Computational Methods in the Study of Diagram Algebras Using SAGE

Pavel Javornik

1 Introduction

The purpose of the following document is to introduce the reader to combinatorial representation theory, centralizer algebras, and diagram algebras via computational methods. We do so with the assistance of SAGE, an open-source mathematical programming language. In addition to using SAGE's built-in libraries, we design algorithms to implement modern, combinatorial techniques in the study of multiplicity-free families of algebras. To that end, we make some contributions to the development of SAGE's algebra packages by doing the following:

1. defining standard representations of diagram algebras as they act on self-tensored vector spaces;
2. designing and implementing an algorithm to calculate the central, primitive, orthogonal idempotents of the Brauer Algebra;
3. exploring the various ways that the idempotents may help us in understanding how modules decompose into direct sums of isotypic components of the underlying algebras.

This document is meant to be an interactive one. While the following printout could simply be read, we encourage the reader to follow along by downloading the jupyter notebook file (.ipynb format) on my website [Ja]. Once the file is downloaded, it may be opened as a SAGE jupyter notebook [SAGE]. All relevant information about the use of SAGE's various libraries can be found on that website.

Before proceeding any further, we would need to define many of the important objects that we will be studying. We will then motivate the use of combinatorial representation theory by explaining the importance of the Artin-Wedderburn theorems and Schur-Weyl duality, and the meshing of the two to help us understand the semisimple, classical Lie group/Lie algebra representations.

In section 3, we begin using SAGE to see how far canonical linear algebra methods can take us in understanding the one-dimensional irreducible submodules. In section 4, we will begin applying our understanding of idempotents and centralizer algebras to fully decompose symmetric group algebra modules.

In the final section, we apply combinatorial methods to the implementation of standard representations of semisimple diagram algebras acting on self-tensored spaces and the calculation of central, primitive, orthogonal idempotents of the semisimple Brauer subalgebra.

2 Preliminaries on Representation Theory

Many of the following definitions and methods can be found in Chapters 2, 3, 5, and appendix chapters C and D of Goodman and Wallach's *Representations and Invariants of the Classical Groups* [GoWall].

We say that the set A is a unital algebra if $(A, +, \cdot)$ is a ring with unity for which $(A, +)$ is a vector space over a field F, and

$$\alpha(a \cdot b) = (\alpha a) \cdot b = a \cdot (\alpha b)$$

for all $a, b \in A$ and $\alpha \in F$. If G is a group, then the set $FG = \text{span}_F\{g \ : \ g \in G\}$ that is the F-linear span of the elements of G and has a canonical unital algebra structure we refer to as the group algebra. For example if $G = S_2 = \{\text{id}, (1\ 2)\}$ is the symmetric group acting on 2 elements, addition is defined as $(\alpha_1, \alpha_2) + (\beta_1, \beta_2) = (\alpha_1 \text{id} + \alpha_2(1\ 2)) + (\beta_1 \text{id} + \beta_2(1\ 2)) = ((\alpha_1 + \beta_1)\text{id} + (\alpha_2, \beta_2)(1\ 2)) = (\alpha_1 + \beta_1, \alpha_2 + \beta_2)$. As for multiplication,

$$(\alpha_1, \alpha_2)\cdot(\beta_1, \beta_2) = (\alpha_1 \text{id} + \alpha_2(1\ 2))\cdot(\beta_1 \text{id} + \beta_2(1\ 2)) = \alpha_1\beta_1 \text{id}^2 + \alpha_1\beta_2 \text{id}(1\ 2) + \alpha_2\beta_1(1\ 2)\text{id} + \alpha_2\beta_2(1\ 2)^2$$

$$= (\alpha_1\beta_1 + \alpha_2\beta_2, \alpha_1\beta_2 + \alpha_2\beta_1)$$

If R is a ring, we say an abelian group $(M, +)$ is a left R module if there is a left action $R \times M \to M$ $(m \mapsto r \cdot m)$ satisfying:

1. $1_R \cdot m = m$ for all $m \in M$(Identity-Preserving)
2. $a(b \cdot m) = (ab) \cdot m$ for all $a, b \in R$, $m \in M$ (Associative Action)
3. $(a + b) \cdot m = a \cdot m + b \cdot m$ for all $a, b \in R$, $m \in M$ (Right-Distributive over R-addition)
4. $a \cdot (m + n) = a \cdot m + b \cdot n$ for all $a \in R$, $m, n \in M$ (Left-Distributive over M-addition)

If M is a vector space over a field F, then we also require that 5. $a\cdot(\alpha m + \beta n) = \alpha a \cdot m + \beta a \cdot n$ (Left-Distributive and F-linear over M-addition)

Since A is a unital ring, we would define a left A module the exact same way. Let B be another unital algebra.

We say M is a right B module if instead acting by B happened on the right: $m \cdot b \in M$.

1. $m \cdot 1_B = m$ for all $m \in M$(Identity-Preserving)
2. $(m \cdot a) \cdot b = m \cdot (ab)$ for all $a, b \in B$, $m \in M$ (Associative Action)
3. $m \cdot (a + b) = m \cdot a + m \cdot b$ for all $a, b \in B$, $m \in M$ (Left-Distributive over R-addition)
4. $(m + n) \cdot a = m \cdot a + n \cdot a$ for all $a \in R$, $m, n \in M$ (Right-Distributive over M-addition)

We say that M is an (A,B) bimodule if:

1. M is a left A module and a right B module
2. $a(mb) = (am)b$ for all $a \in A, b \in B, m \in M$

Sometimes we call M an $A \otimes B$ left module if $A \otimes B$ has a left action of the form $a \otimes b \cdot m = a(mb) = (am)b$. But this only makes sense when the actions of A and B commute with one another, as it wouldn't matter what action would be applied first.

2.1 Ideals, Semisimplicity, and Decompositions

We say that $Q \subset R$ is a left ideal if $rq \in Q$ for all $r \in R$ and $q \in Q$, and Q is an abelian group with addition is restricted to Q. Q is a right ideal if instead $qr \in Q$.

Let M, N be R modules. We say $N \subset M$ is a submodule if $a \cdot n \in N$ for all $a \in R$ and $(N, +)$ is an abelian group. All rings and algebras, R are natural left R modules under action by multiplication from the left, and therefore any left ideal is automatically a submodule. For modules, we fix a conventional left or right action, and focus explicitly on that.

A set $B \subset A$ is said to be a subalgebra of A if in addition to being a submodule, $(B, +)$ is a subspace of $(A, +)$ over the field F.

A module W is said to be simple if its only submodules are $0 = \{0_W\}$ and itself. An algebra A (or ring R) is simple if its only left ideals are 0 and itself.

We say that V is semisimple if it decomposes into a direct sum of simple submodules. We say that A is semisimple if every A module is semisimple. It turns out that A is semisimple if and only if A as an A module is semisimple.

2.2 Algebra, ring, and group representations

Let R be a unital ring and V a vector space over a field F. We say a representation of a ring R is an F-linear, unital homomorphism $\rho : R \to \text{End}(V)$. satisfying:

1. $\rho(a+b) = \rho(a) + \rho(b)$ for all $a, b \in R$ (Additive-Homomorphism)
2. $\rho(a \cdot b) = \rho(a) \circ \rho(b)$ for all $a, b \in R$ (Multiplicative-Homomorphism)
3. $\rho(1_R) = \text{id}_V$ (Identity-Preserving)

For an algebra A, in order for $\rho : A \to \text{End}(V)$ to be an algebra representation it would have to satisfy:

1. $\rho(\alpha a + \beta b) = \alpha\rho(a) + \beta\rho(b)$ for all $a, b \in A$, $\alpha, \beta \in F$ (Linearity)
2. $\rho(\alpha a \cdot b) = \alpha\rho(a) \circ \rho(b)$ for all $a, b \in A$, $\alpha \in F$ (Multiplicative-Homomorphism)
3. $\rho(1_A) = \text{id}_V$ (Identity-Preserving)

It wouldn't make much sense to have the vector space and the algebra to be over different fields, unless A's field is a subfield of V's. But for our purposes we will just work with examples where they are over the same field.

For a group G, we say that $\rho : G \to \text{End}(V)$ is a group representation if

1. $\rho(a \cdot b) = \rho(a) \circ \rho(b)$ for all $a, b \in G$ (Multiplicative-Homomorphism)
2. $\rho(1_G) = \text{id}_V$ (Identity-Preserving)

Representations are useful tools in understanding rings as endomorphism rings of vector spaces. With these definitions, any vector space inherits a structure as a left R module by defining the action as $r \cdot v = \rho(r)(v)$. It also means that we may rely on linear algebra tools to solve certain problems. We usually refer to a representation as the ordered pair (ρ, V).

Let ρ be an algebra representation of A. We say (π, W) is a subrepresentation of (ρ, V) if W is a subspace of V, and for all $a \in A$ we have that $\pi(a) = \rho(a)|_W$, i.e. the action of A preserves W. In that sense W is just a submodule of V as A modules.

A representation is said to be irreducible (or simple) if its only subrepresentations are itself and the trivial subrepresentation, where the subspace consists of only the zero vector. With these definitions it is easy to see that the irreducible representations of a ring are equivalent to the simple submodules of vector spaces. And therefore, the full decomposition of a space according to the action of algebra is isomorphic to the decomposition of the representation as the direct sum of irreducible representations.

For example, let $G = S_2 = \{1, (1\ 2)\}$ be the set of permutations of $\{1, 2\}$ and $F = \mathbb{C}$. Then FG is a two-dimensional algebra. The standard representation sends

$$1 \mapsto \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, (1\ 2) \mapsto \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

with ordered basis $(1, (1\ 2))$. As a module, this decomposition is $\mathbb{C}\{1+(1\ 2)\} \oplus \mathbb{C}\{1-(1\ 2)\} = \mathbb{C}\{v_1\} \oplus \mathbb{C}\{v_2\}$, and so $\rho = \rho_+ \oplus \rho_-$, where $\rho_+((1\ 2))v_1 = v_1$, and $\rho_-((1\ 2))v_2 = -v_2$. With this new basis,

$$1 \mapsto 1 \oplus 1,\ (1\ 2) \mapsto 1 \oplus (-1)$$

The gist is then in order to understand an A module, it is best to understand the set of irreducible representations, or simple A modules $\{A^\lambda\}$, where A^λ is a summand of the semisimple decomposition of A. If M is an A module, then we define $M^\lambda = \bigoplus_{A^\lambda \cong U \subset M} U \cong \bigoplus_{i=1}^{m_\lambda} A^\lambda = m_\lambda A^\lambda$, i.e. the sum of distinct submodules isomorphic to A^λ. m_λ is the multiplicity of the of A^λ in M.

