5-2014

Computational Topics in Lie Theory and Representation Theory

Thomas J. Apedaile
Utah State University

COMPUTATIONAL TOPICS IN LIE THEORY AND REPRESENTATION THEORY

by

Thomas J. Apedaile

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Mathematics

Approved:

Ian Anderson Major Professor	Nathan Geer Committee Member
Zhaohu Nie Committee Member	Mark R McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2014

ABSTRACT

Computational Topics in Lie Theory and Representation Theory

by

Thomas J. Apedaile, Master of Science

Utah State University, 2014

Major Professor: Dr. Ian Anderson
Department: Mathematics and Statistics

The computer algebra system Maple contains a basic set of commands for working with Lie algebras. The purpose of this thesis was to extend the functionality of these Maple packages in a number of important areas. First, programs for defining multiplication in several types of Cayley algebras, Jordan algebras and Clifford algebras were created to allow users to perform a variety of calculations. Second, commands were created for calculating some basic properties of finite-dimensional representations of complex semisimple Lie algebras. These commands allow one to identify a given representation as direct sum of irreducible subrepresentations, each one identified by an invariant highest weight. Third, creating an algorithm to calculate the Lie bracket for Vinberg's symmetric construction of Freudenthal's Magic Square allowed for a uniform construction of all five exceptional Lie algebras. Maple examples and tutorials are provided to illustrate the implementation and use of the algebras now available in Maple as well as the tools for working with Lie algebra representations.

(255 pages)

PUBLIC ABSTRACT

Computational Topics in Lie Theory and Representation Theory

by

Thomas J. Apedaile, Master of Science

Utah State University, 2014

Major Professor: Dr. Ian Anderson
Department: Mathematics and Statistics

The computer algebra system Maple contains a basic set of commands for working with Lie algebras and matrices. The purpose of this thesis was to extend the functionality of these Maple packages in a number of important areas. First, programs for defining multiplication in several different types of algebras were created to allow users to perform a wider variety of calculations. Second, commands were created for calculating some basic properties of matrix representations of semisimple Lie algebras. This allows a user to identify a given matrix representation by a collection of integers which do not change when the basis of the representation is changed. These integers, called highest weights, uniquely identify the representation. Third, an algorithm was created to allow for a uniform construction of all five exceptional Lie algebras. Maple examples and tutorials are provided to illustrate the implementation and use of the algebras now available in Maple as well as the tools for working with Lie algebra representations.

(255 pages)

ACKNOWLEDGMENTS

I would like to thank Dr. Ian Anderson for his infinite patience, prodding and expertise. Without his help and encouragement, I doubt I would have learned so much.

I would also like to thank Dr. Zhaohu Nie for the time he took out of his busy schedule to sit down with and brainstorm on problems that helped me understand my thesis better.

Most of all, I am grateful for my wonderful family and amazing friends, for their support over the years and for all the times I would force them to listen to me talk about every breakthrough I encountered. I know by the looks on their faces that it was painful at times, but helped keep the fire alive.

Thomas J. Apedaile

CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

SYMBOLS

$[X, Y]$	Commutator/Lie bracket	$XY - YX$
(x, y, z)	Associator	$(xy)z - x(yz)$
X_0	Trace free part of a matrix X	$X - \frac{tr(X)}{n} \cdot I_n$
$x \circ y$	Jordan product	$\frac{1}{2}(xy + yx)$
E_i^j	A square matrix with 1 in the i^{th} row and j^{th} column	
$\Im(A)$	The imaginary part of an element A	$\frac{1}{2}(A - \overline{A})$
$\Re(A)$	The real part of an element A	$\frac{1}{2}(A + \overline{A})$

CHAPTER 1
INTRODUCTION

The study of symmetries plays a prominent role in the study of mathematics. Lie groups provide a way to express the concept of a continuous family of symmetries for geometric objects. The fundamental theorems of Lie describe a connection between a Lie group and a corresponding Lie algebra. Given a Lie algebra, one can obtain connected Lie groups which are (at worst) locally isomorphic. Also, any Lie group produces a Lie algebra. A fundamental discovery of Sophus Lie is that many important questions regarding Lie groups can be answered by studying their counterpart Lie algebras. Questions regarding Lie algebras are often answered using techniques from linear algebra.

The semisimple complex Lie algebras were completely classified (although the proofs were not complete) by Wilhelm Killing in the late 1800s. In 1894, Élie Cartan completed the proofs in his PhD thesis and shortly thereafter classified the semisimple Lie algebras over the real numbers. In 1905, Eugenio Elia Levi proved that any finite-dimensional Lie algebra can be written as a semidirect product of a solvable ideal and a semisimple subalgebra [6], also known as **Levi Decomposition**.

The semisimple Lie algebras can be divided into five different classes. The first four classes each describe an infinite number of Lie algebras. They have a very straightforward construction using matrices, together with the commutator operation acting as the Lie bracket. The fifth class, of which there are only five Lie algebras, are called the **exceptional** Lie algebras, and do not have such a simple method of construction.

The goals of this thesis are three-fold. The first (chapter 2) was to create multiplication rules for several different types of algebras such as the Quaternions, the Octonions and a few other Cayley algebras, Jordan algebras, and Clifford algebras. The libraries used for creating the multiplication rules will allow users to perform many calculations which require the use of these algebras. As an application of chapter 2, we were able

to construct the exceptional Lie algebra $\mathfrak{g}_2$ using the octonions, and $\mathfrak{f}_4$ using a Jordan algebra over the octonions. The second goal (chapter 3) was to write Maple functions which decompose and classify explicit matrix representations of complex semisimple Lie algebras. Programs were written to compute the set of highest weight vectors and the invariant highest weights for all the irreducible subrepresentations. The third goal (chapter 4), building on the first goal, was to use the Cayley algebras, as constructed in chapter 2, to create all five exceptional Lie algebras. A Maple routine was made to implement Vinberg's construction and calculate the structure constants of the exceptional semisimple Lie algebras. Then, we verify Vinbergs version of Freudenthal's Magic Square of Lie algebras.

In chapter 2, we will briefly discuss some of the theory and development of the Cayley algebras. The multiplication rules for the quaternions, the octonions, as well as the split complex, split quaternion, and split octonion algebras will be constructed using the Cayley-Dickson construction. The discussion will continue by defining Jordan algebras, and showing how a Jordan algebra can be created using an associative algebra. Construction of Clifford algebras will be addressed as well as the algorithms used for identifying a basis and computing a product of the basis elements. The final section will be a discussion of Lie algebras. We will examine some of the theory used in the classification of Lie algebras, and illustrate several important properties of the semisimple Lie algebras.

For each of the algebras discussed in chapter 2, examples will be given in the text to illustrate how Maple can be used to create the algebras and show some of the calculations which are available. For example, we can create the algebra of the quaternions,

```
> HData := AlgebraLibraryData("Quaternions",H):
> DGsetup(HData,[e],[omega]);
```

$$algebra\ name:\ \ H$$

and multiply two elements from the algebra together.

```
> evalDG( (2*e1+1/2*e3).(e1-e2-e4) );
```

$$2e1-\frac{5}{2}e2+\frac{1}{2}e3-\frac{3}{2}e4$$

A representation of a Lie algebra is a concrete realization of the Lie algebra as a subalgebra of $\mathfrak{gl}_n$. Such matrix representations arise naturally in a wide range of applications in Differential Geometry and in Physics. For any complex semisimple Lie algebra, there are an infinite number of finite-dimensional representations. However, it is possible to obtain a complete classification of these representations. Chapter 3 summarizes some of the basic theory of Lie algebra representations. In short, every finite-dimensional representation of a semisimple Lie algebra admits a decomposition into a direct sum of **irreducible** representations. Furthermore, each irreducible representation can be identified by a string of nonnegative integers which is invariant. I have written routines in Maple that will extract properties of a representation which are then used to identify the invariants of the irreducible subrepresentations in the decomposition.

For example, let ρ denote the standard representation of $\mathfrak{sl}_3(\mathbb{C})$. We can create this representation in Maple as follows.

```
> LD := SimpleLieAlgebraData("sl(3)",A2):
> DGsetup(LD);
```

Lie algebra: A2

```
> DGsetup([v1,v2,v3],V):
> rho := Representation( A2, V, StandardRepresentation(A2) ):
```

We can also create the representation of the tensor product of the standard representation with itself, namely $\varphi = \rho \otimes \rho$.

```
> DGsetup([w1,w2,w3,w4,w5,w6,w7,w8,w9],W):
> phi := TensorProductOfRepresentations( [rho,rho] , W ):
```

This representation is not irreducible. However, it can be decomposed into a sum of two irreducible representations.

```
> DecomposeRepresentation(phi):
```

$$\left[\ \Gamma_{2,0} \oplus \Gamma_{0,1}\ \right]$$

This output tells us that the representation decomposes into two irreducible representations with highest weights $(2,0)$ and $(0,1)$ respectively.

Chapter 4 will be a detailed discussion on the Vinberg construction of the Freudenthal Magic Square Lie algebras. First, we will prove that the formula

$$D_{a,b}(x) = [[a,b],x] - 3(a,b,x)$$

is a derivation on any alternative algebra. We will then discuss the Vinberg construction as well as explain parts of the complicated Lie bracket which he defined. Afterwards, we will provide a detailed explanation of the Maple procedures used in the calculation of Vinberg's Lie bracket.

As an example, we can create the exceptional Lie algebra by using Vinberg's Lie bracket on the vector space $\mathfrak{der}(\mathbb{O}) \oplus \mathfrak{der}(\mathbb{R}) \oplus \mathfrak{sa}_3(\mathbb{O} \otimes \mathbb{R})$.

```
> LD := MagicSquare("Octonion","Real",F4):
> DGsetup(LD);
```

Lie algebra: F4

Using some of the existing commands in Maple, we can compute a Cartan subalgebra.

```
> CSA := CartanSubalgebra(F4);
```

CSA:=[e1,e2,e15,e42+e49]

Using this Cartan subalgebra, we can compute a root space decomposition and a set of simple roots.

```
> RSD := RootSpaceDecomposition(CSA):
> PR := PositiveRoots(RSD):
> SR := SimpleRoots(PR);
```

$$SR := \left[\begin{bmatrix} 0 \\ 6I \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 2I \\ -4I \\ -2I \\ -2I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ -3I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ 3I \end{bmatrix}\right]$$

This will allow us to compute a Cartan matrix, thus allowing us to identify the Lie algebra we created. We will put this Cartan matrix in standard form to make the identification easier to see that we have created the exceptional Lie algebra $\mathfrak{f}_4$.

```
> CM := CartanMatrix(SR,RSD):
> CartanMatrixToStandardForm(CM,SR);
```

$$\left[\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -2 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}, \left[\begin{bmatrix} 0 \\ 6I \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 2I \\ -4I \\ -2I \\ -2I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ 3I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ -3I \end{bmatrix}\right], \text{\textbackslash}F''$$

The appendices include maple worksheets which provide tutorials and examples of how the commands created for this thesis can be used (see appendix *C*). Also, the code (complete with comments) for the procedures created for this thesis are provided in appendix *B*.

Groundwork has also been laid for the development of software that will explicitly construct irreducible representations for a given semisimple Lie algebra. Programs are also being developed to efficiently construct the irreducible subrepresentations of a given representation. The routines developed for this project use many of the procedures found in the DifferentialGeometry package in Maple 17 and Maple 18. The procedures in which I was instrumental in developing will be identified; otherwise it can be assumed that the commands used already existed in Maple.

CHAPTER 2

ALGEBRAS

2.1 Preliminaries

This chapter will focus on the construction of the Maple routines for performing explicit calculations in the Cayley algebras, Jordan algebras, and Clifford algebras. Each section will contain a brief theoretical discussion followed by examples of how the procedures can be used. The last section of this chapter sets the stage for the remainder of this thesis by providing some basic facts about Lie algebras. The code which I wrote to implement these algebras is given in appendix B. Maple tutorials are also included in appendix C which further illustrate how to initialize and use these algebras. We begin by establishing important definitions and notation conventions.

Definition 2.1. An **algebra**, is a vector space A over a field F with a multiplication operator, $\cdot : A \times A \to A$, satisfying:

- $(x + y) \cdot z = x \cdot y + y \cdot z$ and
  $z \cdot (x + y) = z \cdot x + z \cdot y \quad \forall x, y, z \in A.$ (Distributive Law)
- $a(x \cdot y) = (ax) \cdot y = x \cdot (ay)$ for every $a \in F$ and $x, y \in A.$ (Scalar Multiplication)

We will mainly be interested in algebra over $\mathbb{R}$, and we will only work with finite-dimensional algebras. If $\mathcal{B} = \{e_1, e_2, ..., e_n\}$ is a basis of the n-dimensional vector space A, then every element of the algebra A can be written as a linear combination of the elements in $\mathcal{B}$. The product of two basis elements can always be written as

$$e_i \cdot e_j = c_{ij}^k e_k,$$

where the Einstein summation convention is used. The coefficients $c_{ij}^k \in \mathbb{R}$ are called the **structure constants**. Once multiplication of the basis elements are defined, a general product can be computed using the distributive property.

An algebra A is called a **division algebra** if for any element $a \in A$ and any non-zero element $b \in A$ there exists unique elements $x, y \in A$ such that $a = bx$ and $a = yb$.

An algebra A is said to be **associative** if $(xy)z = x(yz)$ for all $x, y, z \in A$.

An algebra A is called **power-associative** if the subalgebra generated by one element is an associative algebra. Equivalently, this means that if an element x in the algebra is multiplied by itself multiple times, then the order in which the multiplications are carried out do not matter. If A is power associative, then, for example,

$$x(x(xx)) = (xx)(xx) = ((xx)x)x = (x(xx))x = x((xx)x).$$

for every $x \in A$. In practice, we will simple denote such a product as x^p.

An algebra A is said to be **alternative** if every subalgebra generated by two elements is an associative algebra. Equivalently, an algebra is alternative if for every $x, y \in A$, $x(xy) = (xx)y$ and $(yx)x = y(xx)$. As shown in chapter 3 of 'An Introduction to Nonassociative Algebras' [7], this means that for any $x, y \in A$, $(x, x, y) = (x, y, x) = (y, x, x) = 0$. We also have a family of identities for alternative algebras called the **Moufang identities**

$$\begin{aligned} (xax)y &= x(a(xy)) \\ y(xax) &= ((yx)a)x \\ (xy)(ax) &= x(ya)x. \end{aligned} \tag{2.1}$$

Using these identities, we can establish the following theorem.

Theorem 2.1. *The associator is a skew-symmetric operator on an alternative algebra.*

Proof: Let A be an alternative algebra. It suffices to show that $(x, y, z) = -(y, x, z)$ and $(x, y, z) = -(z, y, x)$ for every $x, y, z \in A$.

Let $x, y, z \in A$ be given. We know that $(x, x, y) = 0$ for all $x, y \in A$. Therefore, using the multi-linearity of the associator, we see that

$$\begin{aligned}(x+y, x+y, z) &= (x, x+y, z) + (y, x+y, z) \\ &= (x, x, z) + (x, y, z) + (y, x, z) + (y, y, z) \\ &= 0 + (x, y, z) + (y, x, z) + 0.\end{aligned}$$

Therefore, we have that $(x, y, z) + (y, x, z) = 0$, which implies that $(x, y, z) = -(y, x, z)$. The result $(x, y, z) = -(z, y, x)$ is found similarly. From these two results, we have that the associator is skew-symmetric. □

If A is an associative algebra, then every subalgebra must be associative as well. This means that every associative algebra is power-associative and every associative algebra is also alternative. For more details on power-associative and alternative algebras, see "An Introduction to Nonassociative Algebras" [7].

An algebra A is called a **$*$-algebra** if there is a real-linear map $A \to A$, denoted by $a \mapsto \overline{a}$, with the properties $\overline{\overline{a}} = a$ and $\overline{a \cdot b} = \overline{b} \cdot \overline{a}$ for all $a, b \in A$. This mapping is called **conjugation**. For example, the complex numbers form a $*$-algebra, where conjugation is simply complex conjugation.

We say that a $*$-algebra A is **nicely normed** if $a + \overline{a} \in \mathbb{R}$ and $a \cdot \overline{a} = \overline{a} \cdot a > 0$ for all nonzero $a \in A$. If the A is nicely normed, then we can define the **real** and **imaginary** components of an element a as follows:

$$\mathrm{Re}(a) = (a + \overline{a})/2 \qquad\qquad \mathrm{Im}(a) = (a - \overline{a})/2$$

We can also define a norm on A by $\|a\|^2 = \langle a, a \rangle = a \cdot \overline{a}$.

Given any two real algebras K and M, we can define the tensor product of K and M as the tensor product of the vectors spaces with multiplication defined as

$$(x \otimes a) \cdot (y \otimes b) = (xy) \otimes (ab).$$

Then $K \otimes M$ is an algebra. We can show that the tensor product of two alternative algebras is alternative, and the tensor product of two $*$-algebras is a $*$-algebra.

Theorem 2.2. *The tensor product of two alternative algebras is alternative.*

Proof: Let K and M be alternative algebras. This means that $(x,x,y) = (x,y,x) = (y,x,x) = 0$ for all $x, y \in K$ (and similarly so for any two elements in M). By the multilinearity property of the associator, it suffices to show that the associator on any two tensors in $K \otimes M$ is zero. Let $x \otimes u, y \otimes v \in K \otimes M$. Then

$$\begin{aligned}(x \otimes u, x \otimes u, y \otimes v) &= ((x \otimes u)(x \otimes u))(y \otimes v) - (x \otimes u)((x \otimes u)(y \otimes v)) \\ &= (x^2 \otimes u^2)(y \otimes v) - (x \otimes u)(xy \otimes uv) \\ &= x^2 y \otimes u^2 v - x(xy) \otimes u(uv).\end{aligned}$$

Because the algebras are alternative, this means that $x(xy) = x^2 y$ and $u(uv) = u^2 v$. Therefore, we have that

$$\begin{aligned}(x \otimes u, x \otimes u, y \otimes v) &= x^2 y \otimes u^2 v - x(xy) \otimes u(uv) \\ &= x^2 y \otimes u^2 v - x^2 y \otimes u^2 v \\ &= 0.\end{aligned}$$

The proofs that $(x \otimes u, y \otimes v, x \otimes u) = (y \otimes v, x \otimes u, x \otimes u) = 0$ are similar. □

Theorem 2.3. *The tensor product of two $*$-algebras is a $*$-algebra.*

Proof: Let K and M be $*$-algebras. We know that $\mathbb{K} \otimes \mathbb{M}$ is an algebra. We need only verify that there exists a conjugation map $\mathbb{K} \otimes \mathbb{M} \to \mathbb{K} \otimes \mathbb{M}$. Let $x \otimes u \in \mathbb{K} \otimes \mathbb{M}$ be given and define $\overline{x \otimes u} = \overline{x} \otimes \overline{u}$. We claim that the linear extension of this map satisfies the definition of conjugation. It suffices to show that the properties are satisfied on tensors. Let $x \otimes u, y \otimes v \in \mathbb{K} \otimes \mathbb{M}$ be given. Then

$$\begin{aligned}\overline{(x \otimes u) \cdot (y \otimes v)} &= \overline{xy \otimes uv} \\ &= \overline{xy} \otimes \overline{uv} \\ &= \overline{y}\ \overline{x} \otimes \overline{v}\ \overline{u} \\ &= (\overline{y} \otimes \overline{v}) \cdot (\overline{x} \otimes \overline{u}) \\ &= \overline{y \otimes v} \cdot \overline{x \otimes u}.\end{aligned}$$

We also need to show that conjugating an element twice return the original element.

$$\begin{aligned}\overline{\overline{x \otimes u}} &= \overline{\overline{x} \otimes \overline{u}} \\ &= \overline{\overline{x}} \otimes \overline{\overline{u}} \\ &= x \otimes u.\end{aligned}$$

Thus, these result show that this operation on a tensor is consistent with the definition of conjugation. Therefore, by linearly extending this operation, we have defined conjugation on the algebra $\mathbb{K} \otimes \mathbb{M}$, making $\mathbb{K} \otimes \mathbb{M}$ a $*$-algebra. □

So, given two $*$-algebras K and M, $K \otimes M$ is a $*$-algebra. This means we can define the conjugate transpose of a matrix over $K \otimes M$, $X^* = \overline{X}^T$. Thus we can define Hermitian and Skew-hermitian matrices, $X^* = X$ and $X^* = -X$, over $K \otimes M$. Skew-hermitian matrices over $K \otimes M$ will be used more in chapter 4.

2.2 Cayley-Dickson Construction

The quaternion and octonion algebras are very useful and important in many different fields of study. For many examples of their utility, see the beautiful review article by Baez titled "The Octonions." This section will talk about how the quaternions and octonions can be created in theory, and show how calculations using these algebras can be done in Maple. Each of the Cayley algebras $\mathbb{R}$, $\mathbb{C}$, $\mathbb{H}$, and $\mathbb{O}$ will be used later to create the exceptional Lie algebras (see appendices C.6 and C.7).

For details on the Cayley-Dickson construction, see "Spinors and Calibrations" [4]. The Cayley-Dickson Construction allows us to construct a new algebra from a given $*$-algebra. This method of construction will allow us to obtain all of the normed division algebras [1], namely the real numbers, the complex numbers, the quaternions and the octonions. The bases created by the construction outlined below are the bases used in the Maple procedures. We will briefly discuss the general construction, and then look at specific examples which I developed for this project and explore some of the Maple commands for performing calculations.

Let A be a normed $*$-algebra and let λ be an indeterminate with the property $\lambda^2 = \pm 1$ (the two different values that λ^2 can take on will allow us to construct different flavors of algebras using the Cayley- Dickson construction). Then we define two algebras $A(+)$ and $A(-)$, where $A(\pm) = A \oplus A$ as a vector space and multiplication is given to be

$$(a, b) \cdot (c, d) = \Big(ac + \lambda^2 \overline{d} b \ , \ da + b\overline{c}\Big), \tag{2.2}$$

where $\lambda^2 = -1$ for $A(+)$ and $\lambda^2 = 1$ for $A(-)$. We also define conjugation in $A(\pm)$ to be $\overline{(a, b)} = (\overline{a}, -b)$.

The following theorem [4] details some of the algebraic properties which are preserved in the construction.

Theorem 2.4. *Suppose $A(\pm)$ is the algebra defined using the Cayley-Dickson construction from a normed algebra A.*

(i) $A(\pm)$ is commutative if and only if $A = \mathbb{R}$.

(ii) $A(\pm)$ is associative if and only if A is commutative and associative.

(iii) $A(\pm)$ is alternative, $A(\pm)$ is normed, and A is associative are all equivalent.

Beginning with the real numbers, we can construct the complex numbers as $\mathbb{C} = \mathbb{R}(+)$. We can then use the complex numbers to construct the *quaternions* as $\mathbb{H} = \mathbb{C}(+)$. Finally, we will build the *octonions* as $\mathbb{O} = \mathbb{H}(+)$. More on these algebras will be said later. The algebras $\mathbb{R}(-)$, $\mathbb{C}(-)$, and $\mathbb{H}(-)$ are called the split-complex, split-quaternion, and split-octonion algebras respectively. We will discuss these algebras later.

First we will show how the complex numbers can be created from the real numbers using the Cayley-Dickson Construction. Notice that a complex number $a + bi$ can be thought of as an ordered pair (a, b) where $a, b \in \mathbb{R}$ such that addition is done component-wise and multiplication in the complex numbers is given by

$$(a, b) \cdot (c, d) = (ac - db, da + bc).$$

The complex conjugate is defined as

$$\overline{(a, b)} = (a, -b).$$

Because $\overline{a} = a \in \mathbb{R}$, notice that we have created $\mathbb{R}(+)$ as defined above, and that this algebra is exactly the complex numbers, $\mathbb{C}$. Now that we have used the Cayley-Dickson Construction to create the complex numbers from the real numbers, we can now use this construction process on the complex numbers to build the *quaternion* algebra.

2.2.1 Quaternions

The quaternions, denoted by $\mathbb{H}$ in honor of William Rowan Hamilton, can be fully described by the following relations between the basis elements

$$i^2 = j^2 = k^2 = ijk = -1.$$

We can create the quaternions by extending the complex numbers $\mathbb{C}$ using the Cayley-Dickson Construction.

Using the standard basis of $\mathbb{C}$, we can easily create a basis of $\mathbb{C} \oplus \mathbb{C}$, namely,

$$\{(1, 0), (i, 0), (0, 1), (0, i)\}$$

To make calculations easier to follow (especially in the next section on the octonions), let $1 = (1,0)$, $\hat{i} = (i,0)$, $\hat{j} = (0,1)$ and $\hat{k} = (0,i)$.

We can use the multiplication rule given by equation (2.2) to compute structure constants for $\mathbb{C}(+)$ using the above basis elements of $\mathbb{C} \oplus \mathbb{C}$. For example,

$$\hat{i}^2 = (i,0) \cdot (i,0) = (i \cdot i - 0, 0 - 0) = (-1,0) = -1$$
$$\hat{j}^2 = (0,1) \cdot (0,1) = (0 - 1 \cdot 1, 0 - 0) = (-1,0) = -1$$
$$\hat{i} \cdot \hat{k} = (i,0) \cdot (0,i) = (i \cdot 0 - \overline{i} \cdot 0\ ,\ i \cdot i - 0) = (0,-1) = -\hat{j}$$

The rest of the products can be found in table 2.1.

TABLE 2.1: Multiplication Table - Quaternions

$\cdot$	1	$\hat{i}$	$\hat{j}$	$\hat{k}$
1	1	$\hat{i}$	$\hat{j}$	$\hat{k}$
$\hat{i}$	i	-1	$\hat{k}$	$-\hat{j}$
$\hat{j}$	$\hat{j}$	$-\hat{k}$	-1	$\hat{i}$
$\hat{k}$	$\hat{k}$	$\hat{j}$	$-\hat{i}$	-1

Next, we define conjugation on $\mathbb{C}(+)$ to be $\overline{(a,b)} = (\overline{a}, -b)$. This means that $\overline{1} = 1$, $\overline{\hat{i}} = -\hat{i}$, $\overline{\hat{j}} = -\hat{j}$ and $\overline{\hat{k}} = -\hat{k}$. Now that conjugation is defined on $\mathbb{H}$, we can define the **real** and **imaginary** parts of a quaternion. Given $x \in \mathbb{H}$, the real and imaginary parts of x are $Re(x) = \frac{1}{2}(x + \overline{x})$ and $Im(x) = \frac{1}{2}(x - \overline{x})$ respectively. If $Re(x) = x$, then we say that x is real. If $Im(x) = x$, then we say that x is pure imaginary.

We see that $Re(1) = \frac{1}{2}(1 + \overline{1}) = \frac{1}{2}(1+1) = \frac{1}{2}(2 \cdot 1) = 1$ and $Im(1) = \frac{1}{2}(1 - \overline{1}) = \frac{1}{2}(1-1) = 0$, making the basis element 1 real. Also, notice that $Re(\hat{i}) = \frac{1}{2}(\hat{i} + \overline{\hat{i}}) = \frac{1}{2}(\hat{i} - \hat{i}) = 0$ and $Im(\hat{i}) = \frac{1}{2}(\hat{i} - \overline{\hat{i}}) = \frac{1}{2}(\hat{i} + \hat{i}) = \hat{i}$, making $\hat{i}$ pure imaginary. It can also be shown that $\hat{j}$ and $\hat{k}$ are also pure imaginary.

As an aid, we can use figure 2.1 to determine the rules for multiplying the imaginary basis elements.

Using multiplication of the basis elements and the distributive property, we can compute any product of quaternions. Notice that $\hat{i} \cdot \hat{j} = \hat{k}$ but $\hat{j} \cdot \hat{i} = -\hat{k}$. This means that the quaternions form a non- commutative algebra. By checking associativity on the basis elements, we find that the basis elements are all associative. This means that $\mathbb{H}$ is

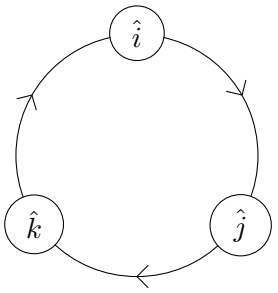


FIGURE 2.1: Multiplication for Imaginary Quaternion Basis Elements

an associative algebra. Furthermore, it can be shown that $\mathbb{H}$ is a nicely normed division algebra.

The code for the quaternion library is located in B.1.1. A tutorial for initializing a quaternion algebra and how it can be used is given in C.1.

Example:

We will illustrate how the quaternions can be used in Maple. The algebras in this chapter can be created using the command `AlgebraLibraryData` which is in the `LieAlgebras` package. The input values for the function are a string identifying the name of the algebra, and the name for the frame.

Although I did not develop the command `AlgebraLibraryData`, I did write up the libraries that `AlgebraLibraryData` calls. First we read in the algebra data and call the name of the frame `H`.

```
> QuaternionData := AlgebraLibraryData("Quaternions",H):
```

Next, we use the command `DGsetup` to initialize the name of the frame. We have labeled the four basis vectors e, i, j, and k to match our table shown above, and have labeled the covectors $\omega 1$, $\omega 2$, $\omega 3$, and $\omega 4$ (we will not be using the covectors here, but a label must be given).

```
> DGsetup(QuaternionData,[e,i,j,k],[omega]);
```

algebra name: H

Now that the frame has been initialized, we are ready to try out some computations. Let's create a general object in $\mathbb{H}$, call it `X`.

```
> X := evalDG(a*e+b*i+c*j+d*k);
```

$$X{:=}ae{+}bi{+}cj{+}dk$$

Now we can multiply this object by another object in the algebra. To keep things simple, let's multiply `X` by another basis element in $\mathbb{H}$. Note that multiplication in the algebra is done by using the period and wrapping the result inside `evalDG` to evaluate.

```
> evalDG(X.k);
```

$$-de{+}ci{-}bj{+}ak$$

2.2.2 Octonions

The algebra of octonions, denoted by $\mathbb{O}$, is an 8-dimensional algebra obtained by extending the quaternions via the Cayley-Dickson Construction. The basis used in the library is the basis obtained by using the previous description of the quaternions in the Cayley-Dickson Construction. The maple code for the octonion library is located in appendix B.1.2. A maple worksheet can be found in appendix C.2 which shows how to initialize an octonion algebra and illustrates some of the available computations.

Let $\{1, \hat{i}, \hat{j}, \hat{k}\}$ be the basis of $\mathbb{H}$ that we just constructed. Then, just like we did with $\mathbb{H}$, the order pairs

$$\mathcal{B} = \{(1,0), (\hat{i},0), (\hat{j},0), (\hat{k},0), (0,1), (0,\hat{i}), (0,\hat{j}), (0,\hat{k})\}$$

form a basis of $\mathbb{H} \oplus \mathbb{H}$. To simplify our notation, let's label these basis elements as $\{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$ and note that the multiplicative identity in $\mathbb{O}$ is denoted by $e_1 = (1,0)$.

Using equation (2.2) for $\mathbb{H}(+),$, let's compute the product of $e_3 \cdot e_7$ where $e_3 = (\hat{j}, 0)$ and $e_7 = (0, \hat{j})$. Doing so yields

$$(\hat{j},0) \cdot (0,\hat{j}) = (\hat{j} \cdot 0 - \overline{\hat{j}} \cdot 0, \hat{j} \cdot \hat{j} + 0 \cdot \overline{0}) = (0,-1) = -(0,1).$$

Therefore, we see that $e_3 \cdot e_7 = -e_5$. Doing these calculations for all of the basis elements yields the following table specifying the product of the basis elements of $\mathbb{O}$.

TABLE 2.2: Multiplication Table - Octonions

$\cdot$	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
e_1	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
e_2	e_2	$-e_1$	e_4	$-e_3$	e_6	$-e_5$	$-e_8$	e_7
e_3	e_3	$-e_4$	$-e_1$	e_2	e_7	e_8	$-e_5$	$-e_6$
e_4	e_4	e_3	$-e_2$	$-e_1$	e_8	$-e_7$	e_6	$-e_5$
e_5	e_5	$-e_6$	$-e_7$	$-e_8$	$-e_1$	e_2	e_3	e_4
e_6	e_6	e_5	$-e_8$	e_7	$-e_2$	$-e_1$	$-e_4$	e_3
e_7	e_7	e_8	e_5	$-e_6$	$-e_3$	e_4	$-e_1$	$-e_2$
e_8	e_8	$-e_7$	e_6	e_5	$-e_4$	$-e_3$	e_2	$-e_1$

From this table, notice that e_i is a square root of -1 for $2 \leq i \leq 8$. Also, $e_i e_j = -e_j e_i$ for every $i \neq j$.

It is a well known fact that the octonions form a nonassociative algebra. This can be verified by finding three basis elements from $\mathbb{H} \oplus \mathbb{H}$ which do not satisfy the associativity property using equation (2.2). For example, let's compute the products $(e_2 \cdot e_3) \cdot e_5$ and $e_2 \cdot (e_3 \cdot e_5)$. Recalling that $e_2 = (\hat{i}, 0)$, $e_3 = (\hat{j}, 0)$ and $e_5 = (0, 1)$, this gives us

$$\Big((\hat{i}, 0) \cdot (\hat{j}, 0)\Big) \cdot (0, 1) = (\hat{k}, 0) \cdot (0, 1) = (0, -\hat{k}) = -(0, \hat{k}),$$
$$(\hat{i}, 0) \cdot \Big((\hat{j}, 0) \cdot (0, 1)\Big) = (\hat{i}, 0) \cdot (0, -\hat{j}) = (0, \hat{k}).$$

Because $-(0, \hat{k}) \neq (0, \hat{k})$, we conclude that $(e_2 \cdot e_3) \cdot e_5 \neq e_2 \cdot (e_3 \cdot e_5)$, making $\mathbb{O}$ a nonassociative algebra.

Although the octonions are noncommutative and nonassociative, because $\mathbb{H}$ is associative, by Theorem 2.4 we know that $\mathbb{O}$ is alternative and normed.

Similar to our analysis of $\mathbb{H}$, we can define conjugation in $\mathbb{O}$, allowing us to also define real and imaginary parts of an element. We define conjugation in $\mathbb{O}$ by defining conjugation on $\mathbb{H} \oplus \mathbb{H}$ as follows, $\overline{(a, b)} = (\overline{a}, -b)$. It is straightforward to check that this rule for conjugation satisfies the conditions to make $\mathbb{O}$ a $*$-algebra. It will not be shown here, but it can easily be checked, that e_1 is real and the elements $e_i \in \mathbb{O}$ (for $i \neq 1$) are purely imaginary.

The product rules for the imaginary elements can be summarized using the **Fano Plane**:

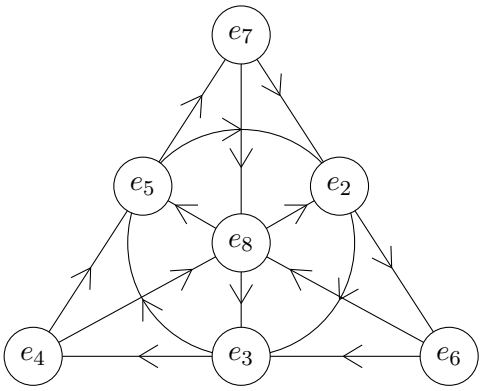


FIGURE 2.2: The Fano Plane: Octonion Multiplication

There are a few additional properties of the octonions that are important to mention at this point. Although the algebra is not associative, there are some very useful properties of the associator map,

$$\begin{aligned}(x,y,z) &= -(y,x,z) = -(z,y,x)\\ (x,y,[u,v]) &= (u,v,[x,y])\end{aligned} \tag{2.3}$$

where $(x,y,z) = (xy)z - x(yz)$ and $[u,v] = uv - vu$ [7]. Because the associator and commutator are multilinear maps, it suffices to verify these formulas on the basis elements.

By the nature of their construction, we know that these properties of the associator are valid for the algebras $\mathbb{R}$, $\mathbb{C}$, $\mathbb{H}$, and $\mathbb{O}$.

At this point we will discuss the **inverse** and **inner product** of the octonions. Let nonzero elements $a, b \in \mathbb{O}$ be given and recall that the norm is defined to be $||a|| = \sqrt{a\overline{a}}$. Because $\mathbb{O}$ is nicely normed, we know that $a \neq 0$ implies $||a|| \neq 0$. We can then define an inverse element of a as follows: $a^{-1} = \overline{a}/||a||^2$. The inner product is defined as $\langle a, b\rangle = \frac{1}{2}(a\overline{b} + b\overline{a})$. Because the complex numbers and quaternions are subalgebras of $\mathbb{O}$, these definitions for the inverse and inner product are also valid for those algebras.

Example:

To conclude our discussion on the octonions, we will demonstrate how the octonion algebra can be obtained in Maple and then we shall illustrate a few computations. As before, we read in the product rules for the basis elements of the algebra using the command `AlgebraLibraryData` (and as before, I did write the library for multiplication of the octonions, but not the command to retrieve and initialize the library).

```
> OctonionData := AlgebraLibraryData("Octonions",O):
> DGsetup(OctonionData);
```

algebra name: O

To work with the algebra, we need to retrieve the basis elements. We do this by using the command `DGinfo`, which is contained in the package `Tools`.

```
> Basis := Tools:-DGinfo(O,"FrameBaseVectors");
```

Basis := [e1, e2, e3, e4, e5, e6, e7, e8]

In order to properly add or multiply any elements together, we must use the command `evalDG`. This command will ensure that the products are evaluated using the DG framework. For example, from the table above, we know that $e_5 \cdot e_6 = e_2$. To evaluate this in Maple,

```
> evalDG( e5.e6 );
```

e2

We can also see that the algebra is non-associative by computing $(e_3 \cdot e_5) \cdot e_6 - e_3 \cdot (e_5 \cdot e_6)$. This will be done by wrapping each product in `evalDG` to make sure that the proper terms are multiplied together. Then we wrap the difference inside `evalDG` to combine the results.

```
> evalDG( evalDG( e3.e5).e6 - e3.evalDG( e5.e6) );
```

2e4

Finally, we can construct a general element in $\mathbb{O}$ as follows:

```
> A := evalDG( add( 'a'[k]*Basis[k] , k=1..8) );
```

$A:=\mathrm{a}_1\mathrm{e}1 + \mathrm{a}_2\mathrm{e}2 + \mathrm{a}_3\mathrm{e}3 + \mathrm{a}_4\mathrm{e}4 + \mathrm{a}_5\mathrm{e}5 + \mathrm{a}_6\mathrm{e}6 + \mathrm{a}_7\mathrm{e}7 + \mathrm{a}_8\mathrm{e}8$

To finish, we will show how to compute the conjugate and inverse of an element. Let's define a new element `X`.

```
> X := evalDG(e1+2*e2+6*e4-4*e5-3*e7-5*e8);
```

$$\texttt{X:=e1+2e2+6e4-4e5-3e7-5e8}$$

We use that command `DGconjugate` to compute the conjugate.

```
> DGconjugate(X);
```

$$\texttt{e1-8e2-6e4+4e5+3e7+5e8}$$

We can find the inverse of the element `X` using the command `AlgebraInverse`.

```
> Y := AlgebraInverse(X);
```

$$\texttt{Y:=}\frac{1}{91}\texttt{e1-}\frac{2}{91}\texttt{e2-}\frac{6}{91}\texttt{e4+}\frac{4}{91}\texttt{e5+}\frac{3}{91}\texttt{e7+}\frac{5}{91}\texttt{e8}$$

We can check this result by multiplying `X` and `Y`.

```
> evalDG(X.Y);
```

$$\texttt{e1}$$

2.2.3 A Brief Discussion of the Cayley Algebras $\mathbf{A}(-)$

The Cayley-Dickson construction allows us to build even more algebras than those we just discussed. In particular, we will construct the algebras $\mathbb{R}(-)$, $\mathbb{C}(-)$, and $\mathbb{H}(-)$. We will not spend time exploring the properties of these algebras other than to discuss the method of their construction. The structure constants for these algebras can also obtained using the command `AlgebraLibraryData`.

