebraic number field.

9.4. Precomputed data for future input

One can generate a file with the data needed for an input file with precomputed data of the cirrent cone using the cone property

`WritePreComp`

The suffix is `precomp.in`. It contains the data that can (and must) go into a file redefining the present cone (except `hilbert_basis_rec_cone`).

9.5. Overview: Output files forced by computation goals

`tri` contains the triangulation.
`tgn` reference generators for the triangulation.
`aut` contains the automorphism group.
`dec` contains the Stanley decomposition.
`fac` contains the face lattice.
`inc` contains the (dual) incidence matrix.
`mrk` contains the Markov basis.
`grb` contains the Gröbner basis.
`rep` contains the representations of the reducible generators of a monoid by the irreducible ones.
`ogn` contains the refernce generators for Markov bases, Gröbner bases and represntations.
`sng` contains the singular locus.
`proj.out` contains the projected cone.
`inthull.out` contains the integer hull.
`symm.out` contains the symmetrized cone.

10. Performance

10.1. Parallelization

The executables of Normaliz have been compiled for parallelization on shared memory systems with OpenMP. Parallelization reduces the "real" time of the computations considerably, even on relatively small systems. However, one should not underestimate the administrative overhead involved.

- It is not a good idea to use parallelization for very small problems.

- On multi-user systems with many processors it may be wise to limit the number of threads for Normaliz somewhat below the maximum number of cores.

By default, Normaliz limits the number of threads to 8. One can override this limit by the Normaliz option `-x` (see Section 6.3).

Another way to set an upper limit to the number of threads is via the environment variable `OMP_NUM_THREADS`:

```
export OMP_NUM_THREADS=<T>      (Linux/Mac)
```

or

```
set OMP_NUM_THREADS=<T>      (Windows)
```

where `<T>` stands for the maximum number of threads accessible to Normaliz. For example, we often use

```
export OMP_NUM_THREADS=20
```

on a multi-user system system with 24 cores.

Limiting the number of threads to 1 forces a strictly serial execution of Normaliz.

The paper [15] contains extensive data on the effect of parallelization. On the whole Normaliz scales very well. However, the dual algorithm often performs best with mild parallelization, say with 4 or 6 threads.

10.2. Running large computations

Note: This section discusses computations in primal mode, and reflects the state of Normaliz discussed in [15]. Especially the computation of lattice points in polytopes and of volumes have been implemented in oher algorithms that are often much faster. However, for Hilbert bases and Hilbert series only refinements of the primal mode have been realized.

Normaliz can cope with very large examples, but it is usually difficult to decide a priori whether an example is very large, but nevertheless doable, or simply impossible. Therefore some exploration makes sense. The following applies to the primal algorithm.

See [15] for some very large computations. The following hints reflect the authors' experience with them.

(1) Run Normaliz with the option `-cs` and pay attention to the terminal output. The number of extreme rays, but also the numbers of support hyperplanes of the intermediate cones are useful data.

(2) In many cases the most critical size parameter for the primal algorithm is the number of simplicial cones in the triangulation. It makes sense to determine it as the next step. Even with the fastest potential evaluation (option `-v` or `TriangulationDetSum`), finding the triangulation

takes less time, say by a factor between 3 and 10. Thus it makes sense to run the example with `-t` in order to explore the size.

As you can see from [15], Normaliz has successfully evaluated triangulations of size $\approx 5 \cdot 10^{11}$ in dimension 24.

(3) Another critical parameter are the determinants of the generator matrices of the simplicial cones. To get some feeling for their sizes, one can restrict the input to a subset (of the extreme rays computed in (1)) and use the option `-v` or the computation goal `TriangulationDetSum` if there is no grading.

The output file contains the number of simplicial cones as well as the sum of the absolute values of the determinants. The latter is the number of vectors to be processed by Normaliz in triangulation based calculations.

The number includes the zero vector for every simplicial cone in the triangulation. The zero vector does not enter the Hilbert basis calculation, but cannot be neglected for the Hilbert series.

Normaliz has mastered calculations with $> 10^{15}$ vectors.

(4) If the triangulation is small, we can add the option `-T` in order to actually see the triangulation in a file. Then the individual determinants become visible.

