Leibniz Universität Hannover
Fakultät für Mathematik und Physik
Institut für Algebraische Geometrie

in Kooperation mit
Technische Universität Kaiserslautern
Fraunhofer-Institut für Techno- und
Wirtschaftsmathematik

Computation of the GIT-fan using a massively parallel implementation

Masterarbeit

Hannover, 11.01.2018
Christian Reinbold
Matrikelnr.: 2942360

Erstprüfer: apl. Prof. Dr. Anne Frühbis-Krüger
Zweitprüfer: Dr. Janko Böhm

Betreuer: apl. Prof. Dr. Anne Frühbis-Krüger
Dr. Janko Böhm
Dr. Mirko Rahn

Abstract

The GIT fan of an algebraic group action on an algebraic variety describes all GIT quotients arising from Mumford's construction in Geometric Invariant Theory. In this thesis we enhance an implementation that computes the GIT fan for toric actions on affine varieties such that it may be executed on scalable hardware systems. For this purpose, we combine the parallelisation framework GPI-Space, developed at the *Fraunhofer-Institut für Techno- und Wirtschaftsmathematik*, and the computer algebra system Singular, developed at the *Technische Universität Kaiserslautern*. In doing so, we are able to compute the Mori chamber decomposition of $\mathrm{Mov}(\overline{M}_{0,6})$ in approximately 21 minutes, utilising 40 *Dell PowerEdge M620 Blade Servers* with 16 cores each. The previous implementation executed on a single machine terminated after a whole day.

Kurzzusammenfassung

Der GIT-Fächer einer algebraischen Gruppwirkung auf einer algebraischen Varietät beschreibt sämtliche GIT-Quotienten, die aus Mumfords Konstruktion im Bereich der geometrischen Invariantentheorie hervorgehen. In dieser Arbeit erweitern wir eine bereits bestehende Implementation zur Berechnung des GIT-Fächers für torische Gruppenwirkungen auf affinen Varietäten so, dass sie auf beliebig großen Rechnerverbunden parallel ausgeführt werden kann. Hierzu kombinieren wir das am *Fraunhofer-Institut für Techno- und Wirtschaftsmathematik* entwickelte Parallelisierungsframework GPI-Space mit dem Computeralgebrasystem Singular der *Technischen Universität Kaiserslautern*. Mit der neuen Implementation können wir die Zerlegung von $\mathrm{Mov}(\overline{M}_{0,6})$ in seine Mori-Kammern in rund 21 Minuten berechnen. Verwendet werden hierbei 40 *Dell PowerEdge M620 Blade Server* mit je 16 Kernen. Die ursprüngliche, auf einer Maschine ausgeführte Implementation terminiert nach einem Tag.

Acknowledgements

First of all I would like to thank apl. Professor Anne Frühbis-Krüger for establishing contact to Kaiserslautern and dealing with all the organisational issues emerging from writing a master's thesis in cooperation with remote institutes. Further, I also thank Doctor Janko Böhm for entrusting me with his research topic and giving a thorough introduction of the problem.

At the *Fraunhofer-Institut für Techno- und Wirtschaftsmathematik*, Doctor Mirko Rahn guided my efforts and has been available for any kind of technical questions. I thank him not only for this, but also for him and Bernd Lörwald reviewing probably every single line of code with meticulous precision, sharing their years of experience with me whenever I produced code making each reasonable software engineer cringe. I am especially grateful for Sonja Menßen who told me about her working on a thesis in cooperation with Fraunhofer. Without her, I never would have seized the opportunity of escaping Hannover and advancing my studies at the exciting Fraunhofer Society. I also thank her and Maya Thürnau for proofreading the upcoming pages and suffering from all my errors. As always, the own flaws are the hardest to detect.

Finally, I want to say thank you to Lukas Ristau who invited me to the *Pfalztheater* and thus made my stay at Kaiserslautern worthwile not only in working but also in cultural aspects.

Contents

1 Introduction

Algebraic group quotients are commonly used when constructing moduli spaces. A naïve moduli problem is a classification problem, that is a collection of objects together with an equivalence relation. Its moduli space sets up a one-to-correspondence of its points and the equivalence classes. The projective line $\mathbb{P}^1$ may be seen as a moduli space of the problem of classifying all nonzero points in the complex plane $\mathbb{C}^2$ such that two points are equivalent iff they are contained in the same line through the origin. The elements of a class are generated by scaling points by a nonzero scalar. In other words, each class is an orbit of an action of the torus $\mathbb{C}^*$ on the affine space $\mathbb{C}^2$ without the origin. The moduli space $\mathbb{P}^1$ is recovered by the orbit space. However, it is not equipped with a geometric structure by default. Thus, we obtain $\mathbb{P}^1$ as a set but loose all geometric data attached to it as a projective variety.

Geometric Invariant Theory (GIT) provides a method for creating algebraic group quotients with a geometric structure such that the quotient map becomes a morphism. In many instances, the classical orbit space cannot be equipped with such a structure. Consider the complex plane $\mathbb{C}^2$ with the torus action as above. Since the closure of each orbit – that is a line – contains the origin, every regular function being constant on orbits has to be constant on all $\mathbb{C}^2$. On that account, the ring of regular functions of the orbit space would be the ground field $\mathbb{C}$ and the orbit space would be a single point. Clearly, this is a contradiction. GIT circumvents this problem by removing problematic orbits beforehand. The remaining ones form an open subset of *semistable points* with a well behaving quotient that comes with a geometric structure. The quotient is *almost geometric*, meaning that there exists a dense, open subset such that the quotient becomes the orbit space for this subset. In our case, we have to remove the origin from $\mathbb{C}^2$ in order to obtain the orbit space $\mathbb{P}^1$ together with its geometric description.

Mumford's construction of subsets of semistable points depends on a lifting of the group action to ample line bundles where different liftings may yield distinct subsets. There exists a quasifan, called *GIT fan*, whose set of cones parametrises all subsets that may be obtained in this fashion. Given an affine variety together with a toric group action $H \circlearrowright X$, we focus on the computation of the GIT fan on

scalable hardware systems. Basing on the work of Boehm, Keicher, and Ren [10], we develop a cluster enabled application computing all chambers of the GIT fan of $H \circlearrowright X$. In doing so, we implement a fan traversal algorithm that is reusable in the context of tropical varieties [11] and Gröbner fans [32, chapter 3].

The thesis is structured as follows: In the preliminaries, we cover the mathematical background encompassed by GIT fans. We introduce algebraic group actions and good categorical quotients, outline Mumford's construction of GIT quotients arising from subsets of semistable points and prove that the GIT cones associated to them indeed form a quasifan with convex support. The next chapter elucidates the algorithm we are going to implement. We describe how to express GIT cones in terms of orbit cones and compute the latter ones. Then, we translate the task of computing the GIT fan into a graph traversal problem and show how symmetries in the geometry of X may be exploited in order to reduce the graph's size. Finally, we point out that Boehm, Keicher, and Ren incorporated an erroneous optimisation which we left out in our implementation. Fortunately, we were able to reproduce the results of their work without relying on the defect.

The fourth chapter covers the technical details of the cluster enabled implementation. Here, we introduce the utilised parallelisation framework GPI-Space and follow up with the integration of Singular code into GPI-Space applications. Furthermore, we present a high level description of our implementation in terms of Petri nets and document the various features of our application such as incorporating precomputed results. We finish off the chapter with a performance analysis for the computation of the Mori chamber decomposition of the cone of movable divisor classes of $\overline{M}_{0,6}$. The mathematical background of this example is outlined in the final chapter.

2 Preliminaries

In this chapter we present the mathematical background used in the thesis. Step by step, we progress towards the construction and meaning of the GIT fan. The first section covers the notion of algebraic group actions which lead to various concepts of quotients introduced in the second section. Afterwards, we provide the details for constructing GIT quotients according to Mumford [25]. Finally, a concrete description of available GIT quotients w.r.t. torus actions in terms of a quasifan is given. Computing this quasifan constitutes the main part of the thesis.

For the remainder of this chapter let $\mathbb{K}$ be an algebraically closed field.

2.1 Algebraic group actions

In this section we extend the notion of groups acting on sets to actions on objects in algebraic geometry. We only cover the case of affine varieties here. For a more general approach considering pre-schemes, see [25]. Broadly speaking, we require the group and the set operated on to carry structures of affine varieties such that the set-theoretical group action becomes a morphism.

Definition 2.1.1 (Affine algebraic groups) An *affine algebraic group* G is an affine variety together with morphisms $m\colon G\times G\to G$ and $\iota\colon G\to G$ such that G carries a group structure with the group law m and inversion ι. A *homomorphism* of algebraic groups G and H is a morphism $f\colon G\to H$ such that f preserves the group structure.

Example 2.1.2 The affine variety

$$(\mathbb{K}^*)^n \cong V(t_1t_1^{-1}-1,\dots,t_nt_n^{-1}-1)\subseteq \mathbb{K}[t_1,t_1^{-1},...,t_n,t_n^{-1}]$$

is an affine algebraic group via the group law

$$m\colon (\mathbb{K}^*)^n\times(\mathbb{K}^*)^n\to(\mathbb{K}^*)^n,\quad ((r_1,\dots,r_n),(s_1,\dots,s_n))\mapsto(r_1s_1,\dots,r_ns_n)$$

and the inversion

$$\iota\colon (\mathbb{K}^*)^n\to(\mathbb{K}^*)^n,\quad (r_1,\dots,r_n)\mapsto(r_1^{-1},\dots,r_n^{-1}).$$

Note that both m and ι are polynomial when formulated over $\mathbb{K}[t_1, t_1^{-1}, ..., t_n, t_n^{-1}]$. We claim that the comultiplication m^*, that is the pullback of the multiplication m, is given by

$$\mathcal{O}((\mathbb{K}^*)^n) \to \mathcal{O}((\mathbb{K}^*)^n) \otimes \mathcal{O}((\mathbb{K}^*)^n), \quad \mathbf{t}^\alpha \mapsto \mathbf{t}^\alpha \otimes \mathbf{t}^\alpha \quad \forall \alpha \in \mathbb{Z}^n.$$

Clearly, this is a $\mathbb{K}$-algebra homormophism. It suffices to check equality with m^* on a set of generators $\{\mathbf{t}^\alpha \mid \alpha \in \mathbb{Z}^n\}$. Now,

$$\mathbf{t}^\alpha \otimes \mathbf{t}^\alpha(P, Q) = P^\alpha \cdot Q^\alpha = (PQ)^\alpha = (\mathbf{t}^\alpha \circ m)(P, Q) = m^*(\mathbf{t}^\alpha)(P, Q)$$

holds for all $\alpha \in \mathbb{Z}^n$ and $P, Q \in (\mathbb{K}^*)^n$. The claim follows.

Definition 2.1.3 An affine algebraic group G is called *(algebraic) torus of dimension* n iff there exists $n > 0$ such that G is isomorphic to the affine algebraic group $(\mathbb{K}^*)^n$.

Definition 2.1.4 (Algebraic Group action) Let G be an affine algebraic group and X be an affine variety. An (algebraic) group action of G on X, denoted by $G \curvearrowright X$, is a morphism $\varphi\colon G \times X \to X$ such that φ defines a set-theoretic group action of G on X via

$$g \cdot x := \varphi(g, x).$$

Construction 2.1.5 We give a concrete description of an algebraic torus T of dimension n acting on an affine variety X via a morphism $\varphi\colon T \times X \to X$. It will turn out that group actions of n-dimensional tori on X correspond to $\mathbb{Z}^n$-gradings on $\mathcal{O}(X)$.

W.l.o.g. assume that $T = (\mathbb{K}^*)^n$ with the group law m as specified in Example 2.1.2. Note that the regular functions of T are given by $\mathbb{K}[\mathbf{t}, \mathbf{t}^{-1}]$ and the comultiplication φ^* is injective since any regular function of X vanishing on TX also vanishes on X by the identity axiom. Furthermore, we have

$$\mathcal{O}(T) \otimes \mathcal{O}(T) \otimes \mathcal{O}(X) \cong \mathcal{O}(X)[\mathbf{t}, \mathbf{t}^{-1}, \mathbf{u}, \mathbf{u}^{-1}] \quad \text{via} \quad \mathbf{t}^\alpha \otimes \mathbf{t}^\beta \otimes f \mapsto f\mathbf{t}^\alpha \mathbf{u}^\beta.$$

For $\alpha \in \mathbb{Z}^n$, define

$$\mathcal{O}(X)_\alpha := \{f \in \mathcal{O}(X) \mid \varphi^*(f) = \mathbf{t}^\alpha \otimes f\},$$

which is a submodule of $\mathcal{O}(X)$. Let $\sum_{\alpha \in I} f_\alpha$ be a finite linear combination of zero with $f_\alpha \in \mathcal{O}(X)_\alpha$. It follows that

$$0 = \varphi^*(0) = \varphi^*\left(\sum_{\alpha \in I} f_\alpha\right) = \sum_{\alpha \in I} \varphi^*(f_\alpha) = \sum_{\alpha \in I} \mathbf{t}^\alpha \otimes f_\alpha.$$

Coefficient comparison in $\mathcal{O}(X)[\mathbf{t}, \mathbf{t}^{-1}, \mathbf{u}, \mathbf{u}^{-1}]$ shows that $f_\alpha = 0$. Hence, the sum

$$\bigoplus_{\alpha \in \mathbb{Z}^n} \mathcal{O}(X)_\alpha \subseteqq \mathcal{O}(X)$$

is direct.

Let $f \in \mathcal{O}(X)$ and write $\varphi^*(f)$ as a finite linear combination

$$\varphi^*(f) = \sum_{\alpha \in I} \mathbf{t}^\alpha \otimes f_\alpha.$$

When moving to $\mathbb{K}$-algebra homomorphisms, the compatibility axiom translates to

$$(m^* \otimes \mathrm{id}_{\mathcal{O}(X)}) \circ \varphi^* = (\mathrm{id}_{\mathcal{O}(T)} \otimes \varphi^*) \circ \varphi^*.$$

Hence, we obtain

$$\sum_{\alpha \in I} \mathbf{t}^\alpha \otimes \mathbf{t}^\alpha \otimes f_\alpha = (m^* \otimes \mathrm{id}_{\mathcal{O}(X)}) \circ \varphi^*(f) = (\mathrm{id}_{\mathcal{O}(T)} \otimes \varphi^*) \circ \varphi^*(f) = \sum_{\alpha \in I} \mathbf{t}^\alpha \otimes \varphi^*(f_\alpha).$$

Comparing coefficients yields $\varphi^*(f_\alpha) = \mathbf{t}^\alpha \otimes f_\alpha$, that is $f_\alpha \in \mathcal{O}(X)_\alpha$. Furthermore,

$$\varphi^*\left(\sum_{\alpha \in I} f_\alpha\right) = \sum_{\alpha \in I} \varphi^*(f_\alpha) = \sum_{\alpha \in I} \mathbf{t}^\alpha \otimes f_\alpha = \varphi^*(f).$$

φ^* being injective implies that

$$f = \sum_{\alpha \in I} f_\alpha,$$

proving that

$$\bigoplus_{\alpha \in \mathbb{Z}^n} \mathcal{O}(X)_\alpha = \mathcal{O}(X).$$

Since φ^* preserves the multiplication of $\mathcal{O}(X)$, it holds that $\mathcal{O}(X)_\alpha \mathcal{O}(X)_\beta \subseteq \mathcal{O}(X)_{\alpha+\beta}$. Hence, $\mathcal{O}(X)$ becomes a $\mathbb{Z}^n$-graded $\mathbb{K}$-algebra. Conversely, every $\mathbb{Z}^n$-grading on $\mathcal{O}(X)$ determines a group action ψ of T on X with the corresponding coaction defined by

$$\psi^*(f) = \mathbf{t}^{\deg(f)} \otimes f \quad \forall f \in \mathcal{O}(X) \text{ homogeneous.}$$

By taking a finite set $\{g_1, \ldots, g_m\}$ of $\mathbb{Z}^n$-homogeneous generators of $\mathcal{O}(X)$, we obtain an embedding $X \subseteq \mathbb{K}^m$ such that the vanishing ideal $\mathfrak{a} := I(X) \subseteq \mathbb{K}[x_1, \ldots, x_m]$ of X is homogeneous w.r.t. the grading

$$\deg(x_i) = \deg(g_i) \in \mathbb{Z}^n$$

on $\mathbb{K}[x_1,\ldots,x_m]$. By construction, we have $\varphi^*(g_i) = \mathbf{t}^{\deg(g_i)} \otimes g_i$ which translates to the coaction $\psi^*(x_i) = \mathbf{t}^{\deg(g_i)} \otimes x_i$ in $\mathbb{K}[x_1,\ldots,x_m]/\mathfrak{a}$. The group action is recovered as follows:

$$T \times X \longrightarrow X, \quad (t,P) \mapsto (\psi^*(x_1)(t,P),\ldots,\psi^*(x_m)(t,P)) = (t^{\deg(g_1)}P_1,\ldots,t^{\deg(g_m)}P_m).$$

We conclude that any action of a torus of dimension n on an affine variety X may be described by a set of integral vectors $q_i \in \mathbb{Z}^n$, $1 \le i \le m$, and a suitable embedding of X in $\mathbb{K}^m$ such that its vanishing ideal is homogeneous w.r.t.

$$\deg(x_i) = q_i, \quad 1 \le i \le m.$$

The group action is given by

$$(\mathbb{K}^*)^n \times X \longrightarrow X, \quad (t,P) \mapsto (t^{q_1}P_1,\ldots,t^{q_m}P_m).$$

Remark 2.1.6 Let G be an affine algebraic group acting on an affine variety X. Let $x \in X$. The stabiliser subgroup G_x is a closed, affine subvariety since it is the preimage of the closed point $\{x\}$ of the morphism $g \mapsto gx$. It follows that G_x is an affine algebraic subgroup.

Remark 2.1.7 (Group action on regular functions) If φ is a group action of an affine algebraic group G on an affine variety X, we obtain a group action of G on the regular functions $K(X)$ of X by

$$G \times K(X) \to K(X), \quad (g, \frac{f}{h}) \mapsto \frac{f \circ \varphi(g^{-1}, \cdot\,)}{h \circ \varphi(g^{-1}, \cdot\,)}.$$

If U is a G-invariant open subset of X, that is $GU = U$, the restriction of $G \circlearrowright K(X)$ to $\mathcal{O}_X(U)$ yields a group action $G \circlearrowright \mathcal{O}_X(U)$. Note that if the denominator h does not vanish on any point in U, then the same holds for $h \circ \varphi(g^{-1}, \cdot\,)$ as we have

$$\varphi(g^{-1}, \cdot\,)(U) \subseteq GU = U.$$

We say that $f \in \mathcal{O}_X(U)$ is *G-invariant* iff $gf = f$ holds for all $g \in G$. The subalgebra of G-invariant functions is denoted by $\mathcal{O}_X(U)^G$.

Example 2.1.8 Consider an algebraic group action $\varphi\colon T \times X \to X$ of a torus T of dimension n on an affine variety X. By Construction 2.1.5, $\mathcal{O}(X)$ is a $\mathbb{Z}^n$-graded $\mathbb{K}$-algebra. For any homogeneous $f \in \mathcal{O}(X)$ and $g \in (\mathbb{K}^*)^n$, we have

$$(g^{-1}f)(x) = f \circ \varphi(g,x) = \varphi^*(f)(g,x) = (\mathbf{t}^{\deg(f)} \otimes f)(g,x) = g^{\deg(f)} \cdot f(x) \quad \forall x \in X.$$

In particular, it is easy to see that

$$g^{-1}f = g^{\alpha}f \quad \forall g \in (\mathbb{K}^*)^n \iff f \in \mathcal{O}(X)_{\alpha},$$

where $\alpha \in \mathbb{Z}^n$, the left multiplication is given by the group action and the right one is the scalar multiplication of $\mathcal{O}(X)$ as $\mathbb{K}$-module. For this reason it follows that

$$\mathcal{O}(X)^T = \mathcal{O}(X)_0.$$

2.2 Good quotients

When an algebraic group G acts on an affine variety X, naturally one questions if the orbit space X/G defined by the group action can be equipped with the structure of an variety such that the quotient map $p\colon X \to X/G$ becomes a morphism. Consider the following example.

Example 2.2.1 Let $G = \mathbb{C}^*$ be a one dimensional algebraic torus action on the affine variety $X = \mathbb{C}^2$ via scaling, that is

$$(t, (x_1, x_2)) \mapsto (tx_1, tx_2).$$

Each line l through the origin in $\mathbb{C}^2$ yields an orbit when removing 0. The origin is G-invariant and thus an orbit on its own. If the set-theoretical quotient map p would be a morphism, a regular function φ in $\mathcal{O}(X/G)$ induces a regular function $\psi = \varphi \circ p$ in $\mathcal{O}(X)$ that is G-invariant. In particular, ψ has to be constant on the closure of each orbit. Since each closure contains the origin, ψ and thus φ have to be constant. It follows that $\mathcal{O}(X/G) \cong \mathbb{K}$, contradicting $|X/G| > 1$. Consequently, the set-theoretical quotient map is no good quotient in the following sense.

Definition 2.2.2 (Good quotient, [13, Definition 5.0.5]) Let G be an affine algebraic group acting on an affine variety X. A G-invariant morphism $p\colon X \to Y$ is called a *good (categorial) quotient*, if

(i) the pullback $p^*\colon \mathcal{O}_Y \to (p^*\mathcal{O}_X)^G$ is an isomorphism of sheaves, that is, for any open $U \subseteq Y$ the pullback $p^*\colon \mathcal{O}_Y(U) \to \mathcal{O}_X(p^{-1}(U))^G$ is an isomorphism of $\mathbb{K}$-algebras (note that preimages of p are G-invariant),

(ii) $p(W)$ is closed for any G-invariant, closed $W \subseteq X$,

(iii) the images of disjoint G-invariant, closed subsets are disjoint.

If $\mathcal{O}(X)^G$ is finitely generated, the canonical embedding $\mathcal{O}(X)^G \hookrightarrow \mathcal{O}(X)$ yields a good quotient $p\colon X \to Y$, where Y is the affine variety with coordinate ring $\mathcal{O}(X)^G$.

The question of $\mathcal{O}(X)^G$ always being finitely generated is addressed by Hilbert's fourteenth problem and has been answered negatively by Nagata. [15, chapter 4]

Good quotients admit the following useful properties.

Proposition 2.2.3 ([13, Theorem 5.0.6 & Proposition 5.0.7]) *Let G, X be as above. Let $p\colon X \to Y$ be a good quotient. Then it holds that*

(i) *p is surjective,*

(ii) *for any G-invariant morphism $\varphi\colon X \to Z$, there exists a unique morphism $\psi\colon Y \to Z$ such that $\varphi = \psi \circ p$,*

(iii) *the topology on Y coincides with the quotient topology induced by p,*

(iv) *p separates closed G-orbits, that is*

$$p(x) \neq p(y) \;\Leftrightarrow\; \overline{Gx} \cap \overline{Gy} = \emptyset$$

for all points $x, y \in X$.

