Conjectures for Coefficients in the Linear Combinations of Partial Derivatives of Interpolation Jack Polynomials

Havi Ellers (hellers@g.hmc.edu)
and Xiaomin Li (xiaomin3@illinois.edu)

1 Executive Summary

Our research this summer focused on certain polynomials called *interpolation Jack polynomials*, P_λ. The polynomials we focused on were in three variables, x, y, and z, and had coefficients that were functions of a fourth variable, κ. It turns out that if we differentiate P_λ with respect to the variable κ, the result can be written as a sum:

$$\sum cP_\mu$$

where each P_μ is an interpolation Jack polynomial, and each c is a function of κ. Our goal this summer was to find a formula for the c. This problem can be broken down into several cases, and for some of these cases we found a complete formula for c, where as in other cases we have only a partial formula.

Contents

2 Background

2.1 Theoretical Motivations

Our research over the summer of 2019 focused on the *interpolation Jack polynomials*. Properties of these polynomials have been extensively studied in the papers of Sahi, Knop-Sahi, and Okounkov-Olshanski. These polynomials have connections to representation theory and to the theory of Lie algebras that motivates their study. In this section we will first present a combinatorial algorithm (discovered by Okounkov) to generate interpolation Jack polynomials, and then discuss some of their connections to representation theory of Lie algebras.

2.1.1 Combinatorial Algorithm

We begin with a quick introduction to partitions and Young diagrams, as these are integral in the study of interpolation Jack polynomials.

Definition 1. A *partition* of an integer m is a tuple $\lambda = (\lambda_1, \ldots, \lambda_n)$ such that $\lambda_1, \ldots, \lambda_n$ are non-negative integers, $\lambda_1 \geq \cdots \geq \lambda_n$, and $\sum \lambda_i = m$. The number n is called the *length* of the partition λ, and the number m is called the *size* of the partition λ, denoted $|\lambda|$.

Young diagrams are certain collections of boxes that are associated to partitions.

Definition 2. (See [4].) Let $\lambda = (\lambda_1, \ldots, \lambda_n)$ be a partition with $|\lambda| = m$. Then a *Young diagram* (also called a *Ferrers diagram*) of shape λ is a collection of m boxes with n left-justified rows, with row i containing λ_i boxes.

Interpolation Jack polynomials are certain polynomials in n variables $x_1, \ldots, x_n$, indexed by partitions λ of length at most n. We now present a series of definitions, which build up to a combinatorial formula for the interpolation Jack polynomials P_λ. We present the algorithm for interpolation Jack polynomials in three variables, but it can be generalized to n variables with only slight modification.

Definition 3. Let $\mu = (\mu_1, \mu_2, \mu_3)$ be a Young diagram. Then for a box s at position (i, j) in μ, we define

$$a_\mu(s) = \mu_i - j \qquad a'_\mu(s) = j - 1$$
$$l_\mu(s) = |\{k > i \,|\, \mu_k \geq j\}| \qquad l'_\mu(s) = i - i$$

Note that $l'_\mu(s), l_\mu(s), a'_\mu(s)$, and $a_\mu(s)$ are respectively the number of boxes in μ to the north, south, east and west of the box s (see [3]).

Definition 4. Let $\mu = (\mu_1, \mu_2, \mu_3)$ be a Young diagram. Then for a box s in μ, we define

$$b_\mu(s, \kappa) = \frac{a_\mu(s) + \kappa(l_\mu(s) + 1)}{a_\mu(s) + \kappa l_\mu(s) + 1}$$

Definition 5. Let $\lambda = (\lambda_1, \lambda_2, \lambda_3)$ be a Young diagram. Then a *reverse tableau* is a filling of the Young diagram λ by the numbers 1, 2, 3 such that the rows are weakly decreasing from left to right and the columns are strictly decreasing from top to bottom.

For a box s in a reverse tableau T, we denote the entry in s by $T(s)$, and we denote the position of s by (i, j), where i denotes the row number and j denotes the column number.

Definition 6. Let $\lambda = (\lambda_1, \lambda_2, \lambda_3)$ be a Young diagram, and let T be a reverse tableau of shape λ. Then for $i = 0, \ldots, 3$, we define

$$\lambda^{(i)} = \text{ the boxes in } T \text{ that contain the numbers } i+1, \ldots, 3.$$

Definition 7. Let $\lambda = (\lambda_1, \lambda_2, \lambda_3)$ be a Young diagram, and let T be a reverse tableau of shape λ. Then for $i = 1, \ldots, 3$, we define $(R/C)_{\lambda^{(i-1)}/\lambda^{(i)}}$ to be the set of boxes s in the Young diagram $\lambda^{(i)}$ with the following two properties:

- *Property 1*: The row that contains s also contains a box from $\lambda^{(i-1)}$ that is not in $\lambda^{(i)}$.
- *Property 2*: The column that contains s does not contain a box from $\lambda^{(i-1)}$ that is not in $\lambda^{(i)}$.

Definition 8. Let $\lambda = (\lambda_1, \lambda_2, \lambda_3)$ be a Young diagram and let T be a reverse tableau of shape λ. Then we define

$$\psi_T(\kappa) = \prod_{i=1}^{3} \prod_{s \in (R/C)_{\lambda^{(i-1)}/\lambda^{(i)}}} \frac{b_{\lambda^{(i)}}(s, \kappa)}{b_{\lambda^{(i-1)}}(s, \kappa)}$$

where the function b is as in definition 4.

Finally we define the interpolation Jack polynomials that we wish to work with.

Definition 9. Let $\lambda = (\lambda_1, \lambda_2, \lambda_3)$ be a partition. Then the *interpolation Jack polynomial* associated with λ is

$$P_\lambda(x_1, x_2, x_3) = \sum_{\substack{T \text{ a reverse} \\ \text{tableau} \\ \text{of shape } \lambda}} \prod_{s \in T} \psi_T(\kappa)(x_{T(s)} - a'(s) + l'(s)\kappa)$$

Note that in much of what follows, we will use the variables x, y, z instead of the variables x_1, x_2, x_3.

2.1.2 Lie algebras

In this section we provide a more theoretical background to the motivation behind studying interpolation Jack polynomials, starting with a brief introduction to Lie algebras.

Definition 10. (See [1].) Let F be a field. A *Lie algebra* over F is an F-vector space L together with a bilinear map

$$L \times L \to L, \quad (x, y) \mapsto [x, y]$$

that satisfies the following properties:

$$[x, x] = 0 \quad \text{for all } x \in L, \tag{1}$$

$$[x, [y, z]] + [y, [x, z]] + [z, [x, y]] = 0 \quad \text{for all } x, y, z \in L \tag{2}$$

The map $(x, y) \mapsto [x, y]$ is often called the *Lie bracket*.

For any Lie algebra L, we can form its *universal enveloping algebra*. We first give a more abstract definition of the universal enveloping algebra, and then a more concrete one involving the tensor algebra of a Lie algebra.

Definition 11. (See [2].) Let L be a Lie algebra over a field F. A *universal enveloping algebra* of L is a pair $(\mathcal{U}, i)$, where $\mathcal{U}$ is an associative algebra with 1 over F, $i : L \to \mathcal{U}$ is a linear map satisfying

$$i([x, y]) = i(x)i(y) - i(y)i(x) \quad \text{for all } x, y \in L, \tag{3}$$

and the following holds: for any associative F-algebra $\mathfrak{U}$ with 1 and any linear map $j : L \to \mathfrak{U}$ satisfying (3), there exists a unique homomorphism of algebras $\phi : \mathcal{U} \to \mathfrak{U}$ (sending 1 to 1) such that $\phi \circ i = j$.

This is a rather abstract definition, and we will not deal with it directly. Instead, we consider the tensor algebra of a vector space V. [1]

Definition 12. Let V be a finite-dimensional vector space over a field F. Let $T^0V = F, T^1V = V$, and for each $m > 0$ let $T^m = V \otimes \cdots \otimes V$ (m times), where $\otimes$ denotes the tensor product. The *tensor algebra* on V is then the vector space

$$\mathcal{I}(V) = \bigoplus_{i=0}^{\infty} T^i V$$

together with the product

$$(v_1 \otimes \cdots \otimes v_k)(w_1 \otimes \cdots \otimes w_m) = v_1 \otimes \cdots \otimes v_k \otimes w_1 \otimes \cdots \otimes w_m$$

Note that this choice of product makes $\mathcal{I}(V)$ an associative algebra with unity.

