AAECC
DOI 10.1007/s00200-017-0346-7

ORIGINAL PAPER

Computing weight q-multiplicities for the representations of the simple Lie algebras

Pamela E. Harris[1] · Erik Insko[2] · Anthony Simpson[1]

Received: 19 April 2017 / Revised: 9 October 2017 / Accepted: 18 October 2017

Abstract The multiplicity of a weight μ in an irreducible representation of a simple Lie algebra $\mathfrak{g}$ with highest weight λ can be computed via the use of Kostant's weight multiplicity formula. This formula is an alternating sum over the Weyl group and involves the computation of a partition function. In this paper we consider a q-analog of Kostant's weight multiplicity and present a SageMath program to compute q-multiplicities for the simple Lie algebras.

Keywords Kostant's weight multiplicity formula · q-analog of Kostant's weight multiplicity formula · Representations of simple Lie algebras

1 Introduction

Throughout this paper we let $\mathfrak{g}$ be a simple Lie algebra of rank r and we let $\mathfrak{h}$ be a Cartan subalgebra of $\mathfrak{g}$. We let Φ be the set of roots corresponding to $(\mathfrak{g}, \mathfrak{h})$, and let $\Phi^+ \subseteq \Phi$ be the set of positive roots, while $\Delta \subseteq \Phi^+$ denotes the set of simple roots. We let $P(\mathfrak{g})$ be the set of integral weights, while $P_+(\mathfrak{g})$ denotes the set of dominant

The first and third author were supported by NSF award DMS-1620202.

✉ Pamela E. Harris
peh2@williams.edu

Erik Insko
einsko@fgcu.edu

Anthony Simpson
als7@williams.edu

1 Department of Mathematics and Statistics, Williams College, Williamstown, MA, USA

2 Department of Mathematics, Florida Gulf Coast University, Fort Myers, FL, USA

integral weights. Let W denote the Weyl group, and for any $w \in W$, we let $\ell(w)$ denote the length of w. Recall that W is generated by the reflections $s_1, \ldots, s_r$, where s_i is the root reflection corresponding to the simple root $\alpha_i \in \Delta$. For good general references see [11,23].

We let $L(\lambda)$ denote the irreducible (complex) representation of $\mathfrak{g}$ with highest weight λ and we let μ be a weight of this representation. To compute the q-multiplicity of the weight μ in $L(\lambda)$ we use the q-analog of Kostant's weight multiplicity, defined by Lusztig [24], as

$$m_q(\lambda, \mu) = \sum_{\sigma \in W} (-1)^{\ell(\sigma)} \wp_q(\sigma(\lambda + \rho) - \rho - \mu). \tag{1}$$

For any weight $\xi \in \mathfrak{h}^*$, we have that $\wp_q(\xi) = c_0 + c_1 q + c_2 q^2 + c_3 q^3 + \cdots + c_k q^k$, where c_i is the number of ways to write ξ as a nonnegative integral linear combination of exactly i positive roots. The q-analog of Kostant's weight multiplicity formula when evaluated at $q = 1$ recovers the weight multiplicity formula [22], which is given by

$$m(\lambda, \mu) = \sum_{\sigma \in W} (-1)^{\ell(\sigma)} \wp(\sigma(\lambda + \rho) - (\mu + \rho)), \tag{2}$$

where $\wp$ denotes Kostant's partition function and $\rho = \frac{1}{2} \sum_{\alpha \in \Phi^+} \alpha$. Kostant's partition function is the nonnegative integer-valued function, $\wp$, defined on $\mathfrak{h}^*$, by $\wp(\xi) =$ number of ways ξ may be written as a nonnegative integral linear combination of positive roots, for $\xi \in \mathfrak{h}^*$. Hence it is clear that $\wp_q|_{q=1} = \wp$ and thus $m_q(\lambda, \mu)|_{q=1} = m(\lambda, \mu)$.

Although a formula to compute q-multiplicities exists, its implementation is often difficult. The main obstructions include the fact that for a Lie algebra of rank r the order of the Weyl group, indexing the number of terms in Eq. (1), is factorial in r. Additionally the partition function involved has no known closed formulas, albeit for some very special cases [16,17]. Even with these complications there has been much work done in computing weight multiplicities [1,2,7,9,10,13–16,26] and algorithms associated to this computation [3–6,8]. However, less is known about the q-analog computation, with the following work in that direction [12].

This paper presents a SageMath program to compute the q-multiplicities of weights in highest weight representations of the simple Lie algebras. The source code for the program is found in [18] and the program can be downloaded from GitHub [19]. Our program computes this multiplicity by exploiting the observation that, in practice, most terms in (1) and (2) are zero. Hence, we reduce the weight multiplicity computation by determining the Weyl group elements that contribute nontrivially to these equations. With this in mind, we give the following definition.

