Computation of denominator polynomials for Poincaré series on Monomial rings

Mikael Johansson

August 31, 2004

Abstract

Given a monomial ideal M in a polynomial ring $Q = k[x_1, \ldots, x_r]$ over a field k, the Poincaré-Betti series $\sum_i \dim_k \operatorname{Tor}_i^{Q/M}(k,k)t^i$ is a rational function of the form $(1+t)^r / b_{Q/M,k}(t)$. The denominator polynomial $b_{Q/M,k}(t)$ depends only on the LCM lattice and GCD graph of a generator set of the monomial ideal.

I intend to present an implementation of a simple computer program that will explicitly calculate the denominator polynomial for a given set of generators for the monomial ideal.

The lattices that may occur as LCM lattices for some monomial ideal with k generators are precisely the k-atomic lattices. I will present work in progress on listing all isomorphism classes of atomic lattices with precisely 6 atoms as well as discuss some problems arising in such a listing.

Sammanfattning

Givet ett monomideal M i en polynomring $Q = k[x_1, \ldots, x_r]$ i r variabler över en kropp k av godtycklig karakteristik definieras Poincaré-Betti-serien som den formella potensserien $\sum_i \dim_k \operatorname{Tor}_i^{Q/M}(k,k)t^i$. Den bildar en rationell funktion på formen $(1+t)^r/b_{Q/M,k}(t)$ för något polynom $b_{Q/M,k}(t)$. Nämnarpolynomet $b_{Q/M,k}(t)$ beror endast på MGM-spaljén och SGD-grafen associerade till en generatormängd till monomidealet.

Jag ämnar härvid presentera ett program för beräknandet av nämnarpolynomet givet en generatormängd för monomidealet.

De spaljéer som kan förekomma som MGM-spaljéer är precis de atomära spaljéerna, och jag ämnar presentera en del pågående arbete med att etablera en lista över alla isomorfiklasser av 6-atomära spaljéer och även diskutera de problem som uppstår i genererandet av en sådan lista.

Summarium

Series $\sum_i \dim_k \operatorname{Tor}_i^{Q/M}(k,k)t^i$, quae Poincaré-Betti nuncupatur, multitudine ideali, in monomine anuli ex polynominibus $Q = k[x_1, \ldots, x_r]$ consistentis quod M dicitur, adhibita, ut series potentiae rationalis $(1+t)^r / b_{Q/M,k}(t)$ definitur. Forma polynominis denominatoris $b_{Q/M,k}(t)$ tantummodo ex adminiculo supremo atque signo infimo pendet.

Institutionem computatoris, quae indicem polynominum, numero monominis idealis dato, computare poterit, exhibiturus sum.

Haec, quae ut adminicula suprema exsistere possunt, vere adminicula atomorum sunt. Ideo exhibiturus sum laborem currentem omnium classium, quae ex adminiculis senorum atomorum constent, adhibendarum. Quaestiones etiam, quae ab talibus adhibendis oriuntur, demonstraturus sum.

Acknowledgements

I would like to thank my thesis advisor, Jörgen Backelin, for presenting these and several other very interesting problems to me. I am also indebted to Alexander Berglund, with whom I started working on these themes, for ideas, advice, source code and general solutions to the specific problems I happened to be working on. Some of my more speculative heuristic arguments would have been even more meaningless without the mathematical statistics knowledge of Andreas Nordvall-Lagerås, to whom I also extend thanks.

Hans Aili, doctori meo linguae latinae, pro summarium meum lecto emendatoque, magnas gratias ago.

The diagrams in this thesis were drawn using Paul Taylor's commutative diagram package. [Taylor, 2004] The reference guide in Appendix B was created using the documentation tool Doxygen. [Doxygen, 2004]

Contents

1 Introduction **5**

1.1 Preface and historical overview 5

1.2 Homological algebra 6

1.3 Simplicial Homology and Hilbert series 8

1.4 The Taylor complex 11

1.5 Lattice theory 12

1.6 Previous results 15

2 Implementation details **18**

2.0.1 Complexity issues 19

2.1 Test suite and verifications 20

3 Enumeration of atomic lattices **27**

3.1 Algorithm analysis 29

3.2 Combinatorial explosion of lattices 30

A Users guide **33**

A.1 Obtaining the program 33

A.2 Build and install 33

A.3 Starting the program 33

A.4 Calculation of Poincaré-Betti series 34
A.5 Calculation of simplicial homology 34
A.6 List of commands 35
A.7 Known quirks and problems 38

B Programmers reference **39**

B.0.1 Introduction 39
B.0.2 Installation 39
B.1 Monomial Class Reference 40
B.1.1 Detailed Description 42
B.1.2 Constructor & Destructor Documentation 42
B.1.3 Member Function Documentation 43
B.2 MonomialIdeal Class Reference 46
B.2.1 Detailed Description 47
B.2.2 Member Function Documentation 47
B.3 SimplicialComplex Class Reference 49
B.3.1 Detailed Description 50
B.3.2 Member Function Documentation 50
B.3.3 Member Data Documentation 51
B.4 Squarefree Class Reference 53
B.4.1 Detailed Description 54
B.4.2 Member Data Documentation 54
B.5 homology.c File Reference 55
B.5.1 Detailed Description 56
B.5.2 Function Documentation 56
B.6 homology.h File Reference 58

B.6.1 Detailed Description 59

B.6.2 Function Documentation 60

B.7 main.c File Reference 61

B.7.1 Detailed Description 62

B.7.2 Function Documentation 62

B.8 main.h File Reference 63

B.8.1 Detailed Description 63

B.9 ui.c File Reference 64

B.9.1 Detailed Description 65

B.9.2 Function Documentation 66

B.10 ui.h File Reference 67

B.10.1 Detailed Description 67

B.10.2 Function Documentation 68

B.10.3 Variable Documentation 68

Chapter 1

Introduction

1.1 Preface and historical overview

This report was originally intended to be a joint work with Alexander Berglund [Berglund, 2003]. Circumstances dictated otherwise in the end. Since Alexander has not quit thinking about these matters just because he started his doctorate, several of the relevant results are to a high degree his work.

For a field k, let $Q = k[x_1, \ldots, x_r]$ and $R = Q/\langle M \rangle$ where M is an antichain M of squarefree monomials[1]. The series

$$P_k^R(t) = \sum_0^\infty \dim_k \operatorname{Tor}_i^R(k,k)t^i$$

is called the Poincaré-Betti series of k over R.

Jörgen Backelin showed in [Backelin, 1982] that $P_k^R(t) = \frac{(1+t)^n}{b_{R,k}(t)}$ holds with $\deg b_{R,k}(t) < \infty$ and Luchezar Avramov asked in [Avramov, 2002] whether $\deg b_{R,k}(t) \leq 2m$ will hold for any field k, any number r of variables and any fixed number m of monomial generators of $\langle M \rangle$. Alexander Berglund recently found an explicit formula for $b_{R,k}(t)$, using minimal models of the ring $Q/\langle M \rangle$.

I have in the course of my thesis work continued Alexander Berglund's development of a Scheme program listing all lattices on a fixed number of atoms. I have furthermore developed a program package, written in C++, capable of calculating simplicial homology over fields of arbritrary characteristic, as well as explicitly calculating $b_{R,k}(t)$ for arbritrary characteristic of the coefficient field k.

I cannot avoid assuming that the reader has some basic knowledge about algebra

[1] i.e. a set of monomials such that no x_i^2 divides any monomial and such that no monomial divides another

and combinatorics. I will assume that the reader has some familiarity with rings, ideals, fields and vector spaces as well as with generating functions. Further, I will assume knowledge about quotient structures and substructures as they may be applied in several different areas – quotients of groups, of vector spaces, of rings et.c. If further material is needed, I will refer to it with some degree of completeness, though often quite briefly. At each point, I will give references the reader can pursue to broaden the understanding of the subject matters touched.

1.2 Homological algebra

Throughout this article, we let k be a field of any characteristic; $Q = k[x_1, \ldots x_r]$; M a finite set of independent monomials, i.e. such that no $m \in M$ is divided by any $m' \in M \setminus \{m\}$; and $R = Q/\langle M \rangle$ the monomial ring we study. I use the notation $\langle M \rangle$ for the ideal generated by the elements in M. Thus, all rings we work with are graded local rings (i.e. commutative unitary noetherian with a unique graded maximal ideal). The image of the ideal $\mathfrak{m} = \langle x_1, \ldots, x_r \rangle$ is the unique graded maximal ideal in all rings we deal with. Recall that an ideal I in a graded ring R is graded if there is a set of generator such that each generator is homogenous in R.

A *module* M over the commutative ring R is a structure following the vector space axioms rather closely; thus an abelian group with the group operation denoted $+$ together with a "scalar" multiplication with ring elements denoted by $\cdot$ or by juxtaposition, such that the operations distribute over each other. Any basic linear algebra book will give a list of axioms for a vector space – those very same axioms hold for modules, except for the minute difference that the scalars of a module need not be invertible.

A module over a field is a vector space. But since the scalars to a module in the general case are from a ring, some of the vector space properties do not necessarily hold. For instance, a module need not have a basis – which is to say a generating set such that each element is uniquely written as a linear combination in the generators. Modules such that there is a basis are called *free*, and may easily be seen to be isomorphic to a direct sum of the scalar ring with itself a number of times as determined by the size of the basis. For certain arguments, it turns out that freeness captures more than is strictly necessary. This leads to the notion of *projective modules*[2]. A treatment of projective modules goes outside the scope of this thesis, but the interested reader is directed to standard works in homological algebra such as [MacLane, 1963] and [Hilton and Stammbach, 1970].

[2]many ideas in Homological algebra benefit from statements with the ideas and tools from category theory. Thus many notions are read off of diagrams asserting existences of functions between objects. Such a diagram is *dualised* by inverting all arrows, and a *dual concept* to something is what you'd get if you only reversed all arrows, that is to say functions, when you talk about it. The dual notion of projective modules is *injective modules*, which in a similar way expands on the notion of *co-free modules*; the dualisation of free modules. We will later on touch a few situations where dualisation is of interest, though in a slightly different context.

A module M is *$\mathbb{Z}$-graded* if it can be decomposed into a direct sum $\bigoplus_{i\in\mathbb{Z}} M_i$ of modules. An endomorphism f on a graded module is said to be *of degree k* if $f(M_i) \subseteq M_{i+k}$ for all i. The *homogenous* components of a graded module – i.e. the direct summand of a particular degree – are denoted by subscripting the degree.

A sequence of modules (C_i) together with homomorphisms ∂_n

$$\boldsymbol{C}: \ \ldots \xrightarrow{\partial_{n+1}} C_n \xrightarrow{\partial_n} C_{n-1} \xrightarrow{\partial_{n-1}} \ldots \tag{1.1}$$

is called a complex if $\partial_n \circ \partial_{n+1} = 0$ for all n. The complex can be described in terms of *differential graded modules* as well – a DG-module is a $\mathbb{Z}$-graded module $\boldsymbol{C}$ together with an endomorphism ∂ of degree -1 such that $\partial^2 = 0$. A complex may be made into a DG-module by taking the direct sum of all modules in the complex, with the appropriate degrees assigned to each. The complex is said to be *positive* if $C_n = 0$ for $n < 0$. It is said to be *concentrated to degree m* if $C_n = 0$ for $n \neq m$

A complex is *free* if all modules in the complex are free.

A complex $\boldsymbol{C}$ is *minimal* if $\partial\boldsymbol{C} \subseteq \mathfrak{m}\boldsymbol{C}$.

The *homology* of a DG-module $\boldsymbol{C}$ is defined to be the quotient $H(\boldsymbol{C}) = Z(\boldsymbol{C})/B(\boldsymbol{C})$ where $Z(\boldsymbol{C}) = \ker\partial$ is called the *cycles*, and $B(\boldsymbol{C}) = \operatorname{im}\partial$ is called the *boundaries*. The motivation for this terminology originates in the historical origins of the study of homology; where features of a topological space were captured by building complexes out of the space and considering the homology of the particular complex built.

A DG-module is called *acyclic* if $H(\boldsymbol{C}) = H_0(\boldsymbol{C})$. It is called *exact* if $H(\boldsymbol{C}) = 0$.

Note that $\boldsymbol{C}$ is acyclic if and only if

$$\ldots \longrightarrow C_1 \longrightarrow C_0 \xrightarrow{\varepsilon} H_0(\boldsymbol{C}) \longrightarrow 0 \tag{1.2}$$

is exact with ε the canonical surjection from C_0 to $C_0/\operatorname{im}\partial_1$.

A *free resolution* of a module A is a free acyclic complex $\boldsymbol{F}$ together with an isomorphism $H_0(\boldsymbol{F}) \cong A$. We will most often tacitly identify $H_0(\boldsymbol{F})$ and A using this isomorphism.

A *minimal free resolution* $\boldsymbol{F}$ of a module A is a free resolution which is minimal as a complex.

Given a free resolution $\boldsymbol{F}$ of the module A, let $\boldsymbol{F}\otimes B = \ldots \overset{\partial_{i+1}\otimes 1}{\to} F_i\otimes B \overset{\partial_i\otimes 1}{\to} \ldots$. We then define the *Tor functor* as $\operatorname{Tor}(A,B) = H(\boldsymbol{F}\otimes B)$ for any module B. Note that when we write $\operatorname{Tor}_i^R(A,B)$, this means precisely $H_i(\boldsymbol{F}\otimes B)$ for a resolution $\boldsymbol{F}$ of A as a R-module and with the tensor product with B as R-modules. It turns out the Tor has quite nice properties. More precisely, if we have a morphism $\alpha : A \to A'$ then we can associate another morphism $\alpha_* :$

$\mathrm{Tor}(A,B) \to \mathrm{Tor}(A',B)$. Similarily, given $\beta : B \to B'$, an induced morphism $\beta_* : \mathrm{Tor}(A,B) \to \mathrm{Tor}(A,B')$. Using category theoretical language, this means that Tor is a *covariant bifunctor*. For more details, I would like to refer the reader to [Hilton and Stammbach, 1970, III.8]

Now, the *Poincaré-Betti series* of the module N over the graded local ring[3] $(R,\mathfrak{m},k)$ is the formal power series

$$P_N^R(t) = \sum_{n\geq 0} \dim_k \mathrm{Tor}_n^R(N,k)t^n \tag{1.3}$$

If $\boldsymbol{F}$ is minimal, then $\partial_n \otimes 1(x\otimes a) = \partial(x)\otimes a = my\otimes a = y\otimes ma = y\otimes 0 = 0$ for $m \in \mathfrak{m}$ and some $y \in F_{n-1}$, since $\partial(x) \in \mathfrak{m}F_{n-1}$. Thus ∂ is the zero map, and the homology groups are given by $\mathrm{Tor}^R(N,k)_n = \ker(\partial_n \otimes_R 1)/\operatorname{im}(\partial_n \otimes_R 1) \cong F_n \otimes_R k$.

Thus, for a minimal free resolution, the $\dim_k \mathrm{Tor}^R(N,k)$ can be read off of the ranks of the individual free modules.

Note that the term *squarefree monomial ring* denotes some ring $k[x_1,\ldots,x_k]/\langle M\rangle$ where M is a set of squarefree monomials.

Theorem 1.1 (Weyman-Fröberg). *If $R = P/\langle M\rangle$ is a monomial ring, then there is a $N \geq r$ and a squarefree monomial ring $R' = k[X_1,\ldots,X_N]/\langle M'\rangle$ for some set M' of monomials in $k[X_1,\ldots,X_N]$ such that*

$$P_k^R(t) = P_k^{R'}(t)/(1+t)^{N-r}$$

.

Proof. The construction is given in [Fröberg, 1982, 30-31] and again in [Berglund, 2003, Prop. 3.6.1]. See these references for detailed proof.

