IMPLEMENTING NON-CANONICAL SYLVAN RESOLUTIONS

PHOEBE KLETT

Abstract. An implementation in Macaulay2 of the Non-Canonical Sylvan Resolutions explicitly defined in [EMO20]. Intuition, explanation is given for the theoretical as well as the applied math, and any choices made are justified. A practical user's guide to the software is provided, and by-hand examples help to give a conceptual underpinning of the work done.

This is a senior thesis submitted to the Mathematics Department. Duke University. Durham, North Carolina. May, 2021.

Contents

1. Introduction

Overview.

This project's main goal is to design and implement an algorithm for computing minimal resolutions of arbitrary monomial ideals. We implement a novel method for computing the boundary maps in the resolution. More formally, given a monomial ideal I, we construct a $\mathbb{N}^n$ graded free resolution of I, as pictured below.

$$0 \leftarrow I \xleftarrow{\partial_0} \bigoplus_{a\in\mathbb{N}^n} S(-a)^{\beta_{0,a}} \xleftarrow{\partial_1} \bigoplus_{a\in\mathbb{N}^n} S(-a)^{\beta_{1,a}} \xleftarrow{\partial_2} \dots \xleftarrow{\partial_k} \bigoplus_{a\in\mathbb{N}^n} S(-a)^{\beta_{k,a}} \xleftarrow{\partial_{k+1}} \dots$$

This amounts to the following:

(1) Determine all degrees $\mathbf{a} \in \mathbb{N}^n$ with nonzero Betti number ($\beta_{i,a} \neq 0$) for some $i \in \mathbb{Z}$. Determine these i as well.
(2) Calculate the ∂_i, for all i that appear in the resolution.

We approach these tasks with the following strategy:

(1) Here we use the fact that all degrees with nonzero Betti number will appear in the lcm lattice of the generators of the ideal. So we only need to check these lcm degrees to collect all degrees with nonzero Betti number.
(2) We calculate these maps using the method implied by the recent work of Eagon, Miller, and Ordog [EMO20]. To be more specific, we leverage their explicit formula for the calculation of the differentials in the resolution. Splittings $\partial^{\mathbf{b}+}$ of the differentials $\partial^{\mathbf{b}}$ given in 1.12 are a key ingredient and are in fact themselves differentials yielding a free resolution.

We implement this method of computation in executable software, written in the language of Macaulay2 [M2].

Acknowledgements. I would like to thank Dr. Miller and Dr. Ordog for their immense support in this project. It has been a privilege and an honor to work alongside them. I would also like to thank the Duke Math Department as a whole, for providing a warm and collaborative environment in which to learn math. I have cherished the last four years of mathematical exploration.

1.1. **Background.**

We give some introductory definitions.

Definition 1.1. An ideal is a monomial ideal if its generators are all monomials.

Convention 1.2. $S = \Bbbk[\mathbf{x}] = \bigoplus_{\mathbf{b}\in\mathbb{N}^n} \Bbbk\left\{\mathbf{x}^{\mathbf{b}}\right\}$ where $\mathbf{x}^{\mathbf{b}} = x_1^{b_1}x_2^{b_2}\cdots x_n^{b_n}$ is a monomial.

Definition 1.3. A free S-module F is a direct sum of copies of S. I.e. $F \cong S^r$. F is $\mathbb{N}^n$-graded if $F \cong S(-a_1) \oplus \cdots \oplus S(-a_r)$ for some $a_1, \ldots, a_r \in \mathbb{N}^n$. Note that $S(-a)$ indicates the shifted gradation, so $S(-a)_b = S_{b-a}$ for all degrees $a \in \mathbb{N}^n$.

Definition 1.4. A sequence

$$\mathcal{F}_\bullet : 0 \leftarrow F_0 \xleftarrow{\partial_0} F_1 \xleftarrow{\partial_1} \ldots \xleftarrow{\partial_{\ell-1}} F_{\ell-1} \xleftarrow{\partial_\ell} F_\ell \leftarrow 0$$

of maps of free S-modules is a **complex** if $\partial_i \circ \partial_{i+1} = 0$ for all i. The maps between consecutive pairs of modules are called **differentials**.

Definition 1.5. A sequence $\mathcal{F}_\bullet$ (as in 1.4) is exact if $Im(\partial_{i+1}) = Ker(\partial_i)\ \forall i$.

Definition 1.6. A sequence $\mathcal{F}_\bullet$ (as in 1.4) is a **free resolution** of a module M over $S = \Bbbk[x_1, \ldots, x_n]$ if it is exact everywhere except degree 0, and $M \cong F_0/im(\partial_0)$.

Definition 1.7. A free resolution is **minimal** if each free module has the minimal number of generators. More precisely, the rank in each homological degree should be minimal among all resolutions of M.

In our case, $M = S/I$ is $\mathbb{N}^n$ graded, so our free resolution is $\mathbb{N}^n$ graded, as shown below.

$$0 \leftarrow M \leftarrow \bigoplus_{a\in\mathbb{N}^n} S(-a)^{\beta_{0,a}} \leftarrow \bigoplus_{a\in\mathbb{N}^n} S(-a)^{\beta_{1,a}} \leftarrow \cdots \leftarrow \bigoplus_{a\in\mathbb{N}^n} S(-a)^{\beta_{k,a}} \leftarrow \dots$$

For all non-negative integers i and all degrees $a \in \mathbb{N}^n$, $\beta_{i,a}$ is the number of copies of $S(-a)$ appearing in the i-th module of the resolution and is called the $\mathbb{N}^n$- graded **Betti number**. Formulas for the Betti numbers, the ranks of the free modules in a minimal free resolution, have been known since the 1970s when Hochster [Hoc77] gave a correspondence between $\beta_{i,a}$ and the dimension of the $(i-1)$-st reduced homology of Koszul simplicial complex of I in degree a.

Definition 1.8. For a monomial ideal I and a nonnegative integer vector $b \in \mathbb{N}^n$ with n entries, the (upper) Kozsul simplicial complex of I in degree b is

$$K^b I = \{\tau \in \{0,1\}^n \mid x^{b-\tau} \in I\}$$

Intuitively, the Kozsul simplicial complex (in degree b) encodes all of the combinations of distinct "steps backward" we can take standing at lattice point b, without leaving the ideal.

Theorem 1.9 (Hochster's formula)**.** [Hoc77] *Fix a monomial ideal $I \subseteq \Bbbk[x]$ and a degree vector $b \in \mathbb{N}^n$. Then there is a natural isomorphism of vector spaces*

$$Tor_i(\Bbbk, I)_b \cong \tilde{H}_{i-1}(K^b I; \Bbbk).$$

A direct consequence of this is the identification $\beta_{i,b}(I) = dim_{\Bbbk}\tilde{H}_{i-1}(K^b I; \Bbbk)$.

So in order to determine which degrees $a \in \mathbb{N}^n$ have nonzero Betti number, we calculate $K^a I$ for all a in the lcm lattice of the ideal and $\widetilde{H}_i K^{\mathbf{a}} I$ for all i, allowing us to connect notions of homology with invariants in our resolution. However, the differentials remain more evasive and it was only this past year that closed-form, canonical, explicit and universal formulas for the differentials in the resolutions were given [EMO20].

The construction given in [EMO20] gives the differentials (called sylvan homomorphisms) as matrices, mapping between homology vector spaces of the Koszul simplicial complexes of I in degrees with nonzero Betti numbers. In order to determine the entries of our matrices, we first consider all pairs of degree vectors a, b with nonzero Betti numbers, such that the nonzero Betti number for a is in homology of lower degree than that of b, and $b \succ a$. Then we consider all lattice paths between a and b. For each lattice path between a and b, we construct a chain link fence (or a map from an i simplex to an $(i-1)$ simplex) with edges whose weights depend upon a sequence of hedges, or higher dimensional analogues of spanning trees. A hedge of dimension i is a choice of shrubbery in dimension i and a stake set in dimension $i-1$.

Definition 1.10 ([EMO20, Definition 2.1])**.** Fix non-negative integers $n, m \in \mathbb{N}$.