Definition 1 For λ, μ dominant integral weights of $\mathfrak{g}$ define the *Weyl alternation set* to be

$$\mathcal{A}(\lambda, \mu) = \{\sigma \in W \mid \wp(\sigma(\lambda + \rho) - (\mu + \rho)) > 0\}.$$

Table 1 The exceptional Lie algebras' exponents, Weyl group order, and cardinality of $\mathcal{A}(\tilde{\alpha}, 0)$

Lie algebra	Exponents	$\lvert W\rvert$	$\lvert\mathcal{A}(\tilde{\alpha}, 0)\rvert$
G_2	1, 5	12	2
F_4	1, 5, 7, 11	1152	25
E_6	1, 4, 5, 7, 8, 11	25,920	58
E_7	1, 5, 7, 9, 11, 13, 17	2,903,040	258
E_8	1, 7, 11, 13, 17, 19, 23, 29	696,729,600	2318

There has been recent work in computing the Weyl alternation sets for certain special weights. For example, Harris showed that the number of Weyl group elements contributing nontrivially to the multiplicity of the zero weight in the adjoint representation (the representation with highest weight equal to the highest root) of $\mathfrak{sl}_{r+1}(\mathbb{C})$ was given by Fibonacci numbers [16]. Later, Harris, Insko, and Williams, generalized these results to show that the number of Weyl group elements contributing nontrivially to the multiplicity of the zero weight in the adjoint representation of all classical Lie algebras is governed by linear homogeneous recurrence relations with constants coefficients [20].

As an application of our program we computed the Weyl alternation sets corresponding to the Weyl group elements contributing nontrivially to the multiplicity of the zero weight in the adjoint representation of the exceptional Lie algebras. The computations and elements of $\mathcal{A}(\tilde{\alpha}, 0)$ for all exceptional Lie algebras can be found in [18, Appendix B]. We then used these Weyl alternation sets to verify the following result of Lusztig for the exceptional Lie algebras [24]: if $\mathfrak{g}$ is a simple Lie algebra with highest root $\tilde{\alpha}$, then $m_q(\tilde{\alpha}, 0) = \sum_{i=1}^{r} q^{e_i}$, where $e_1, \ldots, e_r$ are the exponents of $\mathfrak{g}$. We recall that the exponents of $\mathfrak{g}$ are related to the degrees of the basic invariants. That is, the degrees are obtained by simply increasing the exponents by one [21]. For each exceptional Lie algebra $\mathfrak{g}$, Table 1, gives the order of the Weyl group, the cardinality of the Weyl alternation set $\mathcal{A}(\tilde{\alpha}, 0)$ where $\tilde{\alpha}$ is the highest root of $\mathfrak{g}$, and the exponents of $\mathfrak{g}$. From Table 1, it is evident that computing the Weyl alternation set reduces the computation involved in $m_q(\tilde{\alpha}, 0)$ drastically.

Another component of our program is the ability to calculate the values of $\wp(\xi)$ and $\wp_q(\xi)$ for individual weights ξ in two ways. The program can implement the partition via a recursive algorithm or via a geometric series expansion, a generalization of Euler's generating function for integer partitions. This allows the program to be a tool for those interested in partition theory, as the program can be be used for computing the value of the partition function without having to compute weight multiplicities.

2 Computer implementation

The following sections provide our computer implementation of (1) and the structures used, including both versions for the computation of Kostant's partition function and its q-analog.

2.1 Structures

Using SageMath, we automated Kostant's weight multiplicity formula and the q-analog of Kostant's weight multiplicity formula. SageMath provides robust tools for instantiating Lie algebras, in particular, the exceptional Lie algebras, which are the object of our study. This allowed us to compute the result of applying $\sigma \in W$ to any weight in an exceptional Lie algebra. In this process, we wrote supporting code for computing the value of the partition function, and its q-analog. To find values of these functions, we created two classes in Python. The first, called the *Weight class*, provides a computer representation of a weight that affords us the ability, through basic operations and condition testing, to create a structure that finds the partition of a weight as a nonnegative integral sum of positive roots. The second class (structure), called a *Partition Tree*, computes (1) and (2) by instantiating every possible nonnegative integral combination of positive roots equal to the weight passed as an argument to the formulas. When the tree is instantiated, its input is the weight we are trying to partition, and a list of the positive roots that can be used in the partitioning.