For each variable x_i, the maximal occuring degree $D_i = \sup_{m\in M} deg_{x_i}(m)$ is retrieved. For the new ring R', we define D_i new variables $X_{i,j}$ that are associated to x_i, in such a manner that in every monomial $m = x_1^{d_1}\ldots x_r^{d_r}$ gives rise to a new monomial $M = X_{1,1}\ldots X_{1,d_1}X_{2,1}\ldots X_{2,d_2}\ldots X_{r,1}\ldots X_{r,d_r}$. The set of all transformed monomials M forms a generator set for an ideal in a polynomial ring on $N = \sum_i D_i$ variables as required. □

1.3 Simplicial Homology and Hilbert series

The *Hilbert series* of a free graded module $\boldsymbol{C} = (C_i)_{i\in\mathbb{Z}}$ is the formal power series $\boldsymbol{C}(t) = \sum_{i\in\mathbb{Z}} \mathrm{rank}_R\, C_i t^i$. The rank of a free module is the size of a basis for that

[3]The notation $(R,\mathfrak{m},k)$ for any local ring indicates that R is the ring, $\mathfrak{m}$ is the unique maximal ideal and $k = R/\mathfrak{m}$ is the quotient field.

module. We will only deal with Hilbert series of positively graded modules, i.e. $\boldsymbol{C}(t) = \sum_{i \in \mathbb{N}} \operatorname{rank}_R C_i t^i$. Thus, the Poincaré-Betti series of a module N over a ring R is the Hilbert series of the complex $H(\boldsymbol{F} \otimes k)$ for a free resolution $\boldsymbol{F}$ of N.

For the sake of the current discussion we fix a field k. Furthermore, we remind the reader that a module over a field is simply a vector space.

An *abstract simplicial complex* is a collection A of subsets of some finite set such that whenever $I \in A$ and $J \subset I$ it follows that $J \in A$. Partition the collection into subsets A_i such that $I \in A_i$ is equivalent to $|I| = i + 1$. The reason for the shift in index lies in the historical origins of the theory for simplicial complexes – the points in the set (and thus also the points in the subsets included in A_0) correspond to vertices in a triangulation. The subsets of two points correspond to edges, the subsets of three points to triangles. Thus it continues onto higher dimensions until the collection is exhausted. The dimension of each simplex, which is to say generalised triangle, is one less than the number of vertices spanning it. Whenever some simplex is used in the triangulation, all the components of that simplex are also present. A much more geometric view of what is happening here may be found in any textbook on algebraic topology, for instance in [Armstrong, 1983], [Bredon, 1993] or [Hatcher, 2002].

For each partition A_i, we construct a vector space C_i with basis elements indexed by the elements of A_i. In addition, we wish to fix some total order on the points of the set of points occuring in the subsets in the abstract complex. So for the singleton sets in A_0, we have $a_1, a_2, \ldots, a_r$ ordered such that $a_1 < a_2 < \cdots < a_r$. We will denote the basis elements by e_I for $I \in [r] = \{1, \ldots, r\}$ with elements $p_1, \ldots, p_i$ with the numbering fulfilling $a_{p_1} < \cdots < a_{p_i}$. The differential that makes this a complex comes as we define a boundary homomorphism by sending the basis element indexed by some $I = \{p_1, \ldots, p_i\}$ to the sum $\sum_{j=1}^{i} (-1)^j e_{I \setminus \{p_j\}}$ It has degree -1 since each basis element of C_i is mapped to a linear combination of basis elements where one point is removed from the defining set, thus placing each basis element in the linear combination in C_{i-1}. Furthermore, $\partial_{i-1} \circ \partial_i = 0$ since each occurrence of $e_{I \setminus \{p_j, p_{j'}\}}$ will arise in two ways: once from the term $(-1)^j e_{I \setminus \{p_j\}}$ and in addition to that once from the term $(-1)^{j'} e_{I \setminus \{p_{j'}\}}$. Thus we get, within $\operatorname{im} \partial_{i-1} \circ \partial_i$ that the contributions along the basis vector $b = e_{I \setminus \{p_j, p_{j'}\}}$ are given by $(-1)^{j'-1}(-1)^j b + (-1)^j (-1)^{j'} b = (-1)^{j+j'}(b - b) = 0$, since the index of $p_{j'}$ once p_j has been removed will be shifted by one step, whereas the reverse order of removal will not affect the indices.

Now, we need to verify that a different ordering of the points will not change the complex. Recall that any permutation in S_i may be written as a product of transpositions on the form $(m \quad m+1)$ for some $m < i$. Thus, it is enough to find an isomorphism between the complex generated above and the complex which may be generated by taking the basis vectors f_I and repeat the procedure described above, but with the points ordered as $a_1 < \cdots < a_{m+1} < a_m < \cdots < a_r$. Such an isomorphism is given by the function ι that sends $e_I \mapsto -f_I$ precisely when both $m, m+1 \in I$ and $e_I \mapsto f_I$ else. The mapping is motivated

by the observation that the orderings of the subset I in the two cases is identical whenever at most one of the two transposed indices occurs. ι is a bijective map from a vector space basis to another, and thus extends by linearity to a vector space isomorphism for each C_i. It remains to check that $\iota \circ \partial = \partial \circ \iota$ and thus that the two complexes are isomorphic in a manner preserving the complex structure as well.

Again, by the linearity of both ι and ∂, it is enough to check what happens with basis vectors. If a particular basis vector e_I happens to omit at least one of the two permuted points, then ι is the identity and there is not much more to show. The nontrivial case is when $m, m+1 \in I$, say that $p_{j'} = m$.

We calculate

$$\begin{aligned}
\iota \circ \partial(e_I) &= \iota \left(\sum_{j=1}^{i} (-1)^j e_{I \setminus \{p_j\}} \right) \\
&= \sum_{j=1}^{s} (-1)^j \iota(e_{I \setminus \{p_j\}}) \\
&= \sum_{j \notin \{i', i'+1\}} (-1)^j \iota(e_{I \setminus \{p_j\}}) + (-1)^{i'} \iota(e_{I \setminus \{m\}}) + (-1)^{i'+1} \iota(e_{I \setminus \{m+1\}}) \\
&= \sum_{j \notin \{i', i'+1\}} (-1)^j \left(-f_{I \setminus \{p_j\}} \right) + (-1)^{i'} f_{I \setminus \{m\}} + (-1)^{i'+1} f_{I \setminus \{m+1\}} \\
&= - \left(\sum_{j \notin \{i', i'+1\}} (-1)^j f_{I \setminus \{p_j\}} + (-1)^{i'} f_{I \setminus \{m+1\}} + (-1)^{i'+1} f_{I \setminus \{m\}} \right) \\
&= \partial(-f_I) \\
&= \partial \circ \iota(e_I)
\end{aligned} \tag{1.4}$$

Since $\boldsymbol{C}$ and $\boldsymbol{C}'$ are isomorphic complexes, they have in particular the same homological properties. We see that everything important is captured regardless of the order on the points.

Thus, we have constructed a complex $\boldsymbol{C}$ based on the abstract simplicial complex. We call this complex the *chain complex* associated with the given abstract simplicial complex. The homology of the abstract simplicial complex is defined to be $H(\boldsymbol{C})$.

The *reduced homology* $\widetilde{H}(\boldsymbol{C})$ is obtained by first augmenting the complex $\boldsymbol{C}$ with a map $\epsilon : C_0 \mapsto C_{-1}$, where C_{-1} is the vector space spanned by a single vector, indexed by $\emptyset$ and ϵ sends each generator of C_0 onto this single generator. This augmented complex is indeed a complex, since any cycle in C_0 is the image of a chain in C_1, and thus is a sum of vertices, each weighted $+1$ or -1 in such a manner that there are equally many positively weighted as negatively weighted vertices. They all sum up to 0, thus proving that $\partial^2 = 0$ everywhere. This reduced homology of $\boldsymbol{C}$ is denoted $\widetilde{H}(\boldsymbol{C})$. $H(\boldsymbol{C}) = \widetilde{H}(\boldsymbol{C})$ except for in degree 0, where $H_0(\boldsymbol{C}) = \widetilde{H}_0(\boldsymbol{C}) \oplus k$.

This construction corresponds precisely to including $\emptyset$ as a simplex of degree -1, and performing the same constructions as outlined in this section. It may be noted, though, that $\partial(\emptyset) = 0$.

1.4 The Taylor complex

One of several common ways to obtain a free resolution is due to Diana Taylor [Taylor, 1966]. It is also described in [Avramov, 2002] and [Berglund, 2003]. Given the polynomial ring $Q = k[x_1, \ldots, x_r]$ and the quotient $R = Q/\langle M \rangle$ where $M = \{m_1, \ldots, m_n\}$ we may explicitly construct a free complex $\widetilde{\boldsymbol{F}}$ by generating each $\widetilde{F}_i$ by the symbols $\{T_J \mid J \subseteq [1,n], |J| = i\}$, where $[1,n]$ denotes the set of integers from 1 to n. A differential operator is introduced, similar to the construction above, for $J = \{j_1, \ldots, j_i\}$ with $j_1 < \cdots < j_i$, by

$$\tilde{d}T_J = \sum_{l=1}^{i} (-1)^l \frac{m_J}{m_{J\setminus\{j_l\}}} T_{J\setminus\{j_l\}} \tag{1.5}$$

where $m_J = \operatorname{lcm}(m_i \mid i \in J)$. We can use this complex to form a free resolution of R over Q by completing the sequence of $\widetilde{F}_i$ with R and 0 to form the exact sequence

$$0 \to \widetilde{F}_n \to \cdots \to \widetilde{F}_0 \xrightarrow{\eta} R \to 0 \tag{1.6}$$

where η is the quotient surjection from Q to R, since $\widetilde{F}_0$ is spanned by $T_\emptyset$, and thus isomorphic to Q.

Again, similar to what we did in the earlier definition of Tor, we tensor each point in the complex $\widetilde{\boldsymbol{F}}$ by k to form the complex $\boldsymbol{F} = \widetilde{\boldsymbol{F}} \otimes k$. We can observe that $H(\boldsymbol{F}) = \operatorname{Tor}^Q(R,k)$. The really interesting property is when we find that the complex $\boldsymbol{F}$ with differential $\partial = \tilde{d} \otimes 1$ will decompose into a direct sum of subcomplexes. This is due to the fact that when tensoring with k any element of the differential with a nontrivial quotient $\frac{m_J}{m_{J\setminus\{j_l\}}}$ will vanish since all x_i are killed by the surjection to k. Thus, the only parts left in the image of the differential will be those with a coefficient of ± 1 for each T_J.

To detail this, I would first want to note that we use a graded ring, Q, with one interesting grading given by assigning to each monomial $x_1^{d_1} \ldots x_r^{d_r}$ the point $(d_1, \ldots, d_r) \in \mathbb{N}^r$. Multiplying two monomials will add their respective grades, and keeping track of this *ring grade* independently from the homology grading defined for complexes above, we can talk about most things already presented – but multigraded. A multigraded module will be decomposed into a direct sum of homogenous components. Multiplication by a ring element onto an element of a multigraded module will raise the ring degree of the product.

In this case, the decomposition of $\boldsymbol{F}$ is indexed by the points in the lattice[4] L_M, i.e. over all least common multiples of monomials in M. For some point

[4] more about that later

$l \in L_M$, we can construct a simplicial complex that will be closely related to $\boldsymbol{F}$. Take the set of all T_J such that $m_J = l$. This set has a maximal element under set inclusion, namely $J_l = \{p \in [1,n] \mid m_p | l\}$. Among all these sets, some are smaller than the maximal. Let $\Delta_l = \{J_l \setminus J \mid m_J = l\}$. This makes Δ_l an abstract simplicial complex. Δ_l will be reversed and translated when compared to the homological behaviour of the component of the decomposition of $\boldsymbol{F}$ – when you go down in homology grades in $H(\boldsymbol{F})$, you will correspondingly rise in the homology grades in $H(\Delta_l)$. Using this, you can find that $\dim_k H_i(\boldsymbol{F}) = \sum_l \dim_k H_{k_l - i}(\Delta_l)$ for appropriate choices of k_l.

A similar decomposition is used by Berglund further on when constructing the denominator polynomials.

1.5 Lattice theory

Given the correspondences between points in the LCM lattice of a monomial ideal in the Taylor construction, one would be inspired to see whether the lattice structure contains information about the Tor dimensions as such. This is discussed further later on, and related to this interest springs the interest in listing all lattices that may occur as such LCM lattice. To prepare for these later themes, a slight introduction to lattice theory would be appropriate. Further good information about lattices may be found in [Stanley, 1997] or in [Grätzer, 1998]

A *lattice* is a non-empty set L closed under two associative, commutative and idempotent operations $\vee$ and $\wedge$, called join and meet, satisfying $x \wedge (x \vee y) = x = x \vee (x \wedge y)$. From this follows immediately that $x \vee y = x \Leftrightarrow x \wedge y = y$. When several lattice structures are present, we will keep them separate by labeling the operations $\vee_L$ and $\wedge_L$.

A *$\vee$-semilattice* is a set L closed under an associative, commutative and idempotent operation $\vee$. Thus a lattice is both a $\vee$-semilattice and a $\wedge$-semilattice.

If $x \vee y = x$ in a lattice L, then we say that $y \leq x$. This defines a partial order on the elements of the lattice. Indeed, $x \leq x$ by idempotency, $x \leq y$ and $y \leq x$ implies $x = x \vee y = y$. And finally, if $x \leq y$ and $y \leq z$, then $x \vee z = x \vee (y \vee z) = (x \vee y) \vee z = y \vee z = z$ and thus $x \leq z$.

A subset of a lattice L is an *antichain* if no two elements are comparable, i.e. for any two x, y in the antichain, neither $x \leq y$ nor $y \leq x$ holds.

A *filter* in a lattice L is a subset $F \subseteq L$ such that whenever $l \in F$ and $l' \geq l$, this implies $l' \in F$. An *ideal* in a lattice L is a subset $F \subseteq L$ such that whenever $l \in F$ and $l' \leq l$, this implies $l' \in F$. In other words, filters are closed under "greater than" in the partial order above, and ideals are closed under "less than". A *principal* ideal (or filter) is the set of all elements in the lattice that are less than (greater than) a single generating element.

A lattice is *complete* if for any $S \subset L$, both $\bigvee_{l \in S} l$ and $\bigwedge_{l \in S} l$ exist in L. Similarily, a $\vee$-semilattice is complete if $\bigvee_{l \in S} l$ exists in the semilattice.

Any complete lattice L contains two special elements, $\mathbb{0}$ and $\mathbb{1}$, with the property that $\mathbb{0} \wedge x = \mathbb{0}$ and $\mathbb{1} \vee x = \mathbb{1}$ for any $x \in L$. The elements a_i such that if $\mathbb{0} \leq l \leq a_i$, either $l = \mathbb{0}$ or $l = a_i$ are called *atoms*.

Any finite lattice is complete, since any finite nonempty join or meet exists by associativity. Empty join is taken to be $\mathbb{0}$ and empty meet to be $\mathbb{1}$.

Proposition 1.2. *A complete $\vee$-semilattice L is a lattice.*

Proof. For L to be a lattice, we need to define $\wedge$ of a pair of elements in a way that satisfies the defining identities. Pick any two elements a and b from L. Let $S_{a,b} = \{x \in L \mid x \vee a = a \text{ and } x \vee b = b\}$. Define $a \wedge b = \bigvee_{l \in S_{a,b}} l$. This element exists, since L is complete.

We need to verify that

1. $x \wedge (x \vee y) = x$
2. $x \vee (x \wedge y) = x$

Indeed, $x \wedge (x \vee y)$ is the join of all elements z such that $z \vee x = x$ and $z \vee (x \vee y) = x \vee y$. But associativity reduces the second condition to a consequence of the first. Thus $x \wedge (x \vee y) = \bigvee_{z \mid z \vee x = x} z = x$.