Now let L be a finite-dimensional Lie algebra, and let J be the two-sided ideal in $\mathcal{I}(L)$ generated by elements of the form $x \otimes y - y \otimes x - [x, y]$ (where $x, y \in L$). Let $\mathcal{U}(L) = \mathcal{I}(L)/J$, and let $\pi : \mathcal{I}(\mathcal{L}) \to \mathcal{U}(L)$ be the canonical homomorphism. Then the pair $(\mathcal{U}(L), i)$, where i is the restriction of π to L, is a universal enveloping algebra for L, and furthermore it is the unique universal enveloping algebra of L up to isomorphism.
Thus for a finite-dimensinoal Lie algebra L we can refer *the* universal enveloping algebra of L, which can be thought of as the quotient algebra $\mathcal{U}(L) = \mathcal{I}(L)/J$, as described above.
We now shift our focus to a particular finite-dimensional Lie algebra, the general linear group of order n over $\mathbb{C}$, $\mathrm{gl}\,(n, \mathbb{C})$. It is left as an exercise to the reader to show that with Lie bracket

$$[x, y] := xy - yx \quad \text{for all } x, y \in \mathrm{gl}\,(n, \mathbb{C})$$

the vector space $\mathrm{gl}\,(n, \mathbb{C})$ is a Lie algebra over $\mathbb{C}$. We can now form the universal enveloping algebra of $\mathrm{gl}\,(n, \mathbb{C})$, denoted by $\mathcal{U}(\mathrm{gl}\,(n, \mathbb{C}))$. The Harish-Chandra isomorphsim implies that the center of the latter algebra is isomorphic to the subspace of symmetric polynomials in $\mathbb{C}[x_1, \ldots, x_n]$. The following theorem is due to Okounkov:

[1] Note that this discussion is heavily influenced by chapter 17 in [2].

Theorem 1. There is a basis of $\mathcal{Z}(\mathcal{U}(\mathrm{gl}\,(n,\mathbb{C})))$ that is indexed by partitions with at most n parts (where $\mathcal{Z}(\mathcal{U}(\mathrm{gl}\,(n,\mathbb{C})))$ denotes the center of $\mathcal{U}(\mathrm{gl}\,(n,\mathbb{C}))$).

Let us call the elements in this basis s_λ.
We will now take a brief detour into representation theory. For a vector space V let $\mathrm{gl}\,(V)$ denote the set of all invertible linear transformations from V to itself (which is a group under composition), and recall the following definitions:

Definition 13. Let G is a group and V be a vector space. Then V is a *G-module* (or a *representation* of G) if there is a group homomorphism $\rho : G \to \mathrm{gl}\,(V)$.

Definition 14. Let G be a group and let V be a G-module with group homomorphism $\rho : G \to \mathrm{gl}\,(V)$. Let $W \subseteq V$ be a vector subspace of V. Then W is a *submodule* of V if

$$w \in W \implies (\rho(g))(w) \in W \quad \text{for all } g \in G$$

Definition 15. Let G be a group and let V be a G-module. Then V is *irreducible* if the only submodules of V are $\{0\}$ and V (note that these will always be submodules).

We now present the following theorem, due to Carter and Weyl:

Theorem 2. The set of all irreducible representations of $\mathrm{gl}\,(n,\mathbb{C})$ is indexed by partitions with at most n parts.

Let us call these irreducible representations V_μ. Now, for each s_λ we can write

$$s_\lambda = \left(\sum_i c_i (x_{i1} \otimes \cdots \otimes x_{ir_i}) \right) + J$$

for some $x_{ij} \in \mathrm{gl}\,(n,\mathbb{C})$ and some $c_i \in \mathbb{C}$, where the ideal J is as described above. We can then define an action of the s_λ on the V_μ by

$$s_\lambda \cdot v = \sum_i c_i \rho_\mu(x_{i1}) \circ \cdots \circ \rho_\mu(x_{ir_i})(v)$$

where $\rho_\mu : \mathrm{gl}\,(n,\mathbb{C}) \to \mathrm{gl}\,(V_\mu)$ is the group homomorphism giving rise to the representation V_μ, and $\circ$ denotes composition[2]. Now, at long last, we come to the relevance of the interpolation Jack polynomials. Recall from the previous section that an interpolation Jack polynomial is a specific polynomial in n variables $x_1, \ldots, x_n$, indexed by partitions of length at most n. Let us denote the interpolation Jack polynomial associated to the partition λ by P_λ. Then for any $v \in V_\mu$ we can write

$$s_\lambda \cdot v = P_\lambda^{\kappa=1}(\mu) v$$

Thus interpolation Jack polynomials are interesting because they give the eigenvalues of elements of a basis for the center of the universal enveloping algebra of $\mathrm{gl}\,(n,\mathbb{C})$ when that basis acts on irreducible $\mathrm{gl}\,(n,\mathbb{C})$-modules.

[2]Note that this action is independent of choice of representative for the coset $(\sum_i x_{i1} \otimes \cdots \otimes x_{ir_i}) + J$

3 Research Outcomes

Recall:

Definition 16. A *partition of an integer* m is a tuple $\lambda = (\lambda_1, \ldots, \lambda_n)$ such that $\lambda_1, \ldots, \lambda_n$ are non-negative integers, $\lambda_1 \geq \cdots \geq \lambda_n$, and $\sum_i \lambda_i = m$.

In what follows, let $\lambda = (D_1, D_2, D_3)$ and $\mu = (\mu_1, \mu_2, \mu_3)$ be partitions. Furthermore, for partitions μ and λ, let c_μ^λ be the coefficient of P_μ in the linear combination for $\frac{\partial}{\partial \kappa} P_\lambda$. We break up our conjectures into five sections: section 3.1 deals with which P_μ appear with a non-zero coefficient, section 3.2 deals with the case $\lambda = (D_1, D_2, D_3)$ where $D_1, D_2, D_3 \neq 0$, section 3.3 deals with the case $\lambda = (D_1, 0, 0)$ where $D_1 \neq 0$, section 3.4 deals with the case $\lambda = (D_1, D_2, 0)$ where $D_1, D_2 \neq 0$, and section 3.5 presents a possible generalization to n variables. In sections 3.2 and 3.3 we have a complete classification of c_μ^λ, and section 3.4 we have a partial classification.

3.1 Which P_μ Appear

The first question to be asked is which P_μ appear with a non-zero coefficient in the linear combination for $\frac{\partial}{\partial \kappa} P_\lambda$. In regards to this we have the following conjecture:

Conjecture 1. For partitions λ and μ as above, if

(a) $\sum \mu_i > \sum D_i$, or

(b) $\mu \geq \lambda$ in lexicographic ordering,

then P_μ appears with a zero coefficient in the linear combination for $\frac{\partial}{\partial \kappa} P_\lambda$. If λ has one or two non-zero parts then this is an if and only if condition. The case where λ has three non-zero parts is dealt with in more detail in section 3.2.

In the following, we assume that for $\lambda = (D_1, D_2, D_3)$ and $\mu = (\mu_1, \mu_2, \mu_3)$ we have $\sum \mu_i \leq \sum D_i$ and $\mu <_{lex} \lambda$ ($\mu < \lambda$ in lexicographical order).

3.2 Observation on (D_1, D_2, D_3)

For $D_1, D_2, D_3 \neq 0$ we have

$$c_{(\mu_1,\mu_2,\mu_3)}^{(D_1,D_2,D_3)} = \begin{cases} 0 & \text{if } \mu_3 - D_3 < 0 \\ c_{(\mu_1-D_3,\mu_2-D_3,\mu_3-D_3)}^{(D_1-D_3,D_2-D_3,0)} & \text{otherwise} \end{cases}$$

Thus the case where λ has three non-zero parts can be completely classified by looking at the case when λ has at most two non-zero parts.

3.3 Formulae for $\lambda = (D, 0, 0)$

Since the formulae often involve products of consecutive terms, we give the following definition to make the formulae more succinct.

Definition 17. Let $b > 0$. Then

$$a^{\overline{b}} := (a+0)(a+1)\cdots(a+b-1))$$

Note that in what follows, a will often be a polynomial, such as $\kappa + \mu_1$, $2\kappa + \mu_1$, etc.

Remark: when a consecutive product is written in this form, we should notice that $f(k)$ is the "start" of the product, and b is the number of terms in this product.

Example 1.

$$(2\kappa+\mu_1)^{\overline{D-\mu_1}} := (2\kappa+\mu_1)(2\kappa+\mu_1+1)\cdots(2\kappa+D-1)$$

We found that the formula for the coefficient of μ in $\lambda = (D_1, 0, 0)$ depends on the number of non-zero parts in μ. We thus present three formulae.

Formula 1. Conjecture for the coefficient of $(\mu_1, 0, 0)$ in $(D_1, 0, 0)$, where $\mu_1 \geq 0$:

$$\frac{(-1)^{D_1-\mu_1} \cdot \frac{D_1!}{(D_1-\mu_1)\ \mu_1!} \ \times\ [(2\kappa+\mu_1)^{\overline{D_1-\mu_1}} + (\kappa+\mu_1)^{\overline{D_1-\mu_1}}]}{(\kappa+\mu_1)^{\overline{D_1-\mu_1}}}$$

Formula 2. Conjecture for the coefficient of $(\mu_1, \mu_2, 0)$ in $(D_1, 0, 0)$, where $\mu_1, \mu_2 > 0$:

$$\frac{(-1)^{D_1-\mu_1+\mu_2} \cdot \ \frac{D_1!\ (D_1-\mu_1-1)!}{(D_1-\mu_1-\mu_2)!\ (\mu_1-\mu_2)!\ \mu_2!}}{(\kappa+\mu_1-\mu_2+1)^{\overline{\mu_2}} \cdot (\kappa+D_1-\mu_2)^{\overline{\mu_2}}}$$

Formula 3. Conjecture for the coefficient of (μ_1, μ_2, μ_3) in $(D_1, 0, 0)$, where $\mu_1, \mu_2, \mu_3 > 0$:

$$\frac{(-1)^{D_1-\mu_1+\mu_2} \cdot \frac{D_1!(\mu_2-1)!}{(\mu_1-\mu_2)!(\mu_2-\mu_3)!(\mu_3)!}(\kappa-\mu_3+1)^{\overline{\mu_3}} \cdot (\kappa+\mu_1-\mu_3+1)^{\overline{\mu_3-1}} \times S}{(\kappa+\mu_1-\mu_2+1)^{\overline{D_1-\mu_1+\mu_2-1}} \cdot (\kappa+\mu_2-\mu_3+1)^{\overline{\mu_3}} \cdot (2\kappa+\mu_1-\mu_3+1)^{\overline{\mu_3}}}$$

where S =

$$\sum_{j=0}^{D_1-\mu_1-\mu_2-\mu_3} \binom{D_1-\mu_1-\mu_2-1-j}{\mu_3-1} \binom{j+\mu_2-1}{\mu_2-1} (2\kappa+\mu_1)^{\overline{D_1-\mu_1-\mu_2-\mu_3-j}} \cdot (\kappa+D_1-\mu_2-j)^{\overline{j}}$$

The linear combination for P_λ generated by these three formulae has been checked to be accurate for all $D_1 \leq 33$ (the case of $D_1 \leq 33$ took more than 14 hours).

