# Computing Gröbner fans to understand parameter geometry in mathematical biology

Daniel Le[1] and George Simmons[2]

[1]Department of Mathematics, Colby College, Waterville ME 04901
[2]Department of Mathematics, University of Michigan, Ann Arbor MI 48109

August 14, 2026

**Abstract**

A Gröbner fan is a polyhedral fan that encodes all possible Gröbner bases of a polynomial ideal under different orderings. Gröbner bases are important tools for understanding the geometric structure underlying the parameter relationships and dynamics of applied models in mathematical biology, however they can often only be interpreted when calculated using an optimal ordering. We implement an algorithm to compute Gröbner fans in Mathematica. The source code is publicly accessible in this GitHub repository.

## Contents

**1 Introduction** **2**
1.1 Motivations . . . 2
1.2 Preliminaries . . . 3

**2 Algorithmic Implementations** **5**
2.1 Algorithmic Overview and Pipeline Implementation . . . 5
2.1.1 Overview of the Mathematica Pipeline . . . 5
2.1.2 Implementation Status . . . 6
2.2 Saturation.wl . . . 6
2.2.1 Package Overview . . . 6
2.3 TermOrder.wl . . . 6
2.3.1 Package Overview . . . 6
2.3.2 Complexity . . . 6
2.4 Symmetry.wl . . . 6
2.4.1 Package Overview . . . 6
2.4.2 Complexity . . . 7
2.5 SymmetricComplex.wl . . . 7
2.5.1 Package Overview . . . 7
2.5.2 Complexity . . . 8
2.6 LpSolver.wl . . . 8
2.6.1 Package Overview . . . 8
2.6.2 Complexity . . . 9
2.7 LpSolverMathematica.wl . . . 9
2.7.1 Package Overview . . . 9
2.7.2 Complexity . . . 9
2.8 GroebnerEngine.wl . . . 10
2.8.1 Package Overview . . . 10
2.8.2 Complexity . . . 10
2.9 PolyhedralCone.wl . . . 11
2.9.1 Package Overview . . . 11
2.9.2 Complexity . . . 11

2.10 wallideal.wl . . . 11
2.10.1 Package Overview . . . 11
2.10.2 Complexity . . . 12
2.10.3 issues . . . 12
2.11 gfan_cpp_interface.wl . . . 12
2.11.1 Package Overview . . . 12

**3 Future Directions** **12**

**4 Acknowledgments** **12**

**A Package Specifications** **13**
A.1 Saturation.wl . . . 13
A.1.1 Ideal Operations . . . 13
A.2 TermOrder.wl . . . 13
A.2.1 Generic Order Operations . . . 13
A.2.2 Term Order Constructors . . . 13
A.3 Symmetry.wl . . . 13
A.3.1 Constructors & Group Algebra . . . 13
A.3.2 Group Growth & Generators . . . 14
A.3.3 Orbit & Stabilizer Operations . . . 14
A.3.4 Polyhedral & Geometric Utilities . . . 14
A.3.5 Vector & Polynomial Actions . . . 14
A.4 SymmetricComplex.wl . . . 14
A.4.1 Complex Constructors & Inspection . . . 14
A.4.2 Cone Operations . . . 14
A.4.3 Complex Modifications & Topological Properties . . . 15
A.4.4 Homology & Boundary Calculations . . . 15
A.5 LpSolver.wl . . . 15
A.5.1 Solver Registry & Configuration . . . 15
A.5.2 Polyhedral Feasibility & Ray Operations . . . 15
A.5.3 Conic & Dual Representation . . . 16
A.5.4 Linear Algebra Utilities . . . 16
A.6 LpSolverMathematica.wl . . . 16
A.6.1 Solver Registration . . . 16
A.6.2 Polyhedral & LP Engine Implementations . . . 16
A.6.3 Vector Preprocessing Utility . . . 16
A.7 GroebnerEngine.wl . . . 16
A.7.1 Engine Configuration & Setup . . . 16
A.7.2 Gröbner Basis & Polynomial Reduction . . . 17
A.8 PolyhedralCone.wl . . . 17
A.8.1 Constructor, Initializers & State Machine . . . 17
A.8.2 Basic Accessors & Geometric Indicators . . . 17
A.8.3 Containment & Membership Testing . . . 17
A.8.4 Relative Interior Points & Generators . . . 17
A.8.5 Polyhedral Operations & Transformations . . . 18
A.8.6 Lattice & Algebraic Utilities . . . 18
A.8.7 Metadata, Printing & Comparison . . . 18
A.8.8 Pending / Under Development Functions . . . 18

**References** **19**

# 1 Introduction

## 1.1 Motivations

When dealing with a mathematical model, it is often desired to theoretically identify which parameter combinations of that model are uniquely recovered from ideal, noise-free input-output data. From this desire arises the definition of structural identifiability. We follow the definitions given in [1].

**Definition 1.1** (Structural Identifiability)**.** For a given model with state equations $\dot{\mathbf{x}} = f(\mathbf{x}, t, \mathbf{u}, \mathbf{p})$ and output $\mathbf{y}$, an individual parameter $p$ is *uniquely* (or *globally*) *structurally identifiable* if for almost every value $\mathbf{p}^*$ and almost all initial conditions, the equation $\mathbf{y}(\mathbf{x}, t, \mathbf{p}^*) = \mathbf{y}(\mathbf{x}, t, \mathbf{p})$ implies $\mathbf{p} = \mathbf{p}^*$. An individual parameter $p$ is *non-uniquely* (or *locally*) *structurally identifiable* if for almost every value $\mathbf{p}^*$ and almost all initial conditions, the equation $\mathbf{y}(\mathbf{x}, t, \mathbf{p}^*) = \mathbf{y}(\mathbf{x}, t, \mathbf{p})$ implies $\mathbf{p}$ has a finite number of solutions.

**Definition 1.2.** Similarly, a model with state equations $\mathbf{y} = f(\mathbf{x}, t, \mathbf{u}, \mathbf{p})$ and output $\mathbf{y}$ is said to be *uniquely* (respectively *non-uniquely*) *structurally identifiable* for a given choice of output $\mathbf{y}$ if every parameter is uniquely (respectively non-uniquely) structurally identifiable.

Recent work has demonstrated that computational algebraic geometry tools such as Gröbner bases can be used to gain insights into parameter identifiability and model dynamics across mathematical biology [2]. While Gröbner bases have become a key tool for understanding the geometry of parameter relationships in applied models, their computation is highly sensitive to the choice of monomial ordering. Calculating a Gröbner basis with a suboptimal ordering can lead to explosive computational bottlenecks, yet selecting an optimal ordering is rarely straightforward.

This motivates the desire to compute a *Gröbner fan*, which is a polyhedral fan that encodes all possible monomial orderings for the Gröbner bases of a given ideal. By traversing this geometric object, we can determine an optimal ordering for a Gröbner basis, hence simplifying and speeding up the following algebraic calculations.

To traverse this structure, we turn to the algorithm developed by Fukuda, Jensen, and Thomas [3]. This algorithm enumerates all reduced Gröbner bases and maximal cones of an ideal $I$ by defining a unique sink orientation on the adjacency graph of maximal cones and traversing it via Fukuda's reverse search. Crucially, this approach offers two major computational advantages:

(i) **Memory Efficiency:** It achieves an $O(1)$ space complexity relative to the total number of bases, traversing the entire fan using local facet walks without storing previously visited cones in memory.

(ii) **Generality:** It works uniformly for both **homogeneous and non-homogeneous** ideals, making it particularly well-suited for the non-homogeneous polynomial systems often arising in biological modeling.

While an algorithm for computing Gröbner fans has been implemented in C++ (see the `gfan` package by Fukuda et al. [3]), many identifiability analyses conducted directly within Mathematica. Implementing this natively in Mathematica allows researchers to perform end-to-end identifiability pipelines within a single software environment. Furthermore, Mathematica's functional programming paradigm makes computing and manipulating high-level algebraic objects like Gröbner fans more accessible and readable than in procedural or lower-level languages, where custom structures must be constructed from scratch. Thus, in this project, we implemented the Fukuda–Jensen–Thomas algorithm in Mathematica.