Furthermore, $x \vee (x \wedge y) = x \vee \left(\bigvee_{z \mid z \vee x = x} z \right) = x$ by associativity of $\vee$. □

A *morphism of lattices* is a function $f : L \to M$ such that $f(x \vee y) = f(x) \vee f(y)$ and $f(x \wedge y) = f(x) \wedge f(y)$. Similarily, a *morphism of semilattices* is a function $f : L \to M$ such that $f(x \vee y) = f(x) \vee f(y)$. Obviously, this defines two different categories: Lat and SLat. A morphism is said to be *injective* or *surjective* if it is injective or surjective as a function on the underlying sets.

Two lattices M and L are said to be isomorphic if there is a injective and surjective morphism between them.

A *sublattice* M of a lattice L is a subset, closed under the lattice operations in L. M is a lattice by inheritance. Similarily, *subsemilattices* are defined.

A *congruence relation* on a lattice is an equivalence relation $\sim$ such that if $a_0 \sim a_1$ and $b_0 \sim b_1$, then $a_0 \vee b_0 \sim a_1 \vee b_1$ and $a_0 \wedge b_0 \sim a_1 \wedge b_1$. Congruence relations on semilattices are similarily defined. For the applications later on, the most interesting case is that of semilattice congruences; where the second $\wedge$-related condition is ignored.

Given a congruence relation $\sim$ on a lattice L, the *quotient lattice* $L/\sim$ may be defined as the set of equivalence classes under the relation with the operations $[a]_\sim \vee [b]_\sim = [a \vee b]_\sim$ and $[a]_\sim \wedge [b]_\sim = [a \wedge b]_\sim$. This definition is valid since if $a \sim a_1$ and $b \sim b_1$, then $a \vee b \sim a_1 \vee b_1$ and thus $[a \vee b]_\sim = [a_1 \vee b_1]_\sim$ and similarily for $\wedge$. All quotient lattices are images of surjective lattice morphisms. The definition holds equally well for semilattices.

Proposition 1.3. *Let $f : L \to M$ be a morphism of lattices. Then $f(L)$ is isomorphic to $L/\sim$, where $a \sim b$ whenever $f(a) = f(b)$. This result holds equally well for semilattices.*

Proof. The relation $\sim$ is an equivalence relation, since reflexivity, transitivity and symmetry all can be carried through f; and thus follow from the similar properties of $=$. Further, $\sim$ is a congruence relation since if $a_0 \sim a_1$ and $b_0 \sim b_1$, then $f(a_0 \vee b_0) = f(a_0) \vee f(b_0) = f(a_1) \vee f(b_1) = f(a_1 \vee b_1)$ shows $a_0 \vee b_0 \sim a_1 \vee b_1$. Similarily, we show $a_0 \wedge b_0 \sim a_1 \wedge b_1$.

We thus need to show the existence of an isomorphism $\phi : L/\sim \to f(L)$. Take $\phi([a]_\sim) = f(a)$. The function ϕ is obviously surjective. It's injective, since if $f(a) = f(b)$, then by the definition of $\sim$, $a \sim b$. Finally, it is a morphism, since $\phi([a]_\sim \vee [b]_\sim) = \phi([a \vee b]_\sim) = f(a \vee b) = f(a) \vee f(b) = \phi([a]_\sim) \vee \phi([b]_\sim)$. □

An *atomic lattice* is a lattice L such that any lattice point l can be written as a join of atoms.

The direct product of lattices is formed in the obvious way – $L \times L'$ is the set of ordered pairs (l, l') with $(a, a') \vee (b, b') = (a \vee b, a' \vee b')$ and $(a, a') \wedge (b, b') = (a \wedge b, a' \wedge b')$. This can easily be extended to finite direct products. The direct product $\prod_{i=1}^{n} L$ is denoted by L^n.

Some specific lattices are worth mentioning. The set $[1, n]$ of integers $1, \ldots, n$ forms a lattice with $a \vee b = \max(a, b)$ and $a \wedge b = \min(a, b)$. We will denote this lattice with $\underline{n}$. The lattice $\underline{2}^n$ is called the boolean lattice on n elements, and denoted by B_n. We will habitually use the symbols 0 and 1 instead of 1 and 2 when talking about the elements of $\underline{2}$. The lattice of subsets of a finite set S with n elements is isomorphic to B_n by the following construction. Let the elements of S be ordered in some way, $s_1, \ldots, s_n$. Then a subset $S' \subset S$ is mapped to the element $(b_1, \ldots, b_n) \in B_n$ such that $b_i = 1$ whenever $s_i \in S'$ and $b_i = 0$ otherwise.

This particular mapping will occur a lot further on. It's both the basis of several connections to the boolean lattice and the method chosen to represent sets and subsets in the implementation of simplicial homology in chapter 2.

Given a lattice L, a new lattice L^* can be constructed using the same base set, such that $a \vee_L b = a \wedge_{L^*} b$ and $a \wedge_L b = a \vee_{L^*} b$. The lattice L^* thus constructed is called the *dual lattice* to L. The dual thus defined has the property that $a \leq_L b$ is equivalent to $b \leq_{L^*} a$.

Alternatively, let $\mathrm{Hom}_\vee(L, L')$ be the set of $\vee$-semilattice-morphisms from L to L'. This set forms a semilattice with $\phi \vee \psi$ defined as the morphism that takes $l \in L$ to $\phi(l) \vee \psi(l) \in L'$. This is obviously associative, commutative and idempotent, since $\vee_{L'}$ is. We will take a closer look at the semilattice $\mathrm{Hom}_\vee(L, \underline{2})$. Any morphism $\phi \in \mathrm{Hom}_\vee(L, \underline{2})$ is entirely described by the ideal $\phi^{-1}(0) \subseteq L$. For any elements $a, b \in \phi^{-1}(0)$, we have that $a \vee b \in \phi^{-1}(0)$. Thus $f = \bigvee_{l \in \phi^{-1}(0)} l$ exists, and $\phi^{-1}(0)$ is the principal ideal associated with f. Thus a correspondence $L \to \mathrm{Hom}_\vee(L, \underline{2})$ is established. Any ϕ gives rise to a separate element $f \in L$, and conversely, any f gives rise to a morphism that maps the principal ideal generated by f to 0, and everything else to 1.

Proposition 1.4. $L^* \cong \mathrm{Hom}_\vee(L, \underline{2})$

Proof. We have already established that there is a bijection α between the elements of L and $\mathrm{Hom}_\vee(L, \underline{2})$, that sends $f \in L$ to $\alpha_f \in \mathrm{Hom}_\vee(L, \underline{2})$ defined by $\alpha_f(l) = 0$ if $f \geq l$ and $\alpha_f(l) = 1$ otherwise.

It remains to investigate the action of $\vee$ in $\mathrm{Hom}_\vee(L, \underline{2})$. Let α_f and α_g be morphisms induced by f and g in L. Then $(\alpha_f \vee \alpha_g)(l) = 0$ precisely when $\alpha_f(l) = 0$ and $\alpha_g(l) = 0$ and equal to 1 otherwise. This is precisely the morphism that is 0 on all elements that are less than both f and g, and thus less than $f \wedge g$. Thus, $\alpha_f \vee \alpha_g = \alpha_{f \wedge g}$, and thus the isomorphism above is shown, as $\wedge_L = \vee_{L^*}$. □

Elements l of a lattice L that cannot be written as a join of elements from the set $L \setminus \{l\}$ are called *irreducible*. The set of irreducible elements is denoted by IL. The set of irreducible elements in the dual are called *co-irreducible*, and denoted as I^*L.

We will denote the smallest $\vee$-semilattice containing a subset M by $\langle M \rangle$; the semilattice *spanned by* M. The possible ambiguity of notation when compared to an ideal generated by a set in a ring will be resolved by context.

1.6 Previous results

In particular, we are interested in the behaviour of $\mathrm{Tor}^R(k, k)$ for the monomial ring R over k. In [Backelin, 1982], Jörgen Backelin shows that the Poincaré-Betti series of a monomial ring, such as our R, is a rational function of the form $P_k^R(t) = \frac{(1+t)^n}{b_{R,k}(t)}$.

Let M be a set of monomial generators for some monomial ideal. Given any subset $I \subseteq M$, we may define $m_I = \mathrm{lcm}\{p \mid p \in I\}$. The set of all m_I can be given a lattice structure L_M by $m_I \vee m_J = m_{I \cup J} = \mathrm{lcm}(m_I, m_J)$. This structure induces a partial order by $m_I \preceq m_J$ iff $\mathrm{lcm}(m_I, m_J) = m_J$, which in turn implies that $m_I | m_J$. We further define a graph Γ_M with vertex set M and edges (m, m') precisely when $\gcd(m, m') \neq 1$. Luchezar L. Avramov proves in [Avramov, 2002]

that if $\Gamma_M \cong \Gamma_{M'}$ and $L_M \cong L_{M'}$, then $b_{R,k}(t) = b_{R',k}(t)$, where $R = Q/\langle M\rangle$ and $R' = Q/\langle M'\rangle$. Note that the GCD graph here defined is the complement of the restriction of the GCD graph in [Avramov, 2002] to the atoms of the LCM lattice. Among the consequences of this result is that for a fixed number of generators for the monomial ideal, there can exist only a finite number of different denominator polynomials. Thus, Avramov asked in [Avramov, 2002] whether the degrees of the denominator polynomials is bounded for any field k, any number of variables x_i in the polynomial ring, any set of fixed cardinality of monomial ideal generators

$$d(n) = \sup_k \sup_r \sup_{|M|=n} \deg(b_{k[x_1,\ldots,x_r]/\langle M\rangle,k}(t)) \tag{1.7}$$

Let $M = \{x_1^2, \ldots, x_n^2\}$. Then one may verify that $b_{Q/\langle M\rangle}(t) = (1-t^2)^n$ – a polynomial of degree $2n$. Whence $d(n) \geq 2n$ – and Avramov conjectures in [Avramov, 2002] that equality holds in general. For $n = 2$ and $n = 3$, there are very few possibilities to check to ascertain that equality holds. In [Berglund, 2003], Alexander Berglund showed that equality holds for $n = 4$ and for $n = 5$ using a number of criteria implying equality and a computer-generated list of all cases for $n = 5$.

In particular, in [Berglund, 2003], Berglund gave an algorithm to list all atomic semi-lattices with a fixed number of atoms based on a duality argument relating atomic semi-lattices with join-closed subsets of a boolean lattice. I have based part of my own work on his implementation of this algorithm.

During the spring of 2004, Berglund found a very explicit formula for the denominator polynomials of the Poincaré-Betti series using the theory of minimal models. More specifically, he studies the multigraded Poincaré series

$$P_k^R(x_1, \ldots, x_r, t) = \sum_{i\geq 0, \alpha\in\mathbb{Z}^r} \dim_k \operatorname{Tor}_{i,\alpha}^R(k,k) x^\alpha t^i \tag{1.8}$$

which is a more fine-tuned way of describing the behaviour of $\operatorname{Tor}(k,k)$ than the simple series given before; the $P_k^R(t)$ defined in (1.3) is related to this by $P_k^R(t) = P_k^R(1,\ldots,1,t)$. In the multigraded case, $P_k^R = \frac{\prod_{i=1}^r (1+x_i t)}{b_{R,k}(x_1,\ldots,x_r,t)}$. Berglund finds the formula

$$b_{R,k}(x_1,\ldots,x_r,t) \equiv \prod_{l\in L_M} (1 - l p_l(t)) \pmod{\langle x_1^2,\ldots,x_r^2\rangle} \tag{1.9}$$

multiplied over all monomials in the monomial lattice. Each p_l is the Hilbert series of a specific simplicial complex:

To each lattice point l, we associate the set S_l of atoms a_i fulfilling $a_i | l$. To a set S of monomials, we denote $m_S = \operatorname{lcm}(s \mid s \in S)$. Furthermore, we may construct the complex $\Delta_l = \{T \subseteq S_l \mid m_T \neq l \text{ or } T \text{ is not connected}\}$ where connectedness is taken on the induced subgraphs of Γ_M. Thus, the polynomial $p_l(t)$ is defined by

$$p_l(t) = \sum_{i=2}^{|l|} \dim_k \widetilde{H}_{i-3}(\Delta_l) t^i \tag{1.10}$$

where $|l|$ denotes the total degree of l, i.e. the sum of the degrees of all occuring variables in l and homology of the complex is taken over the field k.

Chapter 2

Implementation details

A user of this tool need not necessarily know much about the internals. Using the method of Weyman-Fröberg, the program handles without problem non-squarefree monomials in the input.

The core work of this thesis lies in the implementation of a command line tool to calculate $b_{R,k}(x_1, \ldots, x_r, t)$ for some ring $R = Q/\langle M \rangle$ where M is a set of monomials in $Q = k[x_1, \ldots, x_r]$. At its current state, the program ties into existing packages for some computational tasks. In particular, it ties into two packages: Pari v2.2 for linear algebra in field of any characteristic, and GMP for keeping track of bitstrings of arbitrary length.

The surrounding code deals with routines to do all the steps of the calculations needed – such as parsing user input; constructing the differential morphisms as matrices in a format that can use Pari for calculating the matrix ranks from which then the homology ranks can easily be found; constructing the appropriate complexes in each lattice point for use with Berglunds formula – which in turn includes the construction of a graph connectivity algorithm.

Internally, a squarefree monomial is just the set of variables that occur in the monomial. Similarily, a simplex is also just a subset of some canonical set. To cover both these needs and encapsulate them in a programmer-friendly environment, I have implemented the class `Monomial`, which contains a multiprecision integer called `bitmask`, a signed `long` called `zdegree` and a static translation table implemented as a STL map[1] which maps STL strings to `unsigned long`. The reason for this is that a subset may be seen as a bitstring representing the characteristic function; i.e. for some enumeration of the (finite) superset, the subset bitstring has a 1 whenever the particular element with that index is a member of the subset and 0 else. Thus, unions and intersection can be taken with the bitlevel and and inclusive or logical operations. The member `zdegree` contains the largest n such that t^n divides the monomial. Usually, `zdegree` is

[1] also known as associative array to programmers from other traditions

equal to 1.

A number of operations and functions are implemented in the class, including wrappers for multiplying a monomial with a variable (i.e. setting a bit) and multiplying two monomials with each other. Care must be taken when using the present code for calculations in $Q[t]/\langle x_1^2, \ldots, x_r^2\rangle$ to check for relative primality externally.

Using the class `Monomial`, three other classes are implemented:

`MonomialIdeal`, which encapsulates manipulations on a monomial ideal and the related LCM lattice and GCD graphs, including functions to calculate the Poincaré-Betti series denominator. Addition of a new monomial to the instantiated monomial ideal is done by overloading the `+=` operator.

`SimplicialComplex`, contains the machinery to handle abstract simplicial complexes, together with routines to calculate the Hilbert series of the homology of the complex.

`Squarefree`, which implements a polynomial in $\mathbb{Z}[x_1, \ldots, x_r, t]/\langle x_1^2, \ldots, x_r^2\rangle$. Such a polynomial is stored internally as a map from `Monomial` to `long`. Various operators, most notably `+=` and `*=` are overloaded to facilitate addition and multiplication both of `Monomials` and of `Squarefree`. Within these multiplications, care is taken to only keep products of relatively prime terms.

2.0.1 Complexity issues

The implementation here offered is in several ways rather naïve, and is focused on correct calculation albeit not necessarily efficient. Further work will include optimisations on several levels, some of which I'd like to outline here.