(1) A **shrubbery** for a surjection $B \twoheadleftarrow \Bbbk^n$ is a subset $T \subseteq \{e_1, \dots, e_n\}$ of the standard basis such that the composite $B \twoheadleftarrow \Bbbk^n \hookleftarrow \Bbbk\{T\}$ is an isomorphism ∂_T.
(2) A **stake set** for an injection $\Bbbk^m \hookleftarrow B$ is a subset $S \subseteq \{e_1, \dots, e_n\}$ of the standard basis such that the composite $\Bbbk\{S\} \twoheadleftarrow \Bbbk^m \hookleftarrow B$ is an isomorphism ∂_S.
(3) If $\Bbbk^m \xleftarrow{\partial} \Bbbk^n$ is a linear map with image $B = \partial(\Bbbk^n)$, then a **hedge** for ∂ is a choice ST of a shrubbery T for ∂ and stake set S for ∂.

A choice of a sequence of hedges is a **community**. We arrive at entries in our differential by summing over these edge weights. By the following theorem 1.11, these differentials induce a minimal free resolution of our monomial ideal.

Theorem 1.11 ([EMO20, Corollary 9.5])**.** *Fix a monomial ideal and a community for each Koszul simplicial complex K^bI. The sylvan homomorphism for these data on each comparable pair $b \succ a$ of lattice points induces a homomorphism $\widetilde{Z}_{i-1}K^{\boldsymbol{a}}I \leftarrow \widetilde{Z}_iK^{\boldsymbol{b}}I$ that vanishes on $\widetilde{B}_iK^{\boldsymbol{b}}I$ and hence it induces a well defined sylvan homology morphism $\widetilde{H}_{i-1}K^{\boldsymbol{a}}I \leftarrow \widetilde{H}_iK^{\boldsymbol{b}}I$. The induced homomorphisms*

$$\widetilde{H}_{i-1}K^{\boldsymbol{a}}I \otimes \Bbbk[x](-\boldsymbol{a}) \leftarrow \widetilde{H}_iK^{\boldsymbol{b}}I \otimes \Bbbk[x](-\boldsymbol{b})$$

of $\mathbb{N}^n$-graded free $\Bbbk[x]$ modules constitute a minimal free resolution of I.

So what are these sylvan homomorphisms? An explicit formula is given in the following theorem.

Theorem 1.12 ([EMO20, Theorem 8.1])**.** *Fix a monomial ideal I. Any splittings $\partial^{\mathbf{b}+}$ of the differentials $\partial^{\mathbf{b}}$ of the Kozsul simplicial complexes $K^{\mathbf{b}}I$ for $\mathbf{b} \in \mathbb{N}^n$ that are themselves differentials satisfying*

(1) $\partial^{\mathbf{b}}\partial^{\mathbf{b}+}\partial^{\mathbf{b}} = \partial^{\mathbf{b}}$ and
(2) $\partial^{\mathbf{b}+}\partial^{\mathbf{b}}\partial^{\mathbf{b}+} = \partial^{\mathbf{b}+}$

yield a free resolution of I whose differential from homological stage $i+1$ to stage i has its component $\widetilde{H}_{i-1}K^{\mathbf{a}}I \otimes \Bbbk[x](-\mathbf{a}) \leftarrow \widetilde{H}_iK^{\mathbf{b}}I \otimes \Bbbk[x](-\mathbf{b})$ induced by the map

$$\widetilde{H}_{i-1}K^{\mathbf{a}}I \xleftarrow{D} \widetilde{H}_iK^{\mathbf{b}}I$$

in $\mathbb{N}^n$-degree $\mathbf{b}$ that acts on any i-cycle in $\widetilde{Z}_iK^{\mathbf{b}}I$ via

$$D = \sum_{\lambda \in \Lambda(a,b)} (I^{\mathbf{a}} - \partial_i^{\mathbf{a}+}\partial_i^{\mathbf{a}})d_1^{\lambda_\ell}(\prod_{j=1}^{\ell-1} \partial_i^{\mathbf{b_j}+}d_1^{\lambda_j})(I^{\mathbf{b}} - \partial_{i+1}^{\mathbf{b}}\partial_{i+1}^{\mathbf{b}+})$$

where $d_1^{\lambda_j}$ takes $\tau \in \{1, \dots, n\}$ to 0 if $\lambda_j \notin \tau$ and to $(-1)^{\tau \smallsetminus \lambda_j \subset \tau}\tau \smallsetminus \lambda_j$ if $\lambda_j \in \tau$.

Note next that when we define $\partial^{\mathbf{b}+}$ as below, we satisfy the requisite conditions as specified by the theorem.

Definition 1.13 ([EMO20, Definition 5.1])**.** Fix a linear map $\Bbbk^m \xleftarrow{\partial} \Bbbk^n$. For any hedge ST, the hedge splitting

$$\partial^+_{ST} : \Bbbk^m \to \Bbbk^n$$

is defined by its action on the basis $\bar{S} \cup \partial T$:

(1) $\partial^+_{ST}(\sigma) = 0$ for any non-stake $\sigma \in \bar{S}$
(2) $\partial^+_{ST}(\partial\tau) = \tau$ for any face $\tau \in T$

Using the above formula for D in combination with the defintion given in 1.13 we can compute these sylvan homomorphisms explicitly. We do so by hand in Section 2 and then using executable code in Section 3.

1.2. **Related work.**
There are existing software packages that compute resolutions of monomial ideals. These include Macaulay2 [EMO20] and CoCoA [CoCoA]. We write the code detailed in this paper using Macaulay2's language, taking advantange of their methods for computing homology, storing simplicial complexes, and more. Research in Topological Data Analysis (TDA) [E020] has shown that these objects carry information in a matroidal fashion. This implies greedy (fast) algorithms may be used to compute them. We also hope to implement a very similar algorithm for Yuzvinsky's resolution of a monomial ideal [Yuz99], upon the completion of an analogous combinatorial formula for the mappings in this resolution.

2. Examples by hand

2.1. **Getting the bettis.** The first step on the way to resolving a monomial ideal is finding all degrees that appear in the grading of the resolution, i.e. finding all the Betti numbers. To find these degrees, we use the fact that all of the degrees with nonzero Betti number will appear in the lcm lattice of the generators of the ideal. Precisely, the degrees with nonzero Betti number will be contained in this set, usually properly. So we only need to check the lcms of all subsets of the generators to find all such degrees.

We do an example computation. Suppose our ideal is given as $I = \langle xy, y^3, z^2, xz, yz\rangle$. We would then consider all subsets of these generators. For example, consider the subset of generators $\langle z^2, xz, yz\rangle = \langle (002), (101), (011)\rangle$. In this case the lcm degree is (112). We take as candidates for degrees that appear in the resolution the degree (112) and all others in L, where $L := \{b \mid b \text{ is the lcm of a subset of generators of } I\}$. In particular, it's important to note that it's not guaranteed that all of these $b \in L$ will have nonzero Betti number, but all degrees with nonzero Betti number will indeed be in L.

Given a complete L, we check to see what subset of L are such degrees. We do this by using the (upper) Kozsul simplicial complex 1.8, and Hochster's formula 1.9. In

order to determine which degrees in L have nonzero Betti number, we calculate $K^b I$ and its reduced homology in all degrees. We do a few example calculations:

$$K^{(112)}I = \{\{001\}, \{011\}, \{101\}\}$$
$$K^{(111)}I = \{\{001\}, \{010\}, \{100\}\}$$
$$K^{(131)}I = \{\{101\}, \{110\}, \{011\}\}$$
$$dim_{\Bbbk}\tilde{H}_{2-1}(K^{(131)}I) = dim_{\Bbbk}\tilde{H}_1(K^{(131)}I) = 1$$

So $\beta_{(2,131)} = 1$. At this point, doing the above for all degrees b and homology degrees n, we have a skeleton of the resolution. That is, we have the grading for all modules in the resolution, but are lacking maps between homology vector spaces.

2.2. **Tripartitions and hedges.** Tripartitions [E020] and hedges will be crucial ingredients for computing the maps between homolgy vector spaces in the resolution. Developed in parallel to the work of [EMO20], the construction and computation of tripartitions as given in [E020] is closely related to the sylvan objects of interest.