3.4 Partial Formulae for $\lambda = (D_1, D_2, 0)$

In this case we again found that the formula for the coefficient of P_μ depends on the number of non-zero parts in μ. We have yet to investigate the case where μ has three non-zero parts, so we present no conjecture for this case. For μ with exactly two non-zero parts we have a general conjecture, which is presented in formula 8. For μ with one non-zero part we were unable to find a general formula but have conjectures about some special cases, and a conjecture for a partial general formula. We start with the case where μ has exactly one non-zero part.

Formula 4. Conjecture for the coefficient of $(\mu_1, 0, 0)$ in $(D_1, 1, 0)$:

If $D_1 - \mu_1 > 0$:

$$\frac{(-1)^{D_1-\mu_1}\left[\frac{(D_1-1)!}{\mu_1!}\right]\ [(\kappa-1)(\kappa)]\ [(2\kappa+\mu_1)^{\overline{D_1-\mu_1-1}}]}{(\kappa+\mu_1-1)^{\overline{D_1-\mu_1}}}$$

If $D_1 - \mu_1 = 0$:

$$(-1)^{D_2} \cdot (D_2-1)! = (-1)^1 \cdot (D_2-1)! = -(D_2-1)!$$

This formula has been checked for all $D_1 \leq 15$ (as of Aug 24 at 11:50 pm).

Formula 5. Conjecture for the coefficient of $(\mu_1, 0, 0)$ in $(D_1, 2, 0)$:

If $D_1 - \mu_1 > 0$:

$$\frac{(-1)^{D_1-\mu_1}\left[\frac{(D_1-2)!}{\mu_1!}\right]\ [(\kappa-1)(\kappa)]\ [(2\kappa+\mu_1)^{\overline{D_1-\mu_1-2}}] \times S}{(\kappa+\mu_1-2)^{\overline{D_1-\mu_1}} \cdot (2\kappa+D_1-2)^{\overline{2-(D_1-\mu_1)}}}$$

where

$$S = (D_1-\mu_1-1)\kappa^2 - (3(D_1-1)+\mu_1)\kappa - 2(\mu_1-1)(D_1-1)$$

If $D_1 - \mu_1 = 0$:

$$(-1)^{D_2} \cdot (D_2-1)! = (-1)^2 \cdot (D_2-1)! = (D_2-1)!$$

This formula has been checked for all $D_1 \leq 15$ (as of Aug 24 at 11:50 pm).

Formula 6. Conjecture for the coefficient of $(\mu_1, 0, 0)$ in $(D_1, 3, 0)$:

If $D_1 - \mu_1 > 0$:

$$\frac{(-1)^{D_1-\mu_1}\left[\frac{(D_1-3)!}{\mu_1!}\right]\ [(3-1)!]\ [(\kappa-1)(\kappa)]\ [(2\kappa+\mu_1)^{\overline{D_1-\mu_1-3}}] \times S)}{(\kappa+\mu_1-3)^{\overline{D_1-\mu_1}} \cdot (2\kappa+D_1-3)^{\overline{3-(D_1-\mu_1)}}}$$

where

$$\begin{aligned}S =&\frac{1}{2}[(D_1-\mu_1-2)(D_1-\mu_1-1)\cdot\kappa^4\\&-4(D_1-\mu_1-1)(2D_1+\mu_1-4)\cdot\kappa^3\\&+((6D_1-7)\mu_1^2+(-6D_1^2+14D_1-9)u+23(D_1-2)(D_1-1))\cdot\kappa^2\\&+((6D_1-10)\mu_1^2+(18D_1^2-64D_1+54)\mu_1-28(D_1-2)(D_1-1))\cdot\kappa\\&+6(\mu_1-2)(\mu_1-1)(D_1-2)(D_1-1)]\end{aligned}$$

If $D_1-\mu_1=0$:

$$(-1)^{D_2}\cdot(D_2-1)!=(-1)^3\cdot(D_2-1)!$$

This formula has been checked for all $D_1\leq 15$ (as of Aug 24 at 11:50 pm).

We now present a partial formula for the coefficient of $(\mu_1,0,0)$ in $(D_1,D_2,0)$.

Formula 7. Conjecture for the coefficient of $(\mu_1,0,0)$ in $(D_1,D_2,0)$:

If $D_1-\mu>0$:

$$\frac{(-1)^{D_1-\mu_1}\left[\frac{(D_1-D_2)!}{\mu_1!}\right][(D_2-1)!]\ [(\kappa-1)(\kappa)]\ [(2\kappa+\mu_1)^{\overline{D_1-\mu_1-D_2}}]\cdot S}{(\kappa+\mu_1-D_2)^{\overline{D_1-\mu_1}}\cdot(2\kappa+D_1-D_2)^{\overline{D_2-(D_1-\mu_1)}}}$$

where S has:

$$\begin{aligned}\text{leading term }&=\binom{D_1-\mu_1-1}{D_2-1}\kappa^{2(D_2-1)}\\\text{constant term }&=(-1)^{D_2-1}\cdot D_2\cdot(\mu_1-D_2+1)^{\overline{D_2-1}}\cdot(D_1-D_2+1)^{\overline{D_2-1}}\end{aligned}$$

If $D_1-\mu_1=0$:

$$(-1)^{D_2}\cdot(D_2-1)!=(-1)^{D_1-\mu_1+D_2}\cdot(D_2-1)!$$

Finally, we present a formula for the coefficient of μ with exactly two non-zero parts in λ with exactly two non-zero parts.

Formula 8. Conjecture for the coefficient of $(\mu_1,\mu_2,0)$ in $(D_1,D_2,0)$ where $\mu_1,\mu_2>0$:

If $\mu_2>D_2$:

$$\frac{(-1)^{D_1+D_2-\mu_1+\mu_2}\cdot\frac{(D_1-D_2)!\ (D_1-\mu_1-1)!}{(\mu_1-\mu_2)!\ (\mu_2-D_2)!\ (D_1+D_2-\mu_1-\mu_2)!}}{(\kappa+\mu_1-\mu_2+1)^{\overline{\mu_2-D_2}}\ \cdot\ (\kappa+D_1-\mu_2)^{\overline{\mu_2-D_2}}}$$

3.5 Generalization to n-variable Case

We think **Formula 2** could be generalized to the n-variable case. That is, we conjecture that the coefficient of $(\mu_1,\mu_2,0,\ldots,0)$ in $(D,0,0\ldots,0)$, where $\mu_1,\mu_2>0$, is:

$$\frac{(-1)^{D-\mu_1+\mu_2}\cdot\ \frac{D!\ (D-\mu_1-1)!}{(D-\mu_1-\mu_2)!\ (\mu_1-\mu_2)!\ \mu_2!}}{(\kappa+\mu_1-\mu_2+1)^{\overline{\mu_2}}\cdot(\kappa+D-\mu_2)^{\overline{\mu_2}}}$$

Note that we have not tested this conjecture for $n\neq 3$.

4 Methods

4.1 Computer Code

(1) **Get polynomials P_λ**

We first followed the definitions of *interpolation Jack polynomials* and wrote the function named *getAnswer(allLambda, plot)* to generate them. The argument "allLambda" is the partition $\lambda = (D_1, D_2, D_3)$ and the argument "plot" is a boolean value passed in, where the function will plot all possible reverse fillings and print their corresponding polynomials if "plot" is set to be "True".