(v) *Each fiber of p contains a unique closed G-orbit. In particular, p induces a bijection*

$$\{Gx \mid x \in X \text{ such that } Gx = \overline{Gx}\} \simeq Y.$$

Indeed, good quotients are categorical quotients by (ii) and the good quotient space Y is unique up to isomorphism. For this reason we also write $X /\!\!/ G$ for Y. Note that by (v), in general $X /\!\!/ G$ does not coincide with the orbit space X/G as we loose all non-closed orbits. In the setting of Example 2.2.1, we have $X /\!\!/ G = \{\mathrm{pt}\}$ and $X/G = \mathbb{P}^1 \cup \{0\}$. This observation motivates the definition of the following class of good quotients.

Definition 2.2.4 (Geometric quotient) Let G, X be as above. A good quotient $p\colon X \to X /\!\!/ G$ is called *geometric quotient* if all G-orbits in X are closed. In that case we write X/G for $X /\!\!/ G$.

By Proposition 2.2.3(v) geometric quotients precisely are the set-theoretical quotient maps that are good quotients. Almost geometric quotients are those good quotients that yield a geometric quotient when restricting to a suitable G-invariant Zariski dense open subset of X. By [13, Proposition 5.0.11], this is equivalent to the following definition.

Definition 2.2.5 (Almost geometric quotient) Let G, X be as above. A good quotient $p\colon X \to X /\!\!/ G$ is called *almost geometric quotient*, if there exists a G-invariant Zariski dense open subset U of X such that all G-orbits in U are closed in X.

Note that the good quotient arising from Example 2.2.1 is no almost geometric quotient. The maximal set containing closed orbits is given by $\{0\}$, which is not dense. On that account the good quotient space $\mathbb{C}^2 /\!\!/ \mathbb{C}^* = \{\text{pt}\}$ is no appropriate replacement for $\mathbb{C}^2/\mathbb{C}^* = \mathbb{P}^1$.

2.3 GIT quotients

GIT is a broad field in mathematics that targets the construction of quotients with respect to group actions in algebraic geometry. It originates from Mumford's work in 1965, see [25]. By lifting the action of G to an ample line bundle on X, Mumford has been able to define an open subset of semistable points such that its good quotient space becomes projective. If so called stable points exist, the resultant quotient becomes an almost geometric quotient.

In this section, we introduce the concept of linearised line bundles and give the construction of the set of semistable points arising from an ample, linearised line bundle. Note that as in the previous sections, we only cover the affine case here.

Definition 2.3.1 (Vector bundle, [13, Definition 6.0.14]) An affine variety V is a *vector bundle of rank r* over an affine variety X if there is a morphism $\pi\colon V \to X$ and an open cover $\{U_i\}_{i\in I}$ of X such that

- for every $i \in I$, there is an isomorphism

  $$\phi_i\colon \pi^{-1}(U_i) \xrightarrow{\sim} U_i \times \mathbb{K}^r$$

  such that ϕ_i followed by the projection onto U_i is $\pi\big|_{\pi^{-1}(U_i)}$.
- for every pair $(i,j) \in I^2$, there is a *transition function* $g_{ij} \in \mathrm{GL}_r(\mathcal{O}_X(U_i \cap U_j))$ such that

  $$(p_j,\ g_{ij}) \circ \phi_j\big|_{\pi^{-1}(U_i\cap U_j)} = \phi_i\big|_{\pi^{-1}(U_i\cap U_j)}$$

  where p_j is the projection from $U_j \times \mathbb{K}^r$ to U_j.

The data $\{U_i, \phi_i\}_{i\in I}$ is called *trivialisation*. For $p \in U_i$, we call $\pi^{-1}(p)$ the *fiber* over p. If p is contained in $U_i \cap U_j$, we have

$$\mathbb{K}^r \cong \{p\} \times \mathbb{K}^r \xleftarrow{\phi_i} \pi^{-1}(p) \xrightarrow{\phi_j} \{p\} \times \mathbb{K}^r \cong \mathbb{K}^r$$

via the linear map $g_{ij}(p) \in \mathrm{GL}_r(\mathbb{K})$, so that the fiber $\pi^{-1}(p)$ has a well-defined vector space structure, hence the name vector bundle.

A *line bundle* over X is a vector bundle of rank 1 over X. The *trivial line bundle* $X \times \mathbb{K}$ is given by the projection onto X where the isomorphism ϕ_i is given by the

identity function on $U_i \times \mathbb{K}$ and the transition function g_{ij} is given by the 1×1 matrix containing $1 \in \mathcal{O}_X(X)$.

Remark 2.3.2 Let Y be an affine variety and $\pi\colon V \to X$ be a vector bundle over an affine variety X with trivialisation $\{U_i, \phi_i\}_{i\in I}$. Then

$$\mathrm{id}_Y \times \pi\colon Y \times V \longrightarrow Y \times X$$

is a vector bundle over $Y \times X$ with trivialisation $\{Y \times U_i, \mathrm{id}_Y \times \phi_i\}_{i\in I}$. If the transition functions on V are denoted by g_{ij}, the transition functions on $Y \times V$ are given by $g'_{ij} \in \mathrm{GL}_r(\mathcal{O}_{Y\times X}((Y \times U_i) \cap (Y \times U_j)))$ such that

$$g'_{ij}(y,p) = g_{ij}(p) \quad \forall (y,p) \in (Y \times U_i) \cap (Y \times U_j).$$

Definition 2.3.3 (Section, [13, Definition 6.0.15]) A *section* of a vector bundle $\pi\colon V \to X$ over an open subset $U \subseteq X$ is a morphism

$$s\colon U \longrightarrow V$$

such that $\pi \circ s = \mathrm{id}_U$. The $\mathbb{K}$-linear space of all sections of V over U is denoted by $\Gamma(U, V)$. A *global section* is a section over X.

Definition 2.3.4 (Vector bundle morphism) Let $\pi_1\colon V_1 \to X_1$ and $\pi_2\colon V_2 \to X_2$ be vector bundles over affine varieties X_1 and X_2 respectively. Then a *vector bundle morphism* from V_1 to V_2 is a pair (f, g) of morphisms $f\colon V_1 \to V_2$ and $g\colon X_1 \to X_2$ such that

$$\begin{array}{ccc} V_1 & \xrightarrow{\ f\ } & V_2 \\ {\scriptstyle \pi_1}\downarrow & & \downarrow{\scriptstyle \pi_2} \\ X_1 & \xrightarrow{\ g\ } & X_2 \end{array}$$

commutes and f is fiberwise linear. That is, for every $x \in X_1$ the restriction of f to $\pi_1^{-1}(x)$ is a linear map between the vector spaces $\pi_1^{-1}(x)$ and $\pi_2^{-1}(g(x))$.

Note that g is uniquely determined by f.

Definition 2.3.5 (G-linearisation, [25, chapter 1.3]) Let G be an affine algebraic group acting on an affine variety X. Let $\pi\colon L \to X$ be a line bundle over X. A *G-linearisation of L* is a morphism (f, h) from the line bundle $G \times L$ to L such that

$$f\colon G \times L \longrightarrow L$$

is an algebraic group action and

$$\begin{array}{ccc} G \times L & \xrightarrow{f} & L \\ {\scriptstyle \mathrm{id}_G \times \pi}\downarrow & & \downarrow{\scriptstyle \pi} \\ G \times X & \xrightarrow{(g,x) \mapsto gx} & X \end{array}$$

commutes. A *G-linearised line bundle* is a line bundle L together with a G-linearisation of L.

Remark 2.3.6 The tensor product $L_1 \otimes L_2$ of G-linearised line bundles L_1 and L_2 over X with G-linearisations f_1, f_2 is the G-linearised line bundle that is obtained by taking the tensor product on the fibers. Its group action is given by tensoring the actions f_1 and f_2 restricted to the fibers. Note that $L_1^{\otimes n}$ is isomorphic to the line bundle L_1 with the G-linearisation

$$G \times L_1 \longrightarrow L_1, \quad (g, \ell) \mapsto f_1(g^n, \ell)$$

Remark 2.3.7 (Group action on sections, [6, Remark 1.7]) Let L be a G-linearised line bundle over X. For each G-invariant open subset $U \subseteq X$ we obtain a group action of G on the space $\Gamma(U, L)$ of sections over U by setting

$$(g \cdot s)(x) := g \cdot s(g^{-1}x).$$

The identity and compatibility axioms are verified easily. We check that $g \cdot s$ again is a section in $\Gamma(U, L)$. It is well defined on U since we have $g^{-1}x \in U$ for all $g \in G$, $x \in U$. Clearly, $g \cdot s$ is the composition of morphisms and thus a morphism itself.

The subspace of G-invariant sections of L over U is denoted by $\Gamma(U, L)^G$.

Now we are ready to describe the construction of semistable and stable points in accordance with [25, Definition 1.7 & 1.8]. Note that for us, stable points mean properly stable points in the sense of Mumford.

Definition 2.3.8 Let L be a G-linearised, ample line bundle over X. A point $x \in X$ is called

(i) *semistable* if there exists a G-invariant global section $s \in \Gamma(X, L^{\otimes n})^G$ for some $n \in \mathbb{N}$ such that $s(x) \neq 0$, where 0 is the origin in the fiber $\pi^{-1}(x)$.

(ii) *stable* if its stabiliser G_x is finite and it is semistable via a global section s such that all G-orbits in

$$X_s = X \setminus V(s) = \{p \in X \mid s(p) \neq 0\}$$

are closed.

The sets of semistable points and stable points are denoted by $X^{ss}(L)$ and $X^{s}(L)$ respectively.

Remark 2.3.9 By Mumford, the G-invariant set $X^{ss}(L)$ permits a good quotient $X^{ss}(L) /\!\!/ G$, called *GIT quotient*. It is the projective variety of the graded ring

$$\bigoplus_{n\geq 0} \Gamma(X, L^{\otimes n})^G.$$

If $X^{s}(L)$ is non-empty, $X^{ss}(L) /\!\!/ G$ is an almost geometric quotient with the restriction to $X^{s}(L)$ being a geometric quotient.

2.4 The GIT fan

The set of semistable points $X^{ss}(L)$ and the corresponding GIT quotient depend on the choice of a G-linearised line bundle. As we will see in this section, we are able to parametrise all nonempty sets of semistable points arising from linearisations of the trivial line bundle by a special quasifan, called GIT fan, whose cones are in inclusion-reversing one-to-one correspondence. We assume that the group G is an algebraic torus acting on an affine variety X. Note that the general case of arbitrary connected reductive groups G may be reduced to the torus case, see [3, Lemma 3.3]. The concepts presented here originate from [6, chapter 2].

For the remainder of this section we assume that X is an affine variety with a $\mathbb{Z}^k$-grading on its ring $\mathcal{O}(X)$ of regular functions. It encodes an action of a k-dimensional torus T by Construction 2.1.5.

Lemma 2.4.1 ([6, Lemma 2.6]) *The T-linearisations of the trivial line bundle*

$$\pi\colon X \times \mathbb{K} \to X$$

correspond one-to-one to vectors in $\mathbb{Z}^k$, where $w \in \mathbb{Z}^k$ defines the G-linearisation

$$t \cdot (x, z) := (t \cdot x, t^w z). \tag{2.1}$$

Proof: One easily computes that (2.1) defines a T-linearisation for all $w \in \mathbb{Z}^k$. Now let $f : T \times X \times \mathbb{K} \to X \times \mathbb{K}$ be an arbitrary T-linearisation. Let p_X and $p_{\mathbb{K}}$ be the projections of $X \times \mathbb{K}$ to X and $\mathbb{K}$ respectively. Since f is defined such that π is equivariant, it follows that $p_X \circ f(t, x, z) = tx$. Furthermore, $p_{\mathbb{K}} \circ f(t, x, z)$ is linear w.r.t. z. For this reason there exists a morphism $c\colon T \times X \to \mathbb{K}$ such that

$$p_{\mathbb{K}} \circ f(t, x, z) = c(t, x)z.$$

Let $x \in X$. The compatibility axiom yields

$$\begin{aligned} c(t_1 t_2, x) &= p_{\mathbb{K}} \circ f(t_1 t_2, x, 1) \\ &= p_{\mathbb{K}}(f(t_1, f(t_2, x, 1))) \\ &= p_{\mathbb{K}}(f(t_1, t_2 x, c(t_2, x))) \\ &= c(t_1, t_2 x) \cdot c(t_2, x). \end{aligned} \tag{2.2}$$

Due to the identity axiom and (2.2)

$$1_{\mathbb{K}} = c(1_T, x) = c(t^{-1}, tx) \cdot c(t, x) \quad \forall (t, x) \in T \times X$$

holds. In particular, c does not vanish somewhere on $T \times X$, so that $c(\,\cdot\,, x)$ becomes an invertible, rational function for every $x \in X$. By [27, Proposition 3], $c(\,\cdot\,, x)$ is a character of T. As every character of T is given by $t \mapsto t^\alpha$ for a suitable $\alpha \in \mathbb{Z}^d$ (see [22, §16]), we conclude that

$$c = \sum_{\alpha \in I} t^\alpha \otimes f_\alpha \quad \text{with suitable } f_\alpha \in \mathcal{O}(X) \setminus \{0\}$$

where for every $x \in X$ there exists exactly one $\alpha \in I$ such that $f_\alpha(x) = 1$ and $f_\beta(x) = 0$ for all $x \in I \setminus \{\alpha\}$.

If $|I| > 1$, we would obtain

$$\bigcup_{\alpha \in I} V(f_\alpha) = X,$$

contradicting the irreducibility of X. Thus, I has to contain a single element which we denote by w. It follows that $c = t^w \otimes 1$. □

Notation 2.4.2 Let $w \in \mathbb{Z}^k$. We denote the G-linearised trivial line bundle corresponding to w by L_w and write $X(w)$ for the set of semistable points $X^{ss}(L_w)$.

Now that we have identified all GIT quotients arising from T-linearisations, we are going to complement them by a combinatorical description in terms of cones. A key role is assigned to orbit cones, from which finitely many exist. These contain all characters $w \in \mathbb{Z}^k$ of T that permit a global section invariant w.r.t. to L_w not vanishing on a fixed point $x \in X$. Non-empty intersections of orbit cones, called GIT cones, represent all non-empty sets of semistable points. Most interestingly, the collection of GIT cones form a quasifan with convex support. We exploit this fact later when giving an algorithmic solution for determining all GIT cones.

Definition 2.4.3 (Orbit cone, [6, Definition 2.1]) Let $x \in X$. The *orbit monoid* of x is given by

$$S_T(x) = \{w \in \mathbb{Z}^k \mid \exists f \in \mathcal{O}(X)_w \colon f(x) \neq 0\}.$$

The convex, polyhedral cone $\omega_T(x)$ generated by $S_T(x)$ is called *orbit cone* of x.

The *weight cone* of X is defined by

$$\Omega_T(X) = \langle w \in \mathbb{Z}^k \mid \mathcal{O}(X)_w \neq 0 \rangle_{\mathbb{Q}_{\geq 0}}.$$

Clearly, the union over all orbit cones is $\Omega_T(X)$.

Lemma 2.4.4 *Let $x \in X$. Then $S_T(x)$ indeed is a submonoid of $\mathbb{Z}^k$.*

Proof: Obviously, $0 \in S_T(x)$ by considering 1 in $\mathcal{O}(X)$. It remains to check that $S_T(x)$ is closed. Let $w_1, w_2 \in S_T(x)$. Hence, there exist $f_1 \in \mathcal{O}(X)_{w_1}$ and $f_2 \in \mathcal{O}(X)_{w_2}$ with $f_1(x) \neq 0 \neq f_2(x)$. The product $f_1 f_2$ is homogeneous of degree $w_1 + w_2$ with $(f_1 f_2)(x) \neq 0$. We conclude that $w_1 + w_2 \in S_T(x)$. □

Proposition 2.4.5 ([6, Proposition 2.5]) *The set of orbit cones $\{\omega_T(x) \mid x \in X\}$ is finite.*

Proof: By Construction 2.1.5, we can embed X into a suitable affine space $\mathbb{K}^m$ such that $\mathcal{O}(X)$ becomes the ring $\mathbb{K}[\mathbf{x}]/I(X)$ with grading

$$\deg(x_i) = q_i \in \mathbb{Z}^k \quad \forall 1 \leq i \leq m.$$

Let $x \in X$ and $w \in \mathbb{Z}^k$ such that $w \in S_T(x)$. Then there exists a polynomial $f \in \mathbb{K}[\mathbf{x}]$ of degree w with $f(x) \neq 0$. It follows that f contains at least one monomial $\mathbf{x}^\alpha$ of degree w that does not vanish when substituting x. In particular

$$(t *_{(\mathbb{K}^*)^m} x)^\alpha = t^\alpha \cdot x^\alpha \neq 0$$

for $t \in (\mathbb{K}^*)^m$, where $*_{(\mathbb{K}^*)^m}$ denotes the standard action of $(\mathbb{K}^*)^m$ on $\mathbb{K}^m$. We conclude that $S_T(x)$ is constant on $(\mathbb{K}^*)^m$-orbits over $\mathbb{K}^m$. Since there exist only a finite number of such orbits, the claim follows. □

Lemma 2.4.6 *Let $w \in \mathbb{Z}^k$. Then it holds that*

$$\begin{aligned} X(w) &= \bigcup_{f \in \mathcal{O}(X)_{nw},\ n \in \mathbb{N}} X_f \\ &= \{x \in X \mid w \in \omega_T(x)\}. \end{aligned}$$

Proof: By Remark 2.3.6, we have $L_w^{\otimes n} = L_{nw}$ for all $n \in \mathbb{N}$. Now fix $n \in \mathbb{N}$ and let $s \in \Gamma(X, L_{nw})$ be a global section. Since we consider the (ample) trivial line bundle, s corresponds to a morphism $f \in \mathcal{O}(X)$ such that

$$s(x) = (x, f(x)) \quad \forall x \in X.$$

and X_f contains all points with $s(x) \neq 0$. The section is G-invariant iff

$$(tx, (t^{-1}f)(x)) = (tx, f(tx)) = s(tx) = (t \cdot s)(tx) = t \cdot s(x) = (tx, t^{nw} f(x))$$

for all $t \in T$, $x \in X$. Taking Example 2.1.8 into account, we obtain

$$s \in \Gamma(X, L_{nw})^G \iff f \in \mathcal{O}(X)_{nw}$$

and the first equation follows. The inclusion $\subseteq$ of the second equation is obvious. For the other direction let $x \in X$ such that $w \in \omega_T(x)$. After eliminating denominators in a $\mathbb{Q}$-linear combination of w over $S_T(x)$, one obtains $n \in \mathbb{N}$ such that $nw \in S_T(x)$. The inclusion $\supseteq$ follows. □

Corollary 2.4.7 *Let $w \in \mathbb{Z}^k$. The following conditions are equivalent:*

(i) $X(w) \neq \emptyset$,

(ii) $w \in \Omega_T(X)$.

Proof: Immediate consequence of Lemma 2.4.6 and the definition of the weight cone $\Omega_T(X)$. □

Definition 2.4.8 (GIT cone, [6, Definition 2.8]) Let $w \in \Omega_T(X)$. The *GIT cone* associated to w is defined by

$$\sigma_T(w) := \bigcap_{w \in \omega_T(x)} \omega_T(x).$$

The collection of all GIT cones

$$\Sigma_T(X) := \{\sigma_T(w) \mid w \in \Omega_T(X)\}$$

is called *GIT fan*.

Proposition 2.4.9 ([6, Proposition 2.9]) *Let $w_1, w_2 \in \Omega_T(x)$. Then*

$$X(w_1) \subseteq X(w_2) \iff \sigma_T(w_1) \supseteq \sigma_T(w_2).$$

In particular, $X(w_1) = X(w_2)$ if and only if $\sigma_T(w_1) = \sigma_T(w_2)$.

Proof: By Lemma 2.4.6 and the definition of GIT cones, for any $v, w \in \mathbb{Z}^k$ we have

$$\begin{aligned} v \in \sigma_T(w) &\Leftrightarrow v \in \omega_T(x) \quad \forall x \in X \text{ with } w \in \omega_T(x) \\ &\Leftrightarrow v \in \omega_T(x) \quad \forall x \in X(w). \end{aligned}$$

The direction $\Rightarrow$ follows. For the converse, consider $X(w_1) \not\subseteq X(w_2)$ so that there exists $x \in X$ with $w_1 \in \omega_T(x)$ and $w_2 \notin \omega_T(x)$ by Lemma 2.4.6. Since $\sigma_T(w_1)$ is a subset of $\omega_T(x)$, we conclude that $w_2 \notin \sigma_T(w_1)$ and therefore $\sigma_T(w_2) \not\subseteq \sigma_T(w_1)$.

The postscript immediately follows from the first claim. □

Theorem 2.4.10 ([6, Theorem 2.11]) *The GIT fan $\Sigma_T(X)$ is a quasifan, that is a finite collection of convex polyhedral cones located in the same ambient space such that for $\sigma, \sigma' \in \Sigma_T(X)$*

(i) *all faces of σ belong to $\Sigma_T(X)$,*

(ii) *$\sigma \cap \sigma'$ is a face of both σ and σ'.*

Its support

$$|\Sigma_T(X)| := \bigcup_{\sigma \in \Sigma_T(X)} \sigma$$

is given by the (convex) weight cone $\Omega_T(X)$.

We postpone the proof and begin with some lemmas. Since the closure of a T-orbit in X is T-invariant, it decomposes into a set of (possibly non-closed) T-orbits. The following lemma translates this setup into the world of orbit cones. We omit the proof and refer to [6].

Lemma 2.4.11 ([6, Corollary 2.4]) *The T-orbits in X are in one-to-one correspondence with orbit cones via*

$$Ty \mapsto \omega_T(y).$$

Let $x \in X$. Then all faces of $\omega_T(x)$ are orbit cones such that $\omega_T(y)$ is a face if and only if the corresponding T-orbit Ty is located in $\overline{Tx}$.

Notation 2.4.12 Let $\sigma \subseteq \mathbb{Q}^k$ be a cone. We write σ° for the relative interior of σ. Note that if σ is of full dimension k, then the interior of σ w.r.t. the euclidean topology coincides with the relative interior σ°.

Lemma 2.4.13 ([6, Lemma 2.12]) *Let $w \in \Omega_T(X)$. Then the following properties hold:*

(i) $w \in \omega_T(x) \Leftrightarrow \sigma_T(w) \subseteq \omega_T(x) \quad \forall x \in X$,

(ii) $w \in \omega_T(x)^\circ \Leftrightarrow \sigma_T(w)^\circ \subseteq \omega_T(x)^\circ \quad \forall x \in X$,

(iii) $\sigma_T(w) = \bigcap_{w \in \omega_T(x)^\circ} \omega_T(x)$,

(iv) $\sigma_T(w)^\circ = \bigcap_{w \in \omega_T(x)^\circ} \omega_T(x)^\circ$,

(v) $w \in \sigma_T(w)^\circ$.