Definition 2.1 ([E020, Theorem 4])**.** A **tripartition** in homology degree p is denoted $A_p \bigsqcup A^p \bigsqcup E_p$, where A_p is a shrubbery, A^p is a stake set, and E_p is called the homology set.

As the name implies, E_p can be used to find the basis for homology in degree p. In particular, $\{z_p(\sigma_j) \mid \sigma_j \in E_p\}$ is a basis for homology H_p. Here $z_p(\sigma_j) = I - \partial^+\partial$. We call this "projecting onto cycles".

We use a tripartition to compute a basis for homology vector spaces. For example, say $\mathbf{b} = (131)$. Our differential will act on i-cycles in $\tilde{Z}_i K^{\mathbf{b}} I$, so the first thing to compute is a basis for homology in degree 131, i.e. $\tilde{H}_1 K^{(131)} I$. Using a tripartition, this is simple. We do an example computation.

We first look at boundary map $\partial_1 : \tilde{C}_1 \to \tilde{C}_0$.

$$\partial_1 : \tilde{C}_1 \xrightarrow{\begin{array}{c} \quad xy \quad xz \quad yz \\ x\begin{bmatrix} -1 & -1 & 0 \\ y\begin{bmatrix} 1 & 0 & -1 \\ z\begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \end{bmatrix} \end{bmatrix}} \tilde{C}_0$$

Where we determine the entries of the matrix using the resulting coefficients of our usual boundary map. For example: $\partial(xy) = y - x$.

Our goal now is to compute the three components of the tripartition, i.e. we want to find $\{A^p, A_p, E_p\}$. To build the shrubbery ($T_1 = A_1$) we take the pivot columns determined by usual row reduction to echelon form [SA10]. In our example, the resulting $A_1 = T_1$ is $\{xy, xz\}$.

To build our stake set, we do something very similar. Because the stake set is exactly dual to the shrubbery, we may simply transpose the appropriate boundary matrix and

take a basis of this. However, we note that in the case of finding a tripartition (different from that of a hedge) we need to find the shrubbery and the stake set in the same dimension, so we need to consider different boundary maps. In this case, we consider $\partial_2^T : \tilde{C}_1 \to \tilde{C}_2$ when constructing the stake set. Lastly, instead of considering the columns left to right, we choose them right to left. We do this to ensure the condition of a community is satisfied, i.e. $T_i \cap S_i = \emptyset$. We are ensured to meet this condition given these algorithms by [E020]. In our example we have $A^1 = S_1 = 0$ since we don't have any 2-simplices.

Lastly, E_p is the leftovers. In this case (in degree 1), we are missing $\{yz\}$. Finally, to get a basis for homology we apply the map $(I^{\mathbf{b}} - \partial_{i+1}^{\mathbf{b}} \partial_{i+1}^{\mathbf{b}+})$ to elements in the homology set, E_p.

$$z_p(yz) = (I - \partial^+\partial)(yz) = yz - \partial^+(z-y) = yz + \partial^+(\partial(xz - xy)) = yz - xz + xy = \zeta_{T_i}(yz)$$

where $\zeta_{T_i}(yz)$ is our notation for the circuit of yz [EMO20]. Note also that ∂^+ is defined by its action on shrubbery elements and non-stakes, so we express $z - y$ as a combination of these elements to compute the result above. This will be a common tactic.

Hedges are similar. A hedge is composed of a shrubbery in degree i and a stake set in degree $i-1$, and we compute these objects in the same way. We simply note that the degree of the shrubbery and stake set are different, so transposing the same boundary map will be enough - rather than taking a map between different degrees.

2.3. **Sylvan maps.** We use the tools from 2.2 to construct a map between homology vector spaces. We walk through an example computation to give intuition to how these sylvan homomorphisms are computed. We are following the explicit formula given for these maps, given by 1.12.

Let's take $I = \langle xy, y^3, z^2, xz, yz \rangle$ to be our ideal. Our first step will be finding all of the degrees that have non-zero Betti number, as described in 2.1. Using the example done in 2.1, we pick our two degrees to be $\mathbf{a} = 131$ and $\mathbf{b} = 131$, having computed $\beta_{(1,111)} = 1$ and $\beta_{(2,131)} = 1$. We now want to consider all lattice paths between a and b. in this case there is only one path: each time we step backwards in y direction. I.e. $\Lambda(a,b) = \{$ all lattice paths from a to b $\} = \{(111, 121, 131)\}$. Call this single path λ. We've now cooked up all the ingredients we need to begin computing our differentials. Recall our path $\lambda = \{(111, 121, 131)\}$. This tells us that we are looking for the map from $\tilde{H}_0 K^{(111)} I \leftarrow \tilde{H}_1 K^{(131)} I$. D acts on i-cycles in $\tilde{Z}_i K^{\mathbf{b}} I$, so the first thing to compute is a basis for homology in degree 131, i.e. $\tilde{H}_1 K^{(131)} I$. As described in 2.2, we obtain this basis using a Tripartition. In fact, the tripartition we computed in 2.2 gives us what we need. Recall our computation: $\{A^1, A_1, E_1\} = \{\emptyset\}, \{xy, xz\}, \{yz\}$. And the resulting basis for homology: $yz - xz + xy$.

Next, we move to the middle piece of the map: $(\prod_{j=1}^{\ell-1} \partial_i^{\mathbf{b_j}+} d_1^{\lambda_j})(yz - xz + xy)$ 1.12. Note that our path has only one middle step (121), so there will only be one term in our

product. Let's start with $d_1^{\lambda_j}$. How do we go about this? Let's give a brief subexample. Say we have $\partial(xy) = y - x$. Then, each of those components are given the notation: $\partial^x = y, \partial^y = -x$. And in our final notation then we assign $d_1^{\lambda_1} = d_1^{\ell_2 = y} = \partial^y$. In our example we have $\lambda = (111, 121, 131)$ (the path from b to a), with $\lambda_1 = (131 - 121) = (010) = y = \ell_2$. Applying this to the result of the first step $(yz - xz + xy)$, we have: $d_1^{\lambda_1}(yz - xz + xy) = (z - y) - 0 + (y - x) = z - x$.

We now compute $K^{121} = \{xy, yz\}$ and $ST_1 = (T_1, S_0) = (\{xy, yz\}, \{z, y\})$ by the methods described in 2.1 and 2.2. Recall that we compute a hedge using the same method as a tripartition, only noting that because the shrubbery and stake set are in different degrees, we simply transpose the appropriate boundary map to compute the stake set. I.e. T_1 is a shrubbery for $\partial_1 : \tilde{C}_1 \to \tilde{C}_0$ and S_0 is a stake set for $\partial_1^T : \tilde{C}_0 \to \tilde{C}_1$.

And lastly we compute: $\partial^+_{ST_1}(z - x) = \partial^+_{ST_1}(\partial(yz + xy)) = yz + xy$ by expressing $z - x$ as a linear combination of elements in ∂T_1 and $(S_0)^C$. We know how $\partial^+_{ST_1}$ acts on elements in the boundary of shrubbery and in the complement of the stake set, as defined in 1.13. This element, $yz + xy$, we pass onto the next map.

Lastly, we need to compute: $(I^{\mathbf{a}} - \partial_i^{\mathbf{a}}\partial_i^{\mathbf{a}+} - \partial_i^{\mathbf{a}+}\partial_i^{\mathbf{a}})d_1^{\lambda_\ell}(z - x)$. Note this last map is slightly different than that in the formula in 1.12, since we would like to project onto homology in this case. We have $d_1^{y=\lambda_\ell}(yz + xy) = z - x$. We need to compute a basis for $\widetilde{H}_0 K^{111}$ now, so we compute one more tripartition, in degree 0. We get $T_0 = A_0 = \{x\}, S_0 = \emptyset$ by noting that S_0 is a shrubbery for $\partial_1^T : \widetilde{C_0} \to \widetilde{C_1}$ and there are no edges in $K^{(111)}I$. Lastly we have $E_0 = \{y, z\}$.