Below are function definitions in Appendix A.1:

- *getCopy(lsOfls)*: Takes in a list of lists named "lsOfls". Return the deep copy (each list in lsOfls is copied by value instead of by reference) of it.
- *removeNone(tList)*: If needed, remove "None" in a list of lists. This function is for display purpose.
- *xi(i)*: $xi(1) = x$, $xi(2) = y$ and $xi(3) = z$
- *aprime(s)*: $a'(s)$, as defined the formula for the interpolation Jack polynomials (see section 2.1.1).
- *lprime(s)*: $l'(s)$, as defined in the formula for the interpolation Jack polynomials (see section 2.1.1).
- *getTabPos(tabi)*: Return the locations (i, j) of a box s in a tableau named "tabi".
- *getTab12(tList)*: Return tableaux $\lambda^{(0)} = T, \lambda^{(1)}$, $\lambda^{(2)}$, $\lambda^{(3)} = \emptyset$ (as defined in the formula for the interpolation Jack polynomials (see section 2.1.1)) and the locations of their boxes.
- *initializeProperty12Marker(tabi)*: The argument "tabi" means "tab i", $\lambda^{(i)}$. We use two lists of lists of boolean values and call them "markers", where each boolean value in marker j represents whether the box s at the corresponding position satisfies property j ($j = 1, 2$, see **Definition 7**). All boxes have initial boolean value as "False".
- *countRowIdx(tabiPos, rowIdx)*: Takes in "tabiPos", which is a list of positions (i, j) of this tab i: $\lambda^{(i)}$ ($i = 0, 1, 2, 3$). Count number of boxes which has row index equal to "rowIdx".
- *countColumnIdx(tabiPos, colIdx)*: Takes in "tabiPos", which is a list of positions (i, j) of this tab i: $\lambda^{(i)}$ ($i = 0, 1, 2, 3$). Count number of boxes which has column index equal to "colIdx".
- *updateProperty1Marker(tList, tabiPos, tabiMinus1Pos, property1Marker)*: Takes in the list of lists which represents the original Young diagram: "tList", the positions of $\lambda^{(i)}$: "tabiPos", $\lambda^{(i-1)}$: "tabiMinus1Pos" (i=1,2,3), list of lists of boolean values: "property1Marker". Mark boxes by property 1. If a box s satisfies property 1, then the boolean value at the corresponding position in marker1 will be set to "True".

- *updateProperty2Marker(tList, tabiPos, tabiMinus1Pos, property2Marker)*: Takes in the list of lists which represents the original Young diagram: "tList", the positions of $\lambda^{(i)}$: "tabiPos", $\lambda^{(i-1)}$: "tabiMinus1Pos" (i=1,2,3), list of lists of boolean values: "property1Marker". Mark boxes by property 2. If a box s satisfies property 2, then the boolean value at the corresponding position in marker2 will be set to "True".
- *getRCi(tList, allTab, allTabPos, i)*: Return $R/C_{\lambda^{(i-1)}/\lambda^{(i)}}$ $(i = 1, 2, 3)$, as defined in the formula for the interpolation Jack polynomials (see section 2.1.1).
- *psi(tList)*: Return $\psi_T(\kappa)$, as defined in the formula for the interpolation Jack polynomials (see section 2.1.1).
- *b(u, s, k)*: Return $b_\mu(s, \kappa)$, as defined in the formula for the interpolation Jack polynomials (see section 2.1.1).
- *a(u, s)*: Return $a_\mu(s)$, as defined in the formula for the interpolation Jack polynomials (see section 2.1.1)
- *l(u, s)*: Return $l_\mu(s)$, as defined in the formula for the interpolation Jack polynomials (see section 2.1.1).
- *initUnfilledPositionsStart(tListStart)*: Takes in a list of lists named "tListStart", which represents the Young diagram. Initialize and return the list of all unfilled positions, which is just the list of all positions (i,j) for all boxes s in this Young diagram.
- *fill(tList, unfilledPositions, allFilledLists)*: Takes in the original reverse filling, represented by a list of lists named "tList", the positions of unfilled boxes: "unfilledPositions", and current list of reverse fillings obtained: "allFilledLists". Pop one box from "unfilledPositions", call *fillEntry()* to fill this box with one of 1,2, or 3 (if possible) and recurse.
- *fillEntry(i, j, num, tList, unfilledPositions, allFilledLists)*: Helper function for *fill(tList, unfilledPositions, allFilledLists)*. Fill entry (i, j) with "num".
- *plotTabs(allFilledLists)*: Each element in "allFilledLists" is a list of lists which represents a reverse filling. This function plots all those filled tableaux.
- *getAllReverseT(allLambda)*: Return tableaux of all possible reverse fillings of shape allLambda.
- *getPHelper(allFilledLists, plot)*: Each element in "allFilledLists" is a list of lists which represents a reverse filling. Return final polynomial P_λ by adding the polynomials for all reverse fillings.
- *getAnswer(allLambda, plot)*: Given a partition "allLambda" = λ= (D_1, D_2, D_3) and $D_1 \geq D_2 \geq D_3 \geq 0$, return the polynomial P_λ by calling *getPHelper()*.

(2) **Get Linear Combinations**

getLinearComb(P) is the function which takes in the polynomial P_λ and outputs the linear combination for $\frac{\partial}{\partial \kappa} P_\lambda$. It first uses the built-in functions of SageMath to get the leading monomial *lm()* and the leading coefficient *lc()* of $\frac{\partial}{\partial \kappa} P_\lambda$. Then, we set

$\mu = (\mu_1, \mu_2, \mu_3)$ to be the degrees of x, y, z in this leading monomial, respectively. Finally we subtract $lc \cdot P_\mu$ from $\frac{\partial}{\partial \kappa} P_\lambda$ and repeat this process.

Pseudo Code for *getLinearComb(P)*:

Algorithm 1 Get the Linear Combination for $\frac{\partial}{\partial \kappa} P_\lambda$

```
dP ← partial derivative of P_λ with respect to κ
linComb = ∅

while dP ≠ 0 do
    lm ← leading monomial of dP
    lc ← leading coefficient of dP
    u1, u2, u3 ← degree of x,y,z in lm, respectively

    Pu ← getAnswer(u) for u = (u1, u2, u3)
    PuCoef ← coefficient of lm in Pu
    coef = (lc / PuCoef)
    dP ← dP - coef * Pu
    linComb.append(coef, (u1, u2, u3))
return linComb
```

Below are function definitions in Appendix A.2:

- *getLmAndDegs(poly)*: Takes in a polynomial "poly". Return a tuple "(lm, lc, degs)", which contains the leading monomial "lm" of "poly", the leading coefficient "lc" of "poly", and the degrees "degs" of x, y, z in that leading monomial.
- *getLinearComb(P)*: Takes in "P" which is P_λ. Return the linear combination for $\frac{\partial}{\partial \kappa} P_\lambda$ as a list of tuple, where each tuple $(coef, \mu)$ contains the coefficient of P_μ in $\frac{\partial}{\partial \kappa} P_\lambda$ and μ itself.

(3) **Other Helper Functions**

Below are function definitions in Appendix A.3:

- *checkSymmetric(poly)*: Check whether a polynomial is symmetric.
- *part(n, k)*: Partition number n into k parts. Return a list of tuples of size k where each tuple is a partition of n.
- *partition(n)*: Partition number n into any number of parts (all possible partitions of n). Return a list of tuples where each tuple is a partition of n.
- *getMuLessThanLam(lams)*: Takes in a partition "lams". Return all $\mu = (\mu_1, \mu_2, \mu_3)$ such that $\sum \mu_i < \sum D_i$ where "lams" is $\lambda = (D_1, D_2, D_3)$.
- *getLexLessThan(lowestU, lams)*: Takes in two partitions "lowestU" and "lams". Return all $\mu = (\mu_1, \mu_2, \mu_3)$ such that $\mu <_{lex}$ "lowerU" and $\sum \mu_i < \sum D_i$ where "lams" is $\lambda = (D_1, D_2, D_3)$.

(4) **Check Conjectures**

Appendix A.4 contains the code to check our conjectures. Below are function definitions in Appendix A.4.

- *formulai(lam, u):* Function to return polynomial conjectured by **Formula i** ($i = 1, 2, \ldots, 6$).
- *CheckFormulas(lam, u):* Function that combines conjectures altogether. If we have a conjecture for the coefficient of P_u in $\frac{\partial}{\partial \kappa} P_{lam}$, it returns the result generated by the conjectured formula. Otherwise, it returns "None".

(5) **Improve Program Efficiency**

- Instead of using *Tableau* class in SageMath, we used the list of lists as the structure for the tableaux and only convert it to tableaux when needed (such as for plotting).
- We used dictionaries to stored the P_λ, $\frac{\partial}{\partial \kappa} P_\lambda$, and the linear combination for $\frac{\partial}{\partial \kappa} P_\lambda$ for those λ we have calculated. The dictioanries are called *pDict*,*dpDict*, and *tripleDict* respectively. We use the "%store magic" to store the dictionaries before shutting down the kernal in Jupyter Notebook and use *%store -r* to refresh from store.

(5) **Checking Conjectures**

- Our first method to check a conjecture for coefficents c_μ^λ is to calculate the linear combination for $\frac{\partial}{\partial \kappa} P_\lambda$ and compare it with our guess.
- The second method is to use our conjectured formulas to calculate our guess for c_μ^λ and then subtract $c_\mu^\lambda \cdot P_\mu$ from $\frac{\partial}{\partial \kappa} P_\lambda$. Check if $\frac{\partial}{\partial \kappa} P_\lambda$ become 0 eventually. In this way, we avoid the process of calculating the linear combinations, and thus would improve the time efficiency.