## 1.2 Preliminaries

The primary objective of this section is to establish the algebraic and geometric machinery needed to define a **Gröbner fan**. We begin with monomial orderings, which allow us to select unique "leading terms" in polynomial equations. We then progress to Gröbner bases, generating sets that preserve these leading-term structures, and observe how varying the term ordering changes the basis. Finally, by parameterizing these orderings geometrically using weight vectors, we collect all possible Gröbner bases of an ideal into a single geometric object: the Gröbner fan. Detailed proofs and foundational results for these concepts can be found in [4].

To analyze polynomials algorithmically, we must first establish a consistent way to compare monomial terms.

**Definition 1.3** (Monomial Ordering)**.** A **monomial ordering** $>$ on $k[x_1, \ldots, x_n]$ is any relation $>$ on $\mathbb{Z}^n_{\geq 0}$, or equivalently, any relation on the set of monomials $x^\alpha$, $\alpha \in \mathbb{Z}^n_{\geq 0}$, satisfying:

(i) $>$ is a total (or linear) ordering on $\mathbb{Z}^n_{\geq 0}$.

(ii) If $\alpha > \beta$ and $\gamma \in \mathbb{Z}^n_{\geq 0}$, then $\alpha + \gamma > \beta + \gamma$.

(iii) $>$ is a well-ordering on $\mathbb{Z}^n_{\geq 0}$ (every non-empty subset of $\mathbb{Z}^n_{\geq 0}$ has a least element under $>$).

While any relation satisfying these three properties forms a valid monomial order, different specific orderings emphasize different structural properties of polynomials. Below, we introduce the three most widely used monomial orderings in computer algebra.

**Definition 1.4** (Lexicographic Order)**.** Let $\alpha = (\alpha_1, \ldots, \alpha_n)$ and $\beta = (\beta_1, \ldots, \beta_n) \in \mathbb{Z}^n_{\geq 0}$. We say $\alpha >_{\text{lex}} \beta$ if, in the vector difference $\alpha - \beta \in \mathbb{Z}^n$, the leftmost nonzero entry is positive. We write $x^\alpha >_{\text{lex}} x^\beta$ if $\alpha >_{\text{lex}} \beta$.

**Proposition 1.5.** *The lexicographic ordering on $\mathbb{Z}^n_{\geq 0}$ is a monomial order.*

*Proof.* A proof is given in [4]. □

**Definition 1.6** (Graded Lexicographic Order)**.** Let $\alpha, \beta \in \mathbb{Z}^n_{\geq 0}$. We say $\alpha >_{\text{grlex}} \beta$ if

$$|\alpha| = \sum_{i=1}^{n} \alpha_i > |\beta| = \sum_{i=1}^{n} \beta_i \quad \text{OR} \quad (|\alpha| = |\beta| \text{ and } \alpha >_{\text{lex}} \beta)\,.$$

**Definition 1.7** (Graded Reverse Lexicographic Order)**.** Let $\alpha, \beta \in \mathbb{Z}^n_{\geq 0}$. We say $\alpha >_{\text{grevlex}} \beta$ if

$$|\alpha| = \sum_{i=1}^{n} \alpha_i > |\beta| = \sum_{i=1}^{n} \beta_i$$

OR $|\alpha| = |\beta|$ and the rightmost nonzero entry of $\alpha - \beta \in \mathbb{Z}^n$ is negative.

Once a monomial order is fixed, every non-zero polynomial possesses a unique highest-degree monomial, known as its leading term. To study the collection of all such terms within a polynomial ideal, we examine monomial ideals and leading-term ideals.

**Definition 1.8** (Ideal)**.** Given a polynomial ring $k[x_1, \ldots, x_n]$. A subset $I \subset k[x_1, \ldots, x_n]$ is an **ideal** if the following statements hold true:

- $0 \in I$.
- If $f, g \in I$, then $f + g \in I$.
- If $f \in I$ and $h \in k[x_1, \ldots, x_n]$, then $hf \in I$.

**Lemma 1.9.** *Let $f_1, \ldots, f_s \in k[x_1, \ldots, x_n]$ be polynomials. The set*

$$\langle f_1, \ldots, f_s \rangle = \left\{ \sum_{i=1}^{s} h_i f_i : h_i \in k[x_1, \ldots, x_n] \right\}$$

*is an ideal. We call it the **ideal generated by** $f_1 \ldots, f_s$.*

*Proof.* A proof is given in [4]. □

**Definition 1.10** (Monomial Ideal)**.** An ideal $I \subset k[x_1, \ldots, x_n]$ is a **monomial ideal** if there is a subset $A \subset \mathbb{Z}^n_{\geq 0}$ such that $I$ consists of all polynomials that are finite sums of the form $\sum_{\alpha \in A} h_\alpha x^\alpha$, where $h_\alpha \in k[x_1, \ldots, x_n]$. In this case, we write $I = \langle x^\alpha : \alpha \in A \rangle$.

**Definition 1.11** (Leading Term Ideal)**.** Let $I \subset k[x_1, \ldots, x_n]$ be an ideal with $I \neq \{0\}$.

(i) We denote by $\text{LT}(I)$ the set of leading terms of elements of $I$:

$$\text{LT}(I) = \{cx^\alpha : \text{there exists } f \in I \text{ with } \text{LT}(f) = cx^\alpha\}\,.$$

(ii) We denote by $\langle \text{LT}(I) \rangle$ the ideal generated by the elements of $\text{LT}(I)$.

In general, taking the leading terms of an arbitrary generating set for $I$ does not necessarily generate the entire leading-term ideal $\langle \text{LT}(I) \rangle$. This distinction motivates the definition of a Gröbner basis, a special generating set whose leading terms completely determine $\langle \text{LT}(I) \rangle$.

**Definition 1.12** (Gröbner Basis)**.** Fix a monomial order $>$. A finite subset $G = \{g_1, \ldots, g_t\}$ of an ideal $I$ is a **Gröbner basis** if $\langle \text{LT}(g_1), \ldots, \text{LT}(g_t) \rangle = \langle \text{LT}(I) \rangle$.

**Corollary 1.13.** *Fix a monomial order. Then every ideal $I \subset k[x_1, \ldots, x_n]$ with $I \neq \{0\}$ has a Gröbner basis. Furthermore, any Gröbner basis for an ideal $I$ is also a generating set (basis) of $I$.*

*Proof.* A proof is given in [4]. □

Crucially, a Gröbner basis is not an invariant property of an ideal alone. It depends fundamentally on the chosen monomial order $>$. Different orderings yield distinct Gröbner bases for the exact same ideal [5].

To capture and organize all possible Gröbner bases of an ideal geometrically, we partition the space of term orderings into polyhedral regions (cones). Within each cone, the relative ordering of terms remains fixed, yielding the same Gröbner basis. We parameterize these regions using exponent differences between the leading term and the remaining support terms of each basis element.

**Definition 1.14** (Exponent Difference Set)**.** Let $I \subset k[x_1, \ldots, x_n]$ be an ideal with $I \neq \{0\}$, and let $G = \{g_1, \ldots, g_t\}$ be its reduced Gröbner basis under a given ordering. We denote

$$\mathrm{DIFF}(G) = \bigcup_{i=1}^{t} \{\log(m_i) - \log(t)\}_{t \in \mathrm{Support}(R_i)},$$

where $m_i = \mathrm{LM}(g_i)$ is the leading monomial, $R_i = g_i - m_i$ is the tail of $g_i$, and $\log(x^\alpha) = \alpha \in \mathbb{Z}^n_{\geq 0}$.

The vectors in $\mathrm{DIFF}(G)$ define linear inequalities in weight space. The set of all weight vectors that preserve $G$ forms a polyhedral cone. Assembling these cones across all possible reduced Gröbner bases yields the overall geometric partition known as the Gröbner fan.

**Definition 1.15** (Gröbner Fan)**.** Given an ideal $I \subset k[x_1, \ldots, x_n]$ with $I \neq \{0\}$, the collection of polyhedral cones

$$F(I) = \{\pi(D_1^*), \ldots, \pi(D_r^*)\}$$

is called the **Gröbner fan** of $I$, where $D_i^*$ is the polar cone of $D_i = \mathrm{DIFF}(G_i)$ for each distinct reduced Gröbner basis $G_i$, and $\pi : \mathbb{R}^{n+1} \to \mathbb{R}^n$ defined by $\pi(a_0, a_1, \ldots, a_n) = (a_1, \ldots, a_n)$ is the projection onto the last $n$ coordinates.