Taking the circuit of all elements in $E_0 = \{y, z\}$ to get a basis for $\widetilde{H}_0 K^{111}$, we have:

(1) $\zeta(y) = y - x = (I - \partial^+\partial)(y)$
(2) $\zeta(z) = z - x$

So our chain from above, $z - x$ can already be written as a linear combination of elements in the basis for homology, this last mapping gives $z - x$ again. Recalling our basis for $\widetilde{H}_1 K^{131}$, which was also the same calculation as the first map in our formula, i.e. $\{yz - xz + xy\}$ we now have:

$$\begin{array}{ccc} & & yz - xz + xy \\ & y - z & \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ & z - x & \\ \tilde{H}_0 K^{111} & \longleftarrow & \tilde{H}_1 K^{131} \end{array}$$

In order to finish this sylvan homomorphism, we'd need to repeat this process for all paths between a, b and for all valid a, b for the given homology degrees, summing over all these combinations. But instead of working through all these computations, why not let an algorithm do the heavy lifting? In the next section we explain how to do all the of above computations with a few lines of code.

3. The SylvanResolutions code base

In this section we provide what serves as a user's guide to the code base SylvanResolutions. This code base is comprised of thirteen methods, written in Macaulay2 [M2], for computing sylvan resolutions quickly and efficiently. We walk through each method, covering what it computes, how it computes this object, and how to use it in a practical sense.

To try out the methods we detail here, please visit `https://faculty.math.illinois.edu/Macaulay2/` to learn more about downloading the Macaulay2 software, or alternatively use the interactive Web Interface available here `https://www.unimelb-macaulay2.cloud.edu.au/#editor`.

Lastly, the methods detailed here rely on the following Macaulay2 packages: SimplicialComplexes, Posets, RandomIdeals. You can find more info on these packages here `https://faculty.math.illinois.edu/Macaulay2/doc/Macaulay2/share/doc/Macaulay2/Macaulay2Doc/html/_packages_spprovided_spwith_sp__Macaulay2.html`. To begin working with the methods provided in the SylvanResolutions file, begin by loading the above packages, as is shown below.

```
load "SimplicialComplexes.m2"
load "Posets.m2"
load "RandomIdeals.m2"
```

3.1. A Laundry List.

We enumerate all available methods and briefly describe their function.

(1) **delPlus()**. This method computes the hedge splitting of the differential, defined by 1.13. This function delegates the computation to the methods **delPlus1()** and/or **delPlus2()** depending on the homological degree.

(2) **dLambda()**. This method computes the value of the function called $d_1^{\lambda_j}$ of the given chain, τ.

(3) **hedge_trimmer()**. This method computes a Hedge ST_i (in specified degree i).

(4) **bettiDs()**. This method produces all pairs of degrees (a, b) such that **a** has nonzero Betti number in homological degree $k-1$ and **b** has nonzero Betti number in homological degree k.

(5) **upperKoszul()**. This method computes the upper Koszul Simplicial Complex of given Ideal and degree.

(6) **lambdaGen()**. This method computes all paths (in differential degrees) between two given degrees.

(7) **Map_1()**, **Map_2()**, **Map_3()**, **Map_4()**. These methods can be called to get intermediate results in the computation of the entire sylvan map, for all homology degrees. The last two of these methods, 3 and 4, call the method **Helper()** which simply computes the last step of the mapping.

(8) **pathS()**. This method computes the path (in monomials) between two monomials.
(9) **sylvanMap()**: This method computes the sylvan homomorphism between two homology degrees.
(10) **sylvanResolution()**. This function computes the entire sylvan resolution (I.e. computes the differentials for all homology degrees).

3.2. **Methods in depth.** Here we discuss the methods in depth, how they function and how use them. We begin with the methods whose output feeds into the larger functions, and then tackle the larger functions.

3.2.1. *Beginning Methods.*

The method **delPlus** computes the hedge splitting in the same way as we did by hand in 2.3. That is, it represents the given chain as a linear combination of elements in the (proper) boundary of the shrubbery and complement of the stake set, and then applies the splitting to this new form of the chain. Again, we do this since we know how the splitting acts on elements in these two sets. Below the code is given, and the algorithm described above is highlighted with comments.

```
1  ------------------------------------------------------
2  -- Method description. (delPlus):
3  -- This method computes the hedge splitting of the differential,
       given a Hedge. See Definition 1.12.
4  -- Inputs: R, multigraded polynomial ring
5  --         S, sequence of:
6  --            H, (Hedge) Sequence of two lists, (shrubbery in dim i,
       stake set in dimension i-1)
7  --            K, SimplicialComplex, (upper Kozsul simplicial complex
       in appropriate degree )
8  --            i, integer, degree of splitting
9  --            d, Matrix, chain to apply splitting
10 -- Outputs: Ring element, (Chain) - result of of Del^+_i (b) as given
        in Definition 1.12
11 ------------------------------------------------------
12 delPlus = method()
13 delPlus(Ring, Sequence) :=(R,S) -> (
14      H = S#0;
15     K = S#1;
16     i = S#2;
17     d = S#3;
18
19     T = H#0;
20     S = H#1;
21
22     -- take boundary of faces in shrubbery --
```

```
23      B = boundary(i,simplicialComplex(T)); -- matrix form --
24      try Bs = flatten(entries(matrix{generators R} * B)) then Bs =
        flatten(entries(matrix{generators R} * B)) else Bs = flatten(
        entries(transpose(matrix{generators R}) * B));
25      -- in list/ring elt form --
26
27      -- get elements of complement of stake set --
28      A = flatten(entries(faces(i-1,K)));
29      if A#0 == 1 then Sc = {0} else Sc = apply(A, i -> if any(S, j ->
        R_(toString(j)) == R_(toString(i))) then null else i);
30      Scf = delete(null,Sc);
31      b0 ={};
32      C= {};
33      try for i in 0..#Scf-1 do for j in 0..#(Scf#i)-1 do b0 = append(
        b0, characters(toString(Scf#i#j))) else C={};
34      try for i in 0..#b0-1 do C = append(C,apply(b0#i, j -> degree(R_(
        toString(j))))) else C={};
35      if C != {} then D = transpose(matrix(apply(C, i->sum(i)))) else D
        =matrix{{0}};
36
37      -- concatenate 2 matrices --
38      P = B | D;
39
40      -- represent x as linear comb of above two --
41      t = entries(d);
42      t2 = apply(t, i-> listForm(i_0));
43      t3 = apply(t2, i->apply(i,j-> j#1*j#0));
44      t4 = sum(flatten t3);
45      if t4 == toList(#t4:0) then t4 = flatten t;
46
47      P1 = lift(P,ZZ);
48      d1 = lift(matrix{t4},ZZ);
49      if d1 !=0 then w = solve(P1,transpose(d1)) else w =0;
50
51      -- evaluate w in its new form  --
52      if w != 0 then t = flatten(entries(w)) else t = {};
53      if t != {} then step1 = take(t,#Bs) else step1 = {0};
54      if t != {}  then step2 = take(t,{#Bs,#t}) else step2 ={0};
55
56      q = {};
57      for i in 0..#step1-1 do if step1#i != 0 then q = append(q,(step1#
        i)*T#i);
58      for i in 0..#step2-1 do if step2#i != 0 then q = append(q,(step2#
        i)*Scf#i);
```

```
    sum(q)
)
```