4.2 Techniques to Obtain Conjectures

(1) **Set** $i = D - \mu_1$

In a lot of cases we will see the patterns are clearer when we change $D - \mu_1$ instead of changing D and μ_1 independently. Therefore, we set $i = D - \mu_1$ and conjecture the formula as a function depending on i and other parameters. In the end, we substitute i with $D - \mu_1$ in our conjecture. One of the advantages is that if we fix $D - \mu_1$ (comparing to fixing D or μ_1), then the change of the coefficients of the polynomial $c_\mu^\lambda(\kappa)$ along the change of μ_1 would be clearer and in many cases it would be easy to guess the coefficients of κ as a polynomial of μ_1 (or D). Especially when expand the polynomial $c_\mu^\lambda(\kappa)$, the coefficients of each κ^n ($n = 0, 1, 2, \ldots$) are usually polynomials of μ_1 with increasing or decreasing degree as n increase.

Example:

Take $i = 7$, $\mu = (D-7, 2, 2)$ for example. When $\mu = (2, 2, 2), (3, 2, 2), (4, 2, 2), (5, 2, 2), \ldots$, the unfactored parts in c_μ^λ are (below we only show the unfactored part):

$$
\begin{aligned}
(9,0,0) \to (2,2,2) &= -(36k^3 + 276k^2 + 696k + 576) \cdot (\dots) \\
(10,0,0) \to (3,2,2) &= -(36k^3 + 345k^2 + 1101k + 1170) \cdot (\dots) \\
(11,0,0) \to (4,2,2) &= -(36k^3 + 414k^2 + 1596k + 2064) \cdot (\dots) \\
(12,0,0) \to (5,2,2) &= -(36k^3 + 483k^2 + 2181k + 3318) \cdot (\dots) \\
&\dots
\end{aligned}
$$

We can see the coefficient of k^3 is constant, the coefficient of k^2 is linear in μ_1, the coefficient of k^1 is quadractic in μ_1, and the coefficient of k^0 is cubic in μ_1. Then it is easy to use Mathematica to solve each coefficient and obtain the general formula of the unfactored part in the above examples:

$$-(36k^3 + (69\mu_1 + 138)k^2 + (45\mu_1^2 + 180\mu_1 + 156)k + (10\mu_1^3 + 60\mu_1^2 + 104\mu_1 + 48)).$$

(2) **Add Cancelled Terms**

In some cases the numerators and the denominator of c_μ^λ can be completely factored into products of linear terms of κ. However, in many cases there's some parts of it that cannot be factored into products of linear terms. In this situation, we first look at more examples, with all other variables fixed except one (for example μ_1). Then determine the degree of the part that cannot be factored. In this step we often need to add some terms to both the numerator and the denominator or give back some terms to the unfactored part.

Example: Take $i = D - \mu_1 = 2$. We use Sage to factor and look at the coefficient c_μ^λ where $\lambda = (D, 0, 0)$ and $\mu = (D-2, 0, 0)$. Below we use "$\lambda \to \mu$" to represent c_μ^λ:

$$
\begin{aligned}
(2,0,0) \to (0,0,0) &= \frac{(5k+3)}{(k+1)} \\
(3,0,0) \to (1,0,0) &= \frac{3 \cdot (5k+4)}{(k+2)} \\
(4,0,0) \to (2,0,0) &= \frac{2 \cdot 3 \cdot (5k^2 + 15k + 12)}{(k+2)(k+3)} \\
(5,0,0) \to (3,0,0) &= \frac{2 \cdot 5 \cdot (5k^2 + 21k + 24)}{(k+3)(k+4)} \\
(6,0,0) \to (4,0,0) &= \frac{3 \cdot 5 \cdot (5k^2 + 27k + 40)}{(k+4)(k+5)} \\
&\dots
\end{aligned}
$$

From these examples it is reasonable to guess that there will be an unfactored part should of degree 2 and that has the form $5k^2 + \dots$. Adding terms to some numerators and denominators would make the pattern clearer:

$$(2,0,0) \to (0,0,0) = \frac{1 \cdot (5k+3)k}{k(k+1)} = \frac{1 \cdot (5k^2+3k+0)}{k(k+1)}$$
$$(3,0,0) \to (1,0,0) = \frac{3 \cdot (5k+4)(k+1)}{(k+1)(k+2)} = \frac{3 \cdot (5k^2+9k+4)}{(k+1)(k+2)}$$
$$(4,0,0) \to (2,0,0) = \frac{6 \cdot (5k^2+15k+12)}{(k+2)(k+3)}$$
$$(5,0,0) \to (3,0,0) = \frac{10 \cdot (5k^2+21k+24)}{(k+3)(k+4)}$$
$$(6,0,0) \to (4,0,0) = \frac{15 \cdot (5k^2+27k+40)}{(k+4)(k+5)}$$
$$\dots$$

(2) **Products of (2k+...)**

Another situation that appears very often is in some part of the c_μ^λ, for example, the denominator, as we fix $D-\mu_1$ and increase μ_1, we might see $(2k+1),(2k+1),(2k+3),(2k+3),(2k+5),(2k+5)\dots$. In fact, we should add the cancelled term and make them $(2k+0)(2k+1),(2k+1)(2k+2),(2k+3)(2k+3),(2k+3)(2k+4),(2k+4)(2k+5),(2k+5)(2k+6)\dots$. The reason is that the pattern usually is: c_μ^λ can be factored into parts where most of the parts are the product of consecutive numbers (such as $\mu_1 \cdot (\mu_1+1)\cdots(D-1)D$) or consecutive linear terms of κ (such as $(k+\mu_1)(k+\mu_1+1)\cdots(k+D-1)(k+D)$).

Example:

$$(9,0,0) \to (3,3,3) = \frac{\cdots}{\dots(2k+1)(2k+3)}$$
$$(10,0,0) \to (4,3,3) = \frac{\cdots}{\cdots(2k+3)}$$
$$(11,0,0) \to (5,3,3) = \frac{\cdots}{\cdots(2k+3)(2k+5)}$$
$$(12,0,0) \to (6,3,3) = \frac{\cdots}{\cdots(2k+5)}$$
$$\dots$$

should be organized to:

$$(9,0,0) \to (3,3,3) = \frac{\cdots}{\dots(2k+1)(2k+2)(2k+3)}$$
$$(10,0,0) \to (4,3,3) = \frac{\cdots}{\cdots(2k+2)(2k+3)(2k+4)}$$
$$(11,0,0) \to (5,3,3) = \frac{\cdots}{\cdots(2k+3)(2k+4)(2k+5)}$$
$$(12,0,0) \to (6,3,3) = \frac{\cdots}{\cdots(2k+4)(2k+5)(2k+6)}$$
$$\dots$$

(3) **Leading Term and Constant Term**

The leading term and the constant term will provide a lot of information, especially when they can be factored. We will illustrate the advantages of looking at them through the following example for $(D,0,0) \to (D-i,0,0)$.

Example:

$(D,0,0) \to (D-\mu_1,0,0)$ has some part which cannot be factored. For example, if we have obtained the following formulas for this unfactored part and want to combine them to get a general formula.

$$
\begin{aligned}
(D,0,0) \to (D-1,0,0) &= \frac{\cdots[3k+2(D-1)]}{\cdots} \\
(D,0,0) \to (D-2,0,0) &= \frac{\cdots[9k^3+\cdots+2(D-2)(D-1)]}{\cdots} \\
(D,0,0) \to (D-3,0,0) &= \frac{\cdots[17k^4+\cdots+2(D-3)(D-2)(D-1)]}{\cdots} \\
(D,0,0) \to (D-4,0,0) &= \frac{\cdots[33k^5+\cdots+2(D-4)(D-3)(D-2)(D-1)]}{\cdots} \\
(D,0,0) \to (D-5,0,0) &= \frac{\cdots[65k^6+\cdots+2(D-5)(D-4)(D-3)(D-2)(D-1)]}{\cdots} \\
&\cdots
\end{aligned}
$$

We first look at the leading terms: $3k, 9k^3, 17k^4, 33k^5, 65k^6$. Notice we have

$$
\begin{aligned}
3 &= 2^1+1 \\
9 &= 2^2+1 \\
17 &= 2^3+1 \\
33 &= 2^4+1 \\
65 &= 2^5+1 \\
&\cdots
\end{aligned}
$$

Then it is reasonable to guess maybe the actual formula is

$$(2k+\dots)\cdots(2k+\dots)+(k^{i-1}+\dots)$$

Combining the constant term, which has $(D-i)\cdots(D-1)$, we now conjecture:

$$(2k+D-i)\cdots(2k+D-1)+(k^{i-1}+\dots)$$

Since the constant term is $2\cdot(D-i)\cdots(D-1)$ and moreover,

$$
\begin{aligned}
3 &= 2^1 + 1 = 2^1 + 1^1 \\
9 &= 2^2 + 1 = 2^2 + 1^2 \\
17 &= 2^3 + 1 = 2^3 + 1^3 \\
33 &= 2^4 + 1 = 2^4 + 1^4 \\
65 &= 2^5 + 1 = 2^5 + 1^5 \\
&\dots,
\end{aligned}
$$

we update the conjecture to be

$$
\begin{aligned}
&(2k + D - i) \cdots (2k + D - 1) + (k + D - i) \cdots (k + D - 1) \\
=&(2k + \mu_1) \cdots (2k + D - 1) + (k + \mu_1) \cdots (k + D - 1) \\
=&(2k + \mu_1)^{\overline{D-\mu_1}} + (k + \mu_1)^{\overline{D-\mu_1}}
\end{aligned}
$$

Indeed, our final conjecture (**Formula 1**) contains this part.