We begin by constructing the **split-complex** algebra, denoted by $\mathbb{C}' = \mathbb{R}(-)$. The elements in $\mathbb{C}'$ are numbers of the form

$$x + \lambda y$$

where $x, y \in \mathbb{R}$ and $\lambda^2 = 1$. We can also think of these numbers as ordered pairs of real numbers with conjugation and multiplication defined as follows for each $(a, b), (c, d) \in \mathbb{C}'$,

1. $\overline{(a, b)} = (a, -b)$,
2. $(a, b) \cdot (c, d) = (ac + bd\ ,\ ad + bc)$.

Let $\mathcal{B} = \{(1, 0), (0, 1)\}$ be a basis of $\mathbb{C}'$. By computing the products for the basis elements using the definition of multiplication, we are compute the product of any two elements in $\mathbb{C}'$.

We can now take this idea and expand upon it. For example, we can construct the split-quaternions by constructing $\mathbb{H}' = \mathbb{C}(-)$. Because $\mathbb{H}$ and $\mathbb{H}'$ are equal to $\mathbb{C} \oplus \mathbb{C}$ as vector spaces, we can use the same basis for $\mathbb{H}'$ as we did for $\mathbb{H}$. Then, using equation (2.2), we can compute the structure constants for the algebra $\mathbb{H}'$. The results are shown in the table below.

TABLE 2.3: Multiplication Table - Split-Quaternions

$\cdot$	1	$\hat{i}$	$\hat{j}$	$\hat{k}$
1	1	$\hat{i}$	$\hat{j}$	$\hat{k}$
$\hat{i}$	$\hat{i}$	-1	$\hat{k}$	$-\hat{j}$
$\hat{j}$	$\hat{j}$	$-\hat{k}$	1	$-\hat{i}$
$\hat{k}$	$\hat{k}$	$\hat{j}$	$\hat{i}$	1

To construct the split-octonions, denoted by $\mathbb{O}'$, we use the standard quaternions, $\mathbb{H}$, and extend them to create $\mathbb{O}' = \mathbb{H}(-)$. Again, we can use the same basis for $\mathbb{O}'$ as we did for $\mathbb{O}$, namely,

$$\begin{aligned}\mathcal{B} = \{\ & e_1 = (1, 0), e_2 = (\hat{i}, 0), e_3 = (\hat{j}, 0), e_4 = (\hat{k}, 0), \\ & e_5 = (0, 1), e_5 = (0, \hat{i}), e_5 = (0, \hat{j}), e_5 = (0, \hat{k})\ \}.\end{aligned}$$

Then we can construct a multiplication table for the basis elements of $\mathbb{O}'$.

The algebras $\mathbb{R}(-)$, $\mathbb{C}(-)$, and $\mathbb{H}(-)$ can be used in the Magic Square to produce different types of Lie algebras (see tables 4.3 and 4.4).

TABLE 2.4: Multiplication Table - Split-Octonions

$\cdot$	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
e_1	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
e_2	e_2	$-e_1$	e_4	$-e_3$	e_6	$-e_5$	$-e_8$	e_7
e_3	e_3	$-e_4$	$-e_1$	e_2	e_7	e_8	$-e_5$	$-e_6$
e_4	e_4	e_3	$-e_2$	$-e_1$	e_8	$-e_7$	e_6	$-e_5$
e_5	e_5	$-e_6$	$-e_7$	$-e_8$	e_1	$-e_2$	$-e_3$	$-e_4$
e_6	e_6	e_5	$-e_8$	e_7	e_2	e_1	e_4	$-e_3$
e_7	e_7	e_8	e_5	$-e_6$	e_3	$-e_4$	e_1	e_2
e_8	e_8	$-e_7$	e_6	e_5	e_4	e_3	$-e_2$	e_1

2.3 Jordan Algebras

The primary motivation for working with Jordan algebras in this thesis is to calculate the structure constants for the exceptional Lie algebra $\mathfrak{f}_4$.[3, Chapter 22] This section will discuss the method used for creating a Jordan algebra and how it can be used in Maple. The code which is used to create a Jordan algebra is given in appendix B.1.3. A tutorial worksheet is provided in appendix C.3, and illustrates some of the ways that a Jordan algebra can be used in Maple.

Definition 2.2. A **Jordan Algebra** is an algebra J over a field F whose multiplication satisfies the following properties:

1. $xy = yx$
2. $(xy)(xx) = x(y(xx))$

for every $x, y \in J$.

There are two important ways that a Jordan algebra can be created using another algebra A. First, given an associative algebra A, one can construct a Jordan algebra by introducing the **Jordan product**

$$x \circ y = \frac{xy + yx}{2}.$$

It is easy to verify that the Jordan product satisfies the conditions for a Jordan algebra given that multiplication in A is associative. Furthermore, if A is associative, then we know that matrix multiplication in $M_n(A)$ is also associative, making the algebra $M_n(A)$ an associative algebra. Therefore, we can create a Jordan algebra by using $n \times n$ matrices over A, together with the Jordan product. We will denote these algebras by $\mathbb{J}_n(A)$.

The other method is to take the set of self-adjoint 3×3 matrices over $\mathbb{O}$ together with the Jordan product as defined above. We will not show that this product satisfies the conditions for a Jordan algebra, but it is indeed remarkable that it does. This algebra, which we will denote by $\mathbb{J}_3(\mathbb{O})$, will be used later on to construct the exceptional Lie algebra $\mathfrak{f}_4$.

The Maple procedure, which is called by the command `AlgebraLibraryData`, to construct the product rules for a Jordan algebra, was created by me. As input values, it takes the size of the matrices used, and the algebra over which the matrices are created, namely "Real," "Complex," "Quaternions," or "Octonions."

Example:

Consider the 3×3 Hermitian matrices over the quaternions, $\mathbb{J}_3(\mathbb{H})$. Matrices in this algebra are of the form

$$\begin{pmatrix} x & \alpha & \beta \\ \overline{\alpha} & y & \gamma \\ \overline{\beta} & \overline{\gamma} & z \end{pmatrix}$$

where $x, y, z \in \mathbb{R}$ and $\alpha, \beta, \gamma \in \mathbb{H}$. Before we begin, we can calculate the dimension of this Jordan algebra, by counting the number of elements needed to make a basis. We have 3 real values on the diagonal. In each upper diagonal slot, we have four elements to choose from. Therefore, we see that $\dim(\mathbb{J}_3(\mathbb{H})) = 3 + 4 \cdot 3 = 15$.

Now we initialize the algebra that we wish to use, in this case the quaternions.

```
> QData:= AlgebraLibraryData("Quaternions",H):
> DGsetup(QData,[‘e‘,‘i‘,‘j‘,‘k‘],[omega]);
```

algebra name: H

The labels for the basis elements are $\{e, i, j, k\}$ where e is the multiplicative identity. Now we can use the command `JordanMatrices` to create a basis for the Jordan algebra. (Note: The commands `JordanMatrices` and `JordanProduct` were not developed by me, but were created to be stand alone commands and are based of off the procedure which

I did create which is used in `AlgebraLibraryData`.) The command `JordanMatrices` returns a set of basis elements for $\mathbb{J}_3(\mathbb{H})$, which are of the form

$$e_k E_i^j - \overline{e_k} E_j^i,$$

where $e_k \in A$ is a basis element of A, and $1 \leq i \leq j \leq n$.

```
> JM := JordanMatrices(3,H);
```

$$\left[\begin{bmatrix} e & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & e \end{bmatrix}, \begin{bmatrix} 0 & e & 0 \\ e & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & e \\ 0 & 0 & 0 \\ e & 0 & 0 \end{bmatrix},\right.$$
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & e \\ 0 & e & 0 \end{bmatrix}, \begin{bmatrix} 0 & i & 0 \\ -i & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & j & 0 \\ -j & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & k & 0 \\ -k & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & i \\ 0 & 0 & 0 \\ -i & 0 & 0 \end{bmatrix},$$
$$\left.\begin{bmatrix} 0 & 0 & j \\ 0 & 0 & 0 \\ -j & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & k \\ 0 & 0 & 0 \\ -k & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & i \\ 0 & -i & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & j \\ 0 & -j & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & k \\ 0 & -k & 0 \end{bmatrix}\right]$$

Now we can use the command `JordanProduct` to compute the Jordan product of two of these matrices.

```
> JordanProduct( JM[7] , JM[10] );
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{2}\mathtt{k} \\ 0 & \frac{1}{2}\mathtt{k} & 0 \end{bmatrix}$$

From this we see that $JM[7] \circ JM[10] = -\frac{1}{2} JM[15]$.

If we are interested in the product rules that the Jordan product creates with these matrices, but we are not interested in working with the matrices explicitly, we can simply use the `AlgebraLibraryData` command. This will compute the product rules behind the scenes and simply return the algebra data structure.

```
> JData := AlgebraLibraryData("Jordan(3,Quaternions)",J3H):
> DGsetup(JData);
```

algebra name: J3H

We can check that the product that we computed using the matrices is the same as the product returned from `AlgebraLibraryData`.

```
> evalDG( e7.e11 );
```

$$-\frac{1}{2}\mathtt{e15}$$

2.4 Clifford Algebras

The Clifford algebras have a variety of applications in geometry [4] and theoretical physics. Our motivation for creating a procedure for building Clifford algebras is for constructing special irreducible representations (called *Spin Representations*) of the special orthogonal Lie algebras [3]. Although this thesis will not deal with the creation of such representations, the ability to create and work with Clifford algebras is required for their explicit construction. The maple code for creating a Clifford algebra can be found in appendix B.1.4. A tutorial worksheet is also provided in appendix C.4 to show how a Clifford algebra is initialized and how it can be used.

We begin by discussing the method for constructing a Clifford algebra. After exploring this method a little, we can discuss how the program was written to compute a basis of the Clifford algebra. Examples will then be given to illustrate their use in Maple.

Let V be a vector space with a nondegenerate quadratic form Q. Q is said to have signature p, q if V admits a decomposition (not necessarily unique),$V = U \oplus W$, with $dim(U) = p$ and $dim(W) = q$, such that for all $u \in U$ and $w \in W$, $Q(u, w) = 0$, $Q(u, u) > 0$, and $Q(w, w) < 0$.

Let Q be a quadratic form over V with signature p, q. We can denote the Clifford algebra by $\mathrm{C}(V, Q)$, $\mathrm{C}(Q)$, or $Cl(p, q)$. The Clifford algebra $Cl(p, q)$ is an associative

algebra with unit 1, freely generated by V modulo the relations

$$v^2 = -Q(v)1 \tag{2.4}$$

for every $v \in V$, where 1 is the multiplicative identity of $C(Q)$. Equivalently, we can write equation (2.4) as

$$vw + wv = -2Q(v, w)1 \tag{2.5}$$

for all $v, w \in V$.

Let $\{e_i\}_{i=1}^n$ be an ordered basis of V. Using equation (2.5), any monomial, $e_{i_1}e_{i_2}...e_{i_k}$, in the free algebra generated by V, can be written as a sum of monomials $e_{j_1}...e_{j_\ell}$, where the indices j_m are increasing order and $\ell \leq k$. Thus,

$$\mathcal{B} = \{e_1 \cdots e_i e_j \cdots e_k \mid 1 \leq i < j \leq k \text{ and } 1 \leq k \leq n\},$$

together with the identity 1, is a basis of $C(Q)$.

Notice that for each value of k, there are $\binom{n}{k}$ basis elements. Therefore the dimension of the Clifford algebra is $\sum_{k=0}^{n} \binom{n}{k} = 2^n$.

The product rules for the basis elements are calculated by multiplying the basis elements together. Using the associative property of the algebra and equation (2.5), the product can be written as a sum of monomials whose indices are in increasing order.

This is concept is illustrated in the following example. Consider the element $e_1e_3e_2e_4$. This can be simplified to get

$$\begin{aligned} e_1e_3e_2e_4 &= e_1\Big(-2Q(e_2, e_3)1 - e_2e_3\Big)e_4 \\ &= -2Q(e_2, e_3) \cdot (e_1e_4) - (e_1e_2e_3e_4) \end{aligned} \tag{2.6}$$

By way of example, if $\dim(V) = 2$, then $\{1, e_1, e_2, e_1e_2\}$ is a basis of $C(V, Q)$. If $\dim(V) = 3$, then $\{1, e_1, e_2, e_3, e_1e_2, e_1e_3, e_2e_3, e_1e_2e_3\}$ is a basis of $C(V, Q)$.

Because $C(Q)$ is an associative algebra, if we have an element of $C(Q)$ which is composed of many basis elements of V, we can find the first index of an element that

is not in the correct order, apply equation (2.5) and use the distributive property as shown in equation (2.6). This procedure may be repeated until every element in the representation is written as a linear combination of the basis elements of $C(Q)$.

The procedures for creating the basis elements of $C(Q)$, computing their products and simplifying the results were created by me. They are accessed by the command `AlgebraLibraryData`. A recursive algorithm is used for simplifying a product of two basis elements. The algorithm searches through the product to find two adjacent elements which are not in order. Then process outlined by equation (2.6) is used and two new elements are created. Each of these are fed back into the procedure. When a given element is properly ordered, the procedure ends, and the object is passed back.

There are two procedures which can be used for simplifying the product of two elements. The first procedure handles the case where the given quadratic form is diagonal. In this instance, we need only consider the cases where $j > i$ or $j = i$ as follows. Supposing that $j > i$, $Q(e_i, e_j) = 0$ so

$$x(e_j e_i)y = x(-2Q(e_i, e_j)1 - e_i e_j)y = -x(e_i e_j)y.$$

Next suppose that $i = j$. Then we use equation (2.4) to get

$$x(e_i e_i)y = x(-Q(e_i)1)y = -xy.$$

In either case, new linear combinations of elements are not created. Because no new monomials are created, the algorithm continues to check if the elements in the monomial are sorted.

In the event that a given quadratic form is *not* diagonal, then we have no choice but to follow the general procedure, where reduction of a monomial is a sum of monomials, and each monomial in the sum must be checked and sorted. Thus, although the programs will work for any symmetric bilinear form, the algorithms are much faster for a diagonal form.

Example:

Let $V = \mathbb{R}^3$ and $Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & -1 \end{pmatrix}$. Letting $\{e_1, e_2, e_3\}$ be the standard basis of V, the basis of $C(Q)$ as defined above is

$$\begin{aligned} B &= \{\ 1\ ,\ e_1\ ,\ e_2\ ,\ e_3\ ,\ e_1e_2\ ,\ e_1e_3\ ,\ e_2e_3\ ,\ e_1e_2e_3\ \} \\ &= \{1, e_1, e_2, e_3, e_{12}, e_{13}, e_{23}, e_{123}\} \end{aligned}$$

where 1 acts as the multiplicative identity in $C(Q)$, namely $v1 = 1v = v$, for all $v \in C(Q)$. Notice that the cumbersome notation has been modified in the second row so that the indices still indicate the order and number of the elements used to create the basis element in $C(Q)$.

Now consider the basis elements e_3 and e_{123}. Let's compute the product of these elements $(e_3)(e_1e_2e_3) \in C(Q)$. This element can be represented using the basis elements as follows

$$\begin{aligned} e_3 \cdot (e_1e_2e_3) &= (e_3e_1)e_2e_3 \\ &= (-2Q(e_1, e_3) - e_1e_3)\, e_2e_3 \\ &= -e_1(e_3e_2)e_3 \\ &= -e_1(-2Q(e_2, e_3) - e_2e_3)e_3 \\ &= -2(-\frac{1}{2}e_1e_3) + e_1e_2(e_3e_3) \\ &= e_1e_3 + e_1e_2(-Q(e_3, e_3)) \\ &= e_1e_3 + e_1e_2 \\ &= e_{12} + e_{13} \end{aligned}$$

Therefore, we see that $e_3 \cdot e_{123} = e_{12} + e_{13}$.

Using the command `AlgebraLibraryData`, we can create the algebra $C(Q)$ using the quadratic form above.

```
> Q := <<1,0,0>|<0,0,1/2>|<0,1/2,-1>>:
> CData := AlgebraLibraryData("Clifford(3)",quadraticform=Q,C3):
```

When we initialize this algebra, we are going to want to label the basis vectors so that they match the notational convention we have chosen.

```
> DGsetup(CData,[‘e‘,‘e1‘,‘e2‘,‘e3‘,‘e12‘,‘e13‘,‘e23‘,‘e123‘],[omega]);
```

algebra name: C3

Now we can check the calculation we did by hand against the result given by Maple,

```
> evalDG(e3.e123);
```

```
e12+e13
```

This procedure allows us to construct a multiplication table for the basis elements of $C(Q)$ as given in table 2.5.

TABLE 2.5: Multiplication Table - Basis Elements for the Clifford Algebra $C(Q)$

$\cdot$	1	e_1	e_2	e_3	e_{12}	e_{13}	e_{23}	e_{123}
1	1	e_1	e_2	e_3	e_{12}	e_{13}	e_{23}	e_{123}
e_1	e_1	-1	e_{12}	e_{13}	$-e_2$	$-e_3$	e_{123}	$-e_{23}$
e_2	e_2	$-e_{12}$	0	e_{23}	0	$-e_{123}$	0	0
e_3	e_3	$-e_{13}$	$-1-e_{23}$	1	e_1+e_{123}	$-e_1$	$-e_2-e_3$	$e_{12}+e_{13}$
e_{12}	e_{12}	e_2	0	e_{123}	0	e_{23}	0	0
e_{13}	e_{13}	e_3	$-e_1-e_{123}$	e_1	$-1-e_{23}$	1	$-e_{12}-e_{13}$	$-e_2-e_3$
e_{23}	e_{23}	e_{123}	$-e_2$	e_2	$-e_{12}$	e_{12}	$-e_{23}$	$-e_{123}$
e_{123}	e_{123}	$-e_{23}$	$-e_{12}$	e_{12}	e_2	$-e_2$	$-e_{123}$	e_{23}

Example:

In this example, we will demonstrate how the quaternions, $\mathbb{H}$, can also be constructed as a Clifford algebra. Let $V = \mathbb{R}^2$ and let Q be the inner product on the standard basis elements $e_1 = (1, 0)$ and $e_2 = (0, 1)$. Then we can create the product rules for the Clifford algebra, $C(I_2)$. In fact, the default quadratic form that is used (in the event that one is not given by the user) is the identity.

```
> CD := AlgebraLibraryData("Clifford(2)",H):
> DGsetup(CD);
```

algebra name: H

Now, we can use the command `MultiplicationTable` to print the product table of the basis elements.

```
> MultiplicationTable(H);
```

		e1	e2	e3	e4
e1	|	e1	e2	e3	e4
e2	|	e2	-e1	e4	-e3
e3	|	e3	-e4	-e1	e2
e4	|	e4	e3	-e2	-e1

Comparing this result with table 2.1, it is easy to see that we have constructed, exactly, the multiplication table for the quaternions. Therefore, we see that $C(I_2) = \mathbb{H}$.

2.5 Lie Algebras

This section will set the stage for the rest of this thesis. We will discuss Lie algebras and some of the basic theory and properties. Many libraries and functions already exist in Maple to work with Lie algebras. We will provide some simple examples to illustrate the properties that will be the most relevant to this thesis. We begin by defining a Lie algebra.

Definition 2.3. A **Lie Algebra**, $\mathfrak{g}$, is a vector space over a field F together with a binary operation,

$$[\ ,\ ] : \mathfrak{g} \times \mathfrak{g} \to \mathfrak{g}$$

satisfying the following properties for every $X, Y, Z \in \mathfrak{g}$ and $a, b \in F$:

- Bilinear: $[aX + bY, Z] = a[X, Z] + b[Y, Z]$ and $[X, aY + bZ] = a[X, Y] + b[X, Z]$
- Skew-symmetric: $[X, Y] = -[Y, X]$
- Jacobi identity: $[X, [Y, Z]] + [Y, [Z, X]] + [Z, [X, Y]] = 0$

To gain some familiarity with this definition, let's look at an example.

Example:

Let V be a finite dimensional vector space over a field F. Let $\mathrm{end}(V)$ denote the set of all endomorphisms, or in other words, all linear maps $f : V \to V$. Let $[f, g] = f \circ g - g \circ f$ for all $f, g \in \mathrm{end}(V)$.

We will show that $\mathrm{end}(V)$ is a Lie algebra, by using the commutator as the bracket, by satisfying the three properties given in the definition. Let $a, b \in F$ and $f, g, h \in \mathrm{end}(V)$ be given. Then, for any $v \in V$, consider the following property:

$$\begin{aligned}\Big(f \circ (g \circ h)\Big)(v) &= f(g(h(v))) \\ &= \Big((f \circ g) \circ h\Big)(v)\end{aligned}$$

This means the composition of endomorphisms is associative, $f \circ (g \circ h) = (f \circ g) \circ h$. Now we can consider the properties of the bracket. First we show that the bracket is

linear in the first argument.

$$\begin{aligned}
[af + bg, h] &= (af + bg) \circ h - h \circ (af + bg) \\
&= (af) \circ h + (bg \circ h) - h \circ (af) - h \circ (bg) \\
&= a(f \circ h) - a(h \circ f) + b(g \circ h) - b(h \circ g) \\
&= a[f, h] + b[g, h].
\end{aligned}$$

Showing that the bracket is linear in the second argument is similar.
We also have that the bracket is skew-symmetric.

$$\begin{aligned}
[f, g] &= f \circ g - g \circ f \\
&= -(g \circ f - f \circ g) \\
&= -[g, f]
\end{aligned}$$

Finally, we need to show that the bracket satisfies the Jacobi property.

$$\begin{aligned}
[f, [g, h]] + [g, [h, f]] + [h, [f, g]] &= [f, g \circ h - h \circ g] + [g, h \circ f - f \circ h] + [h, f \circ g - g \circ f] \\
&= f \circ (g \circ h) - (g \circ h) \circ f - f \circ (h \circ g) + (h \circ g) \circ f + \\
&\quad g \circ (h \circ f) - (h \circ f) \circ g - g \circ (f \circ h) + (f \circ h) \circ g + \\
&\quad h \circ (f \circ g) - (f \circ g) \circ h - h \circ (g \circ f) + (g \circ f) \circ h \\
&= \Big(f \circ (g \circ h) - (f \circ g) \circ h\Big) + \Big((h \circ g) \circ f - h \circ (g \circ f)\Big) + \\
&\quad \Big(g \circ (h \circ f) - (g \circ h) \circ f\Big) + \Big((f \circ h) \circ g - f \circ (h \circ g)\Big) + \\
&\quad \Big(h \circ (f \circ g) - (h \circ f) \circ g\Big) + \Big((g \circ f) \circ h - g \circ (f \circ h)\Big) \\
&= 0
\end{aligned}$$

Therefore, the set of endomorphisms on a vector space over a field forms a Lie algebra.

Another way to look at this example is, by picking a basis of V, to look at the matrices representing the endomorphisms. Then, this means that the set of all $n \times n$ matrices together with the commutator, forms a Lie algebra. In the case where the field $F = \mathbb{C}$, we will denote this Lie algebra by $\mathfrak{gl}_n(\mathbb{C})$.

Example:

Let A be an algebra over a field F. A **derivation** of A is a linear map $D : A \to A$ such that

$$D(x \cdot y) = D(x) \cdot y + x \cdot D(y) \quad \text{for all} \quad x, y \in A. \tag{2.7}$$

Let $\mathfrak{der}(A)$ be the set of all derivations on A. From definition 2.1 we know that A is a vector space together with a multiplication operation. Therefore, notice that if $D \in \mathfrak{der}(A)$ then $D \in \text{end}(A)$ with A as a vector space. Therefore, $\mathfrak{der}\,(A) \subset \text{end}(A)$.

Let $f, g \in \mathfrak{der}\,(A)$ and $x, y \in A$. Then

$$\begin{aligned}
(f \circ g)(x \cdot y) &= f(g(x \cdot y)) \\
&= f(x \cdot g(y) + g(x) \cdot y) \\
&= f(x \cdot g(y)) + f(g(x) \cdot y) \\
&= x \cdot f(g(y)) + f(x) \cdot g(y) + f(g(x)) \cdot y + g(x) \cdot f(y) \\
&= x \cdot (f \circ g)(y) + (f \circ g)(x) \cdot y + f(x) \cdot g(y) + g(x) \cdot f(y).
\end{aligned}$$

From these calculations, we see that the set of derivations is not closed under composition alone. However, letting $[f, g] = f \circ g - g \circ f$, consider the following:

$$\begin{aligned}
[f, g](x \cdot y) &= (f \circ g)(x \cdot y) - (g \circ f)(x \cdot y) \\
&= x \cdot (f \circ g)(y) + (f \circ g)(x) \cdot y + f(x) \cdot g(y) + g(x) \cdot f(y) \\
&\quad - x \cdot (g \circ f)(y) - (g \circ f)(x) \cdot y - g(x) \cdot f(y) - f(x) \cdot g(y) \\
&= x \cdot \Big((f \circ g)(y) - (g \circ f)(y)\Big) + \Big((f \circ g)(x) - (g \circ f)(x)\Big) \cdot y \\
&= x \cdot [f, g](y) + [f, g](x) \cdot y.
\end{aligned}$$

Therefore, if $f, g \in \mathfrak{der}\,(A)$ then $[f, g] \in \mathfrak{der}\,(A)$. Furthermore, because $\mathfrak{der}\,(A) \subset \text{end}(A)$, we know that the commutator is bilinear, skew-symmetric, and satisfies the Jacobi property. Therefore, $\mathfrak{der}\,(A)$ together with the commutator is a Lie algebra.

We will now address some of the definitions and properties of Lie algebras that are used in the classification of Lie algebras. A Lie algebra is called **abelian** if $[X, Y] = 0$ for all $X, Y \in \mathfrak{g}$. We say that a Lie subalgebra $\mathfrak{h} \subset \mathfrak{g}$ of a Lie algebra $\mathfrak{g}$ is an **ideal** if $[X, Y] \in \mathfrak{h}$ for all $X \in \mathfrak{h}$ and $Y \in \mathfrak{g}$. A Lie algebra $\mathfrak{g}$ is called **simple** if $\dim \mathfrak{g} > 1$ and it contains no nontrivial ideals [3].

Let $[\mathfrak{g}, \mathfrak{g}] = \text{span}\{[x, y] \mid x, y \in \mathfrak{g}\}$ as a vector space. Since, $[u, v] \in [\mathfrak{g}, \mathfrak{g}]$ for every $u, v \in [\mathfrak{g}, \mathfrak{g}]$ by definition, we see that $[\mathfrak{g}, \mathfrak{g}]$ is a subalgebra of $\mathfrak{g}$. This subalgebra is called the **derived Lie algebra of** $\mathfrak{g}$, and is denoted by $\mathscr{D}\mathfrak{g}$. By bilinearity of the Lie bracket, it is sufficient to consider the span of the bracket of the basis elements of $\mathfrak{g}$. Thus, to compute a basis of $\mathscr{D}\mathfrak{g}$, we look at the linearly independent set formed from the brackets of the basis elements of $\mathfrak{g}$.

The derived algebra motivates the definitions of two descending chains of subalgebras. First, the **lower central series** of subalgebras $\mathscr{D}_k\mathfrak{g}$ is defined inductively by

$$\mathscr{D}_1\mathfrak{g} = [\mathfrak{g}, \mathfrak{g}] \qquad \text{and} \qquad \mathscr{D}_k\mathfrak{g} = [\mathfrak{g}, \mathscr{D}_{k-1}\mathfrak{g}]. \tag{2.8}$$

The other series, the **derived series**, denoted by $\mathscr{D}^k\mathfrak{g}$, is defined inductively by

$$\mathscr{D}^1\mathfrak{g} = [\mathfrak{g}, \mathfrak{g}] \qquad \text{and} \qquad \mathscr{D}^k\mathfrak{g} = [\mathscr{D}^{k-1}\mathfrak{g}, \mathscr{D}^{k-1}\mathfrak{g}]. \tag{2.9}$$

With these series, we can define the following:

1. We say that $\mathfrak{g}$ is **solvable** if $\mathscr{D}^k\mathfrak{g} = 0$ for some k.
2. We say that $\mathfrak{g}$ is **nilpotent** if $\mathscr{D}_k\mathfrak{g} = 0$ for some k.
3. We say that $\mathfrak{g}$ is **semisimple** if $\mathfrak{g}$ has no nontrivial solvable ideals. Equivalently, a direct sum of simple Lie algebras is semisimple.

Example:

We will now give a simple example of a solvable Lie algebra. To begin, the set of all real 3×3 upper triangular matrices, together with the commutator, is a Lie algebra which we will denote by $\mathfrak{t}_3$. Let $\{E_1^1, E_2^2, E_3^3, E_1^2, E_2^3, E_1^3\}$ be a basis of $\mathfrak{t}_3$. We can obtain a table of the bracket values of the basis elements as given below.

TABLE 2.6: Lie Bracket Table for $\mathfrak{t}_3$

$[\ ,\ ]$	E_1^1	E_1^2	E_1^3	E_2^2	E_2^3	E_3^3
E_1^1	0	E_1^2	E_1^3	0	0	0
E_1^2	$-E_1^2$	0	0	E_1^2	E_1^3	0
E_1^3	$-E_1^3$	0	0	0	0	E_1^3
E_2^2	0	$-E_1^2$	0	0	E_2^3	0
E_2^3	0	$-E_1^3$	0	$-E_2^3$	0	E_2^3
E_3^3	0	0	$-E_1^3$	0	$-E_2^3$	0

By looking at this table, we see that $\text{span}\{[E_i^j, E_h^k]\} = \text{span}\{E_1^2, E_1^3, E_2^3\} = \mathscr{D}\mathfrak{t}_3$.

To see what $\mathscr{D}^2\mathfrak{t}_3 = [\mathscr{D}\mathfrak{t}_3, \mathscr{D}\mathfrak{t}_3]$ will look like, let's draw up another table.

TABLE 2.7: Lie Bracket Table for $[\mathscr{D}\mathfrak{t}_3, \mathscr{D}\mathfrak{t}_3]$

$[\ ,\ ]$	E_1^2	E_1^3	E_2^3
E_1^2	0	0	E_1^3
E_1^3	0	0	0
E_1^2	$-E_1^3$	0	0

From this, we see that $\mathscr{D}^2\mathfrak{t}_3$ is generated by $\{E_1^3\}$. This means that $\mathscr{D}^3\mathfrak{t}_3 = 0$ because $[E_1^3, E_1^3] = 0$. Therefore, we see that the derived series for $\mathfrak{t}_3$ goes to zero, making the Lie algebra solvable.

Example:

Next we will give an example of a nilpotent Lie algebra. Consider the Lie algebra of all 4×4 strictly upper triangular matrices, call it $\mathfrak{b}_4$. Let $\{E_1^2, E_1^3, E_1^4, E_2^3, E_2^4, E_3^4\}$ be a basis of $\mathfrak{b}_4$. Similar to the previous example, we can identify the derived algebra $\mathscr{D}\mathfrak{b}_4$ by looking at a table of brackets for the basis elements.

TABLE 2.8: Lie Bracket Table for $\mathfrak{b}_4$

$[\ ,\ ]$	E_1^2	E_1^3	E_1^4	E_2^3	E_2^4	E_3^4
E_1^2	0	0	0	E_1^3	E_1^4	0
E_1^3	0	0	0	0	0	E_1^4
E_1^4	0	0	0	0	0	0
E_2^3	$-E_1^3$	0	0	0	0	E_2^4
E_2^4	$-E_1^4$	0	0	0	0	0
E_3^4	0	$-E_1^4$	0	$-E_2^4$	0	0

From this table, we see that $\mathscr{D}\mathfrak{b}_4$ is generated by $\{E_1^3, E_1^4, E_2^4\}$. Now we want to compute $\mathscr{D}_2\mathfrak{b}_4$. We can construct another table of Lie brackets, where the rows come from $\mathfrak{b}_4$ and the columns are $\mathscr{D}\mathfrak{b}_4$. Therefore, we see that $\mathscr{D}_2\mathfrak{b}_4$ is generated by the single

TABLE 2.9: Lie Bracket Table for $[\mathfrak{b}_4, \mathscr{D}\mathfrak{b}_4]$

$[\ ,\ ]$	E_1^3	E_1^4	E_2^4
E_1^2	0	0	E_1^4
E_1^3	0	0	0
E_1^4	0	0	0
E_2^3	0	0	0
E_2^4	0	0	0
E_3^4	$-E_1^4$	0	0

element $\{E_1^4\}$. Because dim $\mathscr{D}_2\mathfrak{b}_4 = 1$, this means that $\mathscr{D}_3\mathfrak{b}_4 = 0$. Therefore, because the lower central series for $\mathfrak{b}_4$ becomes zero, we conclude that $\mathfrak{b}_4$ is nilpotent.

A very important tool in the study of Lie algebras is a subalgebra called a **Cartan subalgebra**. We define it below and provide an example to illustrate to calculating one.

Definition 2.4. Let $\mathfrak{h}$ be a subalgebra of $\mathfrak{g}$. The **normalizer** of $\mathfrak{h}$ is the subalgebra $N(\mathfrak{h}) = \{x \in \mathfrak{g} \mid [x, y] \in \mathfrak{h}\ \forall y \in \mathfrak{h}\}$.

Definition 2.5. A **Cartan subalgebra** $\mathfrak{h}$ is a nilpotent Lie subalgebra of $\mathfrak{g}$ which is self-normalizing, meaning $N(\mathfrak{h}) = \mathfrak{h}$.

Definition 2.6. Let $\mathfrak{h}$ be a Cartan subalgebra of $\mathfrak{g}$. Then the **rank** of $\mathfrak{g}$ is $\dim(\mathfrak{h})$.

Example:

In this example, we will compute a Cartan subalgebra of $\mathfrak{t}_3$. Let $\mathcal{B}$ be the basis of $\mathfrak{t}_3$ described in the example on solvable algebras. Notice that $[E_i^i, E_j^j] = 0$ for all i, j. Therefore, the set of diagonal matrices form an abelian subalgebra of $\mathfrak{t}_3$. Denote this subalgebra by $\mathfrak{h}$. Because this subalgebra is abelian, it is clearly nilpotent as well. Now, let's calculate the normalizer of $\mathfrak{h}$.

$$\begin{aligned} N(\mathfrak{h}) &= \left\{x \in \mathfrak{t}_3 \mid [y, x] = 0\ \forall y \in \mathfrak{h}\right\} \\ &= \left\{x \in \mathfrak{t}_3 \mid [E_i^i, x] = 0\right\} \\ &= \left\{x \in \mathfrak{t}_3 \mid x = \sum_{i=1}^{3} a_i E_i^i\right\} \\ &= \mathfrak{h} \end{aligned}$$

Therefore, we have that $\mathfrak{h}$ is a nilpotent subalgebra of $\mathfrak{t}_3$ which is self-normalizing. Therefore, the Lie subalgebra of diagonal matrices is a Cartan subalgebra of $\mathfrak{t}_3$.

Another way to think about a Cartan subalgebra is a subalgebra $\mathfrak{h} \subset \mathfrak{g}$ such that $\mathfrak{h}$ is maximal (in sense of the series) among abelian diagonalizable subalgebras. The Cartan subalgebra of a complex semisimple Lie algebra is unique up to conjugacy. This uniqueness allows us to use the Cartan subalgebra identify and classify the complex semisimple Lie algebras. This classification will be discussed shortly.

Next we introduce the **adjoint action** of an element on the Lie algebra. Given an element $x \in \mathfrak{g}$, we define the adjoint action of x on $\mathfrak{g}$ as the map $\mathrm{ad}_x : \mathfrak{g} \to \mathfrak{g}$ given by $\mathrm{ad}_x(y) = [x, y]$ for all $y \in \mathfrak{g}$. Since the bracket is a bilinear operation, this means that ad_x is a linear mapping. Therefore, by picking a basis of the Lie algebra, we will be able to represent this linear map using matrices.

Example:

Consider the set of trace-free 2×2 matrices over $\mathbb{C}$. Notice that the commutator of any two trace-free matrices is also trace-free. Therefore, this subalgebra of $\mathfrak{gl}_2(\mathbb{C})$ is also a Lie algebra, and will be denoted $\mathfrak{sl}_2(\mathbb{C})$. A basis for this Lie algebra is given by the following matrices

$$H = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \qquad X = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \qquad Y = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}. \tag{2.10}$$

By bilinearity of the Lie bracket, it suffices to identify the Lie bracket on the basis $\{H, X, Y\}$. Using matrix multiplication, we find that $[H, X] = 2X$, $[H, Y] = -2Y$, and $[X, Y] = H$.

Now, we are ready to compute the adjoint mapping. For example, let's calculate ad_X. To do so, notice that

$$\begin{aligned} \mathrm{ad}_X(X) &= [X, X] = 0H + 0X + 0Y, \\ \mathrm{ad}_X(Y) &= [X, Y] = H + 0X + 0Y, \\ \mathrm{ad}_X(H) &= [X, H] = 0H - 2X + 0Y. \end{aligned}$$

Therefore, using the basis $\{H, X, Y\}$, we can represent the mapping ad_X with the matrix

$$\mathrm{ad}_X = \begin{pmatrix} 0 & 0 & 1 \\ -2 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Similar calculations give us the following matrices

$$\mathrm{ad}_Y = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 0 \end{pmatrix}, \qquad \mathrm{ad}_H = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -2 \end{pmatrix}.$$

Now that we have the adjoint representation at our disposal, we are ready to introduce an important tool in studying Lie algebras, namely the Killing form.

Definition 2.7. For any two elements $x, y \in \mathfrak{g}$, the **Killing form** is a symmetric bilinear map, $B : \mathfrak{g} \times \mathfrak{g} \to \mathbb{C}$, given by taking the trace of the composition of the adjoint representation of x and y, namely

$$B(x, y) = Tr(\mathrm{ad}_x \circ \mathrm{ad}_y).$$

Given a basis of a Lie algebra, we can write down a formula for computing the elements of the Killing form with respect to that basis. Let $\{e_i\}$ be a basis of $\mathfrak{g}$ and $\{\varepsilon^i\}$ the set of dual vectors. Next, we compute the structure constants for the Lie algebra, $[e_i, e_j] = c_{ij}^k e_k$. Then, for every $e_i, e_j, e_k \in \mathfrak{g}$,

$$\begin{aligned} \left(\mathrm{ad}_{e_i} \circ \mathrm{ad}_{e_j}\right)(e_k) &= [e_i, [e_j, e_k]] \\ &= [e_i, c_{jk}^{\ell} e_\ell] \\ &= c_{jk}^{\ell} [e_i, e_\ell] \\ &= c_{jk}^{\ell} c_{i\ell}^{m} e_m. \end{aligned}$$

The trace of the composition of these maps can then be computed as follows

$$\begin{aligned} Tr(\mathrm{ad}_{e_i} \circ \mathrm{ad}_{e_j}) &= \varepsilon^k \left(c_{jk}^{\ell} c_{i\ell}^{m} e_m \right) \\ &= c_{jk}^{\ell} c_{i\ell}^{m} \varepsilon^k(e_m) \\ &= c_{jk}^{\ell} c_{i\ell}^{m} \delta_m^k \\ &= c_{jk}^{\ell} c_{i\ell}^{k} \end{aligned}$$

Therefore, we have that $B(e_i, e_j) = c_{i\ell}^{k} c_{jk}^{\ell}$.

The Killing form is often used as the inner product on $\mathfrak{g}$. More on the Killing form will be said in the next chapter. For now, the Killing form is useful in determining if a Lie algebra is semisimple. **Cartan's Criterion** states that a Lie algebra $\mathfrak{g}$ is semisimple if and only if the Killing form on $\mathfrak{g}$ is nondegenerate.

Example:

Consider the set of 2×2 skew hermitian matrices as a real vector space, denoted by $\mathfrak{su}_2$. It is easily verified that this set of matrices make a Lie algebra, using the commutator as the Lie bracket. A basis for this set of matrices is

$$\left\{ e_1 = \begin{pmatrix} -i & 0 \\ 0 & i \end{pmatrix}, e_2 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, e_3 = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix} \right\}.$$

The bracket equations for the basis elements are, $[e_1, e_2] = 2e_3$, $[e_1, e_3] = -2e_2$, and $[e_2, e_3] = 2e_1$. Using this basis, we represent the adjoint mappings of these basis elements using the following matrices:

$$\mathrm{ad}_{e_1} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -2 \\ 2 & 0 & 0 \end{pmatrix}, \quad \mathrm{ad}_{e_2} = \begin{pmatrix} 0 & 0 & 2 \\ 0 & 0 & 0 \\ -2 & 0 & 0 \end{pmatrix}, \quad \mathrm{ad}_{e_3} = \begin{pmatrix} 0 & -2 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

To compute a matrix representation of the Killing form, it suffices to compute the values of the Killing form on the basis elements of $\mathfrak{su}_2$. Doing so gives us

$$B(e_1,e_1) = Tr\left(\left(\begin{pmatrix} 0 & 0 & 0 \\ 0 & -4 & 0 \\ 0 & 0 & -4 \end{pmatrix}\right)\right) = -8,$$

$$B(e_1,e_2) = Tr\left(\left(\begin{pmatrix} 0 & 0 & 0 \\ 4 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}\right)\right) = 0,$$

$$B(e_1,e_3) = Tr\left(\left(\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 4 & 0 & 0 \end{pmatrix}\right)\right) = 0,$$

$$B(e_2,e_2) = Tr\left(\left(\begin{pmatrix} -4 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -4 \end{pmatrix}\right)\right) = -8,$$

$$B(e_2,e_3) = Tr\left(\left(\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 4 & 0 \end{pmatrix}\right)\right) = 0,$$

$$B(e_3,e_3) = Tr\left(\left(\begin{pmatrix} -4 & 0 & 0 \\ 0 & -4 & 0 \\ 0 & 0 & 0 \end{pmatrix}\right)\right) = -8.$$

Therefore, by symmetry of the Killing form, we see that the matrix representation of B in the basis $\{e_1, e_2, e_3\}$ is

$$B = \begin{pmatrix} -8 & 0 & 0 \\ 0 & -8 & 0 \\ 0 & 0 & -8 \end{pmatrix}.$$

Because $B = -8I_3$, we see that the Killing form is nondegenerate, making $\mathfrak{su}_2$ semisimple by Cartan's Criterion.

It can be shown that the sum of two solvable ideals in $\mathfrak{g}$ is solvable. The largest solvable ideal of a Lie algebra, $\mathfrak{g}$, is called the **radical of** $\mathfrak{g}$ and is denoted $\mathrm{Rad}(\mathfrak{g})$ [3].

This means that the subalgebra $\mathfrak{g}/\text{Rad}(\mathfrak{g})$ is semisimple. In 1905, Eugenio Elia Levi proved that every finite-dimensional real Lie algebra is the semidirect product of its radical and a semisimple subalgebra [6]. Levi Decomposition shows us that the study of finite-dimensional Lie algebras can be reduced to the study of solvable and semisimple Lie algebras.

Example:

Consider the following set of linear operators

$$\left\{T_x = \frac{\partial}{\partial x}, T_y = \frac{\partial}{\partial y}, T_z = \frac{\partial}{\partial z}, R_{xy} = xT_y - yT_x, R_{xz} = xT_z - zT_x, R_{yz} = yT_z - zT_y\right\}$$

These operators form a basis of a Lie algebra where the Lie bracket is the commutator. This Lie algebra is called the Euclidean Lie algebra, denoted by $e(3)$, and is the algebra of infinitesimal rotations and translations in $\mathbb{R}^3$. The multiplication table for these basis elements of $e(3)$ is

TABLE 2.10: Lie Bracket Table for $e(3)$

	T_x	T_y	T_z	R_{xy}	R_{xz}	R_{yz}
T_x	0	0	0	T_y	T_z	0
T_y	0	0	0	$-T_x$	0	T_z
T_z	0	0	0	0	$-T_x$	$-T_y$
R_{xy}	$-T_y$	T_x	0	0	$-R_{yz}$	R_{xz}
R_{xz}	$-T_z$	0	T_x	R_{yz}	0	$-R_{xy}$
R_{yz}	0	$-T_z$	T_y	$-R_{xz}$	R_{xy}	0

By looking at this table, we see that the elements $\{T_x, T_y, T_z\}$ form a basis for a subalgebra, call it $R \subset e(3)$. Furthermore, we see that this subalgebra R is abelian, making is a solvable subalgebra. We also see that the bracket of any element of R with any element from $e(3)$ is an element in R, thus making R a solvable ideal of $e(3)$.

To see that this is the largest solvable ideal in $e(3)$, note that if a larger solvable ideal of $e(3)$, then it must contain one of the elements R_{xy}, R_{xz} or R_{yz}. However, any ideal containing one of these elements must contain them all, making the ideal the whole Lie algebra. It is straightforward to check that the Lie algebra $e(3)$ is not solvable. Therefore, R is the largest solvable ideal in $e(3)$, making $\text{Rad}(e(3)) = R$.

Then the algebra $e(3)/R$ is simply the algebra where the basis elements are the cosets $\{R_{xy}+R, R_{xz}+R, R_{yz}+R\}$. Let $\mathfrak{s}$ be the subalgebra of $e(3)$ with basis elements $\{R_{xy}, R_{xz}, R_{yz}\}$. There is a canonical isomorphism $e(3)/R \to \mathfrak{s}$ defined by $x+R \mapsto x$.

To show that the subalgebra $\mathfrak{s}$ is semisimple, we can compute the Killing form on $\mathfrak{s}$ and show that it is nondegenerate. For simplicity in calculation, let $e_1 = R_{xy}$, $e_2 = R_{xz}$ and $e_3 = R_{yz}$, and note that the structure constants for $\mathfrak{s}$ are $c_{12}^k = -\delta_3^k$, $c_{13}^k = \delta_2^k$ and $c_{23}^k = -\delta_1^k$. Therefore, calculating the elements of the Killing b_{ij}, give us $b_{ij} = -2\delta_{ij}$, making $B = -2I_3$. Since the Killing form is nondegenerate, we conclude that the subalgebra $\mathfrak{s}$ is semisimple. Therefore, we have that $e(3)$ decomposes into a semidirect product of R and $\mathfrak{s}$.

For the remainder of this thesis, we will be interested in working with semisimple Lie algebras.

The semisimple Lie algebras over the complex numbers $\mathbb{C}$ have been completely classified and are organized in table 2.11. The table includes the name of each algebra, the rank and the dimension of the algebra.

TABLE 2.11: Classification of semisimple Lie algebras over $\mathbb{C}$

Class	Rank	Dim	Name
$A_n = \mathfrak{sl}_{n+1}$ $(n>1)$	n	$n(n+2)$	Special Linear
$B_n = \mathfrak{so}_{2n+1}$ $(n>2)$	n	$n(2n+1)$	Special Orthogonal
$C_n = \mathfrak{sp}_{2n}$ $(n>3)$	n	$n(2n+1)$	Symplectic
$D_n = \mathfrak{so}_{2n}$ $(n>4)$	n	$n(2n-1)$	Special Orthogonal
$\mathfrak{g}_2$	2	14	Exceptional
$\mathfrak{f}_4$	4	52	
$\mathfrak{e}_6$	6	78	
$\mathfrak{e}_7$	7	133	
$\mathfrak{e}_8$	8	248	

The first four classes of semisimple Lie algebras have very natural matrix constructions. The class A_n is the space of $(n+1)\times(n+1)$ trace-free matrices. Therefore,

$$A_n = \{X \in M_{n+1}(\mathbb{C}) |\ \mathrm{Tr}(X) = 0\}.$$

The class B_n is the space of $(2n+1)\times(2n+1)$ matrices which are skew-symmetric with respect to the form $Q = \left(\begin{array}{c|c|c} 0 & I_n & 0 \\ \hline I_n & 0 & 0 \\ \hline 0 & 0 & 1 \end{array}\right)$. Therefore,

$$B_n = \left\{X \in M_{2n+1}(\mathbb{C}) | X^t Q + QX = 0\right\}.$$

The class C_n is the space of $2n\times 2n$ matrices which are skew-symmetric with respect to the form $Q = \begin{pmatrix} 0 & I_n \\ -I_n & 0 \end{pmatrix}$. Therefore,

$$C_n = \{X \in M_{2n}(\mathbb{C}) \mid X^t Q + QX = 0\}.$$

The class D_n is the space of $2n \times 2n$ matrices which are skew-symmetric with respect to the form $Q = \left(\begin{array}{c|c} 0 & I_n \\ \hline I_n & 0 \end{array}\right)$. Therefore,

$$D_n = \{X \in M_{2n}(\mathbb{C}) \mid X^t Q + QX = 0\}.$$

The exceptional Lie algebras $\mathfrak{g}_2,\ \mathfrak{f}_4,\ \mathfrak{e}_6,\ \mathfrak{e}_7$ and $\mathfrak{e}_8$ are semisimple Lie algebras that do not have such simple and natural matrix representations like the previous four classes. They are called exceptional because they are not part of any family of semisimple Lie algebras. The subject of chapter 4 is a computational implementation to construct each of the exceptional Lie algebras using a uniform approach.

Example:

The Maple programs for working with Lie algebras already exist in the Differential Geometry package and were not developed by me. The commands used in the following example are included to illustrate some of the calculations which are possible in Maple.

In this example, we will show how to create the Lie algebra $e(3)$ using matrices. We will then compute the radical, the semisimple subalgebra and several Cartan subalgebras. The euclidean Lie algebra $e(3)$ is the set of 4×4 matrices of the form $\begin{pmatrix} A & b \\ 0 & 0 \end{pmatrix}$ where A is a 3×3 skew-symmetric matrix. Therefore, we can define a basis of the Lie algebra as follows.