The method **dLambda** computes the result of $d_1^{\lambda_j}(\tau)$ as described in 2.3. The algorithm computes this result in the same manner as we did by hand. I.e. it removes all faces in the chain τ which do not have the ring variable λ_j, takes the boundary of the remaining faces, and selects the part of the resulting chain which we call ∂^{λ_j}. In the case where we are working to compute the map from homological degree 0 to -1, this map can be computed more efficiently. We leverage this by calling a different function in this case. We use **dLambda1** for all cases where we are not in the 1st homology, and **dLambda1** in the case we are in the 1st homology. These steps are outlined by comments in the code below.

```
-------------------------------------------------------------
-- Method description. (dLambda):
-- This method computes the value of the function labelled d_1^(
   lambda_j) of a given chain
-- Inputs: R, polynomial ring
--         L, List of lists (degrees), (path lambda from degree a to
   degree b)
--         L2, List with:
--            j, Integer, (index of interest) goes 1..l
--            k, homology degree
--         M, Matrix, the chain Tau
-- Output: Chain, the result of d^lambda_j(chain Tau) as defined in
   the paper
-------------------------------------------------------------
dLambda = method()
dLambda(Ring,List,List, Matrix) := (R,L,L2,M) -> (
    j = L2#0;
    k = L2#1;
    -- call the appropriate helper function --
    if k ==0 then ret = dLambda2(R,L,j,M);
    if k > 0 then ret = dLambda1(R,L,j,M);
    ret
)
-------------------------------------------------------------
-- Method description. (dLambda1):
-- This method computes the value of the function labelled d_1^(
   lambda_j) of a given chain
--         for case k>0
-- Inputs: R, polynomial ring
--         L, List of lists (degrees), (path lambda from degree a to
   degree b)
```

```
--          j, Integer, (index of interest) goes 1..l
--          M, Matrix, the chain Tau
-- Output: Chain, the result of d^lambda_j(chain Tau) as defined in
   the paper
-----------------------------------------------------------------
dLambda1 = method()
dLambda1(Ring,List, ZZ, Matrix) := (R,L,j,M) -> (
    -- get direction moving in degree form --
    E0 = L#j;
    -- translate to an index --
    E = position(E0, i -> i != 0);
    -- record relevant info --
    M2 = substitute(M,F);
    m = flatten entries M2;
    n = (degree m_0)#0;
    Tau={};
    t = entries(M);
    t2 = apply(t, i-> listForm(i_0));
    t3 = apply(t2, i->apply(i,j-> j#1*j#0));
    Tau = flatten t3;
    -- get complexes and boundary maps --
    s = squareFree(n+1,F);
    s1 = monomialIdeal(s);
    S = simplicialComplex(s1);
    -- determine how to apply boundary map --
    bdy = boundary(n-1,S);
    b2 = apply(entries bdy, i-> apply(i,j->abs(j)));
    testTau = apply(Tau, i-> apply(i,j->abs(j)));

    holder = apply(testTau, i-> position(b2, j-> j == i));
    -- remove face in position lambda_j if there, if absent send to
   0--
    holds = new MutableHashTable;
    for ii in 0..#holder-1 do holds#ii = (Tau#ii,entries bdy_(holder#
   ii));
    for ii in 0..#values(holds)-1 do if holds#ii#0#E ==0 then remove(
   holds,ii);
    for ii in keys(holds) do holds#ii = (holds#ii#0,replace(E,0,holds
   #ii#1));

    -- give the right sign --
    if bdy == 0 then bdy = matrix{new List from #(generators R):0_R};

    G = {};
```

```
    for i in keys(holds) do G = append(G, faces(n-2,S)*transpose(matrix{holds#i#1}));

    sum(G)
)
-----------------------------------------------------------------
-- Method description. (dLambda2):
-- This method computes the value of the function labelled d_1^(lambda_j) of a given chain
--          for case k =0
-- Inputs: R, polynomial ring
--          L, List of lists (degrees), (path lambda from degree a to degree b)
--          j, Integer, (index of interest) goes 1..l
--          M, Matrix, the chain Tau
-- Output: Chain, the result of d^lambda_j(chain Tau) as defined in the paper
-----------------------------------------------------------------
dLambda2 = method()
dLambda2(Ring,List, ZZ, Matrix) := (R,L,j,M) -> (
    -- get direction moving in degree form --
    E0 = L#j;
    -- translate to an index --
    E = position(E0, i -> i != 0);
    -- record information in degree form --
    M2 = substitute(M,F);
    m = flatten entries M2;
    n = (degree m_0)#0;
    Tau={};

    t = entries(M);
    t2 = apply(t, i-> listForm(i_0));
    t3 = apply(t2, i->apply(i,j-> j#1*j#0));
    Tau = flatten t3;
    Tau2 ={};
    for ii in 0..#Tau-1 do if Tau#ii#E != 0 then Tau2 = append(Tau2, Tau#ii);
    -- get complexes and boundary map --
    s = squareFree(n+1,F);
    s1 = monomialIdeal(s);
    S = simplicialComplex(s1);

    s2 = squareFree(n,F);
    s3 = monomialIdeal(s2);
```

```
    S2 = simplicialComplex(s3);

    bdy = boundary(n-1,S);

    -- give the right sign --
    if bdy == 0 then bdy = matrix{new List from #(generators R):0_R};

    if Tau2 != {} then G = apply(Tau2, i-> bdy*transpose(matrix{i}));
    if Tau2 != {} then G2 = apply(G, i -> faces(n-2,S2)* i);
    try G2#0 else 0
)
```

The method **hedge_trimmer** computes a hedge ST_i (in specified degree i). It does this by the same method by hand as is described in 2.2. I.e. it does this by reduced row echelon form of the correct boundary matrix, selecting pivot columns right to left (for the shrubbery), and then left to right (for the stake set).

```
-------------------------------------------------------
-- Method description. (hedge_trimmer):
-- This method computes Hedge ST_i (in specified degree i)
-- Inputs: K, SimplicialComplex, (in most cases upper Kozsul
   simplicial complex)
--         i, homological degree
-- Outputs: Shrubbery (Ti) in dimension i (first entry in output)
--          Stake Set (S(i1)) in dimension i-1 (second entry in
   output)
-------------------------------------------------------
hedge_trimmer = method()
hedge_trimmer(SimplicialComplex, ZZ) := (K,i) -> (
    -- get shrubbery --
    bi = boundary(i,K);
    r = reducedRowEchelonForm bi;
    p = pivots(r);
    p1 = apply(p, i -> i#0);
    p2 = unique(p1);

    f = flatten(entries(faces(i,K)));
    v = new MutableHashTable;
    scan(0 .. #f-1, i -> v#i = f#i);

    Ti = apply(p2, i -> v#i);

    -- get stake set --
    bi1 = transpose(bi); -- same map transposed --
    -- flip matrix to take pivots from right to left --
```

```
    n ={};
    for i in 0..#(entries bi1)-1 do n = append(n,reverse(entries bi1)
   #i);
    r0 = matrix n;
    r1 = reducedRowEchelonForm r0;
    p3 = pivots(r1);
    p4 = apply(p3, i -> i#0);
    p5 = unique(p4);

    f1 = reverse(flatten(entries(faces(i-1,K))));
    v1 = new MutableHashTable;
    scan(0 .. #f1-1, i -> v1#i = f1#i);

    Si1 = apply(p5, i -> v1#i);

    Ti,Si1
```

The method **bettiDs** computes all pairs of degrees (a,b) s.t. **a** has nonzero Betti no. in homological degree $k-1$ and **b** has nonzero Betti no. in homological degree k. It does this in the same manner as the by-hand example in 2.1. That is, it looks at the homology of the upper Koszul simplicial complex for all degrees in the lcm lattice.

```
------------------------------------------------------------
-- Method description. (bettiDs):
-- This method produces all pairs of degrees (a,b) s.t. a has nonzero
    Betti no. in homological degree k-1 and b has nonzero Betti no.
   in homological degree k
-- Inputs: F, (Polynomial Ring), Poly Ring over which you want to
   work (w/o Degree specification)
--         I, (Monomial Ideal), Ideal in question
--         k, (integer), degrees of homology of interest are k, k-1
-- Output: C, (List), contains all pairs of degrees (a,b) where a,b
--                    have non-zero Betti # in homology degree k-1, k
   (respectively)
------------------------------------------------------------
bettiDs = method()
bettiDs(Ring,Ideal, ZZ) := (F,I,k) -> (
    M = I;
    LM = lcmLattice M;
    H = new MutableHashTable;
    As = new MutableHashTable;
    Bs = new MutableHashTable;
    -- check for non-zero homology in degrees k, k-1 --
```

```
    for i in 0..(#(LM_*))-1 do if homology(k,upperKoszul(F,M,
    monomialIdeal(LM_i)),F) != 0 then Bs#(LM_i) = numgens prune
    homology(k,upperKoszul(F,M,monomialIdeal(LM_i)),F);
    for i in 0..(#(LM_*))-1 do if homology(k-1,upperKoszul(F,M,
    monomialIdeal(LM_i)),F) != 0 then As#(LM_i) = numgens prune
    homology(k-1,upperKoszul(F,M,monomialIdeal(LM_i)),F);
    C ={};
    k1 = keys(As);
    k2 = keys(Bs);
    -- pass back all the combinations of (a,b) --
    for i in 0..#As-1 do for j in 0..#Bs-1 do C = append(C,(k1#i,k2#j
    ));
    unique(C);
    -- make sure no a's are 1 --
    D = {};
    for ii in 0..#C-1 do if C#ii#0 != 1 then D = append(D,C#ii);
    D
```