(4) **Pascal's Identity and Binomial Coefficients**

When we compare examples, sometimes if we take out part of the polynomial and compare that part, we would see they satisfy the Pascal's identity. This indicates that the binomial coefficients are possibly involved.

Example:

For example, this shows up when we study the unfactored part of $(D, 0, 0) \to (D - i, \mu_2, \mu_3)$ $(\mu_2, \mu_3 > 0)$. After we fixed $\mu_3 = 1$, we look at the leading coefficient and have noticed that it changes along the change of μ_2 and i. We thus write it as a function $f(\mu_2, i)$ depending on i and μ_2. Then $f(1, 2) = 1, f(1, 3) = 3, f(1, 4) = 7 \dots$.

	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$	$i=8$	$i=9$	$\dots$
$D-i,1,1$	1	$3k$	$7k^2$	$15k^3$	$31k^4$	$63k^5$	$127k^6$	$255k^7$	$\dots$
$D-i,2,1$		1	$4k$	$11k^2$	$26k^3$	$57k^4$	$120k^5$	$247k^6$	$\dots$
$D-i,3,1$			1	$5k$	$16k^2$	$42k^3$	$99k^4$	$219k^5$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$

If we look at the first two rows of the table above, notice $3+1 = 4, 7+4 = 11, 15+11 = 26, 31 + 26 = 57, \dots$ The key observation is that

$$f(\mu_2, i) = f(\mu_2, i - 1) + f(\mu_2 - 1, i - 1).$$

This should remind us of Pascal's identity

$$\binom{n}{k} = \binom{n}{k-1} + \binom{n-1}{k-1}.$$

Moreover, the leading coefficients of the first row has a clear pattern: $2^{i-1}-1$. Using this as an initial condition and the recursive formula above, we could write out the formula for $f(\mu_2, i)$ as a function of i for each fixed μ_2.

Using the pattern $2^{i-1}-1$, we first conjecture it to be the difference of a consecutive product of $(2k+\dots)$ and a consecutive product of $(k+\dots)$, but it does not work. Later we apply the formula for geometric sum:

$$\begin{aligned} &2^{i-1}-1 \\ =&2^{i-2}+2^{i-3}+\cdots+2^1+2^0, \end{aligned}$$

and change the conjecture to be a sum of consecutive products of $(2k+\dots)$. This interpretation was checked to be correct for many examples. By changing the values for μ_3, we obtain another binomial coefficient and conjecture the unfactored part of $(D,0,0) \to (D-i, \mu_2, \mu_3)$ $(\mu_2, \mu_3 > 0)$ as the following:

$$\sum_{j=0}^{D-\mu_1-\mu_2-\mu_3} \binom{D-\mu_1-\mu_2-1-j}{\mu_3-1} \binom{j+\mu_2-1}{\mu_2-1} (2k+\mu_1)^{\overline{D-\mu_1-\mu_2-\mu_3-j}} \cdot (k+D-\mu_2-j)^{\overline{j}}$$

5 Future Plans

All of the formulae presented here are conjectures, based on code written in Sage. In the future we would like to try to prove these formulae using theory, and also to complete the formulas for which we have only a partial formula.

6 Acknowledgements

We would like to thank Dr. Salmasian from the University of Ottawa for mentoring us over the summer of 2019 at the Fields Undergraduate Summer Research Program (FUSRP), and the Fields Institute for Research in the Mathematical Sciences for hosting us for this program.

References

[1] Erdmann, K., Wildon, M. J.. Introduction to Lie Algebras, Springer Undergraduate Mathematics Series. London, Springer-Verlag London, Ltd, 2006.

[2] Humphreys, James. Introduction to Lie algebras and representation theory. New York, Springer-Verlag New York Inc, 1972.

[3] Macdonald, I. G.. Symmetric functions and Hall polynomials, second edition. New York, Oxford University Press Inc, 1995.

[4] Sagan, Bruce. The symmetric group: representations, combinatorial algorithms, and symmetric functions, first edition. Belmont, California, Wadsworth Inc, 1991.