```
> Basis := [Matrix(4,4,{(1,4)=1}),
            Matrix(4,4,{(2,4)=1}),
            Matrix(4,4,{(3,4)=1}),
            Matrix(4,4,{(1,2)=1,(2,1)=-1}),
            Matrix(4,4,{(1,3)=1,(3,1)=-1}),
            Matrix(4,4,{(2,3)=1,(3,2)=-1})]:
```

Next, we use the command `LieAlgebraData` to create the structure constants for the Lie algebra formed by these matrices. Give the frame for the Lie algebra the name `E3`.

```
> LD := LieAlgebraData( Basis, E3);
```

```
LD:=[[e1,e4]=e2,[e1,e5]=e3,[e2,e4]=-e1,[e2,e6]=e3,
[e3,e5]=-e1,[e3,e6]=-e2,[e4,e5]=-e6,
[e4,e6]=e5,[e5,e6]=-e4]
```

Using these structure constants, we can initialize the algebra.

```
> DGsetup(LD);
```

```
Lie algebra:  E3
```

The basis elements in our Lie algebra are {`e1,e2,e3,e4,e5,e6`}. To compare this basis with the example we worked with previously, we have `e1` $= T_x$, `e2` $= T_y$, `e3` $= T_z$, `e4` $= R_{xy}$, `e5` $= R_{xz}$ and `e6` $= R_{yz}$.

Now that we have an initialized Lie algebra, we have a wide variety of functions and programs at our disposal. To begin, let's compute the radical of $e(3)$.

```
> Rad := Radical(E3);
```

```
Rad:=[e3,e2,e1]
```

We see here that the radical we computed previously matches with the Radical computed by Maple.

Next, we can use the command `QuotientAlgebra` to compute the quotient algebra $\mathfrak{g}/I$ where I is an ideal in $\mathfrak{g}$. The input arguments is the set of vectors defining an ideal and a set of vectors defining a vector space complement.

```
> QD := QuotientAlgebra( Rad, [e4,e5,e6]);
```

```
[[e1,e2]=-3,[e1,e3]=e2,[e2,e3]=-e1]
```

We can compare these bracket rules with those in table 2.10. We see that this quotient algebra is isomorphic to the subalgebra {`e4,e5,e6`}, meaning it is semisimple.

Finally, we will compute three different Cartan subalgebras.

