Utrecht University

TWIN-Bachelorthesis

The Hilbert scheme of points

Nicolas Rühling

supervised by
Dr. Martijn Kool

June 13, 2018

Contents

Introduction

The Hilbert Scheme of points $\mathrm{Hilb}^n(\mathbb{C}^d) = H_n^d$ is a so-called parameter or moduli space where a point in H_n^d is determined by a configuration of n points in $\mathbb{C}^d$. It is a fundamental concept in algebraic geometry and finds applications in a wide range of mathematics, among others string theory, the theory of symmetric functions[1] knot theory. In this thesis we will study H_n^d using algebraic as well as combinatorial methods. We will mostly treat the case when those points lie in the complex plane, thus when $d = 2$, to give examples and provide intuition about its geometry. This can be seen as a special case since H_n^2 is a smooth and irreducible variety of dimension $2n$ (Thm. 17). Another main result is that H_n^d is connected for arbitrary d (Prop. 11).
The structure of the thesis follows Chapter 18 from [MS05], in particular sections 18.1, 18.2 and 18.4. That means our focus lies on points in the Hilbert scheme that correspond to monomial ideal which allows us to apply combinatorial methods. However, we deviate in some occasions. We skip the formal construction of the Hilbert scheme as a quasiprojective variety.

[1] We highlight in particular the work of Mark Haiman who used the Hilbert scheme of points to prove the Macdonald positivity conjecture for Macdonald polynomials. See section 18.3 in [MS05] for an introduction and further references.

In section 1.2 we show what it means for a scheme to be flat; an essential notion in the general theory of Hilbert schemes. In section 2 we provide an explicit method to find a *rational curve* inside the Hilbert scheme of points. In section 3 we connect the theory of Haiman arrows from [MS05] to a more general result about the tangent space of the Hilbert scheme of points (Thm. 14) and use this to demonstrate the notion of a *module homomorphism*. Furthermore we have implemented the theory of Haiman arrows in Python to compute the tangent space at a monomial ideal in the Hilbert scheme over $\mathbb{C}^3$ or $\mathbb{C}^4$. An explanation of the program and a short example on how to use it are given in section 4. The full code can be found in the Appendix in Listings 4, respectively 5. In section 5 we present some experimental results found with the help of the program.
As prerequesites some basic knowledge of commutative algebra and algebraic geometry is recommended. A nice introduction to these subjects on undergraduate level can be found in [CLO97]. However, anyone who has studied ring theory before will be able to follow through most of the text. A more advanced, but general, treatment of commutative algebra can be found in [Eis95] and [AM16].

1 Ideals of points in the plane

Let $\mathbb{C}[x, y]$ be the ring of all polynomials in x and y. As a set we can define H_n^2 as follows.

$$H_n^2 = \{I \subseteq \mathbb{C}[x, y] \mid I \text{ ideal with } \dim_{\mathbb{C}}(\mathbb{C}[x, y]/I) = n\}.$$

That is H_n^2 contains all the ideals I such that the quotient ring $\mathbb{C}[x, y]/I$ has dimension n as a vector space over $\mathbb{C}$.
First we want to get a feeling for how the points of the Hilbert scheme look like. When we have n points $P_1 = (x_1, y_1), P_2, ..., P_n \in \mathbb{C}^2$ that are all different from each other, the corresponding ideal I of functions vanishing on these points is generated by products of $(x - x_i)$ and $(y - y_i)$ for $1 \leq i \leq n$. This is a radical ideal and therefore the set $\{P_1, ..., P_n\}$ is a classical algebraic variety.
On the other hand some or all of the points could overlap. The corresponding ideal I is then said to carry a non-reduced scheme structure. The most "special" case of non-reduced schemes is when we only consider the origin

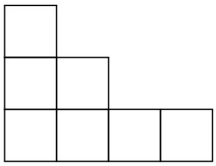
with multiplicity n. The corresponding ideal will then be generated by monomials and that allows us to use combinatorial methods. We will present the combinatorial background first and then briefly explain what it means for a scheme to be flat.

1.1 Combinatorics and monomial ideals

We will start off with some definitions.

Definition 1. A **monomial** in $\mathbb{C}[x, y]$ is a product $x^a y^b$ with $a, b \in \mathbb{N}$. An ideal $I \subseteq \mathbb{C}[x, y]$ is called a **monomial ideal** if it is generated by monomials.

Definition 2. A partition λ of n is a nonincreasing list of positive numbers $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_k$ such that $\sum \lambda_i = n$ and $\lambda_k \neq 0$. A Young diagram is a box diagram such that starting from the bottom the i-th row consists of λ_i boxes and the rows are left aligned. For example the diagram

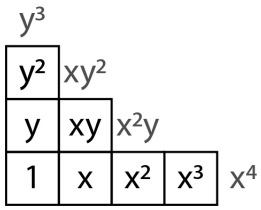


corresponds to the partition $4 + 2 + 1$ of 7.

We can give the boxes coordinates $\mathbf{a} \in \mathbb{N}^2$ to make a distinction between boxes in λ and boxes outside of λ. We can further make a connection between monomial ideals and partitions and use that connection to represent the monomial ideals pictorally.
Recall that per definition if we take any element $x \in I$, with $I \subseteq \mathbb{C}[x, y]$ an ideal, and multiply with an arbitrary polynomial $f \in \mathbb{C}[x, y]$ we again get an element of the ideal, thus $xf \in I$. Considering the coordinates $\mathbf{a} = (a_1, a_2)$ as exponents of a monomial $x^{a_1} y^{a_2}$, boxes outside of λ correspond to monomials in I_λ and boxes inside of λ span the $\mathbb{C}$-vector space of the quotient $\mathbb{C}[x, y]/I_\lambda$.

Example 3. Take a look at the partition $4 + 2 + 1$ of $n = 7$ again. The ideal is $I_{4+2+1} = \langle x^4, x^2 y, xy^2, y^3 \rangle$. The ideal corresponds to boxes to the top and to the right of the box diagram below. As a $\mathbb{C}$-vector space the quotient

$\mathbb{C}[x,y]/I$ is spanned by $\{1, x, x^2, x^3, y, yx, y^2\}$. These are exactly the boxes of the partition.



Remark 1. When d is arbitrary the terminology slightly changes. In three dimensions we speak of *plane partitions* or *staircase diagrams*. In higher dimensions of *multipartitions*.

