Torische Geometrie mit `polymake`

M. Joswig, A. Paffenholz
(TU Darmstadt, Fachbereich Mathematik,
Dolivostr. 15, 64293 Darmstadt, Germany)

joswig@mathematik.tu-darmstadt.de
paffenholz@mathematik.tu-darmstadt.de

Einführung

`polymake` ist Software für ein weites Spektrum von Anwendungen in kombinatorischer Geometrie und benachbarten Gebieten. Der Schwerpunkt liegt auf der Polyedertheorie. Andere Aspekte betreffen Graphen und Matroide, lineare und kombinatorische Optimierung, kombinatorische und algebraische Topologie sowie torische und tropische Geometrie. Zielsetzung und Organisation folgen grundsätzlich immer noch den in [9] und [12] dargelegten Prinzipien, jedoch ist die Funktionalität seitdem in technischer und mathematischer Hinsicht erheblich erweitert worden. Der Zweck dieses Textes ist es, einen Einblick in die aktuelle Entwicklung zu geben. Diese profitiert maßgeblich von der Kooperation innerhalb des DFG-Schwerpunktprogramms 1489 „Algorithmic and Experimental Methods in Algebra, Geometry and Number Theory“; vgl. [8].

`polymake` ist quelloffen und wird unter der GNU Public License auf der Webseite

`www.polymake.org`

vertrieben. Das letzte `polymake`-Release 2.12 wurde im März 2012 veröffentlicht. Es kann auf Unix/Linux- und Mac OS-basierten Systemen kompiliert werden. Um Interessierten den Zugang zur aktuellen Entwicklung zu ermöglichen, kann seit Januar 2013 ein direkter Abzug aus unserem Subversion-Repository bezogen werden.

Interfaces

Wie viele Computeralgebra-Systeme besitzt auch `polymake` ein interaktives Shell-basiertes Interface. Als Eingabesprache kommt ein Perl-Dialekt zum Einsatz. Die Algorithmen sind überwiegend in Perl und C++ implementiert. So lassen sich die Vorteile einer interpretierten Sprache mit denen einer kompilierten kombinieren. Für die Visualisierung wird in erster Linie `jreality` [13] über dessen Java-API verwendet; siehe hierzu [10].

Zusätzlich zu zahlreichen direkt in `polymake` implementierten Verfahren spielen Interfaces zu anderen Systemen eine wichtige Rolle. In jüngster Zeit hinzugekommen sind ein bidirektionales Interface zu `Singular` [4], ein Interface von `GAP` [7], das ebenfalls bidirektional ausgebaut werden soll, sowie Interfaces von `polymake` zu `normaliz` [2], `bliss` [14], `Gfan` [11], `permlib` und `sympol` [20]. Seit der Version 2.12 existiert `polymake` auch in einer Version als C++-Bibliothek, das zugehörige API wird von `GAP` und `Singular` verwendet.

Regeln und Erweiterungen

Grundsätzlich erfolgt die Kommunikation zwischen Benutzer und System regelbasiert. Typische Objekte wie etwa ein konvexes Polytop, sind auf der höchsten Abstraktionsebene innerhalb `polymake` als Listen von *Eigenschaften (properties)* codiert. Zu jedem Zeitpunkt sind einige bekannt, andere noch nicht, und eine Menge von *Regeln (rules)* legt fest, aus welchen Kombinationen bekannter Eigenschaften neue ausgerechnet werden können. Der zentrale *Scheduler* legt fest, in welcher Reihenfolge Regeln ausgeführt werden, um eine Benutzeranfrage zu befriedigen. Der Benutzer kann das Verhalten des Schedulers durch verschiedene Mechanismen beeinflussen und die gesamte Regelbasis modifizieren.