Proof: (i) immediately follows from the definition of GIT cones.

For any orbit cone $\omega_T(x)$ with $w \in \omega_T(x)$ there exists a minimal face $\omega \preceq \omega_T(x)$ containing w in its relative interior. Since ω is an orbit cone by Lemma 2.4.11, it suffices to intersect with orbit cones containing w in its interior, showing (iii).

(iv) and (v) are a consequence of the following fact: If the intersection of the relative interiors of a finite number of convex polyhedral cones is non-empty, then it equals the relative interior of the intersection of the cones. (ii) follows from (iv) and (v). □

Lemma 2.4.14 ([6, Proof of Theorem 2.11]) *Let $w_1, w_2 \in \Omega_T(X)$. Then the inclusion $\sigma_T(w_1) \subseteq \sigma_T(w_2)$ implies $\sigma_T(w_1) \preceq \sigma_T(w_2)$.*

Proof: Let $\omega_T(x_1), \ldots, \omega_T(x_r)$ be the orbit cones with $\sigma_T(w_1)^\circ \subseteq \omega_T(x_i)^\circ$ for $1 \le i \le r$. By Lemma 2.4.13, we have $\sigma_T(w_1) = \omega_T(x_1) \cap \cdots \cap \omega_T(x_r)$.

Choose representatives $x_1, y_1, \ldots, y_\ell$ of the T-orbit decomposition of $\overline{Tx_1}$. Then Lemma 2.4.11 states that $\omega_T(y_i) \preceq \omega_T(x_1)$ is a proper face for all $1 \le i \le \ell$. Because of $\sigma_T(w_1)^\circ \subseteq \omega_T(x_1)^\circ$, it follows that $\omega_T(y_i) \cap \sigma_T(w_1)^\circ = \emptyset$. By Lemma 2.4.13, w_1 is contained in $\sigma_T(w_1)^\circ$ but not in $\omega_T(y_i)$. For this reason $y_i \notin X(w_1)$ holds. Together with $x_1 \in X(w_1)$ due to $w_1 \in \sigma_T(w_1)^\circ \subseteq \omega_T(x_1)^\circ$, we conclude that

$$\overline{Tx_1} \cap X(w_1) = Tx_1, \tag{2.3}$$

i.e. Tx_1 is closed in $X(w_1)$.

By Proposition 2.4.9, we have $X(w_2) \subseteq X(w_1)$. We denote the good quotient maps of $X(w_1)$ and $X(w_2)$ by p_1 and p_2 respectively. The morphism

$$\varphi : X(w_2) \hookrightarrow X(w_1) \to X(w_1) /\!\!/ T$$

is G-invariant. Hence, the universal property of categorical quotients yields a unique morphism $\psi : X(w_2) /\!\!/ T \to X(w_1) /\!\!/ T$ such that the diagram

$$\begin{array}{ccc} X(w_2) & \xrightarrow{\subseteq} & X(w_1) \\ \downarrow{\scriptstyle p_2} & & \downarrow{\scriptstyle p_1} \\ X(w_2) /\!\!/ T & \xrightarrow{\psi} & X(w_1) /\!\!/ T \end{array}$$

commutes. Since both $X(w_1) /\!\!/ T$ and $X(w_2) /\!\!/ T$ are projective over $\mathcal{O}(X)_0$ due to Remark 2.3.9, ψ is projective with closed image in $X(w_1) /\!\!/ T$. Furthermore, $X(w_2)$ is a G-invariant, dense open subset of $X(w_1)$. By the definition of good quotients, the image of the closed complement $X(w_1) \setminus X(w_2)$ under p_1 is closed. Proposition 2.2.3 shows that p_1 is surjective, so that the image of $X(w_2)$ under p_1, which is the image of ψ, contains a non-empty open subset and thus is dense. Overall it follows that ψ is surjective.

Now, there exists a point $z_1 \in X(w_2) \cap p_1^{-1}(p_1(x_1))$. By Proposition 2.2.3, it follows that

$$\overline{Tx_1} \cap \overline{Tz_1} \cap X(w_1) \neq \emptyset.$$

Applying (2.3) yields $Tx_1 \cap \overline{Tz_1} \neq \emptyset$ which results in

$$Tx_1 \subseteq \overline{Tz_1}.$$

By Lemma 2.4.11, $\omega_T(x_1)$ is a face of $\omega_T(z_1)$. Because of $z_1 \in X(w_2)$, it holds that $w_2 \in \omega_T(z_1)$, that is $\sigma_T(w_2) \subseteq \omega_T(z_1)$ by Lemma 2.4.13. Repeating the process for $x_2, \ldots, x_r$ yields orbit cones $\omega_T(z_i)$ such that $\omega_T(x_i) \preceq \omega_T(z_i)$ and $\sigma_T(w_2) \subseteq \omega_T(z_i)$ for all $1 \leq i \leq r$.

Now let $\tau \preceq \sigma_T(w_2)$ be the unique face with $\sigma_T(w_1)^\circ \subseteq \tau^\circ$ by Lemma A.1. In particular,

$$\tau \subseteq \omega_T(z_i) \quad \text{and} \quad \emptyset \neq \sigma_T(w_1)^\circ \subseteq \tau^\circ \cap \omega_T(x_i)^\circ$$

hold for all $1 \leq i \leq r$. Again by Lemma A.1, we conclude that $\tau \subseteq \omega_T(x_i)$. It follows that $\tau \subseteq \sigma_T(w_1)$. The equality is obtained by taking closures in $\sigma_T(w_1)^\circ \subseteq \tau^\circ$. Hence, $\sigma_T(w_1) = \tau$ is a face of $\sigma_T(w_2)$. □

Now we are ready to proof Theorem 2.4.10.

Proof: The finiteness of $\Sigma_T(X)$ is an immediate consequence of Proposition 2.4.5.

Let $\sigma \in \Sigma_T(X)$ be a GIT cone and $\sigma_0 \preceq \sigma_T(w)$ be a face. Choose $w \in \sigma_0^\circ$. By Lemma 2.4.13, σ is the intersection of all orbit cones containing σ. These also contain w, hence $\sigma_T(w) \subseteq \sigma$ by the definition of GIT cones. The prerequisite of Lemma 2.4.14 is satisfied so that $\sigma_T(w) \preceq \sigma$. Now, $\sigma_T(w)$ and σ_0 are both faces of σ containing w in their relative interior by Lemma 2.4.13. This may only be the case when $\sigma_0 = \sigma_T(w)$, proving that $\sigma_0 \in \Sigma_T(X)$.

Next, let $\sigma_1, \sigma_2 \in \Sigma_T(X)$. Let $\tau_i \preceq \sigma_i$ be the unique face such that $(\sigma_1 \cap \sigma_2)^\circ \subseteq \tau_i^\circ$ by Lemma A.1 ($i = 1, 2$). Let $w \in (\sigma_1 \cap \sigma_2)^\circ$. In particular, it holds that $w \in \tau_i^\circ$, so that we obtain $\sigma_T(w) = \tau_i$ and $\sigma_T(w) \subseteq \sigma_i$ as before. Due to $\sigma_1 \cap \sigma_2 \subseteq \tau_i$, it follows that

$$\sigma_1 \cap \sigma_2 = \tau_1 = \tau_2 = \sigma_T(w).$$

For this reason, $\sigma_1 \cap \sigma_2$ is a face of both σ_1 and σ_2. □

3 Concept of the algorithm

In the following sections we are going to develop a mathematical and algorithmic framework that allows us to compute the GIT fan for the case of a torus acting on an affine variety. The concepts presented in this chapter base on [10, 24].

First, we describe the formal setup and introduce the notation being used in the upcoming chapters. In the second section we present an algorithm that is capable of identifying orbit cones. They play an important role in the theory since all GIT cones emerge from their intersections. Afterwards, we transform the problem of computing the GIT fan into the problem of traversing a graph and present a suitable algorithm. Third, we modify the previous algorithms such that symmetries of the affine variety may be exploited in order to reduce the problem size by a factor that relates to the size of the symmetry group. Finally, we give an counterexample that – contrary to [10] – one cannot drop non-minimal orbit cones with respect to inclusion. For this reason, we leave out the minimisation process.

3.1 The setup

Let $\mathbb{K}$ be an algebraically closed field of characteristic zero. We consider an algebraic group action of a k-dimensional torus T on an affine variety X. Due to construction 2.1.5, we describe this setup by a set of integral vectors $q_i \in \mathbb{Z}^k$, $1 \leq i \leq r$, and a suitable embedding of X in $\mathbb{K}^r$ such that its vanishing ideal $\mathfrak{a} := I(X)$ is homogeneous w.r.t.

$$\deg(x_i) = q_i, \quad 1 \leq i \leq r.$$

The vectors q_i form the columns of a matrix $Q \in \mathbb{Z}^{k \times r}$. By restricting T to a lower dimensional subtorus if necessary, we may assume that Q has rank k without changing the orbits in X. Furthermore, we assume that $\mathfrak{a}$ contains no monomials. As we will see later on, this assumption is equivalent to $X \cap (\mathbb{K}^*)^r \neq \emptyset$. On that account and the irreducibility of X, X is located in at least one of the hyperplanes $\{x_i = 0\}$, $1 \leq i \leq r$, iff the assumption does not hold. By "forgetting" all variables with $X \cap \{x_i = 0\} = X$, we obtain an embedding of X such that $X \cap (\mathbb{K}^*)^{r'} \neq \emptyset$ holds.

3.2 Computing orbit cones

In this section we describe orbit cones by faces of the positive orthant $\mathbb{Q}^r_{\geq 0}$, which we denote by γ. From a combinatorial point of view these faces are subsets of $\{1,\ldots,r\}$ such that whenever e_i is a ray of a face, i is located in the corresponding subset. A face $\gamma_0 \preceq \gamma$ defines a restriction z_{γ_0} of any r-tuple of objects $z = (z_1,\ldots,z_r)$ by setting

$$(z_{\gamma_0})_i = \begin{cases} z_i, & e_i \in \gamma_0 \\ 0, & \text{else} \end{cases} \qquad 1 \leq i \leq r.$$

We write $\mathfrak{a}_{\gamma_0}$ for the ideal in $\mathbb{K}[\mathbf{x}_{\gamma_0}]$ that is obtained by sending all x_i in $\mathfrak{a}$ with $x_i \notin \gamma_0$ to zero. Additionally, we define

$$\mathbb{T}_{\gamma_0} := (\mathbb{K}^*)^r \cdot (1,\ldots,1)_{\gamma_0},$$

which is the torus corresponding to γ_0.

Definition 3.2.1 ($\mathfrak{a}$-face) A face $\gamma_0 \preceq \gamma$ is called $\mathfrak{a}$-*face* iff $X \cap \mathbb{T}_{\gamma_0} \neq \emptyset$.

Proposition 3.2.2 *The set*

$$\Omega_{\mathfrak{a}} := \{Q(\gamma_0) \mid \gamma_0 \preceq \gamma \text{ is an } \mathfrak{a}\text{-face}\}$$

equals the collection of orbit cones as in Definition 2.4.3.

Proof: Let $x \in X$. Set $I_x = \{1 \leq i \leq r \mid x_i \neq 0\}$ and $J_x = \{1 \leq i \leq r \mid x_i = 0\}$. We claim that

$$S_T(x) = \langle q_i \mid i \in I_x \rangle.$$

Let $i \in I_x$. Since $x_i \in \mathbb{K}[\mathbf{x}]$ does not send x to zero, we have

$$q_i = \deg(x_i) \in S_T(x).$$

Now let $w \in S_T(x)$. Then there exists $f \in \mathbb{K}[\mathbf{x}]_w$ such that $f(x) \neq 0$. In particular, at least one monomial $\mathbf{x}^\alpha$ of f does not send x to zero, thus $\alpha_j = 0$ for all $j \in J_x$. On that account we have

$$w = \deg(x^\alpha) = \sum_{i=1}^{r} \alpha_i q_i = \sum_{i \in I_x} \alpha_i q_i,$$

proving the claim.

If $\gamma_0 \preceq \gamma$ is an $\mathfrak{a}$-face, we find a point $x \in X \cap \mathbb{T}_{\gamma_0}$. By the previous claim, its orbit monoid is generated by the columns q_i of Q with $e_i \in \gamma_0$. Consequently, $Q(\gamma_0)$ is the orbit cone of x generated by $S_T(x)$.

Conversely, consider the orbit cone of an arbitrary point $x \in X$. Let γ_x be the face of γ that is generated by the rays e_i with $i \in I_x$. Then γ_x is an $\mathfrak{a}$-face, since $x \in X \cap \mathbb{T}_{\gamma_x}$. With the same arguments as before, it follows that $Q(\gamma_x)$ is the orbit cone of x. □

The previous proposition shows that all orbit cones arise from $\mathfrak{a}$-faces and that every $\mathfrak{a}$-face yields an orbit cone. Thus, if we are able to develop computable criteria in order to implement a test for the $\mathfrak{a}$-face property, we can derive all orbit cones by applying Q. The next proposition provides us with a suitable criterion.

Proposition 3.2.3 *Let $\gamma_0 \preceq \gamma$ be a face and $I = \{1 \le i \le r \mid e_i \in \gamma_0\}$. Then the following conditions are equivalent:*

(i) *γ_0 is an $\mathfrak{a}$-face, that is $X \cap \mathbb{T}_{\gamma_0} \neq \emptyset$,*

(ii) *$\mathfrak{a}_{\gamma_0}$ does not contain a monomial,*

(iii) $(\mathfrak{a}_{\gamma_0} : (\prod_{i\in I} x_i)^\infty) = \mathbb{K}[\mathbf{x}_{\gamma_0}]$.

Proof: (ii) ⇔ (iii) immediately follows from the definition of saturated ideals (see Appendix B), which unravels to

$$\left(\mathfrak{a}_{\gamma_0} : \left(\prod_{i\in I} x_i\right)^\infty\right) = \left\{ r \in \mathbb{K}[\mathbf{x}] \;\middle|\; \exists j \in \mathbb{N} : r \cdot \left(\prod_{i\in I} x_i\right)^j \in \mathfrak{a}_{\gamma_0} \right\}.$$

If $\mathfrak{a}_{\gamma_0}$ contains a monomial x^α, we have that $(\prod_{i\in I} x_i)^j \in \mathfrak{a}_{\gamma_0}$, where j is the largest integer occurring in α. Thus, $1 \in (\mathfrak{a}_{\gamma_0} : (\prod_{i\in I} x_i)^\infty)$ and therefore (iii) follows. Otherwise, if (iii) holds, a power of $\prod_{i\in I} x_i$, which is a monomial, is contained in $\mathfrak{a}_{\gamma_0}$.

(i) ⇒ (ii): Choose a point $x \in X \cap \mathbb{T}_{\gamma_0}$. Assume that (ii) does not hold, i.e. $\mathfrak{a}_{\gamma_0}$ contains a monomial $\mathbf{x}^\alpha$, $\alpha = (\alpha_i)_{i\in I}$, that is obtained by sending all x_i, $i \notin I$, in a suitable $f \in \mathfrak{a}$ to zero. Since $x \in \mathbb{T}_{\gamma_0}$, it holds that

$$f(x) = \prod_{i\in I} x_i^{\alpha_i} \neq 0,$$

contradicting $x \in X = V(\mathfrak{a})$.

(ii) ⇒ (i): Assume that $V(\mathfrak{a}_{\gamma_0}) \subseteq V(\prod_{i\in I} x_i)$ holds. Then Hilbert's Nullstellensatz yields $j \in \mathbb{N}$ such that $(\prod_{i\in I} x_i)^j \in \mathfrak{a}_{\gamma_0}$, contradicting (ii). Hence, $V(\mathfrak{a}_{\gamma_0}) \not\subseteq V(\prod_{i\in I} x_i)$. In particular, we find a point $z = (z_i)_{i\in I} \subseteq \mathbb{K}^*$ with $z \in V(\mathfrak{a}_{\gamma_0})$. We extend z to a point $\tilde{z} \in \mathbb{T}_{\gamma_0}$ by setting

$$\tilde{z}_i = \begin{cases} z_i, & i \in I \\ 0, & \text{else} \end{cases} \qquad 1 \le i \le r.$$

Let $f \in \mathfrak{a}$ and $f_{\gamma_0} \in \mathfrak{a}_{\gamma_0}$ be the reduction of f by sending all variables x_i, $i \notin I$, to zero. Then we have

$$f(\tilde{z}) = f_{\gamma_0}(z) = 0,$$

since $z \in V(\mathfrak{a}_{\gamma_0})$. We conclude that $\tilde{z} \in X$, showing (i). □

Next, we implement an algorithm computing $(I : (z_1 \cdots z_n)^\infty)$ for arbitrary ideals $I \in \mathbb{K}[z_1, \ldots, z_n]$. The algorithm presented here originates from [10] and has been implemented in GITFAN.LIB. It relies on the following proposition that allows us to compute $(I : (z_i)^\infty)$ with relative ease by a small modification to the Buchberger's algorithm.

Proposition 3.2.4 ([10, Proposition 3.1]) *Let $>$ be a monomial ordering on $\mathbb{K}[\mathbf{z}]$ and $\mathcal{G}$ be a standard basis of I with respect to $>$. Let $m \in \{1, \ldots, n\}$. If $>$ satisfies*

$$z_m \mid f \;\Leftrightarrow\; z_m \mid LM_>(f), \tag{3.1}$$

then

$$\left\{ g \in \mathbb{K}[\mathbf{z}] \;\middle|\; z_m \nmid g \text{ and } \exists i \in \mathbb{N}_0 : (z_m)^i \cdot g \in \mathcal{G} \right\}$$

is a standard basis for the saturated ideal $I : (z_m)^\infty$.

Remark 3.2.5 The negative reverse lexicographical ordering

$$z^\alpha >_{\mathrm{rs}} z^\beta \;:\Leftrightarrow\; \exists i \in \{1, \ldots, n\} : \alpha_i < \beta_i \text{ and } \alpha_j = \beta_j \; \forall j \in \{i+1, \ldots, n\}$$

ensures that (3.1) is satisfied for $m = n$. Every monomial z^β with $z_n \mid z^\beta$ is strictly smaller than any monomial z^α with $z_n \nmid z^\alpha$, since $\alpha_n = 0 < \beta_n$. Note that (3.1) may hold for other values of m by using a negative reverse lexicographical ordering after reordering the variables $z_1, \ldots, z_n$.

Unfortunately, $>_{\mathrm{rs}}$ is no global ordering. However, if I permits a weight vector $w \in \mathbb{Z}^n_{>0}$ such that I is homogeneous, we consider the w-weighted degree ordering with tie-breaker $>_{\mathrm{rs}}$, that is $(>_w, >_{\mathrm{rs}})$. This ordering is global and satisfies (3.1) on w-homogeneous generators of I.

Since we have

$$(I : (z_1 \cdots z_n)^\infty) = (\ldots((I : (z_1)^\infty) : (z_2)^\infty) : \cdots : (z_n)^\infty) \quad \text{(see Appendix B)},$$

Proposition 3.2.4 allows us to compute $(I : (z_1 \cdots z_n)^\infty)$ by alternating Mora's algorithm with $>_{\mathrm{rs}}$ and the elimination of factors z_i. If a global ordering as in Remark 3.2.5 is available, one can integrate the elimination process into the Buchberger's algorithm in order to speed up convergence. Algorithm 1 (see also [10, Algorithm 3.3]) contains the necessary modifications.

Algorithm 1: Computing the saturation at the product of all variables

Input: A vector $w \in \mathbb{Z}^n_{>0}$ and a set $\mathcal{G}$ of w-homogenous polynomials such that $\langle \mathcal{G} \rangle = I$

Output: A Gröbner basis for the saturation $(I : (z_1 \cdots z_n)^\infty)$ with respect to the ordering $(>_w, >_{\mathrm{rs}})$

$\mathcal{G} \leftarrow \{g/z^\alpha \mid g \in \mathcal{G} \text{ and } \alpha \text{ maximal such that } z^\alpha \mid g\}$;
for $m = 1, \ldots, n$ **do**
  Let $>_m$ be $(>_w, >_{\mathrm{rs}})$ where $>_{\mathrm{rs}}$ is the negative reverse lexicographical ordering after reordering variables such that

$$z_1 >_{\mathrm{rs}} \cdots >_{\mathrm{rs}} z_{m-1} >_{\mathrm{rs}} z_{m+1} >_{\mathrm{rs}} \cdots >_{\mathrm{rs}} z_n >_{\mathrm{rs}} z_m.$$

  repeat
    $\mathcal{H} \leftarrow \mathcal{G}$;
    for $f, g \in \mathcal{H}$ **do**
      $r \leftarrow \mathrm{NF}_{>_m}(\mathrm{spoly}_{>_m}(f, g), \mathcal{H})$;
      if $r \neq 0$ **then**
        $r \leftarrow r/z^\alpha$, where α is maximal such that $z^\alpha \mid g$;
        $\mathcal{G} \leftarrow \mathcal{G} \cup \{r\}$;
      end
    end
  until $\mathcal{G} = \mathcal{H}$;
end
return $\mathcal{G}$;

Proposition 3.2.6 *Algorithm 1 computes the saturation* $(I : (z_1 \cdots z_n)^\infty)$ *as specified in pseudo code.*

Proof: Similarly to the Buchberger's algorithm, whenever a new element r is introduced to $\mathcal{G}$, its lead monomial $\mathrm{LM}(r)$ is not contained in the lead ideal of $\langle \mathcal{G} \rangle$. Consequently, $\langle \mathcal{G} \rangle$ strictly increases. Since $\mathbb{K}[\mathbf{z}]$ is noetherian, eventually $\langle \mathcal{G} \rangle$ becomes stationary, implying that no new elements r have been inserted. Hence, the condition in Line 13 holds eventually and the algorithm terminates.

Denote by $\mathcal{G}_0$ the set $\mathcal{G}$ after executing Line 1 and by $\mathcal{G}_m$ the Gröbner basis after the m-iteration of the outmost loop. Set $I_m = \langle \mathcal{G}_m \rangle$.

First, note that the elements inserted into $\mathcal{G}$ are contained in $(I : (z_1 \cdots z_n)^\infty)$ and are not divisible by any monomials. Furthermore, they are w-homogeneous since NF and spoly preseve homogeneity. Consequently, $\mathcal{G}_m$ satisfies the assumption of Proposition 3.2.4 with respect to the ordering $>_m$. As no element of $\mathcal{G}_m$ is divisible by any monomial, it follows that $\mathcal{G}_m$ is saturated with respect to z_m. In particular, $(I_{m-1} : (z_m)^\infty) \subseteq I_m$. Appendix B states that saturations preserve inclusion, so that $I \subseteq I_0$ together with $(I_{m-1} : (z_m)^\infty) \subseteq I_m$ implies that

$$(I : (z_1 \cdots z_n)^\infty) = (\ldots((I : (z_1)^\infty) : (z_2)^\infty) : \cdots : (z_n)^\infty) \subseteq I_n$$

by a simple induction. It follows that $\langle \mathcal{G}_m \rangle = I_m = (I : (z_1 \cdots z_n)^\infty)$. □

Now we are able to compute all orbit cones $\Omega_{\mathfrak{a}}^{(k)}$ of full dimension k by Algorithm 2. Its correctness is a straightforward conclusion from Proposition 3.2.2 and 3.2.3.