1.2 Flat families and flat limits

An important concept in algebraic geometry and in the theory of Hilbert schemes is that of flatness. As discussed in the beginning, any point $I \in H_n^2$ corresponds to a configuration of n points in $\mathbb{C}^2$. This is the essence of a *flat family*. We will not delve further into the algebraic geometry background, but demonstrate the concept of a flat family for an easier example. For more information about the general definition and proof of existence of the Hilbert scheme we refer the interested reader to [Gro61] and [Leh04].
Let $B = \mathrm{Spec}(\mathbb{C}[t])$ be the affine line in the variable t and consider the following map

$$\begin{array}{c} Z \subseteq \mathbb{C}^2 \times B \\ \big\downarrow \\ B\,. \end{array}$$

Since B is an algebraic variety, it holds that Z is B-flat if and only if for all closed points $t \in B$, the length of Z_t is constant. We further say that B *parametrizes* Z. In the language of ideals the above translates to

$$Z \text{ is } B-\text{flat} \iff \dim_{\mathbb{C}}(\mathbb{C}[x,y][t]/I_{Z_t}) = \text{constant}, \forall t \in \mathbb{C}[t]\,.$$

Example 4. This time we will consider three points approaching each other. We let one point be fixed at the origin and let the other two move along the axes. Our points are

$$(0,0) \qquad (t,0) \qquad (0,t)\,.$$

The corresponding ideals are

$$\langle x, y\rangle \qquad \langle x-t, y\rangle \qquad \langle x, y-t\rangle\,.$$

The ideal of the union of the three points is given by the product of the ideals

$$\langle x^3 - x^2t, x^2t + xyt - xt^2, x^2y, xyt, x^2y - xyt, xyt + y^2t - yt^2, xy^2, y^3 - y^2t\rangle\,.$$

We look at the limit as t goes to zero. If we just set $t = 0$ we get the following ideal

$$\langle x^3, x^2y, xy^2, y^3\rangle$$

which is a colength 6 ideal and so the product is not a flat family. In fact it follows from Lemma 8 that the flat limit at the origin should be

$$\langle x^2, xy, y^2\rangle\,,$$

which is indeed an ideal of colength 3. This flat limit is an example of a non-reduced scheme. We can interpret it as carrying infinitesimal information about the direction where the points came from. It is interesting to note that the Young diagram of the partition corresponding to the flat limit resembles the initial configuration of points. We will make this more precise in Lemma 8.

2 Connectedness

While in general not much is known of H_n^d, there is one "nice" property it possesses: connectedness. In this section we will prove this. Although we only deal with the case $d = 2$, the proof holds for arbitrary d.
The general idea is that we can connect any ideal in H_n^2 to a monomial ideal by a rational curve inside the Hilbert scheme. It follows then that to every

partition belongs a configuration of n points in $\mathbb{C}^2$. By moving these points continuously we can go from one monomial ideal to another and thus connect any two ideals.
First we will provide a method to "degenerate" an ideal I such that we get a *rational curve* $\widetilde{I}_t$ inside the Hilbert scheme that connects I to its initial ideal $\widetilde{I}_0$.
Let $\omega : \mathbb{Z}^2 \to \mathbb{Z}$ be an integral weight function. For convenience of notation we think of ω as a function on monomials, and if $m = x^a$, we write $\omega(m) \in \mathbb{Z}$ in place of $\omega(a)$. Given $f \in \mathbb{C}[x, y]$ we write $\text{in}_\omega(f)$ for the sum of all terms of f that are maximal with respect to ω. If I is an ideal we write $\text{in}_\omega(I)$ for the ideal generated by $\text{in}_\omega(f)$ for $f \in I$. Since ω is a partial order this does not necessarily give us a monomial ideal. Let $\mathbb{C}[x, y][t]$ be a polynomial ring in the variable t over $\mathbb{C}[x, y]$ (the coefficients are polynomials in x and y). For any $f \in \mathbb{C}[x, y]$ we define $\widetilde{f} \in \mathbb{C}[x, y][t]$ as follows. Write $f = \sum c_i m_i$, with m_i the monomials and $0 \neq c_i \in \mathbb{C}$. Let $b = \max \omega(m_i)$, and set

$$\widetilde{f} = t^b f(t^{-\omega(x)}x, t^{-\omega(y)}y) \, .$$

We see that $\widetilde{f}$ is $\text{in}_\omega(f)$ plus t times a polynomial in t, x and y. Setting t to zero thus gives us the the initial term of f with respect to the weight function ω. For any ideal $I \subset \mathbb{C}[x, y]$ let $\widetilde{I}_t$ be the ideal in $\mathbb{C}[x, y][t]$ generated by $\{\widetilde{f} \mid f \in I\}$. Then it follows that $\mathbb{C}[x, y][t]/(\langle t \rangle + \widetilde{I}_t) \cong \mathbb{C}[x, y]/\text{in}_\omega(I)$.

Remark 2. With the above procedure the ideal $\widetilde{I}_0$ is not necessarily monomial since we can have ties between two or more terms. However, according to Exercise 15.12 in [Eis95] there exists an integral weight function such that we can simulate a total order for a finite amount of pairs of monomials. That means if we have an ideal $I = \langle f_1, ..., f_m \rangle$ with a finite amount of generators we can always find an integral weight function that will give us a monomial ideal $\widetilde{I}_0$ which corresponds to the initial ideal of I with respect to the total order. In that case it further follows that $\widetilde{I}_t$ is generated by the $\widetilde{f}_1, ..., \widetilde{f}_m$ (see Lemma 5).

Lemma 5. *Let $<$ be a total monomial order. If $f_1, ..., f_m \in I$ are chosen so that $in_<(f_1), ..., in_<(f_m)$ generate $in_<(I)$, then $\widetilde{f}_1, ..., \widetilde{f}_m$ generate $\widetilde{I}$.*

Proof. It is evident that $\langle \widetilde{f}_1, ..., \widetilde{f}_m \rangle \subseteq \widetilde{I}$. Furthermore, for any $t \in \mathbb{C}$, the leading terms of the $\widetilde{f}_i$ (as polynomials of x and y) with respect to the term order $<$ are the generators of the colength n ideal $\text{in}_<(I)$. Thus colength

$(\langle \widetilde{f}_1, ..., \widetilde{f}_m \rangle) \leq n$. It is sufficient to check this pointwise for every t and therefore

$$\widetilde{I} = \langle \widetilde{f}_1, ..., \widetilde{f}_m \rangle .$$

□

Lemma 6. *Every point $I \in H_n$ is connected to a monomial ideal by a rational curve.*

Proof. We first choose a a total monomial order $<$. The polynomial ring is Noetherian. Thus every ideal $I \in \mathbb{C}[x, y]$ is finitely generated and we can apply the above procedure (see Remark 2). This gives a flat family $\widetilde{I}_t$ of ideals parametrized by t. The points $\widetilde{I}_t$ form a rational curve[2] which lies completely in H_n. When we set $t = 1$ we get back the original ideal and when $t = 0$ we get the initial ideal, which is a monomial ideal. The flatness follows from Theorem 15.17 in [Eis95]. □

Example 7. Take $I = \langle x^2 - xy, xy^2, y^3 - y^2 \rangle$ and choose lexicographic order[3]. We find the following weight function $\omega = (2, 1)$ which simulates the monomial order for I. Applying ω to the first generator $f_1 = x^2 - xy$, we get

$$\begin{aligned} \omega(x^2) &= 4 \\ \omega(xy) &= 3 , \end{aligned}$$

thus $\widetilde{f}_1 = t^4(t^{-4}x^2 - t^{-3}xy) = x^2 - txy$. The other generators become

$$\begin{aligned} \widetilde{f}_2 &= xy^2 \\ \widetilde{f}_3 &= y^3 - ty^2 . \end{aligned}$$

It follows that the rational curve is given by

$$\widetilde{I}_t = \langle x^2 - txy, xy^2, y^3 - ty^2 \rangle .$$

For the limit $t = 0$ we have $\widetilde{I}_0 = \langle x^2, xy^2, y^3 \rangle$ which is indeed the initial ideal of I with respect to lexicographic order and lies in H_5.