To begin with, the formula implemented is possibly not the most efficient form of the formula. By expanding the product $\prod_{l\in L}(1 - lp_l(t))$, it can be observed that many terms will cancel due to the squarefree properties. This cancellation can be made beforehand, and Berglund offers the sum over all subsets N of the generating monomials such that no two elements of N has a common factor, with m_N denoting the product of all monomials in N,

$$b_{R,k}(x_1, \ldots, x_r, t) = 1 + \sum_N m_N(-1)^{|N|+2}\widetilde{H}(\Delta_{M_N})(t) \tag{2.1}$$

For an implementation of this form to be desirable, we would have to have a higher rate of efficiency in finding the independent subsets N than in calculating and discarding terms in products of squarefree polynomials.

Another point of attack is at the simplicial complexes calculated. For each set M of monomials, the complexes calculated in each latticepoint $l \in L_M$ is a

subcomplex of the complex calculated in $\mathbb{1}_{L_M}$. This property is right now not used at all, and could possibly speed up computations significantly.

Lingering at the homology calculations, memory consumption issues currently affect the usability of the program. One of the examples I have tried to work is to calculate the denominator polynomial of the initial terms ideal of a Gröbner basis of the ideal generated by

$$
\begin{aligned}
&x_1 + x_2 + x_3 + x_4 + x_5 + x_6, \\
&x_1x_2 + x_2x_3 + x_3x_4 + x_4x_5 + x_5x_6 + x_1x_6, \\
&x_1x_2x_3 + x_2x_3x_4 + x_3x_4x_5 + x_4x_5x_6 + x_1x_5x_6 + x_1x_2x_6, \\
&x_1x_2x_3x_4 + x_2x_3x_4x_5 + x_3x_4x_5x_6 + x_1x_4x_5x_6 + x_1x_2x_5x_6 + x_1x_2x_3x_6, \\
&x_1x_2x_3x_4x_5 + x_2x_3x_4x_5x_6 + x_1x_3x_4x_5x_6 + \\
&\quad x_1x_2x_4x_5x_6 + x_1x_2x_3x_5x_6 + x_1x_2x_3x_4x_6, \\
&x_1x_2x_3x_4x_5x_6 - y^6
\end{aligned}
$$

The initial terms ideal has 100 generators and produces a lattice with 11443 elements. Invariably, the stack allocated by Pari to fill with calculation elements fills up after between 200 and 400 treated latticepoints, depending on what size we allow the stack to allocate[2]. This problem may be lessened by implementing a linear algebra package over arbritrary characteristic working with sparse matrices – i.e. instead of saving a data structure with each matrix element sequentially stored according to some storage scheme, the matrix is represented by tuples (i, j, a_{ij}) of position as well as the value of the matrix at that particular position. If the matrix, as is the case in these calculations, has a substantial amount of zeroes, storage will be lessened significantly.

Finally, cosmetics can always be improved. In this case, integration with other tools would be highly desirable – for instance through higher customization of the level of interaction available (for batch runs) and output formats compatible with more applications, including cut-and-pasteable into LaTeX.

2.1 Test suite and verifications

I have used several suites of previous investigations to develop a test suite during the development of the program. As a demonstration of the usage of the program, I shall present the test suite here, with input lines to the `poincare` calculator as well as the expected output and a translation to bigraded and singly graded Poincaré-Betti series.

Throughout, I shall change the notation of the original examples slightly, using $a, b, c, \ldots$ as variables in my rings and t as the series variable. Each program run is done from scratch, i.e. either with a `clear` command prepended or immediately after starting the program. In each case, I give the command for

[2]tests have been performed up to a stack size of 1 000 000 000 bytes

preparing the monomial ideal. The Poincaré-Betti series denominator is then calculated using the `denominator` command.

In [Backelin and Fröberg, 1985], Jörgen Backelin and Ralf Fröberg listed the Poincaré-Betti series for a classification of Artinian rings with at most three variables. These rings are classified with a digit denoting the number of monomial generators and a letter indexing different rings with equal number of generators. A list of these rings with commands for entering the monomial ideal into `poincare` is given in table 2.1 on page 21.

0a 1
1a `add monomial a^2` $1 - a^2t^2$
2a `add monomial a^2 b^2` $1 - a^2t^2 - b^2t^2 + a^2b^2t^4$
2b `add monomial a^2 a*b` $1 - a^2t^2 - abt^2 - a^2bt^3$
3a `add monomial a^2 b^2 c^2` $1 - a^2t^2 - b^2t^2 - c^2t^2 + a^2b^2t^4 + a^2c^2t^4 + b^2c^2t^4 - a^2b^2c^2t^6$
3c `add monomial a^2 a*c b^2` $1 - a^2t^2 - act^2 - b^2t^2 - a^2ct^3 + a^2b^2t^4 + ab^2ct^4 + a^2b^2ct^5$
3d `add monomial a^2 a*b b^2` $1 - a^2t^2 - abt^2 - b^2t^2 - a^2bt^3 - ab^2t^3$
3e `add monomial a^2 a*b a*c` $1 - a^2t^2 - abt^2 - act^2 - a^2bt^3 - a^2ct^3 - abct^3 - a^2bct^4$
4a `add monomial a^2 a*b b^2 c^2` $1 - a^2t^2 - abt^2 - b^2t^2 - c^2t^2 - a^2bt^3 - ab^2t^3 + a^2c^2t^4 + abc^2t^4 + b^2c^2t^4 + a^2bc^2t^5 + ab^2c^2t^5$
4b `add monomial a^2 a*c b^2 b*c` $1 - a^2t^2 - act^2 - bct^2 - b^2t^2 - a^2ct^3 - abct^3 - b^2ct^3 + a^2b^2t^4 + a^2b^2ct^5$
4c `add monomial a^2 a*b a*c b^2` $1 - a^2t^2 - abt^2 - act^2 - b^2t^2 - a^2bt^3 - a^2ct^3 - abct^3 - ab^2t^3 - a^2bct^4$
5a `add monomial a^2 a*b a*c b^2 c^2` $1 - a^2t^2 - abt^2 - act^2 - b^2t^2 - c^2t^2 - a^2bt^3 - a^2ct^3 - abct^3 - ab^2t^3 - ac^2t^3 - a^2bct^4 + b^2c^2t^4 + ab^2c^2t^5$
5b `add monomial a^2 a*b a*c b^2 b*c` $1 - a^2t^2 - abt^2 - act^2 - bct^2 - b^2t^2 - a^2bt^3 - a^2ct^3 - 2abct^3 - ab^2t^3 - b^2ct^3 - a^2bct^4 - ab^2ct^4$
6a `add monomial a^2 a*b a*c b^2 b*c c^2` $1 - a^2t^2 - abt^2 - act^2 - bct^2 - b^2t^2 - c^2t^2 - a^2bt^3 - a^2ct^3 - 2abct^3 - ab^2t^3 - b^2ct^3 - ac^2t^3 - bc^2t^3 - a^2bct^4 - ab^2ct^4 - abc^2t^4$

Table 2.1: Calculation of Poincaré-Betti denominators of short Artinian rings

Mordechai Katzman gives in [Katzman, 2004] a number of examples of graph ideals with characteristic dependent Betti numbers. Katzman starts out considering the Stanley-Reisner face ring of the 6-point triangulation of the real projective plane given in figure 2.1. This ring is well known to display different behaviour in characteristic 2 than otherwise. Furthermore, Katzman refines,

from this, two examples of graphs generating squarefree ideals with generator degree 2 such that the Betti numbers vary with the characteristic of the base field. Finally, Katzman lists four graphs which are in some sense minimal with the property of dependence on characteristic. These may be found in table 2.2.

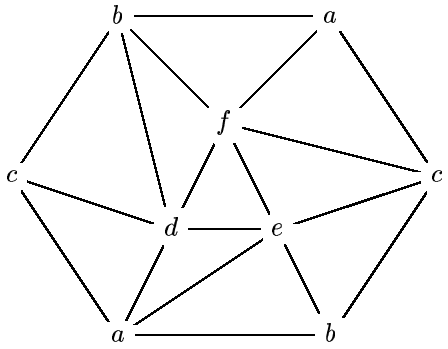

Figure 2.1: A triangulation of the real projective plane

For the graphs G, H, G_1, G_2, G_3 and G_4, the calculations run into the memory management problems outlined earlier, so, alas, I can not give the multigraded Poincaré-Betti series for those cases. For the case of the projective plane, though, we observe that if we evaluate the polynomials found with $a = b = c = d = e = f = 1$, we get the polynomials $b(t) = 1 - 10t^2 - 15t^3 - 6t^4$ and $b_2(t) = 1 - 10t^2 - 15t^3 - 7t^4 - t^5$ (the latter polynomial for the case char = 2. Katzman gives the Betti numbers for the ring as given by the Hilbert polynomials $H(t) = 1 + 10t + 15t^2 + 6t^3$ and $H_2(t) = 1 + 10t + 15t^2 + 7t^3 + t^4$. We note that one interesting property quotient rings R of $Q = k[x_1, \ldots, x_r]$ may have is that of being *Golod*[3]. This happens if

$$P_k^R(x_1, \ldots, x_r, t) = \frac{\prod_{i=1}^{r}(1 + x_i t)}{1 - t(P_R^Q(x_1, \ldots, x_r, t) - 1)} \tag{2.2}$$

where

$$P_{S'}^S(x_1, \ldots, x_r, t) = \sum_{i, \alpha_1, \ldots, \alpha_r} \dim_k \operatorname{Tor}_{i, \alpha_1, \ldots, \alpha_r}^S(S', k) x_1^{\alpha_1} \ldots x_r^{\alpha_r} t^i \tag{2.3}$$

As $b(t)$ and $b_2(t)$ are the denominator polynomials of $P_k^R(1, \ldots, 1, t)$ for R the face ring of the projective plane triangulation given, and the Hilbert polynomials of the Betti numbers are simply $P_R^Q(1, \ldots, 1, t)$ we see that R is Golod precisely if $H(t) = 1 - t(b(t) - 1)$ and $H_2(t) = 1 - t(b_2(t) - 1)$. These identities hold, and thus we may conclude that R is Golod.

One more family of examples of characteristic dependence are the Lens spaces $L(p, q)$. Each $L(p, q)$ for prime p and q will have a fundamental group with p-torsion, and thus characteristic dependent homology. The associated Stanley-Reisner face rings will therefore most probably have characteristic dependent

[3] in particular, it is the surjection $Q \to R$ that is said to be Golod

$\mathbb{P}^2(\mathbb{R})$

```
add monomial a*b*c a*b*e a*c*f a*d*e a*d*f
add monomial b*c*d b*d*f b*e*f c*d*e c*e*f
```

char $\neq 2$ $1 - abct^2 - abet^2 - acft^2 - beft^2 - ceft^2 - bcdt^2 - adet^2 - cdet^2 - adft^2 - bdft^2 - abcet^3 - abcft^3 - abeft^3 - aceft^3 - bceft^3 - abcdt^3 - abdet^3 - acdet^3 - bcdet^3 - abdft^3 - acdft^3 - bcdft^3 - adeft^3 - bdeft^3 - cdeft^3 - abceft^4 - abcdet^4 - abcdft^4 - abdeft^4 - acdeft^4 - bcdeft^4$

char $= 2$ $1 - abct^2 - abet^2 - acft^2 - beft^2 - ceft^2 - bcdt^2 - adet^2 - cdet^2 - adft^2 - bdft^2 - abcet^3 - abcft^3 - abeft^3 - aceft^3 - bceft^3 - abcdt^3 - abdet^3 - acdet^3 - bcdet^3 - abdft^3 - acdft^3 - bcdft^3 - adeft^3 - bdeft^3 - cdeft^3 - abceft^4 - abcdet^4 - abcdft^4 - abdeft^4 - acdeft^4 - bcdeft^4 - abcdeft^4 - abcdeft^5$

G

```
add monomial a*b a*c a*g a*h a*j b*c b*h b*i b*l c*g c*i c*k d*e
add monomial d*f d*h d*k e*f e*g e*l f*i f*j g*j g*k g*l h*j h*k
add monomial h*l i*j i*k i*l j*k j*l k*l
```

H

```
add monomial a*c a*g a*h a*j c*g c*i c*k d*e d*f d*h d*k e*f e*g
add monomial e*l f*i f*j g*l h*j h*k i*k i*l k*l
```

G_1

```
add monomial a*e a*f a*h a*j b*e b*f b*i b*k c*g c*h c*i c*k d*g
add monomial d*h d*j d*k e*h e*i f*j f*k g*i g*j h*k
```

G_2

```
add monomial a*d a*e a*h a*i b*e b*f b*h b*j b*k c*f c*g c*i c*j
add monomial d*g d*h d*k e*i e*j e*k f*h f*i f*k g*j g*k
```

G_3

```
add monomial a*d a*e a*h a*i b*e b*f b*h b*j b*k c*f c*g c*i c*j
add monomial d*g d*h d*k e*i e*j e*k f*h f*i f*k g*j g*k i*k
```

G_4

```
add monomial a*d a*e a*g a*h b*e b*f b*h b*j b*k c*f c*g c*i c*j
add monomial d*h d*i d*j d*k e*g e*i e*k f*h f*i g*j i*k j*k
```

Table 2.2: Calculation of Poincaré-Betti denominators of characteristic dependent graph ideal quotient rings

Poincaré-Betti series. Alas, the complexity issues make me unable to display characteristic dependent Stanley-Reisner rings, but the homology calculations can easily be carried through. Included in the `poincare` source distribution is a source file `lens.c` that contains code for generation of triangulations of $L(p, 1)$ using the algorithm described in [Birmingham, 1995, pp 5762-5764].

Please note that, when duplicating the examples here given, all the commands need to be given on a single line. Should you want to break up the entry of simplicial complexes or for that matter of monomial ideals on several lines, make sure each line starts with one of the commands `add monomial` or `add simplex` as appropriate.