Die Regelbasis ist auf zweierlei Weise modularisiert. Einerseits sind Objekte und zugehörige Regeln in *Anwendungen (applications)* organisiert, die sich im weitesten Sinn ähnlich wie *name spaces* verhalten. Die wichtigsten Standardanwendungen sind derzeit `polytope` (für konvexe Polyeder), `fan` (für polyedrische Fächer und Komplexe) und `topaz` (für Topologie). Andererseits existiert ein hierzu orthogonales Konzept von *Erweiterungen (extensions)*. Eine Erweiterung kann zu einer oder mehreren bestehenden Anwendungen neue Objekte oder neue Regeln hinzufügen. Alternativ kann eine Erweiterung auch eine vollständige neue

Anwendung begründen mit gänzlich eigenen Objekten und Regeln.

Die aktuelle SVN-Version macht bereits in der zentralen Codebasis vom Erweiterungskonzept Gebrauch. Für das Rechnen mit Graphenautomorphismen stehen beispielsweise Interfaces zu `nauty` [17] und `bliss` bereit, die sich alternativ konfigurieren lassen. Das Interface von `polymake` zu `Singular` ist ebenfalls in eine Erweiterung ausgelagert, die nicht zwingend konfiguriert werden muss, so dass es möglich bleibt, `polymake` auch ohne `Singular` zu nutzen; allerdings fehlt dann ein Teil der Funktionalität. Erweiterungen innerhalb der `polymake`-Distribution werden *bundled extensions* genannt.

Beispiel: Hirzebruchflächen

Wir wollen die Benutzung von `polymake` an zwei einfachen Beispielen aus der algebraischen Geometrie demonstrieren. Eine *projektive torische Varietät* X korrespondiert zu einem *vollständigen rationalen polyedrischen Fächer* Σ_X, also einer Familie rationaler polyedrischer Kegel, die sich paarweise in Seiten schneiden und den gesamten Raum überdecken; für einen Überblick siehe [3]. Besonders wichtig ist die folgende Konstruktion. Es sei P ein *Gitterpolytop*, das heißt, die konvexe Hülle endlich vieler ganzzahliger Punkte in $\mathbb{R}^d$. Für jede Seite F von P ist der zugehörige *Normalenkegel* die Menge derjenigen linearen Funktionale, die auf F ihren Maximalwert annehmen. Die Menge aller Normalenkegel bildet den *Normalenfächer* und hierdurch eine projektive torische Varietät. In `polymake` sind die relevanten Methoden auf die beiden Anwendungen `polytope` und `fan` verteilt.

In unserem ersten Beispiel wollen wir *Hirzebruch-Flächen* betrachten. Die Hirzebruchfläche $\mathcal{H}_a$ für ein $a \in \mathbb{Z}_{\geq 0}$ ist eine 2-dimensionale projektive torische Varietät. Der Fläche $\mathcal{H}_a$ ist der Fächer Σ_a mit den durch $(-1,0), (0,1), (1,-a)$, und $(0,-1)$ erzeugten Strahlen zugeordnet. Bestimmte Gitterpolytope H_a mit diesem Normalenfächer erzeugen eine Einbettung von $\mathcal{H}_a$ in den projektiven Raum $\mathbb{CP}^{n-1}$, wobei $n = \#(H_a \cap \mathbb{Z}^2)$ die Anzahl der Gitterpunkte des Polytops ist. Ein geeignetes Gitterpolytop ist in Abbildung 1 links dargestellt. Das *Stanley-Reisner-Ideal* einer regulären und unimodularen Triangulierung ergibt ein Erzeugendensystem des Initialideals des definierenden Ideals der Varietät [21].