To begin the process, the root of the tree is created, then its children are instantiated. This is done by using the first item in the list of positive roots being used to partition the weight stored at the root of the tree. Note that root has two uses, the root of a tree, or a positive root. To avoid confusion, we will always refer to the root of the tree in full. If we let ξ represent the weight we are trying to partition, and r_1 denote the root that we are currently using to do so, then the root has a child for every element of $\{\xi - nr_1 \mid n \in \mathbb{Z}, n \geq 0\}$ that contains no simple roots with negative coefficients. Each child stores a unique new weight $\xi - nr_1$, and n, the number of times r_1 was used in the partitioning. This represents the subtraction of the first positive root from the original weight as many times as possible. From here, the process begins at each of the root's children, but using r_2, the second root in the list of positive roots. The same process starts again at each of the children of the children of the root, but with r_3, the third root in the list of positive roots. In general, a node in the tree at a depth of i from the root, will have its children created through the same process using r_{i+1}, the $(i+1)$th root in the list of positive roots.

The process is complete when either the weight is reduced to the zero weight, or any one of the coefficients in the linear combination of simple roots, equal to the weight stored in the children of the tree, is negative. When it is reduced to zero, we have found a successful partitioning, and so the terminal node, a constant that is instantiated at the beginning of the program, is made to be a child of the node where the weight was reduced to zero. Not all branches of the tree will end in a terminal node, so another function goes through all tree branches and removes those that do not end in the terminal node, since they do not represent successful partitions, i.e. those weights where a coefficient becomes negative.

The following image represents the partition tree of the weight in (8). Note that we provide the list of positive roots to the right of the tree, and underline the one being used to partition at each level of the tree. Note that when we talk about the relationship between nodes in the tree, we say that if one node has a connection to another node, then those nodes have an edge between them, where the first node is the source node and the second is the destination node, with respect to that edge. In this specific tree

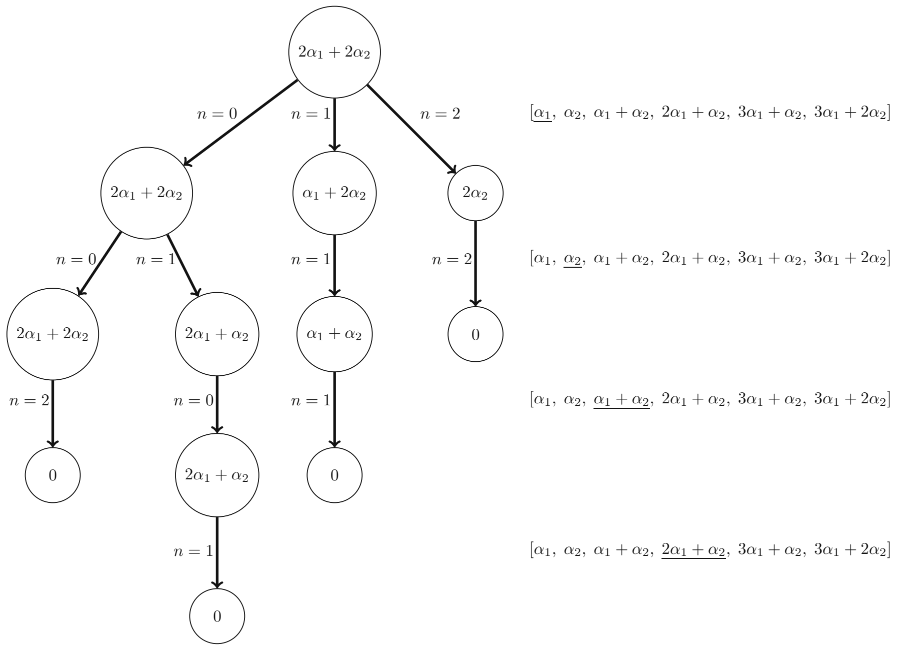
$2\alpha_1 + 2\alpha_2$
$n = 0$
$n = 1$
$n = 2$
$[\underline{\alpha_1}, \alpha_2, \alpha_1+\alpha_2, 2\alpha_1+\alpha_2, 3\alpha_1+\alpha_2, 3\alpha_1+2\alpha_2]$
$2\alpha_1 + 2\alpha_2$
$\alpha_1 + 2\alpha_2$
$2\alpha_2$
$[\alpha_1, \underline{\alpha_2}, \alpha_1+\alpha_2, 2\alpha_1+\alpha_2, 3\alpha_1+\alpha_2, 3\alpha_1+2\alpha_2]$
$2\alpha_1 + 2\alpha_2$
$2\alpha_1 + \alpha_2$
$\alpha_1 + \alpha_2$
0
$[\alpha_1, \alpha_2, \underline{\alpha_1+\alpha_2}, 2\alpha_1+\alpha_2, 3\alpha_1+\alpha_2, 3\alpha_1+2\alpha_2]$
$[\alpha_1, \alpha_2, \alpha_1+\alpha_2, \underline{2\alpha_1+\alpha_2}, 3\alpha_1+\alpha_2, 3\alpha_1+2\alpha_2]$

Fig. 1 Partition tree for $2\alpha_1 + 2\alpha_2$ using the positive roots of the Lie algebra G_2

structure, we have that the weight stored in the destination node is equal to the weight stored in the source node minus n times the highlighted positive root. Note that in the description of the code, the source node holds on to the value of n, but for the sake of the graphic, it is easier to have them associated to the edges between nodes.