An important tool in the characterization of simple submodules is given by the trace form of the representation (ρ, V). If $\rho = \rho_{\lambda_1} \oplus ... \oplus \rho_{\lambda_n}$ is a semisimple decomposition of the representation acting on the semisimple decomposition $V = V^{\lambda_1} \oplus ... \oplus V^{\lambda_n}$, we define the character of a representation ρ_{λ_k} to be $\chi_{\lambda_k} : A \to F$, where $\chi_{\lambda_k}(a) = \text{tr}(\rho_{\lambda_k}(a))$.

2.3 Lie algebras and Lie algebra representations

A Lie algebra is a bit of a misnomer as it is not an algebra at all. A Lie algebra is a vector space, $\mathfrak{g}$, over a field equipped with a bilinear Lie bracket $[\cdot,\cdot] : \mathfrak{g} \otimes \mathfrak{g} \to \mathfrak{g}$ such that:

1. $[x, y] = -[y, x]$ (Skew-symmetric)
2. $[x, [y, z]] + [y, [z, x]] + [z, [y, x]]=0$ (Jacobi Identity)

A common example is $\mathbb{R}^3$ in which the Lie bracket is the cross product of two vectors.

A Lie Algebra representation is a map $\rho : \mathfrak{g} \to \mathrm{End}(V)$ such that for all $x, y \in \mathfrak{g}$ and $\alpha, \beta \in F$:

1. $\rho(\alpha x + \beta y) = \alpha\rho(x) + \beta\rho(y)$ (F-linearity)
2. $\rho([x, y]) = \rho(x)\rho(y) - \rho(y)\rho(x)$ (Commutator Under Bracket)

While a Lie algebra is not an algebra, its universal enveloping algebra, $U_{\mathfrak{g}}$ is. The universal enveloping algebra of a Lie algebra is the (often infinite dimensional) free algebra over F generated by the basis elements of $\mathfrak{g}$, say $x_1, x_2, ...$ with additional relations. If $[x_i, x_j] = \sum_k c_k^{i,j} x_k$ in the Lie algebra, then the relations in $U_{\mathfrak{g}}$ are given as $x_i x_j - x_j x_i = \sum_k c_k^{i,j} x_k$. In essence it encodes every possible representation of a Lie algebra. This means that a representation of a Lie algebra can be extended to its universal enveloping algebra, and the two representations are equivalent and admit the exact same decompositions of spaces.

2.4 A taste of Lie methods

An important consequence is that $(xy) \cdot v = ([x, y] + yx) \cdot v$ for any $x, y \in U_{\mathfrak{g}}$. This identity is extremely useful in the study of Lie algebras. For example if $[x, y] = h$ and v is an eigenvector of h, with eigenvalue λ and we know that $x \cdot v = 0$ then $x \cdot (y \cdot v) = (\lambda + 0)v = \lambda v$, and we see that $(y \cdot v) \mapsto \lambda v$ under the action of x. This is useful in what is known as a highest-weight decomposition by the element h. If V is a complex vector space, there must exist one eigenvector v^+ for which $hv^+ = \mu v^+$, assuming $h \neq 0$. Since V is finite-dimensional, there is some $k > 0$ for which $x^{k+1}v = 0$, but $x^k v = v \neq 0$ as long as $h(x^k)v^+ = \mu_k$, for distinct μ_k. This ensures that each $x^k v^+$ isn't a scalar multiple of another. Therefore, this v exists. Working backwards, let $h \cdot v = \lambda v$, and show that the eigenvalues of $\{v, yv, y^2v, ..., y^dv\}$ under h are all distinct, and $y^{d+1}v = 0$. Then $\{v, yv, y^2v, ..., y^dv\}$ forms a d+1 dimensional subspace, called $L(d)$. If x sends $y^k v$ to a scalar multiple of $y^{k-1}v$ for all $k = 1, ..., d$, (and v to 0) then we know that $L(d)$ is an $\mathbb{C}\{x, y, h\}$-invariant subspace, or simple submodule, of V. The triple (x, y, h) is called an $\mathfrak{sl}_2$ triple, and h is what is known as a Cartan element for x and y.

2.5 Connections between Lie groups and Lie algebras

A Lie algebra is naturally associated to a Lie group, G, as long as G is both a group and a smooth (real or complex) manifold (a manifold whose charts to $\mathbb{R}$ or $\mathbb{C}$ and transition functions are C^∞) and the left action $G \times G \to G$ defined as $(a, b) \mapsto a^{-1}b$ is a smooth map from the product manifold to G. A Lie algebra is then the (complex or real) vector space T_1G, the tangent space of G at the identity element $1 \in G$. For example, the quotient space $\mathbb{R}/\mathbb{Z} \cong S^1$ is a real Lie group, and its associated Lie algebra is isomorphic to $\mathbb{R}$.

2.6 Some more examples

The complex Lie group of 2×2 determinant one matrices with complex entries, $SL_2(\mathbb{C})$ has the Lie algebra $\mathfrak{sl}_2(\mathbb{C})$ of the $\mathbb{C}$ linear span of traceless complex matrices. It has a canonical basis as $\mathbb{C}$ vector space $\{x, y, h\}$, with $x = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$, $y = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$, $h = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$. Moreover, $[x, y] = xy - yx$ is the commutator Lie bracket. The $\mathfrak{sl}_2$ triple that we mentioned earlier here is (x, y, h). In general, $\mathfrak{sl}_n(\mathbb{C})$ is the $\mathbb{C}$ span of traceless $n \times n$ complex matrices.

For the group $SL_2(\mathbb{R})$, the basis is the same, except $\mathfrak{sl}_2(\mathbb{R})$ is the $\mathbb{R}$ span of those matrices.

We define the complexification of a real Lie algebra, $\mathfrak{g}$ as the vector space $\mathfrak{g}_{\mathbb{C}} = \mathbb{C} \otimes_{\mathbb{R}} \mathfrak{g}$. It is no coincidence that the complexification of $\mathfrak{sl}_2(\mathbb{R})$ is $\mathfrak{sl}_2(\mathbb{C})$. It is true that the complex representations of the complexification of real Lie algebra are equivalent to the complex representations of that real Lie algebra.

However, it is true that the representation of a simply connected Lie group G is equivalent to its Lie algebra representation. Lie algebras are therefore important tools in studying how various vector spaces decompose under the action of a Lie group. Moreover, many Lie groups are not finitely generated, e.g. S^1. As finite-dimensional vector spaces, its Lie algebra as a vector space is considered finitely generated. If G is not simply connected, there are other specific criterias for when the complexified algebra representation is equivalent to the real group representation.

For example, let $SO_3(\mathbb{R})$ be the group of rotation matrices. The complexification of its Lie algebra, $\mathfrak{so}_3(\mathbb{R})$ is the $\mathbb{C}$ span of matrices:

$$J_x = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}, J_y = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}, J_z = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

denoted $\mathfrak{so}_3(\mathbb{C})$. The Lie bracket here is also just the matrix commutator. This Lie algebra is isomorphic to $\mathfrak{sl}_2(\mathbb{C})$ by taking $J_x \mapsto \frac{1}{2}(x-y)$, $J_y \mapsto \frac{i}{2}(x+y)$, and $J_z \mapsto \frac{ih}{2}$. A Lie algebra homomorphism is a homomorphism of vector spaces, ψ, such that $\psi([A,B]) = [\psi(A), \psi(B)]$ for all A, B and respective Lie brackets. It is an isomorphism if it is an isomorphism of vector spaces. From here on we consider an $\mathfrak{so}_3(\mathbb{C})$ representation as a $\mathfrak{sl}_2(\mathbb{C})$ representation.

We know of a decomposition of invariant subspaces via the highest weigth decomposition using eigenvalues of $\rho(h)$. One great criteria for checking if a representation of $\mathfrak{sl}_2(\mathbb{C})$ lifts to a representation of $SO_3(\mathbb{R})$ is that it happens when $e^{i\pi\rho(h)} = \mathrm{id}_V$, where $e^A = \sum_{k=0}^{\infty} A^k/k!$ for $A \in \mathrm{End}(V)$. So if $V \cong L(d) = \mathbb{C}v \oplus \mathbb{C}yv \oplus ... \oplus \mathbb{C}y^d v$, and $\rho(h) = \mathrm{diag}(\lambda_0, ..., \lambda_d)$ with respect to this basis, we would require that $e^{i\pi\lambda_k} = 1$ for all k, i.e. each λ_k is an even integer, as seen on pg. 83 in Alexander Kirillov's "An Introduction to Lie Groups and Lie Algebras" [Ki].

The connection here is that the exponential function take the Lie algebra of $SO_3(\mathbb{R})$ to the Lie Group via a parametrization, i.e. e^{tJ_z} maps to the rotation of $\mathbb{R}^3$ about the z-axis by t radians. A Lie algebra representation ψ is equivalent to its Lie group representation if there exists an $\Psi : G \to GL(V)$ for which $\Psi(e^X) = e^{\psi(X)}$ for all X in the Lie algebra.

By our isomorphism, this is identical to saying $e^{2\pi\psi(J_z)}$ acts trivially on V, i.e. the eigenvalues of $\psi(J_z)$ are all of the form im for $m \in \mathbb{Z}$. To prove why this would work, requires us to look at the lifted representation of ψ onto the simply connected universal cover of $SO_2(\mathbb{R})$, which is isomorphic to the real special unitary group $SU(2)$. The complexification of its Lie algebra is $\mathfrak{sl}_2(\mathbb{C})$, and therefore implies that $SU(2)$'s complex representations are equivalent to $\mathfrak{sl}_2(\mathbb{C})$'s complex representations, so there is a representation $\widetilde{\Psi}$ on $SU(2)$ that is equivalent to ψ. If one can show that for the covering map $p : SU(2) \to SO_3(\mathbb{R})$, if $\ker p \subset \ker\widetilde{\Psi}$, then $\widetilde{\Psi}$ descends to a representation Ψ on $SO_3(\mathbb{R})$ for which $\Psi \circ p = \widetilde{\Psi}$.