Example 2.1 Characteristic dependence of homology of triangulations of the Lens spaces $L(p, 1)$

```
% poincare
Welcome to the Poincaré calculator. You can use this program to
calculate simplicial homology over prime fields and to calculate
the denominator polynomial of the Poincaré-Betti series of monomial
rings.
(c) 2004 Mikael Johansson
This program is released under the MIT License

    [add simplex command for L(3,1) cut out]

> homology
Calculating homology ranks...
*****  Hilbert series of simplicial homology *****
ZZ^3
> char 3
New characteristic: 3
> homology
Calculating homology ranks...
*****  Hilbert series of simplicial homology *****
ZZ + ZZ^2 + ZZ^3
> char 0
New characteristic: 0
> clear

    [add simplex command for L(5,1) cut out]

> homology
Calculating homology ranks...
*****  Hilbert series of simplicial homology *****
ZZ^3
> char 5
New characteristic: 5
> homology
Calculating homology ranks...
*****  Hilbert series of simplicial homology *****
ZZ + ZZ^2 + ZZ^3
> quit

Thanks for visiting.
```

Example 2.2 Missing commands from example 2.1 for triangulations of lens spaces.

```
L(3,1):
add simplex 1*2*4*5 1*2*5*6 2*3*5*6 1*2*6*7 2*3*6*7 3*4*6*7
1*2*7*8 2*3*7*8 3*4*7*8 4*5*7*8 1*2*8*9 2*3*8*9 3*4*8*9 4*5*8*9
5*6*8*9 1*3*4*10 1*4*5*10 1*5*6*10 1*6*7*10 1*7*8*10 2*3*9*10
3*4*9*10 4*5*9*10 5*6*9*10 6*7*9*10 1*2*4*12 1*3*4*12 3*4*6*12
3*5*6*12 5*6*8*12 5*7*8*12 1*2*9*12 7*8*10*12 2*9*10*12 7*9*10*12
2*4*11*12 4*6*11*12 6*8*11*12 2*10*11*12 8*10*11*12 2*3*5*13
2*4*5*13 4*5*7*13 4*6*7*13 6*7*9*13 1*8*9*13 6*8*9*13 1*3*10*13
2*3*10*13 1*8*10*13 2*4*11*13 4*6*11*13 6*8*11*13 2*10*11*13
8*10*11*13 1*3*12*13 3*5*12*13 5*7*12*13 1*9*12*13 7*9*12*13

L(5,1):
add simplex 1*2*4*5 1*2*5*6 2*3*5*6 1*2*6*7 2*3*6*7 3*4*6*7
1*2*7*8 2*3*7*8 3*4*7*8 4*5*7*8 1*2*8*9 2*3*8*9 3*4*8*9 4*5*8*9
5*6*8*9 1*2*9*10 2*3*9*10 3*4*9*10 4*5*9*10 5*6*9*10 6*7*9*10
1*2*10*11 2*3*10*11 3*4*10*11 4*5*10*11 5*6*10*11 6*7*10*11
7*8*10*11 1*2*11*12 2*3*11*12 3*4*11*12 4*5*11*12 5*6*11*12
6*7*11*12 7*8*11*12 8*9*11*12 1*2*12*13 2*3*12*13 3*4*12*13
4*5*12*13 5*6*12*13 6*7*12*13 7*8*12*13 8*9*12*13 9*10*12*13
1*3*4*14 1*4*5*14 1*5*6*14 1*6*7*14 1*7*8*14 1*8*9*14 1*9*10*14
1*10*11*14 1*11*12*14 2*3*13*14 3*4*13*14 4*5*13*14 5*6*13*14
6*7*13*14 7*8*13*14 8*9*13*14 9*10*13*14 10*11*13*14 6*7*9*17
1*2*4*16 1*3*4*16 3*4*6*16 3*5*6*16 5*6*8*16 5*7*8*16 7*8*10*16
7*9*10*16 9*10*12*16 9*11*12*16 1*2*13*16 11*12*14*16 2*13*14*16
11*13*14*16 2*4*15*16 4*6*15*16 6*8*15*16 8*10*15*16 10*12*15*16
2*14*15*16 12*14*15*16 2*3*5*17 2*4*5*17 4*5*7*17 4*6*7*17
6*8*9*17 8*9*11*17 8*10*11*17 10*11*13*17 1*12*13*17 10*12*13*17
1*3*14*17 2*3*14*17 1*12*14*17 2*4*15*17 4*6*15*17 6*8*15*17
8*10*15*17 10*12*15*17 2*14*15*17 12*14*15*17 1*3*16*17
5*7*16*17 7*9*16*17 9*11*16*17 1*13*16*17 11*13*16*17 3*5*16*17
```

Chapter 3

Enumeration of atomic lattices

We are interested in studying properties of ideals generated by finitely many monomials. Any minimal generating set $M = \{m_1, \ldots, m_n\}$ of monomials span an atomic semilattice, using lcm for join. This semilattice is finite, and thus complete, and thus may be transformed into a lattice by including the element $1_k = \mathbb{0}$ in the lattice. Any finite antichain of monomials will generate a lattice in this way; a finite atomic lattice.

Any finite atomic lattice L, on the other hand, is the lattice associated to some minimal generating set of monomials. This may be seen by considering the mapping $L \to B_{|L|}$ that sends $l \mapsto F_l$ where $F_l = \{x \mid l \not\leq x\}$. The image under this mapping is a subsemilattice to $B_{|L|}$ isomorphic to L – indeed, it is a mapping between sets of equal sizes, and $F_{x \vee y} = F_x \cup F_y$. There is a bijection between subsets of a fixed totally ordered set $A = \{a_1, a_2, \ldots, a_n\}$ and squarefree monomials, given by $S \mapsto \prod_{i=1}^{r} x^{\varepsilon_i}$ where ε_i is the characteristic function of $S \subseteq A$, i.e. $\varepsilon_i = 1$ if $a_i \in S$ and $\varepsilon_i = 0$ else.

A set of squarefree monomials in $k[x_1, \ldots, x_r]$ corresponds to a subset of B_r by sending variables to atoms, and products of variables to joins of corresponding atoms. In this way, we may embed any monomial lattice in the boolean lattice B_r. By listing the elements in B_r corresponding to the elements in a monomial lattice, we may exhibit the monomial lattice as a matrix where each row corresponds to a monomial. Such a matrix is said to be *column-reduced* if no column can be written as the join of other columns of the matrix, *row-reduced* if no row is the join of other rows, and *reduced* if it is both row- and column-reduced. The matrix is said to be *row-full* or *column-full* if the join of any rows, or any columns, is in the set of rows, or columns, of the matrix. It is said to be *full* if it is both row-full and column-full. Given a $\underline{2}$-matrix T, the set of columns is denoted by CT and the set of rows is denoted by RT.

Proposition 3.1. *For a $\underline{2}$-matrix T, $\langle CT \rangle \cong \langle RT \rangle^*$.*

Proof. This proof is due to [Berglund, 2004].

Let $c \in \langle CT \rangle$ be given. Then duality is exhibited by mapping c to the morphism α_c sending r to the intersection of r and c within T. Denote this intersection point with r_c or c_r interchangeably.

It is obvious that $r_{c \vee c'} = r_c \vee r_{c'}$ and that $(r \vee r')_c = r_c \vee r'_c$. Thus α_c is a morphism $\langle RT \rangle \to \underline{2}$ which gives us that $c \mapsto \alpha_c$ is a morphism $\langle CT \rangle \to \langle RT \rangle^*$. Injectivity is obvious. To show surjectivity, let $\alpha \colon \langle RT \rangle \to \underline{2}$ be a morphism. Set $z = \bigvee_{\alpha(r)=0} r \in \langle RT \rangle$. Then $\alpha(r) = 0$ implies $r \leq z$, since $\alpha(z) = \bigvee_{\alpha(r)=0} \alpha(r) = 0$. Hence, for every r with $\alpha(r) = 1$, we can choose a column $c^r \in \langle CT \rangle$ such that $r_{c^r} = 1$ and $z_{c^r} = 0$.

Let $C = \bigvee_{\alpha(r)=1} c^r \in \langle CT \rangle$. Then $\alpha = \alpha_C$. If $\alpha(x) = 0$, then $x \leq z$ and hence $(c^r)_x = x_{c^r} \leq z_{c^r} = 0$ whenever $\alpha(r) = 1$, which implies $\alpha_C(x) = C_x = 0$. If $\alpha(x) = 1$ then $(c^x)_x = x_{c^x} = 1$ and hence $\alpha_C(x) = C_x = 1$. □

Corollary 3.2. *Any full matrix is quadratic.*

Thus, it is enough to list all possible, non-isomorphic $\underline{2}$-matrices to find all atomic lattices. Here, isomorphism is in the meaning of isomorphic spanned lattices. Such isomorphisms are given by reordering CT and RT, and thus by permutations of rows and columns of the original matrix. Furthermore, we can note that every lattice is the span of its join-irreducible elements. Thus, the sets of irreducibles ICT and IRT are enough to characterise the lattice given.

Now, for our representation of a k-atomic lattice L, we note that the atoms are the only join-irreducible elements. Thus, if we let RT be the set of atoms in L, we see that $\langle RT \rangle \cong L$. Elements in CT will be taken from B_k, and thus our problem is reduced to listing irreducible subsets of B_k. Each such subsets induces a k-atomic lattice by dualising.

Alexander Berglund defines in [Berglund, 2003] the *embedding dimensions* r_k of a monomial lattice. It is the size of the largest *irreducible* subset of B_k, i.e. the largest subset K of B_k such that whenever $a \vee b = c$ for elements $a, b, c \in K$ either $a = c$ or $b = c$. He gives a few recursions helpful in calculating the embedding dimensions of a lattice, one of which is $r_n \leq 2r_{n-1} + 1$ [Berglund, 2003, p 34].

Proposition 3.3. $r_1 = 1, \quad r_2 = 2, \quad r_3 = 4, \quad r_4 = 7, \quad r_5 = 13, \quad 24 \leq r_6 \leq 26$ *and* $r_n \leq 2r_{n-1}$.

Proof. The values for r_1 through r_5 are given in [Berglund, 2003, proposition 4.2.6]. By the construction in [Berglund, 2003, proposition 4.2.8], we can conclude that $24 \leq r_6$.

Assume that a maximal irreducible subset $K \subset B_n$ is given. At least one of the atoms will be absent from the subset, since otherwise any other element would be the join of a set of atoms and the set would not be irreducible. By renumbering, we can assume that the element $10 \ldots 0$ is absent. Then the remaining set will

be partitioned into the elements that start with a 1 and the elements that start with 0; say $K = K_0 \sqcup K_1$ (where $\sqcup$ denotes the disjoint union) for K_i the set of elements that start with i in the first position. Then each of these sets is irreducible, and by truncating all elements will give rise to irreducible subsets in B_{n-1}. Such a set is, by definition, not larger than r_{n-1}. Thus $r_n \leq 2r_{n-1}$ and $r_6 \leq 26$ follows immediately. □

At the current state of lattice generation, the largest observed irreducible subset of B_6 has 21 elements.

3.1 Algorithm analysis

As I began working, Berglund had produced a working implementation of the algorithm stated in [Berglund, 2003] for listing atomic lattices given the above correspondence. The algorithm – using "forbidden sets" to avoid duplicating isomorphism checks – was rather imperfect and produces duplicate lattices in spite of the efforts not to.

The actual algorithm uses a kernel that generates new lattices from a given lattice. This is wrapped in a loop that generates child lattices from each lattice in the previously generated layer, and discards lattices isomorphic to those already produced. As the size of lattices decreases by 1 for each layer traversed, isomorphism within a layer is all that's needed to check.

More specifically, let L be given with an associated full matrix A such that $\langle RA\rangle \cong L$. Then ICA may be partitioned into equivalence classes such that $l \sim l'$ precisely when $\langle CA \setminus \{l\}\rangle \cong \langle CA \setminus \{l'\}\rangle$. For each such equivalence class, pick one representative l and return the set of all such $CA \setminus \{l\}$. We may note that when $l \in ICA$, we can generate $I\langle CA \setminus \{l\}\rangle$ as the set of irreducible elements within the set $ICA \setminus \{l\} \cup \{l \vee x \mid x \in ICA \setminus \{l\}\}$. Thus, we can keep the representation of lattices as join-irreducible matrices throughout our calculations.

The single most expensive part of the algorithm is the problem of isomorphism checking. Isomorphism of lattices forms the basis for generation of equivalence classes of child lattices, as well as the checks needed at adjoining generated child lattices to the list of all atomic lattices.

Berglund used the forbidden sets to try and minimise the number of isomorphism checks performed. Instead of those, I have chosen to store the generated lattices in a hash table, with a signature of the irreducible matrix as hash, namely in the current version a list of two lists, one containing the number of 1:s in each row, and one containing a similar sum for each column. Use of this signature speeds up detection of non-isomorphic lattice pairs considerably; but for better performance, finding a better signature would be paramount.

3.2 Combinatorial explosion of lattices

The problem of listing atomic lattices suffers quite a bit under combinatorial explosion. There is only one lattice on a single atom; and only one lattice on two atoms. There are four different lattices on three atoms. On four atoms, 50 different lattices occur; and on five atoms, that same number is 7443. All these may be generated within a few days on a modern computer, using MIT-Scheme and our implementation of the listing algorithm.

For the case of six atoms though, the case is worse. Since I started listing lattices in the autumn of 2003, I have had a process running continuously on my workstation[1]. On November 7, 2003, I had to do certain rearrangements of the way lattices were stored, moving more of the responsibility to keep track of data out into the file system. At that point, I had generated the first 139605 lattices on 6 atoms, and could no longer hold all data needed in memory within the MIT Scheme memory allocation. In the middle of December, 2003, I had to do a reallocation scheme again as the mere lists of all 6-atomic lattices of the same size grew too large for the memory allocation, and switch to a list of "coordinates" for the lattices generated.

Due to the way that the lattice generation algorithm works the k-atomic lattices are generated in layers such that all lattices in each layer have the same total number of elements. Thus, the generating program works its way through layer by layer until the smallest possible lattice is produced – i.e. the one with $k+2$ elements: the k atoms and $\mathbb{1}$ and $\mathbb{0}$. Since B_k contains 2^k elements, this leads to the conclusion that there are a total of $2^k - k - 2$ layers of k-atomic lattices. Thus for the current run, we observe that we will end up generating a grand total of 56 layers. For four atoms, the corresponding figure is 10 and for five atoms 25.

The relative sizes of the layers of 5-atomic lattices coincide roughly with a bell curve distributing 7443 elements with a mean of 15.3484 and a standard deviation of 3.0745. See figure 3.1. Similarily, the sizes of the layers thus far calculated from the 6-atomic lattices indicate that a bell curve could be fitted. It would then imply that we have a total of about 30 million lattices, see figure 3.2 and 3.3 for comparisons with the thus indicated bell curve[2]. These sketches are mainly viewed as a tool to guess order of magnitude of the lattice layers – I have not done the kind of in-depth analysis needed to present a wellfounded argument for the size distribution of k-atomic lattices, nor have I done much work on finding the asymptotical behaviour. It would not at all be surprising to find between 10 and 50 million lattices, but finding only 3 million, or all of 300 million would be quite surprising.

As of the printing of this thesis, I have a total of 1 685 563 generated lattices of 43 to 64 elements each.

[1] 2GHz ix86 running Redhat and with a single active user
[2] generated with a mean of 30 and a standard deviation of 5

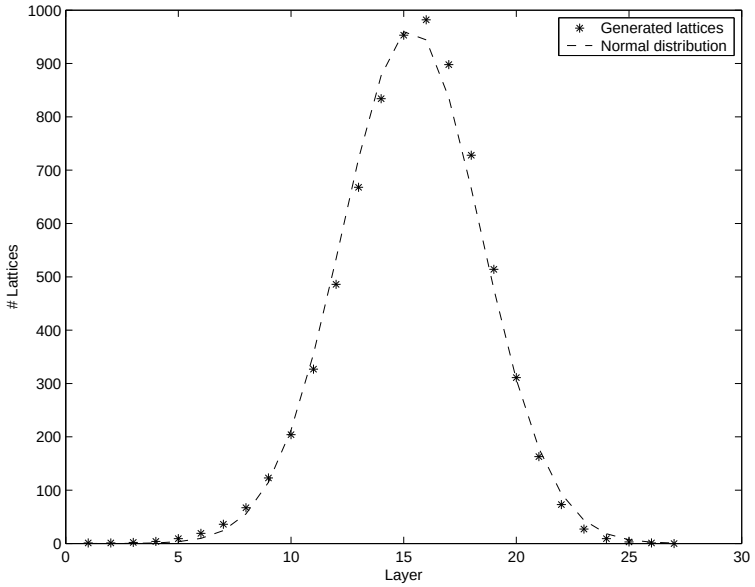

Figure 3.1: 5-atomic lattices and a roughly fitting bell curve

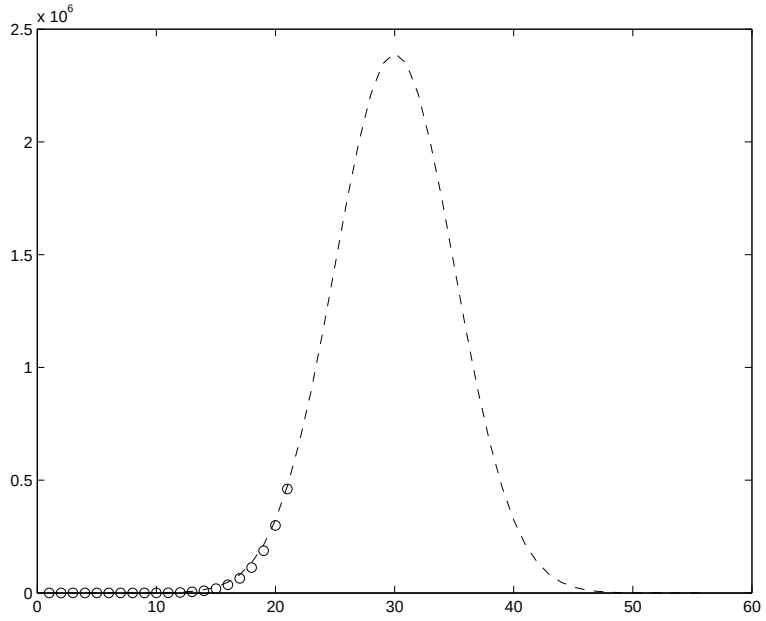

Figure 3.2: 6-atomic lattices and the suggested fitting bell curve

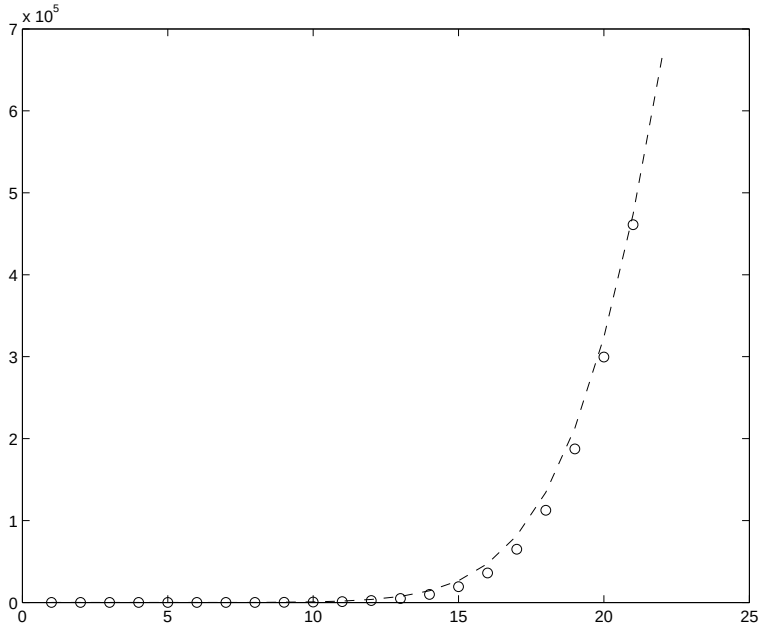

Figure 3.3: Closeup picture on the generated parts of figure 3.2

Appendix A

Users guide

A.1 Obtaining the program

The program is provided available for download as a source package from Stockholm University at `http://www.math.su.se/~mik/poincare.html`.