Recall that when a branch represents a successful partition, the branch ends in the terminal node. For the purpose of simplifying the image in Fig. 1, on the next page only the successful branches are provided, i.e. those branches ending in zero nodes. Using Fig. 1, on the following page we can arrive at the same partitions of $2\alpha_1 + 2\alpha_2$ as we did in Sect. 2.2.1. Note that, if we start from the root of the tree, and follow any path to a node containing zero, we will find a partition of $2\alpha_1 + 2\alpha_2$ using the positive roots of G_2. The collection of all paths from the node with $2\alpha_1 + 2\alpha_2$ to a node containing zero yields all partitions of this weight. For example, the path from the root to the leftmost zero, indicates that α_1 and α_2 were not used, and $\alpha_1 + \alpha_2$ was used twice. Thereby accounting for $2(\alpha_1 + \alpha_2)$ as a partition of $2\alpha_1 + 2\alpha_2$.

2.2 Computations using the structures

Once the partition tree has been created, we can compute $\wp(\xi)$ and $\wp_q(\xi)$. That is, $\wp(\xi)$ can be determined by counting the number of branches of the partitioning tree that end with the terminal node. Because we provided a function that removed the branches that do not represent a successful partition, every branch ends with the terminal node, so computing $\wp(\xi)$ only requires us to count the number of branches in the tree, where a branch is just any path from the root of the tree to any leaf in the tree. Note that there is only one leaf in the tree, namely the terminal node, so we need only check for that to know the end of the tree.

The computation for $\wp_q(\xi)$ requires that we keep track of the number of roots used in the partition. However, this was already done by storing n, from the expression $\xi - nr_i$. Because the number of roots used at each step of the partitioning is stored for every partition, simply summing the values of n stored in every node of a branch gives the total number of roots used in any given partition. Moreover, counting the roots used in a partition is equivalent to determining the length of the branch when one considers the n value of a child to be the distance between that child node and its parent node. Once the number of roots used is known for a single partition of the weight, the corresponding coefficient in our running calculation of $\wp_q(\xi)$ must be incremented. This means that every c_i in $\wp_q(\xi) = c_0 + c_1q^1 + \cdots c_kq^k$ equals the total number of branches of length i in the partitioning tree. From all branches in the tree, we can obtain a value for $\wp_q(\xi)$.

With the ability to obtain $\wp(\sigma(\lambda+\rho)-(\rho+\mu))$ and $\wp_q(\sigma(\lambda+\rho)-(\rho+\mu))$ for any element $\sigma \in W$, finding the values of $m(\lambda,\mu)$ and $m_q(\lambda,\mu)$ just requires us to iterate over all the elements of the Weyl group. Then, using the length of each element to determine the sign of each term's contribution to the sum in Kostant's weight multiplicity formula, yields the desired result.

2.2.1 The Lie algebra of type G_2

We consider the exceptional Lie algebra G_2 to illustrate our computations. The set of positive roots of the exceptional Lie algebra G_2 is

$$\Phi^+ = \{\alpha_1, \alpha_2, \alpha_1+\alpha_2, 2\alpha_1+\alpha_2, 3\alpha_1+\alpha_2, 3\alpha_1+2\alpha_2\} \tag{3}$$

where $3\alpha_1 + 2\alpha_1$ is the highest root, and $\rho = \frac{1}{2}\sum_{\alpha\in\Phi^+}\alpha = 5\alpha_1 + 3\alpha_2$. The Weyl group of G_2 has 12 elements and is isomorphic to the dihedral group D_6. As every element in W is generated by s_1 and s_2, the simple reflections associated with the simple roots α_1 and α_2, we can understand how any reflection acts on any weight by composing the the simple reflections and applying them to each of the simple roots. In this case we observe that

$$s_1 : \alpha_1 \mapsto -\alpha_1 \tag{4}$$
$$s_1 : \alpha_2 \mapsto 3\alpha_1 + \alpha_2 \tag{5}$$
$$s_2 : \alpha_1 \mapsto \alpha_1 + \alpha_2 \tag{6}$$
$$s_2 : \alpha_2 \mapsto -\alpha_2. \tag{7}$$