2.7 Idempotents

Another vastly important tool in understanding module decompositions involves the calculation of primitive, central, orthogonal idempotents. Recall an idempotent of a ring, R, is an element e for which $e^2 = e$. We say that e is:

1. Central if e commutes with every element in R

We say that a collection of distinct idempotents $\{e_\lambda\}$ and $\lambda \in \Lambda$ are:

1. Primitive if $\sum_\lambda e_\lambda = 1$, the identity element in R
2. Orthogonal if for any two e_λ, e_μ we have $e_\lambda e_\mu = 0$ if $\lambda \neq \mu$

Theorem 2.1. *Let A be a non-trivial, finite-dimensional, unital algebra over a field F. Let M be an A module. Then the following are equivalent:*

1. *The left-regular module A admits a full decomposition as $A \cong \bigoplus_{\lambda\in\Lambda} Ae_\lambda$, where $\{e_\lambda\}$ are a set of central, primitive, orthogonal idempotents satisfying $\sum e_\lambda = 1_A$ and $Ae_\lambda = A^\lambda$*
2. *A decomposes as the direct sum of endomorphism rings over division rings, i.e. $A \cong \bigoplus_{i=1}^{\ell} M_{n_i\times n_i}(\Delta_i)$*
3. *A is semisimple*

The equivalency follows directly from a theorem of Wedderburn, and a theorem of Goodman and Wallach's [GoWall, 128,133]. Since A is an algebra, it follows automatically that each Δ_i is actually just F. The idempotents become important in the study of centralizer algebras because they project the module, say V, in question onto a direct sum of isomorphic components $n_\lambda V_\lambda$, where V_λ is a representative of the isomorphism class λ. Thus, up to a choice of basis, one can identify each idempotent with an element of Λ, and $V \cong \bigoplus_{\lambda\in\hat{V}} V^\lambda$, where $V^\lambda \cong n_\lambda Ae_\lambda$, and $\lambda \in \hat{V}$ satisfies that $V^\lambda \neq 0$.

2.8 Idempotents of group algebras

Let FG be a finite-dimensional, semisimple group algebra. We define the standard representation $\rho : FG \to \mathrm{End}(FG)$ by taking the group element to its action on the basis $\{g \ : \ g \in G\}$ and extending F-linearly. Let $\rho = \oplus_\lambda \rho_\lambda$ be the semisimple decomposition of ρ. Then the primitive, central, idempotent corresponding to λ is given as [GoWall, 153]:

$$e_\lambda = \frac{\chi_\lambda(1_G)}{|G|}\sum_{g\in G}\chi_\lambda(g^{-1})g$$

Going back to our previous example, with $F = \mathbb{C}$ and $G = S_2$ we have

$$e_+ = \chi_+(1_G)/|G|(\chi_+(1^{-1})\cdot 1 + \chi_+((1\ 2)^{-1})\cdot(1\ 2)) = \frac{1}{2}(1_G + (1\ 2))$$

$$e_- = \chi_-(1_G)/|G|(\chi_-(1^{-1})\cdot 1 + \chi_-((1\ 2)^{-1})\cdot(1\ 2)) = \frac{1}{2}(1_G - (1\ 2))$$

Note that e_+ projects onto the subspace for which (1 2) acts by scaling by 1, and e_- projects onto the subspace for which (1 2) acts by scaling by -1.

The following is a theorem commonly attributed to Issai Schur and Hermann Weyl, and is a primary focus of this paper. It is the basis of study of centralizer algebras, and goes by the names of Schur-Weyl Duality, the Double Centralizer Theorem, and the Double Commutant Theorem [GoWall, 137].

Theorem 2.2. *Let M be a vector space, and $A \subseteq End(M)$. If the algebra B is semisimple, and $B = End_A(M) = \{\phi \in End(M) \ : \ \phi(\psi(m)) = \psi(\phi(m))\ \forall\psi \in A,\ \forall m \in M\}$, then $End_B(M) = A$ and M has a multiplicity-free complete decomposition:*

$$M \cong \bigoplus_{\lambda\in\Lambda} A^\lambda \otimes B^\lambda$$

as an $A \otimes B$-bimodule. Here B^λ are mutually, non-isomorphic, simple B modules, where $\lambda \in \Lambda$, an indexing set for the simple submodules of B. The B^λ's are known as isotypic components of M.

Moreover, $A^\lambda \otimes B^\lambda \cong m_M(\lambda)A^\lambda \cong \dim(A^\lambda)B^\lambda$ as A modules and B modules, respectively. That is to say that $A^\lambda \otimes B^\lambda$ is isomorphic to a number of copies of isotypic components of M as a B module. Here $m_M(\lambda)$ is the multiplicity of the submodule A^λ in M.

The next sections will primarily focus on studying the centralizer algebras of Lie groups and and Lie algebras to show the interplay between combinatorial representation theory, Artin-Wedderburn, and Schur-Weyl duality.

3 Computing the isotypic components of one-dimensional submodules of the symmetric group algebra

Let $V = \mathbb{C}^n$, and $M = V^{\otimes k}$. We will verify that ${}_{\mathbb{C}S_k}M$'s one-dimensional isotypic components, as a $\mathfrak{g} = U_{\mathfrak{sl}_n(\mathbb{C})}$ module with the diagonal action on simple tensors: $g \cdot (v_1 \oplus ... \oplus v_k) = gv_1 \oplus ... \oplus v_k + ... + v_1 \oplus ... \oplus gv_k$ decomposes according to the Double-Centralizer Theorem. It is thanks to a theorem of Schur that we know $\mathrm{End}_{\mathfrak{g}}(M) = \mathbb{C}S_k$, where S_k acts on the simple tensors of M by permutation of their orders [GoWall, 372]. And so to understand a decomposition of M under the action of the Lie algebra, it suffices to study its decomposition as a $\mathbb{C}S_k$ module.

We need to find a representation $\rho : S = S_k \to \mathrm{End(M)}$ as it permutes the basis vectos of M before extending it linearly to the group algebra. But first, we need to adopt some kind of convention for the basis vectors of M.

Let V be the $\mathbb{C}$-span of standard basis vectors $\{e_0, ..., e_{n-1}\}$ Let $u_{\sum_{j=1}^{k} i_j} = e_{i_1} \otimes ... \otimes e_{i_k}$, with $i_j \in \{0, ..., n-1\}$. This gives a bijective relationship between the n^k basis vectors, and the set $\{0, ..., n-1\}^k$, i.e. the base n representation of the indexing set for $U = \{u_j\}$.

An element $\sigma \in S_k$ acts on a basis element in U by $\sigma \cdot e_{i_1} \otimes ... \otimes e_{i_k} = e_{i_{\sigma(1)}} \otimes ... \otimes e_{i_{\sigma(k)}}$.

If $k = 3$ and $n = 2$, then $(1\ 2) \cdot u_5 = (1\ 2) \cdot e_1 \otimes e_0 \otimes e_1 = e_0 \otimes e_1 \otimes e_1 = u_3$. For the generators of S_3, we have:

$$(1\ 2) \mapsto \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$(1\ 3) \mapsto \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Now we need to create a function that takes an element in S_k to its element in End(M)

```
def rho(g,n,k):
    """
    function takes an element g in S_k and returns
    an n^k x n^k matrix as it acts on basis vectors of M
    """
    z=zero_matrix(n^k)
    for j in range(0,n^k):
        l=NN(j).digits(n) #list of base n digits
        s=[0] # digits returns the minimal list of non-zero elements
        #in the expansion, so we need to make it uniform:
        while len((n^k-1).digits(n))-len(l) !=0:
```

```
            l=l+s
        l.reverse() # makes list left-justified, i.e.
        #3 in base 2 is now [1,1,0]
        perm="("+str(j)+","
        m=g(l) # permutes list
        m.reverse() # makes list friendly for conversion to base 10
        y=0
        for i in range(len(l)):
            y+=m[i]*n^i # converts to base 10
        perm+=str(y)+")"
        for i in range(0,n^k): # goes along column, assigns 1
            #if it hits the target basis vector
            if i==y:
                z[i,j]=1
    return z
```

Let's confirm that (1 2) is indeed sent to its matrix in End(M).

```
n=2
k=3
S_k=SymmetricGroup(k)
g=S_k("(1,2)")
print rho(g,n,k)
```

```
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
```

3.1 Finding the irreducible one-Dimensional submodules

We know that because $\mathbb{C}S_k$ is semi-simple, so are any $\mathbb{C}S_k$ modules. The following function will return a list of the one-dimensional simple submodules of M. It works by a simultaneous eigenspace decomposition of every element in the matrix group $\rho(S_k)$. First it evaluates the spectrum of eigenvalues in the group, and then it finds the intersection of eigenspaces for every element with respect to a single eigenvalue. When it is done, it stores that space in a list of common eigenspaces of $\rho(S_k)$. This process repeats until the spectrum is exhausted. This spectrum must be finite, as $\rho(S_k)$ is finite.

```
def one_dimensional_simple_sub_modules(n,k):
    S_k=SymmetricGroup(k)
    R=QQbar
    matrix_list=[]
    for g in S_k.gens():
        matrix_list.append(rho(g,n,k))
    G=MatrixGroup(matrix_list)
    eigenspectrum=[]
    for i in range(0,G.cardinality()):
        l= matrix(R,G[i]).eigenvalues()
        for j in range(0, n^k):
            if l[j] not in eigenspectrum:
                eigenspectrum.append(l[j])
    common_eigenspace_list=[]
    iden=identity_matrix(R,n^k).column_space()
    for i in eigenspectrum:
        common_eigenspace=iden
```

```
        mat_eig_space=iden
        for j in range(0,G.cardinality()):
            mat = matrix(R,G[j])
            mat_eig_space=iden
            for d in range(0,len(mat.eigenspaces_right())):
                if mat.eigenspaces_right()[d][0]==i:
                    mat_eig_space=mat.eigenspaces_right()[d][1]
            common_eigenspace=common_eigenspace.intersection(mat_eig_space)
        if common_eigenspace.dimension() != 0:
            common_eigenspace_list.append([i,common_eigenspace])
    return common_eigenspace_list
```

Here we will look at the case where $n = 2$ and $k = 3$. As we can see there are only 4 copies of the trivial S_3 module. The first entry is the eigenvalue, and the second is information about the eigenspace.

```
for i in one_dimensional_simple_sub_modules(2,3):
    print i
    print '\n'
```

```
[1, Vector space of degree 8 and dimension 4 over Algebraic Field
Basis matrix:
[1 0 0 0 0 0 0 0]
[0 1 1 0 1 0 0 0]
[0 0 0 1 0 1 1 0]
[0 0 0 0 0 0 0 1]]
```

Thus, by duality, the corresponding $\mathfrak{g}$ module is 4 dimensional and appears with multiplicity 1. The eigenspaces are all one-dimensional.

Now if n=k=3, there are both trivial and sign sub-modules:

```
for i in one_dimensional_simple_sub_modules(3,3):
    print i
    print '\n'
```

```
[-1, Vector space of degree 27 and dimension 1 over Algebraic Field
Basis matrix:
[ 0 0 0 0 0 1 0 -1 0 0 0 -1 0 0 0 1 0 0 0 1 0 -1 0 0 0 0 0]]


[1, Vector space of degree 27 and dimension 10 over Algebraic Field
Basis matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]]
```

According to the theorem, corresponding $\mathfrak{g}$ submodules are one-dimensional (dual with signed submodule) and 10 dimensional (dual with trivial module).

Of course, finding the one-dimensional submodules really comes down to calculating the simultaneous eigenspaces for each eigenvalue for the endomporphisms in $\rho(S_k)$, and intersecting them, but we should strive to solve for the higher-dimensional simple sub-modules as well. This only gives us information on the $\mathfrak{g}$ modules that appear with multiplicity one. Unfortunately, this is the best we can hope for in terms of canonical methods without having to rely on combinatorics and Lie methods.