# 2 Algorithmic Implementations

In this section, we present the Mathematica implementation of the Fukuda–Jensen–Thomas algorithm, adapted from the original C++ implementation in the `gfan` library [3, 6]. Because this package is an ongoing development effort, we explicitly detail here the overall computational pipeline, the data structures implemented to date, and the remaining components currently under development.

## 2.1 Algorithmic Overview and Pipeline Implementation

### 2.1.1 Overview of the Mathematica Pipeline

When complete, the package will expose a top-level command, allowing users to compute the Gröbner fan $F(I)$ of an ideal $I = \langle f_1, \ldots, f_s \rangle$ directly within a Mathematica notebook. Conceptually, the execution pipeline operates in three main stages:

(i) **Initialization & Data Representation:** The input ideal is parsed and saturated if necessary. The package evaluates an initial reduced Gröbner basis $G_0$ and formats it into symbolic data structures parameterized by weight vectors and term orderings.

(ii) **Local Facet Walks & Cone Geometry:** For a given reduced Gröbner basis $G$, the package constructs the exponent difference set $\mathrm{DIFF}(G)$, solves for the defining inequalities of the polyhedral weight cone $C(G)$, and executes local polynomial reductions across facets to identify neighboring bases.

(iii) **Global Traversal Modulo Symmetry:** Guided by Fukuda's memoryless reverse search (or Gröbner walk algorithms), the engine recursively navigates the dual graph of maximal cones to enumerate all reduced Gröbner bases without storing previously visited cones in memory, taking advantage of discrete symmetry group reductions when available.

### 2.1.2 Implementation Status

The foundational algebraic engines, polyhedral geometry representations, LP backends, symmetry modules, and C++ interfaces are fully implemented, tested, and optimized for Mathematica. Active development is currently focused on refining the local facet walk subroutines in `wallideal.wl` and assembling the top-level global reverse search recursion.

The subsequent subsections detail the algorithmic role, mathematical design, and computational complexity of each individual package within this pipeline.

## 2.2 Saturation.wl

### 2.2.1 Package Overview

`Saturation.wl` provides algorithms for ideal saturation and ideal intersection using variable elimination via Gröbner bases. The list of functions can be found in Appendix A.1.

By introducing an auxiliary dummy variable $t$ and placing it first in an elimination term order, the package computes saturations $(I : f^{\infty}) = (I + \langle 1 - tf \rangle) \cap k[\texttt{vars}]$ and uses linear interpolation ideals $(tI + (1-t)J) \cap k[\texttt{vars}]$ to compute ideal intersections $I \cap J$.

## 2.3 TermOrder.wl

### 2.3.1 Package Overview

`TermOrder.wl` implements a unified, functional definition of monomial term orders necessary for commutative algebra and computational algebraic geometry. The list of functions can be found in Appendix A.2.

Rather than relying on fixed symbol definitions, term orders are instantiated as standard Association structures storing functional closures (`RowDotImpl` and `CompareImpl`). This design allows flexible composition, such as breaking ties in weight-based or matrix term orders with secondary orders, while providing exact comparison predicate methods supporting scalar scaling ($c \cdot a < d \cdot b$) and degree-bounded perturbations used during Gröbner basis calculation and Gröbner walk algorithms.

### 2.3.2 Complexity

We consider the main functionalities of this package:

- `TermOrderCompare[to, a, b, scaleA, scaleB, perturbationDegree]`: This function compares two exponent vectors $\vec{a}, \vec{b} \in \mathbb{N}^n$ under a term order to sequentially evaluating row dot products or component comparisons up to the specified `perturbationDegree` threshold. For elementary term orders (such as `LexicographicTermOrder` or `ReverseLexicographicTermOrder`), the algorithm iterates through vector components in worst-case time $\mathcal{O}(n)$, where $n$ is the ambient dimension (number of variables). For composite or matrix orders (such as MatrixTermOrder with $m$ weight rows), the comparison evaluates vector dot products across up to $m$ weight rows before falling back to a tie-breaker comparison, resulting in an overall worst-case complexity of $\mathcal{O}(m \cdot n)$, where $m \leq n$ for full-rank term orders.
- `TermOrderPrintMatrix[to, dim]`: This function generates the explicit matrix representation of a term order up to ambient dimension $n = \dim$ by evaluating TermOrderRowDot on all standard basis vectors $e_j$ for each row $i \in \{0, \ldots, n-1\}$. Because generating each matrix entry $M_{i,j}$ evaluates a row dot action requiring $\mathcal{O}(1)$ or $\mathcal{O}(n)$ arithmetic operations depending on whether the row is a coordinate extraction or an explicit vector dot product, assembling the complete $n \times n$ matrix requires evaluating $n^2$ basis entries, yielding a total time complexity of $\mathcal{O}(n^3)$ in exact arithmetic for matrix/weight-based term orders and $\mathcal{O}(n^2)$ for pure coordinate term orders.

## 2.4 Symmetry.wl

### 2.4.1 Package Overview

`Symmetry.wl` provides a concrete, 0-indexed permutation group engine made for geometric and algebraic applications modulo discrete group symmetries. The module treats group elements as index vectors acting on spatial coordinates, cone ray sums, and polynomial exponent vectors. By computing canonical orbit representatives, point stabilizers, and fundamental domain boundaries, the package enables

algorithms to operate entirely within a fundamental domain or modulo a group action $G$. The list of functions can be found in Appendix A.3.

Computing Gröbner fans modulo symmetry requires tracking how variable permutations act on polynomial ideals, initial term structures, and weight vectors:

(i) *Canonical Orbit Mapping*: `SGOrbitRepresentative` and `SGOrbitRepresentativeIndex` transform any arbitrary weight vector or ray sum into a canonical representative within its group orbit, ensuring that symmetrically equivalent Gröbner cones hash to the identical *SortKey*.

(ii) *Fundamental Domain Reduction*: `SGFundamentalDomainInequalities` finds the hyperplanes that bound the fundamental domain of $G$. Restricting the Gröbner fan search space to this fundamental region avoids redundant Gröbner basis computations.

(iii) *Symmetry Orbit Sizes*: `SGOrbitSize` and `SGStabilizer` evaluate orbit multiplicities, enabling exact computation of total fan $f$-vectors without explicitly generating full symmetric orbits.

### 2.4.2 Complexity

We consider the main functionalities of this package:

- `SGOrbitRepresentative[group, v]`: This function finds the lexicographically maximal vector in the orbit $G \cdot v$. It iterates through every group element $g \in G$, computes $g \cdot v$ via array remapping, and keeps the lexicographically largest vector using explicit array comparisons. The complexity is
$$\mathcal{O}(|G| \cdot n),$$
where $|G|$ is the order of the symmetry group and $n$ is the dimension of vector $v$.
- `SGComputeClosure[group, generators]`: This function computes the closure of a permutation set under composition with a set of generators. It performs a breadth-first search, composing existing group elements with all generators and deduplicating results until no new permutations are produced. The complexity is
$$\mathcal{O}(|G| \cdot |S| \cdot n),$$
where $|G|$ is the order of the closed group, $|S|$ is the number of generators, and $n$ is the base set size (cost of vector composition and deduplication).
- `SGFundamentalDomainInequalities[group]`: This constructs the linear inequality representation of the group's fundamental domain. It maps each permutation $g \in G$ to its first non-fixed position inequality vector, removes zero vectors, and filters duplicate hyperplane normals. The complexity is
$$\mathcal{O}(|G| \cdot n),$$
where $|G|$ is the number of group elements and $n$ is the dimension of the base space.

## 2.5 SymmetricComplex.wl

### 2.5.1 Package Overview

`SymmetricComplex.wl` implements a data structure for representing and manipulating polyhedral fan structures modulo discrete group symmetries. Instead of exhaustively storing every geometric cone in a symmetric polyhedral complex, the module stores canonical orbit representatives of cones under the action of a finite symmetry group $G$. The list of functions can be found in Appendix A.4. The key features include:

(i) *Hash-Based Canonical Storage*: Cones are hashed in an `Association` by a canonical `SortKey` obtained via symmetry orbit reduction (`SymmetrySGOrbitRepresentativeIndex`).