A.2 Build and install

To build the program, unpack the source distribution. It will create a directory of its own. Within that directory, edit the `Makefile` to ensure that the variables `INCLUDEDIR` and `LIBDIR` point to correct directories. This is possibly redundant, somewhat dependent on your computing environment. Once the `Makefile` is sane, run the command `make` within that directory to produce the compiled program.

To install the program, simply copy the produced `poincare` executable file to a directory that lies within your `PATH`. Examples of locations commonly default set in the `PATH` are `/bin`, `/usr/bin`, `/usr/local/bin` and `~/bin`.

A.3 Starting the program

If your system only has the correct version of the Pari library[1] installed, then the program should have no significant problems running. Otherwise, you may want to edit the included `poincare.sh` wrapper script to reflect the correct file path for `poincare` and put `poincare.sh` within the executable path.

[1] the program relies on Pari v2.2.x

The program is started with either `poincare` or `poincare.sh`, depending on whether you needed to install the wrapper script or not.

-v	Verbose – the program will print a progress indicator while calculating denominator polynomials and will report on the monomials/simplices recognised during input.
-d	Debug – the program will print an insane amount of debugging information while calculating.
-h	Help – print out a summary of valid command line flags.
-p *primes*	Tells the Pari library to precalculate *primes* primes and store in a lookuptable for quick access. Default value is 10000000.
-s *stacksize*	Tells the Pari library to allocate a stack of *stacksize* bytes. Default value is 10000000, or just over 9.6 MBytes.

Table A.1: Command line flags

A.4 Calculation of Poincaré-Betti series

To enter and calculate the denominator polynomial of the Poincaré-Betti series of a monomial ring, you will want to use the commands `add monomial` and `denominator`. First, if you are already in a session where you have calculated with monomial ideals, you may want to run `clear` to reset the program to avoid interference from the earlier calculations. Then enter a list of monomials on a line that starts with the command `add monomial`, and make sure that your polynomials have each multiplication written out using the `*` sign. Finally, once all monomials are entered, perform the calculation with `denominator`. This may take a while to finish.

There are examples where the results depend on the chosen field characteristic. One such example, the real projective plane, is small enough for the complexity issues of the current implementation to not play an all too large role. For the projective plane, the denominator polynomial depends on whether the characteristic is equal to 2 or not. The field characteristic can be set with the `char` command.

A.5 Calculation of simplicial homology

For calculation of simplicial homology, you need to enter the simplices with `add simplex`, similar to `add monomial` for monomial ideals. Again, the command `clear` resets the internal storage for simplicial complexes and for monomial ideals so that you can start from scratch. Each call to `add simplex` will only accept squarefree monomial expressions – i.e. variable names delimited by `*` – with the expressions separated by whitespace. The program will internally list all subsimplices of a given simplex and include all of them, so it is only necessary to enter the maximal faces.

Example A.1 A simple monomial ideal calculation

```
% poincare
Welcome to the Poincaré calculator. You can use this program to
calculate simplicial homology over prime fields and to calculate
the denominator polynomial of the Poincaré-Betti series of monomial
rings.
(c) 2004 Mikael Johansson
This program is released under the MIT License

> add monomial a^3 a*b^2 b^3 b*c^2 c^3 c*a^2
> denominator
1 - a^3*ZZ^2 - a*b^2*ZZ^2 - b^3*ZZ^2 - a^2*c*ZZ^2 - b*c^2*ZZ^2 -
c^3*ZZ^2 - a^3*b^2*ZZ^3 - a*b^3*ZZ^3 - a^3*c*ZZ^3 - a^2*b^2*c*ZZ^3 -
a^2*b*c^2*ZZ^3 - a*b^2*c^2*ZZ^3 - b^3*c^2*ZZ^3 - a^2*c^3*ZZ^3 -
b*c^3*ZZ^3 - a^3*b^2*c*ZZ^4 - a^2*b^2*c^2*ZZ^4 - a*b^3*c^2*ZZ^4 -
a^2*b*c^3*ZZ^4

> quit

Thanks for visiting.
```

Homology calculations are initiated by the `homology` command. Again, since homology may depend on the field characteristic, the `char` command can be used to capture such variations. Furthermore, please observe that as all calculations take place with field coefficients, torsion over $\mathbb{Z}$ may only be discovered by calculating homology over the proper finite field.

Please note that, when duplicating the examples here given, each commands need to be given on a single line. Should you want to break up the entry of simplicial complexes or for that matter of monomial ideals on several lines, make sure each line starts with the relevant command – i.e. either `add monomial` or `add simplex`.

A.6 List of commands

add monomial `add monomial` takes a space separated list of monomial expressions, where each monomial expression is a `*` separated list of variable strings with an optional `^` and integer exponent postponed. The program will upon reading this command parse the given list of monomials into a squarefree internal storage form using the Weyman-Fröberg method, and add the monomials with valid syntax to the internal monomial ideal. Note that any string not containing any of the characters[2] `␣+*-/^,.` is a valid variable string. Thus `add monomial '*=*'` would be a valid command.

[2] where ␣ denotes a space

Example A.2 The denominator polynomial for the Stanley-Reisner ring of the 6-point triangulation of $\mathbb{RP}^2$

```
% poincare
Welcome to the Poincaré calculator. You can use this program to
calculate simplicial homology over prime fields and to calculate
the denominator polynomial of the Poincaré-Betti series of monomial
rings.
(c) 2004 Mikael Johansson
This program is released under the MIT License

> add monomial a*b*c a*b*e a*c*f a*d*e a*d*f
> add monomial b*c*d b*d*f b*e*f c*d*e c*e*f
> denominator
1 - a*b*c*ZZ^2 - a*b*e*ZZ^2 - a*c*f*ZZ^2 - b*e*f*ZZ^2 -
c*e*f*ZZ^2 - b*c*d*ZZ^2 - a*d*e*ZZ^2 - c*d*e*ZZ^2 - a*d*f*ZZ^2 -
b*d*f*ZZ^2 - a*b*c*e*ZZ^3 - a*b*c*f*ZZ^3 - a*b*e*f*ZZ^3 -
a*c*e*f*ZZ^3 - b*c*e*f*ZZ^3 - a*b*c*d*ZZ^3 - a*b*d*e*ZZ^3 -
a*c*d*e*ZZ^3 - b*c*d*e*ZZ^3 - a*b*d*f*ZZ^3 - a*c*d*f*ZZ^3 -
b*c*d*f*ZZ^3 - a*d*e*f*ZZ^3 - b*d*e*f*ZZ^3 - c*d*e*f*ZZ^3 -
a*b*c*e*f*ZZ^4 - a*b*c*d*e*ZZ^4 - a*b*c*d*f*ZZ^4 - a*b*d*e*f*ZZ^4 -
a*c*d*e*f*ZZ^4 - b*c*d*e*f*ZZ^4
> char 2
New characteristic: 2
> denominator
1 - a*b*c*ZZ^2 - a*b*e*ZZ^2 - a*c*f*ZZ^2 - b*e*f*ZZ^2 -
c*e*f*ZZ^2 - b*c*d*ZZ^2 - a*d*e*ZZ^2 - c*d*e*ZZ^2 - a*d*f*ZZ^2 -
b*d*f*ZZ^2 - a*b*c*e*ZZ^3 - a*b*c*f*ZZ^3 - a*b*e*f*ZZ^3 -
a*c*e*f*ZZ^3 - b*c*e*f*ZZ^3 - a*b*c*d*ZZ^3 - a*b*d*e*ZZ^3 -
a*c*d*e*ZZ^3 - b*c*d*e*ZZ^3 - a*b*d*f*ZZ^3 - a*c*d*f*ZZ^3 -
b*c*d*f*ZZ^3 - a*d*e*f*ZZ^3 - b*d*e*f*ZZ^3 - c*d*e*f*ZZ^3 -
a*b*c*e*f*ZZ^4 - a*b*c*d*e*ZZ^4 - a*b*c*d*f*ZZ^4 - a*b*d*e*f*ZZ^4 -
a*c*d*e*f*ZZ^4 - b*c*d*e*f*ZZ^4 - a*b*c*d*e*f*ZZ^4 - a*b*c*d*e*f*ZZ^5
> quit

Thanks for visiting.
```

add simplex As with `add monomial`, the command `add simplex` takes a space separated list of monomial expressions. The main difference is that in `add simplex`, use of the integer exponent is forbidden. Similar to `add monomial`, this command takes the space separated list of thus formed squarefree monomial expressions and inserts the corresponding abstract simplices into the internally represented simplicial complex as well as all subsimplices of the simplex entered.

char `char` sets the characteristic of the field from which the coefficients for homology calculations is taken. Valid values are 0 and any prime p, and the default is 0.

clear `clear` will clear out all internally stored data – i.e. the monomial ideal, the simplicial complex and the lookuptable of stored variables. Apart from the characteristic setting, this returns the program to the state it starts up in.

denominator `denominator` will start the calculation of the denominator polynomial for the monomial ideal entered thus far. Beware that for large examples, this may take quite some time as well as use up large amounts of memory. If the program is run with the `-v` flag, it will report on the number of latticepoints spanned by the monomial generators and will run a progress meter printing a row of points in 10 point intervals, 50 to a line and with a numeric counter every 500 points. The point is printed before the calculations at that particular point take place. The output is a multigraded Poincaré-Betti series denominator in a format that will allow you to input it by copy-and-paste to most mainstream computer algebra packages.

help `help` will return a list of available commands together with a summary of their function. By calling `help <command>` for various commands, a more detailed description of calling syntax and function will be delivered.

homology `homology` will start the calculation of the Hilbert series of the homology groups of the simplicial complex currently entered. For very large examples, this can potentially be time- and memory-consuming.

print `print` will output a large information dump printing out all monomials in the monomial lattice and all simplices in the simplicial complex, one on each line, in a binary representation induced by the internal storage of the monomial expressions entered. Furthermore, it will print the sizes of the simplicial complex and the monomial lattice and the current characteristic. Beware that for even modestly large examples this function will print a *lot* of data.

quit Exits the program. Synonyms include `exit`, `q`, `Q`, `bye`, `stop` and `STOP`. Sending an End-Of-File character (`^D`) will also work.

var When outputting Hilbert polynomials and Poincaré-Betti denominator polynomials, one variable string is reserved for the non-squarefree variable. Default is `ZZ`, and by calling `var <string>` the variable string can be changed to any other string.

A.7 Known quirks and problems

There are several things that could work better with the software. As always, better interaction with the surrounding world would be nice.

The program as it is now has several complexity issues outlined earlier. Most notable is the excessive memory consumption. Reasonably large computations generate so large complexes that a single differential will outgrow the allocated Pari stack. This problem may be remedied in the future by constructing a linear algebra package streamlined for this particular application, or by finding a more suitable linear package written.

Furthermore, the calculations are very processor intensive. Analysis of other formulations of the formula for calculating the denominator polynomial as well as an analysis of the possibility to find homology of subcomplexes of a given complex without recalculating everything each time would be valuable.

Appendix B

Programmers reference

B.0.1 Introduction

The `poincare` calculator implements a formula given by Alexander Berglund in 2004 to find the denominator polynomial $b(\mathbf{x}, t)$ to the Poincaré-Betti series $P_k^R(\mathbf{x}, t) = \frac{\Pi(1+x_i t)}{b(\mathbf{x},t)}$

These references try to provide the reader with adequate information about the design and programming of the poincare calculator, in order to enable interested parties to adapt the source code – or other source code – to take advantage of the contents. In particular, emphasis lies on giving additional information about the various classes, members and functions defined in the source code package.

B.0.2 Installation

To build and use the program, position yourself in the directory containing the source distribution. Make sure you have Pari v2.2.x as well as Readline, GMP and libstdc++ installed (the compilation step will tell you if you don't). Then run `make` to start the build.

If you have several versions of Pari installed, you will need to make sure your `$LD_LIBRARY_PATH` contains the path to the directory where the library files for Pari 2.2.x are stored. Then place the executable `poincare` in a standard binary directory (such as `/usr/bin`, `/usr/local/bin` or `~/bin`) for access regardless of your own location within the directory tree. Should you have problems with setting the correct `$LD_LIBRARY_PATH`, use the included `poincare.sh` and edit it to reflect the storage path of `poincare`.

B.1 Monomial Class Reference

`Monomial` implements monomials/simplices with sufficient framework for the calculations at hand.

`#include <homology.h>`

Public Methods

- **Monomial** ()
  Standard constructor.
- **Monomial** (const Monomial &m)
  Copy constructor.
- **Monomial** (const std::string s)
  Constructor parsing user inputted data.
- **Monomial** (const t_monomial m, unsigned long d=0)
  Constructor setting zdegree to `d`.
- ~**Monomial** ()
  Standard destructor.
- bool **operator==** (const Monomial m) const
  Test for equality of **bitmask** *and* **zdegree**.
- bool **operator<** (const Monomial m) const
  Test for `this.zdegree< m.zdegree` *or* `this.bitmask< m.bitmask` *as GMP integers.*
- Monomial & **operator=** (const Monomial &m)
  Sets `this` *to a copy of the argument.*
- Monomial & **operator** *=* (const unsigned long c)
  Multiply with the variable indexed by `c`.
- Monomial **operator** * (const Monomial &m) const
  lcm of the monomials; with summed `zdegrees`.
- Monomial **gcd** (const Monomial m) const
  Returns a new Monomial with bitmasks anded.
- Monomial **lcm** (const Monomial m) const
  Returns a new Monomial with bitmasks ored.