Notation 3.2.7 Let $\mathcal{C}$ be a set of convex cones. Then we write $\mathcal{C}^{(k)}$ for the subset of all k-dimensional cones in $\mathcal{C}$.

3.3 Traversing the GIT fan

With the full dimensional orbit cones at hand, we are able to determine the GIT fan $\Sigma_T(X)$ defined by the torus action T on the variety X. Since the support of $\Sigma_T(X)$ is the image $Q(\gamma)$ and Q has rank k, it follows that $\Sigma_T(X)$ is a k-dimensional quasifan with convex support. By Lemma A.2, it suffices to consider only the full dimensional GIT cones in $\Sigma_T(X)^{(k)}$. These yield a graph structure by relating two GIT cones sharing a common facet.

Definition 3.3.1 The graph $\mathcal{G}(\Sigma_T(X))$ of the GIT fan $\Sigma_T(X)$ is the undirected graph with nodes in $\Sigma_T(X)^{(k)}$ and the symmetric relation

$$E_{\mathcal{G}(\Sigma_T(X))} := \left\{ \{\lambda, \mu\} \mid \lambda, \mu \in \Sigma_T(X)^{(k)} \text{ and } \lambda \cap \mu \text{ is a facet of both } \lambda \text{ and } \mu \right\}.$$

Algorithm 2: Computing all full dimensional orbit cones

Input: $\mathfrak{a}$, Q

Output: All orbit cones of $\Omega_{\mathfrak{a}}^{(k)}$ dimension k

```
Ω ← ∅;
for γ_0 ⪯ γ with dim(Q(γ_0)) = k do
    I ← a_{γ_0};
    I_sat ← (I : (∏_{e_i ∈ γ_0} x_i)^∞);          /* by Algorithm 1 */
    if I_sat = K[x_{γ_0}] then
        Ω ← Ω ∪ {Q(γ_0)};
    end
end
return Ω;
```

Proposition 3.3.2 *The graph $\mathcal{G}(\Sigma_T(X))$ is connected.*

Proof: Let $\lambda_s, \lambda_t \in \Sigma_T(X)^{(k)}$ and choose a point $x_s \in \lambda_s^\circ$. Let S be the set of points that are not located on a line through x_s and a point of a face of dimension $k-2$ or lower, that is

$$S = \left\{ p \in |\Sigma| \;\middle|\; \overline{x_s p} \cap \left(\bigcup_{\lambda \in \Sigma_T(X)^{(\leq k-2)}} \lambda \right) = \emptyset \right\}.$$

Obviously, for every point $p \in |\Sigma| \setminus S$ we can choose $\lambda \in \Sigma_T(X)^{(\leq k-2)}$ such that p is located in the subspace generated by x_s and λ. As the dimension of this subspace is at most $k-1$, we conclude that the complement of S in $|\Sigma|$ is contained in a finite union of hyperplanes. For dimensional reasons, any non-empty open set U in $|\Sigma|$ with respect to the euclidean topology satisfies

$$U \cap S \neq \emptyset.$$

This shows that there exists a point $x_t \in \lambda_t^\circ$ such that $x_t \in S$.

Consider the connecting line $\ell := \overline{x_s x_t}$. We have $\ell \subseteq |\Sigma_T(X)|$, because $|\Sigma_T(X)|$ is convex. By the definition of S, ℓ only contains points in the relative interior of $(k-1)$- or k-dimensional cones. Furthermore, ℓ intersects every $(k-1)$-dimensional cone λ in at most one point. Otherwise, ℓ would be contained in the hyperplane spanned by λ due to Bézout's theorem. Then, ℓ also would intersect a proper face of λ with dimension lower than $k-1$, contradicting $x_t \in S$.

Since the relative interior of a k-dimensional cone is open and convex, its intersection with ℓ is an open interval in ℓ. We conclude that ℓ has the form

with $\lambda_i \in \Sigma_T(X)^{(k)}$ for all $1 \leq i \leq n$ and $\mu_j \in \Sigma_T(X)^{(k-1)}$ for all $1 \leq j \leq n-1$. In particular, the point in $\mu_j^\circ \cap \ell$ is contained in λ_j and λ_{j+1}, so that μ_j is a facet of λ_j and λ_{j+1}. For this reason, $\lambda_1, \lambda_2, \ldots, \lambda_n$ constitutes a path in $\mathcal{G}(\Sigma_T(X))$ from λ_1 to λ_n. Since $x_s \in \lambda_1$ and $x_t \in \lambda_n$, it follows that $\lambda_1 = \lambda_s$ and $\lambda_n = \lambda_t$. □

Proposition 3.3.2 allows us to determine all full dimensional GIT cones by traversing the graph $\mathcal{G}(\Sigma_T(X))$. For this reason we require a routine for computing all GIT cones related to a given GIT cone. We begin with some elementary observations about GIT cones that also become important when encoding GIT cones in section 4.3.

Notation 3.3.3 Let $p \in Q(\gamma)$. Then $\lambda(p)$ denotes the associated GIT cone with respect to the torus action of T on X, that is

$$\lambda(p) := \bigcap_{p \in \sigma \in \Omega_{\mathfrak{a}}} \sigma$$

due to Propositon 3.2.2.

Lemma 3.3.4 *Let $p \in Q(\gamma)$ such that $\lambda := \lambda(p) \in \Sigma_T(X)^{(k)}$. Then we have*

(i) $\lambda = \bigcap_{p \in \sigma \in \Omega_{\mathfrak{a}}^{(k)}} \sigma$,

(ii) $p \in \sigma \Leftrightarrow \lambda \subseteq \sigma \quad \forall \sigma \in \Omega_{\mathfrak{a}}^{(k)}$,

(iii) $\lambda = \bigcap_{\lambda \subseteq \sigma \in \Omega_{\mathfrak{a}}^{(k)}} \sigma$,

(iv) $\lambda = \lambda(q) \quad \forall q \in \lambda^\circ$.

Proof: (i) is trivial since dimensions do not increase when intersecting. (ii) clearly holds by the definition of λ. (iii) is an immediate consequence of (i) and (ii).

Let $q \in \lambda^\circ$. Since q is contained in all $\sigma \in \Omega_{\mathfrak{a}}^{(k)}$ such that $p \in \sigma$, it follows that $\lambda(q) \subseteq \lambda$. If the inequality would be proper, $\lambda(q)$ would be a proper face of λ containing q, contradicting $q \in \lambda^\circ$. □

Algorithm 3: Computing all related GIT cones

Input: A full dimensional GIT cone $\lambda \in \Sigma_T(X)^{(k)}$

Output: All full dimensional GIT cones $\mu \in \Sigma_T(X)^{(k)}$ such that $\lambda \cap \mu$ is a common facet of λ and μ

$\mathcal{N} \leftarrow \emptyset$;
for $\vartheta \preceq \lambda$ being a facet of λ with $\vartheta^\circ \cap \partial Q(\gamma) = \emptyset$ **do**
    Let n be an inner-pointing normal of ϑ;
    Choose $v \in \vartheta^\circ$;
    Choose small $\varepsilon > 0$;
    $q \leftarrow v - \varepsilon n$;
    while $v \notin \lambda(q)$ **do**
        $\varepsilon \leftarrow \varepsilon/2$;
        $q \leftarrow v - \varepsilon n$;
    end
    $\mathcal{N} \leftarrow \mathcal{N} \cup \{\lambda(q)\}$;
end
return $\mathcal{N}$;

Proposition 3.3.5 *Algorithm 3 computes all neighbours for a given node in* $\mathcal{G}(\Sigma_T(X))$.

Proof: First, note that for $\tau \in \Sigma_T(X)^{(k-1)}$ with normal n and relative inner point v every $\sigma \in \Sigma_T(X)^{(k)}$ with $\tau \preceq \sigma$ has to contain either $v - \varepsilon n$ or $v + \varepsilon n$ in its relative interior for sufficiently small ε. For this reason there exist exactly two cones with this property iff $\tau^\circ \cap \partial Q(\gamma) = \emptyset$ and one cone otherwise. If two cones exist, they are separated by the hyperplane spanned by τ, so that the intersection is τ.

We reuse the notations introduced in Algorithm 3. Let ϑ be a facet of λ with $\vartheta^\circ \cap \partial Q(\gamma) = \emptyset$. Let $v \in \vartheta^\circ$ and n an inner-pointing normal of ϑ. Because of $v \in Q(\gamma)^\circ$, there exists an $\varepsilon > 0$ such that the connecting line $\ell_\varepsilon := \overline{v(v - \varepsilon n)}$ is contained in $Q(\gamma)$. Since cones are convex, all cones $\mu \in \Sigma_T(X)^{(<k)}$ intersect with ℓ_ε in a possibly empty interval I_μ. If $v \in I_\mu$, it follows that $\vartheta = \mu$, since $v \in \vartheta^\circ$ and μ has at most dimension $k-1$. Due to $|\Sigma_T(X)^{(<k)}| < \infty$, we can choose ε sufficiently small such that the only cone with dimension $< k$ intersecting ℓ_ε is given by ϑ. By the definition of n, we have $\ell_\varepsilon \cap \vartheta = \{v\}$.

It follows that the connected space $\ell_\varepsilon \setminus \{v\}$ is covered by disjoint open intervals $\ell_\varepsilon \cap \mu^\circ$, $\mu \in \Sigma_T(X)^{(k)}$, and thus we have

$$\ell_\varepsilon \setminus \{v\} = \ell_\varepsilon \cap \mu_0^\circ$$

for a suitable $\mu_0 \in \Sigma_T(X)^{(k)}$. By Lemma 3.3.4, it follows that $\mu_0 = \lambda(v - \varepsilon n)$. The closure μ_0 of μ_0° contains v. Hence, the loop in Line 7 terminates. In particular, $\vartheta \preceq \mu_0$ and μ_0 is the unique cone unequal to λ with this property. □

Remark 3.3.6 Since cones are invariant with respect to scaling, one may assume $v, n \in \mathbb{Z}^k$ and consider $mv - n$ for sufficiently large $m \in \mathbb{N}$ instead of $v - \varepsilon n$. Hence, Algorithm 3 may be realised by only integer operations.

Finally, Algorithm 4 describes a graph traversal and returns all full dimensional GIT cones by Proposition 3.3.2.

Algorithm 4: Computing all full dimensional GIT cones

Input: $Q(\gamma)$, $\Omega_{\mathfrak{a}}^{(k)}$
Output: $\Sigma_T(X)^{(k)}$

```
1  repeat
2  |  Choose random point p ∈ Q(γ);
3  until dim(λ(p)) = k;
4  C ← {λ(p)};
5  F ← {λ(p)};
6  while F ≠ ∅ do
7  |  Choose λ ∈ F;
8  |  F ← F \ {λ};
9  |  for μ with {λ, μ} ∈ E_{G(Σ_T(X))} do        /* by Algorithm 3 */
10 |  |  if μ ∉ C then
11 |  |  |  C ← C ∪ {μ};
12 |  |  |  F ← F ∪ {μ};
13 |  |  end
14 |  end
15 end
16 return C;
```

3.4 Exploiting symmetry

In this section we modify Algorithm 4 such that symmetries of T acting on X are taken into account, reducing the size of the traversed graph without loosing information. On that account, we include a symmetry group $\mathcal{S}$ in our setup.

Definition 3.4.1 ([10, Definition 4.1]) A *symmetry group* of the action of T on X is a subgroup $\mathcal{S}$ of the symmetric group $\mathcal{S}_r$ such that there are group actions

$$\begin{array}{ccccccc} \mathcal{S} & \times & \mathbb{K}[\mathbf{x}] & \to & \mathbb{K}[\mathbf{x}], & (\sigma, x_i) & \mapsto & \sigma x_i := c_{\sigma,i} \cdot x_{\sigma(i)} \\ \mathcal{S} & \times & \mathbb{Q}^r & \to & \mathbb{Q}^r, & (\sigma, e_i) & \mapsto & \sigma e_i := e_{\sigma(i)} \end{array}$$

with $c_\sigma \in (\mathbb{K}^*)^r$ such that $\mathcal{S}\mathfrak{a} = \mathfrak{a}$ holds and

$$\sigma \ker(Q) \subseteq \ker(Q) \quad \forall \sigma \in \mathcal{S}.$$

Remark 3.4.2 Let $\sigma \in \mathcal{S}$. The condition $\sigma \ker(Q) \subseteq \ker(Q)$ allows us to construct $A_\sigma \in \mathrm{GL}(k, \mathbb{Q})$ such that

$$\begin{array}{ccc} \mathbb{Q}^r & \xrightarrow{v \mapsto \sigma v} & \mathbb{Q}^r \\ Q \downarrow & & \downarrow Q \\ \mathbb{Q}^k & \xrightarrow{A_\sigma} & \mathbb{Q}^k \end{array}$$

commutes. Hence, the group action of $\mathcal{S}$ on $\mathbb{Q}^r$ induces a group action of $\mathcal{S}$ on $\mathbb{Q}^k$ with $(\sigma, v) \mapsto A_\sigma v$, such that the map defined by Q becomes equivariant.

We define A_σ as follows: For $i \in \{1, \ldots, k\}$, choose $v_i \in \mathbb{Q}^r$ such that $Qv_i = e_i \in \mathbb{Q}^k$. Then the i-th column of A_σ is given by $Q(\sigma v_i)$. By construction, the diagram commutes on the subspace $T \subseteq \mathbb{Q}^r$ with basis $\{v_1, \ldots, v_k\}$. It also commutes on $\ker(Q)$, sending all $v \in \ker(Q)$ to 0 due to $\sigma v \in \ker(Q)$. We conclude that the diagram commutes on all $\mathbb{Q}^r = \ker(Q) \oplus T$.

Since we have

$$\mathrm{Im}(A_\sigma) = Q(\sigma \mathbb{Q}^r) = Q(\mathbb{Q}^r) = k,$$

A_σ is invertible.

Lemma 3.4.3 ([10, Lemma 4.7]) *The action $\mathcal{S} \circlearrowright \mathbb{Q}^r$ induces an action of $\mathcal{S}$ on the set of $\mathfrak{a}$-faces.*

Proof: Let $\sigma \in \mathcal{S}$ and $\gamma_0 \preceq \gamma$. We have to show that $\sigma\gamma_0$ is again an $\mathfrak{a}$-face. Since $\mathcal{S}$ permutes the canonical unit vectors, $\sigma\gamma_0$ is a face of γ. Let $x \in X \cap \mathbb{T}_{\gamma_0}$. We define $z \in \mathbb{K}^r$ such that

$$z_i = c_{\sigma^{-1},i} \cdot x_{\sigma^{-1}(i)} \quad \forall 1 \le i \le r$$

with $c_{\sigma^{-1}}$ as in Definition 3.4.1. It holds that

$$e_i \in \sigma\gamma_0 \Leftrightarrow e_{\sigma^{-1}(i)} \in \gamma_0 \Leftrightarrow x_{\sigma^{-1}(i)} \neq 0 \Leftrightarrow z_i = c_{\sigma^{-1},i} \cdot x_{\sigma^{-1}(i)} \neq 0 \quad \forall 1 \le i \le r,$$

implying $z \in \mathbb{T}_{\sigma\gamma_0}$. Let $f \in \mathfrak{a}$. By construction, we have $f(z) = (\sigma^{-1}f)(x)$. Now, $\sigma^{-1}f \in \mathfrak{a}$ and x vanishes on $\mathfrak{a}$, so that $f(z) = 0$. For this reason $z \in X = V(\mathfrak{a})$ holds and $\sigma\gamma_0$ is an $\mathfrak{a}$-face. □

Proposition 3.4.4 *The action $\mathcal{S} \circlearrowright \mathbb{Q}^k$ introduced in Remark 3.4.2 induces an action $\mathcal{S} \circlearrowright \Omega_{\mathfrak{a}}^{(k)}$. In particular, $\mathcal{S}$ acts on $\Sigma_T(X)^{(k)}$.*

Proof: Let $\sigma \in \mathcal{S}$ and $Q(\gamma_0) \in \Omega_{\mathfrak{a}}$ with $\gamma_0 \preceq \gamma$ being an $\mathfrak{a}$-face. We show that $\sigma Q(\gamma_0) \in \Omega_{\mathfrak{a}}^{(k)}$. By Remark 3.4.2, Q is equivariant. It follows that $\sigma Q(\gamma_0) = Q(\sigma\gamma_0)$. Since $\sigma\gamma_0$ is an $\mathfrak{a}$-face by Lemma 3.4.3, we have $\sigma Q(\gamma_0) \in \Omega_{\mathfrak{a}}$.

If $\lambda(p)$ is a GIT cone, then $\sigma\lambda(p)$ is the GIT cone associated to $A_\sigma p$, as it holds that

$$A_\sigma p \in A_\sigma \vartheta = \sigma\vartheta \;\Leftrightarrow\; p \in \vartheta \quad \forall \vartheta \in \Omega_{\mathfrak{a}}.$$

Hence, $\mathcal{S}$ also acts on $\Sigma_T(X)$.

Note that the action of $\mathcal{S}$ on cones in $\mathbb{Q}^k$ preserves dimensions because the matrices A_σ are invertible. Consequently, we can restrict the actions on $\Omega_{\mathfrak{a}}$ and $\Sigma_T(X)$ to cones of the fixed dimension k. □

Later on in chapter 5, we are going to intersect the GIT fan with the cone of movable divisor classes of a Mori dream space. If this cone is invariant under the action of $\mathcal{S}$ on $\mathbb{Q}^k$, it suffices to intersect it with a representative system of orbit cone orbits only. The following proposition together with Remark 5.3.2 states the cone's invariance.

Proposition 3.4.5 *The convex cone*

$$\bigcap_{\gamma_0 \preceq \gamma \text{ facet}} Q(\gamma_0) \subseteq \mathbb{Q}^k$$

is invariant with respect to the action $\mathcal{S} \circlearrowright \mathbb{Q}^k$.

Proof: Let $\sigma \in \mathcal{S}$. Since permutations in $\mathcal{S}_r$ permute the subsets of $\{1,\dots,r\}$ with cardinality $r-1$, the facets of γ are permuted by σ. It follows that

$$\sigma \cdot \left(\bigcap_{\gamma_0 \preceq \gamma \text{ facet}} Q(\gamma_0) \right) = \bigcap_{\gamma_0 \preceq \gamma \text{ facet}} \sigma \cdot Q(\gamma_0) = \bigcap_{\gamma_0 \preceq \gamma \text{ facet}} Q(\sigma\gamma_0) = \bigcap_{\gamma_0 \preceq \gamma \text{ facet}} Q(\gamma_0).$$

□

Taking the symmetry group $\mathcal{S}$ into account, it suffices to traverse the following graph in order to determine the GIT fan.

Definition 3.4.6 The graph $\mathcal{G}_{\mathcal{S}}(\Sigma_T(X))$ is the undirected graph with nodes in

$$\Sigma_T(X)^{(k)}/_{\mathcal{S}}$$

and the symmetric relation

$$E_{\mathcal{G}_{\mathcal{S}}(\Sigma_T(X))} := \left\{ \{\mathcal{S}\lambda, \mathcal{S}\mu\} \;\middle|\; \{\lambda,\mu\} \in E_{\mathcal{G}(\Sigma_T(X))} \right\}.$$

Remark 3.4.7 Since $\mathcal{G}(\Sigma_T(X))$ is connected, it follows that the graph $\mathcal{G}_{\mathcal{S}}(\Sigma_T(X))$ is connected.

Algorithm 5: Computing all neighbours of a GIT cone orbit

Input: A GIT cone orbit $\mathcal{O} \in \Sigma_T(X)^{(k)}/_{\mathcal{S}}$

Output: All GIT cone orbits $\mathcal{O}_{\mathcal{N}} \in \Sigma_T(X)^{(k)}/_{\mathcal{S}}$ such that $\{\mathcal{O},\mathcal{O}_{\mathcal{N}}\} \in E_{\mathcal{G}_{\mathcal{S}}(\Sigma_T(X))}$

1 $\mathcal{N} \leftarrow \emptyset$;
2 Choose λ such that $\mathcal{O} = \mathcal{S}\lambda$;
3 **for** μ such that $\{\lambda,\mu\} \in E_{\mathcal{G}(\Sigma_T(X))}$ **do** /* by Algorithm 3 */
4 $\quad \mathcal{N} \leftarrow \mathcal{N} \cup \{\mathcal{S}\mu\}$;
5 **end**
6 **return** $\mathcal{N}$;

Proposition 3.4.8 *Algorithm 5 computes all neighbours of a node in* $\mathcal{G}_{\mathcal{S}}(\Sigma_T(X))$.

Proof: We reuse the notation of Algorithm 5. Clearly, for all $\mathcal{O}_{\mathcal{N}} \in \mathcal{N}$ it holds that $\{\mathcal{O},\mathcal{O}_{\mathcal{N}}\} \in E_{\mathcal{G}_{\mathcal{S}}(\Sigma_T(X))}$.

Conversely, let $\mathcal{O}_{\mathcal{N}} \in \Sigma_T(X)^{(k)}/\mathcal{S}$ such that $\{\mathcal{O},\mathcal{O}_{\mathcal{N}}\} \in E_{\mathcal{G}_{\mathcal{S}}(\Sigma_T(X))}$. Then there exist $\lambda',\mu' \in \Sigma_T(X)^{(k)}$ such that $\{\lambda',\mu'\} \in E_{\mathcal{G}(\Sigma_T(X))}$ and $\mathcal{O} = \mathcal{S}\lambda'$, $\mathcal{O}_{\mathcal{N}} = \mathcal{S}\mu'$. Let $\sigma \in \mathcal{S}$ such that $\lambda' = \sigma\lambda$. Let ϑ be the common facet of λ' and μ' and $n_{\lambda'}$, $n_{\mu'}$ the corresponding inner-pointing normals. Then $\sigma^{-1}\vartheta$ is the common facet of $\sigma^{-1}\lambda'$ and $\sigma^{-1}\mu'$ with inner-pointing normals $(A^*_{\sigma^{-1}})^{-1}n_{\lambda'}$ and $(A^*_{\sigma^{-1}})^{-1}n_{\mu'}$, where $A^*_{\sigma^{-1}}$ is the classical adjoint of $A_{\sigma^{-1}}$. In particular, $\{\lambda,\sigma^{-1}\mu'\} \in E_{\mathcal{G}(\Sigma_T(X))}$. Due to $\mathcal{O} = \mathcal{S}\lambda$ and $\mathcal{O}_{\mathcal{N}} = \mathcal{S}\mu' = \mathcal{S}(\sigma^{-1}\mu')$, it follows that $\mathcal{O}_{\mathcal{N}}$ is added to $\mathcal{N}$ when iterating the loop in Line 3. □

Finally, Algorithm 6 computes all full dimensional GIT cone orbits. It arises from Algorithm 4 by small modifications coloured in red.