(5) If a cone is defined by inequalities and/or equations consider the dual mode for Hilbert basis calculation, even if you also want the Hilbert series.

(6) The size of the triangulation and the size of the determinants are *not* dangerous for memory by themselves (unless `-T` or `-y` are set). Critical magnitudes can be the number of support hyperplanes, Hilbert basis candidates, or degree 1 elements.

11. Distribution and installation

11.1. Docker image

The easiest and absolutely hassle free access to Normaliz is via its Docker image. To run it, you must first install Docker on your system. This is easy on up-to-date versions of the three major platforms. After installation you can issue the command

```
docker run -ti normaliz/normaliz
```

You may have to prefix it with `sudo`. This will download the Docker image if it is not yet present and open a Docker container. As a result you will get a Linux terminal. Normaliz is installed in the standard location `/usr/local`. Moreover, the source is contained in the subdirectory `Normaliz/source` of the home directory. (Your username is `norm`.) In the Docker container, `Normaliz` is the Normaliz directory (independently of the version number).

Try

```
normaliz -c Normaliz/example/small
```

as a first test.

Of course, you want to make your data available to Normaliz in the container. Here is an example:

```
docker run -it -v /home/winfried/my_normaliz:/home/norm/example  normaliz/normaliz
```

Here `/home/winfried/my_normaliz` is the (absolute!) path to the directory that I want to mount into the Docker container and `/home/norm/example` is the (absolute!) path to the location in the container where it should be mounted.

The command above downloads the image labeled "latest". There are also images on Dockerhub with version numbers. You can slo access them adding the sufix `:<version>` to `normaliz/normaliz`.

The Docker image contains a full installation including PyNormaliz.

11.2. Binary release

We provide binary releases for Windows, Linux and Mac. Follow the instructions in

`https://normaliz.uos.de/download/`.

They guide you to our GitHub repository

`https://github.com/Normaliz/Normaliz/releases`.

Download the archive file corresponding to your system `normaliz-3.10.1_<systemname>.zip` in a directory of your choice and unzip it. This process will create the Normaliz directory and store the Normaliz executable in it. In case you want to run Normaliz from the command line

or use it from other systems, you may have to copy the executables to a directory in the search path for executables or update your search path.

The From version 3.9.3 on, MS Windows executable is compiled with all optional packages.

Note:

1. The Linux binary `normaliz` and the MS Windows binary `normaliz.exe` are fully static executables.
2. The Mac OS and the MS Windows binaries cannot be statically linked in the absolute sense. The MS Windows binary depends only on system DLLs. The Mac OS binary depends only on Mac OS system libraries and `libomp.dylib` (contained in the zip file) which makes parallelization possible. *This dynamic library must be kept in the same directory as the binary.*

Unzipping creates the following files and subdirectories in the Normaliz directory:

- In the Normaliz directory you should find `jNormaliz.jar`, and the binary files as indicated above. Furthermore COPYING.
- The subdirectory `doc` contains the file you are reading, the LaTeXfiles from mwhich it is created and the compact overwiew `NmzShortRef.pdf` for fast access to the Normaliz key words.
- In the subdirectory `example` there are the input files for some examples. It contains all named input files of examples of this manual.
- The subdirectory `Singular` contains the SINGULAR library `normaliz.lib` and a PDF file with documentation.
- The subdirectory `Macaulay2` contains the MACAULAY2 package `Normaliz.m2`.
- The subdirectory `lib` contains libraries for jNormaliz.

11.3. Source package

In order to build Normaliz yourself, navigate to our GitHub repository

`https://github.com/Normaliz/Normaliz/releases`.

and download the source package `normaliz-3.10.1.zip` (also available as `.tar.gz`) contains the source files, installation scripts, examples, documentation, the test suite and PyNormaliz.

Then unzip the downloaded file in a directory of your choice and expand it. (If you have installed a binary package, choose the same directory.) This process will create a directory `normaliz-3.10.1` and several subdirectories in it.

If you build Normaliz yourself, the build process will create further subdirectories `build`, `nmz_opt_lib` and `local` (with the default settings).