The method **upperKoszul** computes the upper Koszul Simplicial Complex of given Ideal and degree. It does this by testing all squarefree degrees to see if such a "step backward" keeps you in the ideal.

```
-------------------------------------------------------
-- Method description. (upperKoszul):
-- This method computes the upper Koszul Simplicial Complex of given
   Ideal and degree
-- Inputs: F, polynomial ring -- NOT MULTIGRADED
--         I, Ideal in question
--         b, monomial Ideal containing the one monomial degree in
   question
-- Outputs: List of faces in Koszul Simplicial Complex (empty face is
    implicitly included)
-------------------------------------------------------
upperKoszul = method()
upperKoszul(Ring,Ideal,Ideal) := (F,J,b) -> (
    -- establish objects --
    I = substitute(J,F);
    D = {};
    K = {};
    c = numgens F;
    irrelevant = simplicialComplex monomialIdeal gens F; -- empty
   complex --
    -- get all possible faces (square free) --
    for i in 1..c  do D = append(D, squareFree(i,F));

```

```
    -- make sure Rings match --
    d ={};
    C ={};
    e  =flatten(exponents b_0);
    for i in 0..#e-1 do d= append(d, e#i:(generators F)#i);
    for i in 0..#d-1 do for j in 0..#d#i-1 do C= append(C,F_(toString
   (d#i#j)));
    f = product(C);

    -- test if the resulting face (stepped backwards) is in I, add if
    yes, skip if no --
    for i in 0..#D-1 do for j in 0..numgens(D#i)-1 do if liftable(f/D
   #i_j,F) and isSubset(ideal(lift(f/D#i_j,F)),I) then K = append(K,D
   #i_j);
    -- return complex of all faces which passed --
    if K != {} then simplicialComplex K else irrelevant
)
```

The method **lambdaGen** computes all paths between two given monomial degrees. It does this by permuting over all steps between the two monomial degrees. The output is a list of lists, where each individual list takes the form of $\lambda = \{\lambda_1, \ldots, \lambda_\ell\}$. This sequence can be identified with the sequence $\mathbf{b} = \mathbf{b_0}, \mathbf{b_1}, \ldots, \mathbf{b}_{\ell-1}, \mathbf{b}_\ell = \mathbf{a}$ by $\lambda_j = \mathbf{b_{j-1}} - \mathbf{b_j}$ for $j = 1, \ldots, \ell$.

```
-------------------------------------------------------
-- Method description. (lambdaGen):
-- This method computes all paths between two given degrees
-- Inputs: aD, string representing monomial degree a
--         bD, string representing monomial degree b
--         R: polynomial Ring w/grading
-- Outputs: paths, List of lists, each list is a path (of degree
   differentials) between a and b
-------------------------------------------------------
lambdaGen = method()
lambdaGen(String,String,Ring) := (aD,bD,R) ->(
    -- establish objects --
    g = generators R;
    a = monomialIdeal aD;
    b = monomialIdeal bD;

    a1 = substitute(a,R);
    b1 = substitute(b,R);

    -- get difference --
    c = b1_0/a1_0;
```

```
    -- get degree corresponding to monomial c --
    if liftable(c,R) then d = flatten exponents lift(c,R) else d =
    {};
    -- flatten out into single steps in one degree (direction) at a
    time --
    e ={};
    for i in 0..#d-1 do e = append(e, d#i:g#i);
    f = deepSplice(e);
    -- permute to get all possible sequences --
    paths = unique(permutations(f));
    Da = degree a1_0;
    Db = degree b1_0;
    paths1 = apply(paths, j->apply(j,i -> degree i));
    paths1
)
```

The method **pathS** does largely the same thing as the method **lambdaGen**. However, instead of outputting the sequences in terms of λs, this method outputs the sequences $\mathbf{b} = \mathbf{b_0}, \mathbf{b_1}, \ldots, \mathbf{b}_{\ell-1}, \mathbf{b}_\ell = \mathbf{a}$ given in monomials rather than their degrees.

```
-------------------------------------------------------
-- Method description. (pathS):
-- This method computes all paths between two given degrees
-- Inputs: aD, string representing monomial degree a
--         bD, string representing monomial degree b
--         R: polynomial Ring w/grading
-- Outputs: paths, List of lists (of monomials), each list is a path
   (of monomials) between a and b
-------------------------------------------------------
pathS = method()
pathS(String,String,Ring) := (aD,bD,R) ->(
    -- establish objects --
    g = generators R;
    a = monomialIdeal aD;
    b = monomialIdeal bD;

    a1 = substitute(a,R);
    b1 = substitute(b,R);

    -- get difference --
    c = b1_0/a1_0;
    -- get degree corresponding to monomial c --
    if liftable(c,R) then d = flatten exponents lift(c,R) else d =
    {};
```

```
    -- flatten out into single steps in one degree (direction) at a time --
    e ={};
    for i in 0..#d-1 do e = append(e, d#i:g#i);
    f = deepSplice(e);
    -- permute to get all possible sequences --
    paths = unique(permutations(f));
    p= new MutableHashTable;
    for i in 0..#paths-1 do p#i = {};
    for i in 0..#paths-1 do for j in 0..#paths#i-1 do p#i = append(p#i,product(take(paths#i,{0,j})));
    for i in 0..#p-1 do p#i = reverse(apply(p#i, i-> b1_0/i));
    for i in 0..#p-1 do p#i = prepend(a1_0,p#i);
    for i in 0..#p-1 do p#i = append(p#i,b1_0);
    for i in 0..#p-1 do p#i = apply(p#i, j-> lift(j,R));
    for i in 0..#p-1 do p#i = unique(p#i);
    values(p)
)
```

3.2.2. *Iterative Methods.*

The method **Map_1** computes the output of **pathS** and **lambdaGen** for a chosen **a** and **b**. We will use this method to iterate over choices of **a** and **b**.

```
-----------------------------------------------------------
-- Method description. (Map_1):
-- This method is for iterating over degree choices (a,b)
-- Inputs: C, List, C, (List), contains all pairs of degrees (a,b) where a,b have non-zero Betti # in homology degree k-1, k (respectively)
--         i, degree for iterating over elements of C
--         Rs, list with
--             R, polynomial ring w/grading
--             F, polynomial ring w/o grading
--         I, Ideal in question
-- Outputs: Lambda, List of paths (in degrees) from a to b
--          p, List of paths (in monomials) from a to b
-----------------------------------------------------------
Map_1 = method()
Map_1(List,ZZ,List,Ideal):= (C,i,Rs,I) ->(
    -- simply call functions --
    p = pathS(toString(C#i#0#0),toString(C#i#1#0),Rs#0);
    Lambda = lambdaGen(toString(C#i#0#0),toString(C#i#1#0),Rs#0);
    Lambda,p
)
```