A Appendix: Code

A.1 Code to Generate Polynomials

```
#Deep copy a list of list
def getCopy(lsOfls):
    ret = []
    for ls in lsOfls:
        ret.append(copy(ls))
    return ret

#If needed, remove "None" in a list of list (see "getTab12(tList)")
def removeNone(tList):
    tListCopy = getCopy(tList)
    tList = []
    for idx, row in enumerate(tListCopy):
        newRow = []
        for e in row:
            if e!=None:
                newRow.append(e)
        tList.append(newRow) #we need to keep [] for the u tableau later
    return tList


#x1 is x; x2 is y; x3 is z
def xi(i):
    if i == 1:
        return x-2*k
    elif i == 2:
        return y-k
    return z

#a'(s)
def aprime(s): # position s = tuple (i,j)
    (i,j) = s #Xiaomin: Notice i,j should start with 1, so we add 1 below:
    return (j+1)-1
#l'(s)
def lprime(s): # position s = tuple (i,j)
    (i,j) = s #Xiaomin: Notice i,j should start with 1, so we add 1 below:
    return (i+1)-1



#-------Calculate lambda^(0)=T, lambda^(1), lambda^(2), lambda^(3)={} ---------
```

```
#Return the locations (i,j) of boxes s in a tableau
def getTabPos(tabi):
    ret = []
    for r in range(len(tabi)):
        for c in range(len(tabi[r])):
            if tabi[r][c] != None:
                ret.append((r,c))
    return ret


#Return  tableaus lambda^(0)=T, lambda^(1), lambda^(2), lambda^(3)={}
# and the locations of their boxes
def getTab12(tList):
    T = Tableau(tList)
    tab12 = []
    tab12Pos = []
    for i in [1,2]:
        tabi = T.anti_restrict(i)
        tabi = tabi.to_list()
        tabiPos = getTabPos(tabi)

#         tabi = removeNone(tabi)
#Xiaomin: if needed, could keep the 'None'.
# This is more for display purpose

        tab12.append(tabi)
        tab12Pos.append(tabiPos)
    return (tab12, tab12Pos)



#----------------------Calculate  R/C... ----------------------
#We use two boolean markers to mark if a box satisfies properties or not.

#Initialize the markers for proerty 1 and proerty 2 (all boxes have "False")
def initializeProperty12Marker(tabi):
    marker = []
    for i,row in enumerate(tabi):
        newRow = []
        for e in row:
            if e!=None:
                newRow.append(False)
        if newRow != []:
            marker.append(newRow)
    marker2 = getCopy(marker)
```

```
    return marker, marker2

#count number of boxes in row with index = rowIdx
def countRowIdx(tabiPos, rowIdx):
    count = 0
    for (i,j) in tabiPos:
        if i==rowIdx:
            count += 1
    return count

#count number of boxes in row with index = colIdx
def countColumnIdx(tabiPos, colIdx):
    count = 0
    for (i,j) in tabiPos:
        if j==colIdx:
            count += 1
    return count

#Mark boxes by property1
def updateProperty1Marker(tList, tabiPos, tabiMinus1Pos, property1Marker):
    for rowIdx in range(len(property1Marker)):
        countI = countRowIdx(tabiPos, rowIdx)
        #count (rowIdx, ?) from tabiPos = num of boxes in tabi for row idx
        countIMinus1 = countRowIdx(tabiMinus1Pos, rowIdx)
        #similar, count row idx elements in tabiMinus1

        if countIMinus1 > countI: #then all s in this row has property1
            property1Marker[rowIdx] = [True for ij in property1Marker[rowIdx]]
            #update this row!


#Mark boxes by property2
def updateProperty2Marker(tList, tabiPos, tabiMinus1Pos, property2Marker):
    if property2Marker==[]:
        return
    for colIdx in range(len(property2Marker[0])):
        countI = countColumnIdx(tabiPos, colIdx)
        #count (colIdx, ?) from tabiPos = num of boxes in tabi for column idx
        countIMinus1 = countColumnIdx(tabiMinus1Pos, colIdx)
        #similar, count column idx elements in tabiMinus1

        if countIMinus1 <= countI: #then all s in this column has property 2
            for i in range(len(property2Marker)):
                if len(property2Marker[i]) >= (colIdx+1):
                    property2Marker[i][colIdx] = True #update this row!
```

```
#Return R/C_(i-1)/i i=1,2,3
def getRCi(tList, allTab, allTabPos, i):
    tabiPos = allTabPos[i]
    tabiMinus1Pos = allTabPos[i-1]
    RCi = []

    property1Marker, property2Marker = initializeProperty12Marker(allTab[i])
    updateProperty1Marker(tList, tabiPos, tabiMinus1Pos, property1Marker)
    updateProperty2Marker(tList, tabiPos, tabiMinus1Pos, property2Marker)

    for (i,j) in tabiPos: #since all s in R/C come from tableau i
        if property1Marker[i][j]==False:
            continue
        if property2Marker[i][j]==False:
            continue
        RCi.append((i,j))
    return RCi




#-----------------------Calculate  psi_T(k) ----------------------

# psi_T(k)
def psi(tList):

    # Get Tableau lambda^(0),lambda^(1),lambda^(2),lambda^(3)
    tab0 = tList #Young diagram with boxes of numbers 1,2,3
    (tab1, tab2), (tab1Pos, tab2Pos) = getTab12(tList)
    #tab1 = Young diagram with boxes of numbers 2,3
    #tab2 = Young diagram with boxes of numbers 3
    tab3 = {}
    allTab = [tab0, tab1, tab2, tab3]

    # Get Tableau positions (i,j)
    tab0Pos = getTabPos(allTab[0])
    tab3Pos = getTabPos(allTab[3])
    allTabPos = [tab0Pos, tab1Pos, tab2Pos, tab3Pos]


    # Now calculate all RC
    allRC = []
    for i in [1,2,3]:
```

```
        RCi = getRCi(tList, allTab, allTabPos, i)
        allRC.append(RCi)


    # Finally, calculate psi_T(k)
    ret = 1
    for i in [1,2,3]:
        for s in allRC[i-1]: #since the first of allRC is allRC[0]
            tabi = allTab[i]
            tabiMinus1 = allTab[i-1]
            tabi = removeNone(tabi)
            tabiMinus1 = removeNone(tabiMinus1)

            ret *=  b(tabi, s, k) / b(tabiMinus1, s, k)  #s = position (i,j)
    return ret

#b_u(s,k)
def b(u, s, k): #u is a Young diagram.
    aa = a(u, s)
    ll = l(u, s)
    return (aa + k*(ll+1)) / (aa + k*ll + 1)
#a_(s)
def a(u, s):
    (i,j) = s
    return len(u[i]) - (j+1) #Xiaomin: Notice i,j should start with 1
#l_(s)
def l(u, s):
    (i,j) = s
    i+=1; j+=1 #Xiaomin: Notice i,j should start with 1

    count = 0
    for m in range(i+1, 4):  #i < m <=3
        if len(u)>=m and len(u[m-1]) >= j:
            count += 1
    return count


#-------Find all valid filling (reverse tableau) of 3,2,1----------------


#Initialize unfilledPositionsStart, return the list of all positions
def initUnfilledPositionsStart(tListStart):
    unfilledPositionsStart = []
    for i in range(len(tListStart)):
        for j in range(len(tListStart[i])):
```

```
            unfilledPositionsStart.append((i,j))
    return unfilledPositionsStart



#Fill entry (i,j) with num
def fillEntry(i,j,num, tList, unfilledPositions, allFilledLists):
    #fill num into entry i,j:
    copyList = getCopy(tList)
    copyList[i][j] = num

    #pop unfilledPositions:
    copyUnfilledPositions = copy(unfilledPositions)
    copyUnfilledPositions.pop(0)

    #continue fill:
    fill(copyList, copyUnfilledPositions, allFilledLists)


# Fill the whole tableau with numbers 1,2,3
def fill(tList, unfilledPositions, allFilledLists):
    #-------------Done! return--------------
    if len(unfilledPositions) == 0:
        allFilledLists.append(tList)
        return


    #---------------Fill:------------------
    (i,j) = unfilledPositions[0]
    if i==0: #1st row
        if j==0:
            for num in [3,2,1]:
                fillEntry(i,j,num, tList, unfilledPositions, allFilledLists)
        else: #j>0:
            reverseRange = range(1, tList[i][j-1]+1)
            reverseRange.reverse()
            for num in reverseRange:
                fillEntry(i,j,num, tList, unfilledPositions, allFilledLists)

    else: #2nd or 3rd row
        if j==0:
            for num in [3,2,1]:
                if num < tList[i-1][j]: # compare with the "up" neighber
                    fillEntry(i,j,num, tList, unfilledPositions, allFilledLists)
        else: #j>0:
```

```
            reverseRange = range(1, tList[i][j-1]+1)
            reverseRange.reverse()
            for num in reverseRange:
                if num < tList[i-1][j]:
                    fillEntry(i,j,num, tList, unfilledPositions, allFilledLists)



# Plot the filled tableaus
def plotTabs(allFilledLists):
    tableaus = []
    for tList in allFilledLists:
        T = Tableau(tList)
        tableaus.append(T)
        show(T.plot(descents=False))
    #     print ascii_art(T), "\n"
    print "count tableaus = ", len(tableaus)


# Get all Reverse Tableaus of filling 1,2,3
def getAllReverseT(allLambda):
    tListStart = []
    for i in range(3):
        if allLambda[i]!=0:
            tListStart.append([None for i in range(allLambda[i])])

    unfilledPositionsStart = initUnfilledPositionsStart(tListStart)

    allFilledLists = []   # will store all Reverse Tableaus of filling 1,2,3
    fill(tListStart, unfilledPositionsStart, allFilledLists)

    return allFilledLists




#Given lambda = (lambda1, lambda2, lambda3).
# Already know lambda1 >= lambda2 >= lambda3 >=0

# -------MAIN: All fillings will be stored into "allFilledLists"-----

def getPHelper(allFilledLists, plot):
    P = R(0) # poly ring
    if plot:
        print "count tableaus = ", len(allFilledLists)
```

```
    for T in allFilledLists:
        currT = 1
        for i in range(len(T)):
            for j in range(len(T[i])):
                s = (i,j)
                currT *= (xi(T[i][j]) - aprime(s) + lprime(s)*k)
        P += currT * psi(T)
        if plot:
            print "====================================\n***Curr Tableau is:"
            show(Tableau(T).plot(descents=False))
            print "its polynomial is = "
            show(currT)
            print "===================================="
    return P



def getAnswer(allLambda, plot):
    allFilledLists = getAllReverseT(allLambda)
#     print "After filling, allFilledLists =", allFilledLists

    answer = getPHelper(allFilledLists, plot)
    return answer









#Main code to set up environment=============================================

# # T.<k> = QQbar[]
#Xiaomin: 1. construct a rational field of variable 'k'
# T.<k> = ZZ[]
#Xiaomin: 1. construct a rational field of variable 'k'

# FT = FractionField(T)
#Xiaomin: 2. Extend it to a rational function field of 'k'
# # FT = T

# R.<x,y,z> = PolynomialRing(FT, order='lex')
#Xiaomin: polynomial of x,y,z over 'FT' (rational function field of 'k')
```

```
# # print R
# F = FractionField(R) #
#Xiaomin: extend the x,y,z polynomial ring to a rational function field


# # print F

# #-------------above is just to construct correct ring or field---------------
# pDict = {}
# tripleDict = {}


# # pDictSmall = {}

# # import time
# # start_time = time.time()
# # %store -r pDictSmall
# # print "pDictSmall retrieve done"
# # print("---took time  %s seconds ---" % (time.time() - start_time))



# # start_time = time.time()
# # %store -r tripleDict
# # print "tripleDict retrieve done"
# # print("---took time %s seconds ---" % (time.time() - start_time))
```

A.2 Code to Get the Linear Combinations

```
# Return the leading monomial and the degrees of x,y,z in it.
def getLmAndDegs(poly):
    lm = poly.lm()
    lc = poly.lc()

    xdeg = lm.degree(x)
    ydeg = lm.degree(y)
    zdeg = lm.degree(z)

    return lm, lc, (xdeg,ydeg,zdeg)


    return part1 + part2 + part3
```

```
#Return the linear combination for dP_lambda
def getLinearComb(P):
    diffP = diff(P, k)
    poly = diffP

    # triples of (coef, u, Pu)--changed to tuple of (coef, u)
    ret = []

    while poly != 0:

        lm, lc,degs = getLmAndDegs(poly)
        Pu = getAnswer(degs, plot=False)
        Pu *= x^0 #Xiaomin: for the integers to have type "polynomial"
        coef1 = lc
        coef2 = Pu.monomial_coefficient(lm)
        coef = coef1 / coef2

        ret.append((coef,degs))
        poly = poly - coef * Pu

    return ret
```

A.3 Other Helper Functions

```
# Check if a polynomial is symmetric
def checkSymmetric(poly):
    if poly != poly.subs(x=y, y=x):
        return False
    if poly != poly.subs(x=z, z=x):
        return False
    if poly != poly.subs(y=z, z=y):
        return False
    return True

import numpy as np

#Partition number n into k parts
def part(n, k):
    def _part(n, k, pre):
        if n <= 0:
            return []
        if k == 1:
            if n <= pre:
```

```
                return [[n]]
            return []
        ret = []
        for i in range(min(pre, n), 0, -1):
            ret += [[i] + sub for sub in _part(n-i, k-1, i)]
        return ret
    return _part(n, k, n)

#Partition number n into any number of parts
def partition(n):
    part3 = part(n,3)
    part2 = part(n,2)
    part1 = part(n,1)
    for ls in part2:
        ls.append(0)
    for ls in part1:
        ls += [0,0]


# Get all u such that sum(u) < sum(lambda)
def getMuLessThanLam(lams):
    count = 0
    ret = []
    s = sum(lams)
    print "All u such that sum(u)<sum(lambda):"
    for x1 in [0..s]:
        for y1 in [0..min(s-x1, x1)]:
            for z1 in [0..min(s-x1-y1, y1)]:
                if x1 > lams[0]:
                    continue
                if (x1,y1,z1)==lams:
                    continue
                else:
                    count += 1
                    ret.append((x1,y1,z1))

    # print "COUNT =", count
    return ret


# Return A > B in lex order
def lexBigger(A, B):
    (a1,a2,a3) = A
    (b1,b2,b3) = B
```

```
    if a1 > b1:
        return True
    elif a1==b1 and a2>b2:
        return True
    elif a1==b1 and a2==b2 and a3>b3:
        return True
    else:
        return False


# Return u < lowerU in lexicographical order, but
# still the sum cannot be bigger than lams
def getLexLessThan(lowestU, lams):
    count = 0
    ret = []
    print "All u s.t.lex(u)<lex(lowestU): (sum at least should < sum(lams))"


    for x1 in [0..lowestU[0]-1]:
        for y1 in [0..x1]:
            for z1 in [0..y1]:
                    if x1+y1+z1 > sum(lams):
                        continue
                    count += 1
                    ret.append((x1,y1,z1))


    for x1 in [lowestU[0]]:
        for y1 in [0..x1]:
            for z1 in [0..y1]:
                if lexBigger(lowestU, (x1,y1,z1)):
                    if x1+y1+z1 > sum(lams):
                        continue
                    count += 1
                    ret.append((x1,y1,z1))

    # print "COUNT =", count
    return ret
```