- void **setgcd** (const Monomial m)

  Sets `this.bitmask` *to the same anded with* `m.bitmask` *(i.e., this.bitmask&=m.bitmask).*

- void **setlcm** (const Monomial m)

  Sets `this.bitmask` *ored with* `m.bitmask`.

- bool **is_relprime** (const Monomial m) const

  Returns true if no variable divides both `this` *and* `m`. *This is equivalent to the bitmasks anded being equal to 0.*

- bool **divides** (const unsigned long idx) const

  Checks whether the bit indexed by `idx` *is set.*

- unsigned long **next_index** (const unsigned long idx) const

  Returns next index after `idx` *with bit set.*

- long int **dim** () const

  Returns `degree()-1`, *which represents topological dimension of an abstract simplex with* **degree()** (p. 41) *elements.*

- unsigned long int **degree** () const

  Returns number of set bits, which is equivalent to the algebraic degree of a corresponding monomial.

- unsigned long int **nvars** () const

  Returns number of allocated string->index relations from **variable_index**.

- std::string **tostring** (const bool do_add_multiply=false) const

  Prettyprinting of the Monomial.

- void **getrepresentation** (t_monomial &m) const

  Allocates memory and returns a copy of the raw bitmask.

Static Public Methods

- unsigned long **get_variable_index** (std::string s)

  Returns `i` *such that* **variable_index**`[string]=i`.

- std::string **get_variable_name** (unsigned long l)

  Returns `string` *such that* **variable_index**`[string]=i`.

- void **clear** ()

  Clears **variable_index**.

Public Attributes

- long **zdegree**

  Support for a free, non-squarefree variable.

Protected Attributes

- t_monomial **bitmask**

  The raw bitstring - a GMP integer.

Static Protected Attributes

- varmapT **variable_index**

  A global lookuptable for user inputted variable names.

B.1.1 Detailed Description

`Monomial` implements monomials/simplices with sufficient framework for the calculations at hand.

Monomial implements squarefree monomials as bitstrings. The class may be used to represent squarefree monomials, or subsets of a set, or abstract simplices.

Internal representation is a GMP integer (accessible through the mpz_* family of functions) together with a global lookuptable for mapping variable strings given by the user to bit indices in the GMP integer, thus using it as a variable sized bitmap.

Furthermore, the member **zdegree** captures a single non-squarefree variable available, for instance for formation of multigraded Hilbert-series and similar exercises.

Definition at line 99 of file homology.h.

B.1.2 Constructor & Destructor Documentation

B.1.2.1 Monomial::Monomial (const std::string *s*)

Constructor parsing user inputted data.

Parameters:

s The code expects this to be a set of variable expressions, separated by '*', where each variable expression is a string avoiding whitespace and '+*^/-' followed by optionally ^ and an integer If the power expression (^ and integer) is present, the code performs Weyman-Fröberg automatically, i.e. allocates several indices for the same variable string.

Definition at line 114 of file homology.c.

References bitmask, get_variable_index(), Tokenize(), and zdegree.

B.1.3 Member Function Documentation

B.1.3.1 unsigned long Monomial::get_variable_index (std::string *s*) [static]

Returns `i` such that **variable_index**`[string]=i`.

Given a string (supposedly a variable), return an index for representation in monomials. If the variable is seen for the first time, allocate a new index.

Definition at line 237 of file homology.c.

References z_var_str().

Referenced by Monomial().

B.1.3.2 std::string Monomial::get_variable_name (unsigned long *l*) [static]

Returns `string` such that **variable_index**`[string]=i`.

Given an index, return the variable name allocated there, removing the Weyman-Fröberg distinguishing strings. If the variable is unknown, return the empty string.

Definition at line 255 of file homology.c.

References variable_index, and z_var_str().

Referenced by tostring().

B.1.3.3 void Monomial::getrepresentation (t_monomial & *m*) const

Allocates memory and returns a copy of the raw bitmask.

WARNING: Use with caution. Free memory on your own. Allocates memory and then fills the argument t_monomial with the raw representation data in the internals of the class.

Parameters:

m The GMP integer to be allocated and filled with content.

Definition at line 370 of file homology.c.

References bitmask.

Referenced by SimplicialComplex::add_simplex(), operator<<(), and MonomialIdeal::simplicialseries().

B.1.3.4 Monomial Monomial::operator * (const Monomial & *m*) const

lcm of the monomials; with summed `zdegrees`.

For "proper" multiplications in squarefree rings, check for relative primality first – the product of two monomials in $Q[t]/(x_1^2, \ldots, x_r^2)$ will vanish precisely if there is a nontrivial common divisor to the two monomials. Thus the product will only survive if `this.is_relprime(m)`

Definition at line 225 of file homology.c.

References bitmask, and zdegree.

B.1.3.5 Monomial & Monomial::operator *= (const unsigned long *c*)

Multiply with the variable indexed by `c`.

In practice this sets the bit indexed by `c`.

Definition at line 211 of file homology.c.

References bitmask, and zdegree.

B.1.3.6 std::string Monomial::tostring (const bool *do_add_multiply* = false) const

Prettyprinting of the Monomial.

Parameters:

do_add_multiply If true, a '∗' character is prepended for easy printing of coefficients.

Definition at line 338 of file homology.c.

References divides(), get_variable_name(), nvars(), z_var_str(), and zdegree.

The documentation for this class was generated from the following files:

- **homology.h**
- **homology.c**

B.2 MonomialIdeal Class Reference

Implements monomial ideals and handles the generator lcm lattice, gcd graph and wrappers for calculating the Poincaré-Betti series denominator polynomial.

`#include <homology.h>`

Public Methods

- **MonomialIdeal** ()
  Default constructor.
- **MonomialIdeal** (const MonomialIdeal &mi)
  Copy constructor.
- **~MonomialIdeal** ()
  Default destructor.
- MonomialIdeal & **operator+=** (const **Monomial** m)
  Add a new generator to the **Monomial** (p. 40) *unless it already is present.*
- void **clear** ()
  Reset the object completely.
- void **print** () const
  Output a mainly debugging-friendly infodump of the ideal to std::cout.
- **Squarefree denominator** () const
  Calculates the denominator polynomial and returns it as a **Squarefree** (p. 53).

Protected Methods

- **Squarefree denominatorfactor** (const **Monomial** latticepoint) const
  Wrapper function between denominator and simplicialseries.
- **Squarefree simplicialseries** (const **Monomial** &latticepoint) const
  Calculates the proper index shifted Hilbert series for $p_\alpha(\mathbf{x}, t)$ as described in Berglund, 2004.
- bool **is_connected** (const std::vector< **Monomial** > &M, const mpz_t &setmask) const
  Returns true if the subgraph indexed by the bitmask of the GMP integer in the argument is connected as a subgraph of the GCD graph.

Protected Attributes

- std::set< **Monomial** > **generators**
  Monomial (p. 40) *generators for the LCM lattice.*
- std::set< **Monomial** > **lattice**
  The LCM lattice.

B.2.1 Detailed Description

Implements monomial ideals and handles the generator lcm lattice, gcd graph and wrappers for calculating the Poincaré-Betti series denominator polynomial.

Definition at line 170 of file homology.h.

B.2.2 Member Function Documentation

B.2.2.1 Squarefree MonomialIdeal::denominator () const

Calculates the denominator polynomial and returns it as a **Squarefree** (p. 53).

the degree of the lcm of all monomial generators. By Alexander Berglunds results, $p(t) = \prod_{l \in L}(1 - lp_{M_l}(t))$ where L is the monomial lattice and $p_{M_l}(t)$ is calculated in **MonomialIdeal::simplicialseries** (p. 46)

Definition at line 435 of file homology.c.

References denominatorfactor(), generators, and lattice.

Referenced by cmd_den().

B.2.2.2 Squarefree MonomialIdeal::denominatorfactor (const Monomial *latticepoint*) const [protected]

Wrapper function between denominator and simplicialseries.

Returns $(1 - lp_{M_l}(t)$ for the latticepoint l

Parameters:
latticepoint Corresponds to l in $(1 - lp_l(t))$.

Definition at line 465 of file homology.c.

References simplicialseries().

Referenced by denominator().

B.2.2.3 bool MonomialIdeal::is_connected (const std::vector< Monomial > & *M*, const mpz_t & *setmask*) const [protected]

Returns true if the subgraph indexed by the bitmask of the GMP integer in the argument is connected as a subgraph of the GCD graph.

Parameters:

M A list of Monomials.

setmask An GMP integer such that the bitmask has a 1 at the indices of `M` that indicate Monomials included in the set.

Definition at line 577 of file homology.c.

Referenced by simplicialseries().

The documentation for this class was generated from the following files:

- **homology.h**
- **homology.c**

B.3 SimplicialComplex Class Reference

Implements handling of simplicial complexes and calculation of their homology.

`#include <homology.h>`

Public Methods

- **SimplicialComplex** ()
  Default constructor.
- **SimplicialComplex** (const SimplicialComplex &sc)
  Copy constructor.
- **~SimplicialComplex** ()
  Default destructor.
- void **add_simplex** (**Monomial** m, bool subs=true)
  Adds a simplex to the simplicial complex.
- void **add_simplex** (t_simplex s, bool subs=true)
  Adds a simplex to the simplicial complex.
- std::vector< long > **homology_ranks** (unsigned long **chr**)
  Calculates the sequence of homology ranks of the current simplicial complex.
- bool **is_empty** () const
  Returns true if the simplicial complex is empty.
- void **clear** ()
  Resets the simplicial complex, erasing all stored content.
- void **print** () const
  Prints a debugging dump to std::cout.

Protected Attributes

- std::vector< unsigned long > **kerrank**
  Caching of kernel ranks.
- std::vector< unsigned long > **imrank**
  Caching of image ranks.

- std::set< **Monomial** > **simplices**

  The set of simplices in the complex.

- unsigned long **calculated**

  Holds the characteristic for which the results are cached.

B.3.1 Detailed Description

Implements handling of simplicial complexes and calculation of their homology.

Internally, the results are cached by storing the calculated image and kernel ranks in **imrank** and **kerrank** and the characteristic for which homology was calculated in **calculated**. Upon modification of the simplices or upon calculating homology for another characteristic, **kerrank** and **imrank** are recalculated, but when no such modifications have been made, the stored result is returned.

Definition at line 216 of file homology.h.

B.3.2 Member Function Documentation

B.3.2.1 void SimplicialComplex::add_simplex (t_simplex *s*, bool *subs* = true)

Adds a simplex to the simplicial complex.

Parameters:

s The simplex to be added.

subs If false, then no subsimplices are added. The responsibility to add all relevant subsets is delegated to the programmer.

Definition at line 638 of file homology.c.

References calculated, and simplices.

B.3.2.2 void SimplicialComplex::add_simplex (Monomial *m*, bool *subs* = true)

Adds a simplex to the simplicial complex.

Parameters:

m The simplex to be added, encapsulated in a **Monomial** (p. 40) object. Calls `add_simplex` with the internal GMP integer, and cleans up afterwards.

subs If false, then no subsimplices are added. The responsibility to add all relevant subsets is delegated to the programmer.

Definition at line 630 of file homology.c.

References calculated, and Monomial::getrepresentation().

Referenced by clear(), cmd_add(), SimplicialComplex(), and MonomialIdeal::simplicialseries().

B.3.2.3 std::vector< long > SimplicialComplex::homology_ranks (unsigned long *chr*)

Calculates the sequence of homology ranks of the current simplicial complex.

Takes the characteristic to calculate over, and returns the successive ranks of the homology modules as a vector of type int. Please note that the ranks will be returned with an index shift – $\dim_k \tilde{H}_i(\Delta)$ is given in `homology_ranks(chr)[i+1]`

Definition at line 676 of file homology.c.

References calculated, chr, imrank, kerrank, MAX, and simplices.

Referenced by cmd_hom(), and MonomialIdeal::simplicialseries().

B.3.3 Member Data Documentation

B.3.3.1 unsigned long SimplicialComplex::calculated [protected]

Holds the characteristic for which the results are cached.

Within the implementation in **homology.c**, **calculated** takes on the special value of 1 to signify the non-calculated state, since 1 never is a valid characteristic.

Definition at line 226 of file homology.h.

Referenced by add_simplex(), clear(), homology_ranks(), and SimplicialComplex().

The documentation for this class was generated from the following files:

- **homology.h**
- **homology.c**

B.4 Squarefree Class Reference

Implements a polynomial class modulo the ideal generated by all variable squares.

`#include <homology.h>`

Public Methods

- **Squarefree** ()

  Default constructor.

- **Squarefree** (const Squarefree &p)

  Copy constructor.

- **Squarefree** (const **Monomial** &m)

  Initiates the polynomial 1∗m.

- **~Squarefree** ()

  Default destructor.

- bool **operator==** (const Squarefree p) const

  Test for `this.monomials==p.monomials`.

- Squarefree & **operator=** (const Squarefree &p)

  Set `this.monomials=p.monomials`.

- Squarefree & **operator+=** (const std::pair< **Monomial**, long > &m)

  Set `this.monomials[m.first]+=m.second`.

- Squarefree & **operator+=** (const Squarefree &p)

  Performs termwise addition of `p` *to* `this`.

- Squarefree & **operator** ∗= (const Squarefree &p)

  Performs a basic multiplication algorithm of `p` *onto* `this`.

- Squarefree & **operator** ∗= (const long l)

  Multiplies each coefficient in `this` *by* `l`.

- const std::map< **Monomial**, long > **getmonomials** () const

  Returns the map **monomials** *containing the coefficients of the polynomial.*

Protected Methods

- void **clearzeroes** ()

  Prunes off unnecessary zero monomials from the map.

Protected Attributes

- std::map< **Monomial**, long > **monomials**

  Stores the coefficients of the terms of the polynomial as a map from **Monomial** *to* `long`.

B.4.1 Detailed Description

Implements a polynomial class modulo the ideal generated by all variable squares.

Be cautioned that this means that the only Monomials to survive a multiplication are those that are relatively prime.

Definition at line 271 of file homology.h.

B.4.2 Member Data Documentation

B.4.2.1 std::map<Monomial,long> Squarefree::monomials [protected]

Stores the coefficients of the terms of the polynomial as a map from **Monomial** to `long`.

Thus the coefficients of a **Monomial** (p. 40) m in the polynomial may be accessed using **monomials**`[m]`

Definition at line 276 of file homology.h.

Referenced by clearzeroes(), getmonomials(), operator *=(), operator+=(), operator=(), operator==(), and Squarefree().

The documentation for this class was generated from the following files:

- **homology.h**
- **homology.c**

B.5 homology.c File Reference

Contains code implementing the definitions given in **homology.h**.

```
#include <cstdlib>

#include <ctype.h>

#include <climits>

#include <sstream>

#include <iostream>

#include <stdexcept>

#include <deque>

#include "homology.h"

#include "main.h"
```

Functions

- std::string **z_var_str** ("ZZ")

  The variable containing the output representation of the free variable in power series and polynomials.

- void **Tokenize** (const std::string &str, std::vector< std::string > &tokens, const std::string &delimiters)

  Helper function to split a string into a vector of substrings across a set of delimiter characters.

- std::ostream & **operator**<< (std::ostream &in, **Monomial** const &m)

  Overloaded ostream operator<< preamble for stream printing of a **Monomial**.

- std::ostream & **operator**<< (std::ostream &in, **Squarefree** const &p)

  Overloaded ostream operator<< preamble for stream printing of a **Squarefree**.

Variables

- unsigned long **chr** = 0

  Global variable controlling the field characteristic.

B.5.1 Detailed Description

Contains code implementing the definitions given in **homology.h**.

More precisely, this file contains tools, functions and handling of:

- Simplicial complexes
- Squarefree monomials
- Homology calculations

Definition in file **homology.c**.