11.4. Conda

The platform independent package manager Conda provides executables for all three operating systems. See

```
https://github.com/conda-forge/normaliz-feedstock
```

In addition to the binaries you get the files that are usually installed: header files and libraries.

11.5. Cloning the GitHub repository

Another way to download the Normaliz source is cloning the repository from GitHub by

```
git clone https://github.com/Normaliz/Normaliz.git
```

The Normaliz directory is then called `Normaliz`. You may need

```
sudo apt-get install autoconf libtool
```

if you want to build Normaliz by autotools. To this end change to the Normaliz directory and run

```
./bootstrap.sh
```

After this step you can follow the instruction in the next section.

Note that he GitHub repository Normaliz/Normaliz does not contain PyNormaliz. You can clone them from the repository Normaliz/PyNormaliz.

12. Building Normaliz yourself

We recommend building Normaliz through the install scripts described below. They use the `autotools` scripts have been written by Matthias Köppe. The Normaliz team thanks him cordially for his generous help.

If you don't want to use the Normaliz install scripts, you can of course take the usual configure-make-make install path. The dependencies of Normaliz on external packages are listed in `INSTALL`.

12.1. Prerequisites

We require a compiler supporting C++14:

- GNU g++ 5,
- clang++ 3.4,
- Intel icpc 16.0

The mentioned compilers are also able to handle OpenMP 3.0, with the exception of clang++, there the first OpenMP support was introduced in 3.7.

For compiling Normaliz the following libraries are needed:

- GMP including the C++ wrapper (libgmpxx and libgmp)

We will only discuss how to build Normaliz with the install scripts in the distribution. See the file `INSTALL` for additional information.

Any optional package that you want to use, must be installed before the compilation of Normaliz, independently of the method used for building Normaliz. The installation scripts mentioned below make and use directories within the Normaliz directory.

12.1.1. Linux

The standard compiler choice on Linux is `g++`. We do no not recommend `clang++` since its support for OpenMP is not as comprehensive as that of `g++`.

On Ubuntu we suggest

```
sudo apt-get install tar g++ libgmp-dev wget make libboost-all-dev
```

for the basic installation of the required libraries (plus compiler).

12.1.2. Mac OS X

Currently Apple does not supply a compiler which supports OpenMP. The install scripts discussed below *require LLVM 3.9 or newer from Homebrew.* See `https://brew.sh/` from where you can also download GMP:

```
brew install gmp llvm wget boost automake
```

It may be necessary to replace `install` by `reinstall` since the Xcode compiler may be newer that the one from Homebrew.

You also need to download and install the Xcode Command Line Tools from the AppStore:

```
xcode-select --install
```

12.2. Normaliz at a stroke

Navigate to the Normaliz directory. The command

```
./install_normaliz_with_eantic.sh
```

installs the all packages that are needed for the computation of rational and algebraic polyhedra (including CoCoALib and Flint) and does the full compilation.

If you don't want algebraic polyhedra, call

```
./install_normaliz_with_opt.sh
```

It downloads CoCoALib, Flint and nauty and compiles Normaliz.

The sources of the optional packages are downloaded to the subdirectory `nmz_opt_lib` of the Normaliz directory. They are installed in the subdirectory `local` (imitating `/usr/local`) where they exist in static and dynamic versions (except CoCoALib and nauty that can only be built statically).

If you don't want the optional packages or if you have them properly installed,

```
./install_normaliz.sh
```

compiles Normaliz, using the optional packages that it can find.

The library `libnormaliz` is compiled statically and shared. It is installed in `local` as well.

The binary `normaliz` is stored in `local/bin`, but it is also copied to the Normaliz directory.By default, it is statically linked on Linux. On MacOS the binary is compiled with shared libraries.

Remarks:

(1) If you want a global installation (and have the rights to do it), you can ask for

```
sudo cp -r local /usr
```

at the end.

(2) Another way to a global installation (or to an installation in a place of your choice) is to use

```
export NMZ_PREFIX=<your choice>
./install_normaliz_...
```

For the typical choice `/usr/local` you need superuser rights (as in (1)). Note that `NMZ_PREFIX` must be an absolute path name.