(ii) *Symmetric Combinatorics*: Maximality checks (`SCIsMaximal`) and subset inclusions automatically consider all group transformations $g \in G$.

(iii) *Homological Analysis*: The package provides orientation and boundary operator routines to compute the homology and invariants ($f$-vectors) of polyhedral fans (`SCBoundaryMap`).

Computing the Gröbner fan of a polynomial ideal requires identifying all polyhedral cones corresponding to distinct initial ideals, a process that benefits significantly from symmetric fan representations. Because Gröbner fans frequently exhibit rich geometric symmetries, such as variable permutations

under the symmetric group $S_n$, storing and traversing the fan modulo symmetry reduces memory and computational complexity by up to a factor of $|G|$, where $|G|$ is the order of the symmetry group.

During reverse search or boundary traversal algorithms, such as the Gröbner walk or secondary fan computations, operations like `SCInsert` and `SCIsMaximal` prevent the redundant processing of symmetrically equivalent maximal cones. Furthermore, topological verification methods including `SCFvector` (which computes orbit sizes) and `SCBoundaryMap` allow verification of fan completeness, Euler characteristics, and connectivity without expanding the full symmetric orbits in memory.

### 2.5.2 Complexity

We consider the most important functionalities of this package, which are `SCIsMaximal[complex, cone]` and `SCBoundaryMap[complex, d]`.

- `SCIsMaximal[complex, cone]`: This function determines whether a given cone is maximal in the complex modulo symmetry by iterating through every group element $g \in G$, permuting the cone's vertices, and checking for strict subset containment against all stored orbit-representative cones. This process yields a worst-case computational complexity of

$$\mathcal{O}(|G| \cdot |C| \cdot k \log k),$$

  where $|G|$ is the order of the symmetry group, $|C|$ is the total number of stored orbit-representative cones in the complex, and $k$ is the maximum number of vertices per cone, with the $k \log k$ factor accounting for vertex index sorting and set inclusion checks.
- `SCBoundaryMap[complex, d]`: This function constructs the cellular boundary matrix $\partial_d$ for dimension $d$ by iterating over all $|C_d|$ cones of dimension $d$ and invoking `SCBoundary` on each to determine its constituent facet boundaries and relative orientation signs. For every facet of a $d$-dimensional cone, this computation requires solving exact matrix kernel problems, computing spatial determinants in ambient dimension $n$, and performing canonical position lookups via `SCDimensionIndex`, resulting in an overall time complexity of

$$\mathcal{O}\left(|C_d| \cdot F \cdot \left(n^3 + \log |C_{d-1}|\right)\right),$$

  where $F$ denotes the maximum number of facets per $d$-cone, $n^3$ accounts for exact linear algebra operations in $n$-dimensional space, and $\log |C_{d-1}|$ represents the binary search overhead for indexing boundary facets within the $(d-1)$-dimensional cone set.

## 2.6 LpSolver.wl

### 2.6.1 Package Overview

`LpSolver.wl` establishes an abstract linear programming and polyhedral solver interface designed to decouple high-level geometric algorithms from low-level linear programming solvers. The list of functions can be found in Appendix A.5.

By defining generic function wrappers (`isFacet`, `hasHomogeneousSolution`, `relativeInteriorPoint`, etc.), the package allows polyhedral fan computations to swap between different exact rational or floating-point LP backends. Additionally, the module implements exact rational matrix preprocessing, such as clearing entry denominators via `primitiveMatrix`, and provides optimized subroutines for cone membership testing (`isInNonNegativeSpan`) and simplicial cone verification.

Computing Gröbner fans relies heavily on linear programming to navigate, traverse, and analyze Gröbner cones. During fan traversal algorithms such as the Gröbner walk or reverse search, operations like `isFacet` and `removeRedundantRows` determine the irreducibility of bounding hyperplanes and identify valid facet boundaries shared between adjacent maximal cones. To uniquely represent initial ideals associated with these cones, `relativeInteriorPoint` and `interiorPoint` extract valid weight vectors from the relative interior of a Gröbner cone. Furthermore, `isInNonNegativeSpan` verifies whether candidate weight vectors lie within the non-negative conic span of ray generators, while subroutines such as `extremeRaysInequalityIndices` and dual compute the extreme rays and polar dual representations required for secondary fan construction and tropical geometric analysis.

### 2.6.2 Complexity

We consider the most important functionalities:

- `isInNonNegativeSpan[v, rays, linealitySpace]`: This function determines whether a vector $\vec{v} \in \mathbb{R}^n$ belongs to the non-negative span of $m$ ray vectors and $l$ lineality generators by constructing a system of linear constraints $A_1 x \geq 0$ and $A_2 x = 0$, where $A_1$ is an $m \times (1+m+l)$ bound matrix and $A_2$ is an $n \times (1+m+l)$ block matrix containing $-\vec{v}$, the rays, and the lineality basis. Preprocessing $A_1$ and $A_2$ into primitive integer matrices via primitiveMatrix requires $\mathcal{O}(n(m+l) \cdot \mathrm{M}(L))$ operations to compute GCDs and LCMs across entry denominators (where $L$ is the maximum bit length of the input fractions and $\mathrm{M}(L)$ is the cost of integer multiplication). Solving the resulting homogeneous LP with $n+m$ constraints in $1+m+l$ variables takes polynomial time $\mathcal{O}\left((n+m+l)^{3.5} \cdot L\right)$ arithmetic operations using exact rational interior-point LP methods, making the overall time complexity $\mathcal{O}\left(n(m+l) \cdot \mathrm{M}(L) + (n+m+l)^{3.5} \cdot L\right)$.
- `extremeRaysInequalityIndices[inequalityList]`: This function determines the facial index combinations defining extreme rays for a system of $m$ linear inequalities in $n$-dimensional space by first checking whether the cone is simplicial via `rankOfMatrix`. If the constraint matrix has full rank equal to $m$, the function executes an optimized combinatorial path that constructs index set complements in $\mathcal{O}(m^2)$ time without calling an LP solver. Otherwise, for non-simplicial cones, the function delegates to the active LP backend, which tests each of the $m$ inequalities for extreme ray status by solving $m$ linear sub-programs or dual ray enumerations, yielding an overall computational complexity of $\mathcal{O}\left(m \cdot \mathrm{LP}(m, n)\right)$, where $\mathrm{LP}(m, n)$ represents the time required to solve a linear program with $m$ constraints and $n$ variables (typically $\mathcal{O}(m^{3.5} \cdot n \cdot L)$ under exact arithmetic).
- `removeRedundantRows[inequalities, equalities, removeInequalityRedundancies]`: This function eliminates redundant linear constraints from a system of $m$ inequalities and $k$ equality constraints in $n$ dimensions by invoking the backend LP solver's redundancy detection routines. The underlying algorithm tests whether each constraint is redundant by setting up a linear program that maximizes the slack of that constraint subject to all remaining inequalities and equations; if the maximum feasible slack is non-positive, the constraint is classified as redundant and removed. Iterating this feasibility test across all $m$ inequalities requires solving up to $m$ individual linear programs, resulting in a worst-case computational complexity of $\mathcal{O}\left(m \cdot \mathrm{LP}(m+k, n)\right)$, which translates to $\mathcal{O}\left(m \cdot (m+k)^{3.5} \cdot n \cdot L\right)$ arithmetic operations when using exact rational LP algorithms.

## 2.7 LpSolverMathematica.wl

### 2.7.1 Package Overview

`LpSolverMathematica.wl` implements a Wolfram Language solver engine for the LpSolver registry interface. The file provides a self-contained, cross-platform LP and polyhedral backend. It allows exact rational polyhedral geometric operations—such as relative interior point extraction, dual cone computation, and redundancy elimination—to execute directly inside Mathematica without external binary dependencies. The list of functions can be found in Appendix A.6.