Algorithm 6: Computing all full dimensional GIT cone orbits

Input: $Q(\gamma)$, $\Omega_{\mathfrak{a}}^{(k)}$

Output: ${}^{\Sigma_T(X)^{(k)}}\!/_{\mathcal{S}}$

```
repeat
    Choose random point p ∈ Q(γ);
until dim(λ(p)) = k;
C ← {Sλ(p)};
F ← {Sλ(p)};
while F ≠ ∅ do
    Choose O ∈ F;
    F ← F \ {O};
    for O_N with {O, O_N} ∈ E_{G_S(Σ_T(X))} do        /* by Algorithm 5 */
        if O_N ∉ C then
            C ← C ∪ {O_N};
            F ← F ∪ {O_N};
        end
    end
end
return C;
```

3.5 Counterexample: Removal of non-minimal orbit cones

By discarding as many orbit cones as possible in $\Omega_{\mathfrak{a}}^{(k)}$ without affecting the resultant GIT fan, one can speed up the computation of the associated GIT cone $\lambda(p)$ of an arbitrary point $p \in Q(\gamma)$. Clearly, one can discard all orbit cones being a union or intersection of the remaining ones. However, the approach in [10] of removing all non-minimal cones in $\Omega_{\mathfrak{a}}^{(k)}$ with respect to inclusion is flawed. Fortunately, [10, Theorem 1.1] still holds as all non-minimal orbit cones are unions of minimal ones for the $\overline{M}_{0,6}$ example.

Consider the following scenario. Let $f = x_1x_3 + x_2x_4 + x_5 \in \mathbb{K}[x_1, x_2, x_3, x_4, x_5]$. Set $\mathfrak{a} := \langle x_3, x_4 \rangle \cdot \langle f \rangle$ and

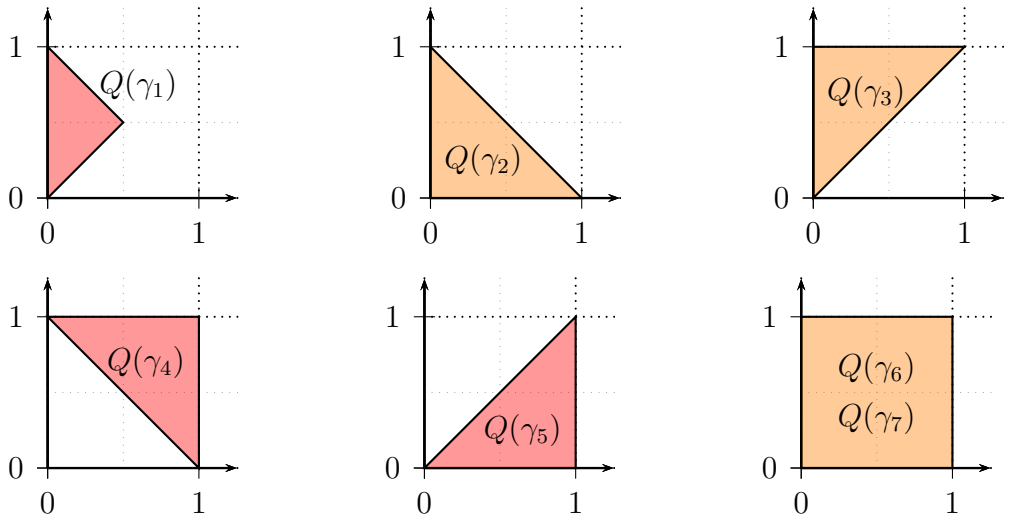
$$Q = (q_1, q_2, q_3, q_4, q_5) := \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 2 \end{pmatrix}.$$

It is easy to see that the matrix Q is of full rank 3 and that f and thus $\mathfrak{a}$ are homogeneous with respect to the $\mathbb{Z}^3$-grading induced by Q. Furthermore, $\mathfrak{a}$ does not contain any monomials. Overall, $\mathfrak{a}$ and Q constitute a valid setup in the sense of section 3.1. The $\mathfrak{a}$-faces with full dimensional image in $Q(\gamma)$ are given by

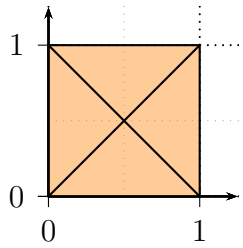
$$\begin{aligned} \gamma_1 &:= \langle e_1, e_2, e_5 \rangle \\ \gamma_2 &:= \langle e_1, e_2, e_3, e_5 \rangle \\ \gamma_3 &:= \langle e_1, e_2, e_4, e_5 \rangle \\ \gamma_4 &:= \langle e_1, e_3, e_4, e_5 \rangle \\ \gamma_5 &:= \langle e_2, e_3, e_4, e_5 \rangle \\ \gamma_6 &:= \langle e_1, e_2, e_3, e_4 \rangle \\ \gamma_7 &:= \langle e_1, e_2, e_3, e_4, e_5 \rangle \end{aligned}$$

and may be computed manually by taking all faces $\gamma_0 \preceq \gamma$ with dimension ≥ 3 and identifying those such that $\mathfrak{a}_{\gamma_0}$ does not contain a monomial.

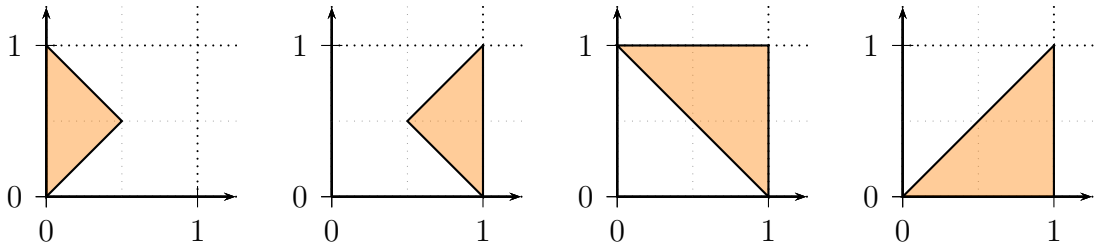
The orbit cones $Q(\gamma_j) \in \mathbb{Q}^3_{z_1,z_2,z_3}$ yield the following polytopes when intersecting them with the $\{z_3 = 1\}$-plane:



When taking all orbit cones into account, the intersection of the resultant GIT fan and the $\{z_3 = 1\}$-plane yields



However, when considering only the red orbit cones, which are minimal with respect to inclusion, all full dimensional intersections of orbit cones containing a common point – that is maximal GIT cones – have the following form when intersecting them with the $\{z_3 = 1\}$-plane:



It is evident that these cannot constitute the maximal cones of a fan. Most of all, we are not able to recover the GIT fan. Note that $Q(\gamma_1)$, $Q(\gamma_6)$ and $Q(\gamma_7)$ are the only orbit cones that may be discarded. They are either the union or intersection of a subset of $Q(\gamma_2), \ldots, Q(\gamma_5)$.

4 Implementation

In the following we present the implementation of the algorithm discussed in the previous chapter. In contrast to the existing Singular implementation gitfan.lib [9], which forms the foundation of our work, the approach described here utilises high performance computing methods in order to speed up the execution by running the algorithm on any number of machines simultaneously. Naturally, one has to identify independent components permitting concurrent execution on the one hand and inherent sequential processes on the other hand. As the sequential components of the algorithm mostly agree with gitfan.lib, we discuss only reworked parts and skip the remaining ones, referring the interested reader to [9]. The chapter is structured as follows: First, we discuss the employed parallelisation framework GPI-Space developed at the *Fraunhofer-Institut für Techno- und Wirtschaftsmathematik* (Fraunhofer ITWM) and follow up with the integration of Singular code into GPI-Space applications. After encoding GIT cone orbits, we present a parallel design of the previously discussed algorithm that is applicable for any kind of fan traversals. Finally, we describe input and output formats, additional features such as speeding up the execution by precomputed results, and executed software tests in order to verify the functional correctness of the program.

4.1 GPI-Space

GPI-Space is a workflow management system developed at the Fraunhofer ITWM which supports the execution of arbitrary workflows on ultra scale systems [18]. It consists of three key components:

- The distributed run-time engine (DRTS), which is responsible for building and managing arbitrary worker topologies based on the available compute nodes, resource management and job scheduling. It supports the reallocation of jobs in case of hardware failures and further integrates dynamically added hardware.
- The workflow engine (WE). It tracks the state of the workflow, identifies a front of activities for which all dependencies are resolved and laces jobs

from input data and active transitions which are then sent to the DRTS for scheduling.

- A virtual memory layer, which provides the necessary infrastructure for the partitioned global address space (PGAS) programming model. It relies on GPI by Fraunhofer ITWM [33].

The framework has been developed with separation of concerns in mind, whereby the concerns here are given by computation and coordination [17]. In our case the computation takes place in sequential Singular-routines. The dependencies and data transfers between this routines, which assemble the particular routines into a complex, parallel algorithm, are described by a special domain language in the the coordination layer. In GPI-Space, this domain language is chosen to be an extension of the well known Petri net model introduced in 1962 by Carl Adam Petri [26]. Its formal nature permits a vast range of analysis and verification techniques. Concurrency is easily described due to locality of states. Furthermore, Petri nets yield precise execution semantics, supporting interruptions and restarts by differentiating between activation and execution of functional units. For this reason, fault tolerance to hardware failures is achieved by simply restarting failed executions. [1]

GPI-Space is shipped with an appropriate compiler that generates all files which are necessary for execution from an XML description of a given Petri net. Furthermore, GPI-Space also provides a basic visualization tool that comes in handy when debugging small to medium sized nets.

Petri nets

Formally, an (unweighted) *Petri net* is a triple (P, T, F) of *places* P, *transitions* T and directed *arcs* $F \subseteq (P \times T) \cup (T \times P)$ relating the former concepts. We demand that $P \cap T = \emptyset$. A function $M\colon P \to \mathbb{N}$ is called *marking* and describes a possible state of the Petri net. If we have $M(p) = k$ for a place p and the current state of the Petri net is given by M, we say that p holds k *tokens*. Visually, circles depict places, rectangles are transitions and arrows between circles and rectangles represent arcs whose directions are indicated by the tip. The current state M of the Petri net is displayed by placing $M(p)$ dots in the circle representing the place p. Figure 1 gives an example for the graphical representation of a Petri net.

A transition t is said to be *active*, iff

$$\forall p \in P\colon (p,t) \in F \Rightarrow M(p) > 0$$

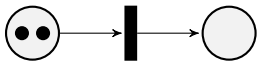


Figure 1: Graphical representation of a Petri net with two places and one transition. The current state is a marking that sends the left place to 2 and the second place to 0.

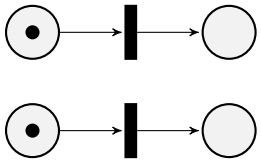


Figure 2: Modelling concurrent processes in a Petri net.

holds. In this case, *firing* t means to transform the current state M into M' by the update rule

$$M'(p) := \begin{cases} M(p) - 1 & (p,t) \in F,\ (t,p) \notin F \\ M(p) + 1 & (p,t) \notin F,\ (t,p) \in F \\ M(p) & \text{else} \end{cases}$$

for all $p \in P$. We interpret this as follows: For every incoming arc (p,t) from p, a token in p is consumed and for every outgoing arc (t,p') to p', a new token is placed into p'. Note that the notion of tokens "moving through transitions" instead of consumption and creation of tokens is erroneous as we will see later on when attaching data to them.

An important property of Petri nets is given by the fact that in many cases the same result is obtained if the order in which two transitions t_1 and t_2 fire is reversed. This allows the modelling of concurrent behaviour. When executing the Petri net model on a machine, the necessary steps for firing t_1 and t_2 respectively may be performed simultaneously. Figure 2 depicts such a situation. Note that Figure 1 also shows an example for concurrent behaviour in which the transition may fire twice. However, the order in which the tokens in the left place are consumed is of no relevance. This example seems trivial as tokens located at the same place are indistinguishable. However, this is not the case anymore when attaching data to tokens. In fact, Figure 1 depicts the common scenario of concurrent behaviour in our algorithm, as parallel execution is mostly achieved by invoking the same routine for varying, independent input data.

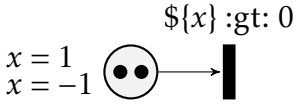


Figure 3: Guards in a Petri net. Only the token with $x = 1$ activates the transition.

Coloured Petri nets and guards

Workflows describe a (possibly concurrent) progression of processes, hence it is evident to model a process by a transition. The input and output data of a process is taken into account by attaching data values of a predefined type to each token. Then every incoming arc provides input data for the process corresponding to the target transition, whereas every outgoing arc describes output data of the source transition that should be put into the target place. A place may only hold tokens of a single type, introducing strong typing to Petri nets. This kind of Petri net is called *coloured Petri net*, developed by Kurt Jensen in 1980. The term originates from the analogy of understanding each distinct type as a unique color. A formal treatment of this concept may be found in [23].

In order to control the data flow, each transition may be tagged with a predicate over the types of places with incoming arcs, which is called *condition* or *guard* [23]. The activation rule for a transition is extended such that not only a token has to be present for every incoming arc, but also the data values of the consumed tokens have to satisfy the condition. GPI-Space provides its own expression language in order to specify conditions. Figure 3 shows a transition that consumes tokens with positive data values only.

Expressions

GPI-Space supports two kinds of transitions. The first kind triggers a module call whenever firing, executing arbitrary C++code where the input and output tokens are mapped to references. The module call is packaged as a job and scheduled by the DRTS. A lightweight approach that requires no scheduling is given by the second kind of transition. The expression language provided by GPI-Space allows to specify the output data values of a transition by means of simple operations on the input data values. This extension is intended to be used for minor computations and saves the overhead of generating and scheduling a job by executing the expression directly within the WE. For instance, counting consumed tokens may be realised that way, see Figure 4. If the counting would be

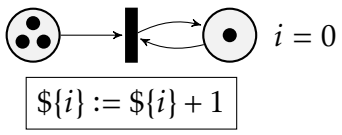


Figure 4: Expressions in a Petri net. After firing the transition three times, the left place contains no tokens anymore and the right place contains one token with $i = 3$.

realised with C++, each increment will cause a disproportionate communication and scheduling overhead.

Executing Petri nets

The WE is responsible for executing a given Petri net. It manages its current state and identifies active transitions. Whenever such a transition is detected, its input tokens are consumed, bundling the attached data values into a job that describes the execution of the transition. Then, this job is submitted to the DRTS where it awaits its execution. The set of active transitions not being executed already is called front of activities and generally contains numerous elements at once due to the locality of Petri nets. Thus, the DRTS is able to utilise the available capacities by scheduling multiple jobs in parallel.

When a job finishes, the results are returned to the WE which extracts the output tokens and updates the state of the Petri net accordingly. Note that during the whole process described above, the input tokens for an active transition are consumed whereas the output tokens are not available yet. In order to fully describe the state of an executable Petri net, one has to incorporate the knowledge about all active transitions and its cached tokens into the state description. Hence, the WE has to manage *timed*, coloured Petri nets. Figure 5 illustrates the process of executing two active transitions in parallel. The module called by the transition sleeps for x seconds, where x is the input data value, and then returns the increment $x + 1$.

The WE itself is executed on a single compute node of the underlying system. Hence, special care must be taken when designing Petri nets. Instead of firing numerous transitions with execution times of milliseconds, one is advised to aggregate these transitions into a single one with an execution time of several seconds. Otherwise, the WE quickly becomes a bottleneck since the amount of scheduled transitions per second does not scale with the number of available compute nodes. Furthermore, the marking of the Petri net is stored in the RAM of a single compute node. If large quantities of data have to be managed, the tokens

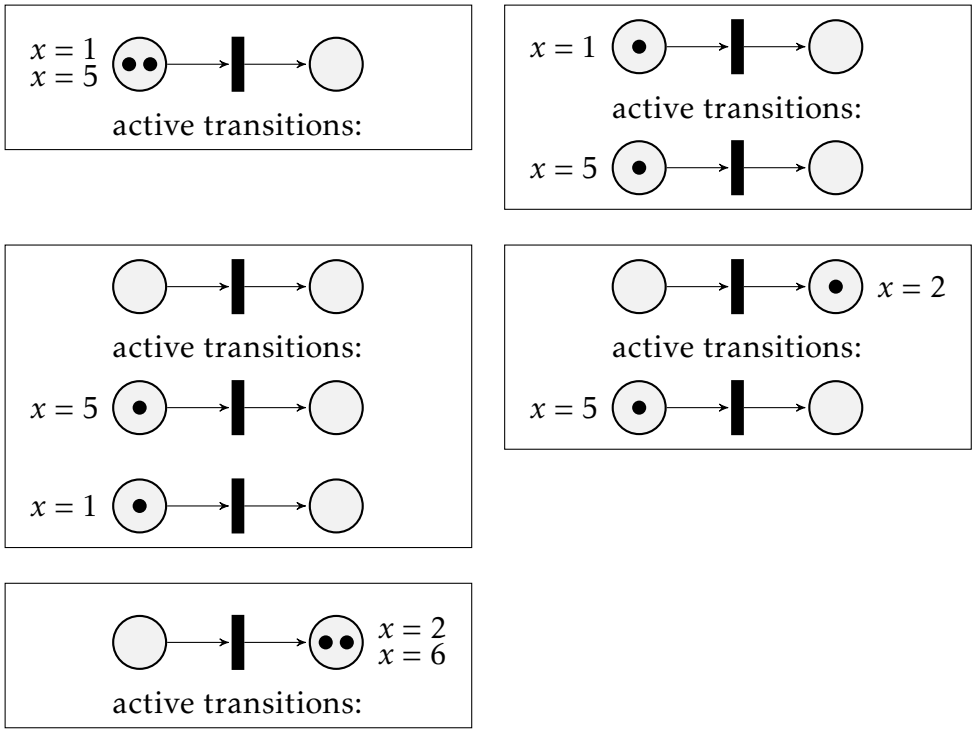


Figure 5: Possible execution of a Petri net. The module called by the transition sleeps for x seconds and then returns the increment $x + 1$.

should only contain a reasonable amount of metadata pointing to the real data which – for instance – is stored in a shared directory.

GPI-Space assumes that module calls have no side effects. Thus, when implementing the execution code for the transitions, possible racing conditions and concurrency issues have to be considered properly. Furthermore, each worker managed by the DRTS is executed in its own process, but beyond that, jobs assigned to the same worker run in the same environment. For this reason, jobs are influenced by changes of global state by prior executions. This behaviour causes several issues when integrating Singular, which is implemented as a large, global state machine.

4.2 Integration of Singular

Singular is a computer algebra system for polynomial computations, with special emphasis on commutative and non-commutative algebra, algebraic geometry and singularity theory [28]. Typically, algorithms utilising Singular are written in its own C-like programming language, which is interpreted at runtime. Besides

providing a console frontend for code execution, Singular also comes with the C library libSingular that implements its full backend. In particular, C methods for interpreting and executing Singular code are available. libSingular also exposes the internal data structures used by Singular, so that one can write conversion methods between Singular types and user defined types, i.e. GPI-Space types.

Since module calls in GPI-Space permit the execution of arbitrary C++ code, every legacy application with a C interface such as Singular can easily be integrated and executed. In order to execute Singular code in a module call, one has to perform the following steps:

1. Check if a Singular instance is already running on the current worker. If not, initialise Singular.
2. Convert GPI-Space types to Singular types.
3. Map the converted data to identifiers in order to address it by Singular code.
4. Formulate a C string containing the Singular code to be executed and pass it to the Singular interpreter via libSingular. Note that instead of hard coding large blocks of code, it is reasonable to outsource them into a Singular library. Then, this step reduces to loading the library and invoking a single method.
5. Extract the data mapped to all identifiers that contain results of the computation.
6. Convert the extracted Singular types to GPI-Space types that can be attached to tokens.

When integrating Singular, we were able to greatly benefit from Ristau's current work [8] on a smoothness test implementation utilising GPI-Space.

Drawback: Global state

Although integrating Singular code is sufficiently easy, the approach described above suffers from a serious issue that originates from Singular heavily relying on global state. For instance, loaded libraries, declared variables and the current basering are managed in global data structures. If module calls are supposed to have no side effects, all changes concerning global state have to be reverted before returning from them. Unfortunately, libSingular lacks the feature of creating and restoring images of its global state. In fact, there seems to be no way of resetting Singular to its initial state short of killing the current process, which – obviously – is not desirable.

Not being able to reset the Singular instance also makes writing decoupled test cases more problematic. In order to avoid tests being influenced by one another due to global Singular state, each test has to be executed in a separate process. Unfortunately, this prevents the utilisation of many Google Test features such as bundling test cases, reusing data configurations and parametrising tests.

Drawback: No in memory serialization of Singular types

When GPI-Space has to share Singular data between module calls without knowing the internal structure of the data, this can be achieved by serialising the data into Binary Large Objects (BLOBs) which then are managed by the WE. Singular supports the serialization of its common data types by using *simple singular interface (ssi) file links*. As the name suggests, the serialization string always is written to a file. It would be desirable to also obtain a serialization string in memory without the overhead of using virtual files or reading the created ssi files. Currently, we pass metadata (that is file paths) pointing to ssi files in a shared directory instead of acquiring and passing BLOBs.

Drawback: Singular programming language lacking data structure implementations

Since the purpose of developing a parallel implementation is the computation of large examples within a reasonable timeframe, it is crucial to choose optimal data structures for large sets of data. Unfortunately, the Singular programming language lacks tree-like data structures with $\mathcal{O}(\log n)$ characteristics and, more importantly, hash tables with average time complexity of $\mathcal{O}(1)$ for insertion, searching and deletion. For this reason, all tasks dealing with the detection of duplicates were moved from Singular into the C++ environment, where the Standard Template Library (STL) provides all necessary data structure implementations. Hence, the set of found GIT cones is managed in C++. The elimination of duplicates in orbit computations as well as the computation of the symmetry group action on GIT cones have been reworked in C++, resulting in a substantial performance gain compared to the gitfan.lib implementation.

4.3 Encoding GIT cone orbits

In chapter 3, we suggested to traverse a graph that reflects the structure of the maximal cones located in the GIT fan. Since nodes of the graph may be reachable

by multiple paths, the traversal algorithm has to manage a list of found nodes such that no node is expanded twice. For this reason it is crucial to develop an encoding of nodes, i.e. GIT cone orbits, that

(i) conserves memory, permitting fast bit-per-bit comparisons,

(ii) allows to recover the encoded object from the code word,

(iii) does not require additional effort when computing code words and

(iv) is easily accessible for operations that are frequently performed on the nodes.