The method **Map_2** computes the map between homology vector spaces of degree k and $k-1$, given a pair (**a**, **b**) and a choice of path between them, by calling the appropriate functions. This method calls the sub-methods **Map_3** and **Map_4**. We will iterate over this Map to sum over all maps and over all homology spaces.

```
1  ------------------------------------------------------
2  -- Method description. (Map_2):
3  -- This method is for iterating over path choices (from a to b)
4  -- Inputs: F, polynomial ring w/o grading
5  --         R, polynomial ring w/ grading
6  --         I, ideal in question
7  --         L, List including: (this is necessary bc we can have only
       <=4 total parameters)
8  --           k, homology degree
9  --           j, index of Lambda to use (i.e. which path from a to b
       to use)
10 --           Lambda, List of paths from a to b
11 --           paths
12 -- Outputs: G, Matrix, resulting sylvan matrix
13 ------------------------------------------------------
14 Map_2 = method()
15 Map_2(Ring,Ring,Ideal,List):=(F,R,I,L) ->(
16     -- establish objects --
17     k = L#0;
18     j = L#1;
19     Lambda = L#2;
20     ps = L#3;
21     -- select path --
22     if Lambda != {} then lambda = Lambda#j;
23     if Lambda != {} then l = #lambda ;
24     pth = ps#j;
25     -- calculate Koszul along chosen path --
26     Ks = new MutableHashTable;
27     for i in 0..#(pth)-1 do Ks#i = upperKoszul(F,I,monomialIdeal(pth#
       i));
28     -- get homology in terms of the generators of R --
29     Kb = Ks#(max(keys(Ks)));
30     h = homology(k,Kb,F);
31     hom1 = generators h;
32     -- represent it in terms of all faces in appropriate degree --
33     h2 = faces(k,Kb);
34     h7 = h2*hom1;
35
36     -- based on conditions finish map in 1 of 3 ways --
```

```
    if h !=0 then H = matrix{flatten(entries(faces(k,Kb)))} *
   generators h else H =0;
    if flatten Lambda == {} then G = 0;
    if flatten Lambda != {} and #(ps#j) != 2 then G = Map_3({F,R},I,L
   ,{H,Ks});
    if flatten Lambda != {} and #(ps#j) == 2 then G = Map_4({F,R},I,L
   ,{H,Ks});
    h7,G
)
```

The method **Map_3** is a sub-method of **Map_2** and handles the case in which λ is longer than one step. Specifically, in this case there is a product in the middle of the map in 1.12 that we need to compute.

```
-------------------------------------------------------
-- Method description. (Map_3):
-- This method is for computing the middle and ending part of the Map
    in Theorem 1.11
-- Inputs: Rs, List including:
--            F, polynomial ring w/o grading
--            R, polynomial ring w/ grading
--         I, ideal in question
--         L, List including: (this is necessary bc we can have only
   <=4 total parameters)
--           k, homology degree
--           j, index of Lambda to use (i.e. which path from a to b
   to use)
--           Lambda, List of paths from a to b
--           paths
--         L2, List including:
--             H, Matrix, basis for homology of K^bI
--             Ks, list of Koszul simplicial complexes along the path
    lambda
-- Outputs: G, Matrix, resulting sylvan matrix
-------------------------------------------------------
Map_3 = method()
Map_3(List,Ideal,List,List):= (Rs,I,L,L2) -> (
    -- initialize objects --
    F = Rs#0;
    R = Rs#1;
    k = L#0;
    j = L#1;
    Lambda = L#2;
    ps = L#3;
    H = L2#0;
```

```
    Ks = L2#1;
    -- select path --
    lambda = Lambda#j;
    l = #lambda ;
    -- following the formula in Thm 1.11, calculate middle part of
   map--
    H1 = flatten entries H;
    J = new MutableHashTable;
    -- iteratively compute delPlus(dLambda(basis for homK^bI)) for j
   =1,..ell --
    for i in 1..#(ps#j)-2 do for j in 0..#H1-1 do J#i = delPlus(R,(
   hedge_trimmer(Ks#(i),k),Ks#(i),k,(dLambda(R,lambda,{i,k},matrix{{
   H1#j}}))));
    -- take product
    P = product(values(J));
    D = 1_F;
    scan(0..#J-1, i -> D = D *J#((keys(J))#i));
    -- ind = ell --
    ind = -1;
    -- calc last step in map dLambda_ell --
    try U = dLambda(R,lambda,{ind,k},matrix{{D}}) then  U = dLambda(R
   ,lambda,{ind,k},matrix{{D}}) else U = 0;
    -- get basis for homK^aI --
    Ka = Ks#(min(keys(Ks)));
    hom = homology(k-1,Ka,F);
    hom1 = generators hom;
    -- represent basis in terms of all faces of appropriate degree --

    if (flatten entries U)#0 != -1 and (flatten entries U)#0 != 1 and
    (flatten entries U)#0 != 0 then ret = Helper(U,Ka) else ret = (U,
   hom1);
    ret)
)
```

The method **Map_4** is a sub-method of **Map_2** and handles the case in which λ is only one step. Or, in this case there is not a product in the middle of the map in 1.12 that we need to compute.

```
-----------------------------------------------------
-- Method description. (Map_4):
-- This method is for computing the ending part of the map in Theorem
    1.11 (when there is no middle map to compute)
-- Inputs: Rs, List including:
--            F, polynomial ring w/o grading
--            R, polynomial ring w/ grading
```

```
--          I, ideal in question
--          L, List including: (this is necessary bc we can have only <=4 total parameters)
--            k, homology degree
--            j, index of Lambda to use (i.e. which path from a to b to use)
--            Lambda, List of paths from a to b
--            paths
--          L2, List including:
--              H, Matrix, basis for homology of K^bI
--              Ks, list of Koszul simplicial complexes along the path lambda
-- Outputs: G, Matrix, resulting sylvan matrix
------------------------------------------------------------
Map_4 = method()
Map_4(List,Ideal,List,List):= (Rs,I,L,L2) -> (
    -- initialize objects --
    F = Rs#0;
    R = Rs#1;
    k = L#0;
    j = L#1;
    Lambda = L#2;
    ps = L#3;
    H = L2#0;
    Ks = L2#1;
    -- select path --
    lambda = Lambda#j;
    l = #lambda ;
    -- last index --
    ind = -1;
    -- calc dLambda_ell --
    U = dLambda(R,lambda,{ind,k},H);
    -- get basis for homology of K^aI --
    Ka = Ks#(min(keys(Ks)));
    hom = homology(k-1,Ka,F);
    hom1 = generators hom;

    if (flatten entries U)#0 != -1 and (flatten entries U)#0 != 1 and (flatten entries U)#0 != 0 then ret = Helper(U,Ka) else ret = (U, hom1);
    ret)
)
```

The **Helper** method finishes off the sylvan map computation by expressing the resulting chain as a linear combination of elements in the basis for homology in degree

a. We split this off from the above methods as it's sometimes not necessary (this is often the case for k=0).

```
-------------------------------------------------------------
-- Method description. (Helper):
-- This method finishes the computation of the sylvan map by
   expressing the result of the mappings as a linear comb. of
   elements in the basis for homology
-- Inputs: U, matrix, output of sylvan mappings
--         Ka, Kozsul simplicial complex in degree a
-- Outputs: sylvan map, basis elts for homK^aI
-------------------------------------------------------------
Helper =method()
Helper(Matrix,SimplicialComplex) := (U,Ka) -> (
    -- transform result from dLambda into coefs --
    U2 = listForm((flatten entries U)_0);
    U3 = apply(U2, i-> i#1*i#0);
    U4 = sum(U3);
    -- represent basis in terms of all faces in appropriate degree (i
   .e. x,y,z instead of y,z)--
    h2 = faces(k-1,Ka);
    h3 = h2*hom1;
    h4 = apply(flatten entries h3, i->listForm i);
    h5 = apply(h4, j-> apply(j, i->i#1*i#0));
    h6 = apply(h5, k->sum(k));
    -- lift coefs into same ring --
    H2 = lift(matrix(h6),ZZ);
    U5 = lift(matrix{U4},ZZ);
    -- solve lin eq --
    if U != 0 then G = solve(transpose(H2),transpose(U5)) else G =0;
    G,h3
)
```