(3) As already said, the scripts compile a fully static binary under Linux. You can choose a dynamically linked binary by

```
export NMZ_SHARED=yes
./install_normaliz_...
```

`NMZ_SHARED` is set automatically on Linux if a compiler from the clang family is used since a statically linked binary cannot be built by them (`libomp.a` is missing).

On MacOS there is no choice—the binary is dynamically linked. If you want a binary that is as static as possible, download the MacOS binary distribution from GitHub (see Section 11.2). Its production is a bit tricky.

(4) The install scripts can be further customized. Have a look at them or at `INSTALL`.

(5) Precise information on the versions of the optional packages that should be used with Normaliz 3.10.1 is contained in `INSTALL` as well.

(6) The install script creates a directory for VPATH builds,

`build`

It is *not* removed by the script so that you can use it for further `make` actions.

(7) To run the test suite, go to `build` and run `make check`. For more information on the test suite see `INSTALL`.

12.3. Packages for rational polyhedra

12.3.1. CoCoALib

Normaliz can be built without CoCoALib, which is however necessary for the computation of integrals and weighted Ehrhart series and, hence, for symmetrization. If you don't want to use the scripts in Section 12.2, but nevertheless want to compile Normaliz with CoCoALib, install CoCoALib first by navigating to the Normaliz directory and typing the command

```
install_scripts_opt/install_nmz_cocoa.sh
```

CoCoALib is downloaded and compiled as described above.

If you want to use a preinstalled version of CoCoALib: for parallelization it must be configured as

```
./configure --threadsafe-hack --no-boost
```

12.3.2. nauty

Normaliz can be built without nauty, which is however necessary for the computation of automorphism groups. If you don't want to use the scripts in Section 12.2, but nevertheless want to compile Normaliz with nauty, install nauty first by navigating to the Normaliz directory and typing the command

```
install_scripts_opt/install_nmz_nauty.sh
```

nauty is downloaded and compiled as described above.

You can or course use a preinstalled version of nauty.

We thank Brendan McKay for his help in the integration of nauty to Normaliz.

12.3.3. Hash library

for the computation of SHA256 hash values Normaliz uses by Stephan Brumme:

```
install_scripts_opt/install_nmz_hash-library.sh
```

12.3.4. Flint

Flint does not extend the functionality of Normaliz (for rational polytopes), and is therefore truly optional. However, the ultrafast polynomial arithmetic of Flint is very useful if quasipolynomials with large periods come up in the computation of Hilbert series or weighted Ehrhart

series. If you don't want to use the scripts in Section 12.2, but nevertheless want to compile Normaliz with Flint, install Flint (and its prerequisite MPFR) by navigating to the Normaliz directory and entering the commands

```
install_scripts_opt/install_nmz_mpfr.sh
install_scripts_opt/install_nmz_flint.sh
```

12.4. Packages for algebraic polyhedra

The basic classes for algebraic polyhedra are defined in the package

e-antic by V. Delecroix (`https://github.com/flatsurf/e-antic`).

In its turn it is based on

(1) Flint maintained by W. B. Hart (`https://www.flintlib.org/`),
(2) antic by W. B. Hart, (`https://github.com/wbhart/antic`),
(3) arb by F. Johansson (`https://arblib.org/`).

The dependence on antic and arb is only through e-antic.

For Normaliz it is useless to install any of them separately (except Flint, see above). Again, if you don't want to use the ready-made install scripts for Normaliz as a whole, you can install e-antic and its prerequisites separately by

```
install_scripts_opt/install_eantic_with_prerequisites.sh
```

12.5. MS Windows

We compile Normaliz for MS Windows 64 under MSYS2. See the last section of `INSTALL` for the details.

13. Copyright and how to cite

Normaliz 3.1 is free software licensed under the GNU General Public License, version 3. You can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

It is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with the program. If not, see `https://www.gnu.org/licenses/`.