[2] Also called a Gröbner degeneration

[3] That is $1 < x < xy < ... < x^2 < x^2y < ... < x^3 < ...$

The next Lemma is a preparation for Lemma 10 and states an interesting fact about the relation between partitions, monomial ideals and radical ideals.

Lemma 8. *There is a bijection between partitions λ of n and monomial ideals $I \subseteq \mathbb{C}[x,y]$ of colength n. Further given a partition λ and considering the exponent vectors (h,k) on monomials $x^h y^k$ outside I_λ as n points in $\mathbb{N}^2 \subseteq \mathbb{C}^2$ gives us a radical ideal I'_λ whose flat limit at the origin is I_λ for every term order.*

Proof. The first statement is evident from looking at the figures given in section 1.1. The radical ideal I'_λ is called the *distraction* of I_λ. Suppose $I_\lambda = \langle x^{a_1} y^{b_1}, ..., x^{a_m} y^{b_m} \rangle$ and consider the polynomials

$$f_i = x(x-1)(x-2)\cdots(x-a_i+1)y(y-1)\cdots(y-b_i+1)\,.$$

First we prove that the f_i generate I'_λ. We have $\langle f_1, ..., f_m \rangle \subseteq I'_\lambda$ because the polynomials f_i vanish at the given points (a_j, b_j) and we have colength $(\langle f_1, ..., f_m \rangle) \leq n$ because the leading terms of the f_i are the generators of the colength n ideal I_λ. Therefore

$$I'_\lambda = \langle f_1, ..., f_m \rangle\,.$$

From the definition above we further see that when expanding f_i there will be a term $x^{a_i} y^{b_i}$. We claim that this is the initial term of f_i for every term order. Then it is easy to see that any Gröbner degeneration $(I'_\lambda)_t$ constructed as above will have the flat limit $(I'_\lambda)_0 = I_\lambda$.
To prove the claim assume that there was a term $x^r y^s$ of f_i and a term order where $x^r y^s > x^{a_i} y^{b_i}$. From the way we defined f_i, it must hold that r and s are smaller or equal than a_i and b_i and that at least one of them is strictly smaller. Per definition of a term order $'1'$ is the least element and multiplying by the same monomial on both sides preserves the order. Consider

$$1 < x^{a_i - r} y^{b_i - s}\,,$$

multiplying by $x^r y^s$ gives

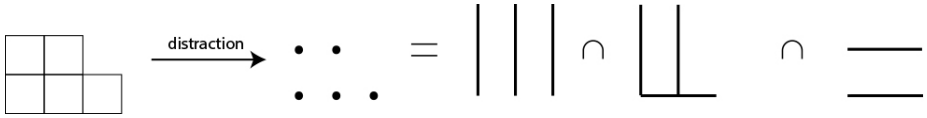
$$x^r y^s < x^{a_i} y^{b_i}\,,$$

which is a contradiction. This proves the claim. □

Example 9. Take the partition $\lambda = 3 + 2$. The ideal is $I_{3+2} = \langle x^3, x^2y, y^2 \rangle$ where the elements of I_{3+2} correspond to the boxes outside the partition. Clearly I_{3+2} lies in H_5. The distraction is given by

$$I'_{3+2} = \langle x(x-1)(x-2), x(x-1)y, y(y-1) \rangle .$$

The zero set of each generator is a union of lines, the zero set of the distraction is the intersection of these lines and it can easily be seen that this gives a set of five points. Our radical ideal I'_{3+2} thus lies in H_5 as well. Note the similarity of the partition and the point configuration of the distraction below.



Lemma 10. *For every partition λ of n, the point $I_\lambda \in H_n$ lies in the closure of the locus $(S^n\mathbb{C}^2)^\circ$ of all radical ideals in the Hilbert scheme H_n.*

Proof. Take the distraction I'_λ. The ideal $(I'_\lambda)_t$ constructed as in the proof of Lemma 6 is radical for each $t \neq 0$. Hence $I_\lambda = (I'_\lambda)_0$ lies in the closure of $(S^n\mathbb{C}^2)^\circ$. □

Proposition 11. *The Hilbert scheme H_n is connected*

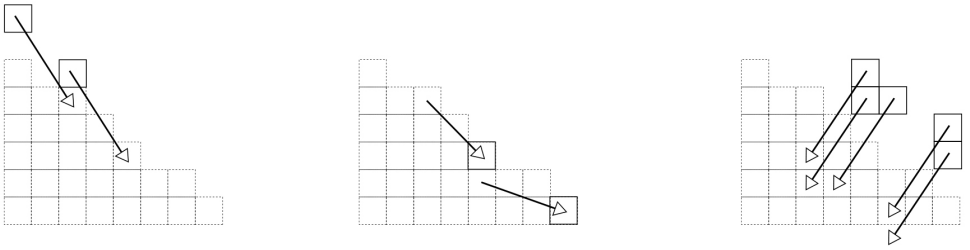
Proof. Take any two points I and J in H_n. We go from I to its initial monomial ideal I_λ and then to its distraction I'_λ. Similarly we go from J to its initial monomial ideal J_μ and then to its distraction J'_μ. Now I'_λ and J_μ are radical ideals of n points in $\mathbb{C}^2$ and we connect the two ideals by continuously moving one point configuration into the other. □

3 The tangent space of H_n^d and Haiman arrows

In this section we will show how one can calculate the tangent space at points of H_d^n that correspond to monomial ideals. The following theory has been developed by Mark Haiman and makes use of combinatorial methods. We will again start with the case $d = 2$ but this holds for arbitrary d.

Definition 12. Given a partition λ, a *Haiman arrow* is an arrow whose tail lies in a box $rs \notin \lambda$ and whose head in a box $hk \in \lambda$. The head is allowed to lie in boxes with $h < 0$ or $k < 0$ but those arrows are set to zero. Further it is permitted to translate arrows vertically or horizontally as long as the head stays in the partition and the tail outside of the partition. Two arrows that can be moved into each other belong to the same equivalence class and are considered equal.

Example 13. The following three box diagrams all represent the same partition: $\lambda = 8 + 7 + 5 + 4 + 3 + 1$. The two arrows in the left diagram lie in the same equivalence class. The middle diagram shows two different and valid arrows. The arrows in the right diagram are all equal but zero, since the head can cross the axis while the tail stays outside the partition.