B.5.2 Function Documentation

B.5.2.1 std::ostream& operator<< (std::ostream & *in*, Monomial const & *m*)

Overloaded ostream operator<< preamble for stream printing of a **Monomial**.

Outputs the internal representation of the monomial as one bitstring with the state, followed by space and the zdegree field.

Todo:
Finding faces of simplices/generators of lattices and output only those, possibly in more legible form

Definition at line 196 of file homology.c.

B.5.2.2 void Tokenize (const std::string & *str*, std::vector< std::string > & *tokens*, const std::string & *delimiters*)

Helper function to split a string into a vector of substrings across a set of delimiter characters.

This function was taken from

`http://www.faqs.org/docs/Linux-HOWTO/C++Programming-HOWTO.html`

Parameters:
str The string to tokenize

tokens A std::vector of tokens

delimiters A string of characters, at which the boundary of tokens lie.

Definition at line 67 of file homology.c.

Referenced by Monomial::Monomial(), and parse_eval().

B.6 homology.h File Reference

Contains declarations and definitions used for homology calculations.

```
#include <pari.h>

#include <gmp.h>

#include <vector>

#include <set>

#include <map>

#include <string>

#include <iostream>
```

Compounds

- class **Monomial**

  `Monomial` *implements monomials/simplices with sufficient framework for the calculations at hand.*

- class **MonomialIdeal**

  Implements monomial ideals and handles the generator lcm lattice, gcd graph and wrappers for calculating the Poincaré-Betti series denominator polynomial.

- class **SimplicialComplex**

  Implements handling of simplicial complexes and calculation of their homology.

- class **Squarefree**

  Implements a polynomial class modulo the ideal generated by all variable squares.

Defines

- #define **MAX**(a, b) (((a)<(b))?(b):(a))

  Returns the maximum of the two elements. Evaluates each element twice.

- #define **MIN**(a, b) (((a)>(b))?(b):(a))

  Returns the minimum of the two elements. Evaluates each element twice.

Functions

- void **Tokenize** (const std::string &str, std::vector< std::string > &tokens, const std::string &delimiters="")

  Helper function to split a string into a vector of substrings across a set of delimiter characters.

- std::ostream & **operator<<** (std::ostream &in, **Monomial** const &m)

  Overloaded ostream operator<< preamble for stream printing of a **Monomial**.

- std::ostream & **operator<<** (std::ostream &in, **Squarefree** const &p)

  Overloaded ostream operator<< preamble for stream printing of a **Squarefree**.

Variables

- std::string **z_var_str**

  A string containing the output form of the free variable of the power series.

- unsigned long **stacktop**

  Handling of memory management with Pari.

- unsigned long **chr**

  Global variable controlling the field characteristic.

B.6.1 Detailed Description

Contains declarations and definitions used for homology calculations.

Defines datatypes for manipulations on monomials and polynomials in $k[x_1, ..., x_r, z]/(x_1^2, x_2^2, ..., x_r^2)$, for calculation of denominator polynomials for the Poincaré-Betti series, for simplicial homology calculations.

Depends on: PARI, GMP, standard libraries

Definition in file **homology.h**.

B.6.2 Function Documentation

B.6.2.1 std::ostream& operator<< (std::ostream & *in*, Monomial const & *m*)

Overloaded ostream operator<< preamble for stream printing of a **Monomial**.

Outputs the internal representation of the monomial as one bitstring with the state, followed by space and the zdegree field.

Todo:
Finding faces of simplices/generators of lattices and output only those, possibly in more legible form

Definition at line 196 of file homology.c.

References Monomial::getrepresentation(), Monomial::nvars(), and Monomial::zdegree.

B.6.2.2 void Tokenize (const std::string & *str*, std::vector< std::string > & *tokens*, const std::string & *delimiters*)

Helper function to split a string into a vector of substrings across a set of delimiter characters.

This function was taken from

`http://www.faqs.org/docs/Linux-HOWTO/C++Programming-HOWTO.html`

Parameters:
str The string to tokenize

tokens A std::vector of tokens

delimiters A string of characters, at which the boundary of tokens lie.

Definition at line 67 of file homology.c.

Referenced by Monomial::Monomial(), and parse_eval().

B.7 main.c File Reference

Main program file - herein is contained the actual executable code.

```
#include <iostream>

#include <cstdlib>

#include <sys/types.h>

#include <sys/errno.h>

#include <unistd.h>

#include <getopt.h>

#include <sstream>

#include <readline/readline.h>

#include <readline/history.h>

#include <pari.h>

#include "main.h"

#include "ui.h"

#include "homology.h"
```

Global variables

- unsigned long **PARI_STACKSIZE** = 10000000
  Pari stacksize allocation in bytes.
- unsigned long **PARI_MAXPRIME** = 10000000
  Number of primes Pari precalculates.
- bool **VERBOSE** = false
  Output control.
- bool **DEBUG** = false
  Output control.
- int **done** = 0
  Loop breaking for the REPL loop.

Functions

- void **initialization** ()

  Initialises the `readline`, `history` *and* `Pari` *libraries.*

- int **main** (int argc, char **argv)

  Starting point for program execution.

B.7.1 Detailed Description

Main program file - herein is contained the actual executable code.

Definition in file **main.c**.

B.7.2 Function Documentation

B.7.2.1 int main (int *argc*, char ** *argv*)

Starting point for program execution.

Parses command line arguments and starts the REPL-loop.

Definition at line 80 of file main.c.

References DEBUG, done, initialization(), PARI_MAXPRIME, PARI_STACKSIZE, parse_eval(), strip_whitespace(), and VERBOSE.

B.8 main.h File Reference

Header for the main program.

Output control variables

By command line flags, the user can vary the degree of output wanted from the program.

The flags set global booleans that are defined here.

- bool **VERBOSE**
  Output control.
- bool **DEBUG**
  Output control.

B.8.1 Detailed Description

Header for the main program.

Definition in file **main.h**.

B.9 ui.c File Reference

User Interface implementation.

```
#include <iostream>

#include <cstdlib>

#include <string>

#include <ctype.h>

#include <exception>

#include "homology.h"

#include "ui.h"
```

Callbacks

The parse loop uses several specific functions to handle different tasks.

- int **cmd_quit** (std::vector< std::string > t)
  Quits the program.
- void **cmd_char** (std::vector< std::string > t)
  Sets the field characteristic.
- void **cmd_add** (std::vector< std::string > t)
  Adds either a set of monomial to `mi` *or a set of simplices to* `sc`.
- void **cmd_help** (std::vector< std::string > t)
  Prints out various help texts. A rather basic help system.
- void **cmd_hom** (std::vector< std::string > t)
  Initiates calculation of homology ranks for the simplicial complex in `sc`.
- void **cmd_den** (std::vector< std::string > t)
  Initiates calculation of the denominator polynomial of the monomial ideal specified in `mi`.
- void **cmd_clear** (std::vector< std::string > t)
  Clears all previous work from memory.
- void **cmd_print** (std::vector< std::string > t)

Gives a large information dump with all simplices and all monomials in a raw bitstring form.

- void **cmd_var** (std::vector< std::string > t)

  Set or view the name of the free variable.

Global objects

This is the instances of **MonomialIdeal** and **SimplicialComplex** in which all userdriven calculations take place.

- **MonomialIdeal mi**
- **SimplicialComplex sc**

Functions

- void **init_stack** ()

  Initializes data for facilitating garbage collection with Pari.

- std::string **strip_whitespace** (std::string &l)

  Utility function to strip whitespace from the beginning and the end of the string `l`.

- int **parse_eval** (std::string s)

  The parse and evaluate parts of the REPL loop.

Variables

- unsigned long **stacktop**

  Handling of memory management with Pari.

B.9.1 Detailed Description

User Interface implementation.

Implements the functions defined in **ui.h** More specifically contains the parse and evaluate parts of the REPL loop and its callback functions

Definition in file **ui.c**.

B.9.2 Function Documentation

B.9.2.1 int parse_eval (std::string *l*)

The parse and evaluate parts of the REPL loop.

Parameters:
l The latest line read from user input.

Definition at line 251 of file ui.c.

References cmd_add(), cmd_char(), cmd_clear(), cmd_den(), cmd_help(), cmd_hom(), cmd_print(), cmd_quit(), cmd_var(), and Tokenize().

Referenced by main().

B.10 ui.h File Reference

Header file for the UI code for the Poincaré command line calculator.

```
#include <string>
```

Help texts

The help texts for Poincaré are stored in constant string variables.

- const std::string **WELCOME**
- const std::string **COMMANDLINE_HELP**
- const std::string **ADD_HELP**
- const std::string **CHAR_HELP** = "Usage: char <coefficient field characteristic: prime number or zero>"
- const std::string **MONOMIAL_HELP**
- const std::string **SIMPLEX_HELP**
- const std::string **HOMOLOGY_HELP**
- const std::string **DENOMINATOR_HELP**
- const std::string **HELP_HELP**
- const std::string **VAR_HELP**
- const std::string **BAD_COMMAND** = "Unknown command: "
- const std::string **TOO_FEW_VAR** = "Too few arguments."
- const std::string **HELP**

Functions

- int **parse_eval** (std::string l)

  The parse and evaluate parts of the REPL loop.

- std::string **strip_whitespace** (std::string &l)

  Utility function to strip whitespace from the beginning and the end of the string `l`.

- void **init_stack** ()

  Initializes data for facilitating garbage collection with Pari.

B.10.1 Detailed Description

Header file for the UI code for the Poincaré command line calculator.

Definition in file **ui.h**.

B.10.2 Function Documentation

B.10.2.1 int parse_eval (std::string *l*)

The parse and evaluate parts of the REPL loop.

Parameters:
l The latest line read from user input.

Definition at line 251 of file ui.c.

References cmd_add(), cmd_char(), cmd_clear(), cmd_den(), cmd_help(), cmd_hom(), cmd_print(), cmd_quit(), cmd_var(), and Tokenize().

Referenced by main().

B.10.3 Variable Documentation

B.10.3.1 const std::string ADD_HELP

Initial value:

```
"Usage:
add monomial <monomial-list>
or
add simplex <simplex-list>\n"
```

Definition at line 64 of file ui.h.

B.10.3.2 const std::string COMMANDLINE_HELP

Initial value:

```
"Valid commandline options for the Poincaré calculator are:
-s <size>          Sets the stack size of Pari to <size> bytes
-p <size>          Tells Pari to precalculate <size> primes
-h                 Displays this text.\n"
```

Definition at line 58 of file ui.h.

B.10.3.3 const std::string DENOMINATOR_HELP

Initial value:

```
"Syntax: denominator
Denominator returns the denominator polynomial for the generating
function for the Poincare-Betti series of a field of given characteristic
over the monomial ring defined by the entered monomials.\n"
```

Definition at line 87 of file ui.h.

B.10.3.4 const std::string HELP

Initial value:

```
"Command syntax for the Poincaré calculator:

help         - bring up this text
quit         - end program execution
char         - set the characteristic to work with
add monomial - add monomials to the ideal generators
add simplex  - add simplices to the simplicial complex
homology     - return the homology rank function of the complex
denominator  - return the denominator polynomial of the monomial ring
               Poincaré-Betti series
clear        - reset the working copies of monomials and simplices
var          - set the homology variable string
print        - print out the current working environment
"
```

Definition at line 105 of file ui.h.

B.10.3.5 const std::string HELP_HELP

Initial value:

```
"Syntax: help [command]
Help returns either a list of available commands or more information about
a specific command.\n"
```

Definition at line 92 of file ui.h.

B.10.3.6 const std::string HOMOLOGY_HELP

Initial value:

```
"Syntax: homology
Homology returns the Hilbert series for simplicial homology of
the simplex generated by the added simplices, taken with coefficients
within a field of prime or zero characteristic (set characteristic of the
field by means of char)\n"
```

Definition at line 81 of file ui.h.

B.10.3.7 const std::string MONOMIAL_HELP

Initial value:

```
"Syntax: add monomial <monomial-list>
Add monomial is used for adding any number of space separated
 monomials as generators for a monomial ideal for
calculation of denominator polynomials.\n"
```

Definition at line 71 of file ui.h.

B.10.3.8 const std::string SIMPLEX_HELP

Initial value:

```
"Syntax: add simplex <simplex-list>
Add simplex is used for adding any number of space separated
simplices (i.e. squarefree monomials) as faces of a simplicial complex,
the homology of which may be calculated.\n"
```

Definition at line 76 of file ui.h.

B.10.3.9 const std::string VAR_HELP

Initial value:

```
"Syntax: var [string]
Var sets the variable used for the series variable used when printing
Poincare-Betti series denominator polynomials, or Hilbert polynomials
of simplicial complex homology.\n"
```

Definition at line 96 of file ui.h.

B.10.3.10 const std::string WELCOME

Initial value:

```
"Welcome to the Poincaré calculator. You can use this program to
calculate simplicial homology over prime fields and to calculate
the denominator polynomial of the Poincaré-Betti series of monomial
rings.
(c) 2004 Mikael Johansson
This program is released under the MIT License\n"
```

Definition at line 50 of file ui.h.

Bibliography

[Armstrong, 1983] Armstrong, M. A. (1983). *Basic Topology.* Undergraduate Texts in Mathematics. Springer Verlag.

[Avramov, 2002] Avramov, L. L. (2002). Homotopy Lie algebras and Poincaré series of algebras with monomial relations. *Homology Homotopy Appl.*, 4(2):17–27. The Roos Festschrift, vol. 1.

[Backelin, 1982] Backelin, J. (1982). Les anneaux locaux à relations monomiales ont des séries de Poincaré-Betti rationelles. *Comptes Rendus de l'Académie des Sciences, Paris*, 295:607–610.

[Backelin and Fröberg, 1985] Backelin, J. and Fröberg, R. (1985). Poincaré series of short Artinian rings. *Journal of Algebra*, 96(2):495–498.

[Berglund, 2003] Berglund, A. (2003). Degree bounds for Poincaré series denominators of monomial rings with few relations. Master's thesis, Stockholms Universitet.

[Berglund, 2004] Berglund, A. (2004). Personal communication.

[Birmingham, 1995] Birmingham, D. (1995). Lens spaces in the Regge calculus approach to quantum cosmology. *Physical Review*, D 52(10):5760–5772.

[Bredon, 1993] Bredon, G. E. (1993). *Topology and Geometry.* Number 139 in Graduate Texts in Mathematics. Springer Verlag.

[Doxygen, 2004] Doxygen (2004). Doxygen documentation generator. `http://www.stack.nl/~dimitri/doxygen/index.html`.

[Fröberg, 1982] Fröberg, R. (1982). A study of graded extremal rings and of monomial rings. *Mathematica Scandinavica*, 51:22–34.

[Grätzer, 1998] Grätzer, G. (1998). *General Lattice Theory.* Birkhäuser, second edition.

[Hatcher, 2002] Hatcher, A. (2002). *Algebraic Topology.* Cambridge University Press, `http://www.math.cornell.edu/~hatcher`.

[Hilton and Stammbach, 1970] Hilton, P. J. and Stammbach, U. (1970). *A Course in Homological Algebra.* Number 4 in Graduate Texts in Mathematics. Springer Verlag.

[Katzman, 2004] Katzman, M. (2004). Characteristic-independence of Betti numbers of graph ideals. arXiv:math.AC/0408016v1.

[MacLane, 1963] MacLane, S. (1963). *Homology*. Number 114 in Die Grundlehren der mathematischen Wissenschaften. Springer Verlag.

[Stanley, 1997] Stanley, R. P. (1997). *Enumerative Combinatorics*, volume 1 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press.

[Taylor, 1966] Taylor, D. (1966). *Ideals generated by monomials in an R-sequence*. PhD thesis, Chicago.

[Taylor, 2004] Taylor, P. (2004). Commutative diagrams in TEX (version 3.90). `http://www.dcs.qmw.ac.uk/~pt/diagrams/`.