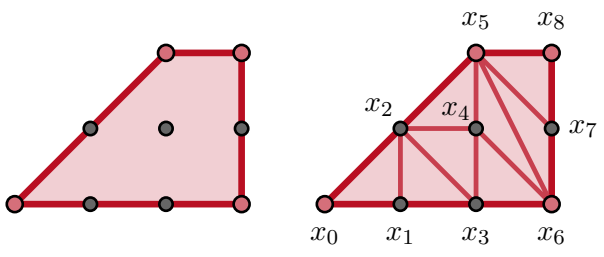

Abbildung 1: *Links: Gitterpolytop H, dessen Normalenfächer die Hirzebruchfläche $\mathcal{H}_1$ liefert. Rechts: unimodulare Triangulierung.*

Wir wollen dieses Ideal mit `polymake` bestimmen. Nach dem Aufruf in einem Terminal erhalten wir den `polymake`-prompt zur Anwendung `polytope`, die als Standard eingestellt ist.

```
polytope> $V=new Matrix<Rational>(
  [[1,0,1],[1,1,1],[1,1,-1],[1,-2,-1]]);
polytope> $H=new Polytope<Rational>(POINTS=>$V);
polytope> print $H->FACETS
1 1 -1
1 0 1
1 -1 0
1 0 -1
```

Wir haben das Polytop `$H` als konvexe Hülle der vier Punkte

$$(0,1) \qquad (1,1) \qquad (1,-1) \qquad (-2,-1)$$

definiert und uns die irredundante Ungleichungsbeschreibung ausgeben lassen. In `polymake` wird ein Polytop P durch seine Homogenisierung $\{1\} \times P$ dargestellt, daher wird bei der Eingabe allen Punkten eine 1 vorangestellt. Listen von Vektoren werden zeilenweise ein- und ausgegeben. `polymake` unterscheidet zwei Darstellungen von Punktmengen. Die Eingabe `POINTS` darf redundant sein; dagegen listet die Eigenschaft `VERTICES` nur die Ecken des Polytops. Ein Objekt vom Typ `Polytope` darf auch ein unbeschränktes Polyeder sein; die Eigenschaft `LINEALITY_SPACE` gibt eine Basis des Linealitätsraums an (die redundante Form zur Eingabe heißt `INPUT_LINEALITY`). Das Paar aus `INEQUALITIES` und `EQUATIONS` erlaubt die redundante Beschreibung eines Polytops durch Ungleichungen. Die nicht-redundanten Entsprechungen sind `FACETS` bzw. `AFFINE_HULL`. `polymake` kennt verschiedene Algorithmen um von `POINTS` zu `FACETS` zu gelangen. Welcher zum Einsatz kommt, hängt von der lokalen Konfiguration der Regelbasis ab.

Fast alle Datentypen, wie oben `Matrix` und `Polytope`, können von anderen Datentypen abhängen. In unserem Fall haben wir den Typ der Koordinaten auf `Rational` festgelegt, was rationalen Zahlen mit exakter Arithmetik entspricht. Es sind auch komplexere Varianten möglich, z. B. ergibt `new Array<Vector<Integer>>(5)` einen Array von fünf ganzzahligen Vektoren. Die Notation bildet das *template*-Konzept der `C++`-Klassen ab, das den Datentypen zugrunde liegt.

Bei der Erzeugung eines Objekts z. B. vom Typ `Polytope` können beliebig viele Eigenschaften in der Form `property=>value` angegeben werden. Dabei liegt es in der Verantwortung des Benutzers, dass die Eingabe gültig ist, `polymake` überprüft dies bewusst nicht — aus Gründen der Effizienz bzw. wegen Grenzen der Entscheidbarkeit. Nach der Definition des Polytops können Eigenschaften berechnet werden, die sich mit Hilfe der Regelbasis ableiten lassen. Neben der direkten Konstruktion durch Eigenschaften kennt `polymake` viele Standardkonstruktionen. Beispiele sind `cube(d)` für einen d-dimensionalen Würfel oder `product($P,$Q)` für das Produkt zweier Polytope `$P` und `$Q`.