```
> CartanSubalgebra(E3,contains=[e1]);
```

```
[e1,e6]
```

```
> CartanSubalgebra(E3,contains=[e2]);
```

```
[e2,e5]
```

```
> CartanSubalgebra(E3,contains=[e3]);
```

```
[e3,e4]
```

CHAPTER 3
DECOMPOSITION OF LIE ALGEBRA REPRESENTATIONS

3.1 Introduction

In this chapter, we will review the theory of the classification of the representations of semisimple Lie algebras. We will see that every finite-dimensional representation of a semisimple Lie algebra is equivalent to a direct sum of irreducible representations. We will show that the irreducible subrepresentations can be generated by the **highest weight vectors**. Furthermore, the eigenvalues of the highest weight vectors can be used to compute invariants of the irreducible subrepresentations called a **highest weights**. As we outline the theory for classifying the representations, we will discuss the computational procedures used for identifying the irreducible subrepresentations and classifying the given representation.

The maple code for computing the highest weight vectors and the highest weights can be found in appendix B.2. Tutorials are provided in appendix C.5 to illustrate how a matrix representation of a semisimple Lie algebra can be decomposed into a direct sum of irreducible subrepresentations identified by the highest weights.

Often times, in practice one encounters Lie algebras by a representation of the Lie algebra, namely, linear transformations on a vector space preserving the structure of the Lie algebra. Recall that we showed that $\mathrm{end}(V)$ is a Lie algebra using the commutator as the Lie bracket. Therefore, treating $\mathrm{end}(V)$ as a Lie algebra, we can talk about an Lie algebra homomorphism from a Lie algebra $\mathfrak{g}$ to $\mathrm{end}(V)$. This leads us to the definition of a representation.

Definition 3.1. A **representation** of a Lie algebra $\mathfrak{g}$ on a vector space V is a homomorphism of Lie algebras

$$\rho : \mathfrak{g} \to \mathrm{end}(V) \tag{3.1}$$

where $x \in \mathfrak{g}$ is sent to an element $\rho_x \in \text{end}(V)$ such that $\rho_{[x,y]}(v) = \rho_x(\rho_y(v)) - \rho_y(\rho_x(v))$ for all $v \in V$.

In layman's terms, a representation is a family of linear transformations of a vector space which exhibit the same algebraic structure as the Lie algebra under the commutator operation.

There are several different notations that are common when working with representations. One method uses the notation of Lie algebra homomorphisms as given in the definition. Another method allows us to simplify the notation substantially by considering the representation as a module. In many texts, the authors simply state that one can consider V as a $\mathfrak{g}$-module. This can be seen by letting the action $x \cdot v = \rho_x(v)$. This will allow us to shorten the notation and refer the representation by V.

For details on the equivalence of a representation $\rho : \mathfrak{g} \to V$ and a $\mathfrak{g}$-module V, see appendix *A*.

Next we define what it means for two representations to be equivalent.

Definition 3.2. Two Lie algebra representations $\rho : \mathfrak{g} \to \text{end}(V)$ and $\sigma : \mathfrak{g} \to \text{end}(W)$ are said to be **equivalent** if there exists a vector space isomorphism $A : V \to W$ such that $A \circ \rho_x = \sigma_x \circ A$ for all $x \in \mathfrak{g}$.

One might be inclined to ask when two representations of a Lie algebra are equivalent. This leads us to the issue of classifying the representations of the semisimple Lie algebras.

A **subrepresentation** is a vector subspace W of V which is preserved by the action of $\mathfrak{g}$. This means that $\rho_x(W) \subset W$ for all $x \in \mathfrak{g}$. In the language of modules, $W \subset V$ is a subrepresentation if W is a algebra $\mathfrak{g}$-submodule of V which is invariant under the action of $\mathfrak{g}$. A representation V is said to be **irreducible** if the only subrepresentations of V are (0) and V.

There are three fundamental theorems which are used in the classification of finite-dimensional representations of semisimple Lie algebras. They are given below.

Theorem 3.1. *For every finite-dimensional representation V (as a module) of a complex semisimple Lie algebra $\mathfrak{g}$, there is a decomposition*

$$V = V_1^{\oplus n_1} \oplus \cdots \oplus V_k^{\oplus n_k}$$

where each V_i is a distinct (up to equivalence) irreducible $\mathfrak{g}$-module. This decomposition is unique up to ordering.

Theorem 3.2. *Let $\mathfrak{g}$ be a complex semisimple Lie algebra of rank ℓ. Then for every irreducible representation of $\mathfrak{g}$ there exists a tuple of nonnegative integers $(a_1, ..., a_\ell)$, called the highest weight, which is an invariant of the representation. Furthermore, any two irreducible representations are equivalent if and only if they have the same highest weight.*

Theorem 3.2 states that if ρ and σ are two equivalent irreducible representations, then the highest weights of both representations will be the same tuple of nonnegative integers. If $\lambda = (a_1, a_2, ..., a_\ell)$ is the highest weight of an irreducible representation, then we will denote the representation by Γ_λ. Then, given any finite-dimensional representation V of a semisimple Lie algebra over $\mathbb{C}$, Theorems 3.1 and 3.2 allow us to decompose the representation into a direct sum of irreducible representations which are identified by their unique highest weights. Therefore, letting $\{\lambda_i\}$ be the set of highest weights of V, then

$$V = \bigoplus_{k=1}^{m} \Gamma_{\lambda_k}^{\oplus n_k}. \tag{3.2}$$

For any complex semisimple Lie algebra $\mathfrak{g}$, we will refer to the irreducible representation with highest weight $\lambda = (1, 0, ..., 0)$ as the **Standard Representation**. When working with explicit examples, we will often denote the standard representation with the $\mathfrak{g}$-module V. We will also refer to the irreducible representation given by $(0, ..., 1, ..., 0)$ (all zeros except for a 1 in the i^{th} slot) as the i^{th} **Fundamental Representation**.

Theorem 3.3. *Let $\mathfrak{g}$ be a complex semisimple Lie algebra of rank ℓ. Then for every tuple of nonnegative integers $(a_1, ..., a_\ell)$ there exists an irreducible representation such that $(a_1, ..., a_\ell)$ is the highest weight of the representation.*

Theorems 3.1-3.3 give a complete classification of the finite-dimensional representations of the semisimple Lie algebras. Theorem 3.3 allows one to build a representation by first constructing irreducible subrepresentations. Historically, this theorem has proven to be the most difficult to prove. It has also proven to be the most difficult to implement on the computer.

Sections 3.2 – 3.4 will be discussions on these three classification theorems. Section 3.2 will be an outline of the key elements required to prove of Theorem 3.1 as well as an outline of the proof. Section 3.3 will be a discussion on the implementation of Theorem 3.2 by outlining the procedure to calculate the highest weights of a representation. These highest weights allow us to identify, abstractly, the irreducible subrepresentations. Section 3.4 will discuss briefly what work has been done to identify an irreducible representation for a given highest weight.

Many of the key definitions and theorems required to work with Theorems 3.1 – 3.3 will be stated. After the discussion on the classification theorems, we will proceed to outline the analysis of the representations of $\mathfrak{sl}_2(\mathbb{C})$ in section 3.5. We will show exactly what every irreducible representation of $\mathfrak{sl}_2(\mathbb{C})$ looks like, up to equivalence.

We will then expand upon many of the concepts from the analysis of the representations of $\mathfrak{sl}_2(\mathbb{C})$ in section 3.6, and develop the necessary tools to work with the representations of $\mathfrak{sl}_3(\mathbb{C})$. Towards the end of the section, we will state exactly what every irreducible representation of $\mathfrak{sl}_3(\mathbb{C})$ is up to equivalence.

We will see that many of the tools required to identify the irreducible representations of $\mathfrak{sl}_3(\mathbb{C})$ only require results from linear algebra and the fact that $\mathfrak{sl}_3(\mathbb{C})$ is semisimple. In section 3.7 we will develop a general procedure to obtain the decomposition of a representation of any semisimple Lie algebra as given in equation (3.2).

3.2 Overview of Theorem 3.1

For the remainder of this work, it will be assumed that $\mathfrak{g}$ is a complex semisimple Lie algebra and $\mathfrak{h}$ is a Cartan subalgebra of $\mathfrak{g}$.

Definition 3.3. A **weight** of the representation V is a linear functional $\alpha \in \mathfrak{h}^*$ such that there exists a nonzero vector $v \in V$ such that for every $h \in \mathfrak{h}$, $h \cdot v = \alpha(h)v$. The nonzero vector satisfying this relation is called a **weight vector** corresponding to α.

Let $\alpha \in \mathfrak{h}^*$ be a weight of V. Recall that $\mathfrak{h}$ is the maximal set of diagonalizable elements in $\mathfrak{g}$. This means that the action of the elements from $\mathfrak{h}$ are semisimple. Furthermore, because $\mathfrak{h}$ is an abelian subalgebra of $\mathfrak{g}$ when $\mathfrak{g}$ is semisimple, this means that $[h_i, h_j] = 0$ for all $h_i, h_j \in \mathfrak{h}$. Recall, from linear algebra, that the action of a semisimple linear transformation on a vector space allows us to decompose the vector space into a direct sum of eigenspaces. Because the elements of a Cartan subalgebra are simultaneously diagonalizable, we can decompose the vector space V where

$$V = \bigoplus V_\alpha$$

and $V_\alpha = \{v \in V \mid h \cdot v = \alpha(h)v \ \forall h \in \mathfrak{h}\}$. The subspaces V_α are called **weight spaces**. The **multiplicity** of a weight α is the dimension of V_α.

The weights obtained from the adjoint representation are called the **roots** of the Lie algebra and will be denoted by Δ. The weight spaces corresponding to the roots are called **root spaces**. If α is a root, then we will denote the root space corresponding to α by $\mathfrak{g}_\alpha$. By convention, we denote $\mathfrak{g}_0 = \mathfrak{h}$, and do not consider zero to be a root.

Therefore, the action of $\mathfrak{h}$ allows us to decompose the vector space $\mathfrak{g}$ as follows:

$$\mathfrak{g} = \mathfrak{h} \oplus \Big(\bigoplus \mathfrak{g}_\alpha\Big).$$

This decomposition is called a **Root Space Decomposition**.

We will discuss this in further detail at the appropriate time, but now it is noteworthy to observe that we can split the set of roots into **positive roots**, denoted by Δ^+, and **negative roots**, denoted by Δ^- such that $\Delta^+ \cap \Delta^- = \emptyset$, $\Delta = \Delta^+ \cup \Delta^-$ and if $\alpha, \beta \in \Delta^+$ such that $\alpha + \beta \in \Delta$, then $\alpha + \beta \in \Delta^+$. In other words, a set of positive roots is a set of roots such that if the sum of any two positive roots is a root, then the sum is also a positive root.

We will then show that every set of positive roots admits a subset, called the **simple roots**, such that every positive root can be written as a sum of simple roots with nonnegative coefficients. We denote the set of simple roots by Δ^0. We will also see that Δ^0 forms a basis of $\mathfrak{h}^*$.

We will see that, for each root α, $\dim(\mathfrak{g}_\alpha) = 1$. This means that we can pick a basis $\{x\}$ of $\mathfrak{g}_\alpha$ such that if $x' \in \mathfrak{g}_\alpha$, then $x' = ax$ for some scalar value a. For a positive root $\alpha \in \Delta^+$, we will call the basis element $x \in \mathfrak{g}_\alpha$ a **positive root vector** and $y \in \mathfrak{g}_{-a}$ a **negative root vector**.

We will denote the adjoint action of one root space $\mathfrak{g}_\alpha$ simply by $\mathfrak{g}_\alpha$. So, given $x \in \mathfrak{g}_\alpha$ and $y \in \mathfrak{g}$, $\mathfrak{g}_\alpha : \mathfrak{g} \to \mathfrak{g}$ such that $\mathfrak{g}_\alpha(y) = [x, y]$.

Definition 3.4. A nonzero vector $v \in V$ is called a **highest weight vector** if it is *both* a weight vector for the action of $\mathfrak{h}$ and in the kernel of $\mathfrak{g}_\alpha$ for all $\alpha \in \Delta^+$. The weight corresponding to a highest weight vector is called a **highest weight**.

With these definitions, we now have enough terminology to discuss the key concepts required to show that every finite-dimensional representation of a complex semisimple Lie algebra decomposes into a direct some of irreducible representations. The key points needed are listed as follows:

1. Let V be a representation of $\mathfrak{g}$ and $W \subset V$ a subrepresentation. Then there exists a subrepresentation $W' \subset V$ complementary to W, so that $V = W \oplus W'$ (see Theorem 3.4).

2. Shur's Lemma: If V and W are irreducible $\mathfrak{g}$-modules and $\psi : V \to W$ is a $\mathfrak{g}$-module homomorphism, then $\psi = 0$ or ψ is an isomorphism, and if $V = W$ then $\psi = \lambda I$ for some $\lambda \in \mathbb{C}$ (see appendix *A*).

3. Every representation V admits a decomposition into weight spaces, $V = \bigoplus V_\alpha$ (see equation 3.8).

4. The kernel of the action of the positive root vectors is nonempty, $\bigcap_{\alpha \in \Delta^+} ker(\mathfrak{g}_\alpha) \neq \emptyset$ (see Theorem 3.5)

5. Every representation V possesses a highest weight vector (see Theorem 3.5).

6. Given a highest weight vector, let W be the space generated by the images of the highest weight vector under the action of the negative root vectors. Then $W \subset V$ is an irreducible subrepresentation of V (see Theorem 3.6).

Using these ideas, we can outline the proof of Theorem 3.1.

Let V be a finite-dimensional representation of $\mathfrak{g}$. By property (3), we know that there exists a decomposition of the vector space V into a direct sum of weight spaces. Then, by property (4) we know that there exists a nonzero vector $v \in V$ such that $v \in ker(\mathfrak{g}_\alpha)$ for every $\alpha \in \Delta^+$. By property (3), $v \in V_\lambda$ for some weight λ. Thus, by definition, $v \in V$ is a highest weight vector. Let V_1 be the subrepresentation generated by the action of the negative root vectors on the highest weight vector v as given in part (6). Therefore, V_1 is an irreducible subrepresentation of V. By Schur's Lemma, we know that this irreducible representation is unique up to equivalence. By part (1), there exists a subrepresentation $V' \subset V$ such that $V = V_1 \oplus V'$.

By following the same analysis on the subrepresentation V', we can show that there exists a (unique) irreducible subrepresentation $V_2 \subset V'$ and an invariant complement $V'' \subset V'$ such that $V' = V_2 \oplus V''$, making $V = V_1 \oplus V_2 \oplus V''$. Because V is finite-dimensional, we can proceed by induction and decompose V completely into a direct sum of irreducible subrepresentations.

3.3 Overview of Theorem 3.2

One of the motivations for this thesis is to decompose a given representation into a direct sum of irreducible subrepresentations using Theorem 3.1 and identify the irreducible subrepresentations using the highest weights as specified in Theorem 3.2. We will do this by obtaining the highest weights of the representation, and then use properties of the Lie algebra to get the tuple of nonnegative integers which uniquely (up to equivalence) identify the irreducible subrepresentations. We will not prove that the tuple obtained is an invariant, but we will discuss how it can be obtained. To do so, we must first establish some terminology.

Definition 3.5. Given a root $\alpha \in \Delta$, a **coroot** of α is an element $H_\alpha \in \mathfrak{h}$ satisfying the following conditions for all $X_\alpha \in \mathfrak{g}_\alpha$ and $Y_\alpha \in \mathfrak{g}_{-a}$:

$$[H_\alpha, X_\alpha] = 2X_\alpha \ , \qquad [H_\alpha, Y_\alpha] = -2Y_\alpha \ , \qquad [X_\alpha, Y_\alpha] = H_\alpha \ .$$

For any simple root $\alpha \in \Delta^0$, the coroot $H_\alpha \in \mathfrak{h}$ is unique. When explicitly working with the simple roots, we will often denote them with a superscript (as they will sorted with respect to some ordering). So, if $\alpha^i \in \Delta^0$ then we will denote the coroot of α^i by H_i.

Definition 3.6. Let $\Delta^0 = \{\alpha^1, ..., \alpha^\ell\}$ be a set of simple roots and $\{H_1, ..., H_\ell\}$ the corresponding set of coroots. The **fundamental weights** of $\mathfrak{g}$ is a set of elements $\{\omega^1, ..., \omega^\ell\} \subset \mathfrak{h}^*$ dual to $\{H_1, ..., H_\ell\}$.

Let $\{H_1, ..., H_\ell\}$ be the set of coroots for a given choice of Δ^0. This means that $\omega^i(H_j) = \delta^i_j$. Letting $v \in V$ be the highest weight vector of an irreducible representation V, then $H_i \cdot v = a_i v$ for every coroot H_i and the eigenvalues a_i are precisely the integers given in Theorem 3.2. We will now explore a method of obtaining the fundamental weights, and thus the coefficients a_i, without explicitly computing the coroots.

Letting V be an irreducible representation of $\mathfrak{g}$, the key concepts for using Theorem 3.2 are given as follows:

1. After choosing a Cartan subalgebra $\mathfrak{h}$, $\mathfrak{g}$ admits a root space decomposition, and V admits a decomposition into eigenspaces as a vector space, $V = \bigoplus V_\alpha$.
2. Choosing a set of positive roots allows a calculation of a highest weight vector.
3. The highest weight vector of an irreducible representation is unique up to scalars. (see Theorem 3.6)
4. If $v \in V$ is a highest weight vector of V, then $v \in V_\lambda$ for some weight λ. Therefore, λ is a highest weight of V.
5. Computation of the highest weight vector, and the highest weight by extension, depends on the choice of $\mathfrak{g}$ and the choice of positive roots.
6. There exists a basis of $\mathfrak{h}^*$, called the set of fundamental weights, such that the representation of the highest weight is invariant under a choice of $\mathfrak{h}$ and Δ^+.
7. The coefficients of the highest weight in the basis of the fundamental weights are nonnegative integers.

Theorem 3.2 is focused on classifying the irreducible representations by identifying an invariant of the representation. This means we will be interested in finding a property of the representation which depends solely on the Lie algebra $\mathfrak{g}$ and the $\mathfrak{g}$-module V. However, in calculating the highest weight vector of a representation (which we use in generating an irreducible subrepresentation), a choice of a Cartan subalgebra and a

choice of the splitting of the roots are required. To be more explicit, an invariant should not depend on any choice of basis (of either $\mathfrak{g}$ or V), the choice of $\mathfrak{h}$, nor the choice of the splitting of the roots for a given $\mathfrak{h}$.

The proof of Theorem 3.2 will not be dealt with in this thesis. However, we will be interested in a method of identifying the coefficients of the highest weight in the basis of the fundamental weights, thus allowing us to uniquely classify the irreducible representations.

3.4 Overview of Theorem 3.3

The third classification theorem will not be dealt with in this thesis beyond this brief discussion. Although, it may have appeared to be a disjoint project, much of the work in chapter 2 has been done to pave the way for explicitly building the irreducible representations of the complex semisimple Lie algebras. The Clifford algebras are used in calculating some of the fundamental representations of the special orthogonal Lie algebras. The octonions are used to create a fundamental representation of $\mathfrak{g}_2$. The octonions are also used in conjunction with the Jordan algebras to create a fundamental representation of the exceptional Lie algebra $\mathfrak{f}_4$.

In the text *Representation Theory: A First Course* [3], it is shown that the irreducible representation $\Gamma_{a_1,\ldots,a_n}$ of $\mathfrak{sl}_{n+1}(\mathbb{C})$ will appear as a subspace of the tensor product

$$Sym^{a_1}V \otimes Sym^{a_2}(\wedge^2 V) \otimes \ldots \otimes Sym^{a_n}(\wedge^n V).$$

Producing irreducible representations of $\mathfrak{sp}_{2n}(\mathbb{C})$ are a bit more complicated. The kernel of the map $\varphi_k : \wedge^k V \to \wedge^{k-2} V$ is the kth fundamental representation denoted by $V^{(k)} = \Gamma_{0,\ldots,0,1,0,\ldots,0}$. Then the irreducible representation $\Gamma_{a_1,\ldots,a_n}$ will occur in the product

$$Sym^{a_1}V \otimes Sym^{a_2}V^{(2)} \otimes \ldots \otimes Sym^{a_n}V^{(n)}.$$

The irreducible representations for $\mathfrak{so}_m(\mathbb{C})$ are even more complicated. Simply describing a vector space which contains the irreducible representation $\Gamma_{a_1,\ldots,a_n}$ requires more definitions than are warranted for this discussion. It suffices to mention that these spaces can be constructed in Maple. Hence, we can produce representation which will decompose to produce a specified irreducible representation. However, creating the larger representations to decompose is extremely time consuming and quickly becomes

highly unpractical for Lie algebras of moderate rank (like $\ell > 5$) and representations of moderate dimension.

Maple worksheets are provided in appendix C which illustrate how we can create a specified irreducible representation for several low-rank Lie algebras. However, in general, this method of creating irreducible representations is not practical, and methods are still being developed to make this process more efficient.

3.5 Representations of $\mathfrak{sl}_2(\mathbb{C})$

Recall from section 2.5, the matrix algebra of 2×2 trace-free matrices over $\mathbb{C}$ is a Lie algebra where the bracket is given to be the commutator, $[A, B] = AB - BA$. In short, we have

$$\mathfrak{sl}_2(\mathbb{C}) = \left\{ \begin{pmatrix} a & b \\ c & -a \end{pmatrix} \mid a, b, c \in \mathbb{C} \right\}$$

There is a natural basis for $\mathfrak{sl}_2$ that we can choose, namely the three matrices given in (2.10). Then the bracket rules on the basis elements are found to be

$$[H, X] = 2X, \quad [H, Y] = -2Y, \quad [X, Y] = H. \tag{3.3}$$

Now, let V be an irreducible finite-dimensional representation of $\mathfrak{sl}_2(\mathbb{C})$. By the preservation of Jordan decomposition, we have that the action of H on V is diagonalizable. This implies that the representation V can be decomposed as

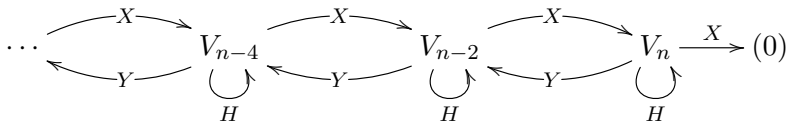
$$V = \bigoplus V_\alpha \tag{3.4}$$

where the α are a collection of (complex) eigenvalues of H, namely that for any $v \in V_\alpha$, $H(v) = \alpha \cdot v$. Next on the agenda is to determine the action of X and Y on each of the spaces V_α. Let $v \in V_\alpha$ be given. Then, since $[H, X](v) = H(X(v)) - X(H(v))$, we have

$$\begin{aligned} H(X(v)) &= X(H(v)) + [H, X](v) \\ &= X(\alpha \cdot v) + 2X(v) \\ &= (\alpha + 2) \cdot X(v) \end{aligned}$$

From this, we see that if α is an eigenvector of H with eigenvalue of α, then $X(v)$ is *also* an eigenvector of H with eigenvalue $\alpha + 2$. This means that the action of X on a vector from V_α produces a vector in $V_{\alpha+2}$. In other words, $X : V_\alpha \longrightarrow V_{\alpha+2}$. Similarly, we find that $Y : V_\alpha \longrightarrow V_{\alpha-2}$. Furthermore, by induction can obtain $H(X^k(v)) = (\alpha+2k)X^k(v)$ and $H(Y^k(v)) = (\alpha - 2k)Y^k(v)$. By assumption, since V is irreducible, we know that there are no proper subrepresentations. This means that each of the eigenvalues α that appear in the decomposition of V, given above, must be congruent to each other modulo 2. This means that the set of eigenvalues of H form a sequence of numbers of the form $\alpha, \alpha + 2, ..., \alpha + 2k$ for some $\alpha \in \mathbb{C}$ and $m \in \mathbb{Z}$ and $V = \bigoplus_{k\in\mathbb{Z}} V_{\alpha+2k}$. Because the representation is finite-dimensional, let n denote the last element in the string of eigenvalues. Keep in mind that we only know that n is a complex number. Next, we will show that it is, in fact, an integer.

If we were to draw a picture of the action of X, Y, and H on the eigenspaces of V, we would see that



Choose any nonzero vector $v \in V_n$. By the maximality of n, we know that $V_{n+2} = (0)$, which also implies that $X(v) = 0$.

Proposition 3.1. $\{v, Y(v), Y^2(v), ...\}$ *spans the representation* V.
(The proof of this can be found in chapter 11 of [3].)

From this result, we observe that $dim(V_\alpha) = 1$ for every α.

Again, because V is finite-dimensional, we know that there exists a lower bound on the eigenvalues α such that $V_\alpha \neq (0$ so that $Y^k(v) = 0$ for sufficiently large k. Let m be the smallest positive integer such that $Y^m(v) = 0$. Then we have that

$$0 = X(Y^m(v)) = m(n - m + 1)Y^{m-1}(v).$$

Together with the fact that $Y^{m-1}(v) \neq 0$, we see that $n - m + 1 = 0$. Because m is an integer, we conclude that n must be a non-negativeinteger also. Therefore, we have that the eigenvalues α of H on V form a string of integers which differ by 2 and are symmetric

about the origin. This means that for each integer n, there exists a unique $(n+1)$-dimensional representation $V^{(n)}$ with H having eigenvalue $\{n, n-2, ..., -n+2, -n\}$.

To recap, let's identify some of the major observations that we made in working with the representations of $\mathfrak{sl}_2(\mathbb{C})$. First, once we had obtained a basis, we identified the element with was diagonalizable, in our case H was already diagonal. The next major step was to find a non-zero vector which was an eigenvector of H which was also killed by X. The eigenvalue corresponding to this eigenvector, was shown to be a non-negative integer and identified the irreducible representation $V^{(n)}$. This was then shown to be a symmetric power of the standard representation $V^{(n)} = \mathrm{Sym}^n(\mathbb{C}^2)$.

Working with the representations of $\mathfrak{sl}_2(\mathbb{C})$ will give us the proper direction for working with $\mathfrak{sl}_3(\mathbb{C})$.

3.6 Representations of $\mathfrak{sl}_3(\mathbb{C})$

The Lie algebra $\mathfrak{sl}_3(\mathbb{C})$ is the space of 3×3 trace-free matrices over $\mathbb{C}$ together with the commutator. Letting E_i^j be a 3×3 matrix with one in the (i,j)th entry and zeros elsewhere, then a natural basis to select is

$$H_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}, \quad H_2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

together with the matrices E_i^j with $1 \leq i \neq j \leq 3$. From the previous section, we saw that we needed to first find the elements of the basis which were diagonalizable. Notice, that we have two matrices H_1 and H_2 which satisfy that criteria. This means that we need to replace the element $H \in \mathfrak{sl}_2(\mathbb{C})$ with a subspace $\mathfrak{h} \subset \mathfrak{sl}_3(\mathbb{C})$ of diagonalizable elements. Furthermore, we observe that commuting diagonalizable matrices are simultaneously diagonalizable. Therefore, for any finite-dimensional representation V of $\mathfrak{sl}_3(\mathbb{C})$ admits a decomposition

$$V = \bigoplus V_\alpha$$

where every vector $v \in V_\alpha$ is an eigenvector for every element $H \in \mathfrak{h}$. In our analysis of $\mathfrak{sl}_2(\mathbb{C})$ we discussed and used the eigenvalues of the diagonal matrix H. In our case, we need to modify things a bit. Because we are dealing with a space of matrices $\mathfrak{h}$, we

say that $v \in V$ is *eigenvector for* $\mathfrak{h}$ if v is an eigenvector for each $H \in \mathfrak{h}$. For such an eigenvector v, we write for each $H \in \mathfrak{h}$

$$H(v) = \alpha(H) \cdot v$$

where $\alpha(H)$ is a scalar value which is linearly dependent on H. In other words, $\alpha : \mathfrak{h} \to \mathbb{C}$ or $\alpha \in \mathfrak{h}^*$. From this, we can define what we mean by an *eigenvalue* for the action of $\mathfrak{h}$.

Definition 3.7. An element $\alpha \in \mathfrak{h}^*$ is called an *eigenvalue* of $\mathfrak{h}$ if there exists a nonzero $v \in V$ such that $H(v) = \alpha(H) \cdot v$ for each $H \in \mathfrak{h}$.

The space of eigenvectors $V_\alpha = \{v \in V \mid H(v) = \alpha(H) \cdot v \ \forall H \in \mathfrak{h}\}$ is called the *eigenspace* associated to the eigenvalue α. By the preservation of Jordan decomposition we have

Proposition 3.2. *Any finite-dimensional representation V of $\mathfrak{sl}_3(\mathbb{C})$ has a decomposition*

$$V = \bigoplus V_\alpha \tag{3.5}$$

where V_α is an eigenspace for $\mathfrak{h}$ and α ranges over a finite subset of $\mathfrak{h}^*$.

Now that we have decided on an analogue for the diagonal element from $\mathfrak{sl}_2(\mathbb{C})$ which has allowed us to decompose the representation into a direct sum of eigenspaces, we now move on to establishing analogues for the elements X and Y in $\mathfrak{sl}_3(\mathbb{C})$.

Recall that $[H, X] = 2X$ and $[H, Y] = -2Y$. Another way to think of this is that X and Y are eigenvectors for the adjoint action of H on $\mathfrak{sl}_2(\mathbb{C})$. This means that we could decompose the adjoint representation of $\mathfrak{sl}_2(\mathbb{C})$ as $\mathfrak{sl}_2(\mathbb{C}) = H \oplus (X \oplus Y)$. Although this seems ridiculously obvious, this same line of reasoning allows us to decompose the algebra $\mathfrak{sl}_3(\mathbb{C})$ by using the eigenvectors for the adjoint action of $\mathfrak{h}$ on $\mathfrak{sl}_3(\mathbb{C})$. Using the proposition above, we get a decomposition

$$\mathfrak{sl}_3(\mathbb{C}) = \mathfrak{h} \oplus (\bigoplus \mathfrak{g}_\alpha) \tag{3.6}$$

where α ranges over a finite set of $\mathfrak{h}^*$ and for each $H \in \mathfrak{h}$ and $X \in \mathfrak{g}_\alpha$, $[H, X] = \alpha(H) \cdot X$. The next step to take is to discuss what the eigenvalues of $\mathfrak{h}$ look like and how elements

from an eigenspace act on another eigenspace. Note that the subalgebra

$$\mathfrak{h} = \left\{ \begin{pmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{pmatrix} \mid a_1 + a_2 + a_3 = 0 \right\}.$$

Then the space $\mathfrak{h}^*$ is spanned by a set of linear functionals L_i where

$$L_i \begin{pmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{pmatrix} = a_i$$

and $L_1 + L_2 + L_3 = 0$. As an example, we see that $L_1(H_1) = 1$ and $L_1(H_2) = 0$, and $L_3(H_1) = -1$ and $L_3(H_2) = -1$. So in the basis we have established for $\mathfrak{h}$, we can represent $L_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $L_3 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$.

Proposition 3.3. *The linear functionals $\alpha \in \mathfrak{h}^*$ from equation (3.6) are the functionals $L_i - L_j$. Furthermore, the space $\mathfrak{g}_{L_i - L_j}$ will be generated by E_i^j.*

Proof: Let $H = \begin{pmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{pmatrix}$ and $\alpha = L_i - L_j \in \mathfrak{h}^*$. Then

$$\begin{aligned} \left[H, E_i^j\right] &= a_i E_i^j - a_j E_i^j \\ &= (a_i - a_j) E_i^j \\ &= (L_i(H) - L_j(H))\, E_i^j \\ &= \alpha(H) E_i^j. \end{aligned}$$

By repeating this procedure for each of the matrices E_i^j, we obtain all the functionals of the decomposition. This means that $\pm(L_1 - L_2)$, $\pm(L_1 - L_3)$ and $\pm(L_2 - L_3)$ are the six linear functionals in the decomposition (zero. From this, we see that the space $\mathfrak{g}_{L_i - L_j}$ will be generated by the matrix E_i^j. □

We see that the action of an element from $\mathfrak{h}$ sends $\mathfrak{g}_\alpha$ to itself. Letting $X \in \mathfrak{g}_\alpha$, we want to determine where $ad(X)$ sends a vector $Y \in \mathfrak{g}_\beta$. This can be done by finding the

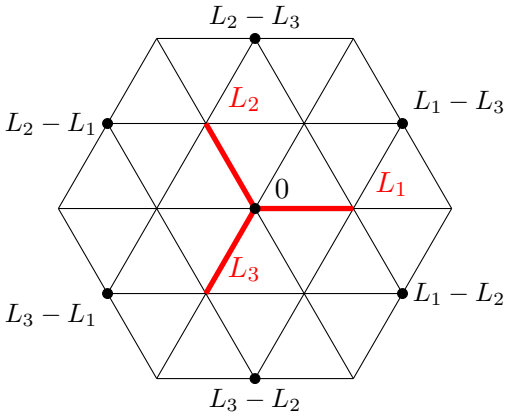


FIGURE 3.1: Diagram for the eigenvalues corresponding to the decomposition (3.6).

action of an element $H \in \mathfrak{h}$ on $ad(X)(Y)$.

$$\begin{aligned}
[H,[X,Y]] &= [X,[H,Y]] + [[H,X],Y] \\
&= [X,\beta(H)Y] + [\alpha(H)X,Y] \\
&= \beta(H)[X,Y] + \alpha(H)[X,Y] \\
&= (\alpha(H) + \beta(H))[X,Y].
\end{aligned}$$

This means that $[X,Y]$ is an eigenvector for $\mathfrak{h}$ with eigenvalue $\alpha + \beta$. Therefore, the adjoint action of an eigenspace $\mathfrak{g}_\alpha$ on another eigenspace $\mathfrak{g}_\beta$ is

$$ad(\mathfrak{g}_\alpha) : \mathfrak{g}_\beta \longrightarrow \mathfrak{g}_{\alpha+\beta}$$

This means that the adjoint action carries an eigenspace into another eigenspace. In the case that $\alpha + \beta$ is not an eigenvalue present in the decomposition in 3.6, then the action kills the eigenspace (carries it to the zero space). For example, consider the action of $\mathfrak{g}_{L_2-L_3}$ on each of the six eigenspaces. We see that the action of $\mathfrak{g}_{L_2-L_3}$ carries: $\mathfrak{g}_{L_1-L_2}$ into $\mathfrak{g}_{L_1-L_3}$, $\mathfrak{g}_{L_3-L_1}$ into $\mathfrak{g}_{L_2-L_1}$, $\mathfrak{h}$ into $\mathfrak{g}_{L_2-L_3}$, $\mathfrak{g}_{L_3-L_2}$ into $\mathfrak{h}$, and kills $\mathfrak{g}_{L_1-L_3}$, $\mathfrak{g}_{L_2-L_3}$, and $\mathfrak{g}_{L_2-L_1}$. This process gives us a picture of how each of the elements act on each other.

Now we need to repeat this analysis on the eigenspace decomposition of the representation $V = \oplus V_\alpha$. Let $X \in \mathfrak{g}_\alpha$ and let $v \in V_\beta$. Again, we will now how X acts on v if

we can determine exactly how any $H \in \mathfrak{h}$ acts on $X(v)$. Thus we see that

$$\begin{aligned} H(X(v)) &= X(H(v)) + [H, X](v) \\ &= X(\beta(H) \cdot v) + (\alpha(H) \cdot X)(v) \\ &= (\alpha(H) + \beta(H)) \cdot X(v). \end{aligned}$$

From this, we see that $X(v)$ is an eigenvector for the action of $\mathfrak{h}$ with eigenvalue $\alpha + \beta$. Similar to before, we see that this means the action of $\mathfrak{g}_\alpha$ takes V_β into $V_{\alpha+\beta}$. Similar to the result we obtained for $\mathfrak{sl}_2(\mathbb{C})$, we obtain the following:

Proposition 3.4. *The eigenvalues α occurring in an irreducible representation of $\mathfrak{sl}_3(\mathbb{C})$ differ from one another by integral linear combinations of $L_i - L_j \in \mathfrak{h}^*$.*

Recall that the eigenvalues $\alpha \in \mathfrak{h}^*$ of the action of $\mathfrak{h}$ on the representation are called **weights**. The corresponding eigenvectors in V_α are called the **weight vectors** and the space V_a is called a **weight space** of the representation. The weights occurring in the adjoint representation, the representation which was used to develop this analysis, are special enough to be given a distinguishing name called **roots** of the Lie algebra, and the subspaces $\mathfrak{g}_\alpha$ are called the root spaces. The lattice $\Lambda_R \subset \mathfrak{h}^*$ generated by the roots is called the **root lattice**. Note that in the case of $\mathfrak{sl}_3(\mathbb{C})$, $\mathfrak{h}^*$ is two-dimensional, so it forms a plane.

Recall from our treatment of $\mathfrak{sl}_2(\mathbb{C})$, that once we had identified the eigenvalues of the representation, we then identified an extremal eigenspace of the representation. Retrieving a vector from the extremal eigenspace, we found that the action of X killed the vector. Thus finding an eigenvector of H which was killed by X allowed us to completely characterize the representation. We will now develop a process for a representation of $\mathfrak{sl}_3(\mathbb{C})$ which is motivated by this procedure.

To start, we need to figure out what it means for a weight space V_α to extremal. For $\mathfrak{sl}_2(\mathbb{C})$, the eigenvalues were scalar values which differed by integral multiples of two. This made an ordering very natural. However, we are now dealing with linear functionals in $\mathfrak{h}^*$ where no such ordering intuitively exists. What we can do is choose a division of $\mathfrak{h}^*$ such that the set of roots are divided in twain, namely a set of **positive roots** and **negative roots**. This is motivated by the observation that $L_i - L_j$ and $-(L_i - L_j)$ are both roots.

To achieve this splitting, consider a line ℓ in $\mathfrak{h}^*$ passing through the origin. This line cuts $\mathfrak{h}^*$ into two half planes. If we let ℓ have an irrational slope, then observe that the division of $\mathfrak{h}^*$ (and in turn Λ_R) is such that $L_i - L_j$ for $i < j$ will lie in one half-plane and the $L_i - L_j$ for $j > i$ lie in the other half-plane. Next, choose a vector $\ell^\perp$ which is perpendicular to the line ℓ. Now, we call the roots $L_i - L_j$ which lie in the same half-plane as $\ell^\perp$ to be the positive roots. In practice, this process is much easier and much less ambiguous than it appears here.

This division of the root lattice into positive and negative roots gives us a space of vectors analogous to $X \in \mathfrak{sl}_2(\mathbb{C})$ (positive) and a space of vectors analogous to $Y \in \mathfrak{sl}_2(\mathbb{C})$ (negative). We can now state the following claims.

Claim: There exists a vector $v \in V$ such that v is an eigenvector for $\mathfrak{h}$ and $X_\alpha(v) = 0$ for every X_α in the positive root space.

For any representation V of $\mathfrak{sl}_3(\mathbb{C})$, a vector $v \in V$ with the both properties as stated in the claim is called a **highest weight vector**. The eigenvalue of the highest weight vector v is called the **highest weight** corresponding to v. Recall that the existence of such a vector from an irreducible representation of $\mathfrak{sl}_2(\mathbb{C})$ allowed us to generate the representation by using the images of v under the successive applications of Y. In the case of $\mathfrak{sl}_3(\mathbb{C})$, we have a similar case.

Claim: Let V be an irreducible representation of $\mathfrak{sl}_3(\mathbb{C})$, and $v \in V$ be a highest weight vector. Then V is generated by the images of v under successive applications of the vectors from the positive root space.

Using this claim, we obtain the following propositions which are used to completely classify the irreducible representations of $\mathfrak{sl}_3(\mathbb{C})$.

Proposition 3.5. *If V is any representation of $\mathfrak{sl}_3(\mathbb{C})$ and $v \in V$ is a highest weight vector, then the subrepresentation W of V generated by the images of v by the successive application of the vectors from the positive root space is irreducible.*

Proposition 3.6. *All the eigenvalues of any irreducible finite-dimensional representation of $\mathfrak{sl}_3(\mathbb{C})$ must lie in the weight lattice $\Lambda_W \subset \mathfrak{h}^*$ generate by $\{L_i\}$ and must be congruent modulo the root lattice $\Lambda_R \subset \mathfrak{h}^*$ generate by $\{L_i - L_j\}$.*

Proposition 3.7. *For any pair of natural numbers a and b, there exists a unique (up to isomorphism) irreducible, finite-dimensional representation of $\mathfrak{sl}_3(\mathbb{C})$ with highest weight $aL_1 - bL_3$. This representation is denoted by $\Gamma_{a,b}$.*

The proofs of each of these propositions will not be included here. In fact, they will be the results of more general theorems that will be discussed in the next section.

3.7 Detailed Discussion of Theorems 3.1 and 3.2

Notice that many of the key points in our analysis of the representations of $\mathfrak{sl}_3(\mathbb{C})$ were motivated by the standard 3×3 matrices representing the algebra, but ultimately did not require any properties specific to the algebra. We only needed results from Linear algebra, and properties of finite-dimensional semisimple Lie algebras over $\mathbb{C}$. What this means is that we can generalize the process to working with any finite-dimensional semisimple Lie algebra over $\mathbb{C}$.

To begin, let V be a representation of a finite-dimensional semisimple Lie algebra over $\mathbb{C}$, $\mathfrak{g}$. Let $\mathfrak{h}$ be a Cartan subalgebra of $\mathfrak{g}$. Once we have a Cartan subalgebra, we can calculate the set of roots, Δ, giving us a root space decomposition.

One of the fundamental theorems of representation theory is stated below and will given without proof.

Theorem 3.4. *Let V be a finite-dimensional representation of a complex semisimple Lie algebra. Then for every subrepresentation $W \subset V$, there exists a subrepresentation $W' \subset V$ which is an invariant complement to W, so $V = W \oplus W'$.*

Lemma 3.1. *For any $\alpha, \beta \in \Delta$, $[\mathfrak{g}_\alpha, \mathfrak{g}_\beta] \subset \mathfrak{g}_{\alpha+\beta}$.*

Proof: Let $\alpha, \beta \in \Delta$ be roots. Recall that $[\mathfrak{g}_\alpha, \mathfrak{g}_\beta] = \{[u, v] \mid u \in \mathfrak{g}_\alpha \text{ and } v \in \mathfrak{g}_\beta\}$. Therefore, letting $x \in \mathfrak{g}_\alpha$ and $y \in \mathfrak{g}_\beta$, we have that $[x, y] \in [\mathfrak{g}_\alpha, \mathfrak{g}_\beta]$. Then, for any

$H \in \mathfrak{h}$,

$$\begin{aligned}
[H,[x,y]] &= [[H,x],y] + [x,[H,y]] \\
&= [ad_H(x),y] + [x,ad_H(y)] \\
&= [\alpha(H)x,y] + [x,\beta(H)y] \\
&= \alpha(H)[x,y] + \beta(H)[x,y] \\
&= (\alpha(H)+\beta(H))[x,y].
\end{aligned}$$

Therefore, we see that $ad_H([x,y]) = (\alpha+\beta)(H)[x,y]$, so that $[x,y] \in \mathfrak{g}_{\alpha+\beta}$. □

Although we will be working with the representation V, many of the properties which will be used in the classification of V will come from the root lattice. This means that we will need to use the adjoint representation of $\mathfrak{g}$ in our analysis. Let $\{h_1, ..., h_\ell\}$ be a basis of $\mathfrak{h}$. For each h_i, the linear transformation $ad(h_i)$ is diagonalizable over $\mathbb{C}$. Therefore, because $[\mathfrak{h},\mathfrak{h}] = 0$, we know from basic linear algebra that the matrices ad_{h_i} are simultaneously diagonalizable.

This means that there exists a vector $x \in \mathfrak{g}$ such that $ad_{h_i}(x) = [h_i, x] = a_i x$ for each $h_i \in \mathfrak{h}$. The ℓ-tuple $\alpha = (a_1, ..., a_\ell) \in \mathfrak{h}^*$ is a root of $\mathfrak{g}$ with respect to $\mathfrak{h}$ and the Lie algebra decomposes as

$$\mathfrak{g} = \mathfrak{h} \oplus \left(\bigoplus \mathfrak{g}_\alpha\right) \tag{3.7}$$

which we recognize as a root space decomposition.

The following are some useful properties of a root space decomposition:

1. $\dim(\mathfrak{g}_\alpha) = 1 \ \forall \alpha \in \Delta$
2. Δ will generate a lattice $\Lambda_\Delta \subset \mathfrak{h}^*$ called a **root lattice**.
3. If $\alpha \in \Delta$ is a root, then $-\alpha \in \Delta$ is also a root.
4. If $x \in \mathfrak{g}_\alpha$ and $y \in \mathfrak{g}_\beta$, then $[x,y] \in \mathfrak{g}_{\alpha+\beta}$ if $\alpha+\beta \in \Delta$, otherwise $[x,y] = 0$.

Now, let $\mathfrak{g}_\alpha \subset \mathfrak{g}$ be a one dimensional root space (as given by property 1). By property 3, we know that $\mathfrak{g}_{-\alpha} \subset \mathfrak{g}$ is also a root space. Furthermore, notice that

$[\mathfrak{g}_\alpha, \mathfrak{g}_{-\alpha}] \subset \mathfrak{g}_0 = \mathfrak{h}$, so that the bracket of the root spaces is a subspace of $\mathfrak{h}$ of dimension at most one. Therefore, the adjoint action of $[\mathfrak{g}_\alpha, \mathfrak{g}_{-\alpha}]$ takes $\mathfrak{g}_\alpha$ into itself, and likewise so for $\mathfrak{g}_{-\alpha}$. This means that $\mathfrak{g}_\alpha \oplus \mathfrak{g}_{-\alpha} \oplus [\mathfrak{g}_\alpha, \mathfrak{g}_{-\alpha}] = \mathfrak{s}_\alpha$ is a subalgebra of $\mathfrak{g}$.

The following lemma will be given without proof.

Lemma 3.2. $[\mathfrak{g}_\alpha, \mathfrak{g}_{-\alpha}] \neq 0$ *and* $[\mathfrak{g}_\alpha, \mathfrak{g}_{-\alpha}] \neq 0$

From the lemma it follows that $\mathfrak{s}_\alpha \cong \mathfrak{sl}_2\mathbb{C}$. Therefore, we see that for each $\alpha \in \Delta$, there are vectors $X_\alpha \in \mathfrak{g}_\alpha$, $Y_\alpha \in \mathfrak{g}_{-\alpha}$ and $H_\alpha \in \mathfrak{h}$ such that $[H_\alpha, X_\alpha] = 2X_\alpha$, $[H_\alpha, Y_\alpha] = -2Y_\alpha$ and $[X_\alpha, Y_\alpha] = H_\alpha$. Therefore, an immediate consequence of this is that for any root $\alpha \in \Delta$ there will exist a unique coroot $H_\alpha \in \mathfrak{h}$.

This is a good place for us to stop and discuss the induced inner product on $\mathfrak{h}^*$. Recall that the Killing form B is an inner product on $\mathfrak{g}$. We can restrict the Killing form to the Cartan subalgebra $\mathfrak{h}$. One of the key results of semisimple Lie algebras is that this restriction is positive definite. This means that for any $H \in \mathfrak{h}$, $B(H, H) > 0$. Therefore, consider the coroots of the roots in Δ. Define an element in the Cartan subalgebra $T_\alpha = 2H_\alpha / B(H_\alpha, H_\alpha)$, which is just a scaled version of the coroot H_α.

Then there is an isomorphism of $\mathfrak{h}^*$ and $\mathfrak{h}$ determined by the Killing form B which takes α to T_α. With this, we can define the induced Killing form on $\mathfrak{h}^*$ by $\tilde{B}(\alpha, \beta) = B(T_\alpha, T_\beta)$. In practice, we will abbreviate the notation for the induced Killing form by writing $(\alpha, \beta) = \tilde{B}(\alpha, \beta)$.

Given this system of roots, we want the decomposition $\Delta = \Delta^+ \cup \Delta^-$ so that the following conditions are satisfied:

1. For every $\alpha \in \Delta$, either $a \in \Delta^+$ or $-\alpha \in \Delta^+$.
2. For any two distinct $\alpha, \beta \in \Delta^+$ such that $\alpha + \beta$ is also a root, $\alpha + \beta \in \Delta^+$.

Definition 3.8. The set Δ^+ are called the **positive roots**, and the set $\Delta^- = -\Delta^+$ are called the **negative roots**. A positive root is called **simple** if it cannot be written as a sum of positive roots with positive coefficients. The set of all simple roots is denoted by Δ^0.

To determine the positive roots in Δ, consider a linear map $P : \Lambda_\Delta \to \mathbb{R}$ such that $ker(P) = 0$. This means that $P(x) = 0$ if and only if $x = 0$. Because zero is not a root,

by convention, this means that $P(\alpha) > 0$ or $P(\alpha) < 0$. Let $\Delta^+ = \{\alpha \in \Delta \mid P(\alpha) > 0\}$ and $\Delta^- = \{\alpha \in \Delta \mid P(\alpha) < 0\}$. Once the positive roots have been chosen, we can systematically obtain the simple roots. It is useful to note that $\dim(\mathfrak{h}) = \dim(\mathfrak{h}^*) = |\Delta^0|$.

So far, all of the calculations we have made are inherent properties of the algebra itself. Now we are ready to discuss properties of the representation V of the algebra $\mathfrak{g}$.

Because the action of the elements from the Cartan subalgebra are simultaneously diagonalizable, this means that the vector space V admits a decomposition

$$V = \bigoplus_{\alpha} V_\alpha \tag{3.8}$$

where α is an eigenvalue of the action of $\mathfrak{h}$, or a **weight**, and V_α is the corresponding weight space.

Similar to the analysis given in the previous section, once we have a root space decomposition of the Lie algebra, we will be interested in see how the action of the root spaces affects the weight vectors.

Let $X \in \mathfrak{g}_\alpha$, $v \in V_\beta$ and $H \in \mathfrak{h}$ be given. Then

$$\begin{aligned} H \cdot (X \cdot v) &= X \cdot (H \cdot v) + [H, X] \cdot v \\ &= X \cdot (\beta(H)v) + (\alpha(H)X) \cdot v \\ &= (\alpha(H) + \beta(H))X \cdot v \end{aligned}$$

Therefore, we see that $X \cdot v$ is a weight vector for the action of $\mathfrak{h}$ with weight $\alpha + \beta$. In other words, we have that $\mathfrak{g}_a : V_\beta \to V_{\alpha+\beta}$.

Now we are ready to consider the following theorem.

Theorem 3.5. *For any semisimple complex Lie algebra $\mathfrak{g}$, every finite-dimensional representation V of $\mathfrak{g}$ possesses a highest weight vector.*

Proof: Let V be a finite-dimensional representation of a complex semisimple Lie algebra $\mathfrak{g}$. Let $\mathfrak{h}$ be a Cartan subalgebra. Then there is a set of roots Δ such that $\mathfrak{g}$ admits a root space decomposition. Let $P : \mathfrak{h}^* \to \mathbb{R}$ be a linear functional which splits the roots Δ into sets of positive and negative roots. This means that $\Delta^+ = \{\alpha \in \Delta \mid P(\alpha) > 0\}$.

Now, let $\lambda \in \mathfrak{h}^*$ be a weight of V such that $P(\lambda)$ is maximal, i.e. $P(\lambda) \geq P(\mu)$ for every weight μ of V. Then pick a vector $v \in V_\lambda$. Then, for any $\alpha \in \Delta^+$, we know that the action of $\mathfrak{g}_\alpha$ takes V_λ to $V_{\lambda+\alpha}$. However, by the maximality of λ we know that $\lambda + \alpha$ is not a weight of V, so $V_{\lambda+\alpha} = (0)$. So, for any $x \in \mathfrak{g}_\alpha$, $x \cdot v = 0$. Thus, $v \in ker(\mathfrak{g}_\alpha)$ for any $\alpha \in \Delta^+$.

Thus, we see that v is a weight vector for the action of $\mathfrak{h}$ and it is in the kernel of the action of the positive root spaces. This makes $v \in V$ a highest weight vector with highest weight λ. □

Now that we have proven that every finite-dimensional representation of $\mathfrak{g}$ has a highest weight vector, we are now in a position to refine the conditions for determining a highest weight vector. Recall that every positive root is a linear combination of the simple roots. This seems to suggest that we need only consider the simple roots when finding the highest weight vector. To prove this, we need the following definition and lemma.

Definition 3.9. Let $\Delta^0 = \{\alpha^1, ..., \alpha^\ell\}$ be a set of simple roots and let $\beta = m_i\alpha^i$ be a root. The **level** of the root $\beta = m_i\alpha^i$ is the sum of the coefficients, namely $\ell(\beta) = \sum_{i=1}^{\ell} m_i$.

By the definition of the simple roots, we know that for every positive root $\beta \in \Delta^+$, $\beta = b_i\alpha^i$ where $b_i \geq 0$ and $b_k \neq 0$ for at least one k. This means that $\ell(\beta) > 0$ for every positive root. Likewise, we also see that $\ell(\gamma) < 0$ for every $\gamma \in \Delta^-$.

Lemma 3.3. *Every positive root space is generated by the action of the simple root vectors by the adjoint representation.*

Proof: The proof of this lemma will proceed by induction on the level of the positive roots. Let Δ^+ be a set of positive roots and $\Delta^0 = \{\alpha^1, ..., \alpha^m\}$ be a set of simple roots.

Case: $\ell(\beta) = 1$.

Let $\beta \in \Delta^+$ be given such that $\ell(\beta) = 1$. This means that $\beta = \alpha^i$ for some i, so β is a simple root. So, $\mathfrak{g}_\beta$ is the root space of a simple root.

Case: $\ell(\beta) = 2$.

Let $\beta \in \Delta^+$ be a root such that $\ell(\beta) = 2$. Then $\beta = \alpha^i + \alpha^j$ for some i and j. Let $x \in \mathfrak{g}_{\alpha^i}$ and $y \in \mathfrak{g}_{\alpha^j}$ be nonzero positive root vectors. Since $\beta = \alpha^i + \alpha^j \in \Delta^+$, we know that $[x, y] \neq 0$. By Lemma 3.1 we know that $[x, y] \in \mathfrak{g}_{\alpha^i + \alpha^j} = \mathfrak{g}_\beta$. Since $dim(\mathfrak{g}_\beta) = 1$, we see that $x \cdot y = [x, y] \in \mathfrak{g}_\beta$ generates the whole root space.

Case: $\ell(\beta) = k$.

Suppose that for every positive root $\beta \in \Delta^+$ with level $\ell(\beta) = k$, the roots space $\mathfrak{g}_\beta$ is generated by the action of the simple root spaces.

Now consider a positive root $\gamma \in \Delta^+$ such that $\ell(\gamma) = k + 1$. This means that $\gamma = \beta + \alpha_i$ for some simple root $\alpha_i \in \Delta^0$ and positive root $\beta \in \Delta^+$ with $\ell(\beta) = k$.

Letting $y \in \mathfrak{g}_\beta$ and $x \in \mathfrak{g}_{\alpha_i}$ be nonzero vectors, we see that $[x, y] \neq 0$ (because γ is a root) and $[x, y] \in \mathfrak{g}_\gamma$. Finally, the dimension of $\mathfrak{g}_\gamma$ allows us to see that $\mathfrak{g}_\gamma$ is generated by the action of the simple root vector x on the positive root vector y. By the induction hypothesis, we know that y is generated by the action of the simple root vectors. Therefore, $[x, y] \in \mathfrak{g}_\gamma$ is generated by the action of the simple root vectors, thus proving our claim. □

Proposition 3.8. *A nonzero vector $v \in V$ is a highest weight vector of V if and only if v is an eigenvector for the action of $\mathfrak{h}$ and is in the kernel of the action of all the simple root vectors.*

Proof: Let $\Delta^0 = \{\alpha^i\}$ be a set of simple roots and suppose that $v \in V$ is a highest weight vector of V. By definition, this means that v is an eigenvector for the action of $\mathfrak{h}$. We also know that $v \in ker(\mathfrak{g}_\alpha)$ where $\alpha \in \Delta^+$. However, since $\Delta^0 \subset \Delta^+$, this means that $v \in ker(\mathfrak{g}_\alpha)$ for every $\alpha \in \Delta^0$.

Next, suppose that $v \in V$ is an eigenvector for the action of $\mathfrak{h}$ and that $\mathfrak{g}_\alpha(v) = 0$ for every simple root $\alpha \in \Delta^0$. Let $\beta \in \Delta^+$ be given.

If $\beta \in \Delta^0$ then we're done. Therefore, suppose that $\beta \notin \Delta^0$. For each α^i, we know that $\mathfrak{g}_{\alpha^i}$ is a one-dimensional subspace. Therefore, let $x_i \in \mathfrak{g}_{\alpha^i}$ be a basis vector of the root space. We know that every positive root can be expressed as a nonnegative integer

sum of simple roots, namely that $\beta = a_i\alpha^i$ where $\alpha^i \in \Delta^0$ and $a_i \in \mathbb{Z}^+$. Note that for every $\alpha^i \in \Delta^0$, $x_i \cdot v = 0$ where $x_i \in \mathfrak{g}_{\alpha^i}$.

By Lemma 3.3, $y \in \mathfrak{g}_\beta$ can be generated by action of the basis vectors $x_i \in \mathfrak{g}_{\alpha^i}$ where $\alpha^i \in \Delta^0$. We also see that the adjoint action of any of these basis vectors on v is zero because $[x_i, x_j] \cdot v = x_i \cdot (x_j \cdot v) - x_j \cdot (x_i \cdot v) = x_i \cdot (0) - x_j \cdot (0) = 0$. Therefore, we have that $y(v) = 0$, so $v \in ker(\mathfrak{g}_\beta)$. Combined with the fact that v is still an eigenvector for the action of $\mathfrak{h}$, we conclude that $v \in V$ is a highest weight vector of V. □

From a computational perspective, to find the highest weight vector, we need only look at the kernel of the action of the vectors from the simple root spaces. This allowing us to significantly reduce the number of computations required. Furthermore, now that we have shown that the highest weight vector must exist, we know that the simultaneous kernel of the actions of the simple roots vectors must be nonempty. Therefore, we can find the set of vectors in the kernel of each map. Then, we determine linear combinations of these vector so that they form eigenvectors of the action of the Cartan subalgebra. Such vectors will be highest weight vectors of the representation.

Now that we have identified that every finite-dimensional representation admits a highest weight vector, and we have developed a more efficient means of obtaining one, we can now use the highest weight vectors to obtain the irreducible subrepresentations of V and classify them.

Theorem 3.6. *Let $\mathfrak{g}$ be a semisimple complex Lie algebra with a finite-dimensional representation V.*

i The subspace W of V generated by the images of a highest weight vector v under the successive applications of root spaces $\mathfrak{g}_\beta$ for $\beta \in \Delta^-$ is an irreducible subrepresentation of V;

ii An irreducible representation possesses a unique highest weight vector up to scalars.

Proof:

(i) Let $v \in V$ be a highest weight vector with weight λ. For any negative root $\beta \in \Delta^-$, denote an element from the root space by $Y_\beta \in \mathfrak{g}_\beta$.

Let W_n be the subspace spanned by all elements of the form $w_n \cdot v$ where w_n is a word of length at most n, or in other words, $w_n = Y_{\beta^n} \cdot Y_{\beta^{n-1}} \cdot ... \cdot Y_{\beta^1}$ where $\beta^k \in \Delta^-$. By definition, notice that $W_0 = V_\lambda$.

Claim: For any positive root vector X, $X \cdot W_n \subset W_n$. We will prove this claim by induction on n.

Case $n = 1$: Let $w_1 \in W_1$ be given. Notice that we can write this as $w_1 = Y \cdot v$ for some negative root vector Y. Then consider the following calculation:

$$\begin{aligned} X \cdot w_1 &= X \cdot (Y \cdot v) \\ &= Y \cdot (X \cdot v) + [X, Y] \cdot v. \end{aligned}$$

Because v is in the kernel of the action of the positive root vectors, this means that $X \cdot v = 0$. We also know that $[X, Y] \in \mathfrak{h}$ and that v is a weight vector of the action of $\mathfrak{h}$. Therefore, we see that

$$\begin{aligned} X \cdot w_1 &= [X, Y] \cdot v \\ &= \alpha([X, Y])v. \end{aligned}$$

Therefore, $X \cdot w_1 \in W_0 \subset W_1$.

Now, suppose that $X \cdot w \in W_{n-1}$ for every positive root vector X and every vector $w \in W_{n-1}$. Then, note that for any negative root vector Y, $Y \cdot w \in W_n$ for all $w \in W_{n-1}$. Then, from the following calculation,

$$X \cdot (Y \cdot w) = Y \cdot (X \cdot w) + [X, Y] \cdot w$$

we see that $Y \cdot (X \cdot w) \in W_n$ by the induction hypothesis, and that $[X, Y] \cdot w \in W_{n-1}$. Therefore, we see that $X \cdot (Y \cdot w) \in W_n$. Therefore, we conclude that $X \cdot W_n \subset W_n$ for any positive root vector X.

Now consider the union of all the subspaces W_n, and call it W. Clearly, $W \subset V$ is a vector subspace. We also have that $X \cdot W \subset W$ for every positive root vector X. We also have that $Y \cdot W \subset W$ for every negative root vector Y by the definition of

the subspaces W_n. Furthermore, because $[X, Y] \in \mathfrak{h}$, we have that $[X, Y] \cdot W \subset W$. Therefore, $W \subset V$ is a subrepresentation of V.

To show that W is an irreducible subrepresentation of V, suppose that W admits a decomposition $W = W' \oplus W''$. Note that one of the subrepresentations must contain the weight space V_λ. Suppose, without loss of generality, that $V_\lambda \subset W'$. Then, by the repeated actions of the negative root vectors on V_λ, we see that $W' = W$ and $W'' = (0)$. Therefore, the only subrepresentations of W are W and (0), making W an irreducible representation of V.

(ii) Suppose now that V is an irreducible representation of $\mathfrak{g}$. To prove that the highest weight vectors of an irreducible representation are unique, we need to consider the linear functional P which was used to obtain the splitting of the roots, namely the function P such that $\Delta^+ = \{\alpha \in \Delta \mid P(\alpha) > 0\}$. Then, we choose the highest weight λ to be a weight such that $P(\lambda)$ is maximal. This tells us that a highest weight vector v is contained in V_λ.

Suppose that there exists another highest weight vector $u \in V_\mu$, not a scalar multiple of v. This means that $P(\mu) \geq P(\lambda)$. By the maximality of λ, this means that $P(\lambda) = P(\mu)$. However, this can only happen if $\lambda = \mu$. Thus, $v, u \in V_\lambda$. Now all we have to do is show that $dim(V_\lambda) = 1$. □

By Theorem 3.5, we know that every finite-dimensional representation V has a highest weight vector. If V is irreducible, then we know exactly what the decomposition of V will be. Therefore, suppose that V is not irreducible. We know that there will be a highest weight vector $v \in V$. By part 1 of Theorem 3.6, we also know that the subspace W of V generated by the action of the negative root spaces on v will be an irreducible representation.

Furthermore, given a highest weight vector $v_1 \in V$, we can uniquely generate an irreducible subrepresentation $V_1 \subset V$. With this irreducible subrepresentation, and by Theorem 3.4, we know that there exists a complementary subrepresentation $W \subset V$ such that $V = V_1 \oplus W$, where W may or may not be irreducible.

Therefore, we can go again and decompose W using the theorem. This will give us another irreducible subrepresentation V_2 and a subrepresentation W' which is complementary to V_1 and V_2, making $V = V_1 \oplus V_2 \oplus W'$. We can continue to decompose the complementary subrepresentations created with each application of the theorem. Because V is finite-dimensional, we know that this process must terminate eventually. Therefore, this theorem leads us to the first fundamental theorem that was given, namely, that every finite-dimensional representation admits a decomposition into a finite direct sum of irreducible subrepresentations.

Notice that for each subrepresentation $W \subset V$. If $v \in W$ is a highest weight vector of W, then $v \in V$ is also a highest weight vector of V. So, in practice, we will want to find all the highest weight vectors of a representation. If V is irreducible, then there will be only one such vector. If there are n highest weight vectors, then we know that there will be n irreducible subrepresentations in the decomposition of V.

Now that we have decomposed the representation V into a direct sum of irreducible representations, we will now focus on a means of identifying what the irreducible representations are. Because the highest weight vector of an irreducible representation is unique, an irreducible representation is determined by its highest weight vector. However, a highest weight vector will certainly depend on a choice of basis of the vector space V. So, the highest weight vector is not a choice candidate for classifying the representation. Recalling that eigenvalues of a linear transformation are an invariant property of the linear transformation, we can conclude that the highest weight will not depend on a basis of V, and hence is a candidate for classifying the representation.

It is important to note that in finding the highest weight λ of an irreducible representation, we had to specify a splitting of the roots, and the roots were obtained by a choice of $\mathfrak{h}$. In short, this means that λ depends entirely on our set of simple roots Δ^0. So, although the highest weight is independent of any choice of basis for the vector space V, it *does* depend on our choice of properties from $\mathfrak{g}$. We would like to find a way of representing the highest weight λ in way which is also independent of any choices of $\mathfrak{h}$ and Δ^0. To accomplish this, we must use an invariant property of $\mathfrak{g}$ called the Cartan matrix.

Definition 3.10. Let $B(\cdot,\cdot)$ be the Killing form on $\mathfrak{g}$, $\Delta^0 = \{\alpha^1, ..., \alpha^\ell\}$ be a set of simple roots and $\{H_1, ..., H_\ell\}$ the corresponding set of coroots. The **Cartan matrix**

corresponding to Δ^0 is the square matrix C^i_j whose entries are given by

$$C^i_j = 2\frac{B(H_j, H_i)}{B(H_i, H_i)}.$$

The Cartan matrix is independent of the Cartan subalgebra $\mathfrak{h}$, but *is* dependent on the (although equivalent to another) ordering of the simple roots. Therefore, some care will be required in ensuring that the Cartan matrix used is consistent with the given ordering of the simple roots. To make sure all our calculations are consistent, we choose an ordering convention. By using the **standard form** of the Cartan matrix, we can fix the ordering of the simple roots.

If the calculated Cartan matrix C is not in the standard form, then the simple roots are permuted so that the matrix created using that ordering of the simple is the Cartan matrix in standard form. We will call this ordering of the simple roots the **standard ordering**.

Definition 3.11. Let $\Delta^0 = \{\alpha^1, ..., \alpha^\ell\}$ be a set of simple roots. The **fundamental weights** are the set of linear functionals in $\mathfrak{h}^*$, $\Omega = \{\omega_1, ..., \omega_\ell\}$, which form a basis of $\mathfrak{h}^*$ which is dual to the basis $\{H_1, ..., H_\ell\}$ of $\mathfrak{h}$. Therefore, $\omega^i(H_j) = \delta^i_j$.

Let $\{\alpha_1, ..., \alpha_\ell\} = \Delta^0$ be a set of simple roots. From the root space decomposition, we know that we also have a set $\{X_i, Y_i, H_i \mid i = 1...\ell\}$ where $[X_i, Y_i] = H_i$ and $\alpha_i(H_i) = 2$ where $X_i \in \mathfrak{g}_{\alpha_i}$, $Y_i \in \mathfrak{g}_{-\alpha_i}$ and $H_i \in [\mathfrak{g}_\alpha, \mathfrak{g}_{-\alpha_i}]$. This means that $\alpha^i(H_j) = C^j_i$ is the (j, i)-entry of the Cartan matrix.

The following is a brief discussion on how the fundamental roots are computed from a given set of simple roots.

Let H_j be the coroot of a simple root $\alpha^j \in \Delta^0$ and let ω^i be a fundamental root. Because $\omega^i \in \mathfrak{h}^*$, this means that there is a change-of-basis matrix A such that $\omega^i = A^i_j \alpha^j$. Similarly, this also means that $\tilde{A}^j_i \omega^i = \alpha^j$ where $\tilde{A} = A^{-1}$. Since $\alpha^j(H_k) = C^k_j$

for each H_k, we have that

$$\begin{aligned} C_j^k &= \alpha^j(H_k) \\ &= \tilde{A}_i^j \omega^i(H_k) \\ &= \tilde{A}_i^j \delta_k^i \\ &= \tilde{A}_j^k. \end{aligned}$$

Therefore, we have that $A^{-1} = C$ where C is the Cartan matrix. Thus, we can compute the fundamental weights by using the formula:

$$\omega^i = (C^{-1})_j^i \alpha^j$$

It is important to recognize a few of the properties of Ω. First of all, since Δ^0 is a basis of $\mathfrak{h}^*$, Ω is also a basis of $\mathfrak{h}^*$. Furthermore, given any set of simple roots, we obtain a basis of $\mathfrak{h}$ via the corresponding set of coroots. We then use this set of coroots to create a dual basis of $\mathfrak{h}^*$. Therefore, it can be shown that the set of fundamental weights does not depend on the choice of simple roots. Therefore, Ω is an inherent property of $\mathfrak{g}$. This makes Ω a good basis for representing the weights of a representation.

Let $\lambda = \tilde{a}_k \alpha^k$ be a highest weight in the basis of the simple roots. Recalling that $\alpha^k = C_i^k \omega^i$, we have that

$$\begin{aligned} \lambda &= \tilde{a}_k \alpha^k \\ &= (\tilde{a}_k C_i^k) \omega^i \\ &= a_i \omega^i \end{aligned}$$

We will not show it here, but it can be shown that each of the coefficients are non-negative, or $\tilde{a}_k C_i^k = a_i \geq 0$ for each $1 \leq i \leq \ell$. Again, these integers do not depend on the choice of a Cartan subalgebra or the choice of the simple roots. The ℓ-tuple only depends on the ordering imposed on Ω.

However, because the Cartan matrix is an invariant property of $\mathfrak{g}$, the standard ordering of the simple roots will not depend on the set of simple roots. Furthermore, by

invoking the standard ordering of the simple roots, we will likewise have an ordering of the fundamental weights.

Therefore, using the standard ordering of the simple roots, we obtain a standard ordering of the fundamental weights. Then, when written in the basis of fundamental weights, λ is a unique property of the irreducible representation V which is independent of $\mathfrak{h}$, Δ^0, or any choice of basis of V. Thus, λ is an invariant property of V, allowing us to uniquely characterize the irreducible representation.

The irreducible representations of $\mathfrak{g}$ are identified by $\Gamma_{a_1,\ldots,a_\ell}$ where a_i are the coefficients of the highest weight when written in the basis Ω.

3.8 Examples

For each of the examples, the following packages will be needed. Also, we will denote the standard representation $\Gamma_{1,0,\ldots} = V$.