For a derivation of the Haiman arrows see [MS05]. The interesting part of the above definition is that the number of different arrows (or equivalence classes) not set to zero is equal to the dimension of the tangent space at a point in the Hilbert scheme. This follows directly from the derivation given in [MS05]. In what follows we will connect the theory of Haiman arrows to another, more general result about the Hilbert scheme of points and its tangent space.

Theorem 14. *The tangent space of* $Hilb^n(\mathbb{C}^d)(= H_d^n)$ *is isomorphic as* $\mathbb{C}$*-vector space to* $Hom_{\mathbb{C}[\vec{\mathbf{x}}]}(I, \mathbb{C}[\vec{\mathbf{x}}]/I)$.

Here we denote by $\mathrm{Hom}_{\mathbb{C}[\vec{\mathbf{x}}]}(I, \mathbb{C}[\vec{\mathbf{x}}]/I)$ the space of all $\mathbb{C}[\vec{\mathbf{x}}]$-module homomorphisms $I \to \mathbb{C}[\vec{\mathbf{x}}]/I$. For a proof of the Theorem see [Leh04].
Let I be a point in H_d^n. It is sufficient to look at arrows starting at minimal generators of I. Denote an arrow by $c_{\mathbf{v}}^{\mathbf{u}}$ with $\mathbf{v}, \mathbf{u} \in \mathbb{N}^d$ exponent vectors of monomials such that $\mathbf{x}^{\mathbf{v}} \notin I$ and $\mathbf{x}^{\mathbf{u}} \in I$. By abuse of notation we will let $c_{\mathbf{u}}^{\mathbf{v}}$ also be a function $I \to \mathbb{C}[\vec{\mathbf{x}}]/I$. We will initiate the function as follows

$$c_{\mathbf{u}}^{\mathbf{v}}(\mathbf{x}^{\mathbf{u}}) = \mathbf{x}^{\mathbf{v}}$$

Since we want this function to be a $\mathbb{C}[\vec{\mathbf{x}}]$-module homomorphism, it must hold that

$$c^{\mathbf{u}}_{\mathbf{v}}(af) = a \cdot c^{\mathbf{u}}_{\mathbf{v}}(f)\,,$$

for all $f \in I$ and $a \in \mathbb{C}[\vec{\mathbf{x}}]$). Using this condition we can extend $c^{\mathbf{u}}_{\mathbf{v}}$ linearly to all of I. Further when extending to $\mathbf{x}^{\mathbf{w}}$ we need to check whether

$$\mathbf{w}_i + \mathbf{v}_i < \mathbf{w}_i \text{ for any } 1 \leq i \leq d\,, \tag{1}$$

because that means the head of the arrow has crossed one of the axes and in that case $c^{\mathbf{v}}_{\mathbf{u}}(\mathbf{x}^{\mathbf{w}}) = 0$.

As stated above the number of equivalence classes of Haiman arrows gives the dimension of the tangent space at a point in the Hilbert scheme. We give the following result without proof but provide two examples of functions $c^{\mathbf{v}}_{\mathbf{u}}$: one that corresponds to a valid arrow, and one that does not.

Proposition 15. *$Hom_{\mathbb{C}[\vec{\mathbf{x}}]}(I, \mathbb{C}[\vec{\mathbf{x}}]/I)$ has a $\mathbb{C}[\vec{\mathbf{x}}]$-basis given by the $\mathbb{C}[\vec{\mathbf{x}}]$-module homomorphisms $c^{\mathbf{u}}_{\mathbf{v}}$ that correspond to Haiman arrows.*

Example 16. Consider the ideal $I = \langle x^2, xy, y^2\rangle$. As a $\mathbb{C}$-vector space the quotient $\mathbb{C}[x,y]/I$ is spanned by $\{1, x, y\}$. There are six different arrows according to Definition 12: c^{02}_{01}, c^{02}_{10}, c^{11}_{01},c^{11}_{10},c^{20}_{01} and c^{20}_{10}.
The function representing the arrow from y^2 to y is c^{02}_{01}. Per definition $c^{02}_{01}(y^2) = y$. We can extend this to other elements of I, e.g. xy^2

$$c^{02}_{01}(xy^2) = x \cdot c^{02}_{01}(y^2) = x \cdot y = 0 \text{ mod I},$$

but also

$$c^{02}_{01}(xy^2) = y \cdot c^{02}_{01}(xy).$$

It must follow that $c^{02}_{01}(xy) = 0$. This coincides with the theory of Haiman. Starting from y^2 it is not possible to move the tail of the arrow to xy. Therefore the function takes the value 0 in the point xy. We can apply the same reasing to xy^2. It is not possible to move the tail of the arrow to that point because then the head would lie in xy which is outside the partition. So $c^{02}_{01}(xy^2) = 0$.
Let's consider the arrow $y^2 \to 1$. In the above notation c^{02}_{00}. We can move this

arrow one to the right and then one to the bottom to push its head underneath the "x-axis". With Definition 12 this arrow is thus equal to zero and does not correspond to a valid arrow. As a $\mathbb{C}[x,y]$-module homomorphism we have

$$c_{00}^{02}(xy^2) = x \cdot c_{00}^{02}(y^2) = x \cdot 1 = x\,,$$

but also

$$c_{00}^{02}(xy^2) = y \cdot c_{00}^{02}(xy) = y \cdot 0 = 0\,.$$

In the second case we have $1+0 < 2$ and thus $c_{00}^{02}(xy) = 0$ (see equation (1)). This is a contradiction and therefore this arrow does not represent an element of $\mathrm{Hom}_{\mathbb{C}[x,y]}(I, \mathbb{C}[x,y]/I)$. Again this is in agreement with the definition of the Haiman arrows and Proposition 15.

Remark 3. When determining the dimension of the tangent space it is of course much easier to use the theory from Definition 12. The example above serves mainly to illustrate Proposition 15

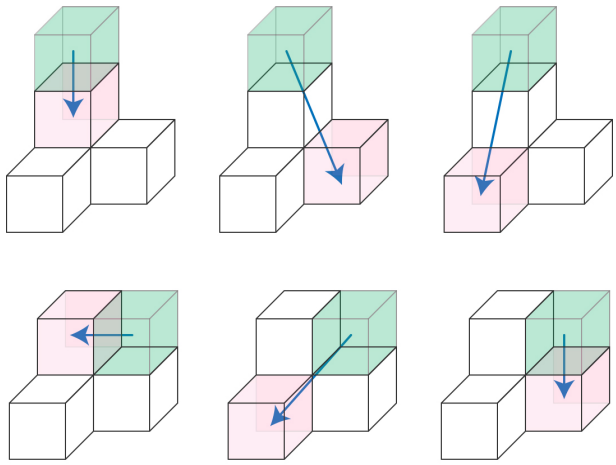
Being able to calculate the dimension of the tangent space we can say something about the smoothness of the Hilbert scheme. When $d = 2$, there is the following result

Theorem 17. *The Hilbert scheme $Hilb^n(\mathbb{C}^2)$ is a smooth and irreducible complex algebraic variety of dimension $2n$.*

Proof see Thm. 18.7 in [MS05].
However, unlike connectedness, this does not hold for arbitrary d. In general for $n > d \geq 3$ the Hilbert scheme is not smooth. The next example illustrates this when $d = 3$ and $n = d + 1$.