So what can we do? Let us consider the Artin-Wedderburn Theorem as it applies to algebras.

If (V, ρ) is an A representation, then V becomes an A module through ρ, and $\rho(e_\lambda)$ is the projection of V onto V^λ. In this context, these idempotents hold the key to understanding how a space decomposes as an A module.

In the case of group algebras, there is a nice formula for calculating idempotents using the elements of the group as standard basis vectors, but it requires that you already understand what the isomorphism classes of the algebra already look like. But in order to generalize for large families of algebras like $\mathbb{C}S_k$, we need to employ some different tactics. We explore some of these methods later on.

4 Central, primitive, orthogonal idempotents of $\mathbb{C}S_k$ as actions on $(\mathbb{C}^n)^{\otimes k}$

We took linear algebra methods as far as they could go, so this time we will calculate the actions of the primitive-central, orthogonal idempotents of $\mathbb{C}S_k$ as they project onto the subspaces of $V = \mathbb{C}^{\kappa^{\otimes k}}$ isomorphic to the simple $\mathfrak{gl}_n(\mathbb{C}) \bigotimes \mathbb{C}S_k$ bimodules of V.

Luckily for us, SAGE already knows what these idempotents are. Let's begin by working with Symmetric Group Algebra. If we assign the group call *.algebra(F)

```
k=3
CSk=SymmetricGroup(k).algebra(QQbar)

CSk_idem=CSk.central_orthogonal_idempotents()
print CSk_idem
```

```
[1/6*() + 1/6*(2,3) + 1/6*(1,2) + 1/6*(1,2,3) + 1/6*(1,3,2) + 1/6*(1,3),
     2/3*() - 1/3*(1,2,3) - 1/3*(1,3,2), 1/6*() - 1/6*(2,3) - 1/6*(1,2) +
     1/6*(1,2,3) + 1/6*(1,3,2) - 1/6*(1,3)]
```

To multiply two elements together, you can simply write out $a * b$. To add them, write $a + b$. We interpret () as the identity permutation, and can pass a permutation (python list) $[x_1, ..., x_n]$ through to the algebra and return an element in the algebra using $CSk([x_1, ..., x_n])$:

```
print CSk([2,1])
print CSk_idem[0]*CSk([2,1])
```

```
(1,2)
1/6*() + 1/6*(2,3) + 1/6*(1,2) + 1/6*(1,2,3) + 1/6*(1,3,2) + 1/6*(1,3)
```

With so much of the work already done for us, we just need to find a way to interpret an element in the algebra and build a true algebra homomorphism $\rho : \mathbb{C}S_k \to \mathrm{End}(V)$.

```
CSk_basis=[p for p in SymmetricGroup(k)]
print CSk_basis
```

```
[(), (1,3,2), (1,2,3), (2,3), (1,3), (1,2)]
```

After building a basis set, we can obtain the $k!$- vector of coefficients from any idempotent.

```
s=CSk_idem[1]
v=zero_vector(QQbar, factorial(k))
print s
for i in range(0,factorial(k)):
    v[i]=s.coefficient(CSk_basis[i])
print v
```

```
2/3*() - 1/3*(1,2,3) - 1/3*(1,3,2)
(2/3, -1/3, -1/3, 0, 0, 0)
```

Now as long as we can define ρ on each element of S_k, we can express the endomorphism as the sum of matrices with these coefficients. So let's bring back our ρ function from last time:

```
def rho_local(g,n,k):
    z=zero_matrix(n^k)
    for j in range(0,n^k):
        l=NN(j).digits(n)
        s=[0]
        while len((n^k-1).digits(n))-len(l) !=0:
            l=l+s
        l.reverse()
        perm="("+str(j)+","
        m=g(l)
        m.reverse()
        y=0
        for i in range(len(l)):
            y+=m[i]*n^i
        perm+=str(y)+")"
        for i in range(0,n^k):
            if i==y:
                z[i,j]=1
    return z

def vectorize_element(s,k):
    v=zero_vector(QQbar, factorial(k))
    for i in range(0,factorial(k)):
        v[i]=s.coefficient(CSk_basis[i])
    return v
```

Now, we can define a morphism on the symmetric group algebra:

```
def rho(s,n,k):
    vec=vectorize_element(s,k)
    mat=zero_matrix(QQbar,n^k)
    for i in range(0,factorial(k)):
        m=rho_local(CSk_basis[i],n,k)
        mat+=vec[i]*m
    return mat
```

Now we will use everything we made to see what our idempotent endomorphisms look like:

```
k=2
n=3
CSk=SymmetricGroup(k).algebra(QQbar)
CSk_idem=CSk.central_orthogonal_idempotents()
CSk_basis=[p for p in SymmetricGroup(k)]

for s in CSk_idem:
    M= rho(s,n,k)
    print M
    print "The rank of this matrix is: " + str(M.rank())
```

```
[ 1 0 0 0 0 0 0 0 0]
[ 0 1/2 0 1/2 0 0 0 0 0]
[ 0 0 1/2 0 0 0 1/2 0 0]
[ 0 1/2 0 1/2 0 0 0 0 0]
[ 0 0 0 0 1 0 0 0 0]
[ 0 0 0 0 0 1/2 0 1/2 0]
[ 0 0 1/2 0 0 0 1/2 0 0]
[ 0 0 0 0 0 1/2 0 1/2 0]
[ 0 0 0 0 0 0 0 0 1]
The rank of this matrix is: 6
[ 0 0 0 0 0 0 0 0 0]
```

```
[ 0 1/2 0 -1/2 0 0 0 0 0]
[ 0 0 1/2 0 0 0 -1/2 0 0]
[ 0 -1/2 0 1/2 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 1/2 0 -1/2 0]
[ 0 0 -1/2 0 0 0 1/2 0 0]
[ 0 0 0 0 0 -1/2 0 1/2 0]
[ 0 0 0 0 0 0 0 0 0]
The rank of this matrix is: 3
```

Just as we had hoped, the rank sums up to the dimension of V. We will now try various values of n, k and see if the column spaces of these matrices intersect trivially, thereby telling us that V is the direct sum of projected spaces, and check if each projected space is invariant under the generators of S_k, telling us that V has a subspace that decomposes as an $\mathbb{C}S_k$ module. Lastly, we see if the ranks of the matrices sum up to the dimension of V, telling us that V in fact admits this decomposition.

4.1 Sanity check

Just to make sure everything works for small values of n, k, the following code will cycle through the actions of the algebra to make sure we have our full decomposition of the module:

```
for n in range(2,4):
    for k in range(2,4):
        print "Using n = "+str(n)+" and k = "+str(k)+": "

        Sk=SymmetricGroup(k)
        CSk=SymmetricGroup(k).algebra(QQbar)
        CSk_idem=CSk.central_orthogonal_idempotents()
        CSk_basis=[p for p in SymmetricGroup(k)]
        mat_list=[]
        for s in CSk_idem:
            mat_list.append(rho(s,n,k))
        dim_sum=0
        for M in mat_list:
            dim_sum+=M.rank()
            Mcol=M.column_space()
            for N in mat_list:
                if M!=N:
                    Ncol=N.column_space()
                    IntSpace=Mcol.intersection(Ncol)
                    if IntSpace.dimension()!=0:
                        print "ERROR"
                        break
            for g in Sk.gens():
                gMat=rho_local(g,n,k)
                Mact=(gMat*M).column_space()
                if not Mact.is_subspace(Mcol):
                    print "Error"
                    break
        if dim_sum != n^k:
            print "Error"
            break
        print "All checks out!"
```

```
Using n = 2 and k = 2:
All checks out!
```

```
Using n = 2 and k = 3:
All checks out!
Using n = 3 and k = 2:
All checks out!
Using n = 3 and k = 3:
All checks out!
```

5 Brauer algebra idempotents

The Brauer Algebra, $B_k(z)$, is an important subalgebra of the Partition Algebra, $P_k(z)$. We view $P_k(z)$ as the set of partitions of $\{-k, ..., -1, 1, ..., k\}$. The element $z \in \mathbb{C}$ is the deformation parameter of the algebra, and simply gives us a means to scale the product of two partitions should a "loop" occur. The product of two partitions, a, b is the partition $a * b$ obtained by identifying the positive elements of a with their negative counterparts of b and forming a new partition that way. Call this new partition c. Then $a * b = z^{m_b^a} c$, where m_b^a is the number of loops formed. Two partitions form a loop if $\{-i, -j\}$ is in b and $\{i, j\}$ is in a.

For example, when $k = 2$, let's take $a = \{\{1, 2\}, \{-1, -2\}\}$. Then $a * a = za$, because of the occurence of a loop. Now let $k = 3$, $a = \{\{1, 2\}, \{-1, -2\}, \{3, -3\}\}$, and $b = \{\{1, -1\}, \{-2, -3\}, \{2, 3\}\}$. Then $b * a = \{\{1, 2\}, \{-1, 3\}, \{-2, -3\}\}$. The symmetric group algebra is actually a subalgebra of the partition algebra, but it does not require a deformation parameter because no loops can occur with the partition presentation of S_k. If σ is a permutation, then its diagram in $P_k(z)$ is simply the partition $\{\{1, -\sigma(1)\}, ..., \{k, -\sigma(k)\}\}$. In which case, the product of two permutation diagrams is akin to function composition of the two permutations.

$P_k(z)$ has a well defined action on $V = (\mathbb{C}^n)^{\otimes k}$. Let d be a diagram in P_k, i.e. a basis vector. Then the action of d on a tensor is given as:

$$d \cdot (v_{i_1} \otimes ... \otimes v_{i_k}) = \sum_{1 \le i_{-1}, ..., i_{-k} \le n} \delta(d)_{i_1, ..., i_k}^{i_{-1}, ..., i_{-k}} v_{i_{-1}} \otimes ... \otimes v_{i_{-k}},$$

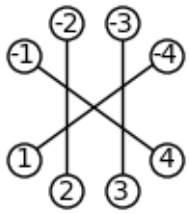
where

$\delta(d)_{i_1, ..., i_k}^{i_{-1}, ..., i_{-k}}$ is 1 if {s,t} with $s, t \in \{-k, ..., -1, 1, ..., k\}$ are in the same block ($\{s, t\}$ is a connected component of d) implies $i_s = i_t$. It is 0 otherwise. We extend that action $\mathbb{C}$-linearly, and define it that way on $P_k(z)$. For a presentation of the Brauer and Partition algebras, we refer the reader to a paper by Arun Ram and Tom Halverson [HaRa]. This paper also explains which Lie groups/Lie algebras are central to the various diagram algebras, and how multiplication of the algebra elements works. But for our purposes, we will just focus on the algebras themselves.