A.4 Code to Check Conjectures

```
import math
def fac(x):
    return math.factorial(x)*k^0
```

```
# (D,0,0) -> (u1,0,0) #old name: formula1_1
def formula1(lam, u):
    D,D2,D3 = lam
    u1,u2,u3 = u

    numerator = (-1)^(D-u1) * fac(D) / (fac(u1)*(D-u1))

    # (2*k+u1)..(2*k+D-1)
    part2k = 1
    for n in [(2*k+u1)..(2*k+D-1)]:
        part2k *= n

    #  (*k+u1)..(k+D-1)
    partk = 1
    for n in [(k+u1)..(k+D-1)]:
        partk *= n

    return numerator * (part2k+partk) / partk


# (D,0,0) -> (u1,u2,0) #old name: formula1_2
def formula2(lam, u):
    D,D2,D3 = lam
    u1,u2,u3 = u

#     ----------numerator------------
    numerator = (-1)^(D-u1+u2) * fac(D)*fac(D-u1-1) / (fac(D-u1-u2)*fac(u1-u2)*fac(u2))

#     ----------denominator----------
    denominator = 1
    #(k+u1-u2+1)..(k+u1)
    for n in [(k+u1-u2+1)..(k+u1)]:
        denominator *= n

    #(k+D-u2)..(k+D-1)
    for n in [(k+D-u2)..(k+D-1)]:
        denominator *= n

    return numerator/denominator
```

```
# (D,0,0) -> (u1,u2,u3) #old name: formula1_3
from scipy.special import comb
def formula3(lam, u):
    D,D2,D3 = lam
    u1,u2,u3 = u

#     -----------numerator------------
#   Constant Part
    const = (-1)^(D-u1+u2) * fac(D)*fac(u2-1) / (fac(u1-u2)*fac(u2-u3)*fac(u3))


#   (k-u3+1)..k
    partkUp = 1
    for i in [(k-u3+1)..k]:
        partkUp *= i

#   (k-u3+1)..k
    partkUp2 = 1
    for i in [(k+u1-u3+1)..(k+u1-1)]:
        partkUp2 *= i


#   lastPart
    lastPart = 0
    for j in [0..D-u1-u2-u3]:
        curr = Integer(comb(D-u1-u2-1-j,u3-1))  *  Integer(comb(j+u2-1,j))
        for n in [(2*k+u1)..(2*k+ D-u2-u3-1 -j)]:
            curr *= n
        for n in [(k+ D-u2-j)..(k+ D-u2-1)]:
            curr *= n
        lastPart += curr


#     -----------denominator----------

#   (k+u1-u2+1)..(k+D-1)
    partDown1 = 1
    for i in [(k+u1-u2+1)..(k+D-1)]:
        partDown1 *= i

#   (k+u2-u3+1)..(k+u2)
    partDown2 = 1
    for i in [(k+u2-u3+1)..(k+u2)]:
        partDown2 *= i
```

```
#   (2k+u1-u3+1)..(2k+u1)
    part2k = 1
    for i in [(2*k+u1-u3+1)..(2*k+u1)]:
        part2k *= i

    return (const*partkUp*partkUp2*lastPart) / (partDown1*partDown2*part2k)




# (D1,1,0) -> (u1,0,0) #old name: formula2_1
def formula4(lam, u):
    D1,D2 = lam[0],lam[1]
    u1,u2,u3 = u


#Case 1: D1-u1==0
    if D1-u1==0:
        return (-1)^D2 * fac(D2-1)


#Case 2: D1-u1>0
    numerator = (-1)^(D1-u1) * fac(D1-D2) / fac(u1) *  k*(k-1)
    # (2*k+u1)..(2*k+D1-2)
    for n in [(2*k+u1)..(2*k+D1-D2-1)]:
        numerator *= n

    denominator = 1
    for n in [(k+u1-1)..(k+D1-2)]:
        denominator *= n

    return numerator / denominator




# (D1,2,0) -> (u1,0,0) #old name: formula2_2
def formula5(lam, u):
    D1,D2 = lam[0],lam[1]
    u1,u2,u3 = u


#Case 1: D-u1==0
```

```
    if D1-u1==0:
        return (-1)^D2 * fac(D2-1)


#Case 2: D1-u1>0
    numerator = (-1)^(D1-u1) * fac(D1-D2) / fac(u1) *  k*(k-1)
    # (2*k+u1)..(2*k+D1-3)
    for n in [(2*k+u1)..(2*k+D1-D2-1)]:
        numerator *= n
    lastPart = (D1-u1-1)*k^2 - (3*D1+u1-3)*k - 2*(D1-1)*(u1-1)
    numerator *= lastPart


    denominator = 1
    # (k+u1-2)..(k+D1-3)
    for n in [(k+u1-D2)..(k+D1-D2-1)]:
        denominator *= n

    # (2k+D1-2)..(2k+u1-1)
    for n in [(2*k+D1-D2)..(2*k+u1-1)]:
        denominator *= n

    return numerator / denominator



# (D1,3,0) -> (u1,0,0) #old name: formula2_3
def formula6(lam, u):
    D1,D2 = lam[0],lam[1]
    u1,u2,u3 = u


#Case 1: D1-u1==0
    if D1-u1==0:
        return (-1)^D2 * fac(D2-1)


#Case 2: D1-u1>0
    numerator = (-1)^(D1-u1) * fac(D1-D2) / fac(u1) *  k*(k-1)  * fac(D2-1)
    # (2*k+u1)..(2*k+D1-4)
    for n in [(2*k+u1)..(2*k+D1-D2-1)]:
        numerator *= n

    lastPart = (D1-u1-2)*(D1-u1-1)/2 * k^4
    lastPart += -4*(D1-u1-1)*(2*D1+u1-4)/2 * k^3
```

```
    lastPart += +((6*D1-7)*u1^2+ (-6*D1^2+ 14*D1-9)*u1+ 23*(D1-2)*(D1-1))/2 * k^2
    lastPart += +((6*D1-10)*u1^2+ (18*D1^2-64*D1+ 54)*u1-28*(D1-2)*(D1-1))/2 * k
    lastPart += +6*(u1-2)*(u1-1)*(D1-2)*(D1-1)/2
    numerator *= lastPart


    denominator = 1
    # (k+u1-3)..(k+D1-4)
    for n in [(k+u1-D2)..(k+D1-D2-1)]:
        denominator *= n
    # (2k+D1-3)..(2k+u1-1)
    for n in [(2*k+D1-D2)..(2*k+u1-1)]:
        denominator *= n

    return numerator / denominator


# Since formula3: (D1,D2,0) -> (u1,u2,0) has many conditions, we leave it out here
# we will write it here after we have a full formula for it


#============================================================================
def CheckFormulas(lam, u):
    D1,D2,D3 = lam
    u1,u2,u3 = u

    if D3>0:
        return None

    # (D1,D2,0) -> u
    if D2 > 0:
        if u2>0:
            return None #(we only have formulas for ())
        if D2==1:
            return formula4(lam,u) #(D1,1,0) -> (u1,0,0)
        elif D2==2:
            return formula5(lam,u) #(D1,2,0) -> (u1,0,0)
        elif D2==3:
            return formula6(lam,u) #(D1,3,0) -> (u1,0,0)

    # (D1,0,0) -> u
    else: #D2==0:
        if u3>0:
            return formula1(lam,u) #(D1,D2,0) -> (u1,u2,u3)
        elif u2>0:
```

```
            return formula2(lam,u) #(D1,D2,0) -> (u1,u2,0)
        else:
            return formula3(lam,u) #(D1,D2,0) -> (u1,0,0)
    return None




#Main code to check formulas
# D2 = 0
# for D in [D2..13]:
#     lam = (D,D2,0)
#     print "-----------------lam=",lam,"---------------"
#     # P = getAnswer(lam, False)
#     # for (coef, u) in getLinearComb(P):
#     for (coef, u) in tripleDict[lam]:
#         D,D2,D3 = lam
#         u1,u2,u3=u
#         conjCoef = CheckFormulas(lam, u)
#         print coef==conjCoef
#         if coef!=conjCoef:
#             print("***coef   =", coef, ",   u = ", u)
#             print("conjCoef  =", conjCoef, ",   u = ", u)
```