### 2.7.2 Complexity

We consider the most important functionalities:

- `mRelativeInteriorPoint[n, g, equalitySet]`: This function extracts a relative interior point. Given $m = |g|$ constraints split into $m_{\text{eq}}$ equations and $m_{\text{ineq}}$ inequalities, the routine sets up a matrix of size $(m+1) \times (n+1)$ and solves $\min(-t)$ subject to $A_{\text{eq}} x = 0$, $A_{\text{ineq}} x - t \geq 0$, and $t \geq 0$. Using exact rational interior-point or Revised Simplex algorithms, solving this LP requires $\mathcal{O}\left((m+n)^{3.5} \cdot L\right)$ arithmetic operations, where $L$ is the maximum bit length of the constraint coefficients. Post-processing the resulting optimal solution vector with `primitiveVector` takes $\mathcal{O}(n \cdot \mathrm{M}(L))$ operations to compute entry clearing, yielding an overall computational complexity of $\mathcal{O}\left(n \cdot \mathrm{M}(L) + (m+n)^{3.5} \cdot L\right)$.
- `mExtremeRaysInequalityIndices[inequalityList]`: This function enumerates extreme rays for a cone defined by $m$ inequalities in $\mathbb{R}^n$ by iterating over all $(n-1)$-element subsets of bounding hyperplanes (a combinatorial double-description approach). For each of the $\binom{m}{n-1}$ subsets, the algorithm computes the 1-dimensional kernel in $\mathcal{O}((m-n+1)n^2)$ arithmetic steps. It then verifies

whether the candidate generator $v$ or $-v$ satisfies all $m$ linear inequalities in $\mathcal{O}(mn)$ steps, followed by exact primitive integer scaling in $\mathcal{O}(n \cdot \mathrm{M}(L))$. The overall time complexity is bounded by $\mathcal{O}\left(\binom{m}{n-1} \cdot \left(mn^2 + \mathrm{M}(L)\right)\right)$, which exhibits exponential scaling in the dimension $n$ for non-simplicial cones, making it ideal for low-dimensional or highly constrained Gröbner cones.

- `mDual[n, inequalities, equations]`: This function computes the polar dual cone representation from a set of $m_{\text{ineq}}$ inequality rays and $m_{\text{eq}}$ equation generators. Defining $M = m_{\text{ineq}} + 2m_{\text{eq}}$ total generator vectors in $\mathbb{R}^n$, the function first extracts the linear subspace of dual equations in $\mathcal{O}(Mn^2)$ steps, leaving a cone of effective dimension $k = n - \text{rank(gens)}$. It then iterates through all $\binom{M}{k-1}$ subsets of generators, computing 1-dimensional nullspaces in $\mathcal{O}(kn^2)$ steps and evaluating non-negativity across all $M$ generators in $\mathcal{O}(Mn)$ steps per subset. The total computational complexity is $\mathcal{O}\left(Mn^2 + \binom{M}{k-1} \cdot \left(Mn + kn^2\right)\right)$ rational arithmetic operations, converting V-representations to minimal H-representations for dual polyhedral cones.

## 2.8 GroebnerEngine.wl

### 2.8.1 Package Overview

`GroebnerEngine.wl` provides an abstract execution engine bridging custom monomial ordering representations (from `TermOrder.wl`) and ideal saturation utilities (from `Saturation.wl`) with Wolfram Language's native polynomial reduction system (`GroebnerBasis` and `PolynomialReduce`). It translates custom term order structures—such as lexicographic, degree lexicographic, and standard degree reverse lexicographic orders—into Mathematica's built-in MonomialOrder directives ("Lexicographic", "DegreeLexicographic", and "DegreeReverseLexicographic"). Additionally, the module integrates optional non-homogeneous ideal saturation on algebraic tori prior to basis construction and implements polynomial set inter-reduction. The list of functions can be found in A.7.

Computing Gröbner fans requires generating reduced Gröbner bases across a sequence of weight-driven monomial orderings corresponding to polyhedral cones in weight space. This package's functions are used as the primary computational engine during fan traversal, facet crossing, and initial ideal evaluation by converting abstract term order definitions into native Mathematica execution calls. When navigating from one Gröbner cone to an adjacent cone, the engine evaluates the necessary Gröbner bases to determine whether two weight vectors yield identical initial ideals. Furthermore, its integration with `Saturation` ensures that non-torus components supported on coordinate hyperplanes $x_i = 0$ are removed before basis calculation, while `GEAutoReduce` maintains minimal canonical representations of initial ideal generators across the Gröbner walk.

### 2.8.2 Complexity

We consider the most important functionalities:

- `GEGroebnerBasis[idealGenerators, vars, termOrder, autoreduce, allowSaturation]`: First, this maps the term order association to a native monomial order specification in $\mathcal{O}(n)$ time, where $n = |\texttt{vars}|$. If `allowSaturation` is set to True, it adds an auxiliary variable $t$ and computing an elimination Gröbner basis in time $\mathcal{O}\left(d^{2^{\mathcal{O}(n+1)}}\right)$, where $d$ is the maximum polynomial degree. The final basis computation via Mathematica's native GroebnerBasis using Buchberger's algorithm requires worst-case doubly exponential time $\mathcal{O}\left(d^{2^{\mathcal{O}(n)}}\right)$ arithmetic operations for general ideals (or singly exponential time $d^{\mathcal{O}(n)}$ for zero-dimensional ideals under graded monomial orders like grevlex). Thus, the overall computational complexity is dominated by the Gröbner basis computation, bounded by $\mathcal{O}\left(d^{2^{\mathcal{O}(n)}}\right)$.
- `GEAutoReduce[idealGenerators, vars, recipe]`: This function inter-reduces $|\texttt{idealGenerators}|$ polynomials by computing the multivariate remainder of each generator against all remaining $m-1$ generators. For a polynomial containing up to $N$ monomial terms in $n$ variables, a single multivariate division step against $m-1$ polynomials takes $\mathcal{O}(m \cdot N^2 \cdot n)$ polynomial coefficient operations in the worst case. Executing this division for each of the $m$ input generators results in an overall computational complexity of $\mathcal{O}(m^2 \cdot N^2 \cdot n)$ arithmetic operations.
- `GEOrderSpec[termOrder, vars]`: This function translates custom term orders into native Mathematica `MonomialOrder` parameters. It inspects the order name, performs list rotation on `vars` to

position the primary variable in $\mathcal{O}(n)$ time, and verifies weight vectors using vector equality checks in $\mathcal{O}(n)$ time. The entire term order mapping routine operates in deterministic linear time $\mathcal{O}(n)$, where $n$ is the number of variables in the polynomial ring.

## 2.9 PolyhedralCone.wl

### 2.9.1 Package Overview

`PolyhedralCone.wl` provides the core polyhedral geometry data structures and geometric algorithms required for exact computations over convex rational polyhedral cones in $\mathbb{R}^n$. The list of functions can be found in Appendix A.8.

It manages cone state transitions across three levels:

(i) *State 1*: Linear equivalence reduction via Gaussian elimination
(ii) *State 2*: Redundancy removal of bounding hyperplanes via linear programming
(iii) *State 3*: Canonical sorting for exact equality and total ordering comparisons

`PolyhedralCone.wl` serves as the primary polyhedral engine for representing and manipulating individual Gröbner cones, which form the maximal cones of a Gröbner fan. During fan traversal, it extracts boundary facets shared between adjacent initial ideals and generates valid weight vectors to evaluate initial monomial ideals. Functions like `PCDualCone`, `PCGivenByRays`, and `PCInducedLink` are essential for converting between weight vector representations and secondary fans, while `PCQuotientLatticeBasis` and `PCGetMultiplicity` compute the lattice multiplicities and algebraic degrees needed when constructing tropical varieties and computing intersection numbers in toric geometry.

There are two functions that remain under development. The first one is `PCHasFace[cone, f]`. When completed, this function will take two polyhedral cones, and return True if f is a valid polyhedral face of cone (i.e., $f = \text{cone} \cap \{x : w \cdot x = 0\}$ for some weight vector $w$ valid on cone). The second function is `PCTriangulation[cone]`. When completed, this function will decompose a non-simplicial polyhedral cone into a disjoint union of simplicial cones sharing the same extreme rays without adding new vertices.