Nun testen wir, ob unser Polytop eine Einbettung ergibt, bestimmen eine reguläre Triangulierung und speichern sie in einem neuen Objekt vom Typ `SimplicialComplex` aus der Anwendung `topaz`.

```
polytope> $T=new topaz::SimplicialComplex(FACETS=>
  placing_triangulation($H->LATTICE_POINTS));
polytope> print $T->FACETS;
{0 1 2}
{1 2 3}
{2 3 4}
{2 4 5}
{3 4 6}
{4 5 6}
{5 6 7}
{5 7 8}
polytope> $I=ideal::stanley_reisner($T);
polytope> print $I->GENERATORS;
x0*x3 x0*x4 x0*x5 x0*x6 x0*x7 x0*x8 x1*x4
x1*x5 x1*x6 x1*x7 x1*x8 x2*x6 x2*x7 x2*x8
x3*x5 x3*x7 x3*x8 x4*x7 x4*x8 x6*x8
```

Die Triangulierung ist in Abbildung 1 rechts dargestellt. Nach der Definition haben wir uns die Liste der maximalen Simplizes der Triangulierung ausgeben lassen. Die Numerierung (ab 0) der Ecken der Triangulierung entspricht dabei der Reihenfolge in der Liste aller Gitterpunkte `$H->LATTICE_POINTS`. Anschließend haben wir das *Stanley-Reisner-Ideal* der Triangulierung bestimmt und uns die Erzeuger anzeigen lassen. Da unser Polytop `$H` eben ist, ist die Triangulierung unimodular. Im Stanley-Reisner-Ideal entspricht jeder Gitterpunkt einer Variablen, und die Monome entsprechen minimalen Teilmengen von Punkten, die keine Seite der Triangulierung bilden. Das zugehörige torische Ideal und Gröbnerbasen können mit der Erweiterung `polymake_algebra` weiter untersucht werden [16].

Beispiel: Polyedrische Adjunktion

In diesem Abschnitt wollen wir die Erweiterung `PolyhedralAdjunction` vorstellen [18]. Jedes *rationale* d-dimensionale Polytop P lässt sich in der Form

$$P := \{x \in \mathbb{R}^d \mid \langle a_i, x \rangle \leq b_i \text{ für } 1 \leq i \leq n\}$$

beschreiben, wobei $a_i \in (\mathbb{Z}^d)^*$ ein primitiver ganzzahliger Vektor ist, und keine der n Ungleichungen weggelassen werden kann, ohne das Polytop zu verändern (das System ist *irredundant*). Das zu P mit Parameter $c > 0$ *adjungierte* Polytop ist dann

$$P^{(c)} := \{x \in \mathbb{R}^d \mid \langle a_i, x \rangle \leq b_i - c \text{ für } 1 \leq i \leq n\},$$

siehe auch [5]. Vor der ersten Verwendung muss eine Erweiterung mit `import_extension` initialisiert werden,

```
polytope> import_extension("/path/to/extension");
```

wobei `/path/to/extension` durch den korrekten Pfad ersetzt werden sollte.

```
polytope> $P=new Polytope(INEQUALITIES=>
  [[4,-1,1],[0,1,0],[0,0,1],[3,0,-1],[5,-1,0]]);
polytope> $adj_P1=adjoint_polytope($P,1);
polytope> print $adj_P1->VERTICES;
1 1 1
1 1 2
1 4 2
1 4 1
```

Hier haben wir ein durch fünf Ungleichungen bestimmtes Polytop definiert und in der Variablen `$P` gespeichert. Dabei werden Ungleichungen in `polymake` in der Form $0 \leq b + \langle a, x \rangle$ interpretiert, der Vektor $(4, -1, 1)$ entspricht also der Ungleichung $x_1 - x_2 \leq 4$.

Anschließend haben wir via `adjoint_polytope` aus der zuvor geladenen Erweiterung das Polytop $P^{(1)}$ berechnet und uns die Ecken ausgeben lassen. Das Polytop und sein Adjungiertes für $c = 1$ sind in folgender Abbildung dargestellt.