Example 18. H_4^3 contains a smooth subvariety of dimension $d \cdot n = 12$ that contains all radical ideals. From Lemma 10 it follows that the monomial ideals lie in the closure of this smooth subvariety. For H_4^3 to be smooth, it is therefore necessary that the tangent space at any monomial ideal has dimension equal to 12
Take the monomial ideal $I = \langle x^2, y^2, z^2, xy, xz, yz\rangle$. The quotient $\mathbb{C}[x,y,z]/I$ is a $\mathbb{C}$-vector space of dimension 4. For each of the six minimal generators of I we can find 3 different arrows that are not zero. The six valid arrows for the generators z^2 and yz are depicted below. By symmetry we can just

multiply that number by three to obtain the total number of valid arrows. The dimension of the tangent space is thus 18. It follows that I is a singular point and H_4^3 is not smooth.



4 Algorithm to determine dimension of tangent space at monomial ideals

We have implemented the theory from Definition 12 in Python to computationally determine the dimension of the tangent space of the Hilbert scheme at any monomial ideal in $\mathbb{C}^3$ or $\mathbb{C}^4$. In this section we will explain how the program works for monomial ideals in H_n^3. The corresponding code can be found in the Appendix in Listing 4. For monomial ideals in H_n^4 the program works very similar with the only exception that a fourth variable t is added. The code can be found in Listing 5. Anyone just interested in calculating the dimension at a given monomial ideal, can follow the example given in Listing 1.
In addition to calculating the dimension of the tangent space, the two programs can also find all plane-, respectively multi partitions in three and four dimensions. However, this only works in a reasonable time for up to $n \approx 10$. We will not explain the functions behind that part of the program but give an example of code to find all partitions, their minimal generators as ideals

and the corresponding dimensions of the tangent space. In this way the most singular points of H_n^d can be found. The results of this are listed in section 5. The code that needs to be inserted in the program can be found in Listing 3 and can be used for both programs.
The program is structured as follows. First the monomial ideal is entered and assigned to a variable. Preferably only the minimal generators of the ideal are listed. There are two ways to do this. Either as a list of strings or as a list of 3-tuples. While the first is the most convenient, the program actually works with 3-tuples so we need to convert it with a function *convertIdeal*. The three coordinates are denoted x, y and z.
Take for example the ideal $I = \langle x^2, y^2, z^2, xy, xz, yz \rangle$. As a list of strings we would it enter it as ["x2", "y2", "z2", "xy", "xz", "yz"]. As a list of tuples we have [[2,0,0],[0,2,0],[0,0,2],[1,1,0], [1,0,1],[0,1,1]].
In the next step we need to determine the corresponding plane partition. A function *createPartition*, taking as an argument the monomial ideal in tuple form, will be called for this purpose. This function works as follows: First the maximum x, y and z values from all generators of I are determined and a (hyper)rectangle of boxes is created. The numbers of boxes in each direction correspond to the maximum values in those directions. In the program this (hyper)rectangle is a list of 3-tuples. Then all boxes that do in fact not lie in the partition are removed. For this we have a for-loop running through all generators of I and three nested for-loops, one per direction, running from the generator to the maximum values, removing boxes that lie in the positive direction of the generators. The function returns a list of 3-tuples which is the desired partition.
Then a function *arrows* is called, taking as arguments the partition just generated and the monomial ideal. It is sufficient to only check arrows starting at minimal generators. In the first step the function creates all possible arrows from minimal generators to boxes in the partition and stores them in a variable **all_arrows**. Formally an arrow in the program is a list of two 3-tuples, the tail and the head. Then the first arrow of **all_arrows** is selected and is stored in a variable **a_class** which is its equivalence class. The goal is to find all arrows that belong to that equivalence class. Starting with the initial arrow, the function recursively determines if the arrow can be moved. A function *moveArrow* tries translating the arrow one unit in the positive and negative x, y and z direction and stores all moves that are valid according to Definition 12 in a variable **waiting_list**. To avoid infinite recursions only moves (which are in fact arrows as well) that are not yet stored in the

Listing 1: Calculating the dimension of the tangent space at $I = \langle x, y, z \rangle^2 \in H_4^3$. For convenience several values of interest are printed on the screen. The code below needs to be inserted at the end of the program in Listing 4 to properly work! For the output generated by the program see Listing 2.

```
mon_id = ["x2", "xy", "y2","xz", "yz", "z2"]
mon_ideal = convertIdeal(mon_id)
partition = createPartition(mon_ideal)

print("Partition: ", partition)
print("Colength ideal (=n): ", len(partition))
print("d*n: ", len(partition)*3)

arrows = arrows(partition, mon_ideal)
print("Number of arrows: ", len(arrows))
```

Listing 2: Output of code in Listing 1 when inserted at the end of Listing 4

```
Partition: [[0, 0, 0], [0, 0, 1], [0, 1, 0], [1, 0, 0]]

Colength ideal (=n): 4
d*n:  12
Number of arrows: 18
```

equivalence class or in the waiting list can be added to that variable. Furthermore arrows which have crossed any of the axes will not be added to **waiting_list**. Instead if such an arrow is encountered, the value 0 is inserted at the beginning of the list **a_class**. Once the recursion has ended, all arrows that are in the equivalence class and were in the initial list of all arrows are removed from **all_arrows**. If the initial value of the list **a_class** is not zero, it is in fact a valid arrow and will be added to a list **equiv_classes**. Then, as long as there are elements left in **all_arrows**, this process gets repeated. In the end the list **equiv_classes** will be returned by the function *arrows*. The number of elements in **equiv_classes** is the dimension of the tangent space of the Hilbert scheme at the monomial ideal.