It is often helpful to view partitions using diagrams, and SAGE provides tools for just that.

```
import sage.combinat.diagram_algebras as da

def partition_to_graph(m,k):
    g=da.to_graph(m)
    pos_dict={}
    for j in range(-k,0):
        pos_dict[j]=[-cos(pi*(-j)/(k+1)),sin(pi*(-j)/(k+1))]
    for j in range(1,k+1):
        pos_dict[j]=[-cos(pi*(-j)/(k+1)),sin(pi*(-j)/(k+1))]
    return g.graphplot(pos=pos_dict,vertex_size=150,vertex_color='white',figsize=1.7)

B4=SetPartitionsBk(4)
s=B4[3]
print s
partition_to_graph(s,4).show(figsize=1.7)
```

```
{{-4, 1}, {-3, 3}, {-2, 2}, {-1, 4}}
```



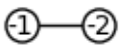
This element is actually in the Symmetric Group, S_4, and we can identify it with the permutation (1 4). Let $n = 2$, $k = 4$. We should want that $e_1 \otimes e_1 \otimes e_1 \otimes e_2$ is sent to $e_2 \otimes e_1 \otimes e_1 \otimes e_1$ under this permutation. Surely enough, $\delta(d)^{i_{-1},i_{-2},i_{-3},i_{-4}}_{1,1,1,2}$ will be zero everywhere except for $(i_{-1}, i_{-2}, i_{-3}, i_{-4}) = (2, 1, 1, 1)$.

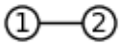
Brauer diagrams can be defined as the partitions of P_k for which every set in the partition has cardinality 2.

Here's a really basic, but important element of $B_2(z)$:

```
B2=SetPartitionsBk(2)
s=B2[2]
print s
partition_to_graph(s,2).show(figsize=1.7)
```

```
{{-2, -1}, {1, 2}}
```





With $n = 2$, this element will do the following to our simple basis tensors:

$$e_1 \otimes e_1 \mapsto e_1 \otimes e_1 + e_2 \otimes e_2$$

$$e_2 \otimes e_1 \mapsto 0$$

$$e_1 \otimes e_2 \mapsto 0$$

$$e_2 \otimes e_2 \mapsto e_1 \otimes e_1 + e_2 \otimes e_2$$

As a matrix representation, this would look like:

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

So how can we generalize this to the fullest extent for any n and any k? We need to define a representation, $\psi : B_k(z) \to \mathrm{End}(V)$. Just like last time, we will be using the set $\{0, ..., n-1\}$ to index our basis vectors e_i in order to create a convention on the n-ary expansion of the vector corresponding to $(i_1, ..., i_k)$. What we will do differently is define our local function not on the Symmetric Group, but on the Set Partitions themselves. This serves as a basis, but we do not get a simple permutation as a group action on the list $(i_1, ..., i_k)$ like we did with the basis set S_k. So, we need to get a little more creative. The upshot here is that we can adapt this setup for any general diagram algebra, and not just B_k.

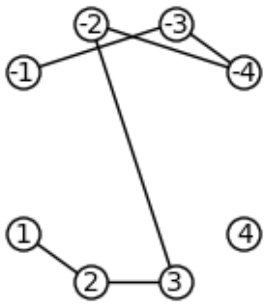
One obstacle to overcome is defining the δ function. How do we tell that two vertices are connected in our graph? One thing we can do is find all of the connected components of the graph itself!

```
def connected_components(m):
    collection=[]
    for X in m:
        collection.append(X)

    subsetsofsizetwo=[]
    for X in collection:
        for Y in Set(X).subsets(2):
            subsetsofsizetwo.append(Y)
    return subsetsofsizetwo
```

Let's see if this works!

```
P4=SetPartitionsPk(4)
s=P4[1]
print s
partition_to_graph(s,4).show(figsize=2.7)

print connected_components(s)
```

```
{{-4, -3, -2, -1, 1, 2, 3}, {4}}
```



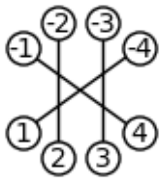

```
[{1, 2}, {1, 3}, {1, -1}, {1, -4}, {1, -3}, {1, -2}, {2, 3}, {2, -1}, {2,
    -4}, {2, -3}, {2, -2}, {3, -1}, {3, -4}, {3, -3}, {3, -2}, {-4, -1},
    {-3, -1}, {-2, -1}, {-4, -3}, {-4, -2}, {-3, -2}]
```

Knowing what two vertices are connected in our graph should make defining δ a breeze.

```
def delta(m,k,pos_index=[],neg_index=[]):
    """
    returns 0 or 1
    pos/neg_index are lists of size k with entries {0,...,n-1}
    """
    #first we check the for the negation of {r,s}-->i_r=i_s
    del_dict={}
    for i in range(1,k+1):
        del_dict[i]=pos_index[i-1]
    for i in range(-k,0):
        del_dict[i]=neg_index[-i-1]
    ConnectedComp=connected_components(m)
    for x in ConnectedComp:
        if del_dict[x[0]] != del_dict[x[1]]:
            return 0
    #since the negation is false, we must have that delta is 1
    return 1
```

Let's look at $\delta^{1,1,1,1}_{1,1,1,2}((1\ 4))$ and $\delta^{2,1,1,1}_{1,1,1,2}((1\ 4))$:

```
B4=SetPartitionsBk(4)
s=B4[3]
partition_to_graph(s,4).show(figsize=1.7)
```




```
print "delta(s,k,[1,1,1,2],[1,1,1,1]) is: "
print delta(s,4,[1,1,1,2],[1,1,1,1])
print "delta(s,k,[1,1,1,2],[2,1,1,1]) is: "
print delta(s,4,[1,1,1,2],[2,1,1,1])
```

```
delta(s,k,[1,1,1,2],[1,1,1,1]) is:
0
delta(s,k,[1,1,1,2],[2,1,1,1]) is:
1
```

Now, let's define ψ on the basis elements of P_k:

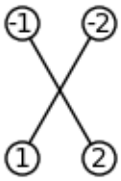
```
def psi_local(m,n,k):
    z=zero_matrix(n^k)
    for j in range(0,n^k): #columns
        l=NN(j).digits(n) #list of base n digits
        l.reverse() #read right to left in expansion
        s=[0]
        while len((n^k-1).digits(n))-len(l) !=0:
            l=s+l
        for i in range(0,n^k): #rows
            h=NN(i).digits(n)
            h.reverse()
            while len((n^k-1).digits(n))-len(h) !=0:
                h=s+h
            z[i,j]=delta(m,k,l,h)
    return z
```

This next chunk of code will let us know if everything is running smoothly:

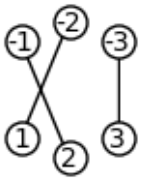
```
n=2
k=2
Sk=SetPartitionsSk(k)
transposition=Sk[1]
print str(transposition)+" is identified with the permutation (1 2)."
print "It can be visualized as: "
partition_to_graph(transposition,k).show(figsize=1.7)
print "Then rho((1 2))=psi((1 2)):"
print psi_local(transposition,n,k)
print "Just to be sure, when n=2,k=3:"
k=3
Sk=SetPartitionsSk(k)
transposition=Sk[2]
print str(transposition)+" goes to (1 2)"
print "It can be visualized as: "
partition_to_graph(transposition,k).show(figsize=1.7)
print "And it's representation is: "
print psi_local(transposition,n,k)
```

```
print ""
print "Now for some random old diagram: "
Sk=SetPartitionsPk(k)
s=Sk[21]
print s
print "It can be visualized as: "
partition_to_graph(s,k).show(figsize=1.7)
print "And it's representation is: "
print psi_local(s,n,k)
```

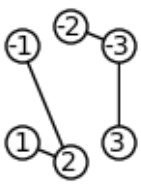
```
{{-2, 1}, {-1, 2}} is identified with the permutation (1 2).
It can be visualized as:
```




```
Then rho((1 2))=psi((1 2)):
[1 0 0 0]
[0 0 1 0]
[0 1 0 0]
[0 0 0 1]
Just to be sure, when n=2,k=3:
{{-3, 3}, {-2, 1}, {-1, 2}} goes to (1 2)
It can be visualized as:
```




```
And it's representation is:
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]

Now for some random old diagram:
{{-3, -2, 3}, {-1, 1, 2}}
It can be visualized as:
```

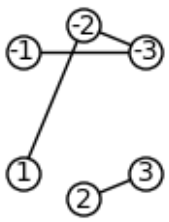


```
And it's representation is:
[1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1]
```

More exotic examples!

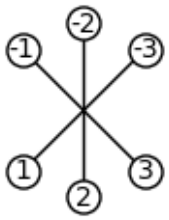
```
n=2;k=3
Ak=SetPartitionsAk(k)
Bk=SetPartitionsBk(k)
Pk=SetPartitionsPk(k)
a=Ak[3]
b=Bk[0]
p=Pk[1]

print str(a) + " from A_k has diagram: "
partition_to_graph(a,k).show(figsize=2.0)
print "and it's representation is: "
print psi_local(a,n,k)
print ""

print str(b) + " from B_k has diagram: "
partition_to_graph(b,k).show(figsize=2.0)
print "and it's representation is: "
print psi_local(b,n,k)
print ""


print str(p) + " from P_k has diagram: "
partition_to_graph(p,k).show(figsize=2.0)
print "and it's representation is: "
print psi_local(p,n,k)
print ""
```

```
{{-3, -2, -1, 1}, {2, 3}} from A_k has diagram:
```

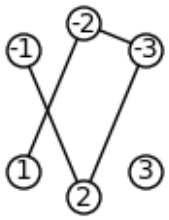



```
and it's representation is:
[1 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
```

```
[0 0 0 0 1 0 0 1]

{{-3, 1}, {-2, 2}, {-1, 3}} from B_k has diagram:
```




```
and it's representation is:
[1 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 1 0]
[0 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 1]

{{-3, -2, -1, 1, 2}, {3}} from P_k has diagram:
```




```
and it's representation is:
[1 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 1]
```

Now, we go on to define ψ on an arbitrary element of the Brauer algebra.

```
def vectorize_element_Bk(s,lst):
    vec=zero_vector(QQbar,len(lst))
    for i in range(0,len(lst)):
        vec[i]=s.coefficient(lst[i])
    return vec

def psi_Bk(v,n,k):
    lst=da.BrauerDiagrams(k).list() #basis
    vec=vectorize_element_Bk(v,lst) #coefficient vector
    mat=zero_matrix(QQbar,n^k)
    for i in range(0,len(lst)):
        if vec[i]!=0:
```

```
            mat+=vec[i]*psi_local(lst[i],n,k)
    return mat
```

Let's see what it gets us for a Jucys Murphy element of the algebra!

```
k=3
n=2
Bk_alg=BrauerAlgebra(k,n,QQbar)
v=Bk_alg.jucys_murphy(3)
print "The element is: "
print v
print "It's action on V is: "
print psi_Bk(v,n,k)
```

```
The element is:
-B{{-3, -2}, {-1, 1}, {2, 3}} - B{{-3, -1}, {-2, 2}, {1, 3}} + B{{-3, 1},
     {-2, 2}, {-1, 3}} + B{{-3, 2}, {-2, 3}, {-1, 1}} + 1/2*B{{-3, 3}, {-2,
     2}, {-1, 1}}
It's action on V is:
[1/2 0 0 -1 0 -1 0 0]
[ 0 1/2 1 0 1 0 0 0]
[ 0 1 1/2 0 0 0 0 -1]
[ -1 0 0 1/2 0 0 1 0]
[ 0 1 0 0 1/2 0 0 -1]
[ -1 0 0 0 0 1/2 1 0]
[ 0 0 0 1 0 1 1/2 0]
[ 0 0 -1 0 -1 0 0 1/2]
```

The Jucys-Murphy elements give us a means to calculate the idempotents of a multiplicity-free family of unital algebras. Therefore, knowing the Jucys-Murphy elements and the combinatorics behind the Induction and Restriction operations on the family gives us a recursive formula for the idempotents of the irreducible representations of $B_k(z)$. Our methods closely follow those of Stephen Doty, Aaron Lauve, and George H. Seelinger, and we refer their paper to the reader to review before proceeding further [DoLaSe].