To compute $s_1(\widetilde{\alpha}+\rho)-\rho$, we note that $\widetilde{\alpha}+\rho = 8\alpha_1 + 5\alpha_2$ and by (4) and (5), it follows that

$$\begin{aligned} s_1(\widetilde{\alpha}+\rho)-\rho = s_1(8\alpha_1+5\alpha_2)-(5\alpha_1+3\alpha_2) = 8s_1(\alpha_1) \\ + 5s_1(\alpha_2)-(5\alpha_1+3\alpha_2) = 2\alpha_1+2\alpha_2. \end{aligned} \tag{8}$$

Table 2 Weyl alternation sets and q-analog values for G_2

$\sigma \in W$	$\ell(\sigma)$	$\sigma(\widetilde{\alpha}+\rho)-\rho$	$\wp_q(\xi)$
1	0	$3\alpha_1+2\alpha_2$	$q^1+2q^2+2q^3+q^4+q^5$
s_1	1	$2\alpha_1+2\alpha_2$	$2q^2+q^3+q^4$
s_2	1	$3\alpha_1$	q^3
$m_q(\widetilde{\alpha},0)=q+q^5$			

We can now apply Kostant's partition function to $2\alpha_1+2\alpha_2$, and by using the positive roots of the Lie algebra of type G_2 as given in (3), we see that one can write $2\alpha_1+2\alpha_2$ as a nonnegative integral sum of positive roots in the following four ways $2(\alpha_1)+2(\alpha_2)$, $2(\alpha_1+\alpha_2)$, $1(\alpha_1)+1(\alpha_2)+1(\alpha_1+\alpha_2)$, and $1(\alpha_1)+1(2\alpha_1+\alpha_2)$, which use 4, 2, 3, and 2 positive roots, respectively. Thus $\wp_q(2\alpha_1+2\alpha_2)=2q^2+q^3+q^4$. To determine the value that s_1 contributes to (1), we note that $(-1)^{\ell(s_1)}=(-1)^1=-1$, hence

$$(-1)^{\ell(s_1)}\wp(s_1(\widetilde{\alpha}+\rho)-\rho)=-2q^2-q^3-q^4.$$

Repeating this procedure for every remaining element of W, we arrive at the Table 2, which only lists the three elements of W contributing nontrivially to $m_q(\widetilde{\alpha},0)$. From this one can verify that $m_q(\widetilde{\alpha},0)=q+q^5$ as expected.

2.3 Alternate approach to computing Kostant's partition function

It turns out that, while the exhaustive method detailed above works for G_2, F_4, E_6, and E_7, the approach required too much memory for E_8. Even with a machine that had 32 GB of memory, partitioning a single weight took too much memory. This is an astounding fact considering that the computation for E_7 completed without a problem. However in the E_8 case, the partitions do not complete as we are using 120 positive roots. If we consider the number of nodes that would appear in the partition tree of a weight just when we consider either including, or not including each positive root, then we have $2^{120}\approx 1.33\times 10^{36}$ nodes in the tree already. Hence, we implement a well-known method to determine the value of the partition function, which is typically used for partitioning integers, and we adapt it for our purposes.

The new partition function method focuses on expanding a geometric series. We adapted Euler's formula for finding the number of partitions of an integer, denoted $p(n)$. We begin by showing how to derive this formula first by following the work presented in [25]. Begin by examining

$$\sum_{n\geq 0} p(n)x^n, \tag{9}$$

which is a formal power series, where the coefficient of x^n counts the number of partitions of n. We will derive a formula for the generating function given in (9).

Define S to be the set of partitions of an arbitrary integer. We denote the partition of an integer as $(1^{m_1}, 2^{m_2}, \ldots, n^{m_n})$, where m_i is the number of times i is used in the partition. From this, we can see that for every partition, κ,

$$\begin{aligned}\kappa = (1^0 \in \kappa \ or\ 1^1 \in \kappa \ or\ 1^2 \in \kappa \ or\ \cdots)\ and \\ (2^0 \in \kappa \ or\ 2^1 \in \kappa \ or\ 2^2 \in \kappa \ or\ \cdots)\ and \\ (3^0 \in \kappa \ or\ 3^1 \in \kappa \ or\ 3^2 \in \kappa \ or\ \cdots)\ and\ \cdots.\end{aligned} \tag{10}$$

From the definition of an element in S, the generating function for the elements of S will be

$$(x^0 + x^1 + x^{1+1} + \cdots)(x^0 + x^2 + x^{2+2} + \cdots)(x^0 + x^3 + x^{3+3}) \cdots . \tag{11}$$