Listing 3: Finding all plane-, respectively multi partitions for $n = 8$. Then the minimal generators of the corresponding ideals and the dimension of their tangent spaces are determined. Per partition the dimension of the tangent space, the number of minimal generators of the corresponding ideal, the list of the partition and the list of the minimal generators is put in a list 'lst'. The code below can be used for both programs but needs to be inserted at the end to properly work!

```
1
2  partitions = findAllPartitions(8)
3  min_gen = []
4  for p in partitions:
5      min_gen.append(findMinGen(p))
6
7  tangent_space = []
8  for i in range(len(partitions)):
9      tangent_space.append(arrows(partitions[i], min_gen[i]))
10
11 dim = [len(tan) for tan in tangent_space]
12 num_gen = [len(gen) for gen in min_gen]
13 lst = sorted(list(zip(dim, num_gen, partitions, min_gen)))
14
15 l = lst[-1]
```

5 Experimental results

An open question in the study of Hilbert schemes of points is to find the most singular point of $\mathrm{Hilb}^n(\mathbb{C}^d)$, that is the ideal I for which the dimension of the tangent space $\mathrm{Hom}_{\mathbb{C}[\vec{\mathbf{x}}]}(I, \mathbb{C}[\vec{\mathbf{x}}]/I)$ is maximal. This is a question of looking at monomial ideals and thus a combinatorial one. In this section we will present some experimental results.
For $2 \leq n \leq 12$ we have generated all possible plane partitions and their corresponding minimal generators and tangent space dimensions (for larger n this calculation takes too long on an ordinary computer).
For $2 \leq n \leq 9$ we have generated all possible multipartitions in four dimensions and their corresponding minimal generators and tangent space dimensions. The results can be found in the table below
The first conjecture one could make is that the most singular monomial ideal is the one with most minimal generators. However, when $n = 8$ one finds the following counterexample.

Example 19. We have $d = 3$ and $n = 8$. The ideals with most generators are

$$\langle x, xz, yz^2, y^2z, x^2, y^3, z^4 \rangle \text{ and } \langle xy, xz^2, yz^2, x^2, z^3, y^3 \rangle .$$

Each of them has seven minimal generators. At both points the tangent space of the Hilbert scheme has dimension 32. However, the point in H_8^3 given by

$$I = \langle x^2, xy, y^2, xz^2, yz^2, z^4 \rangle ,$$

has only 6 minimal generators, but the dimension of the tangent space is 36.

Remark 4. In three dimensions there exists a generating function for the number of plane partitions, sometimes referred to as the *MacMahon function*

$$\prod_{k=1}^{\infty} \frac{1}{(1-x^k)^k} = 1 + x + 3x^2 + 6x^3 + 13x^4 + 24x^5 + \dots .$$

This is in agreement with the values listed in Table 1.

	H_n^3			H_n^4		
	max dim	# min gen	# part	max dim	# min gen	# part
n						
2	6	3	3	8	4	4
3	9	3	6	12	5	10
4	18	6	13	22	7	26
5	21	6	24	40	10	59
6	24	6	48	44	10	140
7	29	6	86	48	10	307
8	36	7	160	57	11	684
9	43	6	282	62	10	1464
10	60	8	500			
11	60	10	859			
12	63	10	1479			

Table 1: Dimensions of the most singular points (max dim) in H_n^d for $d = 3, 4$, together with the number of minimal generators the corresponding ideals have and the number of different plane-, respectively multi partitions.

6 Appendix

Listing 4: Pythoncode to calculate dimension of tangent space at a monomial ideal in $\mathrm{Hilb}^n(\mathbb{C}^3)$. For further explanation see section 4.

```
"""
Write e.g. the monomial (x^2 y^3 z) as "x2y3z"

The program can calculate the dimension of the tangent space at a
    monomial ideal in the Hilbert scheme of points over C^3 and all
    possible plane partitions for a given n.
"""

def convertIdeal(strIdeal):
    """
    Input: monomial ideal with monomials as strings
    Output: List of 4-tuples (exponent vector format)
    e.g. ["x2", "y", "z", "t"] -->
        [[2,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]]
    """
    ideal = []
    for mon in strIdeal:
        ideal.append(convMonomial(mon))
    return ideal

def convMonomial(mon):
    """
    Input: monomial ideal as strings
    Output: 4-tuple (exponent vector format)
    example see convertIdeal
    """
    triple = [0,0,0]
    xpos = mon.find("x")
    ypos = mon.find("y")
    zpos = mon.find("z")

    if xpos != -1:
        try:
            triple[0] = int(mon[xpos+1])
        except:
```

```
            triple[0] = 1
        if ypos != -1:
            try:
                triple[1] = int(mon[ypos+1])
            except:
                triple[1] = 1
        if zpos != -1:
            try:
                triple[2] = int(mon[zpos+1])
            except:
                triple[2] = 1

        return triple

    def tupleProduct(list1, list2):
        """
        Input: two lists
        Output: List of 2-tuples
        [a1,...,an] and [b1,,...,bn] --> [[a1,b1],...,[an,bn]]
        """
        lst = []
        for item1 in list1:
            for item2 in list2:
                lst.append([item1, item2])
        return lst

    def isPartition(partition):
        """
        Input: List of 4-tuples (multipartition)
        Output: Boolean

        checks whether a list of 4-tuples is a valid multipartition
        """
        for block in partition:
            for i in range(3):
                b = list(block)
                if b[i] != 0:
                    b[i] += (-1)
                    if b not in partition:
                        return False
```

```
    return True

def createPartition(ideal):
    """
    Input: List of 4-tuples (monomial ideal in exponent vector
       format)
    Output: List of 4-tuples (multipartition)

    creates the multipartition corresponding to a monomial ideal
    """
    xmax = max([mon[0] for mon in ideal])
    ymax = max([mon[1] for mon in ideal])
    zmax = max([mon[2] for mon in ideal])


    partition = []
    for x in range(xmax):
        for y in range(ymax):
            for z in range(zmax):
                partition.append([x,y,z])

    for i in ideal:
        for x in range(i[0], xmax):
            for y in range(i[1],ymax):
                for z in range(i[2],zmax):
                    if [x,y,z] in partition:
                        partition.remove([x,y,z])

    if isPartition(partition):
        return partition
    else:
        raise ValueError("Not a partition")

def isValidMove(arrow, partition):
    """
    Input: arrow, Tuple of 4-tuple
           partition, List of 4-tuples
    Output: Boolean

    Checks whether an arrow is a valid Haiman arrow
```

```
    """
    if arrow[0] not in partition and arrow[1] in partition:
        return True
    elif arrow[0] not in partition and any(n < 0 for n in arrow[1]):
        return True
    elif arrow[0] in partition:
        return False

def moveArrow(arrow, partition):
    """
    Input: arrow, Tuple of 4-tuple
           partition, List of 4-tuples
    Output: list of two 4-tuples (arrows)

    gives all possible moves in the x-,y-,z- and t-direction.
    """
    moves = []
    xp = [[arrow[0][0] + 1, arrow[0][1], arrow[0][2]], [arrow[1][0]
        + 1, arrow[1][1], arrow[1][2]]]
    if isValidMove(xp, partition):
        moves.append(xp)

    yp = [[arrow[0][0], arrow[0][1] + 1, arrow[0][2]],
        [arrow[1][0], arrow[1][1] + 1, arrow[1][2]]]
    if isValidMove(yp, partition):
        moves.append(yp)

    zp = [[arrow[0][0], arrow[0][1], arrow[0][2] + 1],
        [arrow[1][0], arrow[1][1], arrow[1][2] + 1]]
    if isValidMove(zp, partition):
        moves.append(zp)

    if arrow[0][0] != 0:
        xm = [[arrow[0][0] - 1, arrow[0][1], arrow[0][2]],
            [arrow[1][0] - 1, arrow[1][1], arrow[1][2]]]
        if isValidMove(xm, partition):
            moves.append(xm)

    if arrow[0][1] != 0:
```

```
        ym = [[arrow[0][0], arrow[0][1] - 1, arrow[0][2]],
            [arrow[1][0], arrow[1][1] - 1, arrow[1][2]]]
        if isValidMove(ym, partition):
            moves.append(ym)

    if arrow[0][2] != 0:
        zm = [[arrow[0][0], arrow[0][1], arrow[0][2]-1],
            [arrow[1][0], arrow[1][1], arrow[1][2]-1]]
        if isValidMove(zm, partition):
            moves.append(zm)
    return moves


def arrows(partition, ideal):
    """
    Input: partition, List of 4-tuples
           ideal, list of 4-tuples
    Output: list of two 4-tuples (equivalence classes of arrows)

    Determines all different equivalence classes of Haiman arrows.
    """
    all_arrows = tupleProduct(ideal,partition)
    equiv_classes = []
    while all_arrows:
        a = list(all_arrows[0])
        a_class = [a]
        waiting_list = [a]
        while waiting_list:
            b = list(waiting_list[0])
            waiting_list.pop(0)
            valid_moves = moveArrow(b, partition)
            for move in valid_moves:
                if move not in a_class and move not in waiting_list:
                    a_class.append(move)
                    if not any(n < 0 for n in move[1]):
                        waiting_list.append(move)
                    elif any(n < 0 for n in move[1]):
                        a_class.insert(0,0)
        for arrow in a_class:
            if arrow in all_arrows:
```

```
                all_arrows.remove((arrow))
        if a_class[0] != 0:
            equiv_classes.append(a_class)
    return equiv_classes

def findMinGen(partition):
    """
    Input: partition, List of 4-tuples
    Output: list of 4-tuples (minimal generators of monomial ideals
        corresponding to multipartition)
    """
    min_gen = []
    for b in partition:
        for i in range(3):
            a = list(b)
            a[i] += 1
            if a not in partition:
                for j in range(3):
                    neighbor = list(a)
                    neighbor[j] = neighbor[j] - 1
                    if neighbor not in partition and neighbor[j] >=
                        0:
                        break
                    if j ==2 and a not in min_gen:
                        min_gen.append(a)
    return min_gen

def nextBlock(partitions, current, n):
    """
    Input: partitions, List of List of 4-tuples (all
        multipartitions already found)
            current, 4-tuple (coordinate of the current block in the
                multipartition)
            n, integer for which multipartitions are to be determined
    Output: List of List of 4-tuples (all multipartitions already
        found)

    determines all possible blocks when "building" a partition
    """
    for block in current:
```

```
        for i in range(3):
            new_block = list(block)
            new_block[i] += 1
            if new_block not in current:
                new_part = sorted(current + [new_block])
                if isPartition(new_part):
                    if len(new_part) == n:
                        if new_part not in partitions:
                            partitions.append(new_part)
                    else:
                        partitions = nextBlock(partitions, new_part,
                            n)
    return partitions

def findAllPartitions(n):
    """
    Input: n, integer for which multipartitions are to be determined
    Output: List of List of 4-tuples (all multipartitions already
        found)

    Finds all multipartitions for a given n. Starts with the block
        at the origin
    and expands the multipartition recursively in all directions
        until it has n blocks.
    """
    partitions = []
    current_partition = [[0,0,0]]
    partitions = nextBlock(partitions, current_partition, n)
    return partitions
```