```
> with(DifferentialGeometry):
> with(LieAlgebras):
> with(Tools):
> with(Tensor):
```

3.8.1 $\mathfrak{sl}_3(\mathbb{C})$: Decomposing $V \otimes V$

To begin with, we will start with an example that is very well known, the tensor product of the standard representation of $\mathfrak{sl}_3(\mathbb{C})$. However, to illustrate the capability of the algorithm, we will change things up a bit. To start with, we need to read in and initialize the structure constants for the algebra.

```
> LD := SimpleLieAlgebraData( "sl(3)" , sl3):
> DGsetup(LD);
```

Lie algebra: sl3

To get the standard representation, we can use the command `StandardRepresentation`.

```
> StdRep := StandardRepresentation(sl3);
```

Now, to change things up a bit, let's use the following matrix to do a change-of-basis transformation on the representation.

```
> Q := <<-3,-2,3>|<-1,-4,-5>|<3,5,1>>;
```

$$Q := \begin{bmatrix} -3 & -1 & 3 \\ -2 & -4 & 5 \\ 3 & -5 & 1 \end{bmatrix}$$

```
> Qi := Q^(-1):
> NewRep := map( m->Qi.m.Q , StdRep):
```

Now that we have an interesting representation ready, let initialize a frame for the representation space and then create the representation, and call it `rho`.

```
> DGsetup([seq(v||i,i=1..3)], V);
```

frame name: V

```
> rho := Representation( sl3 , V , NewRep ):
```

Now that we have the standard representation ready, let's compute the tensor product. We know that the representation space of the tensor product will be 9-dimensional.

```
> DGsetup([seq(u||i,i=1..9)] , U):
```

frame name: U

Now we compute the tensor product using the command `TensorProductOfRepresentations`.

```
> phi := TensorProductOfRepresentations( [rho,rho] , U):
```

To see what kind of objects we are dealing with, consider the following object:

```
> ApplyRepresentation( phi , e1 );
```

$$\begin{bmatrix} 12 & -1 & -4 & -1 & 0 & 0 & -4 & 0 & 0 \\ \frac{39}{7} & 4 & -3 & 0 & -1 & 0 & 0 & -4 & 0 \\ \frac{48}{7} & -2 & 2 & 0 & 0 & -1 & 0 & 0 & -4 \\ \frac{39}{7} & 0 & 0 & 4 & -1 & -4 & -3 & 0 & 0 \\ 0 & \frac{39}{7} & 0 & \frac{39}{7} & -4 & -3 & 0 & -3 & 0 \\ 0 & 0 & \frac{39}{7} & \frac{48}{7} & -2 & -6 & 0 & 0 & -3 \\ \frac{48}{7} & 0 & 0 & -2 & 0 & 0 & 2 & -1 & -4 \\ 0 & \frac{48}{7} & 0 & 0 & -2 & 0 & \frac{39}{7} & -6 & -3 \\ 0 & 0 & \frac{48}{7} & 0 & 0 & -2 & \frac{48}{7} & -2 & -8 \end{bmatrix}$$

This certainly looks bad. Maybe not the worst thing conceivable, but certainly not great. The other matrices are likewise formed. The biggest to consider is that this matrix is not immediately recognizable as a diagonalizable matrix.

To compute the decomposition, let's obtain a Cartan subalgebra.

```
> CSA := CartanSubalgebra(sl3);
```

$$CSA := [e1,\ e2]$$

We are now ready to decompose the representation `phi`. As part of the decomposition, we can also compute a change-of-basis matrix that will change the representations of the basis elements of $\mathfrak{sl}_3(\mathbb{C})$ such that the matrix representations are diagonal block sums of irreducible representations and the matrix representations of the given Cartan subalgebra are diagonal matrices.

```
> Marks,R := DecomposeRepresentation(phi , CSA ,
                                     output=["Marks","Transform"],
                                     print=true ):
```

$$\left[\ \Gamma_{2,0} \ \oplus \ \Gamma_{0,1}\ \right]$$

We see that the tensor product of the standard representation is isomorphic to $\Gamma_{2,0} \oplus \Gamma_{0,1}$. Furthermore, we can use the change-of-basis transformation matrix R on a few of the

representations to see what kind of matrices we are working with. Lines have been included to illustrate the blocks.

```
> Ri := R^(-1):
> Ri.ApplyRepresentation(phi,e1).R,
  Ri.ApplyRepresentation(phi,e4).R;
```

$$\left[\begin{array}{cccccc|ccc} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{array}\right], \left[\begin{array}{cccccc|ccc} 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

3.8.2 $\mathfrak{g}_2(\mathbb{C})$: Decomposing $\wedge^2(\mathfrak{der}(\mathbb{O}))$

In this example, we will construct the complex Lie algebra $\mathfrak{g}_2$ by using the representation $\mathfrak{der}(\mathbb{O})$. First, we read in the product rules for the Octonions by using `AlgebraLibraryData` and we call the frame `O`. Then initialize the algebra.

```
> OctData := AlgebraLibraryData("Octonions",O):
> DGsetup(OctData);
```

algebra name: O

Then, we compute a basis of the derivations algebra on the octonions.

```
> DerO := Derivations(O):
```

Now that we have matrices forming a basis of $Der(\mathbb{O})$, we can use these matrices to construct the Lie algebra.

```
> LData := LieAlgebraData(Der0,g2):
> DGsetup(LData);
```

Lie algebra: g2

We have used the matrices to construct the bracket rules for the Lie algebra $\mathfrak{g}_2$, but now we want to use those matrices as a representation of $\mathfrak{g}_2$. To do this, we must first initialize a frame for the representation space. Note that $\dim(Der(\mathbb{O})) = 8$, so

```
> DGsetup([u1,u2,u3,u4,u5,u6,u7,u8] , U);
```

Next, we create the representation and call it `rho`.

```
> rho := Representation(g2, U, Der0):
```

One of the keys to decomposing a representation is the Cartan subalgebra. In the end, although the Cartan subalgebra does not affect the highest weights identifying the irreducible representations, it does have an effect on several of the intermediate steps. The next thing we want to do then, is compute a Cartan subalgebra.

```
> CSA := CartanSubalgebra(g2);
```

CSA := [e1, e4 + e10]

While the algorithms will certainly work for such a Cartan subalgebra, our results will be nicer if we had a better basis to work with. One such basis is a Chevelley basis. To compute a Chevalley basis, we first must find a root space decomposition.

```
> RSD := RootSpaceDecomposition(CSA):
```

From the root space decomposition, we can choose a set of positive roots.

```
> PR := PositiveRoots(RSD);
```

$$\left[\begin{bmatrix} 2I \\ 0 \end{bmatrix}, \begin{bmatrix} I \\ 3I \end{bmatrix}, \begin{bmatrix} I \\ -I \end{bmatrix}, \begin{bmatrix} I \\ I \end{bmatrix}, \begin{bmatrix} I \\ -3I \end{bmatrix}, \begin{bmatrix} 0 \\ 2I \end{bmatrix}\right]$$

Once we have a Cartan subalgebra, a corresponding root space decomposition, and a set of positive roots, we can compute a Chevalley basis.

```
> CB := ChevalleyBasis(CSA, RSD, PR):
```

We are going to want to work with the representation using the Chevalley basis. We can perform the change of basis transformation on the representation using the command `ChangeRepresentationBasis`. First we specify the representation, then the new basis that we wish to use, and finally the frame identifying the space that we are transforming (in this case it is the domain space).

```
Rho := ChangeRepresentationBasis( rho, CB, g2):
```

In order to use this representation, we need to initialize a Lie algebra using the Chevalley basis. Because we will no longer be interested in using the old basis, we can use the same frame name.

```
> DGsetup(LieAlgebraData(CB, g2));
```

Lie algebra: g2

Now that we have applied the change of basis transformation to the Lie algebra, we need to recompute a Cartan subalgebra and a root space decomposition.

```
> CSA := CartanSubalgebra(g2):
> RSD := RootSpaceDecomposition(CSA):
```

We now have a good basis for the domain of the representation $\mathfrak{der}(\mathbb{O})$. At this point we want to use this representation to create a new representation.

Now we want to take the wedge product of the representation. We do this by working with the representation space. First we take the tensor product of each of the basis vectors. Then we skew-symmetrize the indices.

```
> UBasis := DGinfo(U, "FrameBaseVectors"):
> T := GenerateTensors([ UBasis , UBasis ]):
> T := map( SymmetrizeIndices , T , [1,2], "SkewSymmetric"):
```

Once we have done this, we can use the command `DGbasis` to create a set of basis vectors.

```
> T := DGbasis(T):
```

We can also see what the dimension of the representation space is.

```
> nops(T);
```

28

Once, we have taken the wedge product of the representation space, we now need to initialize a new frame for the representation space of the wedge product.

```
> DGsetup([seq(v||k, k=1..28)] , V);
```

frame name: V

To determine how the matrices are transformed by the wedge product, we use the command `TensorProductOfRepresentations`.

```
> phi := TensorProductOfRepresentations(Rho, T, V):
```

The representation `phi` is a matrix representation of $\wedge^2(\mathfrak{der}(\mathbb{O}))$ using the Chevalley basis of $\mathfrak{der}(\mathbb{O})$. We are now ready to decompose this representation. One of the available options is to create a change-of-basis transformation matrix that can be used to transform `phi` into diagonal block sums of irreducible representations. We would also like to see what the decomposition looks like, so we will include the option to "print."

```
> Q := DecomposeRepresentation(phi,output="Transform",
                                        print=true):
```

$$[\ 2\Gamma_{1,0} \oplus \Gamma_{0,1}\ ]$$

One of the nice properties of the transformation matrix `Q`, is that after transforming the representations of the basis elements, the matrices corresponding to the Cartan subalgebra are diagonal, the matrices corresponding to the positive root spaces are upper triangular, and the matrices corresponding to the negative root spaces are lower triangular *in addition to* all of the matrices being a block sum of matrices. In short, this means that `Q` gives us a basis of the representation space so that the matrix representations are as simple and well-behaved as can be expected.

Before we dive into the transformation, let's first address the fundamental representations of $\mathfrak{g}_2$. Because $\mathfrak{g}_2$ is rank 2, we know that there are only two. They are identified by $\Gamma_{1,0}$ and $\Gamma_{0,1}$. The dimension of $\Gamma_{1,0}$ is 7, and it corresponds to the action of $\mathfrak{g}_2$ on the imaginary octonions. The second fundamental representation, $\Gamma_{0,1}$, is the adjoint representation and is 14-dimensional. In particular, while the adjoint representation is relatively simple to compute, the representations of the Cartan subalgebra need not be diagonal, and certainly the representations of the positive (or negative) root spaces need not be triangular.

However, after our decomposition, we can easily extract the irreducible representations $\Gamma_{1,0}$ and $\Gamma_{0,1}$ that *do* have these properties. To do so, we again use the command `ChangeRepresentationBasis`.

```
> Qi := Q^(-1):
> Basis := DGinfo(g2, "FrameBaseVectors"):
> TransformRep := map(x->Qi.ApplyRepresentation(phi,x).Q,Basis):
```

Now, let's extract the irreducible representations (note that $\Gamma_{1,0}$ appears twice in the decomposition, so we can skip the second occurrence).

```
> F1Rep := map(x->x[1..7,1..7], TransformRep ):
> F2Rep := map(x->x[15..28,15..28], TransformRep ):
```

Now that we have the matrices corresponding to the two irreducible representations, let's go back to the Lie algebra g_2 and extract some of the properties of the algebra.

We compute a Cartan subalgebra (using the Chevalley basis now), then a root space decomposition followed by a set of positive roots. Once we have the positive roots, we can use the look-up table for the root space decomposition, `RSD`, to find the positive root spaces.

```
> CSA := CartanSubalgebra(G2):
> RSD := RootSpaceDecomposition(CSA):
> PR  := PositiveRoots(RSD):
> PRS := map(x->RSD[convert(x,list)], PR);
```

$$PRS := [e3, e6, e8, e7, e10, e11]$$

Now, we can compute and display the adjoint representation of one of these elements and compare it to the corresponding representation we computed using the decomposition. (To save on space, the full output will not be printed, but will be shown using the symbols E_i^j instead).

```
> Adjoint(e3);
```

$$-E_1^9 - 2E_3^1 + E_3^2 - E_5^4 - 3E_7^8 - 2E_8^5 + 3E_{10}^{11} + 2E_{11}^{14} + E_{14}^{13}$$

```
> F2Rep[3];
```

$$-6E_1^4 - 6E_2^6 + 2E_3^9 + 4E_4^{10} + 12E_6^{11} - 12E_7^{12} + 24E_9^{12} - 18E_{10}^{13} - 6E_{11}^{14}$$

Here, we clearly see that the adjoint representation of $e3$ is not triangular, and the representation we extracted is. But we see that the two matrices are similar

```
> LinearAlgebra:-IsSimilar(Adjoint(e3) , F2Rep[3]);
```

$$true$$

3.8.3 $\mathfrak{so}_5(\mathbb{C})$: Creating an "Optimal" Basis of V.

For this example, we will show how one can use the highest weight vector of a representation to create a new basis of the vector space. This new basis will have the property that the representations of the cartan subalgebra will be strictly diagonal, the representations of the positive root vectors will be strictly upper triangular, and the negative root vectors will be strictly lower triangular. To illustrate this, we will use the standard representation of $\mathfrak{so}_5(\mathbb{C})$.

First, we read in the structure constants for the Lie algebra. Then we can initialize the frame.

```
> LD := SimpleLieAlgebraData("so(5)",B):
> DGsetup(LD);
```

Lie algebra: B

Now, let's create the standard representation of the Lie algebra.

```
> DGsetup([V1,V2,V3,V4,V5)],V);
```

frame name: V

```
> rho := Representation(B,V, StandardRepresentation(B) ):
```

In the last example, we used a Chevalley basis to create a basis for the Lie algebra that had some very nice properties. We would like to do the same here. Then we can look at the standard representation after the change of basis of the Lie algebra.

To compute the Chevalley basis, we need a Cartan subalgebra, the corresponding root space decomposition, and a choice of positive roots.

```
> CSA := CartanSubalgebra(B);
```

CSA:=[e1,e8]

```
> RSD := RootSpaceDecomposition(CSA);
```

$$RSD:=table([[-2,2]=e7,\ [-1,0]=e9,\ [-1,2]=e4,\ [1,-2]=e8,\ [0,2]=e10,\ [1,0]=e5,\ [0,2]=e6,\ [2,-2]=e3])$$

```
> PR := PositiveRoots(RSD);
```

$$PR:=\left[\left[\begin{bmatrix} 1 \\ -2 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ -2 \end{bmatrix}\right]\right]$$

Next we use the Maple command to create a Chevalley basis.

```
> CB := ChevalleyBasis( CSA, RSD, PR);
```

$$CB:=\Big[\ -I(e1-e8),\ -2Ie8,\ -\frac{1}{4}(e2+e6+Ie3-Ie5),\ -e9+Ie10,\ \frac{1}{2}(-e4+Ie7),\ \frac{1}{4}(-e2+e6+Ie3+Ie5),\ -e2-e6+Ie3-Ie5,\ -e9-Ie10,\ -2(e4+Ie7),\ -e2+e6-Ie3-Ie5\ \Big]$$

Now, using this basis, we use the command `ChangeRepresentationBasis` to change the matrix representations of the basis elements of $\mathfrak{so}_5(\mathbb{C})$ to the matrix representations of the elements in the Chevalley basis.

```
> tau := ChangeRepresentationBasis(rho,CB,B);
```

In order to continue our analysis, we need to define a Lie algebra frame using the Chevalley basis. We will simply reuse the same frame name. Then we have to recalculate the Cartan subalgebra and root space decomposition using this new basis.

```
> DGsetup( LieAlgebraData( CB, B)):
> CSA := CartanSubalgebra(B):
> RSD := RootSpaceDecomposition(CSA):
```

Now, we can compute the highest weight vector of the representation 'tau'.

```
> v := DecomposeRepresentation(tau,output="Vector");
```

$$v := \begin{bmatrix} I \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Because there is only one highest weight vector, we know that the representation is irreducible. Of course, this was to be expected because we are using the standard representation. Recall that the representation generated by the repeated actions of the negative root vectors on the highest weight vector is an irreducible representation. However, since the negative roots can be written as sums of the negative simple roots, we need only consider the action of the vectors corresponding to the negative simple roots. This begs the question of what will happen if we compute the actions of the negative root vectors on the highest weight vector we calculated.

To find out, we must first identity the positive roots. Once we have a set of positive roots, we can find the simple roots. Then, we will find the negative simple roots.

```
> PR := PositiveRoots(RSD):
> SR := SimpleRoots(PR):
> NSR := map(x->(-x), SR);
```

$$NR := \left[\begin{bmatrix} 0 \\ -2 \end{bmatrix}, \begin{bmatrix} -1 \\ 2 \end{bmatrix} \right]$$

Now that we have the negative simple roots, we can use these to extract the negative simple root vectors from the root space decomposition table.

```
> NSRV := map( r->RSD[ convert(r,list) ], NSR);
```

$$NSRV := [e10,\ e4]$$

We are going to want to compute the action of the negative root vectors on the highest weight vector. So let's see what the matrix representations of these vectors are:

```
> Y1 := ApplyRepresentation(tau,e10);
```

$$Y1 := \begin{bmatrix} 0 & 0 & 1 & I & 0 \\ 0 & 0 & I & -1 & 0 \\ -1 & -I & 0 & 0 & 0 \\ -I & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
> Y2 := ApplyRepresentation(tau,e4);
```

$$Y1 := \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -I \\ 0 & 0 & -1 & I & 0 \end{bmatrix}$$

Now, we will begin computing the actions of these two matrices on the highest weight vector. We should anticipate that we will obtain five linearly independent nonzero vectors. Although there is an algorithmic method for quickly obtaining these vectors, we will not go into that subject. We will only show what the vectors are and how they are created. The first vector we have is the highest weight vector v:

```
> v1 := v;
> u1 := v1/norm(v1):
```

The reason for normalizing the vectors will be clear in a moment. For now, it is convenient to include this operation here. The other four vectors are given below. The output display has been suppressed until the end to conserve space.

```
> v2 := Y1.v;
> u2 := v2/norm(v2):

> v3 := Y2.Y1.v;
> u3 := v3/norm(v3):
```

```
> v4 := Y2.Y2.Y1.v;
> u4 := v4/norm(v4):

> v5 := Y1.Y2.Y2.Y1.v;
> u5 := v5/norm(v5):

> [v1,v2,v3,v4,v5];
```

$$\left[\begin{bmatrix} I \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ -2I \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 4I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 4I \\ 4 \\ 0 \end{bmatrix}, \begin{bmatrix} 8I \\ -8 \\ 0 \\ 0 \\ 0 \end{bmatrix}\right]$$

It is easy to see that this set of five vectors are linearly independent. Furthermore, we can see that they span the space V. Thus they form a basis of V. Then, to make the numbers look nicer, we can normalize each of these vectors. This gives us the set of vectors $[u1, u2, u3, u4, u5]$ which can be written explicitly:

$$\left[\begin{bmatrix} I \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ -I \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} I \\ -1 \\ 0 \\ 0 \\ 0 \end{bmatrix}\right].$$

To finish this off, we want to see what the representation `tau` looks like in this basis. To see this, we can again use the command `ChangeRepresentationBasis` to change the basis of the representation space. Because `tau` was written using the standard basis, the change of basis transformation will simply be the matrix

```
Q := Matrix([u1,u2,u3,u4,u5]);
```

$$Q := \begin{bmatrix} I & 0 & 0 & 0 & I \\ 1 & 0 & 0 & 0 & -1 \\ 0 & -I & 0 & I & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & I & 0 & 0 \end{bmatrix}.$$

Then, the representation after the change of basis transformation becomes:

```
phi := ChangeRepresentationBasis(tau, Q, "Range", V):
```

The representations of the Cartan subalgebra are strictly diagonal;

```
> map[2]( ApplyRepresentation, phi, [e1,e2] );
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

the representations of the positive root vectors are strictly upper triangular;