Notice that the terms in the product of (11) are geometric sums and $\sum_{n=0}^{\infty} r^n = \frac{1}{1-r}$. Hence, we can rewrite (9) as

$$\sum_{n \geq 0} p(n)x^n = \frac{1}{1-x} \cdot \frac{1}{1-x^2} \cdot \frac{1}{1-x^3} \cdots , \tag{12}$$

which can be rewritten as

$$\sum_{n \geq 0} p(n)x^n = \lim_{n\to\infty} \prod_{i=1}^{n} \frac{1}{1-x^i}. \tag{13}$$

Though (13) involves an infinite product, when we want to find the number of partitions for a specific integer, we can change the limit in (13) to be set to that integer. This is because no integer greater than the one that we are partitioning could be used. For example, if we look at partitions of 2, we see that its partitions still have the general form given in (10), but since no number greater than three could be used, we know that the every partition of 2 will be of the form $(1^{m_1}, 2^{m_2}, 1, \ldots, 1)$. This means that every term past 2 in (13), on either side of the equals sign, can be ignored.

We now adapt this approach to count the number of partitions for a weight using a certain set of positive roots. To be able to do this, we must first provide a translation from linear combinations of simple roots to products of the variables used in the series. We take a linear combination of simple roots, $a_1\alpha_1 + \cdots + a_n\alpha_n$, and change it to $A_1^{a_1} \cdots A_n^{a_n}$. We illustrate the technique with an example. If we take the positive root $3\alpha_1 + 2\alpha_2$ in G_2, then the term in our geometric sum will be $A_1^3 A_2^2$. If we denote the set of these translations of positive roots to these terms as

$$\Phi^+_{var} = \{A_1^{a_1} A_2^{a_2} \cdots A_r^{a_r} : a_1\alpha_1 + a_2\alpha_2 + \cdots + a_r\alpha_r \in \Phi^+\},$$

then the series that we are interested in expanding is

$$\prod_{x \in \Phi^+_{var}} \frac{1}{1-qx}. \tag{14}$$

We introduce the q into our adaptation of (13), to count the number of roots used in each partition. Note that in this product, we have geometric sums and we can expand (14) as

$$\prod_{x\in\Phi^+_{var}} \sum_{n=0}^{\infty}(qx)^n. \tag{15}$$

From here, we return to the specific example of G_2. In G_2, we know the positive roots are given by $\Phi^+ = \{\alpha_1, \alpha_2, \alpha_1+\alpha_2, 2\alpha_1+\alpha_2, 3\alpha_1+\alpha_2, 3\alpha_1+2\alpha_2\}$, so $\Phi^+_{var} = \{A_1, A_2, A_1A_2, A_1^2A_2, A_1^3A_2, A_1^3A_2^2\}$. We can use this to rewrite (15) for this specific case, which becomes

$$\sum_{n=0}^{\infty}(qA_1)^n\cdot\sum_{n=0}^{\infty}(qA_2)^n\cdot\sum_{n=0}^{\infty}(qA_1A_2)^n\cdot\sum_{n=0}^{\infty}(qA_1^2A_2)^n\cdot\sum_{n=0}^{\infty}(qA_1^3A_2)^n\cdot\sum_{n=0}^{\infty}(qA_1^3A_2^2)^n. \tag{16}$$

By expanding (16), the value of $\wp_q(\xi)$ is the coefficient of the translation of ξ, just in the same way that the coefficient of x^n was the number of partitions of n in (9). You can think of the translation as the bijection $trans(a_1\alpha_1+\cdots+a_r\alpha_r) = A_1^{a_1}\cdots A_r^{a_r}$ for any weight $a_1\alpha_1+\cdots+a_r\alpha_r$. Thus, we can limit the upper bound on the individual sums in (16) to the maximum number of times the associated positive root could have been used. We will again examine the weight $2\alpha_1+2\alpha_2$ from (8). Our translation of the weight $2\alpha_1+2\alpha_2$ is $A_1^2A_2^2$. We know that α_1, α_2, and $\alpha_1+\alpha_2$ can only be used a maximum of 2 times, $2\alpha_1+\alpha_2$ can only be used once, and the other positive roots of G_2 cannot be used when partitioning $2\alpha_1+2\alpha_2$. If we are examining the weight $2\alpha_1+2\alpha_2$ as we did at the end of Sect. 2.2.1, then we need only expand the polynomial

$$(1+qA_1+q^2A_1^2)(1+qA_2+q^2A_2^2)(1+qA_1A_2+q^2A_1^2A_2^2)(1+qA_1^2A_2)(1)(1). \tag{17}$$

Notice that we do not have to expand (17) fully. Since we know exactly for which term we are searching, we need only multiply everything that will give us a coefficient of $A_1^2A_2^2$. Thus we can choose the terms

$$(q^2A_1^2)(q^2A_2^2) = q^4A_1^2A_2^2 \tag{18}$$

$$(qA_1)(qA_2)(qA_1A_2) = q^3A_1^2A_2^2 \tag{19}$$

$$(qA_2)(qA_1^2A_2) = q^2A_1^2A_2^2 \tag{20}$$

$$(q^2A_1^2A_2^2) = q^2A_1^2A_2^2. \tag{21}$$

The sum of these relevant terms yields $(2q^2+q^3+q^4)A_1^2A_2^2$, whose coefficient is exactly the value of $\wp_q(2\alpha_1+2\alpha_2)$ that we found at the end of Sect. 2.2.1. Proceeding in this fashion with every element of the Weyl group that contributes nontrivially to Kostant's weight multiplicity formula we obtain $m_q(\widetilde{\alpha}, 0)$.