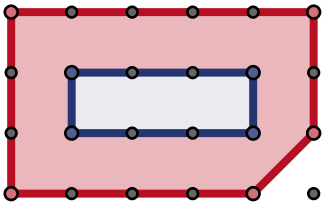

Abbildung 2: *Ein Gitterpolygon und sein Adjungiertes für* $c = 1$

Wie zu sehen ist, kann das Polytop $P^{(c)}$ weniger Facetten haben als P; und für ausreichend großes c ist es leer. Das führt zu zwei interessanten Invarianten. Der *nef-Wert* von P ist das Inverse des kleinsten c, für das die Normalenfächer von P und $P^{(c)}$ nicht mehr übereinstimmen und der $\mathbb{Q}$-*Kograd* ist das Inverse des größten c, für das $P^{(c)}$ nicht leer ist.

```
polytope> print $P->NEF_VALUE;
1
polytope> print $P->Q_CODEGREE;
3/2
```

Die Motivation für das Studium dieser Invarianten kommt aus der algebraischen Geometrie. Wie bereits oben bei den Hirzebruchflächen definiert der Normalenfächer eines Gitterpolytops P eine projektive torische Varietät X. Zusätzlich legt die konkrete Wahl des Polytops auch noch ein Geradenbündel auf X fest. Die adjungierten Polytope gehören dann zu den adjungierten Geradenbündeln $K_X + cL$, siehe [1, 5]. Diese Korrespondenz zwischen algebraischer und kombinatorischer Geometrie ermöglicht es, algebraische Fragen mit kombinatorischen Methoden zu untersuchen. Auf diesem Weg konnte kürzlich z. B. die Spektralvermutung von Fujita im torischen Fall bewiesen werden [19].

Wir wollen den Normalenfächer unseres Polytops P weiter untersuchen. Die meisten Methoden liegen in der Anwendung `fan`. Daher wechseln wir zunächst dorthin.

```
polytope>application "fan";
fan> $nf=normal_fan($P);
fan> print $nf->Q_GORENSTEIN;
1
fan> print $nf->Q_GORENSTEIN_INDEX;
1
```

Die Variable $nf enthält nun den Normalenfächer unseres Polygons. Mit der nachfolgenden Abfrage haben wir polymake ausrechnen lassen, ob unsere Varietät die $\mathbb{Q}$-*Gorenstein-Eigenschaft* besitzt, ob es also eine positive ganze Zahl r gibt, so dass das r-fache des kanonischen Divisors K_X ein Cartier-Divisor ist. Die Ausgabe ist ein boolescher Wert, 1 steht hier für *wahr* (bei *falsch* bliebe die Ausgabe leer). Wir können uns mit Q_GORENSTEIN_INDEX die Zahl r berechnen lassen. Unsere Varietät ist sogar nichtsingulär, wie wir mit

```
fan> print $nf->SMOOTH_FAN;
1
```

verifizieren können. Der Fächer wird durch seine Strahlen und deren Inzidenzrelation vollständig bestimmt. Hier zeigen wir, wie man die implizite Numerierung der Strahlen sichtbar machen kann.

```
fan> print rows_numbered($nf->RAYS);
0: -1  1
1:  1  0
2:  0  1
3:  0 -1
4: -1  0
fan > print $nf->MAXIMAL_CONES;
{1 2}
{0 2}
{0 4}
{3 4}
{1 3}
```

MAXIMAL_CONES listet die inklusionsmaximalen Kegel, wobei für jeden Kegel der Index der Strahlen in der Liste RAYS angegeben wird, der ein Strahl des Kegels ist. Für weitere Eigenschaften, insbesondere für Rechnungen mit Divisoren auf der torischen Varietät steht die Erweiterung polymake_toric zur Verfügung [15].