```
> map[2]( ApplyRepresentation, phi, [e3,e5,e6,e8]);
```

$$\begin{bmatrix} 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & -\frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

and the representations of the negative root vectors are strictly lower triangular

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -2 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Therefore, given a representation of a semisimple Lie algebra, we can use the commands in Maple to change the basis of the domain space and the range space to create a matrix representation of the Lie algebra which are real valued (this property will come from the Chevalley basis transformation), and are either diagonal, strictly upper triangular, or strictly lower triangular.

Furthermore, although this example worked for an irreducible representation, a similar procedure can be followed for any representation. The only big difference is that there will be multiple highest weight vectors to work with. However, each set of vector generated by a given highest weight vector will be a basis of the subspace. So the collection of all these vectors will create a basis of the whole vector space.

CHAPTER 4
MAGIC SQUARE LIE ALGEBRAS

4.1 Introduction

In this chapter, we discuss Vinberg's construction of the Freudenthal magic square Lie algebras. The magic square is interesting because it provides a uniform construction of all five exceptional Lie algebras.

Given any normed division algebras $\mathbb{K}$, let $\mathfrak{der}\,(\mathbb{K})$ be the derivation algebra of $\mathbb{K}$ (see equation 2.7). Given any $*$-algebra A, let $\mathfrak{sa}_3(A) = \{X \in M_3(A) \mid X^* = -X,\ tr(X) = 0\}$ be the set of 3×3 trace-free skew-hermitian matrices over A. For any two normed division algebras $\mathbb{K}$ and $\mathbb{M}$, Vinberg's construction creates the following vector space

$$M(\mathbb{K},\mathbb{M}) = \mathfrak{der}\,(\mathbb{K}) \oplus \mathfrak{der}\,(\mathbb{M}) \oplus \mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M}).$$

It is rather simple to construct a basis of the vector space $M(\mathbb{K},\mathbb{M})$ by using bases of each of the algebras $\mathfrak{der}\,(\mathbb{K})$, $\mathfrak{der}\,(\mathbb{M})$ and $\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$.

A Lie bracket is defined on this vector space and will be discussed in detail later in this chapter. Computing the Lie bracket of the basis elements of $M(\mathbb{K},\mathbb{M})$ is rather difficult. The purpose of this chapter is to create a procedure for computing the Lie bracket on the basis elements of $M(\mathbb{K},\mathbb{M})$. In doing so, we are able to verify that Vinberg's construction creates Freudenthal's Magic Square of Lie algebras as given in table 4.1.

A key component in the construction of the Lie bracket on $M(\mathbb{K},\mathbb{M})$ is the map $D_{a,b} : \mathbb{K} \to \mathbb{K}$ (for any normed division algebra $\mathbb{K}$) given by [1]

$$D_{a,b}(x) = [[a,b],x] - 3(a,b,x).$$

TABLE 4.1: Freudenthal's Magic Square Lie Algebras

$\mathbb{K}/\mathbb{M}$	0	$\mathbb{R}$	$\mathbb{C}$	$\mathbb{H}$	$\mathbb{O}$
0	0	0	0	$\mathfrak{su}_2$	$\mathfrak{g}_2$
$\mathbb{R}$	0	$\mathfrak{so}_3$	$\mathfrak{su}_3$	$\mathfrak{sp}_6$	$\mathfrak{f}_4$
$\mathbb{C}$	0	$\mathfrak{su}_3$	$\mathfrak{su}_3 \oplus \mathfrak{su}_3$	$\mathfrak{su}_6$	$\mathfrak{e}_6$
$\mathbb{H}$	$\mathfrak{su}_2$	$\mathfrak{sp}_6$	$\mathfrak{su}_6$	$\mathfrak{so}_{12}$	$\mathfrak{e}_7$
$\mathbb{O}$	$\mathfrak{g}_2$	$\mathfrak{f}_4$	$\mathfrak{e}_6$	$\mathfrak{e}_7$	$\mathfrak{e}_8$

In the text on the octonions by Baez, this formula is stated to be a derivation on $\mathbb{K}$, but no proof is provided. We will prove, in a more general case, that $D_{a,b} : A \to A$, as defined above, is a derivation for any alternative algebra A. This implies $D_{a,b}$ is, in particular, a derivation for any normed division algebra as well.

We will show that for any normed division algebra, every derivation is of the form $D_{a,b}$. This allows us to use the the formula $D_{a,b}$ to create a basis of the derivation algebras $\mathfrak{der}(\mathbb{R})$, $\mathfrak{der}(\mathbb{C})$, $\mathfrak{der}(\mathbb{H})$ and $\mathfrak{der}(\mathbb{O})$.

The last section of this chapter will be a discussion of the Lie bracket on the vector space $M(\mathbb{K}, \mathbb{M})$, and how it was computed in Maple. Several examples will be given to show that the algorithm created using the bracket given by Vinberg is consistent with the table. Maple worksheets are included in appendix *C* which verify the creation each of the algebras in the magic square.

4.2 Derivations and the Derivation Algebra

In this section we will verify the formula for the mapping $D_{a,b} : \mathbb{K} \to \mathbb{K}$ as given by Baez satisfies the definition of a derivation on a normed division algebra. We will also see that this formula will allow us to create a basis of the derivation algebra. Equation (4.1) is used in computing the Lie bracket for Vinberg's construction. To begin, we define a derivation on an algebra.

Definition 4.1. Given any algebra A, a **derivation** of A is a linear map $D : A \to A$ satisfying the Leibniz property, $\forall x, y \in A$

$$D(x \cdot y) = D(x) \cdot y + x \cdot D(y).$$

For any algebra, let $[x, y] = xy - yx$ be the commutator, and $(x, y, z) = (xy)z - x(yz)$ be the associator. Then the main theorem of this section is given below.

Theorem 4.1. *Let A be any real alternative algebra. For any pair of elements $a, b \in A$, the mapping $D_{a,b} : A \to A$ defined by*

$$D_{a,b}(x) = [[a, b], x] - 3(a, b, x) \tag{4.1}$$

is a derivation of A.

Before we address the proof of this theorem, there are a few identities and properties of the associator and commutator that we need to mention.

Lemma 4.1. *For all $x, y, z \in A$, $(x, y, yz) = (x, y, z)y$.*

Proof: Let $x, y, z \in A$ be given. Using the definition of the associator, we see that

$$(x, y, z)y = ((xy)z - x(yz))y = ((xy)z)y - (x(yz))y.$$

Next, using the Moufang identity $u(vwv) = ((uv)w)v$, we see that $((xy)z)y = x(yzy)$. Then because $(uv)u = u(vu)$, we can write this as $((xy)z)y = x((yz)y)$. Then, using the definition of the associator, we have that

$$\begin{aligned} ((xy)z)y - (x(yz))y &= x((yz)y) - (x(yz))y \\ &= -(x, yz, y). \end{aligned}$$

Finally, using the results from the previous lemma, we conclude that

$$(x, y, z)y = -(x, yz, y) = (x, y, yz).$$

□

Next we establish the following identity of the associator in an alternative algebra A.

Lemma 4.2. $(xa, b, y) + (x, a, by) + (b, a, xy) + (ba, x, y) = 0$ *for all* $a, b, x, y \in A$.

Proof: Let $a, b, x, y \in A$ be given. From the definition of the associator, we obtain the following properties

$$\begin{aligned} (ab)x &= (a, b, x) + a(bx), \\ a(bx) &= (ab)x - (a, b, x). \end{aligned} \tag{4.2}$$

This is useful, for example, in writing something like $x((ab)y)$ in a way that uses the associator. For example, we have that $x(uy) = (xu)y - (x, u, y)$. Letting $u = ab$, we have

$$x((ab)y) = (x(ab))y - (x, ab, y). \tag{4.3}$$

Using the linear properties of the associator, notice that

$$\begin{aligned} (b + x, a, (b + x)y) &= (b + x, a, by + xy) \\ &= (b, a, by + xy) + (x, a, by + xy) \\ &= (b, a, by) + (x, a, by) + (b, a, xy) + (x, a, xy). \end{aligned} \tag{4.4}$$

Also, by Lemma 4.1, we also have that

$$\begin{aligned} (b + x, a, (b + x)y) &= -(a, b + x, (b + x)y) \\ &= -(a, b + x, y) \cdot (b + x) \\ &= (b + x, a, y)(b + x). \end{aligned} \tag{4.5}$$

Therefore, substituting equation (4.5) into (4.5) gives us

$$\begin{aligned} (x, a, by) + (b, a, xy) &= (b + x, a, by + xy) - (b, a, by) - (x, a, xy) \\ &= (b + x, a, (b + x)y) + (a, b, by) + (a, x, xy) \\ &= (b + x, a, y)(b + x) + (a, b, y)b + (a, x, y)x \\ &= (b + x, a, y)(b + x) - (b, a, y)b - (x, a, y)x. \end{aligned}$$

By expanding the term $(b+x,a,y)(b+x) = (b,a,y)b + (x,a,y)b + (b,a,y)x + (x,a,y)x$, we can substitute this into the previous equation to get

$$\begin{aligned}(x,a,by) + (b,a,xy) &= (b,a,y)b + (x,a,y)b + (b,a,y)x + (x,a,y)x - (b,a,y)b - (x,a,y)x \\ &= (x,a,y)b + (b,a,y)x.\end{aligned} \tag{4.6}$$

Similarly, looking at the term $(xa+ba, x+b, y)$, we see that $(xa+ba, x+b, y) = (xa,x,y) + (ba,x,y) + (xa,b,y) + (ba,b,y)$. Then using Lemma 4.1, we get

$$\begin{aligned}(xa,b,y) + (ba,x,y) &= (xa+ba, x+b, y) - (xa,x,y) - (ba,b,y) \\ &= -(y, x+b, (x+b)a) + (y,x,xa) + (y,b,ba) \\ &= -(y,x,a)x - (y,b,a)x - (y,x,a)b - (y,b,a)b \\ &\quad + (y,x,a)x + (y,b,a)b \\ &= -(y,b,a)x - (y,x,a)b.\end{aligned}$$

This means that

$$(xa,b,y) + (ba,x,y) = -(y,b,a)x - (y,x,a)b. \tag{4.7}$$

Adding equations (4.6) and (4.7), and cycling on the terms in the associators, we get

$$\begin{aligned}(xa,b,y) + (x,a,by) + (b,a,xy) + (ba,x,y) &= (x,a,y)b + (b,a,y)x - (y,b,a)x - (y,x,a)b \\ &= (x,a,y)b - (x,a,y)b + (b,a,y)x - (b,a,y)x \\ &= 0.\end{aligned}$$

Therefore, we have the identity $(xa,b,y) + (x,a,by) + (b,a,xy) + (ba,x,y) = 0$. □

Now we are ready to prove Theorem 4.1.

Proof: First, we will show that the map $D_{a,b}$ is linear. Then, using properties of alternative algebras, we will show that $D_{a,b}$ satisfies the Leibniz property.

We know that the commutator operation is linear. Let $r, s \in \mathbb{R}$ and $a, b, c, d \in A$. Then

$$\begin{aligned}(ra + sb, c, d) &= ((ra + sb)c)d - (ra + sb)(cd) \\ &= ((ra)c)d + ((sb)c)d - (ra)(cd) - (sb)(cd) \\ &= r(ac)d - ra(cd) + s(bc)d - sb(cd) \\ &= r(a, c, d) + s(b, c, d).\end{aligned}$$

This means the associator is linear in its first argument. Using the property that $(x, y, z) = (z, x, y) = (y, z, x)$, we can see that the associator is also linear in it's second and third arguments as well. Thus, the associator is a linear operation. Therefore, it suffices to show that $D_{a,b}$ satisfies the Leibniz property.

To show that $D_{a,b}$ satisfies the Leibniz property, we will manipulate terms $x \cdot D_{a,b}(y)$ and $D_{a,b}(x) \cdot y$ and show that their sum is equal to $D_{a,b}(xy)$. For the remainder of the proof, let $a, b, x, y \in A$ be given.

We begin our analysis by manipulating the term $xD_{a,b}(y) = x[[a, b], y] - 3x(a, b, y)$. First, consider the term $x[[a, b], y]$. We can expand using the definition of the commutator and the distributive properties of A to get

$$\begin{aligned}x[[a, b], y] &= x[ab - ba, y] \\ &= x\Big((ab)y - (ba)y - y(ab) + y(ba)\Big) \\ &= x((ab)y) - x((ba)y) - x(y(ab)) + x(y(ba)).\end{aligned}$$

Then, we can rewrite the first two terms using equation (4.3), and rewrite the last two terms using equation (4.2). This gives us

$$\begin{aligned}x[[a, b], y] = (x(ab))y - (x, ab, y) - (x(ba))y + (x, ba, y) \\ - (xy)(ab) + (x, y, ab) + (xy)(ba) - (x, y, ba).\end{aligned}$$

Then, we can rearrange terms together and rewrite this again using the commutator and the associator

$$\begin{aligned} x[[a,b],y] &= \Big((x(ab))y - (x(ba))y\Big) - \Big((xy)(ab) - (xy)(ba)\Big) \\ &\quad + \Big((x,y,ab) - (x,y,ba)\Big) - \Big((x,ab,y) - (x,ba,y)\Big) \\ &= \Big((x[a,b])y - (xy)[a,b]\Big) + (x,y,ab-ba) - (x,ab-ba,y) \\ &= (x[a,b])y - (xy)[a,b] + 2(x,y,[a,b]). \end{aligned}$$

Therefore, we have that

$$x[[a,b],y] = (x[a,b])y - (xy)[a,b] + 2(x,y,[a,b]). \tag{4.8}$$

Next we work with the term $[[a,b],x]y$. Similar to our previous analysis, we we expand this using the definition of the commutator to get

$$\begin{aligned} [[a,b],x]y &= [ab-ba,x]y \\ &= \Big((ab)x - (ba)x - x(ab) + x(ba)\Big)y \\ &= ((ab)x)y - ((ba)x)y - (x(ab))y + (x(ba))y. \end{aligned}$$

Again, using equations (4.2) and (4.3), we can rewrite this as

$$\begin{aligned} [[a,b],x]y = &\Big((ab,x,y) + (ab)(xy)\Big) - \Big((ba,x,y) + (ba)(xy)\Big) \\ &- \Big((x,ab,y) + x((ab)y)\Big) + \Big((x,ba,y) + x((ba)y)\Big). \end{aligned}$$

Rearranging terms and combining using the definitions of the commutator and associator yields

$$\begin{aligned} [[a,b],x]y &= \Big((ab,x,y) - (ba,x,y)\Big) - \Big((x,ab,y) - (x,ba,y))\Big) \\ &\quad + \Big(x((ba)y - x((ab)y)\Big) - \Big((ba)(xy) - (ab)(xy)\Big) \\ &= x([b,a]y) - [b,a](xy) + ([a,b],x,y) - (x,[a,b],y) \\ &= -x([a,b]y) + [a,b](xy) + 2(x,y,[a,b]). \end{aligned}$$

Therefore, we have that

$$[[a,b],x]y = -x([a,b]y) + [a,b](xy) + 2(x,y,[a,b]). \tag{4.9}$$

Adding equations (4.8) and (4.9) together yields

$$\begin{aligned} x[[a,b],y] + [[a,b],x]y &= (x[a,b])y - (xy)[a,b] + 2(x,y,[a,b]) \\ &\quad - x([a,b]y) + [a,b](xy) + 2(x,y,[a,b]) \\ &= 4(x,y,[a,b]) + [a,b](xy) - (xy)[a,b] + (x[a,b])y - x([a,b]y) \\ &= [[a,b],xy] + 4(x,y,[a,b]) + (x,[a,b],y) \\ &= [[a,b],xy] + 3(x,y,[a,b]). \end{aligned}$$

Therefore,

$$x[[a,b],y] + [[a,b],x]y = [[a,b],xy] + 3(x,y,[a,b]). \tag{4.10}$$

This certainly looks promising. Notice that the left hand side involves products of elements with a commutators. On the right hand side, we are able to write the relation as a sum of commutators and associators (which is very similar to how the map $D_{a,b}$ is defined).

Let's summarize what we have so far. Doing so will allow us to see how the remainder of the proof will go.

$$\begin{aligned} xD_{a,b}(y) + D_{a,b}(x)y &= x[[a,b],y] - 3x(a,b,y) + [[a,b],x]y - 3(a,b,x)y \\ &= [[a,b],xy] + 3(x,y,[a,b]) - 3x(a,b,y) - 3(a,b,x)y \\ &= [[a,b],xy] + 3\Big((x,y,[a,b]) - x(a,b,y) - (a,b,x)y\Big). \end{aligned}$$

Therefore, to show that $D_{a,b}$ is a derivation, we need to show that

$$(x,y,[a,b]) - x(a,b,y) - (a,b,x)y = -(a,b,xy).$$

To accomplish this, we need to establish a few intermediate results.

By direct expansion using the definition of the associator, we get that

$$\begin{aligned} x(a,b,y)+(x,a,b)y &= x((ab)y-a(by))+((xa)b-x(ab)y) \\ &= \Big(((xa)b)y-(xa)(by)\Big)-\Big((x(ab))y-x((ab)y)\Big) \\ &\quad +\Big((xa)(by)-x(a(by))\Big) \\ &= (xa,b,y)-(x,ab,y)+(x,a,by). \end{aligned}$$

Now, we can use this result to show that $x(a,b,y)+(a,b,x)y = (a,b,xy)+(x,y,[a,b])$. First of all, notice that

$$\begin{aligned} x(a,b,y)+(a,b,x)y &= x(a,b,y)+(x,a,b)y \\ &= (xa,b,y)+(x,a,by)-(x,ab,y). \end{aligned}$$

Recall the identity from Lemma 4.2, $(xa,b,y)+(x,a,by)+(b,a,xy)+(ba,x,y)=0$. We can rewrite the identity from this claim as $(xa,b,y)+(x,a,by)=(xy,a,b)-(x,y,ba)$. This allows us to write

$$\begin{aligned} x(a,b,y)+(a,b,x)y &= (xy,a,b)-(x,y,ba)-(x,ab,y) \\ &= (a,b,xy)-(x,y,ba)+(x,y,ab) \\ &= (a,b,xy)+(x,y,[a,b]). \end{aligned}$$

Therefore, solving for (a,b,xy), we get

$$x(a,b,y)+(a,b,x)y = (a,b,xy)+(x,y,[a,b]). \tag{4.11}$$

Therefore, using equations (4.10) and (4.11), we have that

$$\begin{aligned} xD_{a,b}(y) + D_{a,b}(y)x &= x[[a,b],y] - 3x(a,b,y) + [[a,b],x]y - 3(a,b,x)y \\ &= \Big(x[[a,b],y] + [[a,b],x]y\Big) - 3\Big(x(a,b,y) + (a,b,x)y\Big) \\ &= \Big([[a,b],xy] + 3(x,y,[a,b])\Big) - 3\Big((a,b,xy) + (x,y,[a,b])\Big) \\ &= [[a,b],xy] - 3(x(a,b,y) + 3(a,b,x)y - (x,y,[a,b])) \\ &= [[a,b],xy] - 3(a,b,xy) \\ &= D_{a,b}(xy). \end{aligned}$$

Thus, $D_{a,b} : A \to A$, defined by $D_{a,b}(x) = [[a,b],x] - 3(a,b,x)$, is a linear map satisfing the Leibniz property, making it a derivation on A. □

Recall that the normed division algebras are also alternative algebras. This gives us the following corollary, which is the result that we are interested in.

Corollary 4.1. *The map $D_{a,b} : A \to A$ given by $D_{a,b}(x) = [[a,b],x] - 3(a,b,x)$ is a derivation for every normed division algebra A.*

Now that we know the map $D_{a,b}$ is a derivation of a normed division algebra $\mathbb{K}$, we can explore some of the properties of the map, particularly those that will allow us to determine a basis of $\mathfrak{der}(\mathbb{K})$.

Notice that $D_{a,b} : \mathbb{K} \to \mathbb{K}$ is also linear with respect to index the elements a and b. Suppose that $X = \{e_i\}$ is a basis of A. Then for some $a^i, b^j \in \mathbb{R}$, $a = a^i e_i$ and $b = b^j e_j$, and any $x \in \mathbb{K}$,

$$\begin{aligned} D_{a,b}(x) &= [[a,b],x] - 3(a,b,x) \\ &= \left[\left[a^i e_i, b^j e_j\right], x\right] - 3\left(a^i e_i, b^j e_j, x\right) \\ &= a^i b^j [[e_i,e_j],x] - a^i b^j 3(e_i,e_j,x) \\ &= a^i b^j \left([[e_i,e_j],x] - 3(e_i,e_j,x) \right) \\ &= a^i b^j D_{e_i,e_j}(x). \end{aligned}$$

Notice also that for any $a, b, x \in \mathbb{K}$,

$$\begin{aligned} D_{a,b}(x) &= [[a,b],x] - 3(a,b,x) \\ &= [-[b,a],x] - 3(-(b,a,x)) \\ &= -([[b,a],x] - 3(b,a,x)) \\ &= -D_{b,a}(x). \end{aligned}$$

Therefore, given a basis $\{e_i\}$ of $\mathbb{K}$, the set of maps $\{D_{e_i,e_j} : \mathbb{K} \to \mathbb{K}\}$ is a subalgebra of $\mathfrak{der}(\mathbb{K})$. To show that this set of maps is a basis, we need to show that they are linearly independent, and that the span of the set is $\mathfrak{der}(\mathbb{K})$. However, in truth, we need only consider the algebras $\mathbb{H}$ and $\mathbb{O}$ because $\mathfrak{der}(\mathbb{R}) = 0$ and $\mathfrak{der}(\mathbb{C}) = 0$. Furthermore, it has been shown by Benkart and Osborn, that if $\dim(\mathbb{K}) = 4$ then $\mathfrak{der}(\mathbb{K}) \cong \mathfrak{su}_2(\mathbb{C})$ or $\dim(\mathfrak{der}(\mathbb{K}))$ is 0 or 1.[2] Furthermore, they showed that there are only five possibilities for $\mathfrak{der}(\mathbb{K})$ when $\dim(\mathbb{K}) = 8$. Therefore, we need only show that the subspace generated by the derivations defined using a basis of $\mathbb{K}$, namely $\{D_{e_i,e_j}\}$, does in fact form the algebras specified.

For example, consider the quaternion algebra $\mathbb{H}$. We can show that the and let $\{1, \hat{i}, \hat{j}, \hat{k}\}$ be the standard basis of $\mathbb{H}$. We can compute the matrix representation of $D_{\hat{i},\hat{j}}$ as follows:

Note that since $\mathbb{H}$ is an associative algebra, $(x, y, z) = 0$ for all $x, y, z \in \mathbb{H}$, so we need only be concerned with the commutator term. Also, note that $[\hat{i}, \hat{j}] = \hat{i}\hat{j} - \hat{j}\hat{i} = \hat{k} - (-\hat{k}) = 2\hat{k}$, so $D_{\hat{i},\hat{j}}(x) = 2[\hat{k}, x]$. Then we see

$$\begin{aligned} D_{\hat{i},\hat{j}}(1) &= 2[\hat{k}, 1] \\ &= 2(\hat{k} \cdot 1 - 1 \cdot \hat{k}) \\ &= 0 \cdot 1 + 0\hat{i} + 0\hat{j} + 0\hat{k}, \end{aligned}$$

$$\begin{aligned}
D_{\hat{i},\hat{j}}(\hat{i}) &= 2[\hat{k},\hat{i}] \\
&= 2(\hat{k}\hat{i} - \hat{i}\hat{k}) \\
&= 2(\hat{j} + \hat{j}) \\
&= 0 \cdot 1 + 0\hat{i} + 4\hat{j} + 0\hat{k}, \\
D_{\hat{i},\hat{j}}(\hat{j}) &= 2[\hat{k},\hat{j}] \\
&= 2(\hat{k}\hat{j} - \hat{j}\hat{k}) \\
&= 2(-\hat{i} - \hat{i}) \\
&= 0 \cdot 1 - 4\hat{i} + 0\hat{j} + 0\hat{k}, \\
D_{\hat{i},\hat{j}}(\hat{k}) &= 2[\hat{k},\hat{k}] \\
&= 0 \cdot 1 + 0\hat{i} + 0\hat{j} + 0\hat{k}.
\end{aligned}$$

Therefore, the matrix representation of the derivation map $D_{\hat{i},\hat{j}} : \mathbb{H} \to \mathbb{H}$ is

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

We also have that $D_{1,b}(x) = [[1,b],x] = [0,x] = 0$. So the matrix representation of any derivation map with the multiplicative identity in the index is the zero map (This is also true for any alternative algebra). The remaining linear maps in the basis of $\mathfrak{der}(\mathbb{H})$ are:

$$D_{\hat{i},\hat{k}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \end{pmatrix} \qquad D_{\hat{j},\hat{k}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & -4 & 0 \end{pmatrix}$$

It is easily verified that these matrices satisfy the structure equations for the Lie algebra $\mathfrak{su}_2(\mathbb{C})$. Therefore, we have that $\left\{D_{\hat{i},\hat{j}}, D_{\hat{i},\hat{k}}, D_{\hat{j},\hat{k}}\right\}$ is a basis of the derivation algebra $\mathfrak{der}(\mathbb{H})$ [2]. We can likewise produce a basis for the derivation algebra $\mathfrak{der}(\mathbb{O})$. It will not be shown here, but by following the tutorial worksheet in section C.6, we will create a Lie algebra using the derivations on the octonions using this formula. Then, we compute a Cartan matrix for this Lie algebra and show that it is the Cartan matrix for $\mathfrak{g}_2$. Thus,

using the formula (4.1) allows us to create the Lie algebra $\mathfrak{g}_2$. Therefore, we know that we only need this derivation map to create a basis of $\mathfrak{der}(\mathbb{O})$.

To finish this section, we will show how, given derivations of two alternative algebras $\mathbb{K}$ and $\mathbb{M}$, we can construct a derivation of $\mathbb{K} \otimes \mathbb{M}$. This will set the stage for working with the bracket as defined by Vinberg.

Let $\mathbb{K}$ and $\mathbb{M}$ be normed division algebras and $\mathfrak{der}(\mathbb{K})$ and $\mathfrak{der}(\mathbb{M})$ be their respective derivation algebras. We will denote the derivations in $\mathfrak{der}(\mathbb{K})$ by $D^{(1)}_{x,y}$, and likewise denote derivations in $\mathfrak{der}(\mathbb{M})$ by $D^{(2)}_{u,v}$.

Proposition 4.1. *For every pair of derivations $A \in \mathfrak{der}(\mathbb{K})$ and $B \in \mathfrak{der}(\mathbb{M})$, define a linear map $\delta_{A,B} : \mathbb{K} \otimes \mathbb{M} \to \mathbb{K} \otimes \mathbb{M}$ by*

$$\delta_{A,B}(x \otimes u) = A(x) \otimes u + x \otimes B(u), \text{ where } x \otimes u \in \mathbb{K} \otimes \mathbb{M}.$$

Then $\delta_{A,B}$ is a derivation on $\mathbb{K} \otimes \mathbb{M}$.

Proof: We only need to show that the map satisfies the Leibniz property on tensors. Let $u \otimes r, v \otimes s \in \mathbb{K} \otimes \mathbb{M}$ be given. Then

$$\begin{aligned}
\delta_{A,B}\Big((u \otimes r)(v \otimes s)\Big) &= \delta_{A,B}(uv \otimes rs) \\
&= A(uv) \otimes rs + uv \otimes B(rs) \\
&= \Big(A(u)v + uA(v)\Big) \otimes rs + uv \otimes \Big(B(r)s + rB(s)\Big) \\
&= A(u)v \otimes rs + uA(v) \otimes rs + uv \otimes B(r)s + uv \otimes rB(s) \\
&= \Big(A(u)v \otimes rs + uv \otimes B(r)s\Big) + \Big(uA(v) \otimes rs + uv \otimes rB(s)\Big) \\
&= \Big(A(u) \otimes r + u \otimes B(r)\Big)(v \otimes s) + (u \otimes r)\Big(A(v) \otimes s + v \otimes B(s)\Big) \\
&= \delta_{A,B}(u \otimes r) \cdot (v \otimes s) + (u \otimes r) \cdot \delta_{A,B}(v \otimes s).
\end{aligned}$$

Therefore, $\delta_{A,B}$ is a derivation of $\mathbb{K} \otimes \mathbb{M}$. □

Proposition 4.2. *For any $A, B \in \mathfrak{der}(\mathbb{K})$ and $C, D \in \mathfrak{der}(\mathbb{M})$,*

$$\delta_{A,C} + \delta_{B,D} = \delta_{A+B,C+D}.$$

Proof: Let $A, B \in \mathfrak{der}(\mathbb{K})$ and $C, D \in \mathfrak{der}(\mathbb{M})$ be derivations and $u \otimes r \in \mathbb{K} \otimes \mathbb{M}$ be given. It suffices to show that $\delta_{A,C} + \delta_{B,D} = \delta_{A+B,C+D}$ holds true on tensors in $\mathbb{K} \otimes \mathbb{M}$.

$$\begin{aligned}(\delta_{A,C} + \delta_{B,D})(u \otimes r) &= \delta_{A,C}(u \otimes r) + \delta_{B,C}(u \otimes r) \\ &= A(u) \otimes r + u \otimes C(r) + B(u) \otimes r + u \otimes D(r) \\ &= (A(u) + B(u)) \otimes r + u \otimes (C(r) + D(r)) \\ &= (A + B)(u) \otimes r + u \otimes (C + D)(r) \\ &= \delta_{A+B,C+D}(u \otimes r).\end{aligned}$$

Therefore, by linearity of the maps, we know this statement holds for any element in $\mathbb{K} \otimes \mathbb{M}$. Therefore, we conclude that $\delta_{A,C} + \delta_{B,D} = \delta_{A+B,C+D}$. □

4.3 Vinberg's Magic Square Construction

This section gives the statement of Vinberg's symmetric construction of the Freudenthal magic square as well as a few variations. This magic square will allow us to compute the exceptional Lie algebras using the normed division algebras. No proofs of this construction will be given. The construction of a Maple procedure to compute the bracket will be addressed in the next section.

Let $\mathbb{K}$ and $\mathbb{M}$ be normed division algebras and let $u \otimes r, v \otimes s \in \mathbb{K} \otimes \mathbb{M}$ be given. We can define a derivation on $\mathbb{K} \oplus \mathbb{M}$ by the linear map

$$D_{u\otimes r, v\otimes s} = \Big(\langle r, s \rangle D^{(1)}_{u,v} \ , \ \langle u, v \rangle D^{(2)}_{r,s} \Big), \tag{4.12}$$

where $D^{(1)}_{u,v} \in \mathfrak{der}(\mathbb{K})$ and $D^{(2)}_{r,s} \in \mathfrak{der}(\mathbb{M})$ are defined by equation (4.1). Written out in full detail, using equation (4.1), we see that for any $(w, t) \in \mathbb{K} \oplus \mathbb{M}$,

$$\begin{aligned}D_{u\otimes r, v\otimes s}(w, t) &= \Big(\langle r, s \rangle D^{(1)}_{u,v}(w), \langle u, v \rangle D^{(2)}_{r,s}(t) \Big) \\ &= \Big(\langle r, s \rangle ([[u, v], w] - 3(u, v, w)), \langle u, v \rangle ([[r, s], t] - 3(r, s, t)) \Big).\end{aligned} \tag{4.13}$$

So, given any two elements in $\mathbb{K} \otimes \mathbb{M}$, we can create an element in $\mathfrak{der}(\mathbb{K}) \oplus \mathfrak{der}(\mathbb{M})$. Furthermore, notice that

$$\begin{aligned} D_{u\otimes r, v\otimes s}(w,t) &= \Big(\langle r,s\rangle([[u,v],w] - 3(u,v,w)), \langle u,v\rangle([[r,s],t] - 3(r,s,t))\Big) \\ &= \Big(\langle r,s\rangle(-[[v,u],w] + 3(v,u,w)), \langle u,v\rangle(-[[s,r],t] + 3(s,r,t))\Big) \\ &= -\Big(\langle s,r\rangle([[v,u],w] - 3(v,u,w)), \langle v,u\rangle([[s,r],t] - 3(s,r,t))\Big) \\ &= -D_{v\otimes s, u\otimes r}(w,t). \end{aligned} \tag{4.14}$$

Recall that $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$ is the space of 3×3 trace-free, skew-hermitian matrices over $\mathbb{K} \otimes \mathbb{M}$. Then Vinberg's definition of the magic square Lie algebras is given by [1]:

$$M(\mathbb{K}, \mathbb{M}) = \mathfrak{der}(\mathbb{K}) \oplus \mathfrak{der}(\mathbb{M}) \oplus \mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$$

where the Lie bracket in $M(\mathbb{K}, \mathbb{M})$ is defined in three parts.

1. For any $(A, C), (B, D) \in \mathfrak{der}(\mathbb{K}) \oplus \mathfrak{der}(\mathbb{M})$,

$$[(A, C), (B, D)]_1 = (AB - BA, CD - DC) = ([A, B], [C, D]).$$

This tells us that $[(A, 0), (0, C)]_1 = 0$. This makes $\mathfrak{der}(\mathbb{K}) \oplus \mathfrak{der}(\mathbb{M})$ a Lie subalgebra of $M(\mathbb{K}, \mathbb{M})$.

2. For any $(A, C) \in \mathfrak{der}(\mathbb{K}) \oplus \mathfrak{der}(\mathbb{M})$ and $X \in \mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$,

$$[(A, C), X]_2 = \delta_{A,C}(X) \qquad \text{and} \qquad [X, (A, C)]_2 = -\delta_{A,C}(X)$$

where $\delta_{A,C}$ acts on every entry of the matrix X. Let $A \in \mathfrak{der}(\mathbb{K})$, $C \in \mathfrak{der}(\mathbb{M})$ and $X \in \mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$ be a matrix with entries $X_{ij} = \chi_{ij}^{mn} u_m \otimes r_n$. Then, we see that

$$\begin{aligned} \delta_{A,C}(X_{ij}) &= \chi_{ij}^{mn} \delta_{A,B}(u_m \otimes r_n) \\ &= \chi_{ij}^{mn} \left(A(u_m) \otimes r_n + u_m \otimes C(r_n)\right) \\ &= \chi_{ij}^{mn} \left(A_m^k u_k \otimes r_n + C_n^\ell u_m \otimes r_\ell\right) \\ &= \chi_{ij}^{mn} A_m^k u_k \otimes r_n + \chi_{ij}^{mn} C_n^\ell u_m \otimes r_\ell \\ &= \left(\chi_{ij}^{m\ell} A_m^k + \chi_{ij}^{kn} C_n^\ell\right) u_k \otimes r_\ell. \end{aligned}$$

3. Given $X, Y \in \mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$,

$$[X, Y] = \frac{1}{3} \sum_{i,j=1}^{3} D_{X_{ij}, Y_{ij}} \oplus [X, Y]_0$$

where $D_{X_{ij},Y_{ij}}$, defined using equation (4.12), creates elements in $\mathfrak{der}(\mathbb{K})$ and $\mathfrak{der}(\mathbb{M})$ for every $1 \leq i, j \leq 3$ and $[X, Y]_0 = XY - YX - \frac{1}{3} Tr(XY - YX) I_3$ is the commutator with the trace removed.

Before we go into more detail on the bracket, first we will show that this Lie bracket is skew-symmetric and linear. It is beyond the scope of this thesis to verify that this Lie bracket satisfies the Jacobi property (although we will have a few remarks regarding the Jacobi property).

To show that the Lie bracket is linear, it suffices to show that the second and third rules for the bracket are linear. Let $(A, C), (B, D) \in \mathfrak{der}(\mathbb{K}) \oplus \mathfrak{der}(\mathbb{M})$, $X, Y, Z \in \mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$ and $a, b \in \mathbb{R}$ be given. First we show that the second rule is linear.

$$\begin{aligned}
[a(A, C) + b(B, D), X] &= [(aA + bB, aC + bD), X] \\
&= \delta_{aA+bB, aC+bD}(X) \\
&= (\delta_{aA+bB, aC+bD}(X_{ij}))_{ij} \\
&= (a\delta_{A,C}(X_{ij}) + b\delta_{B,D}(X_{ij}))_{ij} \\
&= a\delta_{A,C}(X) + b\delta_{B,D}(X).
\end{aligned}$$

To show that the third bracket rule is linear, recall that the commutator is linear and the trace is linear, making the composition of the two operations linear. Therefore, we have that $[aX + bZ, Y]_0 = a[X, Y]_0 + b[Z, Y]_0$ and $[X, aY + bZ]_0 = a[X, Y]_0 + b[X, Z]_0$. Furthermore, using equation (4.13), we see that the map $D_{X_{ij},Y_{ij}}$ is linear in all of its arguments. Thus, $[aX + bZ, Y] = a[X, Y] + b[Z, Y]$ and $[X, aY + bZ] = a[X, Y] + b[X, Z]$.

Therefore, we see that the Lie bracket as defined by Vinberg is linear. Next we will show that the bracket is skew-symmetric. We know that the bracket defined by the first rule is skew-symmetric because $\mathfrak{der}(\mathbb{K}) \oplus \mathfrak{der}(\mathbb{M})$ is a Lie algebra. By definition of the second rule, we see that the bracket it skew-symmetic, namely

$$[\delta_{A,B}, X] + [X, \delta_{A,B}] = \delta_{A,B}(X) - \delta_{A,B}(X) = 0.$$

Next we show that the third bracket rule is skew-symmetric. From equation (4.14), we have $D_{X_{ij},Y_{ij}} = -D_{Y_{ij},X_{ij}}$ for each i and j. This means $\frac{1}{3}\sum_{ij=1}^{3} D_{X_{ij},Y_{ij}} = -\frac{1}{3}\sum_{ij=1}^{3} D_{Y_{ij},X_{ij}}$. Notice also that

$$\begin{aligned}[X,Y]_0 &= XY - YX - \frac{1}{3}Tr(XY-YX)I_3 \\ &= -(YX - XY - \frac{1}{3}Tr(YX-XY)I_3) \\ &= -[X,Y]_0.\end{aligned}$$

Therefore, we see that $[X,Y] = -[Y,X]$, making the third rule skew-symmetric. Thus we see that the Lie bracket as defined by Vinberg is skew-symmetric.

So far, we have shown that each rule for the Lie bracket satisfies the properties of linearity and skew-symmetry. We can likewise show that the first and second properties satisfy the Jacobi property. Showing that the third rule satisfies the Jacobi property has proven to be difficult and will not be addressed here. However, we will show that the trace-free commutator does not satisfy the Jacobi property (which is why the additional terms are needed in the third rule).

For example, we will show that the algebra $\mathfrak{sa}_3(\mathbb{H}\otimes\mathbb{R}) \cong \mathfrak{sa}_3(\mathbb{H})$, together with the trace-free commutator, does not satisfy the Jacobi property. It suffices to show that the following three matrices do not satisfy the Jacobi property:

$$U = \begin{pmatrix} i & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -i \end{pmatrix}, V = \begin{pmatrix} j & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -j \end{pmatrix}, W = \begin{pmatrix} 0 & 0 & 0 \\ 0 & i & 0 \\ 0 & 0 & -i \end{pmatrix}.$$

We begin by computing $[U,V] = UV - VU$.

$$[U,V] = \begin{pmatrix} 2k & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 2k \end{pmatrix}$$

We see that the commutator produces a matrix which is not trace free. However, we see that the trace is $4k$. Therefore we can subtract the term $4/3k$ from the diagonal entries.

This gives us

$$[U,V]_0 = \begin{pmatrix} \frac{2}{3}k & 0 & 0 \\ 0 & -\frac{4}{3}k & 0 \\ 0 & 0 & \frac{2}{3}k \end{pmatrix}.$$

We can take this result and compute the commutator with W and remove the trace. Doing so gives us

$$[[U,V]_0,W]_0 = \begin{pmatrix} \frac{4}{3}j & 0 & 0 \\ 0 & -\frac{4}{3}j & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Similarly, we compute the following results

$$[[V,W]_0,U]_0 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -\frac{4}{3}j & 0 \\ 0 & 0 & \frac{4}{3}j \end{pmatrix} \quad \text{and} \quad [[W,U]_0,V]_0 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Summing these results together yields the matrix $\begin{pmatrix} \frac{4}{3}j & 0 & 0 \\ 0 & -\frac{8}{3}j & 0 \\ 0 & 0 & \frac{4}{3}j \end{pmatrix}$, which is certainly not zero. Therefore, we have shown that the trace-free commutator for elements in $\mathfrak{sa}_3(\mathbb{H})$ does not satisfy the Jacobi property. From this one can easily show that the trace-free commutator will not satisfy the Jacobi property for $\mathfrak{sa}_3(\mathbb{O}\otimes\mathbb{R})$ (because $\mathbb{H}$ is a subalgebra of $\mathbb{O}$). Furthermore, this result is true for $\mathfrak{sa}_3(A)$ where A is a tensor product with either $\mathbb{H}$ or $\mathbb{O}$. These calculations motivate the addition of the terms present in the third rule of the Lie bracket.

Note, that every element of $\mathbb{K}\otimes\mathbb{M}$ can be written as $\chi^{ma}u_m\otimes r_a$ where $\chi^{ma}\in\mathbb{R}$. Then, for $X,Y\in\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$, the entries of these matrices can be written as $X_{ij} = \chi_{ij}^{ma}u_m\otimes r_a$ and $Y_{ij} = \psi_{ij}^{nb}u_n\otimes r_b$.

Then we can write out the derivation $D_{X_{ij},Y_{ij}}$ in full detail.

$$D_{X_{ij},Y_{ij}} = \chi_{ij}^{ma}\psi_{ij}^{nb}\left(\langle r_a,r_b\rangle D^{(1)}_{u_n,u_m}\ ,\ \langle r_a,r_b\rangle D^{(2)}_{r_a,r_b}\right).$$

By writing out the derivation $D_{X_{ij},Y_{ij}}$, we can denote the third rule for the bracket by

$$[X,Y]_3 = [X,Y]_{3_{\mathbb{K}}} \oplus [X,Y]_{3_{\mathbb{M}}} \oplus [X,Y]_0 \tag{4.15}$$

where

$$[X,Y]_{3_{\mathbb{K}}} = \frac{1}{3}\sum_{ij=1}^{3} \chi_{ij}^{ma}\psi_{ij}^{nb}\langle r_a, r_b\rangle D^{(1)}_{u_n,u_m}$$
$$[X,Y]_{3_{\mathbb{M}}} = \frac{1}{3}\sum_{ij=1}^{3} \chi_{ij}^{ma}\psi_{ij}^{nb}\langle u_m, u_n\rangle D^{(2)}_{r_a,r_b}.$$

Next, we will briefly discuss the computations which are required in computing the Lie bracket. Notice that every element in $M(\mathbb{K},\mathbb{M})$ can be written as

$$(A,C,X) = (A,C,0) + (0,0,X),$$

where $A \in \mathfrak{der}\,(\mathbb{K})$, $C \in \mathfrak{der}\,(\mathbb{M})$ and $X \in \mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$.

Then we have

$$\begin{aligned}[(A,C,X),(B,D,Y)] &= [(A,C,0)+(0,0,X),(B,D,0)+(0,0,Y)]\\ &= [(A,C,0),(B,D,0)] + [(A,C,0),(0,0,Y)]\\ &\quad + [(0,0,X),(B,D,0)] + [(0,0,X),(0,0,Y)].\end{aligned}$$

Using the Lie bracket rules, these can be simplified to get

$$\begin{aligned}[(A,C,X),(B,D,Y)] &= ([(A,C),(B,D)]_1,0) + (0,0,[(A,C),Y]_2)\\ &\quad + (0,0,[X,(B,D)]_2) + (0,0,[X,Y]_3)\\ &= ([A,B],[C,D],\delta_{A,C}(Y) - \delta_{B,D}(X)) + (0,0,[X,Y]_3).\end{aligned}$$

Using equation (4.15), this can be simplified further to get

$$\begin{aligned}&[(A,C,X),(B,D,Y)] =\\ &\qquad \Big([A,B] + [X,Y]_{3_{\mathbb{K}}}\ ,\ [C,D] + [X,Y]_{3_{\mathbb{M}}}\ ,\ \delta_{A,C}(Y) - \delta_{B,D}(X) + [X,Y]_0\Big).\end{aligned}$$

Writing out the Lie bracket in this form allows us to identify where the results of a given calculation will be located. The results are summarized in the following table.

TABLE 4.2: Decomposition of the Lie Bracket

	$\mathfrak{der}(\mathbb{K})$	$\mathfrak{der}(\mathbb{M})$	$\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$
$\mathfrak{der}(\mathbb{K})$	$\mathfrak{der}(\mathbb{K})$	0	$\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$
$\mathfrak{der}(\mathbb{M})$	0	$\mathfrak{der}(\mathbb{M})$	$\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$
$\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$	$\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$	$\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$	$M(\mathbb{K},\mathbb{M})$

Therefore, we see that we need only write procedures for computing the bracket in the following cases:

1. $[\mathfrak{der}(\mathbb{K}), \mathfrak{der}(\mathbb{K})]$,
2. $[\mathfrak{der}(\mathbb{M}), \mathfrak{der}(\mathbb{M})]$,
3. $[\mathfrak{der}(\mathbb{K}), \mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})]$,
4. $[\mathfrak{der}(\mathbb{M}), \mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})]$,
5. $[\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M}), \mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})]$.

Programs were written in Maple to compute the bracket of two derivations (easy), the bracket of a derivation and a matrix, and for computing the bracket of two matrices.

With the the vector space $M(\mathbb{K},\mathbb{M})$ and the Lie bracket defined, Vinberg was able to construct the Freudenthal magic square of Lie algebras as given in table 4.1. Because there is a natural isomorphism $\mathbb{K}\otimes\mathbb{M} \cong \mathbb{M}\otimes\mathbb{K}$, we have that $\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M}) \cong \mathfrak{sa}_3(\mathbb{M}\otimes\mathbb{K})$. Therefore, there is a clear symmetry in the bracket operation as given above which leads to the symmetry observed in table 4.1, namely

$$\mathfrak{der}(\mathbb{K}) \oplus \mathfrak{der}(\mathbb{M}) \oplus \mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M}) \cong \mathfrak{der}(\mathbb{M}) \oplus \mathfrak{der}(\mathbb{K}) \oplus \mathfrak{sa}_3(\mathbb{M}\otimes\mathbb{K}).$$

I created the procedure `MagicSquare` to create the structure constants for the Lie algebras given in the table by computing the bracket as defined by Vinberg. The command `MagicSquare` takes arguments specifying the algebras $\mathbb{K}$ and $\mathbb{M}$. It can also take other forms of these normed division algebras to produce alternate forms of the Lie algebras. The other valid algebras which can be used are the split complex ($\mathbb{C}'$),

split quaternion ($\mathbb{H}'$), and split octonion ($\mathbb{O}'$) algebras. Using these algebras allows us to construct the following table.

TABLE 4.3: Symmetric Magic Square $M(\mathbb{K}', \mathbb{M}')$

$\mathbb{K}/\mathbb{M}$	$\mathbb{R}$	$\mathbb{C}'$	$\mathbb{H}'$	$\mathbb{O}'$
$\mathbb{R}$	$\mathfrak{so}_3$	$\mathfrak{sl}_3(\mathbb{R})$	$\mathfrak{sp}_6(\mathbb{R})$	$\mathfrak{f}_{4(4)}$
$\mathbb{C}'$	$\mathfrak{sl}_3(\mathbb{R})$	$\mathfrak{sl}_3(\mathbb{R}) \oplus \mathfrak{sl}_3(\mathbb{R})$	$\mathfrak{sl}_6(\mathbb{R})$	$\mathfrak{e}_{6(6)}$
$\mathbb{H}'$	$\mathfrak{sp}_6(\mathbb{R})$	$\mathfrak{sl}_6(\mathbb{R})$	$\mathfrak{so}_{6,6}$	$\mathfrak{e}_{7(7)}$
$\mathbb{O}'$	$\mathfrak{f}_{4(4)}$	$\mathfrak{e}_{6(6)}$	$\mathfrak{e}_{7(7)}$	$\mathfrak{e}_{8(8)}$

Also, a non-symmetric method can be used to obtain even more new Lie algebras.

TABLE 4.4: Non-Symmetric Magic Square $M(\mathbb{K}', \mathbb{M})$

$\mathbb{K}/\mathbb{M}$	$\mathbb{R}$	$\mathbb{C}$	$\mathbb{H}$	$\mathbb{O}$
$\mathbb{R}$	$\mathfrak{so}_3$	$\mathfrak{su}_3$	$\mathfrak{sp}_3$	$\mathfrak{f}_4$
$\mathbb{C}'$	$\mathfrak{sl}_3(\mathbb{R})$	$\mathfrak{sl}_3(\mathbb{C})$	$\mathfrak{sl}_3(\mathbb{H})$	$\mathfrak{e}_{6(-26)}$
$\mathbb{H}'$	$\mathfrak{sp}_6(\mathbb{R})$	$\mathfrak{su}_{3,3}$	$\mathfrak{so}_6^*(\mathbb{H})$	$\mathfrak{e}_{7(-25)}$
$\mathbb{O}'$	$\mathfrak{f}_{4(4)}$	$\mathfrak{e}_{6(2)}$	$\mathfrak{e}_{7(-5)}$	$\mathfrak{e}_{8(-24)}$

The Lie algebras given in these tables (as well as table 4.1) are real Lie algebras. This means that, for example, the Lie algebra $\mathfrak{sl}_3(\mathbb{H})$ is the algebra over $\mathbb{R}$ where elements are 3×3 trace-free matrices with entries coming from $\mathbb{H}$. Therefore, the Lie bracket for this Lie algebra will have to be $[X, Y]_0$, namely, the commutator with the trace removed.

The index in for the exceptional Lie algebras denotes the rank of the algebra and the term in parentheses is the signature of the Killing form. For example, $\mathfrak{e}_{6(2)}$ is the real Lie algebra $\mathfrak{e}_6$ and the signature of the Killing form is 2. A Maple tutorial for creating the Lie algebra $\mathfrak{f}_{4(4)}$ and calculating the signature of the Killing form is given in the appendix.

4.4 Computing the Lie Bracket on $M(\mathbb{K}, \mathbb{M})$

This section will discuss how a procedure for computing the Lie bracket as defined in the previous section was implemented. Maple worksheets verifying table 4.1 can be found in appendix *C.7*.

In order to calculate the bracket of two elements in $M(\mathbb{K}, \mathbb{M})$, we need to be able to perform computations in the algebras $\mathbb{K}$, $\mathbb{M}$, $\mathbb{K} \otimes \mathbb{M}$, and $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$. The following topics will be addressed in this section:

1. Using the standard orthonormal bases of $\mathbb{K}$ and $\mathbb{M}$ to create lookup tables for efficiently performing calculations in $\mathbb{K}$ and $\mathbb{M}$ such as multiplication and conjugation (requires the `DifferentialGeometry` package).

2. Using equation (4.1), create basis matrices of $\mathfrak{der}(\mathbb{K})$ (and $\mathfrak{der}(\mathbb{M})$) which can be identified by $D_{a,b}$ where $a, b \in \mathbb{K}$ (or $a, b \in \mathbb{M}$ respectively). This requires the `DifferentialGeometry` package.

3. Using the standard bases of $\mathbb{K}$ and $\mathbb{M}$ to create a basis of $\mathbb{K} \otimes \mathbb{M}$.

4. Creating procedures to efficiently perform computations in $\mathbb{K} \otimes \mathbb{M}$ such as multiplication, addition, and conjugation using the basis of $\mathbb{K} \otimes \mathbb{M}$.

5. Creating procedures for multiplication and addition of matrices over $\mathbb{K} \otimes \mathbb{M}$.

6. Creating a procedure, using Proposition 4.1, for computing the action of $\delta_{A,B}(u \otimes r)$ for a given pair of derivations $A \in \mathfrak{der}(\mathbb{K})$ and $B \in \mathfrak{der}(\mathbb{M})$ and a tensor $u \otimes r$.

7. Creating a procedure for computing the trace-free commutator of two matrices over $\mathbb{K} \otimes \mathbb{M}$.

8. Creating a basis of $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$.

9. Using the bases of $\mathfrak{der}(\mathbb{K})$ and $\mathfrak{der}(\mathbb{M})$, create a procedure for computing the derivations $\langle r, s\rangle D^{(1)}_{u,v}$ and $\langle u, v\rangle D^{(2)}_{r,s}$ for a given pair of tensors $u \otimes r$ and $v \otimes s$.

10. Creating procedures for computing the Lie bracket of two basis elements of $M(\mathbb{K}, \mathbb{M})$.

11. Identifying linear combinations of basis elements of $M(\mathbb{K}, \mathbb{M})$.

The commands required to calculate the Lie bracket on $M(\mathbb{K}, \mathbb{M})$ can be very computationally intensive. To reduce the cost of calculations, I created a method for multiplying elements using arrays. Because the algebras are simply vector spaces equipped with additional multiplicative structure, we can encode multiplication in the algebra by looking at how the coefficients of vectors behave when multiplied together.

Looking at the product of two general elements in the algebra tells us how to encode the multiplication process as specific ways of combining lists of coefficients. Once the procedures for multiplication have been established, we can simply use these rules to construct rules for multiplication on $\mathbb{K} \otimes \mathbb{M}$ and then $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$ by simply specifying

how coefficients combine. This eliminates the need to use the command `evalDG` when combining elements of the algebras. Furthermore, because the bracket involves matrix multiplication of elements from $\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$, procedures were created to utilize the skew-hermitian structure of the matrices, allowing for quicker calculations. In the following steps, the three routines for quickly computing multiplication in $\mathbb{K}$, $\mathbb{K}\otimes\mathbb{M}$ and $\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$ will be discussed.

With this in mind, the `DifferentialGeometry` package will only be used to read in the multiplication rules for the algebras $\mathbb{K}$ and $\mathbb{M}$ and creating the derivation algebras at the very beginning and again at the very end to store the structure constants of the created Lie algebra.

Some of the details and variable definitions which should be clear to any reader will be omitted. For example, if we are working with an algebra we have called `K`, then the variable specifying the dimension of the vector space `K` will be `dimK`, but we don't need to explicitly show the details of how this value was obtained or where the variable assignment was given. For this, the reader can refer to the appendix for the code. The purpose of this discussion is to explain the strategy for computing the Lie bracket on $M(\mathbb{K},\mathbb{M})$.

Step 1: Multiplication in $\mathbb{K}$ and $\mathbb{M}$

First of all, we must read in the algebras that are requested. To best illustrate the procedures, we will let $\mathbb{K}=\mathbb{O}$ and $\mathbb{M}=\mathbb{C}$. These two algebras will allow us to explore some of the more interesting parts of the procedure without being bogged down by the size of the elements being created. The product rules for each of the algebras are read in using the `AlgebraLibraryData` command.