We continue to use the notation from chapter 3. Recall that the matrix $Q \in \mathbb{Z}^{k\times r}$ of rank r with columns $q_1,\dots,q_r$ encodes a torus action of a torus T on the affine variety $X = V(\mathfrak{a}) \subseteq \mathbb{K}^r$ where $\mathfrak{a} \subseteq \mathbb{K}[x_1,\dots,x_r]$ is a homogeneous ideal with respect to the grading $\deg(x_i) = q_i$, $1 \le i \le r$. The positive orthant $\mathbb{Q}^r_{\ge 0}$ is denoted by γ. Furthermore, $\mathcal{S}$ (possibly trivial) denotes a symmetry group of the action of T on X.

Encoding GIT cones

In order to encode GIT cone orbits, we require an appropriate encoding for GIT cones that allows us to apply the group action of $\mathcal{S}$ with minimal costs. The encoding described here originates from [10, Construction 4.3] and has been implemented in GITFAN.LIB. It utilises the result of Lemma 3.3.4, stating that every full dimensional GIT cone is the finite intersection of all full dimensional orbit cones containing the GIT cone. Thus, every GIT cone λ can be uniquely identified by bits indexed over $\Omega^{(k)}_{\mathfrak{a}}$ such that the bit indexed by $\vartheta \in \Omega^{(k)}_{\mathfrak{a}}$ is set to 1 iff $\lambda \subseteq \vartheta$. Formally:

$$\operatorname{code}\colon \Lambda(\mathfrak{a},Q)(k) \to \{0,1\}^{\Omega^{(k)}_{\mathfrak{a}}}, \quad \lambda \mapsto (b_\vartheta)_{\vartheta\in\Omega^{(k)}_{\mathfrak{a}}} \quad \text{with} \quad b_\vartheta = \begin{cases} 1, & \vartheta \subseteq \lambda \\ 0, & \vartheta \not\subseteq \lambda \end{cases}.$$

Obviously, this mapping is injective and yields an encoding with code words of length $|\Omega^{(k)}_{\mathfrak{a}}|$, guaranteeing the afore mentioned property (i). The GIT cone is recovered by intersecting all orbit cones indexing a one in the code word. Hence, (ii) holds.

Next, we verify property (iii). Since every GIT cone $\lambda(p)$ occurring in the algorithm is constructed by taking the relative inner point p and intersecting all orbit cones containing p, the code word can be computed alongside the construction of $\lambda(p)$ due to Lemma 3.3.4.

Finally, in order to verify (iv), we have to check that the group action of $\mathcal{S}$ on the code words defined by

$$\sigma \cdot \mathrm{code}(\lambda) := \mathrm{code}(\sigma \cdot \lambda) = \mathrm{code}(A_\sigma \cdot \lambda)$$

is realised by a low-cost operation on bit strings. Since we have

$$\lambda \subseteq \vartheta \;\Leftrightarrow\; A_\sigma \cdot \lambda \subseteq A_\sigma \cdot \vartheta,$$

the group action simply permutes the characters of code words in the same way as $\mathcal{S}$ acts on $\Omega_{\mathfrak{a}}^{(k)}$. We are going to compute the group action $\mathcal{S} \circlearrowright \Omega_{\mathfrak{a}}^{(k)}$ in the next section, so that it is available during the traversal process. Hence, applying a group element to a code word is implemented as the application of a permutation which is stored in a lookup table.

Extending the encoding to orbits

After encoding GIT cones, we are able to encode GIT cone orbits by mapping each orbit to the code word of an uniquely identified representative. We choose the representative with the minimal code word regarding the lexicographical order over the alphabet $\{0,1\}$. Hence, the encoding is given by

$$\mathrm{code}\colon \Lambda(\mathfrak{a},Q)(k)/\mathcal{S} \to \{0,1\}^{\Omega_{\mathfrak{a}}^{(k)}}, \quad \mathcal{O} \mapsto \min_{\leq_{\mathrm{lex}}} \{\mathrm{code}(\lambda) \mid \lambda \in \mathcal{O}\}$$

and clearly is injective. Property (i) is inherited from the GIT cone encoding. The same applies to property (ii), since we already presented an approach for applying group elements to code words, so that we are able to recover the orbit from a single representative. Property (iii) holds since orbits typically are constructed from a representative and finding the minimal representative is cheap due to property (iv) of the GIT cone encoding. Finally, since operations on orbits usually are given by well defined operations on their representatives, (iv) also is inherited from the GIT cone encoding. One simply applies the operation on the GIT cone that is represented by the code word of the orbit.

4.4 Application flow & Concurrency

We differentiate between three stages when implementing the algorithm described in chapter 3. During the first stage, preprocessing of the input data and the identification of all orbits of $\mathfrak{a}$-faces with full dimensional image takes place. The second stage covers the computation of the orbit cones and the symmetry

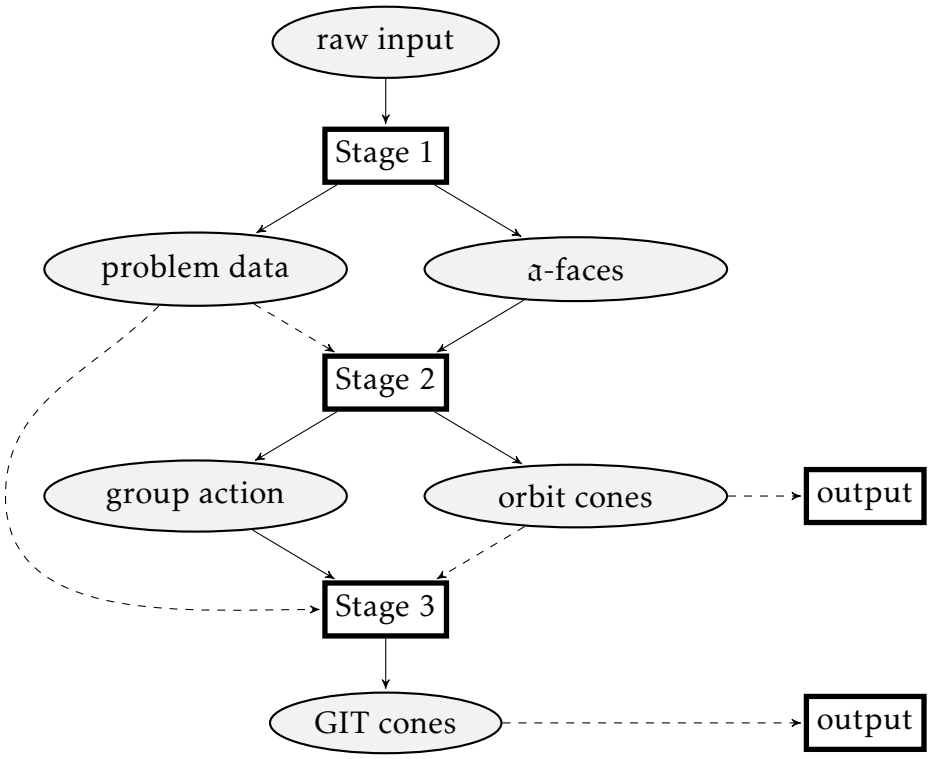


Figure 6: The data flow of the overall algorithm modelled as a Petri net with collapsed stages.

group action on GIT cone-hashes. This data is mandatory when determining the GIT fan in the third stage by traversing all maximal cones of the fan. Figure 6 depicts the simplified data flow of the algorithm in which each stage is collapsed into a single transition. Dashed arcs indicate read-only connections, that is data values of tokens in the source place are passed to the target transition without consuming them when it fires. Hence, *read-only* connections give access to static data computed once (such as processed input data), without discarding it due to consumption.

Stage 1: Preprocessing & $\mathfrak{a}$-faces

Stage 1 starts with preprocessing the raw input supplied by the user. During the process, we compute the following objects sequentially:

- The amount r of variables occurring in the basering and the dimension k of the ambient space in which the GIT fan lives,
- the moving cone (optionally, see section 4.7),

- orbit representatives for the faces of the simplex γ under the symmetry group $\mathcal{S}$ in case it is non-trivial and no set of precomputed orbit representatives is provided (see section 4.7).

In a next step, the set of orbit representatives for the faces of the simplex γ is partitioned into parts of equal size. A token is generated for every part, which then is processed in parallel, selecting only the $\mathfrak{a}$-faces with full dimensional image under Q. Finally, the reduced parts are merged into a single token that then is passed to the next stage. The Petri net modelling this process is depicted in Figure 7.

Since the [partition]-transition has to modify a state token which describes the faces assigned to a part already, it may execute sequentially only. This behaviour is enforced by adding the [partitioner-state]-place with a token describing the initial state of the partitioning process. The [$\mathfrak{a}$-faces]-place is initialised with a token containing an empty list. Since the [merge]-transition is synchronised via this token, it also may execute sequentially only. However, the computationally heavy task of identifying $\mathfrak{a}$-faces is represented by the [select $\mathfrak{a}$-faces]-transition, which is not synchronised. Therefore, any amount of faces may be processed simultaneously.

Stage 2: Orbit cones & group action on encoded GIT cones

After identifying a representative system for the $\mathfrak{a}$-faces with full dimensional image, we are able to determine all full dimensional orbit cones by taking the image under Q and computing orbits under the symmetry group $\mathcal{S}$ acting on $Q(\gamma)$. The following steps are performed in successive order:

1. Compute the cones $Q(\gamma_0)$ for all $\mathfrak{a}$-faces γ_0 returned by stage 1. These images contain a representative system for the set of full dimensional orbit cones $\Omega_{\mathfrak{a}}^{(k)}$ by the proof of Proposition 3.4.4
2. Optionally (see section 4.7): Intersect all obtained cones with the moving cone. Since the moving cone is invariant under the symmetry group $\mathcal{S}$ by Proposition 3.4.5, the resulting cones contain a representative system for $\Omega_{\mathfrak{a}}^{(k)}$ intersected with the moving cone.
3. For each obtained cone σ: Compute the finite orbit $\mathcal{S}\sigma$ and the group action of $\mathcal{S}$ on $\mathcal{S}\sigma$.
4. Eliminate duplicate orbits so that the remaining orbits form a partition of $\Omega_{\mathfrak{a}}^{(k)}$.

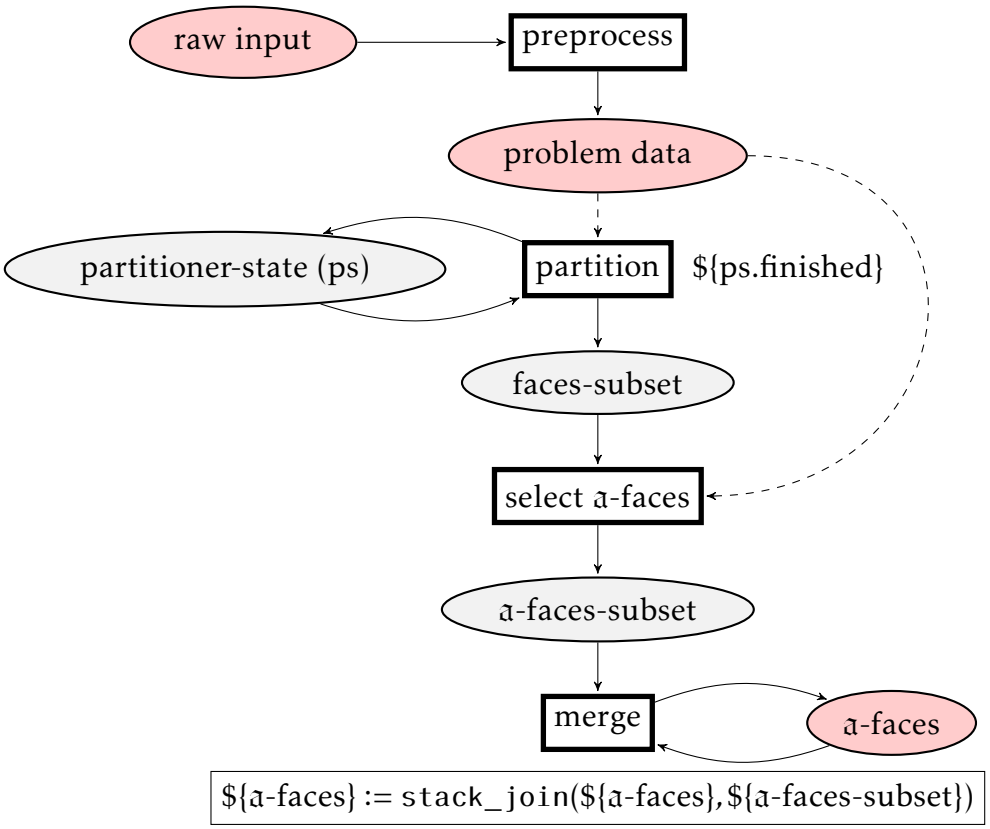


Figure 7: Stage 1 of the algorithm. After preprocessing the input and partitioning all faces, each part is reduced and merged into a single token. Highlighted places correspond to places in figure 6.

5. Aggregate all orbits into a single list containing $\Omega_{\mathfrak{a}}^{(k)}$. Utilise the previously determined group actions on the orbits in order to construct the group action of $\mathcal{S}$ on $\Omega_{\mathfrak{a}}^{(k)}$.

With the exception of Step 5, in all steps a prescribed operation has to be executed on each element of a list of either cones or orbits. Hence, we are able to parallelise the computations in each step by partitioning the list into equally sized parts as described for stage 1. Note that the elimination of duplicates in Step 4 translates to selecting all list entries such that no subsequent list entry equals the selected one.

The most computationally intensive operation is performed in step 3. Every group element is applied to the representative. Since the stabiliser subgroup may be non-trivial, we have to take care of possible duplicates. In order to do so, whenever a new cone is determined, the GITFAN.LIB implementation runs expensive comparisons with all previously found orbit elements and discards the cone if any

matches are detected. We reworked the approach by exchanging the expensive comparison of cones with a string comparison of the cone's serialization, which is one to one for cones returned from Singular's `canonicalizeCone` routine. Furthermore, we manage the found cones in a hash table, using a hash function on the serialization string. Since the maximal orbit size is known to be the cardinality of $\mathcal{S}$, we can adjust the size of the hash table such that hash collisions are unlikely. Hence, the duplicate elimination comes for free by performing only one string comparison on average.

The gitfan.lib implementation also compares cones in order to determine the action of $\mathcal{S}$ on the git cone orbits. However, the action may be constructed from the cayley table of $\mathcal{S}$ as long it is known which group elements map to the same orbit element. For this reason we drop the cone comparisons and compute the action by composing permutations in $\mathcal{S}$, which is a very cheap operation scaling with the amount r of variables occurring in the basering.

Stage 3: GIT fan traversal

Now we are ready to determine the GIT fan. As described in chapter 3, this can be achieved by traversing the graph $\mathcal{G}_{\mathcal{S}}(\Sigma_H(X))$. In order to traverse any undirected, connected graph $G = (V, E)$, the following has to be provided:

- A computable encoding of V, that is an injective function mapping V to binary words.
- A routine `START` that computes an initial node $v_0 \in V$.
- A routine `NEIGHBOURS` computing the function $\mathcal{N} : V \to \mathcal{P}(V)$ which lists all neighbours of a node v, that is $f(v) = \{w \mid \{v, w\} \in E\}$.

In our case, we use the GIT cone orbit encoding described in section 4.3. `START` is implemented as in gitfan.lib, that is selecting random points $p \in Q(\gamma)$ until the corresponding GIT cone $\lambda(p)$ has full dimension. Then, `START` returns the GIT cone orbit-code of $\mathcal{S}\lambda(p)$. The routine `NEIGHBOURS` is given by Algorithm 5.

It remains to model a Petri net describing a parallel graph traversal. Note that this model is applicable for any kind of fan traversal, e.g. in the context of tropical varieties [11] and Gröbner fans [32, chapter 3], as long as the above mentioned demands are satisfied. Figure 8 depicts an appropriate Petri net together with a storage interface that allows to record visited nodes outside of the WE. The highlighted [data]-place combines the results of stage 2, which are required by the `START` and `NEIGHBOURS` routines for the GIT fan application.

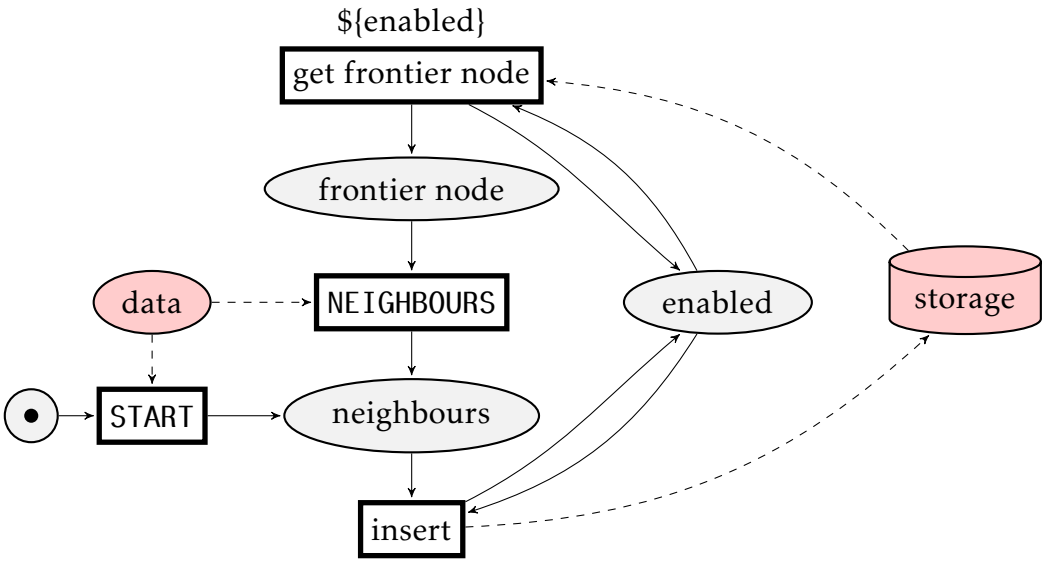


Figure 8: Modelling a graph traversal algorithm as a Petri net. Highlighted elements correspond to places in figure 6.

We distinguish between two types of nodes. *Frontier nodes* are known by the traversal process. However, their neighbours have not been calculated yet. Hence, frontier nodes form the boundary of the ongoing graph traversal. The boundary is expanded by computing the neighbours of a frontier node, obtaining a set of frontier nodes. Nodes whose neighbours have been determined already are called *expanded nodes*.

The storage interface has to provide insertion and retrieval operations, which are used by the [get frontier node]- and [insert]-transition respectively. Whenever a node should be inserted, the storage implementation has to check if it is an unknown node, meaning that is does neither occur in the set of frontier nodes, nor in the set of expanded nodes. In this case the node is inserted into the set of frontier nodes. Otherwise, the node is discarded in order to avoid repeated expansion of the same node. A boolean return value denotes whether an insertion took place. The retrieval operation returns a frontier node and relocates it to the set of expanded nodes. If the set of frontier nodes is empty, no node is returned.

The graph traversal is initiated by computing an initial node, executing `START`. The incoming place of `START` containing exactly one token ensures that the transition fires once only. The initial node is put into the [neighbours]-place so that it is published to the storage interface as the first frontier node when the [insert]-transition fires.

[get frontier node] frequently polls the storage interface for frontier nodes and generates tokens for each returned node. It is synchronised via the [enabled]-place containing a single token, preventing simultaneous access to the storage implementation that may not be thread safe. As soon as the storage interface does not return any frontier node, [get frontier node] disables itself by setting the value of the [enabled]-token to `false`.

The tokens representing frontier nodes are processed by computing adjacent nodes in [`NEIGHBOURS`]. Then, the results are inserted by invoking the storage interface. Note that the [insert]-transition also is synchronised via the [enabled]-place in order to prevent simultaneous access to the storage implementation. Whenever the storage interface indicates that at least one node has been inserted, the value of the [enabled]-token is set to `true`, reactivating [get frontier node].

The traversal is finished when no frontier nodes are available anymore and no nodes are processed currently. These conditions are met when the Petri net described in Figure 8 deadlocks. In our implementation, we extended the Petri net by detecting such deadlocks and outputting all nodes, i.e. GIT cone orbits, then.

4.5 Storage implementations used in graph traversals

In the previous section we described a parallel graph traversal algorithm in order to compute the GIT fan. This algorithm depends on a storage interface which is accessed synchronously. On that account, it poses a potential performance issue, slowing down the computation up to a point where all but one process are blocked due to storage access. For this reason it is of utmost importance that the storage implementation is able to handle hundreds of requests during a single call to `NEIGHBOURS`. Furthermore, it has to cope with arbitrary large sets of nodes. We pursue two approaches, storing the nodes either in a shared directory on disk or in memory on a single compute node.

4.5.1 Storing nodes on disk

Our first approach stores nodes by exploiting the scalable cluster file system BeeGFS, developed at the Fraunhofer ITWM [5]. Every stored node is represented by an empty file with a path that depends on the binary representation of the node. Thus, searching for nodes is realised by the system call `fstat`, whereas insertions are realised with `touch`. This implementation offers the following advantages:

- It is easy to implement since BeeGFS already is installed on the cluster system of the Fraunhofer ITWM. Mirroring the storage directory on every compute node of the cluster system is realised by BeeGFS and hidden from the application.
- Straightforward recovery in case of failures, e.g. power failures. The current state of the graph traversal is saved in persistent storage at any time.
- Practically infinite scaling due to BeeGFS.

However, load tests have shown that the response time by BeeGFS are far too high, taking up over 90% of the total computation time when computing the GIT fan of the $\overline{M}_{0,6}$ example on a moderate amount of 10 compute nodes with 16 cores each. For this reason we provide another, less scalable implementation with better response times that easily suffices the needs for computing the $\overline{M}_{0,6}$ example.

4.5.2 Storing nodes in memory

Instead of storing nodes on disk, we host an RPC-server on a single compute node of the cluster. This server implements the interface by managing the sets of frontier and expanded nodes in a hash table implementation provided by the STL. Consequently, all nodes sent to the RPC-Server are stored in the RAM of a single machine. Since insert and search operations perform in $\mathcal{O}(1)$ as long as enough memory is available, this approach scales until a certain threshold is passed.

In the $\overline{M}_{0,6}$ example we have to store up to 249 604 entries of 106 bits each, summing up to about 3 MB of data. Thus, we are far away from reaching the limit of up-to-date RAM sizes. If one intends to run graph traversals consuming more memory than being available on a single machine, the current storage solution may be extended by an implementation of a distributed hash table such as *Chord*, utilising the RAM of every available compute node. [31]

When considering response timings, the second approach performs significantly better than the implementation using BeeGFS. Overhead induced by awaiting responses of the RPC-server is negligible even when executing the algorithm on 40 compute nodes with 16 cores each.