Let λ be in the indexing set for the irreps of $B_k(z)$, $Tab(k)$. Let μ be in the indexing set for the irreps of $B_{k-1}(z)$, $Tab(k-1)$. The Jucys-Murphy sequence $(J_0, J_1, ..., J_k)$ can be seen as elements in $B_k(z)$ by inclusion.

If $T = (\lambda_0, ..., \lambda_k = \lambda)$ corresponds to the Gelfand-Tsetlin basis element v_T, and $S = (\mu_0, ..., \mu_k)$ corresponds to v_S, we define:

$$c_T(k) = \begin{cases} j - i & \text{if } \lambda_k \text{ has one more box than than } \lambda_{k-1} \\ (1-z) + i - j & \text{if } \lambda_k \text{ has one less box than than } \lambda_{k-1} \end{cases}$$

The Gelfand-Tsetlin basis for $B_k(z)^\lambda$ is the set $\{v_T \ : \ T \in Tab(k)\}$. For $T = (\lambda_0, ..., \lambda_k)$, the truncation is the element $\bar{T} = (\lambda_0, ..., \lambda_{k-1}) \in Tab(k-1)$. We define the interpolating polynomial as:

$$P_T(x) = \prod_{S \in Tab(k)} \prod_{S \neq T} \prod_{\bar{S} = \bar{T}} \frac{x - c_S(k)}{c_T(k) - c_S(k)}$$

We usually use the notation P_μ^λ to denote that this function really only depends on the μ's for which $\mu = \lambda_{n-1}$ (we say then that $\mu \vdash \lambda$). Finally, we obtain a recursive formula for the idempotent of λ as:

$$e_\lambda = \sum_{\mu \mu \vdash \lambda} P_\mu^\lambda(J_k) e_\mu$$

In general, the sum $z_k = J_1 + ... + J_k$ is in the center of $B_k(z)$ and it acts diagonally on $B_k(z)^\lambda$. The T content of J_k is the $\mathbb{C}$ vector $(c_T(1), ..., c_T(k))$, where $J_m \cdot v_T = c_T(m)v_T$. This can be shown pretty straightforwardly, since $J_m = z_m - z_{m-1}$ and $v_T = v_{\bar{T}}$. Therefore, for μ with $\mu = \lambda_{k-1}$, $z_k \cdot v_T = a_\lambda v_T$ and $z_{k-1} \cdot v_T = a_\mu v_T$ and it follow that $c_T(k) = a_\lambda - a_\mu$.

In order to make this work, we need a Jucys-Murphy sequence that agrees with the T content. We will follow the cue of Stephen Doty, Aaron Lauve, AND George H. Seelinger (https://arxiv.org/pdf/1606.08900.pdf) and define our sequence as follows:

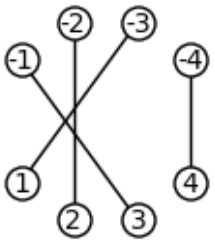
$$J_k = \sum_{i=1}^{k-1} [(i\ k) - \overline{(i\ k)}]$$

with $J_1 = 0$, $(i\ k) = \{\{1, -1\}, ..., \{i, -k\}, \{i+1, -(i+1)\}, ..., \{k, -i\}\}$, the standard transposition, and $\overline{(i\ k)} = \{\{1, -1\}, ..., \{i, k\}, \{-i, -k\}, \{i+1, -(i+1)\}, ..., \{k-1, -(k-1)\}\}$, the cap-cup.

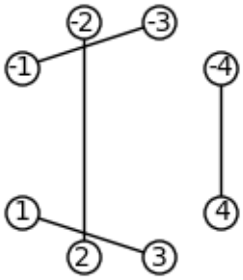
For example:

```
k=4
Sk=SetPartitionsBk(k)
s=Sk[45]
d=Sk[58]
print d
print "The element (1 3) in B_4(z) is: "
print s
partition_to_graph(s,k).show(figsize=2.2)
print "The cap-cup version of (1 3) is:"
print d
partition_to_graph(d,k).show(figsize=2.5)
```

```
{{-4, 4}, {-3, -1}, {-2, 2}, {1, 3}}
The element (1 3) in B_4(z) is:
{{-4, 4}, {-3, 1}, {-2, 2}, {-1, 3}}
```




```
The cap-cup version of (1 3) is:
{{-4, 4}, {-3, -1}, {-2, 2}, {1, 3}}
```



Now let's work out a function that gives us the Jucys-Murphy elements we want. People have already implemented Jucys-Murphy elements for the Brauer algebra in SAGE, but they differ from the ones we want by $\frac{z-1}{2}$.

```
def our_jucys_murphy(n,k,z,R=QQbar):
    Bk=BrauerAlgebra(k,z,R)
    assert(n<=k)
    s=Bk.jucys_murphy(n)
    z=s-((z-1)/2)
    return z
```

The business with the boxes lets us interpret $Tab(k)$ as a set of integer partitions, i.e. $[3,1,1]$ corresponds to:

```
p=Partition([3,1,1])
p.pp()
```

```
***
*
*
```

The pair (i,j) is the (row,column) location of a box (in our case asterisk) in the partition. The content of a box is the value $j-i$. We can print the contents like this:

```
p=Partition([5,4,3,2,1])
p.pp()
p.contents_tableau().pp()
```

```
*****
****
***
**
*
  0 1 2 3 4
 -1 0 1 2
 -2 -1 0
 -3 -2
 -4
```

The contents of boxes added or removed becomes a vital part of how we understand the action of the Jucys-Murphy elements on the GT basis of our algebra. Let us first try and understand what the simple subalgebras of $B_k(z)$ look like. We assume that either $z > k$ when z is an integer, or $z \in \mathbb{C}\backslash\mathbb{Z}$ so that $B_k(z) = B_k$ is semisimple.

It can be shown for example that the set of irreducible representations are indexed by the following partitions:

$$Irr(k) = \{\lambda \ : \ \lambda \vdash (k-2l), \quad 0 \leq 2l \leq k\}$$

.

We understand $\lambda \vdash (k-2l)$ to mean that the partition sums up to $k-2l$, e.g. $[2]$ is in $Irr(4)$ because it sums up to $2 = 4-2$, but $[1,1,1]$ is not because its sum is $3 \neq 4-2l$. Let's create a function that returns a list of such partitions:

```
def list_of_brauer_irreducibles(k):
    if k==0:
        return [Partition([])]
    lst=[]
    possible_sums=[]

    for w in range(0,k+1):
        if w%2==0: #even
```

```
            possible_sums.append(k-w)
    for x in possible_sums:
        for y in Partitions(x):
            lst=[y]+lst
    return lst
```

And to draw them out:

```
def partition_to_figure_list(p):
    lst=[]
    i=0
    for y in p:
        for z in range(0,y):
            lst+=[polygon2d([[z,i],[z+1,i],[z+1,i+1],[z,i+1]],edgecolor='black',fill=False)]
        i=i-1
    return lst
```

```
def plot_partition_figure_list(lst):
    plotlst=[]
    for x in lst:
        plotlst+=[plot(x)]
    show(sum(plotlst),axes=False,figsize=1.2)
```

```
x=Partition([3,2,1])
plot_partition_figure_list(partition_to_figure_list(x))
```

Now all of the k^{th} irreps can be viewed as:

```
for x in list_of_brauer_irreducibles(3):
    plot_partition_figure_list(partition_to_figure_list(x))
```

The empty partition [] will appear as a 0. One could use these objects as vertices, and program a Bratelli Diagram as a graph, which would help in understanding the GT basis for

each irreducible representation. But for our purposes, we will focus namely on computing idempotents. We will view elements of the GT basis as $k+1$ vectors of partitions $T = (\lambda_0, ..., \lambda_k)$. Let us begin by defining $c_T(k)$:

```
def T_content_at_k(T,k,z):
    M=Set(T[k].cells())
    N=Set(T[k-1].cells())
    if T[k].size()>T[k-1].size():
        O=M.difference(N)
        return ZZ(O[0][0])-ZZ(O[0][1])
    else:
        O=N.difference(M)
        return ZZ(O[0][1])-ZZ(O[0][0])+1-z
```

Let's look at both cases:

```
k=3
T={}
T[k]=Partition([3,2,1])
T[k-1]=Partition([3,2])
print T_content_at_k(T,k,0)
```

```
2
```

The content of the box missing! Now if instead we look at $T[k] = [3, 2]$ and $T[k-1] = [3, 2, 1]$, λ_k has one less box than λ_{k-1}, the box at $(i, j) = (2, 0)$. Let $z = 0$, then $c_T(k) = 1-z+i-j = 1 - 0 + 2 - 0 = 3$. Let's see if it worked.