```
> DGsetup(AlgebraLibraryData("Octonions",K)):
```

To save on processing time, and eliminating the future need to use the command `evalDG`, we create a lookup table for the products. The table takes two integers, which identify the basis elements of the algebra, and returns the index of the basis element created along with the sign. Therefore, to create the lookup table, we use a custom built function `StructureTable` which works by multiplying the i^{th} and j^{th} basis elements together, identifying the result as the (positive/negative) k^{th} element. This result is then stored in the table using only the integers i, j, and $\pm k$.

```
> KTable := StructureTable( K ):
```

For example, letting $\{e_i\}$ be an orthonormal basis of $\mathbb{O}$, we know that $e_7 \cdot e_5 = -e_3$. So,

```
> KTable[ 7 , 5 ];
```

```
-3
```

Step 2: Derivations of $\mathbb{K}$ and $\mathbb{M}$.

Next, we create the algebra of derivations using the formula (4.1) for each of the algebras $\mathfrak{der}(\mathbb{K})$ and $\mathfrak{der}(\mathbb{M})$. I created the command `DerivationsTable` to create a lookup table to identify an element of the derivation algebra given basis elements from the algebra. `DerivationsTable` takes the lookup table for the algebra and the frame name of the algebra. The input `type=1` is to distinguish the algebra $\mathbb{K}$ from the algebra $\mathbb{M}$ in the creation process (required for technical reasons which are not important to the discussion). The output for the command is a table which accepts two values. These values represent the indices identifying basis elements of the algebra, given in the lower index of $D_{a,b}$, and returns a matrix representing the linear transformation.

```
> DerK := DerivationsTable(KTable, K ,type=1):
```

Once the product rules for multiplication in $\mathbb{K}$ and a basis for the derivation algebra for $Der(\mathbb{K})$ has been created using equation (4.1), we do the same for the algebra $\mathbb{M}$. For example, to obtain the derivation D_{e_2,e_4}, we simply evaluate:

```
> DerK[ 2 , 4 ];
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \end{bmatrix}.$$

Step 3: A basis of $\mathbb{K} \otimes \mathbb{M}$

We are now ready to construct the procedures for computing and identifying basis elements of the tensor product $\mathbb{K} \otimes \mathbb{M}$. First, we recognize that

$$\dim(\mathbb{K} \otimes \mathbb{M}) = \dim(\mathbb{K}) \cdot \dim(\mathbb{M}).$$

Using this formula, we initialize arrays and tables that will be used to store and identify basis elements of $\mathbb{K} \otimes \mathbb{M}$.

```
> dimKM := dimK*dimM:
> BasisKM := Array(1..dimKM):
> BasisIndex := table([]):
```

We use the canonical basis of $\mathbb{K} \otimes \mathbb{M}$, namely, by letting $\{u_i\}$ be a basis of $\mathbb{K}$ and $\{v_\alpha\}$ be a basis of $\mathbb{M}$, $\{u_i \otimes v_\alpha\}$ is a basis of $\mathbb{K} \otimes \mathbb{M}$. For our example, this means that

$$\begin{array}{llllll} \{ & f_1 = e_1 \otimes 1, & f_2 = e_1 \otimes i, & f_3 = e_2 \otimes 1, & f_4 = e_2 \otimes i, & f_5 = e_3 \otimes 1, & f_6 = e_3 \otimes i, \\ & f_7 = e_4 \otimes 1, & f_8 = e_4 \otimes i, & f_9 = e_5 \otimes 1, & f_{10} = e_5 \otimes i, & f_{11} = e_6 \otimes 1, & f_{12} = e_6 \otimes i, \\ & f_{13} = e_7 \otimes 1, & f_{14} = e_7 \otimes i & f_{15} = e_8 \otimes 1, & f_{16} = e_8 \otimes i & \} & \end{array}$$

is our basis of $\mathbb{O} \otimes \mathbb{C}$ where $\{1, i\}$ is a basis of $\mathbb{C}$. Notice that we fix the first element in the tensor product and cycle through the basis elements of the second algebra.

To make identification of the basis elements of $\mathbb{K} \otimes \mathbb{M}$ easier, it makes sense to treat them as vectors ($\mathbb{K} \otimes \mathbb{M}$ is still a vector space after all), and systematically fill in the look up table with indices which correspond to the basis vectors in $\mathbb{K} \otimes \mathbb{M}$ given the ordering of the creation method we used. This is done using nested `for` loops which cycle through the basis elements of $\mathbb{K}$ and $\mathbb{M}$ respectively.

For example, we can regard the element $e_3 \otimes i$ as the sixth basis element, so we'd represent it as

$$f_6 = e_3 \otimes i = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

At the same time that we create this vector, we also update the lookup table `BasisIndex` as follows

```
> BasisIndex[3,2] := 6:
> BasisIndex[6] := [3,2]:
```

This tells us that the tensor product of the third basis element from $\mathbb{K}$ and the second basis element of $\mathbb{M}$ create the sixth basis element of $\mathbb{K}\otimes\mathbb{M}$ and vise versa. This table then allows us to move back and forth from the algebra $\mathbb{K}\otimes\mathbb{M}$ and the individual algebras $\mathbb{K}$ and $\mathbb{M}$ (which will be needed when we begin working with multiplication).

Step 4: Calculations in $\mathbb{K}\otimes\mathbb{M}$

Next, we need to create a simple lookup table, called `ConjugateTable`, to specify the sign of the conjugate of a basis element in $\mathbb{K}\otimes\mathbb{M}$. In the case of our example, we see that

$$\overline{e_3\otimes i} = \overline{e_3}\otimes\bar{i} = (-e_3)\otimes(-i) = e_3\otimes i.$$

So we get

```
> ConjugateSign[6];
```

1

With basis elements in the tensor product created and easy to identify, we are now ready to build routines for performing arithmetic in the tensor algebra. First, we need to retrieve the sign (positive/negative) of an object. This function will allow us (in a round about way) to determine if a given basis element is positive or negative.

```
> sgn := x->piecewise(x>0,1,x<0,-1,0):
```

Recall that multiplication of two tensors in the algebra $\mathbb{K}\otimes\mathbb{M}$ is given by

$$(a\otimes x)\cdot(b\otimes y) = (a\cdot b)\otimes(x\cdot y).$$

Therefore, we define the product of two basis elements by finding the product in each of the algebras using the tables. Then, we extract the sign of the products $a\cdot b$ and $x\cdot y$ and multiply them together. Because the bases of the algebras are orthonormal, we know that the product of two basis elements of $\mathbb{K}\otimes\mathbb{M}$ will be exactly one basis element of

$\mathbb{K} \otimes \mathbb{M}$ with a possible change of sign. Therefore, to encode the multiplication process, we need only track the indices in the product, the index of the result, and the sign of the result.

For example, let's multiply $e_3 \otimes i$ and $e_7 \otimes 1$. We see that

```
> KTable[3,7];
```

```
-5
```

and

```
> MTable[2,1];
```

```
2
```

We see that the product of the signs will be negative.

```
> sgn(-5)*sgn(2);
```

```
-1
```

Finally, we identify which place in the array the basis element $e_5 \otimes i$ is stored in.

```
> BasisIndex[5,2];
```

```
10
```

The procedure for the product of two basis elements in $\mathbb{K} \otimes \mathbb{M}$ is given by

```
> prod:=(u,v)->sgn(KTable[u[1],v[1]])*sgn(MTable[u[2],v[2]])*
                  BasisIndex[abs(KTable[u[1],v[1]]),
                             abs(MTable[u[2],v[2]])
                            ]:
```

This function can be used to compute the product shown above as follows:

```
> prod( BasisIndex[6] , BasisIndex[13] );
```

```
-10
```

which matches the results we computed by hand.

Next, using the table for the sign of the conjugate of a basis element, we define a procedure for computing the actual conjugate of a general object of the tensor algebra.

```
> conj := U->Array(1..dimKM,i->ConjugateSign[i]*U[i]):
```

Because the bases for the algebra $\mathbb{K}$ and $\mathbb{M}$ are orthonormal, we can use the Kronecker delta function for calculating the inner product.

```
> delta := (i,j)->piecewise(i=j,1,0):
```

Now we are ready to construct a rule for multiplying elements of $\mathbb{K} \otimes \mathbb{M}$ together. This product procedure is defined to be `&*`. To save space in memory, a zero element will be identified as a scalar, not an array. This means that we need to have a check in place to identify arrays and scalars. If one of the input values is zero, then a zero is immediately returned. If two arrays are given, call them `X` and `Y`, we search for the nonzero entries in each array.

```
> nx := ArrayTools:-SearchArray(X):
> ny := ArrayTools:-SearchArray(Y):
```

These nonzero entries tell us which tensors are used to make the object as well as the coefficients in front of the tensor. So, for example,

$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 \end{pmatrix} = 2(e_1 \otimes 1) - (e_3 \otimes i) - 3(e_8 \otimes i).$$

We then loop through each of these nonzero entries and compute the product using the rule for tensor multiplication. The results of each tensor product are then added together. This is essentially the distributive property in action. In the following segment of code, the variable `Z` is a new element in $\mathbb{K} \otimes \mathbb{M}$. We start by creating a zero array and fill in the entries.

```
> Z:=Array(1..dimKM):
> for xi in nx do
>   for yi in ny do
```

```
>     # Compute the product of elements in KxM.
>     nn := prod(BasisIndex[xi],BasisIndex[yi]):
>     # Note that 'nn' identifies the tensor created and X[xi]
>     # and Y[yi] are the coefficients. So we just need to add
>     # the new coefficient in the proper slot in the array
>     Z(abs(nn)) := Z(abs(nn))+sgn(nn)*X[xi]*Y[yi]:
>   od:
> od:
```

Consider the product of the following elements:

$$\begin{aligned}(2e_2 \otimes 1 - e_3 \otimes i)(3e_2 \otimes i) &= 6(e_2 \cdot e_2) \otimes (1 \cdot i) - 3(e_3 \cdot e_2) \otimes (i \otimes i) \\ &= 6(-1 \otimes i) - 3(-e_4 \otimes -1) \\ &= -6(1 \otimes i) - 3(e_4 \otimes 1).\end{aligned}$$

The computer will multiply the arrays

$$x = \begin{pmatrix} 0 & 0 & 2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$
$$y = \begin{pmatrix} 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and return the array $z = \begin{pmatrix} 0 & -6 & 0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$.

Step 5: Matrices over $\mathbb{K} \otimes \mathbb{M}$

Now that we are able to multiply any two objects from $\mathbb{K} \otimes \mathbb{M}$ together, we are ready to define matrix-matrix multiplication where the entries in the matrices come from $\mathbb{K} \otimes \mathbb{M}$. This will be done simply by using the regular definition of matrix-matrix multiplication,

$$(AB)_{ij} = \sum_{k=1}^{n} A_i^k B_k^j$$

except that we replace the standard product of elements with the procedure '`&*`'. We define the procedure for matrix-matrix multiplication as

```
> '&.' := (U,V)->Matrix(3,3,
                        (ii,jj)->add( U[ii,k]&*V[k,jj] ,k=1..3)
                        ):
```

This convention mirrors the usage of multiplication (`*`) and matrix-matrix multiplication (`.`) typically used in Maple.

Step 6: Making a procedure to compute $\delta_{A,B}(u \otimes r)$.

Next we create a procedure for computing the action of derivations from $\mathfrak{der}(\mathbb{K})$ and $\mathfrak{der}(\mathbb{M})$ on a tensor. Then by linearity of the map, this can be used to compute the action on any element from $\mathbb{K} \otimes \mathbb{M}$. This procedure is called `DerAction`.

```
DerAction:=proc(A,B,ind)
  local r,c,v, action;
  global BasisKM, BasisIndex;

  r,c,v:=ArrayTools:-SearchArray(A[1..-1,ind[1]]):
  if (op(1,r)>0) then
    action:=add(v[k]*BasisKM[BasisIndex[r[k],ind[2]]],k=1..op(1,r)):
  else
    action:=0:
  fi:
  r,c,v:=ArrayTools:-SearchArray(B[1..-1,ind[2]]):
  if op(r)[1]=0 then
    return action;
  fi:
  return action +
         add(v[k]*BasisKM[BasisIndex[ind[1],r[k]]],k=1..op(1,r)):
end proc:
```

The input arguments are matrices `A` and `B` (coming from $\mathfrak{der}(\mathbb{K})$ and $\mathfrak{der}(\mathbb{M})$ respectively) and an integer `ind` specifying a basis element in $\mathbb{K} \otimes \mathbb{M}$. The process returns an array specifying an element from $\mathbb{K} \otimes \mathbb{M}$.

Step 7: Computing the Trace and the Commutator.

Because we will be interested in working with the trace-free part of a matrix, we will create a simple procedure to extract the trace-free part of a matrix. We do this by computing the trace of the matrix and then subtracting one-third of this value (because

the matrix is 3×3) from the diagonal entries. This will ensure that the trace of the output matrix is zero.

```
> TrFree:=proc(U)
    local Tr;
    Tr:=add(U[k,k],k=1..3)/3:
    return Matrix(3,3,(i,j)->piecewise(i=j,U[i,j]-Tr,U[i,j])):
  end proc:
```

Now is a good time to illustrate how some of these procedures work. This means it's time for another example. For simplicity in displaying the objects, we will use the symbolic notation for the basis elements. The code displayed will use the appropriate notation. Let $u = (e_1 \otimes 1) - (e_4 \otimes i) + (e_7 \otimes i)$, $v = (e_3 \otimes 1) - (e_8 \otimes 1)$ and $w = (e_1 \otimes i) + (e_4 \otimes 1)$.

```
> u := Array(1..16, {(1) = 1 , (8) = -1 , (14) = 1} ):
> v := Array(1..16, {(5) = 1, (15) = -1}):
> w := Array(1..16, {(2) = 1, (7)  = 1}):
```

Now let's use these to make three different matrices. Let $U = uE_2^3 - \overline{u}E_3^2$, $V = vE_2^2 - vE_3^3$ and $W = wE_1^1 - wE_3^3$. Note that the spaces in the arrays have been removed so that the full matrix can be displayed on the page. For example,

```
( 1 0 0 0 0 0 0 -1 0 0 0 0 0 1 0 0 )=(1000000-100000100).
```

```
> U := Matrix(3,3,{(2,3)=u,(3,2)=-conj(u)});
```

$$U := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & (1000000\text{-}100000100) \\ 0 & (\text{-}1000000100000\text{-}100) & 0 \end{bmatrix}$$

```
> V := Matrix(3,3,{(2,2)=v,(3,3)=-v});
```

$$V := \begin{bmatrix} 0 & 0 & 0 \\ 0 & (000010000000000\text{-}10) & 0 \\ 0 & 0 & (0000\text{-}100000000010) \end{bmatrix}$$

```
> W := Matrix(3,3,{(1,1)=w,(3,3)=-w});
```

$$W := \begin{bmatrix} (0100001000000000) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & (0\text{-}10000\text{-}1000000000) \end{bmatrix}$$

Now we can multiply the these elements together. Because we will be interesting in computing the commutator of matrices in a short while, we might as well compute that. Before we test the programs, let's make sure we know what we'll be looking for.

$$\begin{aligned} UV &= \left(uE_2^3 - \overline{u}E_3^2\right)\left(vE_2^2 - vE_3^3\right) \\ &= uvE_2^3E_2^2 - \overline{u}vE_3^2E_2^2 - uvE_2^3E_3^3 + \overline{u}vE_3^2E_3^3 \\ &= -\overline{u}vE_3^2 - uvE_2^3 \end{aligned}$$

At this point, we need to compute uv and $\overline{u}v$.

$$\begin{aligned} uv &= (e_1 \otimes 1 - e_4 \otimes i + e_7 \otimes i)(e_3 \otimes 1 - e_8 \otimes 1) \\ &= (e_1 \otimes 1)(e_3 \otimes 1) - (e_1 \otimes 1)(e_8 \otimes 1) - (e_4 \otimes i)(e_3 \otimes 1) + \\ &\quad (e_4 \otimes i)(e_8 \otimes 1) + (e_7 \otimes i)(e_3 \otimes 1) - (e_7 \otimes i)(e_8 \otimes 1) \\ &= e_3 \otimes 1 - e_8 \otimes 1 + e_2 \otimes i - e_5 \otimes i + e_5 \otimes i + e_2 \otimes i \\ &= 2(e_2 \otimes i) + e_3 \otimes 1 - e_8 \otimes 1 \end{aligned}$$

Also, note that $\overline{u} = u$. This means that

$$UV = \left(-2(e_2 \otimes i) - e_3 \otimes 1 + e_8 \otimes 1\right)\left(E_2^3 + E_3^2\right).$$

We can likewise compute the product VU to get

$$VU = \left(-2(e_2 \otimes i) + e_3 \otimes 1 - e_8 \otimes 1\right)\left(E_2^3 + E_3^2\right)$$

so that

$$UV - VU = 2(-e_3 \otimes 1 + e_8 \otimes 1)\left(E_2^3 + E_3^2\right).$$

We can now check our result against the computer and see that they are in agreement.

```
> (U &. V) - (V &. U);
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & (0000\text{-}200000000020) \\ 0 & (0000\text{-}200000000020) & 0 \end{bmatrix}$$

For the next computation, we will simply illustrate the use of the procedures. First off, let's compute the commuator of V and W.

```
> (V &. W) - (W &. V);
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & (00200000\text{-}20000000) \end{bmatrix}$$

Because this result is not trace-free, this is a good matrix to use to test our function `TrFree`.

```
> TrFree( (V &. W) - (W &. V) );
```

$$\begin{bmatrix} \left(00\text{-}\frac{2}{3}00000\frac{2}{3}0000000\right) & 0 & 0 \\ 0 & \left(00\text{-}\frac{2}{3}00000\frac{2}{3}0000000\right) & 0 \\ 0 & 0 & \left(00\frac{4}{3}00000\text{-}\frac{4}{3}0000000\right) \end{bmatrix}$$

Symbolically, this result is equivalent to

$$[V, W]_0 = \frac{2}{3}(e_2 \otimes 1 + e_5 \otimes 1)\left(E_1^1 + E_2^2 - 2E_3^3\right).$$

Step 8: Making a basis of $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$.

At this point we need to create a basis for the algebra $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$. The following operations will be contained in a procedure called `sa3_Basis`. While there are many ways to go about this, if we are particular about the construction, it will be easier to identify a given matrix in $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$ as a linear combination of the basis elements.

We know that matrices with entries on the main diagonal must be pure imaginary (because they are skew-Hermitian). The other matrices in the basis will be skew-Hermitian matrices whose entries do not lie on the main diagonal. There are two types

of matrices to consider. The first type of matrix is $uE_i^i - uE_3^3$ where $u \in \mathrm{Im}(\mathbb{K} \otimes \mathbb{M})$ and $i = 1, 2$. The second matrix is of the form $vE_i^j - \overline{v}E_j^i$ where $i \neq j$ and $v \in \mathbb{K} \otimes \mathbb{M}$.

First we identify which basis elements of $\mathbb{K} \otimes \mathbb{M}$ are equal their conjugate. If it is equal to its conjugate, then a `1` (true) is returned, otherwise a `0` (false) is returned.

```
> Test := map(x->piecewise(ArrayTools:-IsEqual(x,conj(x)),1,0),
                  BasisKM):
```

We can then use this array to identify the real basis elements in $\mathbb{K} \otimes \mathbb{M}$

```
> RealInd := ArrayTools:-SearchArray(Test):
> numR := ArrayTools:-NumElems(RealInd):
```

as well as the imaginary basis elements.

```
> ImInd := ArrayTools:-SearchArray(1-Test):
> numI := ArrayTools:-NumElems(ImInd):
```

Having identified the real and imaginary basis elements of $\mathbb{K} \otimes \mathbb{M}$, we are now prepared to begin building the skew-Hermitian matrices whose entries are not on the main diagonal. First, initialize a storage array `HM`, and a lookup table `saBasisIndex`.

The dimension of $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$ is found to be the number of ways we can have an element from $\mathbb{K} \otimes \mathbb{M}$ in an entry from the upper triangular part of the matrix plus the number choices there are for putting an imaginary basis element in the first or second entry on the diagonal (the third entry is already taken). These values allow us to accurately preallocate space for the storage array. We will also initialize a lookup table for identifying the basis elements.

```
> num:= 3*dimKM+2*numI:
> HM := Array(1..num):
> saBasisIndex := table( [] ):
```

For each entry which is not on the main diagonal, we place a basis element in the upper part of the matrix, and the negative of its conjugate in the lower part of the matrix.

Because these matrices are created by systematically cycling through the basis elements of $\mathbb{K} \otimes \mathbb{M}$, we can identify them by storing the order in which they are created along with the index of the entry and the entry itself (similar to the lookup table for elements in the tensor product).

To be consistent in the order in which the matrices are created, we first build the matrices with real objects and then build the matrices with imaginary objects.

```
> inc := 0:
> for ii from 1 to 2 do
>   for jj from (ii+1) to 3 do
>     # Compute the matrices that have real entries.
>     for kk from 1 to numR do
>       inc := inc + 1:
>       HM[inc] := Matrix(3,3,{(ii,jj)=BasisKM[RealInd[kk]],
                               (jj,ii)=-BasisKM[RealInd[kk]]},
                               fill=0):
>       saBasisIndex[ii,jj,RealInd[kk]]:=inc:
>       saBasisIndex[inc]:=[ii,jj,RealInd[kk]]:
>     od:
>     # Compute the matrices that have imaginary entries.
>     for kk from 1 to numI do
>       inc:=inc+1:
>       HM[inc] := Matrix(3,3,{(ii,jj)=BasisKM[ImInd[kk]],
                               (jj,ii)=BasisKM[ImInd[kk]]},
                               fill=0):
>       saBasisIndex[ii,jj,ImInd[kk]]:=inc:
>       saBasisIndex[inc]:=[ii,jj,ImInd[kk]]:
>     od:
>   od:
> od:
```

Next we build the matrices whose entries lie on the main diagonal. For the basis elements, we will use matrices of the form

$$\begin{bmatrix} x & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -x \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & x & 0 \\ 0 & 0 & -x \end{bmatrix}$$

where $x \in \Im(\mathbb{K} \otimes \mathbb{M})$ is a pure imaginary basis element.

```
> for ii from 1 to 2 do
>   for kk from 1 to numI do
>     inc:=inc+1:
>     HM[inc] := Matrix(3,3,{(ii,ii)=BasisKM[ImInd[kk]],
                            (3,3)=-BasisKM[ImInd[kk]]},fill=0):
>     saBasisIndex[ii,ii,ImInd[kk]]:=inc:
>     saBasisIndex[inc]:=[ii,ii,ImInd[kk]]:
>   od:
> od:
```

The array `HM` is the array storing the 3×3 matrices. The table `saBasisIndex` allows us to look up a basis matrix by identifying the index for an entry as well as the entry itself. This means that we can determine the order in which the matrix with Nth basis element of $\mathbb{K} \otimes \mathbb{M}$ in the $(i, j)th$ entry can be found by entering these three integers into the table `saBasisIndex`. For example,

```
> saBasisIndex[1,3,4];
                                  18
```

Similarly, we can find that the 18^{th} basis element of $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$ can be found by

```
> saBasisIndex[18];
                                [1,3,4]
```

which means that we want the matrix with the 4^{th} basis element of $\mathbb{K} \otimes \mathbb{M}$ in the 1^{st} row and 3^{rd} column of the matrix (as well as an entry in the 3^{rd} row and 1^{st} column).

To build the array of basis matrices and the corresponding table explicitly, we call the command `sa3Basis`, which executes the procedures we just outlined,

```
> sa3,saBasisIndex,dN := sa3_Basis():
```

which returns the array of matrices, the lookup table, and the dimension of $\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$.

Step 9: Creating derivations for two given tensors.

Now that we have built the matrices for $\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$, we need to build a procedure for computing the derivation defined by two matrices, $\sum_{ij=1}^{3} D_{X_{ij},Y_{ij}}$. First all notice that the term $D_{X_{ij},Y_{ij}}$ will be zero if either of the entries X_{ij} or Y_{ij} are zero. Therefore, in the procedure, we need only perform the computations for the indices in which both $X_{ij} \neq 0$ and $Y_{ij} \neq 0$. The procedure `TensorDerivation` will take two matrices (`X` and `Y`) and the indices identifying the entries of X and Y which are both nonzero (`NonZeroIndex`).

For each nonzero entry in the matrix, we want to identify the nonzero entries of the array defining an object in $\mathbb{K}\otimes\mathbb{M}$. These arrays of indices for X_{ij} and Y_{ij} are `indx` and `indy` respectively.

Once we have identified the nonzero basis elements from $\mathbb{K}\otimes\mathbb{M}$ that we will be working with, we are ready to begin computing the derivations. First, recall that the derivation is linear with respect to all the arguments. So for example, we can pull off the coefficients of each element and simply multiply those coefficients by the derivation action on the basis elements. For example, consider the following the calculation

$$\begin{aligned}
D_{e_2\otimes i+3e_3\otimes 1,2e_3\otimes i+e_2\otimes 1} &= 2D_{e_2\otimes i,e_3\otimes i} + D_{e2\otimes i,e_2\otimes 1} + 6D_{e_3\otimes 1,e_3\otimes i} + 3D_{e_3\otimes 1,e_2\otimes 1} \\
&= 2\left(\langle i,i\rangle D^{(1)}_{e_2,e_3} \oplus \langle e_2,e_3\rangle D^{(2)}_{i,i}\right) + \left(\langle i,1\rangle D^{(1)}_{e_2,e_2} \oplus \langle e_2,e_2\rangle D^{(2)}_{i,1}\right) \\
&+ 6\left(\langle 1,i\rangle D^{(1)}_{e_3,e_3} \oplus \langle e_3,e_3\rangle D^{(2)}_{1,i}\right) + 3\left(\langle 1,1\rangle D^{(1)}_{e_3,e_2} \oplus \langle e_3,e_2\rangle D^{(2)}_{1,1}\right) \\
&= \left(2D^{(1)}_{e_2,e_3} + 3D^{(1)}_{e_3,e_2}\right) \oplus 0 \\
&= -D^{(1)}_{e_2,e_3} \oplus 0.
\end{aligned}$$

We see that the derivation acting on a combination of tensors generates derivations in $\mathfrak{der}(\mathbb{K})$ and $\mathfrak{der}(\mathbb{M})$, meaning that there are two objects created. We will compute them individually (mostly for the purpose of making the code more readable). Each derivation is produced as follows (note that `xi` and `yi` are the indices specifying basis elements in $\mathbb{K}\otimes\mathbb{M}$).

```
> # Compute the product of the coefficients.
> C := X[xi]*Y[yi]:
> # Compute the derivation action based on the elements from the
  # first algebra.
> DER := DerK[ BasisIndex[xi][1] , BasisIndex[yi][1] ]:
> dij := delta(BasisIndex[xi][2],BasisIndex[yi][2]):
> # Multiply them all together with the delta function.
> C*dij*DER;
```

The three steps outlined above will be combined into one step down below (see `D1` and `D2`). The same procedure is followed for the derivation action on elements from the algebra $\mathbb{M}$. The terms produced are then added together (in their respective algebras) with previously computed terms. The general procedure for computing the derivation action on two matrices is given in the next segment of code.

```
> TensorDerivation:=proc(X,Y,NonZeroIndex)
    local indx, indy, DD, xi, yi, ii, jj, N;
    # Initialize a zero derivation.
    DD := [ DerK[1,1] , DerM[1,1] ]:

    for N in NonZeroIndex do
      # Find the nonzero coefficients of the entry in X.
      indx := ArrayTools:-SearchArray(X(N)):
      # Find the nonzero coefficients of the entry in Y.
      indy := ArrayTools:-SearchArray(Y(N)):
      for xi in indx do
        for yi in indy do
          # convert the running index to the tuple index to
          # identify the basis elements from the underlying
          # algebras K and M.
          D1 := X(N)[xi]*Y(N)[yi]*
                delta(BasisIndex[xi][2],BasisIndex[yi][2])*
                DerK[BasisIndex[xi][1],BasisIndex[yi][1]]:
```

```
            D2 := X(N)[xi]*Y(N)[yi]*
                  delta(BasisIndex[xi][1],BasisIndex[yi][1])*
                  DerM[BasisIndex[xi][2],BasisIndex[yi][2]]:
            DD := DD+[D1,D2]:
          od:
        od:
      od:
      return eval(DD);
    end proc:
```

Step 10: Computing the Lie bracket.

Now that we have the ability to perform each of the necessary calculations present in the definition of the Lie bracket, we are ready to discuss the computation of the bracket and outline what the Lie bracket of two general elements looks like. Note that we will represent an element in the algebra in

$$(U, X) \in \Big(\mathfrak{der}(\mathbb{K}) \oplus \mathfrak{der}(\mathbb{M})\Big) \oplus \mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$$

as an ordered pair. In the code, however, we will represent an object as an ordered triple by splitting the derivation element $U = U_1 \oplus U_2$ (rather than building a large matrix and then trying to break it apart again). So, the symbolic development will treat an element as an ordered pair, the code will treat an element as an ordered triplet.

To understand how the following segment of code works it will be easier to see how the calculation proceeds by using ordered pairs and working out the bracket symbolically. Let $(U, X), (V, Y) \in M(\mathbb{K}, \mathbb{M})$ be given. Note that the brackets occurring in the individual algebras is the commutator.

$$\begin{aligned}
\Big[(U, X), (V, Y)\Big] &= \Big[(U, 0), (V, 0)\Big] + \Big[(U, 0), (0, Y)\Big] + \Big[(0, X), (V, 0)\Big] + \Big[(0, X), (0, Y)\Big] \\
&= \Big([U, V], 0\Big) + \Big(0, U(Y)\Big) + \Big(0, -V(X)\Big) + \Big(\frac{1}{3}\sum_{ij=1}^{3} D_{X_{ij}, Y_{ij}}, [X, Y]_0\Big) \\
&= \Big([U, V] + \frac{1}{3}\sum_{ij=1}^{3} D_{X_{ij}, Y_{ij}} \ , \ [X, Y]_0 + U(Y) - V(X)\Big).
\end{aligned}$$

This calculation helps us understand the code for computing the bracket. The input arguments are two ordered triples, `UU` and `VV`. First we compute the derivation action on the two matrices, and call that `DD`. Recall that this object is a direct sum of linear transformations, so `DD` will be a list of two matrices.

The part of the bracket which is contained in $\mathfrak{der}(\mathbb{K})$ is

```
> W1 := UU[1].VV[1]-VV[1].UU[1]+DD[1]/3:
```

The part of the bracket which is contained in $\mathfrak{der}(\mathbb{M})$ is

```
> W2 := UU[2].VV[2]-VV[2].UU[2]+DD[2]/3:
```

Next we need to compute the parts which are contained in $\mathfrak{sa}_3(\mathbb{K}\otimes\mathbb{M})$. First we compute the commutator of the matrices `UU[3]` and `VV[3]` and throw this into the function `TrFree` to get the trace-free part. Call this matrix `Buv`. Finally, we compute the action of the derivation `[UU[1],UU[2]]` on each element in the matrix `VV[3]` and the action of the derivation `[VV[1],VV[2]]` on each element in the matrix `UU[3]`. We then add all these together (with the appropriate signs!) to obtain `W3`.

```
> Bracket:=proc(UU,VV)
    description "Compute the Lie bracket of elements from
                 der(K)+der(M)+sa3(KxM).";
    local BothNonzero,NonZeroIndex, DD, W1, W2, W3, Bxy, Tr;

    BothNonZero := (x,y)->piecewise(x<>0 and y<>0,1,0):
    # Find the indices for which BOTH matrices are nonzero.
    NonZeroIndex := ArrayTools:-SearchArray(
                    Matrix(3,3,(i,j)->BothNonZero(UU[3][i,j],
                                                  VV[3][i,j]))
                                             ):
    # If there are nonzero entries then compute the
    # derivation action for those entries.
    # Otherwise, the derivation is zero.
```

```
    if op(NonZeroIndex)[1]<>0 then
      DD := TensorDerivation(UU[3],VV[3],NonZeroIndex):
    else
      DD := [DerK[1,1],DerM[1,1]]:
    fi:
    # Compute the commutator of the derivations + one third
    # of the Tensor derivation for each derivation algebra.
    W1 := UU[1].VV[1]-VV[1].UU[1]+DD[1]/3:
    W2 := UU[2].VV[2]-VV[2].UU[2]+DD[2]/3:
    # Compute the commutator for the matrices in sa3(KxM),
    # compute the trace to find the trace-free part of the
    # commutator. Then combine this result with the derivation
    # action on the 3x3 matrices.
    Buv := TrFree( UU[3]&.VV[3]-VV[3]&.UU[3] ):
    W3:=Buv-
        Matrix(3,3,{(1,1)=Tr,(2,2)=Tr,(3,3)=Tr})+
        DerivationAction(UU[1],UU[2],VV[3]-
        DerivationAction(VV[1],VV[2],UU[3])):
    return [W1,W2,W3]:
  end proc:
```

The result of the bracket is another ordered triple. In order to really be able to use these procedures, we need to construct a basis of $M(\mathbb{K},\mathbb{M}) = \mathfrak{der}(\mathbb{K}) \oplus \mathfrak{der}(\mathbb{M}) \oplus \mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$ using the bases we have created for each of the algebras used in the construction.

Step 11: Identifying Linear Combinations of Basis Elements.

We start by initializing another basis element lookup table (just as we have done multiple times already). This table will be used for identifying the structure constants for the Lie algebra being created.

```
> LieBasisIndex := table([]):
```

Because we can identify a general object as an ordered triple, and each entry in the triple can be written as a linear combination of the basis elements of the respective algebra,

we will arrange the lookup table using triples of integers where the integer in each slot identifies the basis element for that particular algebra.

We begin by identifying the elements in the derivation algebra $\mathfrak{der}(\mathbb{K})$.

```
> inc := 0:
> for ii from 1 to DerK_dim do
>   inc:=inc+1:
>   LieBasisIndex[inc]:=[ii,0,0]:
>   LieBasisIndex[ii,0,0]:=inc:
> od:
```

Then we do the same for the derivation algebra $\mathfrak{der}(\mathbb{M})$.

```
> for jj from 1 to DerM_dim do
>   inc:=inc+1:
>   LieBasisIndex[inc]:=[0,jj,0]:
>   LieBasisIndex[0,jj,0]:=inc:
> od:
```

Finally, we assign the indices for the elements of $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$.

```
> for kk from 1 to ArrayTools:-NumElems(sa3) do
>   inc:=inc+1:
>   LieBasisIndex[inc]:=[0,0,kk]:
>   LieBasisIndex[0,0,kk]:=inc:
> od:
```

Next, we define a simple procedure for using the indices from the look up table to explicitly build a basis element of the full algebra. The procedure `AlgebraBasis` takes three integers as input (though in practice, only one at a time will be nonzero). If the first entry is nonzero, then the first entry in the triple will be a basis element from $\mathfrak{der}(\mathbb{K})$ and the other two are zero. If the second argument is nonzero, then the second entry is a basis element from $\mathfrak{der}(\mathbb{M})$ and the others are zero. Finally, if the third argument is nonzero, then the third entry is a basis element from $\mathfrak{sa}_3(\mathbb{K} \otimes \mathbb{M})$ and the first two entries are zero. If any other arguments are used, a zero object is returned.

```
> AlgebraBasis := (i,j,k) -> piecewise(
                      i>0 and j=0 and k=0,
                         [DerK_Basis[i],DerM[-1,-1],Matrix(3,3)],
                      i=0 and j>0 and k=0,
                         [DerK[-1,-1],DerM_Basis[j],Matrix(3,3)],
                      i=0 and j=0 and k>0,
                         [DerK[-1,-1],DerM[-1,-1],sa3[k]],
                         [DerK[-1,-1],DerM[-1,-1],Matrix(3,3)]
                                      ):
```

With the exception of the procedure for writing a given object as a linear combination of the basis elements (which will be addressed below), we are now ready to begin computing the structure constants. First we create a storage array for the structure constants. Because the bracket of two elements may be a linear combination of multiple elements, this makes it incredibly difficult to determine beforehand how large the matrix will be. Therefore, we will have to increase the size of the array with each new result computed. Therefore, the initial array will be of size zero.

```
> MultTable:=Array(1..0):
```

Now for each pair of basis elements, we use the lookup table `LieBasisIndex` to extract the ordered triple, and then stick that triple into the function `AlgebraBasis` to create the basis element. We then compute the bracket and keep a running index (`inc`) of the objects created.

```
> inc := inc+1:
> obj := Bracket( AlgebraBasis( op(LieBasisIndex[ni]) ),
                  AlgebraBasis( op(LieBasisIndex[nj]) )
                ):
```

We then break apart `obj` into a linear combination of basis elements.

```
> bk := FindLinearCombination( obj, ni, nj):
```

If a zero object was created, then `FindLinearCombination` will return `NULL`. Otherwise, the procedure returns a list of lists. Each element in the (larger) list is of the form `[[ni,nj,nk],L]` which is to be interpreted as (using $\tilde{e}$ to denote basis elements of $M(\mathbb{K},\mathbb{M})$).

$$[\tilde{e}_{ni},\tilde{e}_{nj}] = ... + L\cdot\tilde{e}_{nk} + ...$$

So, for example, using the basis elements we've constructed for $M(\mathbb{O},\mathbb{C})$, let's consider the following elements

$$\tilde{e}_{16} = \Big(0,0,(e_2\otimes i)\cdot\big(E_1^3 - E_3^1\big)\Big)$$
$$\tilde{e}_{17} = \Big(0,0,(e_3\otimes i)\cdot\big(E_1^3 - E_3^1\big)\Big).$$

If we compute the bracket of these two elements, then `Bracket` will return (we will use the symbolic notation to make the display more readable)

$$\frac{2}{3}\Big(-D^{(1)}_{e_2,e_3}\ ,\ 0\ ,\ (e_4\otimes 1)\cdot\big(E_1^1 + E_2^2 - 2E_3^3\big)\Big).$$

If we take this result and evaluate it in the function `FindLinearCombination`, it returns

```
[[[16, 17, 13], -2/3], [[16, 17, 66], 2/3], [[16, 17, 74], 2/3]] ,
```

which means that

$$[\tilde{e}_{16},\tilde{e}_{17}] = -\frac{2}{3}\tilde{e}_{13} + \frac{2}{3}\tilde{e}_{66} + \frac{2}{3}\tilde{e}_{74}.$$

For each bracket, the list of lists which identify the bracket of the two elements is stored in the array. At the end of the whole procedure, the array is converted into a list and passed forward to the '`_DG`' command to create the structure constants for the Lie algebra.