4.6 Input & Output structure

The application is controlled by various command line parameters. In particular, the user provides command line argument values specifying the input data and

the output directory. In the following we describe all parameters concerning input and output:

`ideal`: The path to a file which contains the vanishing ideal $\mathfrak{a}$ in the ssi format. The monomial ordering associated to the basering over which $\mathfrak{a}$ is defined has to be a global, weighted reverse lexicographical ordering such that the generators of $\mathfrak{a}$ are homogeneous with respect to the weight vector (see Remark 3.2.5).

`torus-action`: The path to a file which contains the matrix Q encoding the torus action in the ssi format.

`symmetry-group (optional)`: The path to a file which contains the symmetry group S as a list of permutation-types as defined in GITFAN.LIB. The data has to be present in the ssi format. If no path is supplied, the trivial group is assumed.

`working-directory`: A non-existent directory which the application uses as working and output directory.

When the application terminates, the following files have been written to the working directory:

`args`: This file logs all command line argument values that have been passed to the application.

`rev`: This file contains the SHA1 git commit hash of the current build.

`load_git_cones.sing`: A SINGULAR script that may be piped into a SINGULAR instance in order to load all orbit and GIT cones into the variables `ORBIT_CONES` and `GIT_CONES`.

`load_git_cones.lib`: A helper library that is used by the previously mentioned script.

Furthermore, a subdirectory `result` is created, containing the following:

`orbit_cones`: This subdirectory contains all orbit cones, written to files in the ssi format, and the file `list`. Each line of `list` describes the file name for a single orbit cone and thus induces an order on the set of orbit cones. This order has been used when encoding GIT cones.

`git_cones`: This file contains all full dimensional GIT cones encoded as bit strings (see section 4.3)). Groups of four bits are combined to a hex digit. The strings are separated by new lines.

`symmetry_group_on_hashes`: This file contains the group action of S on the GIT cone code words. Similarly to the `symmetry-group` parameter, the action is

stored as a list of permutations on the set of orbit cones ordered by the `list` file.

Arguments for GPI-Space

The following parameters configure the GPI-Space environment:

`nodefile`: A path to a nodefile describing all available compute nodes. In a cluster management and job scheduling system such as SLURM [29] a nodefile may be obtained by `generate_pbs_nodefile`.

`rif-strategy`: Selects the strategy used by the Remote Interface Daemon (RIFD). Currently, `ssh` and `pbsdsh` are available. We use `ssh`.

`rif-strategy-parameters`: Additional parameters passed to the bootstrapping routine of the RIFD strategy. We do not supply any argument.

`rif-port`: Specifies a port on which RIFD listens. We do not supply any argument.

`worker-per-node`: The amount of workers that are set up per compute node. In our case, we set this value to 16, matching the amount of cores per compute node.

4.7 Additional features

The application includes several extensions such as intersecting with the moving cone, incorporating precomputed data, computing only a predefined amount of GIT cones, clustering calculations, configuring the storage implementation and logging performance records. All features are accessible via command line arguments. An overview of all supported arguments is obtained by typing `--help`.

Intersecting with the moving cone

Instead of computing the GIT fan with support $Q(\lambda)$, the application may also compute the intersection of the GIT fan with the moving cone for Q. In order to do so, the user has to supply a file path for the optional command line argument `moving-cone`. If the path refers to a non-existent file, the application computes the moving cone as in Remark 5.3.2 and outputs it to the given path. If the path refers to an existent file instead, the application assumes that the file contains the precomputed moving cone in the ssi format and loads it instead.

Acceleration by using precomputed data

One can not only supply a precomputed moving cone, but also a precomputed set of all orbit cones including the group action of S on this set. Hence, the first two stages of the algorithm may be skipped. If a path for the optional command line argument `precomputed-orbit-cones` is supplied, the file associated to this path is assumed to contain a list of all orbit cones. Each line in this file has to describe a path, relative to the supplied path, that represents a file containing an orbit cone in the ssi format. Note that the `symmetry-group` parameter is ignored in this case. Instead, the `precomputed-symmetry-group-on-hashes` parameter can be used in order to specify a precomputed group action of S on the GIT cone code words. If no value is supplied, the trivial group action is assumed.

Finally, the parameter `simplex-orbit-representatives` allows to provide the application with precomputed orbit representatives with respect to the group action of S on the faces of γ. Each line constitutes a single representative by containing a comma separated list of integers. The integers define which of the rays e_i generate the corresponding face. Although this parameter is optional, it is highly advised to determine the orbit representative with GAP [16] beforehand, since the current version of the application does not implement an efficient, parallelised approach for computing the representatives.

Clustering calculations

In order to reduce the load on the WE, one can modify the value of the `job-size` parameter, clustering more calculations into a single execution of a transition. As a downside, increasing this value may reduce the degree of parallelism if the amount of available cores exceeds the number of available jobs divided by the job size.

Aborting the GIT fan traversal

The GIT fan traversal may be aborted after expanding a predefined amount of GIT cone orbits that is given by the `max-cones` parameter. On that account the third stage of the algorithm may be skipped by setting `max-cones` to zero. Negative values are considered to be infinity. This feature mainly has been used for debugging purposes.

Selecting and configuring the storage implementation

We provide embedded command line arguments in order to select and configure the storage implementation used during the GIT fan traversal. All values for the following parameters have to be provided in a block starting with `+CONESTORAGE` and terminating with `-CONESTORAGE`.

implementation: Selects the utilised storage implementation (see section 4.5). One of `rpc`, `linear` or `constant`. `rpc` refers to the in-memory solution. `linear` and `constant` refer to the BeeGFS solution, where `linear` consumes less memory than `constant` but requires linear time in order to determine inserted nodes that are frontier nodes. If no value is specified, this parameter defaults to `rpc`.

subdirectory-name-length: This parameter modifies the depth of the resultant file tree created by the BeeGFS solution. For every `subdirectory-name-length` characters in a GIT cone orbit code word, a new level is introduced.

A possible configuration of the storage implementation is given by the following command line:

```
+CONESTORAGE --implementation=constant
 --subdirectory-name-length=4 -CONESTORAGE
```

Logging performance records

The execution time of all module calls and the total program runtime are measured and stored in the directory given by the `logs` parameter of the embedded command line beginning with `+LOGGING` and terminating with `-LOGGING`. The application comes with a `generate_statistics` binary that is able to process the stored records and generate statistics from it.

4.8 Software testing

Throughout the development of the application, we thoroughly implemented and maintained automated functional unit tests by using the Google Test-framework. We differentiate between three types of tests.

Component tests verify that the implementations of utility classes used in module calls display the correct behaviour. For instance, the storage implementations, logging classes, Singular wrapper routines, type conversion routines and a LRU Cache implementation are automatically tested.

Module call tests check all module calls that occur in the modelled Petri nets. Each test loads a set of valid sample data derived from the $\overline{M}_{0,5}$ example (see chapter 5 or [10, Example 5.2]) and invokes a single module call, passing in the loaded data. Then the returned values are checked against the expected behaviour.

System tests execute the complete application or the Petri nets of each stage separately. The input data is derived from the $\overline{M}_{0,5}$ example with a trivial symmetry group and a $\mathcal{S}_5$-action respectively. Furthermore, we check if the first two stages are skipped when supplying precomputed data as described in section 4.7. The routines for verifying the returned results do not depend on the output order of the orbit cones and thus are not susceptible to indeterministic behaviour due to job scheduling.

All input data sets and expected data sets have been computed by using the GITFAN.LIB implementation. Hence, in the $\overline{M}_{0,5}$ test scenario our calculations comply with the results returned by GITFAN.LIB.

4.9 Performance

In order to evaluate the efficiency and scalability of our implementation, we executed the algorithm for the $\overline{M}_{0,6}$ example on the cluster of the Fraunhofer ITWM. Each compute node is operated by a *Dell PowerEdge M620 Blade Server* with the following specification:

- dual Intel Xeon E5-2670 ("Sandy Bridge") – i.e. (physical) 16 CPU cores
- 64 GB RAM
- 300 GB HDD
- 2x Gigabit Ethernet
- FDR Infiniband

The total runtime of the algorithm as well as the cumulated processing time of all workers have been recorded for varying numbers of available compute nodes. Hence, we are able to compute the degree of utilisation for every configuration. Furthermore, we differentiated between the first two stages, computing the orbit cones and the group action on encoded GIT cones, and the third stage, traversing the GIT fan. The reasoning behind this becomes clear when analysing the results, since the traversal permits a significantly higher degree of parallelisation than the orbit cone calculation. Note that the cumulated runtime of all stages is in accordance with the runtime of the algorithm without running the third stage separately. We executed all stages at once on five compute nodes and concluded

# nodes	# cores	timing/s	timing/min	speedup
1	1	2561	~43	1
1	16	507	~8	5.0
2	32	499	~8	5.1
3	48	498	~8	5.1
4	64	500	~8	5.1
5	80	497	~8	5.1

Table 4.1: Timings for computing the orbit cones of the $\overline{M}_{0,6}$ example with varying numbers of CPU cores.

that the difference in measured runtime compared to the cumulated runtime after separating the third stage is insignificant.

Computation of the orbit cones

We computed the orbit cones from the reduced ideal and torus action of the $\overline{M}_{0,6}$ example, eliminating the variables corresponding to Keel-Vermeire divisors. Hence, we consider a ring with 25 variables only and the orbit cones live in a 16 dimensional ambient space. Furthermore, we supplied the $\mathcal{S}_6$-group action as symmetry group and precomputed orbit representatives for the faces of the simplex $\mathbb{Q}^{25}_{\geq 0}$. We set the `job-size` parameter to 100. Finally, we also enabled the computation of the moving cone in order to determine the Mori chamber decomposition of the cone of movable divisor classes. All executions yielded 106 orbit cones that partition into two orbits of size 45, one orbit of size 15 and one orbit of size one. These display the same characteristics as in [10, Remark 6.7], which is expected as the GITFAN.LIB implementation is not supposed to return different results. Note that the orbit of size one consists of the moving cone. GITFAN.LIB discards this orbit since the moving cone contains other orbit cones as proper subsets. The timing results are shown in Table 4.1.

Compared to a sequential execution on one core, we obtain a maximal speedup of factor 5. This is due to the fact that the costly operation of computing an orbit for a representative – represented by Step 3 in stage 2 – is implemented as a sequential operation. Hence, we obtain a theoretical speedup of 9, which is the amount of orbits that have to be computed. Since the required time for computing different orbits varies greatly between 226 and 312 seconds due to different numbers of facets for each orbit cone, we obtain a lower speedup when joining the results of

# nodes	# cores	timing/s	timing/min	speedup	utilisation
1	16	25116	~419	1	0.996
5	80	5224	~87	4.8	0.990
10	160	2635	~44	9.5	0.974
15	240	1842	~31	13.6	0.911
20	320	1414	~24	17.8	0.885
25	400	1189	~20	21.1	0.837
30	480	979	~16	25.6	0.845
35	560	895	~15	28.0	0.790
40	640	775	~13	32.4	0.799

Table 4.2: Timings for traversing the GIT fan of the $\overline{M}_{0,6}$ example with varying numbers of CPU cores.

Step 3. Further potential speedup is lost due to the large job size, which relieves the WE when performing the a-face-test for all of the 6993 representative of the faces of γ, but prohibits parallel execution when intersecting the remaining orbit cone representatives with the moving cone. We expect a higher speedup for problems with more orbit cone orbits.

GIT fan traversal

In order to profile the GIT fan traversal, precomputed values for the moving cone, the orbit cones and the group action on encoded GIT cones are provided (see section 4.7). We set the `job-size` parameter to 10 and use `rpc` as storage implementation. The timing results are shown in Table 4.1.

Figure 9 and 10 indicate that our algorithm scales almost linearly up to 40 compute nodes with a total of 640 cores. Only slight losses concerning utilisation have been recorded. We noted that on average only 560 of the 640 available cores run module code simultaneously, implying that the WE begins to strain when generating jobs for 40 compute nodes. Furthermore, a minute elapses before the set of frontier nodes has grown to a reasonable size such that enough jobs can be generated in order to utilise the majority of all cores. The timing diagram displayed in Figure 11 matches a hyperbolic curve, which is in accordance with the linear speedup we observed previously.

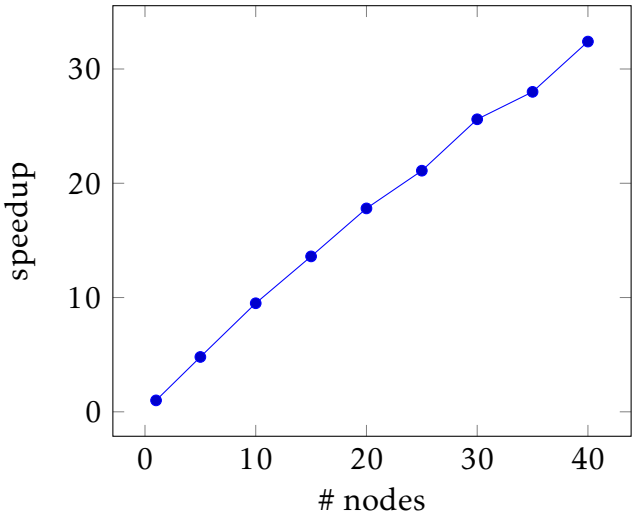


Figure 9: Observed speedup when traversing the GIT fan with varying numbers of compute nodes.

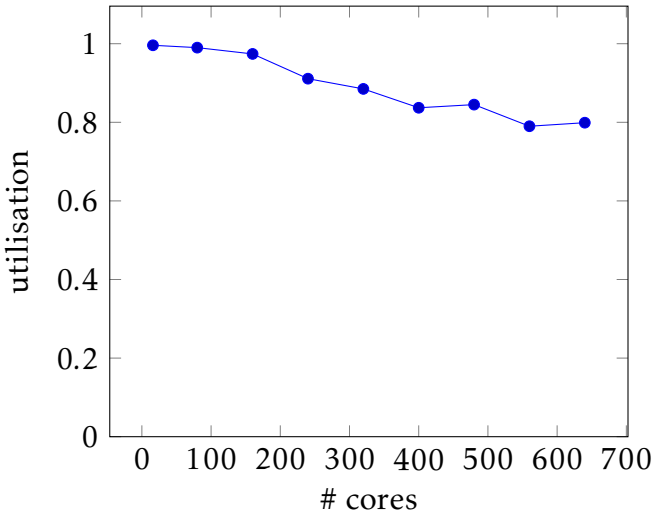


Figure 10: Observed utilisation when traversing the GIT fan with varying numbers of compute nodes.

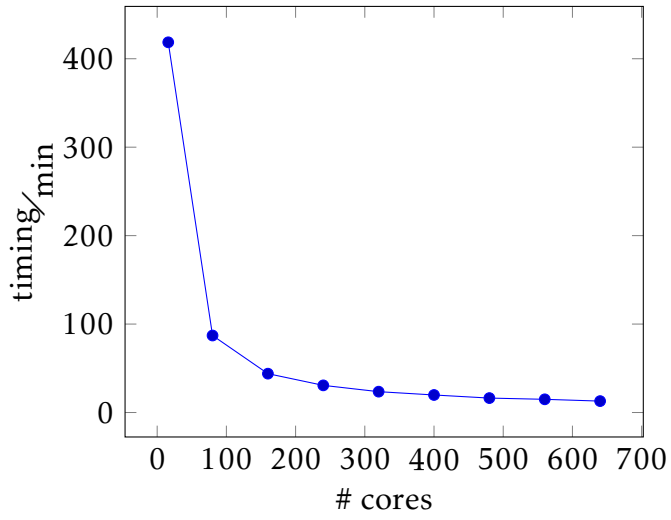


Figure 11: Required computation time when traversing the GIT fan with varying numbers of compute nodes.

5 Application: Moduli space of n-pointed stable curves

In order to profile and test our implementation, we are going to compute the Mori chamber decomposition of the moduli space $\overline{M}_{0,n}$ of n-pointed stable curves of genus 0 for $n = 5, 6$. Essentially, this is achieved by determining the GIT fan of a graded divisoral algebra, where the grading specifies the group action on its variety. This chapter is devoted to this specific application. First, we give a concise introduction of the moduli space $\overline{M}_{0,n}$. In the second section, we define Mori dream spaces and remark that $\overline{M}_{0,n}$ indeed is such a space. Finally, we present the notion of cones of divisors and Mori chamber decompositions by means of $\overline{M}_{0,n}$.

For the remainder of this chapter, fix $n \in \mathbb{N}$.

5.1 Moduli space of n-pointed stable curves of genus 0

We consider the moduli space $M_{0,n}$ of n-pointed rational curves of genus 0 up to isomorphism, that is $M_{0,n}$ consists of classes of (ordered) configurations $(C, p_1, \ldots, p_n)$ with C being a smooth, projective curve of genus 0 and the p_i are distinct, smooth points located on C. Two configurations $(C, p_1, \ldots, p_n)$ and $(C', p_1', \ldots, p_n')$ are said to be equivalent iff there exists an isomorphism $\varphi : C \longrightarrow C'$ such that $\varphi(p_i) = p_i'$ for all $1 \leq i \leq n$.

We give a concrete description of $M_{0,n}$. Each smooth, projective curve of genus 0 is isomorphic to $\mathbb{P}^1$ [20, Example IV.1.3.5]. For this reason any configuration in $M_{0,n}$ may be represented by $(\mathbb{P}^1, p_1, \ldots, p_n)$ with $p_i \in \mathbb{P}^1$ and $p_i \neq p_j$ for $i \neq j$. It is well known that for any triple (q_1, q_2, q_3) of distinct points there exists exactly one automorphism of $\mathbb{P}^1$ sending it to $(0, 1, \infty)$. By applying the automorphism sending the first three points of our configuration to 0, 1 and ∞ respectively, we obtain a unique representative $(\mathbb{P}^1, 0, 1, \infty, p_4', \ldots, p_n')$ such that p_4' up to p_n' are distinct points contained in $\mathbb{P}^1 \setminus \{0, 1, \infty\}$. By going over to the affine chart $\mathbb{P}^1 \setminus \{\infty\} \cong \mathbb{A}^1$ of $\mathbb{P}^1$, we conclude that

$$M_{0,n} \cong \{(p_1,\dots,p_{n-3}) \in (\mathbb{C}\setminus\{0,1\})^{n-3} \mid p_i \neq p_j \text{ for } i \neq j\}.$$

By alleviating the condition on the curve C, Deligne and Mumford were able to construct a compactification $\overline{M}_{0,n}$ that does not only parametrise n-pointed smooth curves, but also the following n-pointed stable curves. [14]

Definition 5.1.1 ([30, section 1.2]) An *n-pointed stable curve of genus* 0 is a configuration $(C, p_1,\dots,p_n)$ where C is a connected, projective curve and the marked points $p_i \in C$ are distinct and smooth such that

(i) Every irreducible component of C is isomorphic to $\mathbb{P}^1$,

(ii) Irreducible components intersect in nodal points. A nodal point is an ordinary double point that is passed by exactly two branches of C with different tangent lines.

(iii) If δ is the number of nodal points, there are $\delta+1$ irreducible components.

(iv) Every irreducible component contains at least three of the marked or nodal points on C.

Remark 5.1.2 Note that all curves C satisfying the conditions (i)-(iii) in Definition 5.1.1 have arithmetic genus 0. Furthermore, any automorphism of C fixing all marked points is trivial, hence the name "stable". The set $M_{0,n}$ appears as open subset in $\overline{M}_{0,n}$ and corresponds to n-pointed stable curves which are smooth.

5.2 Divisors, Cox rings & Mori dream spaces

We introduce the notions of Cox rings and Mori dream spaces, cones of divisors and chamber decompositions as in [2]. They relate to $\overline{M}_{0,n}$ in the sense that $\overline{M}_{0,n}$ is a Mori dream space for which we compute the Mori chamber decomposition. Let X be an irreducible, normal prevariety over an algebraically closed field $\mathbb{K}$ of characteristic zero.

Definition 5.2.1 (Divisor) A *prime divisor* of X is an irreducible subvariety $D \subseteq X$ of codimension 1. The free abelian group generated by all prime divisors of X is called the *divisor group* of X, denoted by $\mathrm{Div}(X)$. Its elements are called *divisors*.

The *support* of a divisor $D = z_1D_1 + \cdots + z_sD_s$ with prime divisors D_i and integers z_i is given by

$$\mathrm{Supp}(D) := \bigcup_{z_i \neq 0} D_i.$$

The divisor D is called *effective*, denoted by $D \geq 0$, if $z_i \geq 0$ for all i.

Note, that for us, divisors are weil but not necessarily cartier.

Each prime divisor D corresponds to a generic point η_D of X. Since X is normal, its local ring $\mathcal{O}_{X,\eta_D}$ becomes a discrete valuation ring with discrete valuation $\nu_D\colon K(X)^* \to \mathbb{Z}$. For this reason, any nonzero rational function $f \in K(X)^*$ defines a divisor as follows:

Definition 5.2.2 Let $f \in K(X)^*$. Then we define

$$\operatorname{div}(f) := \sum_{D \text{ prime}} \nu_D(f) \cdot D.$$

Divisors of the form $\operatorname{div}(f)$ are called *principal divisors*. The set of all principal divisors is denoted by $\operatorname{PDiv}(X)$.

Remark 5.2.3 The map $\operatorname{div}\colon K(X)^* \to \operatorname{Div}(X)$ is well defined and a group homomorphism. In particular, $\operatorname{PDiv}(X)$ is a subgroup of $\operatorname{Div}(X)$. The factor group

$$\operatorname{Cl}(X) := \operatorname{Div}(X) \big/ \operatorname{PDiv}(X)$$

is called the *divisor class group* of X.

Definition 5.2.4 Let $U \subseteq X$ be open. We define the restriction $\operatorname{Div}(X) \to \operatorname{Div}(U)$ by the $\mathbb{Z}$-linear extension of

$$D\big|_U := \begin{cases} D \cap U, & D \cap U \neq \emptyset \\ 0, & \text{else} \end{cases}$$

for any prime divisor D of X.

Definition 5.2.5 Let $D \in \operatorname{Div}(X)$. Then we define the sheaf $\mathcal{O}_X(D)$ of $\mathcal{O}_X$-modules as follows:

$$\mathcal{O}_X(D)(U) := \{f \in K(X)^* \mid (\operatorname{div}(f) + D)\big|_U \geq 0\} \cup \{0\} \quad \forall U \subseteq X \text{ open.}$$

The restriction map is given as in Definition 5.2.4. Note that we have

$$\mathcal{O}_X(D_1)(U) \cdot \mathcal{O}_X(D_2)(U) \subseteq \mathcal{O}_X(D_1 + D_2)(U)$$

for any open subset $U \subseteq X$ and divisors $D_1, D_2 \in \operatorname{Div}(X)$.

Definition 5.2.6 (Movable divisor) Let D be a divisor of X. The *base locus* of D is defined by

$$\operatorname{Bs}|D| := \bigcap_{f \in \mathcal{O}_X(D)(X)} \operatorname{Supp}(\operatorname{div}(f) + D)$$

and the *stable base locus* is

$$\mathbf{B}\,|D| := \bigcap_{n\in\mathbb{N}} \mathrm{Bs}\,|nD|.$$

We say that D is *movable* iff its stable base locus is of codimension at least 2 in X, that is, it does not contain any prime divisors.