2.4 Generating the Weyl alternation set

Even with the faster method for computing the partition function, there were still too many elements of the Weyl group of E_8 for the computation to finish in a feasible amount of time. Because of this, we implemented a way to compute the alternation set $\mathcal{A}(\lambda, \mu)$ before proceeding to find the multiplicity of μ with respect to λ. This allowed us to work with a much smaller subset of the Weyl group, and thus reduce the computational time associated to the partitioning algorithms.

To compute the alternation set, we first check to see if the expression $1(\lambda + \rho) - (\rho + \mu)$ has any negative coefficients when expressed as a sum of simple roots. If so, then it cannot be partitioned using positive roots and the alternation set is the empty set. If the coefficients are all nonnegative integers, then we know that at least the identity element is in the alternation set, so we append the identity element to the alternation set. We then build the alternation set by taking the elements already in the set, concatenating by simple reflections, and computing $\sigma(\lambda+\rho)-(\rho+\mu)$, where σ is the result of concatenating a simple reflection to an existing element of the alternation set. If the result of this computation results in a nonnegative integral combination of simple roots, then we include the element in the alternation set and iterate the process. Once we cannot append any simple reflections to any of the elements of the alternation set, we know that we have the full alternation set, as we have run through every element in the Weyl group.

This process is implemented with Python's list data structure, rather than sets, so to disambiguate the process further, we will explain in this context. We start the alternation set as a list with just the identity, then we begin iterating through the list. We will denote our index with the variable i, which will initially be set to 0. We start by taking the element in the alternation set at position i. We iterate over each simple reflection, append it to the reflection at position i to make a new reflection and compute the same expression mentioned before. If the expression results in a nonnegative integral linear combination of simple roots, then the new reflection is appended to the end of the list. Notice that this element will also have simple reflections appended to it as i increases and eventually reaches this element. This process is guaranteed to work because we know that once the expression $\sigma(\lambda + \rho) - (\mu + \rho)$ yields a negative coefficient when expressed as a sum of simple roots, appending more reflections to σ which make the length of σ increase will only decrease the coefficients of the simple roots further, as was proven by the first and second author in [20, Proposition 3.4]. We present this result below, but omit its proof as it is both technical and lengthy.

Proposition 2.1 *Let s_i denote the simple root reflection corresponding to $\alpha_i \in \Delta$. Write the weights $\sigma(\rho) - \rho$, $\sigma s_i(\rho) - \rho$, and $s_i\sigma(\rho) - \rho$ as linear combinations of positive roots as follows:*

- $\sigma(\rho) - \rho = -\sum_{\alpha_j\in\Delta} c_j\alpha_j$ *with* $c_j \geq 0$,
- $\sigma s_i(\rho) - \rho = -\sum_{\alpha_j\in\Delta} d_j\alpha_j$ *with* $d_j \geq 0$, *and*
- $s_i\sigma(\rho) - \rho = -\sum_{\alpha_j\in\Delta} e_j\alpha_j$ *with* $e_j \geq 0$.

Then:

1. *If $\ell(\sigma s_i) > \ell(\sigma)$, then $d_j \geq c_j$ for all j and $d_k > c_k$ for at least one $\alpha_k \in \Delta$.*
2. *If $\ell(s_i\sigma) > \ell(\sigma)$, then $e_j \geq c_j$ for all j with $e_i > c_i$.*

Proposition 2.1 implies that by appending simple reflections that increase the length of elements known to be in the alternation set until the Weyl group element created results in a weight with at least one negative coefficient results in an exhaustive list of elements in the Weyl alternation set.

2.5 Program optimizing

Though this project is complete, it may be worthwhile to optimize the program. Should the partition tree program be optimized for memory usage, it could be possible to complete the E_8 computation. The advantage of the first program is that it allows the actual individual partitions to be listed from the partition tree. However, the method of series expansion allows for much quicker computations. Where the first method took approximately 169.33 s to run E_6, the method by series expansion only took 109.14 s. Notice that this was run on a build of SageMath that did not allow for the code to be run in parallel. From a computer scientist's viewpoint, it would be interesting to see the increase in speed if the first program was parallelized and possibly run on a graphics processing unit (GPU), instead of a CPU, especially because of the independent nature of each branch of the partition tree.