The method **sylvanMap** iterates over all choices of **a** and **b**, computing the non-canonical sylvan homomorphism between homology vector spaces k and $k-1$.

```
-------------------------------------------------------
-- Method description. (sylvanMap):
-- This method iterates over all choices (a,b) and choices of paths
   between a and b
-- Inputs: R, polynomial ring w/grading
--         F, w/o grading
--         I, ideal
--         k, homology degree
-- Outputs: : Q4, Hashtable where Q4#(degree a, degree b) = (basis
   elts for homK^aI, map between homology, basis elts for homK^bI)
```

```
-----------------------------------------------------
sylvanMap = method()
sylvanMap(Ring,Ring,Ideal,ZZ):= (R,F,I,k)-> (
    -- initialize objects --
    Q = new MutableHashTable;
    Q2 = new MutableHashTable;
    Q3 = new MutableHashTable;
    Q4 = new MutableHashTable;
    -- get Bettis --
    Cs = bettiDs(F,I,k);
    -- get paths between a,b --
    for i in 0..#Cs-1 do for j in 0..(#((Map_1(Cs,i,{R,F},I))#0))-1
   do Q#(i,j,Cs#i) = Map_1(Cs,i,{R,F},I);
    -- calculate sylvanMap between each a,b for each lattice path --
    for ii in keys(Q) do Q2#ii= Map_2(F,R, I,{k, ii#1, Q#ii#0, Q#ii
   #1});
    -- sum over different lattice paths --
    for k in 0..#Cs-1 do Q3#k = sum(apply(delete(null,apply(keys(Q2),
    i -> if i#0 == k then i else null)), j-> (j,Q2#j)));
    -- organize into a final output hashTable
    for ii in values(Q3) do Q4#(take(ii#0,{2,3})) = ii#1;
    for ii in keys(Q4) do Q4#ii = deepSplice Q4#ii;
    -- remove pairs w/o a path --
    for ii in keys(Q4) do if #(Q4#ii)== 2 then remove(Q4,ii);
    for ii in keys(Q4) do Q4#ii = reverse(Q4#ii);
    Q4
)
```

The method **sylvanResolution** iterates over **sylvanMap** to get the entire resolution.

```
-----------------------------------------------------
-- Method description. sylvanResolution():
-- This function computes the entire sylvan resolution (I.e. computes
    the differentials for all homology degrees)
-- Inputs: R, multigraded polynomial ring
--         I, ideal
-- Outputs:  SR, List, where H#i = output Hashtable from sylvanMap(R,
   F,I,k)
-----------------------------------------------------
sylvanResolution = method()
sylvanResolution(Ring,Ring,Ideal):=(R,F,I) ->(
     -- iterate over all homology degrees in the resolution --
    -- By Hilbert Syzygy Thm we know the resolution is at most of
   length N (as computed below) --
```

```
    N = numgens R;
    SR ={};
    for i in 0..N-1 do SR = append(SR,(i,sylvanMap(R,F,I,i)));
    SR
)
```

3.3. **Code examples.**

In this section we provide example code to demonstrate how to use our methods, as described and enumerated above. Note that some of the output may be edited here slightly for ease of display.

We first demonstrate how to compute $K^{\mathbf{b}}I$, where $b = 121$.

```
i0: needsPackage "SimplicialComplexes.m2"
i1: needsPackage "Posets.m2"
i2: needsPackage "RandomIdeals.m2"
i3: F = QQ[x,y,z];
i4: I = monomialIdeal(x*y,y*z,x*z,y^3,z^2);
i5: b = monomialIdeal(x*y^2*z);
i6: Kb = upperKoszul(F,I,b)
o6: SimplicialComplex{yz,xy}
o6: SimplicialComplex
```

Say we'd now like to compute ST_i where $i = 1$. We can do the following.

```
i7: ST = hedge_trimmer(Kb,1)
o7: ({xy,yz},{z,y})
o7: Sequence
```

Recall the computation in 2.2, $\partial^+_{ST_1}(z-x) = yz + xy$. Here ST_1 is our hedge as just calculated in the previous 2 lines of code. We can now do the computation $\partial^+_{ST_1}(z-x)$ as follows.

```
i8: R = QQ[x,y,z,Degrees=>{{1,0,0},{0,1,0},{0,0,1}}];
i9: delPlus(R,(ST,Kb,1,matrix{{z-x}}))
o9: xy+yz
o9: QQ[x..z]
```

Say now we'd like to compute $\Lambda(a, b)$. This can easily be done using the following method.

```
i10: a = "xyz";
i11: b = "xy3z";
i13: Lambda = lambdaGen(a,b,R)
o13: {{{0,1,0},{0,1,0}}}
o13: List
```

Related is the command below, which enumerates the entire path of monomials.

```
i10: a = "xyz";
i11: b = "xy3z";
i13: Ps = pathS(a,b,R)
o13: {{xyz,xy^{2}z,xy^{3}z}}
o13: List
```

The computation $d_1^{\lambda_j}(yz - xz + xy)$ is similarly simple to execute. The following lines of code suffice.

```
i14: lambda = Lambda#0;
i15: dLambda(R,lambda,{1,1},matrix{{y*z -x*z + x*y}})
o15: (  x  +z)
o15: Matrix F1 <- F1
```

Now, suppose we want to find all pairs (a, b) such that the nonzero Betti number for a is in homology of lower degree than that of b, and $b \succ a$. We can do this using the following methods.

```
i16: bettiDs(F,I,1)
o16: {(xyz,xy^3z),(xyz,xyz^2),(y^3,xy^3z),(y^3,xyz^2),(xz^2,xy^3z)
  ,(xz^2,xyz^2),
    (xy^3,xy^3z),(xy^3,xyz^2),(yz^2,xy^3z),(yz^2,xyz^2)}
o18: List
```

And lastly, we can do the following to compute the sylvan homomorphism itself.

```
i19: SylMap = sylvanMap(R,F,I,1)
o19: MutableHashTable {...6...}
i20: peek SylMap
o20: MutableHashTable{
(x*y*z, x*y*(z^2))) =>
(|  - x + y  - x + z  |, | -1|,|  x - y + z  | )
                          | 0 |

 (x*y*z, x*(y^3)z)) =>
 ( |  - x + y  - x + z  |, |  0  |, |  x - y + z  |)
                           |  1  |

 (x*z^2, x*y*z^2)) =>
 ( |  - x + z  |, |  1  |,|  x - y + z  |)

 (y*z^2, x*y*z^2)) =>
 ( |  - y + z  |, |  1  |,|  x - y + z  |)

 (x*y^3, x*y^3z)) =>
 ( |  - x + y  |, |  -1  |,|  x - y + z  |)

 (y^3z, x*y^3z)}) =>
```

```
23   ( |  - y + z  |, |  1  |,|  x - y + z  |)
```

Recall that in the final output, the key of the hash table encodes the degrees a,b, and the values encodes the basis for homology of K^aI, the map itself, and the basis for homology of K^bI appears last.

If desired, we can skip all the above steps and simply cut to the chase by calling the below method to calculate the entire resolution.

```
1    i20: sylvanResolution(R,F,I)
2    020: {(0, MutableHashTable {...11...}),(1, MutableHashTable
     {...6...}),(2, MutableHashTable {})}
3    o20: List
```

This first list entry is what we see as output on line o20.

4. Future directions

We hope to formulate strategies for avoiding the enumeration of every lattice path, having ultimate efficiency in mind. While there may be arbitrarily many of these for a given ideal, we believe reducing the lattice paths to a certain fixed number of cases ("types" of lattice paths) we will be able increase efficiency of our algorithm.

We hope to implement a very similar algorithm for Yuzvinsky's resolution of a monomial ideal [Yuz99] upon the completion of an analogous combinatorial formula for the mappings in this resolution.

References

[M2] Grayson, Daniel R. and Stillman, Michael E., *Macaulay2, a software system for research in algebraic geometry*, Available at http://www.math.uiuc.edu/Macaulay2/.

[E020] Edelsbrunner, H. and Ölsböck, K., *Tri-partitions and Bases of an Ordered Complex.*, Discrete Comput Geom 64, 759–775 (2020) https://doi.org/10.1007/s00454-020-00188-x.

[EMO20] John Eagon, Ezra Miller, and Erika Ordog, *Minimal resolutions of monomial ideals*, (submitted), 2020. arXiv:1906.08837 (v2).

[Yuz99] Yuzvinsky, Sergey., *Taylor and Minimal Resolutions of Homogeneous Polynomial Ideals. Mathematical Research Letters.*, (1999). 6.10.4310/MRL.1999.v6.n6.a14.

[SA10] Shifrin, Theodore and Adams, Malcolm, *Linear Algebra: A Geometric Approach*, W. H. Freeman; Second edition (July 30, 2010).

[CoCoA] Abbott, J. and Bigatti, A. M., *CoCoALib: a C++ library for doing Computations in Commutative Algebra*, Available at http://cocoa.dima.unige.it/cocoalib.

[Hoc77] Melvin Hochster, *Cohen–Macaulay rings, combinatorics, and simplicial complexes*, Ring theory, II, Lecture notes in pure and applied mathematics **26** (1977), 171–223.