### 2.9.2 Complexity

We consider the most important functionalities:

- `pcState1[cone]`: This function simplifies constraints and canonicalizes inequalities. Given $k$ equation vectors and $m$ inequality vectors in $\mathbb{R}^n$, computing RowReduce on equations requires $\mathcal{O}(k^2 n)$ arithmetic operations. Canonicalizing each of the $m$ inequalities against the row-reduced echelon form takes $\mathcal{O}(m \cdot kn)$ operations. Dropping zero vectors and deduplicating takes $\mathcal{O}(mn \log m)$ operations. The total time complexity for State 1 reduction is $\mathcal{O}\left(k^2 n + mkn + mn \log m\right)$.
- `PCQuotientLatticeBasis[cone]`: This function computes an integer basis for $(\mathbb{Z}^n \cap \text{span})/(\mathbb{Z}^n \cap \text{lineality})$. Extracting primitive integer kernels L1 and L2 on $k_1$ equation rows and $k_2$ combined rows requires $\mathcal{O}(n^3 \cdot \mathrm{M}(L))$ operations over $n \times n$ matrices with coefficient bit-length $L$. Solving $k_2$ linear systems to represent L2 in L1 coordinates takes $\mathcal{O}(k_2 \cdot k_1^3)$ arithmetic steps, followed by a second $k_2 \times k_1$ decomposition taking $\mathcal{O}(\max(k_1, k_2)^3 \cdot \mathrm{M}(L))$ steps. Matrix multiplication back to $\mathbb{Z}^n$ takes $\mathcal{O}(k_1 k_2 n)$ operations, yielding an overall computational time complexity of $\mathcal{O}\left(n^3 \cdot \mathrm{M}(L)\right)$.
- `PCDualCone[cone]` *(Dual case)*: For a linear subspace cone defined by $k$ equations in $\mathbb{R}^n$ with no inequalities, it evaluates NullSpace on the $k \times n$ equation matrix to find generator span vectors in $\mathcal{O}(k^2 n)$ steps. It then computes NullSpace a second time on the resulting $(n-k) \times n$ basis matrix to recover the row-space equations of $L^\perp$ in $\mathcal{O}((n-k)^2 n)$ steps. The overall time complexity for the subspace dual cone computation is $\mathcal{O}\left(n^3\right)$ exact rational arithmetic operations.

## 2.10 wallideal.wl

### 2.10.1 Package Overview

`wallideal.wl` implements the core GroebnerWalk algorithm using 4 core functions:

(i) *initialForm*: finds the initial form of an ideal with respect to a weight vector, $In_w(I)$ in the paper
(ii) *findNextFacet*: finds the first facet in the line segment between the current ideal and the target

(iii) *performFlip*: performs a local flip across a facet (algorithm 4.2 in the paper)

(iv) *groebnerWalk*: performs local flips from an initial ideal along cones until reaching the target ideal

### 2.10.2 Complexity

Let E be the edges and V be the vertices of the graph. The total time complexity of the enumeration of (V, E) is:

$$O(\sum_{G\in V} T_{facets}(G) + \sum_{G_1,G_2\in E} T_{shoot}(G_1) + \sum_{G_1,G_2\in E} T_{flip}(G_1,G_2))$$

where $T_{facets}$ is the time to compute the facet normals of the Grobner cone of a marked reduced Grobner basis G, $T_{shoot}(G)$ is the time to compute a search edge given a marked reduced Grobner basis G (both done in the findNextFacet function), and $T_{flip}(G)$ is the time to convert a marked reduced Grobner basis $G_1$ into a corresponding $G_2$ that share a facet (done in the performFlip function).

According to Fukada et. al. in their complexity section, this evaluates to:

$$O(\sum_{G\in V} T_{lp}(n, r(G))r(G) + \sum_{G_1,G_2\in E} r(G_1)n^2 + T_{lp}(n, G_1) + \sum_{G_1,G_2\in E} T_{flip}(G_1,G_2))$$

where r(G) is the number non-leading terms in the marked reduced Grobner basis G, $T_{lp}(n, r)$ is the time needed to solve a linear programming problem with n variables and r constraints. The first two terms are bounded by a polynomial in the size of the output.

### 2.10.3 issues

there is currently an issue in the implementation, most likely in the findNextFacet function, where no cones are traversed before the program reaches the ideal.

## 2.11 gfan_cpp_interface.wl

### 2.11.1 Package Overview

`gfan_cpp_interface.wl` provides a way to execute functions/programs in the GFan C++ library by running the code in C++ and returning it to Mathematica, allowing for easy verification of the Mathematica packages and comparison of runtimes, as well as the ability for mathematicians more experienced with Mathematica to take advantage of a highly optimized C++ library without the need to learn the language.

To use properly, one must properly install the GFan library locally, instructions for which can be found here: https://users-math.au.dk/jensen/software/gfan/gfanmanual0.7.pdf. Make sure the command directory of your Mathematica file can execute the GFan program.

# 3 Future Directions

We are working towards the following goals:

- Implement the full functional public library
- For structural identifiability problems, we want to use the aforementioned pipeline to find the optimal Gröbner basis ordering, eventually minimalizing the runtime of the Gröbner basis algorithm.
- Since Mathematica is a language optimized to "doing" math, we want to explore if the algorithm runs faster in Mathematica than in C++, and if yes, by how much.

# 4 Acknowledgments

It is a pleasure to thank our mentor, Dr. Sabina Haque, for her constant support and encouragement throughout the program. Without her, this project could not have been possible. We also want to express our gratitude towards the University of Michigan, Department of Mathematics for having organized a wonderful REU program, without which our undergraduate experience would have been vastly different.

# A Package Specifications

## A.1 Saturation.wl

### A.1.1 Ideal Operations

- `NonHomogeneousSaturation[idealGenerators, vars]`: Computes a Gröbner basis for the saturation ideal $(I : (x_1 x_2 \cdots x_n)^\infty)$ of ideal $I = \langle$`idealGenerators`$\rangle$ with respect to the product of all variables in `vars`
- `IdealIntersection[a, b, vars]`: Computes a Gröbner basis for the intersection of two ideals $I = \langle a \rangle$ and $J = \langle b \rangle$ in the polynomial ring $k$[`vars`].

## A.2 TermOrder.wl

### A.2.1 Generic Order Operations

- `TermOrderRowDot[to, row, v]`: Evaluates the inner product or component extraction for row index row of term order to against exponent vector $\vec{v}$
- `TermOrderCompare[to, a, b, scaleA, scaleB, perturbationDegree]`: Compares two exponent vectors $\vec{a}$ and $\vec{b}$ (with optional scaling factors and perturbation degree limits) under term order to, returning `True` if $\vec{a} < \vec{b}$ and `False` otherwise
- `TermOrderPrint[to]`: Prints the name and internal parameters of the term order structure
- `TermOrderPrintMatrix[to, dim]`: Constructs and prints the weight matrix corresponding to the term order up to ambient dimension `dim`.

### A.2.2 Term Order Constructors

- `LexicographicTermOrder[largest]`: Constructs a standard lexicographic term order starting from variable index largest
- `LexicographicInvertedTermOrder[]`: Constructs an inverted lexicographic term order starting comparisons from the last variable
- `ReverseLexicographicTermOrder[largest]`: Constructs a degree-reverse lexicographic order prioritizing smaller last exponents
- `ReverseLexicographicInvertedTermOrder[]`: Constructs an inverted reverse lexicographic order
- `StandardGradedLexicographicTermOrder[largest]`: Constructs a graded lex order comparing total degree first, then breaking ties lexicographically
- `WeightTermOrder[weight]`: Constructs a term order defined by a single weight vector, breaking ties with standard lexicographic order
- `WeightReverseLexicographicTermOrder[weight]`: Constructs a term order defined by a single weight vector, breaking ties with reverse lexicographic order
- `MatrixTermOrder[weights]`: Constructs a matrix-defined term order using a list of weight vectors, breaking remaining ties with inverted reverse lexicographic order
- `TotalDegreeTieBrokenTermOrder[tieBreaker]`: Constructs a graded term order using total degree first, breaking ties using an arbitrary specified secondary term order `tieBreaker`.

## A.3 Symmetry.wl

### A.3.1 Constructors & Group Algebra

- `SGNew[n]`: Creates a new `SymmetryGroup` association representing the trivial group (containing only the identity permutation on $n$ elements)
- `SGIdentity[n]`: Generates the identity permutation $\{0, 1, \ldots, n-1\}$
- `SGInverse[a]`: Computes the functional inverse of permutation $\vec{a}$
- `SGIsPermutationQ[a]`: Validates whether vector $\vec{a}$ is a valid permutation of $\{0, \ldots, n-1\}$
- `SGCompose[perm, b]`: Composes permutation perm with vector or permutation $\vec{b}$ via index remapping (`b[perm[i]]`).
- `SGComposeInverse[perm, b]`: Applies the inverse permutation action to vector $\vec{b}$

- `SGIsTrivial[group]`: Checks if the group consists solely of the identity element
- `SGSizeOfBaseSet[group]`: Returns the size of the base element set $n$.