Listing 5: Pythoncode to calculate dimension of tangent space at a monomial ideal in $\mathrm{Hilb}^n(\mathbb{C}^4)$. For an explanation see section 4.

```
"""
Write e.g. the monomial (x^2 y^3 z t^2) as "x2y3zt2"

The program can calculate the dimension of the tangent space at a
    monomial ideal in the Hilbert scheme of points over C^4 and all
    possible multi partitions for a given n.
"""
```

```
def convertIdeal(strIdeal):
    """
    Input: monomial ideal with monomials as strings
    Output: List of 4-tuples (exponent vector format)
    e.g. ["x2", "y", "z", "t"] -->
        [[2,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]]
    """
    ideal = []
    for mon in strIdeal:
        ideal.append(convMonomial(mon))
    return ideal

def convMonomial(mon):
    """
    Input: monomial ideal as strings
    Output: 4-tuple (exponent vector format)
    example see convertIdeal
    """
    triple = [0,0,0,0]
    xpos = mon.find("x")
    ypos = mon.find("y")
    zpos = mon.find("z")
    tpos = mon.find("t")
    if xpos != -1:
        try:
            triple[0] = int(mon[xpos+1])
        except:
            triple[0] = 1
    if ypos != -1:
        try:
            triple[1] = int(mon[ypos+1])
        except:
            triple[1] = 1
    if zpos != -1:
        try:
            triple[2] = int(mon[zpos+1])
        except:
            triple[2] = 1
    if tpos != -1:
```

```
        try:
            triple[3] = int(mon[tpos+1])
        except:
            triple[3] = 1
    return triple

def tupleProduct(list1, list2):
    """
    Input: two lists
    Output: List of 2-tuples
    [a1,...,an] and [b1,,...,bn] --> [[a1,b1],...,[an,bn]]
    """
    lst = []
    for item1 in list1:
        for item2 in list2:
            lst.append([item1, item2])
    return lst

def isPartition(partition):
    """
    Input: List of 4-tuples (multipartition)
    Output: Boolean

    checks whether a list of 4-tuples is a valid multipartition
    """
    for block in partition:
        for i in range(4):
            b = list(block)
            if b[i] != 0:
                b[i] += (-1)
                if b not in partition:
                    return False
    return True

def createPartition(ideal):
    """
    Input: List of 4-tuples (monomial ideal in exponent vector
        format)
    Output: List of 4-tuples (multipartition)
```

```
    creates the multipartition corresponding to a monomial ideal
    """
    xmax = max([mon[0] for mon in ideal])
    ymax = max([mon[1] for mon in ideal])
    zmax = max([mon[2] for mon in ideal])
    tmax = max([mon[3] for mon in ideal])

    partition = []
    for x in range(xmax):
        for y in range(ymax):
            for z in range(zmax):
                for t in range(tmax):
                    partition.append([x,y,z,t])

    for i in ideal:
        for x in range(i[0], xmax):
            for y in range(i[1],ymax):
                for z in range(i[2],zmax):
                    for t in range(i[3], tmax):
                        if [x,y,z,t] in partition:
                            partition.remove([x,y,z,t])

    if isPartition(partition):
        return partition
    else:
        raise ValueError("Not a partition")

def isValidMove(arrow, partition):
    """
    Input: arrow, Tuple of 4-tuple
           partition, List of 4-tuples
    Output: Boolean

    Checks whether an arrow is a valid Haiman arrow
    """
    if arrow[0] not in partition and arrow[1] in partition:
        return True
    elif arrow[0] not in partition and any(n < 0 for n in arrow[1]):
        return True
    elif arrow[0] in partition:
```

```
        return False

def moveArrow(arrow, partition):
    """
    Input: arrow, Tuple of 4-tuple
           partition, List of 4-tuples
    Output: list of two 4-tuples (arrows)

    gives all possible moves in the x-,y-,z- and t-direction.
    """
    moves = []
    xp = [[arrow[0][0] + 1, arrow[0][1], arrow[0][2], arrow[0][3]],
        [arrow[1][0] + 1, arrow[1][1], arrow[1][2], arrow[1][3]]]
    if isValidMove(xp, partition):
        moves.append(xp)

    yp = [[arrow[0][0], arrow[0][1] + 1, arrow[0][2], arrow[0][3]],
        [arrow[1][0], arrow[1][1] + 1, arrow[1][2], arrow[1][3]]]
    if isValidMove(yp, partition):
        moves.append(yp)

    zp = [[arrow[0][0], arrow[0][1], arrow[0][2] + 1, arrow[0][3]],
        [arrow[1][0], arrow[1][1], arrow[1][2] + 1, arrow[1][3]]]
    if isValidMove(zp, partition):
        moves.append(zp)

    tp = [[arrow[0][0], arrow[0][1], arrow[0][2], arrow[0][3] + 1],
        [arrow[1][0], arrow[1][1], arrow[1][2], arrow[1][3] + 1]]
    if isValidMove(tp, partition):
        moves.append(tp)

    if arrow[0][0] != 0:
        xm = [[arrow[0][0] - 1, arrow[0][1], arrow[0][2],
            arrow[0][3]], [arrow[1][0] - 1, arrow[1][1],
            arrow[1][2], arrow[1][3]]]
        if isValidMove(xm, partition):
            moves.append(xm)

    if arrow[0][1] != 0:
        ym = [[arrow[0][0], arrow[0][1] - 1, arrow[0][2],
```

```
            arrow[0][3]], [arrow[1][0], arrow[1][1] - 1,
            arrow[1][2], arrow[1][3]]]
        if isValidMove(ym, partition):
            moves.append(ym)

    if arrow[0][2] != 0:
        zm = [[arrow[0][0], arrow[0][1], arrow[0][2]-1,
            arrow[0][3]], [arrow[1][0], arrow[1][1], arrow[1][2]-1,
            arrow[1][3]]]
        if isValidMove(zm, partition):
            moves.append(zm)

    if arrow[0][3] != 0:
        tm = [[arrow[0][0], arrow[0][1], arrow[0][2], arrow[0][3] -
            1], [arrow[1][0], arrow[1][1], arrow[1][2], arrow[1][3]
            - 1]]
        if isValidMove(tm, partition):
            moves.append(tm)

    return moves


def arrows(partition, ideal):
    """
    Input: partition, List of 4-tuples
           ideal, list of 4-tuples
    Output: list of two 4-tuples (equivalence classes of arrows)

    Determines all different equivalence classes of Haiman arrows.
    """
    all_arrows = tupleProduct(ideal,partition)
    equiv_classes = []
    while all_arrows:
        a = list(all_arrows[0])
        a_class = [a]
        waiting_list = [a]
        while waiting_list:
            b = list(waiting_list[0])
            waiting_list.pop(0)
            valid_moves = moveArrow(b, partition)
```

```
            for move in valid_moves:
                if move not in a_class and move not in waiting_list:
                    a_class.append(move)
                    if not any(n < 0 for n in move[1]):
                        waiting_list.append(move)
                    elif any(n < 0 for n in move[1]):
                        a_class.insert(0,0)
        for arrow in a_class:
            if arrow in all_arrows:
                all_arrows.remove((arrow))
        if a_class[0] != 0:
            equiv_classes.append(a_class)
    return equiv_classes

def findMinGen(partition):
    """
    Input: partition, List of 4-tuples
    Output: list of 4-tuples (minimal generators of monomial ideals
        corresponding to multipartition)
    """
    min_gen = []
    for b in partition:
        for i in range(4):
            a = list(b)
            a[i] += 1
            if a not in partition:
                for j in range(4):
                    neighbor = list(a)
                    neighbor[j] = neighbor[j] - 1
                    if neighbor not in partition and neighbor[j] >=
                        0:
                        break
                    if j == 3 and a not in min_gen:
                        min_gen.append(a)
    return min_gen


def nextBlock(partitions, current, n):
    """
    Input: partitions, List of List of 4-tuples (all
```

```
        multipartitions already found)
            current, 4-tuple (coordinate of the current block in the
                multipartition)
            n, integer for which multipartitions are to be determined
        Output: List of List of 4-tuples (all multipartitions already
            found)

        determines all possible blocks when "building" a partition
        """
        for block in current:
            for i in range(4):
                new_block = list(block)
                new_block[i] += 1
                if new_block not in current:
                    new_part = sorted(current + [new_block])
                    if isPartition(new_part):
                        if len(new_part) == n:
                            if new_part not in partitions:
                                partitions.append(new_part)
                        else:
                            partitions = nextBlock(partitions, new_part,
                                n)
        return partitions

    def findAllPartitions(n):
        """
        Input: n, integer for which multipartitions are to be determined
        Output: List of List of 4-tuples (all multipartitions already
            found)

        Finds all multipartitions for a given n. Starts with the block
            at the origin
        and expands the multipartition recursively in all directions
            until it has n blocks.
        """
        partitions = []
        current_partition = [[0,0,0,0]]
        partitions = nextBlock(partitions, current_partition, n)
        return partitions
```