Please refer to Normaliz in any publication for which it has been used: been used:

W. Bruns, B. Ichim, C. Söger and U. von der Ohe: Normaliz. Algorithms for rational cones and affine monoids. Available at `https://normaliz.uos.de`

The corresponding `\bibitem`:

```
\bibitem{Normaliz} W.~Bruns, B.~Ichim, C.~S\"oger and U.~von~der~Ohe:
Normaliz. Algorithms for rational cones and affine monoids.
Available at \url{https://normaliz.uos.de}.
```

A BibTeX entry:

```
@Misc{Normaliz,
author = {W. Bruns and B. Ichim, C. S\"oger and U. von der Ohe},
title = Normaliz. Algorithms for rational cones and affine monoids,
howpublished ={Available at \url{https://normaliz.uos.de}}
```

It is now customary to evaluate mathematicians by such data as numbers of publications, citations and impact factors. The data bases on which such dubious evaluations are based do not list mathematical software. Therefore we ask you to cite the article [15] in addition. This is very helpful for the younger members of the team.

A. Mathematical background and terminology

For a coherent and thorough treatment of the mathematical background we refer the reader to [11].

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

$$M = \{a_1 x_1 + \cdots + a_n x_n : a_1, \ldots, a_n \in \mathbb{Z}_+\}.$$

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 [15, 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 $e(N) = \operatorname{mrank}(N)e(M)$ 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 [11, Section 4.F], and especially [11, 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 $\operatorname{gp}(M)$ to the vector $\sigma(x)$ where σ is the collection of support forms $\sigma_1, \ldots, \sigma_s$ of M: $\operatorname{Cl}(M) = \mathbb{Z}^s/\sigma(\operatorname{gp}(M))$. Finding this quotient amounts to an application of the Smith normal form to the matrix of σ.

A.8. Affine monoid algebras and their defining ideals

In addition to [11], the reader may want to consult De Loera, Hmmecke nd Köppe [22] and Sturmfels [36] for the discussion of Markov and Gröbner bases in our context.

As soon as one wants to understand 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 [11].) Therefore we start from a field K and a K-subalgebra A of a Laurent polynomial ring $K[X_1^{\pm 1}, \ldots, X_n^{\pm 1}]$ 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, switching 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 such that most one entry v_i^+ and v_i^- is nonzero in computing I. So we restrict our use of the term "inomial" by assuming that both monmials in it 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 system 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 the Hilbert series, one even needs a Gröbner basis of I (unless A is normal).

Toric ideals are generalized by *lattice ideals* J; these are 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.

The computation of Markov bases (that in all algorithms we know is based on Gröbner bases) is often time consuming. Normaliz uses a reimplementation of the project-and-lift algorithm of Hemmecke and Malkin [29] for the computation of Markov bases, realized by them in 4ti2 [1]. The project-and-lift algorithm is also explained in [22].

A.9. Affine monoid algebras from binomial ideals

The typical starting point is an ideal $J \subset P = K[Y_1,\dots,Y_m]$ generated by binomials

$$Y^v - Y^w, \qquad v,w \in \mathbb{Z}_+^n.$$

In general the residue class ring P/J is not a monoid ring, let alone an affine monoid ring. To understand in which way J nevertheless defines an affine monoid algebra, we extend J to JQ where $Q = K[Y_1^{\pm 1},\dots,Y_m^{\pm 1}]$ is the Laurent polynomial extension of P.

The term ' ''lattice ideal" needs an explanation. In Q the monomial Y^w is a unit, so that

$$Y^v/Y^w - 1 \in JQ \quad \Longleftrightarrow \quad Y^v - Y^w \in JQ \cap P.$$

The quotients Y^v/Y^w generate a sublattice of the unit group of Q which can be identified with ZZ^m if one passes from a monomial to its exponent vector. The binomials Y^v/Y^w for which $Y^v/Y^w - 1 \in JQ$ form a sublattice L of the unit group. So the smallest lattice ideal containing J is $JQ \cap P$. Normaliz computes it from the input type `lattice_ideal` via the sublattice L of the unit group of the Laurent polynomial ring.

Let us now assume that J is a lattice ideal. Then P/J is a monoid ring, but not necessarily an affine monoid ring since $\mathbb{Z}^m/L$ need not be torsionfree. In order to get an affine monoid ring, we must increase by the preimage of the torsion subgroup of $\mathbb{Z}^m/L$ in $\mathbb{Z}^m$. In other words, L is replaced by its saturation $\overline{L}$ in ZZ^m. Computing the smallest toric ideal T defined by our binomials means to find $\overline{L}$ and the ideal in P generated by all binomials $X^v - X^w$ for which $v - w \in \overline{L}$.