### A.3.2 Group Growth & Generators

- `SGComputeClosure[group, generators]`: Computes the group closure under multiplication (composition) with a list of generator permutations.
- `SGGetUniqueGenerators[group]`: Extracts a minimal generating set for the group using exhaustive closure testing.

### A.3.3 Orbit & Stabilizer Operations

- `SGOrbitRepresentative[group, v]`: Identifies the lexicographically maximal vector in the orbit of vector $\vec{v}$ under the group action and returns {`representative`, `permutationUsed`}.
- `SGOrbitRepresentativeIndex[group, v]`: Returns {`representative`, `indexOfPermutationUsed`} using the 1-based index of the permutation in `group["Elements"]`
- `SGReferenceToIthVector[group, index]`: Fetches the $i$-th permutation element from the stored group array
- `SGOrbitRepresentativeFixing[group, v, fixed]`: Finds the lexicographically maximal representative of vector v under the subgroup that fixes the vector fixed
- `SGOrbitSize[group, stable]`: Calculates the size of the orbit of vector stable
- `SGStabilizer[group, v]`: Computes the pointwise stabilizer subgroup

$$\mathrm{Stab}_G(v) = \{g \in G : g \cdot v = v\}.$$

### A.3.4 Polyhedral & Geometric Utilities

- `SGFundamentalDomainInequality[perm, alternative]`: Returns a linear inequality vector defining the fundamental domain wall corresponding to a given permutation
- `SGFundamentalDomainInequalities[group, alternative]`: Returns the complete non-redundant set of linear inequalities cutting out the primary fundamental domain of the group
- `SGMergeSort[v]`: Sorts vector $\vec{v}$ and returns {`sortedVector`, `numberOfSwaps`}, where the swap count equals the number of inversions.

### A.3.5 Vector & Polynomial Actions

- `SGPermuteIntegerVectorList[l, v]`: Applies permutation $v$ to each vector in list $l$
- `SGPermuteInverseIntegerVectorList[perm, l]`: Applies inverse permutation perm to each vector in list $l$
- `SGAppendPermutedIntegerVectorList[l, perm, dest]`: Appends the permuted vectors of $l$ directly onto destination list `dest`.

## A.4 SymmetricComplex.wl

### A.4.1 Complex Constructors & Inspection

- `SCNew[n, vertexList, sym]`: Instantiates a new `SymmetricComplex` association for dimension $n$, given a list of ambient vertex vectors and a symmetry group sym
- `SCGetVertices[complex]`: Returns the list of ambient vertex vectors stored in the complex
- `SCIndexOfVertex[complex, v]`: Returns the 1-based index of a specific vertex vector $\vec{v}$
- `SCGetMaxDim[complex]`: Returns the maximum dimension among all cones in the complex
- `SCGetMinDim[complex]`: Returns the minimum dimension among all cones in the complex

### A.4.2 Cone Operations

- `SCConeNew[indexSet, dimension, multiplicity, sortWithSymmetry, complex]`: Constructs a Cone structure, deriving its canonical sort key (and orbit representative index) via group symmetry

- `SCConeIndexSet[cone]`: Returns the sorted list of 1-based vertex indices defining the cone
- `SCConeIsSubsetOf[cone, c]`: Checks whether cone is a subcone (face) of cone c using vertex index inclusion
- `SCConePermuted[cone, permutation, complex, withSymmetry]`: Applies a symmetry group permutation to a cone's vertices and constructs the transformed cone
- `SCConeLess[a, b]`: Compares two cones canonically based on their symmetry sort keys
- `SCConeIsSimplicial[cone, lineality Dim]`: Checks if a cone is simplicial
- `SCConeOrthogonalComplement[cone, complex]`: Computes integer primitive basis vectors for the orthogonal complement of the cone's vertex span

### A.4.3 Complex Modifications & Topological Properties

- `SCContains[complex, cone]`: Checks if a cone's orbit representative is already present in the complex
- `SCInsert[complex, cone]`: Inserts a cone into the complex (or updates non-maximality flags) keyed by its canonical orbit sort key
- `SCIsMaximal[complex, cone]`: Determines if a cone is maximal under the action of the symmetry group
- `SCIsPure[complex]`: Determines if all maximal cones in the complex have the same dimension
- `SCIsSimplicial[complex]`: Determines if every cone stored in the complex is simplicial
- `SCDimensionsAtInfinity[complex]`: Computes matrix ranks for rays intersecting the boundary at infinity in compactified/projective settings
- `SCFvector[complex, boundedPart]`: Calculates the orbit-weighted $f$-vector
- `SCRemap[complex]`: Normalizes vertex indices of stored cones using the symmetry group's reference transformations.

### A.4.4 Homology & Boundary Calculations

- `SCNumberOfConesOfDimension[complex, d]`: Counts the number of $d$-dimensional cones in the complex
- `SCDimensionIndex[complex, cone]`: Returns the 0-based position index of a $d$-dimensional cone within the lexicographically sorted array of all $d$-dimensional cones
- `SCBoundary[complex, cone]`: Computes the boundary facets of a cone along with their relative orientation signs
- `SCBoundaryMap[complex, d]`: Constructs the cellular boundary matrix $\partial_d : C_d \to C_{d-1}$ for algebraic homology computations
- `SCToString[complex, dimLow, dimHigh, onlyMaximal, group, opts...]`: Handles the textification of the complex structure.

## A.5 LpSolver.wl

### A.5.1 Solver Registry & Configuration

- `lpSetSolver[name]`: Sets the active LP backend solver
- `CreateLpSolver[name, methodsAssoc]`: Registers a new LP solver implementation by associating method names with dynamic function handlers
- `PrintSolverList[]`: Prints the list of currently linked and registered LP backend solvers.

### A.5.2 Polyhedral Feasibility & Ray Operations

- `isFacet[g, i]`: Determines if the $i$-th inequality in matrix $g$ defines a facet of the polyhedral cone
- `interiorPoint[g, result, strictlyPositive, equalitySet]`: Computes an interior point of a polyhedral cone under strict inequality and equality constraints
- `hasInteriorPoint[g, strictlyPositive, equalitySet]`: Returns a boolean indicating whether a valid interior point exists for the constraint system

- `relativeInteriorPoint[n, g, equalitySet]`: Computes a point lying in the relative interior of an $n$-dimensional polyhedral cone
- `shootRay[g]`: Computes a ray originating within or passing through the polyhedral region defined by $g$
- `hasHomogeneousSolution[n, inequalities, equations]`: Determines whether a non-trivial homogeneous solution $x$ exists for the system $Ax \geq 0, Bx = 0$
- `isInNonNegativeSpan[v, rays, linealitySpace]`: Checks if a target vector $v$ lies within the non-negative conic span of a set of rays plus a linear sub-space.

### A.5.3 Conic & Dual Representation

- `extremeRaysInequalityIndices[inequalityList]`: Identifies the index sets of inequalities defining extreme rays, featuring an optimized combinatorial shortcut for simplicial cones
- `removeRedundantRows[inequalities, equalities, removeInequalityRedundancies]`: Removes redundant linear inequalities and equations from a polyhedral representation
- `dual[n, inequalities, equations, dualInequalities, dualEquations]`: Computes the polar dual cone representation from hyperplanes and generators
- `positiveVectorInKernel[g, result]`: Finds a strictly positive vector contained within the kernel of matrix $g$.

### A.5.4 Linear Algebra Utilities

- `rankOfMatrix[g]`: Evaluates the linear rank of matrix $g$ via the active backend solver

## A.6 LpSolverMathematica.wl

### A.6.1 Solver Registration

- `CreateLpSolver["Mathematica", ...]/lpSetSolver["Mathematica"]`: Sets the "Mathematica" concrete backend as the default LP solver upon package load.