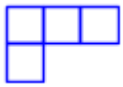
```
T={}
T[k]=Partition([3,2])
T[k-1]=Partition([3,2,1])
print T_content_at_k(T,k,0)
```

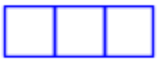
```
-1
```

Now given a partition λ_k, how do we get the list of partitions μ such that μ can have a box added or removed to become λ_k? Well, we already know what partitions are available at the level $k-1$!

```
def permissible_paths_to_partition(P,k):
    assert(P in list_of_brauer_irreducibles(k))
    lst=[]
    box_num=len(P.cells())
    for x in list_of_brauer_irreducibles(k-1):
        if abs(len(x.cells())-box_num)==1:
            lst+=[x]
    new_lst=[]
    for x in lst:
        if box_num-len(x.cells())==1:
            for y in P.cells():
                if y in x.addable_cells():
                    if x.add_cell(y[0])==P:
                        new_lst+=[x]
        elif box_num-len(x.cells())==-1:
            for y in x.cells():
                if y in x.removable_cells():
                    if x.remove_cell(y[0])==P:
                        new_lst+=[x]
    return new_lst

d=Partition([3,1])
plot_partition_figure_list(partition_to_figure_list(d))
```

```
for x in permissible_paths_to_partition(d,4):
    plot_partition_figure_list(partition_to_figure_list(x))
```





In order to get a list for the GT basis, we need to recursively find all possible paths. We employ the use of a global list variable,GT_basis, to keep track of the paths, and a recursive Depth-First Search algorithm to do so. We pass a list and an integer every time in order to add this new partition to the list.

It is important to always initialize GT_basis with GT_basis=[] before running this function. Otherwise, it will just keep adding paths.

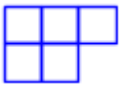
```
def GelfandTsetlinBasis(P,k,GT_basis,path=[]):
    path+=[P]
    if k==0:
        GT_basis.append(list(path))
        #python passes a list by reference,
        #so we use list() to create a new one
    branches=permissible_paths_to_partition(P,k)
    for x in branches:
        GelfandTsetlinBasis(x,k-1,GT_basis,path)
    path.pop(len(path)-1)
```

Just in case any duplicates arise, let's define a function to take care of that! And let's look at an example.

```
def list_of_tableaus_to_partition(P,k):
    GT_list=[]
    GelfandTsetlinBasis(P,k,GT_list,[])
    GT_basis=[]
    for x in GT_list:
        x.reverse() # read paths to start at level k=0
    for x in GT_list: # gets rid of duplicates
        if x not in GT_basis:
            GT_basis.append(x)
    return GT_basis
```

As an example:

```
d=Partition([3,2])
plot_partition_figure_list(partition_to_figure_list(d))
list_of_tableaus_to_partition(d,5)
```



```
[[[], [1], [1, 1], [2, 1], [2, 2], [3, 2]],
 [[], [1], [2], [2, 1], [2, 2], [3, 2]],
 [[], [1], [1, 1], [2, 1], [3, 1], [3, 2]],
 [[], [1], [2], [2, 1], [3, 1], [3, 2]],
 [[], [1], [2], [3], [3, 1], [3, 2]]]
```

Let T be a tableau of $k+1$ partitions, and T_i correspond to its truncation taken $k-i$ times. This way $T_k = T$, and $T_0 = (\lambda_0)$. We use $T[k]$ to denote the partition entry of the tableau at the k^{th} level. Then the idempotent of the Gelfand-Tsetlin vector v_T in the algebra corresponding to the partition $T[k]$ is given as:

$$e_T = \prod_{i=1}^{k} P_{T_i}(J_i)$$

,

where J_i is the i^{th} Jucys-Murphy element. The idempotent then corresponding to the partition $T[k]$ is given as:

$$e_{T[k]} = \sum_{T \in Tab(k)} e_T$$

Let's try and program a way to truncate tableaus:

```
def truncate_tableau(T,k,i):
    #take a tableau at the k^{th} level and truncate it k-i times
    S=[]
    S=list(T)
    if i == k:
        return S
    for p in range(0,k-i):
        S.pop()
    return S
```

Now to define the interpolating polynomial given a tableau T with k+1 entries:

```
def interpolating_polynomial(x,T,k,z):
    assert(T[k] in list_of_brauer_irreducibles(k))
    Bk_alg=BrauerAlgebra(k,z,QQbar)
    iden=Bk_alg.one()
    val=iden
    if k==0:
        return val
    lst=[]
    brauer_irred_lst=list_of_brauer_irreducibles(k)
    for t in brauer_irred_lst:
        tmp_lst=list_of_tableaus_to_partition(t,k)
        for y in tmp_lst:
            lst.append(y)
    lst.remove(T)
    truncatedT=truncate_tableau(T,k,k-1)
    new_lst=[]
    for S in lst:
        truncatedS=truncate_tableau(S,k,k-1)
        if truncatedS==truncatedT and T!=S:
```

```
            new_lst.append(S)
    if len(new_lst)==0:
        return iden
    c_T=T_content_at_k(T,k,z)
    for S in new_lst:
        c_S=T_content_at_k(S,k,z)
        new_val=(x-c_S*iden)*((1/(c_T-c_S))*iden)
        val=val*new_val
    return val

k=4
z=10
Bk_alg=BrauerAlgebra(k,z,QQbar)
elem=Bk_alg.jucys_murphy(3)
iden=Bk_alg.one()
print "The element we want to pass through P_T:"
print elem
lst=list_of_tableaus_to_partition(Partition([2,2]),4)
print "The T we are going to use to do it:"
vec=lst[0]
print vec
print "The final result:"
print interpolating_polynomial(elem,vec,k,z)
print "In vector form:"
print vectorize_element_Bk(interpolating_polynomial(elem,vec,k,z),SetPartitionsBk(k).list())
```

```
The element we want to pass through P_T:
-B{{-4, 4}, {-3, -2}, {-1, 1}, {2, 3}} - B{{-4, 4}, {-3, -1}, {-2, 2}, {1,
     3}} + B{{-4, 4}, {-3, 1}, {-2, 2}, {-1, 3}} + B{{-4, 4}, {-3, 2}, {-2,
     3}, {-1, 1}} + 9/2*B{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
The T we are going to use to do it:
[[], [1], [1, 1], [2, 1], [2, 2]]
The final result:
139/80*B{{-4, 4}, {-3, -2}, {-1, 1}, {2, 3}} - 11/128*B{{-4, 4}, {-3, -2},
     {-1, 2}, {1, 3}} + 419/640*B{{-4, 4}, {-3, -2}, {-1, 3}, {1, 2}} -
     11/128*B{{-4, 4}, {-3, -1}, {-2, 1}, {2, 3}} + 139/80*B{{-4, 4}, {-3,
     -1}, {-2, 2}, {1, 3}} + 419/640*B{{-4, 4}, {-3, -1}, {-2, 3}, {1, 2}} +
      419/640*B{{-4, 4}, {-3, 1}, {-2, -1}, {2, 3}} - 1089/160*B{{-4, 4},
     {-3, 1}, {-2, 2}, {-1, 3}} - 891/640*B{{-4, 4}, {-3, 1}, {-2, 3}, {-1,
     2}} + 419/640*B{{-4, 4}, {-3, 2}, {-2, -1}, {1, 3}} - 891/640*B{{-4,
     4}, {-3, 2}, {-2, 1}, {-1, 3}} - 1089/160*B{{-4, 4}, {-3, 2}, {-2, 3},
     {-1, 1}} + 27/160*B{{-4, 4}, {-3, 3}, {-2, -1}, {1, 2}} - 9/40*B{{-4,
     4}, {-3, 3}, {-2, 1}, {-1, 2}} - 61317/5120*B{{-4, 4}, {-3, 3}, {-2,
     2}, {-1, 1}}
In vector form:
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1089/160,
      -891/640, 419/640, -891/640, -1089/160, 419/640, -9/40, -61317/5120,
     27/160, 139/80, -11/128, 419/640, -11/128, 139/80, 419/640, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Before we can begin multiplying elements in B_{k-i} with those in B_k, we need to find a way to inject one into another. Otherwise, we will turn up an error. The injection is pretty simple. If v is in B_{k-i}, then every diagram of v is passed into a diagram of rank k by adding the pairs $\{(k-i+1),-(k-i+1)\},...,\{k,-k\}$.

```
def inject_kminusi_into_k_local(d,k,i):
    lst=[]
    for x in d:
        lst+=[x]
```

```
    Bk=da.BrauerDiagrams(k)
    for x in range(k-i+1,k+1):
        lst+=[(-x,x)]
    return Bk(Set(lst))
```

Now we define it for arbitrary algebra elements:

```
def inject_kminusi_into_k(v,k,i,z):
    Bk_alg=BrauerAlgebra(k,z,QQbar)
    val=Bk_alg.zero()
    kminusibasis=da.BrauerDiagrams(k-i).list()
    kminusivec=vectorize_element_Bk(v,kminusibasis)
    for y in range(0,len(kminusibasis)):
        if kminusivec[y]!=0:
            val+=kminusivec[y]*Bk_alg(inject_kminusi_into_k_local(kminusibasis[y],k,i))
    return val
```

And test to see if it works:

```
k=4
i=2
z=5
Bk_alg=BrauerAlgebra(k,z)
Bkminusi_alg=BrauerAlgebra(k-i,z)
elem=Bkminusi_alg.jucys_murphy(2)
print elem
kminusibasis=da.BrauerDiagrams(k-i).list()
Bk=da.BrauerDiagrams(k)
inject_kminusi_into_k(elem,k,i,z)
```

```
-B{{-2, -1}, {1, 2}} + B{{-2, 1}, {-1, 2}} + 2*B{{-2, 2}, {-1, 1}}




-B{{-4, 4}, {-3, 3}, {-2, -1}, {1, 2}} + B{{-4, 4}, {-3, 3}, {-2, 1}, {-1,
    2}} + 2*B{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
```

The idempotent of a tableau can be acquired as:

$$e_T = \prod_{i=1}^{k} P_{T[i]}(J_i)$$

```
def idempotent_of_tableau_at_k(T,k,z):
    Bk_alg=BrauerAlgebra(k,z,QQbar)
    J={}
    prod=Bk_alg.one()
    if k==0:
        return prod
    for y in range(1,k+1):
        J[y]=our_jucys_murphy(y,y,z)
    F=[]
    F=list(T)
    for j in range(0,k):
        val=interpolating_polynomial(J[k-j],F,k-j,z)
        F=truncate_tableau(F,k,k-1)
        new_val=inject_kminusi_into_k(val,k,j,z)
        prod=new_val*prod
    return prod
```

The idempotent of a partition $T[k]$ is then the sum:

$$e_{T[k]} = \sum_{S \in \mathrm{Irr}(k)} \sum_{S[k]=T[k]} e_S$$

```
def idempotent_of_partition_at_k(P,k,z):
    Bk_alg=BrauerAlgebra(k,z,QQbar)
    val_sum=Bk_alg.zero()
    basis=da.BrauerDiagrams(k)
    if k==0:
        return Bk_alg.one()
    lst=list_of_tableaus_to_partition(P,k)
    for x in lst:
        idempotent_of_x=idempotent_of_tableau_at_k(x,k,z)
        val_sum+=idempotent_of_x
    return val_sum
```

5.1 Sanity checks

Lastly, we check if these are indeed the primitive, central, mutually orthogonal idempotents we seek! We must have that:

1. $e_T e_S = 0$ for $T \neq S$
2. $e_T^2 = e_T$ for all T
3. $\sum e_T = 1$

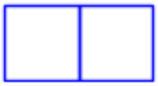
This chunk of code will test for all of that:

```
k=4
z=5
Bk_alg=BrauerAlgebra(k,z,QQbar)
lst=list_of_brauer_irreducibles(k)
basis=da.BrauerDiagrams(k)
the_sum=Bk_alg.zero()
for p in lst:
    x=idempotent_of_partition_at_k(p,k,z)
    the_sum+=x
    print "Testing the partition:"
    plot_partition_figure_list(partition_to_figure_list(p))
    print "Is e_T^2=e_T?"
    if x^2==x:
        print "Yes!"
    else:
        print "No!"
    print "Is it mutually orthogonal with the rest?"
    flag=True
    for q in lst:
        if q!=p:
            y=idempotent_of_partition_at_k(q,k,z)
            if x*y!=0:
                flag=False
    if flag:
        print "Yes!"
    else:
        print "No!"
print "The sum of all these idempotents are:"
print the_sum
```

```
Testing the partition:
```

```
Is e_T^2=e_T?
Yes!
Is it mutually orthogonal with the rest?
Yes!
Testing the partition:
```

```
Is e_T^2=e_T?
Yes!
Is it mutually orthogonal with the rest?
Yes!
Testing the partition:
```

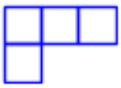


```
Is e_T^2=e_T?
Yes!
Is it mutually orthogonal with the rest?
Yes!
Testing the partition:
```

```
Is e_T^2=e_T?
Yes!
Is it mutually orthogonal with the rest?
Yes!
Testing the partition:
```

```
Is e_T^2=e_T?
Yes!
Is it mutually orthogonal with the rest?
Yes!
Testing the partition:
```

```
Is e_T^2=e_T?
Yes!
Is it mutually orthogonal with the rest?
Yes!
Testing the partition:
```



```
Is e_T^2=e_T?
Yes!
Is it mutually orthogonal with the rest?
Yes!
Testing the partition:
```

```
Is e_T^2=e_T?
Yes!
Is it mutually orthogonal with the rest?
Yes!
The sum of all these idempotents are:
B{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
```

And now we can figure out what representations of these idempotents project onto isotypic components corresponding to these partitions, deduce from the ranks of these matrices and the dimensions of the irreducible submodules of $B_k(z)$ exactly how our spaces decompose. The size of the Gelfand-Tsetlin basis will tell us exactly what this dimesnion is.

It is important to note that $B_k(z)$ is semisimple for positive integers n only when $n \geq k$. We could potentially run into division by zero errors from the interpolating polynomial functions. Let's try an example.

```
k=3
n=3
Bk_alg=BrauerAlgebra(k,n,QQbar)
lst=list_of_brauer_irreducibles(k)
mat_sum=zero_matrix(n^k)
for p in lst:
    x=idempotent_of_partition_at_k(p,k,n)
    print "Testing the partition:"
    plot_partition_figure_list(partition_to_figure_list(p))
    print "The dimension of this submodule is: "
    l=len(list_of_tableaus_to_partition(p,k))
    print l
    mat_x=psi_Bk(x,n,k)
    print "The rank of the representation of the corresponding idempotent is: "
```

```
    r=mat_x.rank()
    print r
    print "Therefore, the irreducible representation of this element decomposes into :"
    print str((r/l))+" copies of the submodule"
    mat_sum+=mat_x
print "Do the representations sum up to the identity?"
if mat_sum.is_one():
    print "Yes!"
else:
    print "No!"
```

```
Testing the partition:
```

```
The dimension of this submodule is:
3
The rank of the representation of the corresponding idempotent is:
9
Therefore, the irreducible representation of this element decomposes into :
3 copies of the submodule
Testing the partition:
```

```
The dimension of this submodule is:
1
The rank of the representation of the corresponding idempotent is:
7
Therefore, the irreducible representation of this element decomposes into :
7 copies of the submodule
Testing the partition:
```

```
The dimension of this submodule is:
2
The rank of the representation of the corresponding idempotent is:
10
Therefore, the irreducible representation of this element decomposes into :
5 copies of the submodule
Testing the partition:
```

```
The dimension of this submodule is:
1
The rank of the representation of the corresponding idempotent is:
1
Therefore, the irreducible representation of this element decomposes into :
1 copies of the submodule
Do the representations sum up to the identity?
Yes!
```

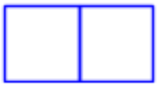
Now, to see what some of these matrix representations of idempotents in End(V) look like, let's try some smaller values for k:

```
k=2
n=3
Bk_alg=BrauerAlgebra(k,n,QQbar)
lst=list_of_brauer_irreducibles(k)
mat_sum=zero_matrix(QQbar,n^k)
for p in lst:
    x=idempotent_of_partition_at_k(p,k,z)
    print "Testing the partition:"
    plot_partition_figure_list(partition_to_figure_list(p))
    print "The dimension of this submodule is: "
    l=len(list_of_tableaus_to_partition(p,k))
    print l
    mat_x=psi_Bk(x,n,k)
    print mat_x
    print "The rank of the representation of the corresponding idempotent is: "
    r=mat_x.rank()
    print r
    print "Therefore, the irreducible representation of this element decomposes into :"
    print str((r/l))+" copies of the submodule"
    mat_sum+=mat_x
print "Do the representations sum up to the identity?"
if mat_sum.is_one():
    print "Yes!"
else:
    print "No!"
print mat_sum
```

```
Testing the partition:
```

```
The dimension of this submodule is:
1
[1/5 0 0 0 1/5 0 0 0 1/5]
[ 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0]
[1/5 0 0 0 1/5 0 0 0 1/5]
[ 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0]
[1/5 0 0 0 1/5 0 0 0 1/5]
The rank of the representation of the corresponding idempotent is:
1
Therefore, the irreducible representation of this element decomposes into :
1 copies of the submodule
Testing the partition:
```

```
The dimension of this submodule is:
1
[ 4/5 0 0 0 -1/5 0 0 0 -1/5]
[ 0 1/2 0 1/2 0 0 0 0 0]
[ 0 0 1/2 0 0 0 1/2 0 0]
[ 0 1/2 0 1/2 0 0 0 0 0]
[-1/5 0 0 0 4/5 0 0 0 -1/5]
[ 0 0 0 0 0 1/2 0 1/2 0]
[ 0 0 1/2 0 0 0 1/2 0 0]
[ 0 0 0 0 0 1/2 0 1/2 0]
[-1/5 0 0 0 -1/5 0 0 0 4/5]
The rank of the representation of the corresponding idempotent is:
6
Therefore, the irreducible representation of this element decomposes into :
6 copies of the submodule
Testing the partition:
```



```
The dimension of this submodule is:
1
[ 0 0 0 0 0 0 0 0 0]
[ 0 1/2 0 -1/2 0 0 0 0 0]
[ 0 0 1/2 0 0 0 -1/2 0 0]
[ 0 -1/2 0 1/2 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 1/2 0 -1/2 0]
[ 0 0 -1/2 0 0 0 1/2 0 0]
[ 0 0 0 0 0 -1/2 0 1/2 0]
[ 0 0 0 0 0 0 0 0 0]
The rank of the representation of the corresponding idempotent is:
3
Therefore, the irreducible representation of this element decomposes into :
3 copies of the submodule
Do the representations sum up to the identity?
Yes!
[1 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 1]
```

5.2 Moving forward

There is one crucial known bug with the following code. Because the polynomial is evaluated term by term, any sort of cancellation that ought to occur to avoid a division by zero error does not. In particular for the case where $n = 3$ and $k = 4$, there is a $z - 4$ term in the denominator of the polynomial when passing the third Jucys-Murphy element through it [DoLaSe, 31]. A potential solution is to return the polynomial, cancel out those terms, and then pass the element through it.

As of writing this document, this implementation of the idempotents of the Brauer algebra is being adapted to be called on as one would call for the idempotents of the symmetric group algebra. The current branch can be found at https://trac.sagemath.org/ticket/28279.

In moving forward, our goals are to optimize our methods and work around the division by zero error. We then aim to adapt this to the other subalgebras of the partition algebra, as this general set up only requires that we know what the isomorphism classes as indexed by partitions at the k^{th} level are and a combinatorial formula for the content vectors.

The Kronecker coefficient problem is an open problem that revolves around the full decomposition of the tensor product of Specht modules (symmetric group algebra modules that are isomorphic to a simple symmetric group subalgebra) and the existence of a combinatorial formula for the multiplicities of the isotypic components in the decomposition. In 2012, a paper by Bowman, De Visscher, and Orellana explored the use of the partition algebra that is the center of the action of the symmetric group on Kronecker products of Specht modules [BoDVOr]. There is an important interplay between the two algebras, and understanding the representation theory of the partition algebra may be key to solving this problem once and for all. We saw previously that we may easily extract information about the Lie group/algebra module decompositions just by knowing the dimension of the isotypic component, and the rank of the associated idempotent's image under the representation. By adapting this method further, and exploring the robust relationship between Schur-Weyl duality and the Artin-Wedderburn thoerems, we come ever closer to solving problems that have remained a mystery for decades.

Acknowledgements

I would like to thank my mentor for this project, Dr. Zajj Daugherty, who has been incredibly patient and kind to me throughout this project. With her guidance and assignments, I was able to fully understand and appreciate the math that went into this field of study. I would also like to thank the individuals of the Math Department at the City College of New York and the proponents of the Dr. Barnett and Jean Hollander Rich Summer Internship for funding my work this summer, and giving me the opportunity to undertake such an important project to myself and the open-source community of mathematicians worldwide.

References

[Ja] Javornik, Pavel, "Scrapbook," *Pavel Javornik's Personal Website*. Web. (https://sites.google.com/site/pjavornikmath/home/scrapbook/).

[SAGE] "SAGE," *SageMath*. Web. (http://www.sagemath.org/).

[GoWall] Goodman, R.; Wallach, N.R., "Representations and Invariants of the Classical Groups," *Encyclopedia of Mathematics and its Applications, vol. 68*, Cambridge University Press, 2000.

[HaRa] Halverson, T.; Ram, Arun, "Partition Algebras," *arXiv Mathematics e-prints*. Web. 2004. (https://arxiv.org/abs/math/0401314)

[Ki] Kirillov, A. "An Introduction to Lie Groups and Lie Algebras," *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 2008.

[DoLaSe] Doty, Stephen; Lauve, Aaron; Seelinger, George H., "Canonical idempotents of multiplicity-free families of algebras," *arXiv Mathematics e-prints*. Web. 2016. (https://arxiv.org/abs/1606.08900)

[BoDVOr] Bowman, Christopher; De Visscher, Maud; Orellana, Rose, "The partition algebra and the Kronecker coefficients," *arXiv Mathematics e-prints*. Web. 2012. (https://arxiv.org/abs/1210.5579)