A.10. Local properties of affine monoid algebras

Let R be a commutative Noetherian ring. By a "local property" $\mathscr{P}$ we mean a property that is described in terms of the localizations R_P running over all prime ideals P of R. For an extensive discussion of the following we refer the reader to [11, Chap. 4] (including the exercises).

Certain local properties of affine monoid algebras R depend only on the underlying monoid and can be tested by computations applied to the latter. The transition from general prime ideals P to their combinatorial counterparts proceeds in two steps. The first is the observation that the ideal P^* generated by all monomials in P is itself a prime ideal. For suitable $\mathscr{P}$, R_P satisfies $\mathscr{P}$ if and only if R_{P^*} has $\mathscr{P}$. The second step is the passage from $R_{\mathscr{P}^*}$ to the ring $R[S^{-1}]$ where S is the set of monoid elements outside P (or P^*). The crucial point now is that $R[S^{-1}]$ is again a monoid algebra and the underlying monoid is (in additive notation) $N[-S]$. The set set S is the intersection of N witha facet of the cone generated by N. To sum up: Determining the monomial prime ideals in R amounts to the computation of the face lattice, and certain propefrties $\mathscr{P}$ can be tested on the "ocalizations" of N.

Suitable properfties are regularity and normality. For regularity we have realized the computation of the singular locus, i.e., the set of prime ideals for which the corresponding localization is not regular. See Section 7.24.

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

F.1. 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`.

F.2. Lattice points via patching

The patching algorithm (Section 7.2.4) allows distributed computation. However, the implementation is still rudimentary since it uses only a static splitting which (in contrast to the volume computation) often does not efficiently use the number of parallel instances that are allowed by the high performance cluster.

We run a precomputation

```
./normaliz -c <project> --DistributedComp
```

The only purpose it to find the lowest level where splitting makes sense. (For small computations such a level may not exist.) The answer is written to the file

`<project>.split.data`

It is always a default choice:

```
1 <l> 1000
```

where <l> is the lowest level. Often it is necessary to edit this file by hand. The general format is

```
<n> <l1> <s1>  ... <ln> <sn>
```

where <n> is the number of split levels defined by <l1>,...<ln> in increasing order, and the numbers <s1>,...,<sn> list the number of splits as these levels. Example:

```
2 1 10 3 100
```

We split at level 1 into 10 parts, and then each of these parts is split at level 3 into 100 subparts. The total number of parts run in parallel is the product of <s1>,...,<sn>, 1000 in the example (and the default choice).

We use a job array defined by

```
#!/bin/sh
#SBATCH --job-name="tough"
#SBATCH --comment="tough"
#SBATCH --time=24:00:00
#SBATCH --ntasks=8
#SBATCH --threads-per-core=1
#SBATCH --mem=50000
#SBATCH --array=0-999

# each job will see a different ${SLURM_ARRAY_TASK_ID}
time ../normaliz -c tough --Split -X=${SLURM_ARRAY_TASK_ID}
```

for the input file tough.in. We need the option

Split

to indicate that this is a split computation, and

-X=<s>

gives the number <s>of the split to be run by the Normaliz instance identified by ${SLURM_ARRAY_TASK_ID}. Note that every split reads the input file, but it does not write a file with suffix out.

Each job writes the computed lattice points to

<project>.<s>.lat

where <s> is the number of the split given by -X=<s>. After the computation has been finished the lattice points are harvested by running Normaliz with the option

CollectLat

for example ./normaliz␣<project>␣--CollectLat. The total list of lattice points is written to

<project>.total.lat

in the standard format: number of rows, number of columns, the vectors.

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] M.A. Alekseyev, W. Bruns, S. Palcoux and F. V. Petrov, *lassification of modular data of integral modular fusion categories up to rank 11*. Preprint arXiv:2302.01613.

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

[5] 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/.

[6] 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.

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

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

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

[10] 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.

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

[12] 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.

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

[14] 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.

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

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

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

[18] 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.

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

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

[21] 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.

[22] 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.

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

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

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

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

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

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

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

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

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

[32] 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).

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

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

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

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