References

1. Baldoni, W., Beck, M., Cochet, C., Vergne, M.: Volume computation for polytopes and partition functions for classical root systems. Discret. Comput. Geom. **35**(4), 551–595 (2006)
2. Baldoni, W., Vergne, M.: Kostant partitions functions and flow polytopes. Transform. Groups **13**(3–4), 447–469 (2008)
3. Barvinok, A.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Math. Oper. Res. **19**(4), 769–779 (1994)
4. Barvinok, A.: Lattice points and lattice polytopes. In: Handbook of Discrete and Computational Geometry, CRC Press Ser. Discrete Math. Appl., pp. 133–152. CRC, Boca Raton (1997). ,
5. Barvinok, A., Pommersheim, J.E.: An algorithmic theory of lattice points in polyhedra. In New Perspectives in Algebraic Combinatorics (Berkeley, CA, 1996–97). Volume 38 of Math. Sci. Res. Inst. Publ., pp. 91–147. Cambridge University Press, Cambridge (1999)
6. Berenstein, A.D., Zelevinsky, A.V.: Tensor product multiplicities and convex polytopes in partition space. J. Geom. Phys. **5**(3), 453–472 (1988)
7. Billey, S., Guillemin, V., Rassart, E.: A vector partition function for the multiplicities of $\mathfrak{sl}_k(\mathbb{C})$. J. Algebra **278**(1), 251–293 (2004)
8. Cochet, C.: Vector partition function and representation theory. In: Conference Proceedings Formal Power Series and Algebraic Combinatorics, p. 12, 2005
9. Deckart, R.W.: On the combinatorics of Kostant's partition function. J. Algebra **96**(1), 9–17 (1985)
10. Fernández-Núñez, J., García-Fuertes, W., Perelomov, A.M.: On the generating function of weight multiplicities for the representations of the Lie algebra C_2. J. Math. Phys. **56**(4), 041702 (2015)
11. Goodman, R., Wallach, N.R.: Symmetry. Representations and Invariants. Springer, New York (2009)
12. Gupta, R.K.: Characters and the q-analog of weight multiplicity. J. Lond. Math. Soc. **2**(1), 68–76 (1987)
13. Harris, P.E.: Chapter 9. In: Wootton, A., Peterson, V., Lee, C. (eds.) A Primer for Undergraduate Research, Foundations for Undergraduate Research in Mathematics. Birkhäuser, Basel (to appear)

14. Harris, P.E.: Combinatorial problems related to Kostant's weight multiplicity formula. Doctoral dissertation, University of Wisconsin-Milwaukee, Milwaukee, WI (2012)
15. Harris, P.E.: Kostant's weight multiplicity formula and the Fibonacci numbers. arXiv:1111.6648 [math.RT]
16. Harris, P.E.: On the adjoint representation of $\mathfrak{sl}_n$ and the Fibonacci numbers. C. R. Math. Acad. Sci. Paris **349**, 935–937 (2011)
17. Harris, P.E., Insko, E., Omar, M.: The q-analog of Kostant's partition function and the highest root of the simple Lie algebras (2016). http://arxiv.org/pdf/1508.07934
18. Harris, P.E., Insko, E., Simpson, A.: Computing weight q-multiplicities for the representations of the simple Lie algebras (2017). http://arxiv.org/pdf/1710.02183
19. Harris, P.E., Insko, E., Simpson, A.: GitHub code download. https://github.com/antman1935/lie_algebras
20. Harris, P., Insko, E., Williams, L.: The adjoint representation of a classical Lie algebra and the support of Kostant's weight multiplicity formula. J. Comb. **7**(1), 75–116 (2016)
21. Humphreys, J.E.: Reflection Groups and Coxeter Groups. Cambridge Universty Press, Cambridge (1997)
22. Kostant, B.: A formula for the multiplicity of a weight. Proc. Nat. Acad. Sci. USA **44**, 588–589 (1958)
23. Knapp, A.W.: Lie Groups Beyond an Introduction. Birkhäuser Boston Inc., Boston (2002)
24. Lusztig, G.: Singularities, character formulas, and a q-analog of weight multiplicities. Astérisque **101**(102), 208–229 (1983)
25. Sagan, B.: The Symmetric Group: Representations, Combinatorial Algorithms, and Symmetric Functions, pp. 141–144. Springer, New York (2001)
26. Schmidt, J.R., Bincer, A.M.: The Kostant partition function for simple Lie algebras. J. Math. Phys. **25**(8), 2367–2373 (1984)