Definition 5.2.7 (Cox ring) Let X be such that its divisor class group $\mathrm{Cl}(X)$ is finitely generated and there exists a subgroup K of $\mathrm{Div}(X)$ that is a representative system of the orbits of $\mathrm{Div}(X)$ w.r.t. $\mathrm{PDiv}(X)$. Then the *Cox sheaf* of X is the sheaf of divisoral algebras given by

$$\mathcal{R} := \bigoplus_{D\in K} \mathcal{R}_{[D]} = \bigoplus_{[D]\in \mathrm{Cl}(X)} \mathcal{R}_{[D]}, \qquad \mathcal{R}_{[D]} := \mathcal{O}_X(D).$$

Up to isomorphy, it does not depend on the choice of K. The *Cox ring* of X is the $\mathrm{Cl}(X)$-graded algebra of global sections $\mathcal{R}(X)$.

In the construction of Cox rings presented here, we do rely on the existence of a subgroup of $\mathrm{Div}(X)$ being a representative system. In general, it suffices to demand a subgroup of $\mathrm{Div}(X)$ such that the equivalence classes of its elements cover all $\mathrm{Cl}(X)$, see [2, Construction 1.4.2.1].

Definition 5.2.8 (Mori dream space) Let X be an irreducible, normal projective variety. It is called a *Mori dream space* if its divisor class group $\mathrm{Cl}(X)$ and its Cox ring $\mathcal{R}$ are finitely generated.

Remark 5.2.9 Whereas the Deligne-Mumford compactification $\overline{M}_{0,n}$ of the moduli space of n-pointed stable curves of genus 0 is no Mori dream space for $n \geq 10$ [21, Addendum 1.4], for $n = 5, 6$, its Cox ring is finitely generated. A set of homogenous generators of $\mathcal{R}(\overline{M}_{0,5})$ is identified by [4] since $\overline{M}_{0,5}$ is a del Pezzo surface [7]. In [12], Castravet determines a set of homogenous generators for $\mathcal{R}(\overline{M}_{0,6})$. Bernal Guillén [7] then computes the relations between the generators so that a presentation of the Cox rings in terms of a zero locus is obtained. Furthermore, she provides an explicit description of the symmetry group action on $\mathcal{R}(\overline{M}_{0,5})$ and $\mathcal{R}(\overline{M}_{0,6})$ respectively that arises from permuting marked points in a configuration, that is

$$\sigma \cdot (C, p_1, \ldots, p_n) := (C, p_{\sigma(1)}, \ldots, p_{\sigma(n)}) \quad \forall \sigma \in \mathcal{S}_n.$$

Hence, we obtain embeddings of the Cox rings as in 2.1.5. The torus action is defined by the grading living in the divisor class group, which is a finitely generated subgroup of some $\mathbb{Z}^d$. This defines the matrix Q in our setup of section 3.1. The vanishing ideal $\mathfrak{a}$ is given by the relations between the generators. Bernal Guillén's description of the symmetry group action extends our setup in the sense of section 3.4.

5.3 Mori chamber decomposition

In this section we refer to $\overline{M}_{0,n}$ as a Mori dream space and tacitly assume that $n = 5$ or $n = 6$. With Remark 5.2.9 in mind, we see that the lattice points of the weight cone of $\mathcal{R}(\overline{M}_{0,n})$ as in section 2.4 are in fact divisor classes of $\overline{M}_{0,n}$. It lives in the ambient space $\mathrm{Cl}(\overline{M}_{0,n}) \otimes_{\mathbb{Z}} \mathbb{Q}$. For this reason, all cones arising in section 2.4 are considered to be cones of divisors classes such that its lattice points are elements in $\mathrm{Cl}(\overline{M}_{0,n})$. We are interested in two particular cones:

Definition 5.3.1 The *cone of effective [movable] divisor classes* of $\overline{M}_{0,n}$ is given by

$$\mathrm{Eff}(\overline{M}_{0,n}) := \langle [D] \otimes 1 \mid D \in \mathrm{Div}(\overline{M}_{0,n}) \text{ effective} \rangle_{\mathbb{Q}_{\geq 0}},$$
$$\mathrm{Mov}(\overline{M}_{0,n}) := \langle [D] \otimes 1 \mid D \in \mathrm{Div}(\overline{M}_{0,n}) \text{ movable} \rangle_{\mathbb{Q}_{\geq 0}}.$$

Note that the sets of effective and movable divisors respectively are recovered by intersecting the above convex cones with $\mathrm{Cl}(\overline{M}_{0,n}) \otimes 1$. We also call the cone of movable divisor classes the *moving cone*.

Remark 5.3.2 By [2, Proposition 3.3.2.1], the cone of effective divisor classes of $\overline{M}_{0,n}$ coincides with the weight cone of $\overline{M}_{0,n}$. Furthermore, in the setup of section 3.1, the moving cone is easily computed by intersecting all cones generated by all but one column of Q, that is

$$\mathrm{Mov}(\overline{M}_{0,n}) = \bigcap_{\gamma_0 \preceq \gamma \text{ facet}} Q(\gamma_0)$$

with the notation of chapter 3. This claim is an immediate consequence of [2, Proposition 3.3.2.3].

Definition 5.3.3 The *Mori chamber decomposition* of the cone of effective divisor classes is the collection of chambers – i.e. full dimensional cones – of the GIT fan of $\overline{M}_{0,n}$. The *Mori chamber decomposition* of the moving cone arises from the chambers of the fan obtained by intersecting the GIT fan with $\mathrm{Mov}(\overline{M}_{0,n})$.

By [2, Remark 3.3.4.2], the inner cones of the Mori chamber decomposition of the moving cone $\mathrm{Mov}(\overline{M}_{0,n})$ parametrise all Mori dream spaces with Cox ring $\mathcal{R}(\overline{M}_{0,n})$ up to isomorphism. Each divisor D such that $[D] \in \mathrm{Eff}(\overline{M}_{0,n})^\circ$ defines a birational map

$$\varphi(D) : \overline{M}_{0,n} \dashrightarrow \mathrm{Proj}(\Gamma(\overline{M}_{0,n}, \mathcal{S}^+(D))), \quad \mathcal{S}^+(D) := \bigoplus_{n \in \mathbb{Z}_{\geq 0}} \mathcal{O}_{\overline{M}_{0,n}}(nD).$$

It is a *small quasimodification*, that is a rational map defining an isomorphism between open subsets with complement of codimension at least 2, iff $[D] \in \mathrm{Mov}(\overline{M}_{0,n})^\circ$.

The implementation described in chapter 4 allows us to compute the Mori chamber decompositions for $\mathrm{Mov}(\overline{M}_{0,n})$ efficiently. The input data for the algorithm is derived as in 5.2.9. The explicit descriptions may be found in [10].

In order to speed up the computation of the Mori chamber decomposition of $\mathrm{Mov}(\overline{M}_{0,6})$, we dropped the generators of $\overline{M}_{0,6}$ corresponding to Keel-Vermeire divisors. Due to [10, Remark 6.7], the Mori chamber decomposition of the moving cone is not altered by this modification. However, we reduce the problem size significantly, considering an embedding into $\mathbb{K}^{25}$ instead of $\mathbb{K}^{40}$.

6 Conclusion

In this thesis, we presented a massively parallel approach for computing the GIT fan of a torus action on an affine variety. After explaining the mathematical background and the algorithmic idea of Boehm, Keicher, and Ren, our realisation of the algorithm has been discussed in detail. GPI-Space is the key component when it comes to developing an application that executes the GIT fan algorithm on large scale systems such as the cluster system of the Fraunhofer ITWM. Concurrent parts have been reformulated in terms of Petri nets, whereas the sequential core operations are carried out by the computer algebra system Singular.

We have shown that the computation of the GIT fan translates to a graph traversal problem which is easily parallelised by expanding frontier nodes simultaneously. However, it is crucial that every compute node has a global view of the frontier set. Otherwise, nodes may be expanded several times, wasting potential computation time on the cluster system. For this reason, we provided several storage implementations that manage a list of expanded nodes and frontier nodes. In our use case – the computation of the Mori chamber decomposition of $\mathrm{Mov}(\overline{M}_{0,6})$ – an in-memory solution on a single compute node, which is accessible via an RPC-server, prevailed.

Our implementation scales almost linearly with the amount of available cores, obtaining a utilisation of still 80% when using 640 cores. We were able to determine the Mori chamber representatives in approximately 21 minutes from a precomputed set of representatives for the faces of the simplex γ under the $\mathcal{S}_6$-action. Previous implementations executed on a single machine with 16 cores terminated after a whole day. [10, Rermark 6.8]

In order to determine GIT fans with feasible effort, Boehm, Keicher, and Ren introduced a fast monomial containment test and exploited symmetries occurring in the problem formulation. We contributed to this goal by optimising existing code and rewriting it for scalable hardware systems. However, some optimisations are still left open and may be revisited in future work. For one thing, we still rely on a precomputation by GAP that determines all representatives of the faces of γ. Although we provide a routine for this task, it does not utilise several cores and is too time-consuming when applied to the $\overline{M}_{0,6}$ example. Parallel implementations

would be of high interest, increasing the scalability of our implementation further. Additionally, our storage solution fails if the memory requirements – given by the size of the GIT fan – exceed the available RAM of a single compute node. The development of a distributed hash table with load balancing features would make the RAM of all compute nodes available such that scaling memory requirements may be satisfied without falling back to significant slower hard drives. This becomes particularly interesting in scenarios where cones cannot be encoded by short bit vectors of fixed length. Storing generating rays or defining half spaces requires an uncontrollable amount of memory due to exact representations of numbers in computer algebra systems.

A Lemmas in the field of convex geometry

Lemma A.1 *Let τ and σ be convex cones with $\tau \subseteq \sigma$. Then there exists a unique face ϑ of σ such that $\tau^\circ \subseteq \vartheta^\circ$. If γ is a face of σ, the condition $\tau^\circ \cap \gamma^\circ \neq \emptyset$ holds if and only if $\gamma = \vartheta$.*

Furthermore, $\tau \subseteq \vartheta$ holds.

Proof: Assume that there exist two distinct faces γ_1, γ_2 of σ defined by linear functionals λ_1, λ_2 such that $\tau^\circ \cap \gamma_1^\circ \neq \emptyset \neq \tau^\circ \cap \gamma_2^\circ$. Let $x_i \in \tau^\circ \cap \gamma_i^\circ$. Because of x_1, x_2 being located in the relative interior of τ, we can choose a small $\varepsilon > 0$ such that for all $s \in [-\varepsilon, 1+\varepsilon]$ the point $sx_1 + (1-s)x_2$ is contained in τ. Then the affine linear map $s \mapsto \langle \lambda_2, sx_1 + (1-s)x_2 \rangle$ maps 0 to 0 and 1 to a value greater than 0. However, it would map $-\varepsilon$ to a value smaller than 0, contradicting $-\varepsilon x_1 + (1+\varepsilon)x_2 \in \tau \subseteq \sigma$.

We conclude that there exists only one face ϑ such that $\tau^\circ \cap \vartheta^\circ \neq \emptyset$. Since the set of relative interiors of faces of σ partition σ, we obtain $\tau^\circ \subseteq \vartheta^\circ$.

The remaining claim $\tau \subseteq \vartheta$ follows by taking closures. □

Lemma A.2 *Let $\Sigma \subseteq \mathbb{R}^k$ be a k-dimensional quasifan with convex support. Then every cone in Σ is a face of a cone in $\Sigma^{(k)}$. In particular, Σ is uniquely determined by $\Sigma^{(k)}$.*

Proof: Let $\tau \in \Sigma$ and choose a point $p \in \tau^\circ$. We select hyperplanes H_θ for all $\theta \in \Sigma \setminus \Sigma^{(k)}$ such that $\theta \subseteq H_\theta$. Then we have

$$S := |\Sigma| \setminus \left(\bigcup_{\theta \in \Sigma \setminus \Sigma^{(k)}} H_\theta \right) \subseteq \bigcup_{\sigma \in \Sigma^{(k)}} \sigma =: |\Sigma^{(k)}|.$$

Since $|\Sigma|$ has dimension k and S arises from $|\Sigma|$ by removing only finitely many hyperplanes, we find $q \in S \setminus \{p\}$. By Bézout's theorem, every hyperplane H_θ intersects the line through p and q in at most one point. As $|\Sigma|$ is convex, the connecting line $\overline{pq}$ is contained in $|\Sigma|$ and $\overline{pq} \cap S$ emerges from $\overline{pq}$ by removing a finite amount of points. Hence, we find a sequence $(p_n)_{n \in \mathbb{N}} \subseteq \overline{pq} \cap S$ such that

$\lim_{n\to\infty} p_n = p$. It follows that $p \in \overline{S}$. Since $|\Sigma^{(k)}|$ is closed as a finite union of closed sets, $p \in |\Sigma^{(k)}|$ holds. Let $\sigma \in \Sigma^{(k)}$ such that $p \in \sigma$. Then the common face of σ and τ contains p and thus has to be τ. We conclude that $\tau \preceq \sigma$.

Now, Σ is uniquely determined by $\Sigma^{(k)}$ since quasifans are closed under taking faces. □

B Ideal quotients

Theory about saturated ideals allows us to implement an efficient monomial containment test, see [10]. Here, we establish some elementary properties that are used in this thesis.

Definition B.1 Let I, J be ideals over a commutative ring R. Then we define the *ideal quotient* $(I : J)$ by

$$(I : J) = \{r \in R \mid rJ \subseteq I\}.$$

The *saturation* $(I : J^\infty)$ is given by

$$(I : J^\infty) = \bigcup_{n \in \mathbb{N}} (I : J^n).$$

Proposition B.2 *Let I, J, K be ideals over R and let $f, g \in R$. Then we have:*

(i) $I \subseteq J \;\Rightarrow\; (I : K^\infty) \subseteq (J : K^\infty)$,

(ii) $((I : (f)^\infty) : (g)^\infty) = (I : (fg)^\infty)$.

Proof: (i) immediately follows from the definition. For (ii), note that

$$(I : (f)) = \{r \in R \mid rf \in I\} \quad \text{and} \quad (I : (f)^\infty) = \{r \in R \mid \exists n \in \mathbb{N} : rf^n \in I\}.$$

Let $r \in (I : (fg)^\infty)$. We find $n \in \mathbb{N}$ such that $(rg^n)f^n = r(fg)^n \in I$. Hence, $rg^n \in (I : (f)^\infty)$. It follows that $r \in ((I : (f)^\infty) : (g)^\infty)$.

Conversely, let $r \in ((I : (f)^\infty) : (g)^\infty)$. Then we find $n \in \mathbb{N}$ such that $rg^n \in (I : (f)^\infty)$. This implies that there exists an $m \in \mathbb{N}$ such that $rg^n f^m \in I$. For this reason $r(fg)^{\max\{n,m\}} \in I$ and thus $r \in (I : (fg)^\infty)$ holds, proving (ii). □

Bibliography

[1] W. M. P. van der Aalst. 'Three Good Reasons for Using a Petri-Net-Based Workflow Management System'. In: *Information and Process Integration in Enterprises: Rethinking Documents*. Ed. by Toshiro Wakayama, Srikanth Kannapan, Chan Meng Khoong, Shamkant Navathe, and JoAnne Yates. Boston, MA: Springer US, 1998, pp. 161–182. ISBN: 978-1-4615-5499-8. DOI: 10.1007/978-1-4615-5499-8_10.

[2] Ivan Arzhantsev, Ulrich Derenthal, Jürgen Hausen, and Antonio Laface. *Cox rings*. Vol. 144. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2014.

[3] Ivan V. Arzhantsev and Jürgen Hausen. 'Geometric invariant theory via Cox rings'. In: *Journal of Pure and Applied Algebra* 213.1 (2009), pp. 154–172. ISSN: 0022-4049. DOI: 10.1016/j.jpaa.2008.06.005.

[4] Victor V. Batyrev and Oleg N. Popov. 'The Cox Ring of a Del Pezzo Surface'. In: *Arithmetic of Higher-Dimensional Algebraic Varieties*. Ed. by Bjorn Poonen and Yuri Tschinkel. Boston, MA: Birkhäuser Boston, 2004, pp. 85–103. ISBN: 978-0-8176-8170-8. DOI: 10.1007/978-0-8176-8170-8_5.

[5] BeeGFS. URL: https://www.beegfs.io/ (visited on 12/17/2017).

[6] Florian Berchtold and Jürgen Hausen. 'GIT Equivalence beyond the Ample Cone'. In: *Michigan Math. J.* 54.3 (Nov. 2006), pp. 483–516. DOI: 10.1307/mmj/1163789912.

[7] Martha María Bernal Guillén. 'Relations in the Cox Ring of $\overline{M}_{0,6}$'. PhD thesis. Warwick: University of Warwick, 2012.

[8] Janko Böhm, Wolfram Decker, Anne Frühbis-Krüger, Franz-Josef Pfreundt, Mirko Rahn, and Lukas Ristau. 'Determining smoothness of algebraic varieties using a massively parallel approach'. In preparation.

[9] Janko Böhm, Wolfram Decker, Simon Keicher, and You Ren. *gitfan.lib – A Singular library for computing the GIT fan*. 2016. URL: https://github.com/Singular/Sources (visited on 12/17/2017).

[10] J. Boehm, S. Keicher, and Y. Ren. 'Computing GIT-fans with symmetry and the Mori chamber decomposition of $\overline{M}_{0,6}$'. In: *ArXiv e-prints* (Mar. 2016). arXiv: 1603.09241v2 [math.AG].

[11] T. Bogart, A.N. Jensen, D. Speyer, B. Sturmfels, and R.R. Thomas. 'Computing tropical varieties'. In: *Journal of Symbolic Computation* 42.1 (2007). Effective Methods in Algebraic Geometry (MEGA 2005), pp. 54–73. ISSN: 0747-7171. DOI: 10.1016/j.jsc.2006.02.004.

[12] Ana-Maria Castravet. 'The Cox ring of $\overline{M}_{0,6}$'. In: *Transactions of the American Mathematical Society* 361 (2009), pp. 3851–3878. DOI: 10.1090/S0002-9947-09-04641-8.

[13] David Cox, John Little, and Henry Schenck. *Toric Varieties*. Vol. 124. Graduate studies in Mathematics. American Mathematical Society, 2011. ISBN: 978-0-8218-4819-7.

[14] Pierre Deligne and David Mumford. 'The irreducibility of the space of curves of a given genus'. In: *Publications Mathématiques de l'Institut des Hautes Études Scientifiques* 36 (1969), pp. 75–109. ISSN: 1618-1913. DOI: 10.1007/BF02684599.

[15] Igor Dolgachev. *Lectures on Invariant Theory*. Vol. 296. London Mathematical Society Lecture Note Series. Cambridge University Press, 2003. DOI: 10.1017/CBO9780511615436.

[16] GAP. URL: http://www.gap-system.org/ (visited on 12/17/2017).

[17] David Gelernter and Nicholas Carriero. 'Coordination Languages and Their Significance'. In: *Commun. ACM* 35.2 (Feb. 1992), pp. 96–107. ISSN: 0001-0782. DOI: 10.1145/129630.376083.

[18] GPI-Space. URL: http://www.gpi-space.de/ (visited on 12/17/2017).

[19] Google Test – *A C++ test framework*. URL: https://github.com/google/googletest (visited on 12/17/2017).

[20] Robin Hartshorne. *Algebraic Geometry*. Vol. 52. Graduate Texts in Mathematics. New York: Springer, 1977. DOI: 10.1007/978-1-4757-3849-0.

[21] Jürgen Hausen, Simon Keicher, and Antonio Laface. 'On blowing up the weighted projective plane'. In: *ArXiv e-prints* (Aug. 2016). arXiv: 1608.04542 [math.AG].

[22] James E. Humphreys. *Linear Algebraic Groups*. Vol. 21. Graduate Texts in Mathematics. New York: Springer, 1975. DOI: 10.1007/978-1-4684-9443-3.

[23] Kurt Jensen. 'Coloured Petri nets'. In: *Petri Nets: Central Models and Their Properties*. Ed. by W. Brauer, W. Reisig, and G. Rozenberg. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1987, pp. 248–299. ISBN: 978-3-540-47919-2. DOI: 10.1007/BFb0046842.

[24] Simon Keicher. 'Computing the GIT-fan'. In: *International Journal of Algebra and Computation* 22.07 (2012). DOI: 10.1142/S0218196712500646.

[25] David Mumford, John Fogarty, and Frances Kirwan. *Geometric Invariant Theory*. 3rd ed. Vol. 34. Ergebnisse der Mathematik und ihrer Grenzgebiete. 2. Folge. Springer-Verlag Berlin Heidelberg, 1994.

[26] Carl Adam Petri. 'Kommunikation mit Automaten'. PhD thesis. Institut für Angewandte Mathematik der Universität Bonn, 1962.

[27] Maxwell Rosenlicht. 'Some rationality questions on algebraic groups'. In: *Annali di Matematica Pura ed Applicata* 43.1 (1957), pp. 25–50. ISSN: 1618-1891. DOI: 10.1007/BF02411903.

[28] SINGULAR. Computer algebra system. URL: https://www.singular.uni-kl.de/ (visited on 12/17/2017).

[29] SLURM. URL: https://slurm.schedmd.com/ (visited on 12/17/2017).

[30] 'Stable n-pointed Curves'. In: *An Invitation to Quantum Cohomology: Kontsevich's Formula for Rational Plane Curves*. Boston, MA: Birkhäuser Boston, 2007, pp. 21–45. ISBN: 978-0-8176-4495-6. DOI: 10.1007/978-0-8176-4495-6_3.

[31] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 'Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications'. In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 149–160. ISSN: 0146-4833. DOI: 10.1145/964723.383071.

[32] Bernd Sturmfels. *Gröbner Bases and Convex Polytopes*. Vol. 8. University Lecture Series. American Mathematical Society, 1996. DOI: 10.1090/ulect/008.

[33] GPI. URL: http://www.gpi-site.com/gpi2/ (visited on 12/17/2017).

List of abbreviations

BLOB	Binary Large Object 42
DRTS	distributed run-time engine 35
Fraunhofer ITWM	*Fraunhofer-Institut für Techno- und Wirtschaftsmathematik* . 35
GIT	Geometric Invariant Theory 1
PGAS	partitioned global address space 36
RIFD	Remote Interface Daemon 53
ssi	simple singular interface 42
STL	Standard Template Library 42
WE	workflow engine 35

Index

Erklärung zu dieser Arbeit

Hiermit erkläre ich, die vorliegende Masterarbeit selbstständig erstellt zu haben. Weiterhin versichere ich, dass sämtliche von mir verwendeten Hilfsmittel und Quellen im Literaturverzeichnis aufgeführt sind.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Christian Reinbold, Hannover, den 11.01.2018