Die hier vorgestellten Methoden bilden nur einen sehr kleinen Teil der Funktionalität von polymake ab. In den Tutorials und der Dokumentation auf polymake.org finden sich viele weitere. Dort findet sich auch eine Liste mit wissenschaftlichen Arbeiten, die polymake verwendet haben. Weitere Projekte, die polymake benutzen, stehen auch auf der Seite computeralgebra.de. Direkte Hilfe in der Shell gibt es mit help "<item>" zu fast allen Funktionen und Eigenschaften, z. B.

```
polytope> help "CANONICAL";
objects/LatticePolytope/properties/CANONICAL:
 The polytope is canonical if there is
 exactly one interior lattice point.
```

Für weitergehende Fragen verweisen wir auch auf unser Forum unter forum.polymake.org.

Literatur

[1] M.C. Beltrametti und A.J. Sommese, The adjunction theory of complex projective varieties, Band 16 in *Expositions in Mathematics*, Walter de Gruyter & Co., Berlin, 1995.

[2] W. Bruns, B. Ichim und C. Söger, Normaliz. Algorithms for rational cones and affine monoids, www.math.uos.de/normaliz.

[3] D.A. Cox, J.B. Little und H.K. Schenck, Toric varieties, *Graduate Studies in Mathematics*, Band 124, American Mathematical Society, Providence, RI, 2011.

[4] W. Decker, G.-M. Greuel, G. Pfister und H. Schönemann, SINGULAR 3-1-6 — A computer algebra system for polynomial computations, 2012, www.singular.uni-kl.de.

[5] S. DiRocco, C. Haase, B. Nill und A. Paffenholz, Polyhedral Adjunction Theory, preprint, Mai 2011, arxiv:1105.2415.

[6] T. Fujita, On Kodaira energy and adjoint reduction of polarized manifolds. *Manuscr. Math.*, 76:59–84, 1992.

[7] The GAP Group, GAP – Groups, Algorithms, and Programming, Version 4.5.7, 2012, www.gap-system.org.

[8] github.com/Singular/Sources/wiki/Gap-Polymake-Singular-Applications

[9] E. Gawrilow und M. Joswig, polymake: a framework for analyzing convex polytopes, *Polytopes—combinatorics and computation* (Oberwolfach, 1997), Birkhäuser, Basel, 2000, pp. 43–73.

[10] E. Gawrilow, M. Joswig, T. Rörig und N. Witte, Drawing polytopal graphs in polymake, *Comput. Vis. Sci.*, 2010:13, 99–110.

[11] A.N. Jensen, Gfan, a software system for Gröbner fans and tropical varieties, home.imf.au.dk/jensen/software/gfan/gfan.html.

[12] M. Joswig, polymake, *Computeralgebra-Rundbrief* 2003:33, 15–16.

[13] The jReality team, jReality, www3.math.tu-berlin.de/jreality/

[14] T. Junttila und P. Kaski, bliss: A Tool for Computing Automorphism Groups and Canonical Labelings of Graphs, www.tcs.hut.fi/Software/bliss/, 2012.

[15] L. Kastner, B. Lorenz, A. Paffenholz und A. Winz, polymake_toric (eine polymake Erweiterung), github.com/lkastner/polymake_toric.

[16] L. Kastner, B. Lorenz und A. Winz, polymake_algebra (eine polymake Erweiterung), github.com/lkastner/polymake_algebra.

[17] B. D. McKay, Practical graph isomorphism, *Congressus Numerantium* 1981:30, 45–87.

[18] A. Paffenholz, PolyhedralAdjunction (eine polymake Erweiterung), github.com/apaffenholz/polymake_polyhedral_adjunction.

[19] A. Paffenholz, Finiteness of the Polyhedral $\mathbb{Q}$-Codegree Spectrum, preprint, Januar 2013, arxiv:1301.4967.

[20] T. Rehn und A. Schürmann, C++ Tools for Exploiting Polyhedral Symmetries, *Lecture Notes in Computer Science*, 2010, Band 6327/2010

[21] B. Sturmfels, Gröbner bases and convex polytopes, *University Lecture Series*, 8. American Mathematical Society, Providence, RI, 1996.