References

[AM16] Michael Atiyah and Ian Macdonald. *Introduction to Commutative Algebra*. Addison-Wesley series in Mathematics. Westview Press, 2016, pp. 1–31.

[CLO97] David Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms: An introduction to computational algebraic geometry and commutative algebra*. second. Undergraduate Texts in Mathematics. New York: Springer-Verlag, 1997.

[Eis95] David Eisenbud. *Commutative Algebra, with a view toward algebraic geometry*. Vol. 150. Graduate Texts in Mathematics. New York: Springer-Verlag, 1995, pp. 345–347.

[Gro61] Alexander Grothendieck. "Techniques de construction et théorémes d'existence en géométrie algébrique IV : les schémas de Hilbert". In: Séminaire Bourbaki 221 (1960/61), pp. 249–276.

[Leh04] Manfred Lehn. "Lectures on Hilbert schemes". In: CRM Proceedings & Lecture Notes 38 (2004). (Lectures at the Workshop on Algebraic Structures and Moduli Spaces. CRM Montréal, Juli 2003.), pp. 4–6.

[MS05] Ezra Miller and Bernd Sturmfels. *Combinatorial Commutative Algebra*. Vol. 227. Graduate Texts in Mathematics. New York: Springer-Verlag, 2005, pp. 355–363, 368–373.