### A.6.2 Polyhedral & LP Engine Implementations

- `mRankOfMatrix[g]`: Evaluates the linear rank of matrix $g$ using native `MatrixRank`
- `mRelativeInteriorPoint[n, g, equalitySet]`: Solves a Chebyshev-center linear program via LinearProgramming to find a point in the relative interior of an $n$-dimensional polyhedral cone
- `mExtremeRaysInequalityIndices[inequalityList]`: Identifies the 0-indexed inequality sets defining extreme rays via codimension-1 subset nullspaces and non-negativity tests
- `mRemoveRedundantRows[inequalities, equalities, removeInequalityRedundancies]`: Deletes redundant bounding hyperplanes
- `mHasHomogeneousSolution[n, inequalities, equations]`: Determines if a non-trivial homogeneous vector $x \in [-1, 1]^n$ satisfies $Ax \geq 0$ and $Bx = 0$ via bounded linear programming
- `mDual[n, inequalities, equations]`: Computes the polar dual cone's dual inequalities and equations

### A.6.3 Vector Preprocessing Utility

- `primitiveVector[v]`: Rescales a rational vector $v$ to an integral vector by clearing entry denominators via LCM and dividing by the numerator GCD.

## A.7 GroebnerEngine.wl

### A.7.1 Engine Configuration & Setup

- `GESetEngine[name]`: Configures and validates the active Gröbner basis engine

### A.7.2 Gröbner Basis & Polynomial Reduction

- `GEGroebnerBasis[idealGenerators, vars, termOrder, autoreduce, allowSaturation]`: Computes a reduced Gröbner basis for `idealGenerators` in the polynomial ring $k$[`vars`] with respect to termOrder (lexicographic, degree lexicographic, or degree reverse lexicographic), with optional initial torus saturation $(I : (x_1 \cdots x_n)^\infty)$
- `GEAutoReduce[idealGenerators, vars, recipe]`: Inter-reduces a set of polynomial generators using multivariate polynomial division

## A.8 PolyhedralCone.wl

### A.8.1 Constructor, Initializers & State Machine

- `PCNew[inequalities, equations, n]`: Constructs a new polyhedral cone in $n$-dimensional space (initialized to State 1)
- `PCPositiveOrthant[dimension]`: Returns the positive orthant $x_1 \geq 0, \ldots, x_n \geq 0$ as a cone
- `PCPolyhedralConeWithKnownImpliedEquations[halfSpaces, equations, n]`: Constructor shortcut when implied equations are pre-computed
- `PCFindImpliedEquations[cone]`: Advances cone to State 1 (reduces equality space, canonicalizes inequalities modulo equations)
- `PCFindFacets[cone]`: Advances cone to State 2 (removes redundant bounding inequalities using LP)
- `PCCanonicalize[cone]`: Advances cone to State 3 (canonical sorted ordering for deterministic comparison)
- `PCGetState[cone] / PCIsInStateMinimum[cone, s]`: State accessors
- `PCAreFacetsKnown[cone] / PCAreImpliedEquationsKnown[cone]`: State boolean checks

### A.8.2 Basic Accessors & Geometric Indicators

- `PCGetHalfSpaces[cone] / PCGetEquations[cone]`: Returns the bounding inequalities and defining equations
- `PCAmbientDimension[cone] / PCDimension[cone] / PCCodimension[cone]`: Dimension queries ($n$, $n - \text{rank(Eqs)}$, and codimension)
- `PCIsZero[cone] / PCIsFullSpace[cone]`: Boolean checks for 0-dimensional origin cone or full space $\mathbb{R}^n$
- `PCIsSimplicial[cone]`: Checks if the cone is simplicial

### A.8.3 Containment & Membership Testing

- `PCContains[cone, v or {list of cone}]`: Tests point, matrix, or sub-cone containment
- `PCContainsRelatively[cone, v]`: Tests strict relative interior containment ($Ax = 0, Bx > 0$)
- `PCContainsPositiveVector[cone]`: Determines if the cone intersects the strictly positive orthant
- `PCContainsPerturbed[cone, l]`: Tests containment with respect to a lexicographically perturbed weight vector series
- `PCDoesSatisfyInequalityExpensive[cone, ineq]`: Tests if an inequality holds across all generators and lineality vectors

### A.8.4 Relative Interior Points & Generators

- `PCGetRelativeInteriorPoint[cone]`: Computes a point in the relative interior
- `PCGetUniquePoint[cone]` / `PCGetUniquePointFromExtremeRays[cone, extremeRays]`: Computes canonical representative interior points from extreme rays
- `PCExtremeRays[cone]`: Reconstructs extreme ray generators
- `PCGeneratorsOfSpan[cone]` / `PCGeneratorsOfLinealitySpace[cone]` / `PCLinealitySpace[cone]` / `PCDimensionOfLinealitySpace[cone]`: Computes linear span and lineality space generators

### A.8.5 Polyhedral Operations & Transformations

- `PCDualCone[cone]`: Computes the polar dual cone
- `PCNegated[cone]`: Negates all bounding inequalities
- `PCLink[cone, w]`: Computes the link/face cone orthogonal to weight vector $w$
- `PCFaceContaining[cone, v]` / `PCFaceContainingPerturbed[cone, l]`: Finds the minimal face containing a point $v$ or perturbed series $l$
- `PCIntersection[a, b]`: Computes the geometric intersection of two cones
- `PCGivenByRays[generators, linealitySpace, n]`: Changes V-representation to H-representation
- `PCProjection[cone, newn]`: Projects cone onto its first newn coordinates
- `PCSum[a, b]`: Computes the Minkowski sum of two cones
- `PCInducedLink[c, l]`: Computes the induced link cone of $c$ relative to subspace $l$
- `PCPermuted[cone, v]`: Permutes variable coordinates via symmetric group action
- `PCWithLastCoordinateRemoved[cone]`: Truncates the last coordinate dimension
- `PCProduct[a, b]`: Computes the Cartesian product cone $A \times B$
- `PCSpan[cone]`: Constructs the minimal linear subspace containing the cone

### A.8.6 Lattice & Algebraic Utilities

- `PCQuotientLatticeBasis[cone]`: Computes a $\mathbb{Z}$-basis for the quotient lattice $(\mathbb{Z}^n \cap \text{span})/(\mathbb{Z}^n \cap \text{lineality})$
- `PCSemiGroupGeneratorOfRay[cone]`: Extracts the unique primitive integer semigroup generator of a 1-dimensional ray
- `PCCheckDual[cone, c]`: Assertion helper verifying dual cone consistency
- `PCVolume[cone]`: Computes volume of the cone truncated to $[-1, 1]^n$

### A.8.7 Metadata, Printing & Comparison

- `PCGetLinearForm[cone]` / `PCSetLinearForm[cone, form]`: Manages linear form metadata
- `PCGetMultiplicity[cone]` / `PCSetMultiplicity[cone, m]`: Manages lattice multiplicity metadata
- `PCLess[a, b]` / `PCNotEqual[a, b]`: Lexicographical total ordering over State 3 cones
- `PCPrint[cone]` / `PCPrintAsFan[cone]`: Formatted text printing output

### A.8.8 Pending / Under Development Functions

- `PCHasFace[cone, f]` *(Being developed)*: Will test whether cone $f$ is a face of cone
- `PCTriangulation[cone]` *(Being developed)*: Will compute a simplicial triangulation of cone

# References


[1] Marisa C Eisenberg. Input-output equivalence and identifiability: some simple generalizations of the differential algebra approach. *arXiv preprint arXiv:1302.5484*, February 2013.

[2] Nicolette Meshkat, Marisa Eisenberg, and Joseph J Distefano, 3rd. An algorithm for finding globally identifiable parameter combinations of nonlinear ODE models using gröbner bases. *Math. Biosci.*, 222(2):61–72, December 2009.

[3] Komei Fukuda, Anders N Jensen, and Rekha R Thomas. Computing gröbner fans. *Math. Comput.*, 76(260):2189–2212, 2007.

[4] David A Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Undergraduate Texts in Mathematics. Springer Nature Switzerland, Cham, Switzerland, 5 edition, 2025.

[5] Teo Mora and Lorenzo Robbiano. The gröbner fan of an ideal. *J. Symbolic Comput.*, 6(2–3):183–208, 1988.

[6] Anders N. Jensen. *Gfan Version 0.7: A User's Manual*. Aarhus University, August 2024. Software manual.