Once the structure constants have been initialized using the command `DGsetup`, the Lie algebra is ready to be used in Maple. In appendix C.7, we create and initialize each of the algebras in Freudenthal's Magic Square. Once the algebras are initialized, we can compute certain properties, such as a Cartan subalgebra, which will allow us to verify that the created Lie algebra matches the algebra in the table.

BIBLIOGRAPHY

[1] John Baez. The Octonions. *Bulletin of the American Mathematical Society*, 39(2):145–205, 2002.

[2] Georgia M. Benkart and J. Marshall Osborn. The Derivation Algebra of a Real Division Algebra. *American Journal of Mathematics*, 103(6):1135–1150, Dec 1981.

[3] William Fulton and Joe Harris. *Representation Theory: A First Course*. Springer-Verlag New York, New York, 1991.

[4] F. Reese Harvey. *Spinors and Calibrations, 9 of Perspectives in Mathematics*. Academic Press, Inc., Boston, MA, 1990.

[5] Thomas W. Hungerford. *Algebra, volume 73 of Graduate Texts in Mathematics*. Springer-Verlag, New York, NY, 1980.

[6] E.E. Levi. Sulla struttura dei gruppi finiti e continui. *Accademia delle Scienze di Torino*, 40:551–565, 1905.

[7] Richard Donald Schafer. *An Introduction to Nonassociative Algebras*, volume 22. Courier Dover Publications, Mineola, NY, 1966.

APPENDIX

A Miscellaneous Results

We will show that every finite-dimensional representation of a Lie algebra can be thought of as a $\mathfrak{g}$-module. Conversely, every $\mathfrak{g}$-module allows for the construction of a representation.

Consider the definitions for a module over a ring and a module over an algebra. [5]

Definition .2. Let R be a ring. A left **R-module** is an additive abelian group A together with a function $R \times A \to A$ with $(r, a) \mapsto r \cdot a$ such that for all $r, s \in R$ and $a, b \in A$:

1. $r \cdot (a + b) = r \cdot a + r \cdot b$.
2. $(r + s) \cdot a = r \cdot a + s \cdot a$.
3. $(rs) \cdot a = r \cdot (s \cdot a)$.

If R has an identity element 1_R and $1_R \cdot a = a$ for all $a \in A$, then A is said to be a **unitary R-module**. If R is a division ring, then a unitary R-module is called a **vector space**. Since every field is a division ring, and we will be working over $\mathbb{C}$, we can treat our vector spaces as (unitary) $\mathbb{C}$-modules.

Definition .3. Let A be an algebra over a field K. A **left A-module** is a unitary left K-module M such that M is a left module over the ring A and

4. $k(a \cdot m) = (ka) \cdot m = a \cdot (km)$

for all $k \in K$, $a \in A$, and $m \in M$.

The third condition can be interpreted that the action on the module must preserve the properties of multiplication in the underlying algebraic structure. Because we will be working with Lie algebras, "multiplication" is given by the bracket. Therefore, we will need to modify the third condition to read $[r, s] \cdot a = r \cdot (s \cdot a) - s \cdot (r \cdot a)$. Therefore, we can define a module over a Lie algebra.

Definition .4. Let $\mathfrak{g}$ be a Lie algebra over a field $\mathbb{F}$. Then a **Lie module**, or $\mathfrak{g}$-module, is an abelian group V such that for every $a, b \in \mathbb{F}$, $x, y \in \mathfrak{g}$ and $u, v \in V$:

1. $(ax + by) \cdot v = a(x \cdot v) + b(y \cdot v)$.
2. $x \cdot (au + bv) = a(x \cdot u) + b(x \cdot v)$.
3. $[x, y] \cdot v = x \cdot (y \cdot v) - y \cdot (x \cdot v)$.

Theorem .2. *A finite-dimensional representation of a Lie algebra* $\mathfrak{g}$ *is equivalent to* $\mathfrak{g}$*-module.*

Proof: Let $\mathfrak{g}$ be a semisimple Lie algebra over $\mathbb{C}$, V be a vector space over $\mathbb{C}$, and ρ a representation of $\mathfrak{g}$. We want to show that V is a left $\mathfrak{g}$-module. To do this, we define the action of $\mathfrak{g}$ on V by $x \cdot v = \rho_x(v)$ and show that this action satisfies Definition .4.

Let $a, b \in \mathbb{C}$, $x, y \in \mathfrak{g}$ and $u, v \in V$. Then, notice that:

$$\begin{aligned}(ax + by) \cdot v &= \rho_{ax+by}(v) \\ &= \rho_{ax}(v) + \rho_{by}(v) \\ &= a\rho_x(v) + b\rho_y(v) \\ &= a(x \cdot v) + b(y \cdot v)\end{aligned}$$

To prove the second condition, consider:

$$\begin{aligned}x \cdot (au + bv) &= \rho_x(au + bv) \\ &= \rho_x(au) + \rho_x(bv) \\ &= a\rho_x(u) + b\rho_x(v) \\ &= a(x \cdot u) + b(x \cdot v)\end{aligned}$$

The third condition is satisfied as follows:

$$\begin{aligned}[x, y] \cdot v &= \rho_{[x,y]}(v) \\ &= \rho_x(\rho_y(v)) - \rho_y(\rho_x(v)) \\ &= x \cdot (y \cdot v) - y \cdot (x \cdot v)\end{aligned}$$

Because all of the conditions are satisfied, we can consider the vector space V together with the mapping ρ as a $\mathfrak{g}$-module with the action defined as $x \cdot v = \rho_x(v)$.

Now, let V be a left module over $\mathfrak{g}$. We can define a maping $\rho : \mathfrak{g} \to \text{end}(V)$ where $\rho(x) = \rho_x$ is the linear map given by $v \mapsto x \cdot v$. To show that this defines a representation, we need to show that $\rho_{[x,y]} = \rho_x \circ \rho_y - \rho_y \circ \rho_x$. Let $v \in V$ and $x, y \in \mathfrak{g}$ be given. Then

$$\begin{aligned}\rho_{[x,y]}(v) &= [x, y] \cdot v \\ &= x \cdot (y \cdot v) - y \cdot x \cdot (y \cdot v) \\ &= \rho_x(\rho_y(v)) - \rho_y(\rho_x(v))\end{aligned}$$

Therefore, we see that ρ is a Lie algebra homomorphism from $\mathfrak{g}$ to end(V). Thus, ρ is a representation of $\mathfrak{g}$. □

This means that given a representation of a Lie algebra we can treat it as a $\mathfrak{g}$-module, or given a $\mathfrak{g}$-module we can make a representation. Both approaches have their advantages. One benefit of $\mathfrak{g}$-modules is that the notation is easier to work with when developing theory or stating theorems. Representations are beneficial for working with explicit examples. Therefore, it is important for the reader to understand when V stands for a $\mathfrak{g}$-module or a vector space.

One of the fundamental tools in working with representations is Shur's Lemma.

Lemma .3 (Schur's Lemma)**.** *If V and W are irreducible $\mathfrak{g}$-modules and $\psi : V \to W$ is a $\mathfrak{g}$-module homomorphism, then*

1. *$\psi = 0$ or ψ is an isomorphism.*
2. *If $V = W$, then $\psi = \lambda I$ for some $\lambda \in \mathbb{C}$.*

Proof: Part 1 Suppose that V and W are irreducible $\mathfrak{g}$-modules and that $\psi : V \to W$ is a $\mathfrak{g}$-module homomorphism. This means that for any $x \in \mathfrak{g}$, $x \cdot \psi(v) = \psi(x \cdot v)$.

Claim: $\ker(\psi)$ is an invariant subspace of V.

Let $x \in \mathfrak{g}$ and $v \in \ker(\psi)$ be given. Then

$$0 = x \cdot \psi(v) = \psi(x \cdot v).$$

This means that $x \cdot v \in \ker(\psi)$ for every $x \in \mathfrak{g}$. Therefore, $x \cdot \ker(\psi) = \ker(\psi)$ for all $x \in \mathfrak{g}$.

Because V is irreducible, this means that $\ker(\psi) = 0$ or $\ker(\psi) = V$. Recall that $\mathrm{Im}(\psi) \cong V/\ker(\psi)$. So, if $\ker(\psi) = 0$ then $\mathrm{Im}(\psi) \cong V$. If $\ker(\psi) = V$, then $\mathrm{Im}(\psi) = 0$.

Claim: $\mathrm{Im}(\psi)$ is an invariant subspace of W.

Recall that $\mathrm{Im}(\psi) = \{\psi(v) \in W \mid v \in V\}$. Let $x \in \mathfrak{g}$ be given. Since $x \cdot v \in V$, we have that $x \cdot \psi(v) = \psi(x \cdot v) \in \mathrm{Im}(\psi)$. This implies that $x \cdot w \in \mathrm{Im}(\psi)$ for all $w \in \mathrm{Im}(\psi)$. Thus, $\mathrm{Im}(\psi)$ is an invariant subspace of W.

By this claim and the irreducibility of W, we have that $\mathrm{Im}(\psi) = 0$ or $\mathrm{Im}(\psi) = W$. Therefore, we conclude that $\psi = 0$ or ψ is an isomorphism.

Proof: Part 2

Now suppose that $V = W$. Because $\mathbb{C}$ is algebraically closed, we know that the the map ψ must have an eigenvalue $\lambda \in \mathbb{C}$. Then the map $\psi - \lambda I : V \to W$ is a $\mathfrak{g}$-module homomorphism with a nonzero kernel. By part (1), we know that $\psi - \lambda I = 0$. Therefore, $\psi = \lambda I$. □

Shur's Lemma shows that every Lie algebra homomorphism between two irreducible representations must either be the zero map or an isomorphism. Therefore, an irreducible representation of a complex semisimple Lie algebra is unique up to equivalence.

B Maple Code

B.1 Algebras

B.1.1 Quaternion Library Code

```
#########################################################################
#                                                                       #
#                          QuaternionData                               #
#                                                                       #
#########################################################################
Quaternions:=proc(algName,strtype::string)
  description "Creates the algebra data structure for the Quaternion
               algebra.";
  local MultTable,type;
  if _params['strtype']=NULL then
    type:="standard":
  else type:=strtype:
  fi:
  if (type="standard" or type="Standard") then
    MultTable:=[
         [[1, 1, 1], 1],[[1, 2, 2], 1],[[1, 3, 3], 1],[[1, 4, 4], 1],
         [[2, 1, 2], 1],[[2, 2, 1],-1],[[2, 3, 4], 1],[[2, 4, 3],-1],
         [[3, 1, 3], 1],[[3, 2, 4],-1],[[3, 3, 1],-1],[[3, 4, 2], 1],
         [[4, 1, 4], 1],[[4, 2, 3], 1],[[4, 3, 2],-1],[[4, 4, 1],-1]
               ]:
  elif type="split" or type="Split" then
    MultTable:=[
         [[1, 1, 1], 1],[[1, 2, 2], 1],[[1, 3, 3], 1],[[1, 4, 4], 1],
         [[2, 1, 2], 1],[[2, 2, 1],-1],[[2, 3, 4], 1],[[2, 4, 3],-1],
         [[3, 1, 3], 1],[[3, 2, 4],-1],[[3, 3, 1], 1],[[3, 4, 2],-1],
         [[4, 1, 4], 1],[[4, 2, 3], 1],[[4, 3, 2], 1],[[4, 4, 1], 1]
               ]:
  else  error "Expected argument to be either 'standard' or 'split'.
               Instead received %1.", type;
  fi:
  return _DG([["Algebra", algName, [4, table( [ ] )]], MultTable ]):
end proc:
```

B.1.2 Octonion Library Code

```
#########################################################################
#                                                                       #
#                            OctonionData                               #
#                                                                       #
#########################################################################
Octonions:=proc(algName,strtype::string,{Version:=1})
  description "Creates the algebra data structure for the Octonion
 algebra.";
  local MultTable,type;
  if _params['strtype']=NULL then
    type:="standard":
  else type:=strtype:
  fi:
  if (type="standard" or type="Standard") then
  if Version=1 then
    # This is the standard Cayley-Dickson construction.
    MultTable:=[
      [[1, 1, 1], 1], [[1, 2, 2], 1], [[1, 3, 3], 1], [[1, 4, 4], 1],
      [[1, 5, 5], 1], [[1, 6, 6], 1], [[1, 7, 7], 1], [[1, 8, 8], 1],
      [[2, 1, 2], 1], [[2, 2, 1],-1], [[2, 3, 4], 1], [[2, 4, 3],-1],
      [[2, 5, 6], 1], [[2, 6, 5],-1], [[2, 7, 8],-1], [[2, 8, 7], 1],
      [[3, 1, 3], 1], [[3, 2, 4],-1], [[3, 3, 1],-1], [[3, 4, 2], 1],
      [[3, 5, 7], 1], [[3, 6, 8], 1], [[3, 7, 5],-1], [[3, 8, 6],-1],
      [[4, 1, 4], 1], [[4, 2, 3], 1], [[4, 3, 2],-1], [[4, 4, 1],-1],
      [[4, 5, 8], 1], [[4, 6, 7],-1], [[4, 7, 6], 1], [[4, 8, 5],-1],
      [[5, 1, 5], 1], [[5, 2, 6],-1], [[5, 3, 7],-1], [[5, 4, 8],-1],
      [[5, 5, 1],-1], [[5, 6, 2], 1], [[5, 7, 3], 1], [[5, 8, 4], 1],
      [[6, 1, 6], 1], [[6, 2, 5], 1], [[6, 3, 8],-1], [[6, 4, 7], 1],
      [[6, 5, 2],-1], [[6, 6, 1],-1], [[6, 7, 4],-1], [[6, 8, 3], 1],
      [[7, 1, 7], 1], [[7, 2, 8], 1], [[7, 3, 5], 1], [[7, 4, 6],-1],
      [[7, 5, 3],-1], [[7, 6, 4], 1], [[7, 7, 1],-1], [[7, 8, 2],-1],
      [[8, 1, 8], 1], [[8, 2, 7],-1], [[8, 3, 6], 1], [[8, 4, 5], 1],
      [[8, 5, 4],-1], [[8, 6, 3],-1], [[8, 7, 2], 1], [[8, 8, 1],-1]
              ]:

  elif Version=2 then
```

```
    MultTable:=[
      [[1, 1, 1], 1], [[1, 2, 2], 1], [[1, 3, 3], 1], [[1, 4, 4], 1],
      [[1, 5, 5], 1], [[1, 6, 6], 1], [[1, 7, 7], 1], [[1, 8, 8], 1],
      [[2, 1, 2], 1], [[2, 2, 1],-1], [[2, 3, 5], 1], [[2, 4, 8], 1],
      [[2, 5, 3],-1], [[2, 6, 7], 1], [[2, 7, 6],-1], [[2, 8, 4],-1],
      [[3, 1, 3], 1], [[3, 2, 5],-1], [[3, 3, 1],-1], [[3, 4, 6], 1],
      [[3, 5, 2], 1], [[3, 6, 4],-1], [[3, 7, 8], 1], [[3, 8, 7],-1],
      [[4, 1, 4], 1], [[4, 2, 8],-1], [[4, 3, 6],-1], [[4, 4, 1],-1],
      [[4, 5, 7], 1], [[4, 6, 3], 1], [[4, 7, 5],-1], [[4, 8, 2], 1],
      [[5, 1, 5], 1], [[5, 2, 3], 1], [[5, 3, 2],-1], [[5, 4, 7],-1],
      [[5, 5, 1],-1], [[5, 6, 8], 1], [[5, 7, 4], 1], [[5, 8, 6],-1],
      [[6, 1, 6], 1], [[6, 2, 7],-1], [[6, 3, 4], 1], [[6, 4, 3],-1],
      [[6, 5, 8],-1], [[6, 6, 1],-1], [[6, 7, 2], 1], [[6, 8, 5], 1],
      [[7, 1, 7], 1], [[7, 2, 6], 1], [[7, 3, 8],-1], [[7, 4, 5], 1],
      [[7, 5, 4],-1], [[7, 6, 2],-1], [[7, 7, 1],-1], [[7, 8, 3], 1],
      [[8, 1, 8], 1], [[8, 2, 4], 1], [[8, 3, 7], 1], [[8, 4, 2],-1],
      [[8, 5, 6], 1], [[8, 6, 5],-1], [[8, 7, 3],-1], [[8, 8, 1],-1]
            ]:
else
  error "Expected version to be 1 or 2. Instead received %1.", version;
fi:
elif type="split" or type="Split" then
if Version=1 then
  MultTable:=[
      [[1, 1, 1], 1], [[1, 2, 2], 1], [[1, 3, 3], 1], [[1, 4, 4], 1],
      [[1, 5, 5], 1], [[1, 6, 6], 1], [[1, 7, 7], 1], [[1, 8, 8], 1],
      [[2, 1, 2], 1], [[2, 2, 1],-1], [[2, 3, 4], 1], [[2, 4, 3],-1],
      [[2, 5, 6],-1], [[2, 6, 5], 1], [[2, 7, 8],-1], [[2, 8, 7], 1],
      [[3, 1, 3], 1], [[3, 2, 4],-1], [[3, 3, 1],-1], [[3, 4, 2], 1],
      [[3, 5, 7],-1], [[3, 6, 8], 1], [[3, 7, 5], 1], [[3, 8, 6],-1],
      [[4, 1, 4], 1], [[4, 2, 3], 1], [[4, 3, 2],-1], [[4, 4, 1],-1],
      [[4, 5, 8],-1], [[4, 6, 7],-1], [[4, 7, 6], 1], [[4, 8, 5], 1],
      [[5, 1, 5], 1], [[5, 2, 6], 1], [[5, 3, 7], 1], [[5, 4, 8], 1],
      [[5, 5, 1], 1], [[5, 6, 2], 1], [[5, 7, 3], 1], [[5, 8, 4], 1],
      [[6, 1, 6], 1], [[6, 2, 5],-1], [[6, 3, 8],-1], [[6, 4, 7], 1],
      [[6, 5, 2],-1], [[6, 6, 1], 1], [[6, 7, 4], 1], [[6, 8, 3],-1],
      [[7, 1, 7], 1], [[7, 2, 8], 1], [[7, 3, 5],-1], [[7, 4, 6],-1],
      [[8, 1, 8], 1], [[8, 2, 7],-1], [[8, 3, 6], 1], [[8, 4, 5],-1],
```

```
      [[8, 5, 4],-1], [[8, 6, 3], 1], [[8, 7, 2],-1], [[8, 8, 1], 1]
             ]:
elif Version=2 then
  MultTable:=[
      [[1, 1, 1], 1], [[1, 2, 2], 1], [[1, 3, 3], 1], [[1, 4, 4], 1],
      [[1, 5, 5], 1], [[1, 6, 6], 1], [[1, 7, 7], 1], [[1, 8, 8], 1],
      [[2, 1, 2], 1], [[2, 2, 1],-1], [[2, 3, 5], 1], [[2, 4, 8], 1],
      [[2, 5, 3],-1], [[2, 6, 7], 1], [[2, 7, 6],-1], [[2, 8, 4],-1],
      [[3, 1, 3], 1], [[3, 2, 5],-1], [[3, 3, 1],-1], [[3, 4, 6], 1],
      [[3, 5, 2], 1], [[3, 6, 4],-1], [[3, 7, 8], 1], [[3, 8, 7],-1],
      [[4, 1, 4], 1], [[4, 2, 8],-1], [[4, 3, 6],-1], [[4, 4, 1], 1],
      [[4, 5, 7], 1], [[4, 6, 3],-1], [[4, 7, 5], 1], [[4, 8, 2],-1],
      [[5, 1, 5], 1], [[5, 2, 3], 1], [[5, 3, 2],-1], [[5, 4, 7],-1],
      [[5, 5, 1],-1], [[5, 6, 8], 1], [[5, 7, 4], 1], [[5, 8, 6],-1],
      [[6, 1, 6], 1], [[6, 2, 7],-1], [[6, 3, 4], 1], [[6, 4, 3], 1],
      [[6, 5, 8],-1], [[6, 6, 1], 1], [[6, 7, 2],-1], [[6, 8, 5],-1],
      [[7, 1, 7], 1], [[7, 2, 6], 1], [[7, 3, 8],-1], [[7, 4, 5],-1],
      [[7, 5, 4],-1], [[7, 6, 2], 1], [[7, 7, 1], 1], [[7, 8, 3],-1],
      [[8, 1, 8], 1], [[8, 2, 4], 1], [[8, 3, 7], 1], [[8, 4, 2], 1],
      [[8, 5, 6], 1], [[8, 6, 5], 1], [[8, 7, 3], 1], [[8, 8, 1], 1]]:
else
  error "Expected version to be 1 or 2. Instead received %1.", version;
fi:
  else
    error "Expected first argument to be either 'standard' or 'split'.
 Instead received %1.", type;
  fi:
  return _DG([["Algebra", algName, [8, table( [ ] )]], MultTable ]):
end proc:
```

B.1.3 Jordan Algebra Library Code

```
#############################################################################
#                                                                           #
#                          JordanAlgebraModule                              #
#                                                                           #
#############################################################################
JordanAlgebraData:=module()
  description "Creates the algebra data structure for a Jordan algebra.";
  local ModuleApply, JordanAlgebraBasis, JordanProduct;

ModuleApply:=proc(n::integer,algstr::string,newalg)
  local algstring, AD, B, IND, N, M, f, MultTable, i, ii, jj, G, S,
        CoeffSet, sol, prod, r, c, v, obj, vars, oldSafeMode, RealAlgebra,
        ComplexAlgebra, Prod1, Prod2, inc, CommuteProd;
  global MDim, alg;

  # Create the algebra over the real numbers.
  RealAlgebra:=proc()
    return _DG([["Algebra", alg, [1, table( [ ] )]], [[[1, 1, 1], 1]] ]):
  end proc:
  # Create the algebra over the complex numbers.
  ComplexAlgebra:=proc()
    return _DG([["Algebra", alg, [2, table( [ ] )]],
                [[[1, 1, 1], 1], [[1, 2, 2], 1],
                [[2, 1, 2], 1], [[2, 2, 1], -1]] ]):
  end proc:

  if algstr="R" then
    AD:=RealAlgebra():
  elif algstr="C" then
    AD:=ComplexAlgebra():
  elif algstr="Q" then
    AD:=AlgebraLibraryData("Quaternions",alg,type="Standard"):
  elif algstr="O" then
    AD:=AlgebraLibraryData("Octonions",alg,type="Standard"):
  else
    error "Expected 2nd argument to be of the form \"R\", \"C\",
```

```
        \"Q\", \"O\" ";
fi:
oldSafeMode := DifferentialGeometry:-Preferences("SafeMode", true):
# Load in the algebra that will be used as entries in the matrices
DifferentialGeometry:-DGsetup(AD, [seq(_x||i,i=0..op(AD)[1,3,1]-1)],[o]):
MDim:=n:
B,IND:=JordanAlgebraBasis():

B:=ArrayTools:-Concatenate(2,Array(1..1,i->B[1]-B[2]/2-B[3]/2),
                           Array(1..1,i->B[1]/2+B[2]/2-B[3]),
                           B[4..-1]):
# Because a Jordan algebra is a commutative algebra, we need only
# compute half of the products. The procedure below swaps the indices
# on a product to obtain the remainder of the product rules.

#------------Define the local procedure "CommuteProd"--------------#
CommuteProd:=proc(x)
  local swap;
  swap:=z->[[z[1,2],z[1,1],z[1,3]],z[2]]:
  return map(swap,x):
end proc:
#----------------End local procedure "CommuteProd"-----------------#

# Get the number of basis elements
N:=ArrayTools:-NumElems(B):
# Create a list of dummy variables
vars:=[seq(_t||i,i=1..N)]:
M:=map(evalDG,add(vars[i]*B[i],i=1..N)):
f:=DifferentialGeometry:-Tools:-DGinfo(alg,"FrameBaseVectors"):
# Compute the square of each element.
Prod1:=Array(1..N):
for ii from 1 to N do
  G:=map(evalDG, JordanProduct(B[ii],B[ii])-M ):
  S:=convert(G,set):
  CoeffSet:=map(Tools:-DGinfo,S,"CoefficientSet"):
  sol:=solve( map(op,CoeffSet) ,convert(vars,set)):
  prod:=convert(subs(sol,vars),Array):
  r,c,v:=ArrayTools:-SearchArray(prod):
```

```
    obj:=Array(1..ArrayTools:-NumElems(r),i->[[ii,ii,r[i]],v[i]]):
    Prod1[ii]:=convert(obj,list):
  od:
  # Compute half of all the products which are not the square of an
  # element. Computing these products and then swaping the indices in the
  # multiplication table very nearly cuts the processing time in half. In
  # addition, this allows us to hard code results in a way that does not
  # take up nearly as much space, making the file size of this document
  # much smaller.
  Prod2:=Array(1..1/2*N*(N-1)):
  inc:=1:

  for ii from 1 to N do
    for jj from ii+1 to N do
      G:=map(evalDG, JordanProduct(B[ii],B[jj])-M ):
      S:=convert(G,set):
      CoeffSet:=map(Tools:-DGinfo,S,"CoefficientSet"):
      sol:=solve( map(op,CoeffSet) ,convert(vars,set)):
      prod:=convert(subs(sol,vars),Array):
      r,c,v:=ArrayTools:-SearchArray(prod):
      obj:=Array(1..ArrayTools:-NumElems(r),i->[[ii,jj,r[i]],v[i]]):
      Prod2[inc]:=convert(obj,list):
      inc:=inc+1:
    od:
  od:

  DifferentialGeometry:-RemoveFrame(alg):
  DifferentialGeometry:-Preferences("SafeMode", oldSafeMode);

  # Combine all the product rules along with the commuted product rules.
  MultTable:=ArrayTools:-Concatenate(2,Prod1,
                                     Prod2,
                                     map(CommuteProd,Prod2)
                                    ):
  MultTable:=map(x->x[],MultTable):
  return _DG([["Algebra",newalg,[N,table( [] )]],
                          convert(MultTable,list)]):
end proc:
```

```
#-----------Define the local procedure "JordanAlgebraBasis"------------#
JordanAlgebraBasis:=proc()
  description "Compute the basis elements for the Jordan algebra.";
  local V, Dim, N, Basis, inc, ri, ci, num, IND;
  global MDim, alg;
  # Retrieve the basis elements for the algebra
  V:=Vector(DifferentialGeometry:-Tools:-DGinfo(alg,"FrameBaseVectors")):
  # Get the dimension of the algebra
  Dim:=DifferentialGeometry:-Tools:-DGinfo(alg,"FrameBaseDimension"):
  # Create an array to store the matrices composing the basis.
  N:=1/2*MDim*(MDim-1):
  Basis:=Array(1..N*Dim+MDim,i->Matrix(MDim,MDim)):
  # Create an array to store the index to the nonzero elements for each
  # matrix. This is used for identifying nonzero elements in a product
  IND:=Array(1..N*Dim+MDim):
  inc:=1:
  # The diagonal elements in the Jordan algebra are real. So create the
  # matrices with only a real element on the main diagonal.
  for ri from 1 to MDim do
    Basis[inc][ri,ri]:=V[1]:
    IND[inc]:=<ri,ri>:
    inc:=inc+1:
  od:
  # The off diagonal elements must satisfy the Hermitian condition. So
  # create these matrices and save the index to the location of the
  # nonzero element in the upper triangular half.
  for ri from 1 to MDim do
    for ci from ri+1 to MDim do
      for num from 1 to Dim do
        Basis[inc][ri,ci]:=V[num]:
        Basis[inc][ci,ri]:=Conjugate(V[num]):
        IND[inc]:=<ri,ci>:
        inc:=inc+1:
      od:
    od:
  od:
  return Basis,IND:
```

```
end proc:
#---------------------End "JordanAlgebraBasis"-------------------------#
#-------------Define the local procedure "JordanProduct"---------------#
JordanProduct:=proc(X,Y)
  description "Define the procedure for computing a product in the
               Jordan algebra";
  local M,ii,jj, MDim, ZeroVec, obj;
  global alg;
  MDim:=ArrayTools:-Size(X)[1]:
  M:=Matrix(MDim,MDim):
  ZeroVec:=DifferentialGeometry:-Tools:-DGzero("vector"):
  # Systematically progress through each of the indices in the matrix
  # and compute the product according to the rules of matrix
  # multiplication. Then use 'evalDG' on the result.
  for ii from 1 to MDim do
    for jj from ii to MDim do
      obj:=DifferentialGeometry:-evalDG(
                   1/2*add(X[ii,k].Y[k,jj]+Y[ii,k].X[k,jj],k=1..MDim)):
      if ZeroVec<>obj then
        M[ii,jj]:=obj:
        M[jj,ii]:=Conjugate(obj):
      fi:
    od:
  od:
  return M:
end proc:
#----------------------End "JordanProduct"---------------------------#

#--------------Define the local procedure "Conjugate"----------------#
Conjugate:=proc(obj)
  description "Computes the conjugate of an DG element";
  local xx, comps, vals, ConjVals, X;
  X:=DifferentialGeometry:-evalDG(obj):
  xx:=op(X)[2]:
  # Retreive the components of the element and store in a Vector
  comps:=Vector(nops(xx),i->xx[i][1][1]):
  # Retrieve the coefficients for each component
  vals:=Vector(nops(xx),i->xx[i][2]):
```

```
  # The sign on each imaginary term is changed. The sign on the real
  # term stays the same.
  ConjVals:=Vector(nops(xx),i->piecewise(i=1 and comps[1]=1,
                                         vals[1],-vals[i])):
  return evalDG(add(
          ConjVals[k]*DifferentialGeometry:-Tools:-DGvector(comps[k]),
                k=1..nops(xx))):
end proc:
#------------------------End "Conjugate"-----------------------------#
end module:
```

B.1.4 Clifford Algebra Library Code

```
#########################################################################
#                                                                       #
#                     CliffordAlgebraModule                             #
#                                                                       #
#########################################################################
CliffordAlgebraData:=module()
  local ModuleApply, CliffordProduct_GeneralForm,
        CliffordProduct_DiagonalForm;
  global QF;
  description "Creates the algebra data structure for a Clifford algebra.
               The relation used is u*v+v*u=-2Q(u,v)*1.";

# NOTE: The relation used for the products is given by
#           w*v=-2Q(v,w)*e-v*w
#       where e is the identity element and Q is the quadratic form
#       (notice the sign on Q!). The sign convention was chosen to
#       follow the work of John C. Baez. This also has the perk that
#       Cliff(0), Cliff(1), and Cliff(2) are then isomorphic to R, C,
#       and H respectively.



#--------------------"CliffordProduct_GeneralForm"-------------------#
# Define the general recursive procedure for the computation and
# simplification of the product of elements in the Clifford algebra.
CliffordProduct_GeneralForm:=proc(Storage::Array)
  local i, prod, coef, V, ii, newProd, newCoef, newElem, U, W;
  global QF, Basis, IsSorted;
  V:=Storage:
  prod:=V[1][1]:
  coef:=V[1][2]:
  # If the coefficient is zero, then remove the element from the array.
  if coef=0 then
    return Array(1..0):
  fi:
  # If the element is already a multiple of a basis element, then no
```

```
  # further processing can be done.
  if IsSorted(prod) then
    return V:
  fi:
  # Find the first index for the element that is not strictly less
  # than its next neighbor.
  ii:=1:
  while ii<nops(prod) and prod[ii]<prod[ii+1] do
    ii:=ii+1:
  od:

  # if the loop above did not reach the end of the element, then
  # simplify the element. Otherwise, return it.
  if ii<nops(prod) then
    if prod[ii]=prod[ii+1] then
      # if the indices are equal, then this process will return one
      # element (as shown below).
      # ...x*(u*u)*y...=...x*(-Q(u,u)*e)*y...
      #                 =-Q(u,u)...x*y...
      newProd:=[ op(prod[1..ii-1]) , op(prod[ii+2..-1]) ]:
      if newProd=[] then
        newProd:=[0]:
      fi:
      newCoef:=-QF[prod[ii],prod[ii]]*coef:
      V[1]:=[newProd,newCoef]:
      # If the coefficient of the new product is zero, then we can
      # delete it. If the new product is not a scalar multiple of a
      # basis element, then pass it on for further simplification.
      if newCoef=0 then
        return Array(1..0):
      else
        if not(IsSorted(newProd)) then
          V:=CliffordProduct_GeneralForm(V):
        fi:
      fi:
elif prod[ii]>prod[ii+1] then
  # if the indices are not equal, then this process will return two
  # elements (as shown below).
```

```
# ...x*(w*v)*y...=...x*(-v*w-2Q(v,w)*e)*y...
#                 =..(-1)x*v*w*y... - ...2Q(v,w)x*y...

# Compute the first element and simplify if necessary.
newProd:=[prod[1..ii-1][],prod[ii+1],prod[ii],prod[ii+2..-1][]]:
newCoef:=-coef:

# If the new product is not a scalar multiple of a basis element,
# pass the new element on for further simplification.
if not(IsSorted(newProd)) then
  U:=Array(1..1,i->[newProd,newCoef]):
  U:=CliffordProduct_GeneralForm(U):
  # Replace the current element with the simplified version, and
  # append any new elements produced to the end of the array.
  if ArrayTools:-NumElems(U)>0 then
    V[1]:=U[1]:
    V:=ArrayTools:-Concatenate(2,V,U[2..-1]):
  else
    V:=V[2..-1]:
  fi:
else
  V[1]:=[newProd,newCoef]:
fi:
# Compute the second element and simplify if necessary.
newProd:=[prod[1..ii-1][],prod[ii+2..-1][]]:
if newProd=[] then
  newProd:=[0]:
fi:
newCoef:=( 2*QF[prod[ii],prod[ii+1]] )*coef:

# If the coefficient of the new product is zero, then we can delete
# it. If the new product is not a scalar multiple of a basis element,
# then pass it on for further simplification. Otherwise, pass this
# solution back.
if newCoef=0 then
  W:=Array(1..0):
else
  if IsSorted(newProd) then
```

```
      W:=Array(1..1,i->[newProd,newCoef]):
    else
      W:=Array(1..1,i->[newProd,newCoef]):
      W:=CliffordProduct_GeneralForm(W):
    fi:
  fi:
  # Append the new results onto the storage array.
  V:=ArrayTools:-Concatenate(2,V,W):
    else
      V[1]:=[prod,coef]:
    fi:
  fi:
  return V:
end proc:
#-----------------end "CliffordProduct_GeneralForm"------------------#



#-------------------"CliffordProduct_DiagonalForm"-------------------#
# Define the procedure for the Clifford product if the quadratic form
# is diagonal. This will reduce the processing time for the product.
CliffordProduct_DiagonalForm:=proc(Storage::Array)
  local i, prod, coef, V, ii, newProd, newCoef, newElem, U, W;
  global QF, Basis, IsSorted;
  V:=Storage:
  prod:=V[1][1]:
  coef:=V[1][2]:
  # If the coefficient is zero, then remove the element from the array.
  if coef=0 then
    return Array(1..0):
  fi:
  # If the element is already a multiple of a basis element, then no
  # further processing can be done.
  if IsSorted(prod) then
    return V:
  fi:
  # Find the first index for each element that is not strictly less
  # than its next neighbor.
```

```
ii:=1:
while ii<nops(prod) and prod[ii]<prod[ii+1] do
  ii:=ii+1:
od:
# if the loop above did not reach the end of the element, then
# simplify the element. Otherwise, return it.
if ii<nops(prod) then
  if prod[ii]=prod[ii+1] then
    # if the indices are equal, then this process will return one
    # element (as shown below).
    # ...x*(u*u)*y...=...x*(-Q(u,u)*e)*y...
    #                 =-Q(u,u)...x*y...
    # Compute the new element and simplify if necessary.
    newProd:=[prod[1..ii-1][],prod[ii+2..-1][]]:
    if newProd=[] then
      newProd:=[0]:
    fi:
    newCoef:=-QF[prod[ii],prod[ii+1]]*coef:
    V[1]:=[newProd,newCoef]:
    # If the new product is not a scalar multiple of a basis element,
    # pass the new element on for further simplification.
    if not(IsSorted(newProd)) then
      V:=CliffordProduct_DiagonalForm(V):
    fi:
  elif prod[ii]>prod[ii+1] then
    # If the indices are not equal, then this process will return
    # one element (as shown below).
    # ...x*(w*v)*y...=...x*(-v*w)*y...
    #                 =..(-1)x*v*w*y...
    # Compute the new element and simplify if necessary.
    newProd:=[prod[1..ii-1][],prod[ii+1],prod[ii],prod[ii+2..-1][]]:
    newCoef:=-coef:
    V[1]:=[newProd,newCoef]:
    # If the new product is not a scalar multiple of a basis element,
    # pass the new element on for further simplification.
    if not(IsSorted(newProd)) then
      V:=CliffordProduct_DiagonalForm(V):
    fi:
```

```
    else
      V[1]:=[prod,coef]:
    fi:
  fi:
  return V:
end proc:
#----------------end "CliffordProduct_DiagonalForm"-----------------#



#---------------------------ModuleApply-----------------------------#
ModuleApply:=proc(n::integer,alg,QuadForm::Matrix)
  description "Computes the multiplication table for a Clifford
               algebra";
  local i, dim, ind, kk, indx, MultTable, ii, jj, prod, C, test,
        makeIndex, NewBasis, V, num, ProdFunc, sz, MakeIndicies;
  global QF, Basis, IsSorted;
  if n<0 then
    error "Expected 1st argument to be a nonnegative integer.";
  fi:
  if n=0 then
    return _DG([["Algebra",alg,[1,table( [] )]], [[[1, 1, 1],1]]]):
  fi:

  # This routine tests if the elements are strictly sorted (identifying
  # if two indices are the same). Once a false condition is reached,
  # the process is terminated.
  IsSorted:=proc(x::list)
    local ii;
    ii:=1:
    while ii<nops(x) do
      if x[ii]>=x[ii+1] then
        return false:
      fi:
      ii:=ii+1:
    od:
    return true:
  end:
```

```
# If a quadratic form is not given, then the default is identity.
if _params['QuadForm']=NULL then
  QF:=Matrix(n,n,shape=identity):
else
  # Check that the input quadratic form is symmetric and is of the
  # proper dimension.
  if (LinearAlgebra:-Equal(QuadForm,QuadForm^+) and
       ArrayTools:-Size(QuadForm)[1]=n) then
    QF:=QuadForm:
  else
    error "Expected 3rd argument to be a symmetric matrix with
           dimension %1", n;
  fi:
fi:

# If the quadratic form is diagonal, then use the faster procedure
# (faster because the off diagonal products are zero, so they do not
# need to be computed). Otherwise, use the general product procedure.
if LinearAlgebra:-Equal(LinearAlgebra:-DiagonalMatrix(
                         LinearAlgebra:-Diagonal(QF)),QF) then
  ProdFunc:=x->CliffordProduct_DiagonalForm(x):
else
  ProdFunc:=x->CliffordProduct_GeneralForm(x):
fi:
# The dimension of the Clifford algebra
dim:=2^n:




# MakeIndicies process: 1) creates all the permutations of 'm'
#                          elements taken from the first 'N' integers
#                       2) sorts each of the lists of permutations
#                       3) returns only the unique sorted lists of
#                          permutations
MakeIndicies:=(N,m)->ListTools:-MakeUnique(
                                  map(sort,combinat:-permute(N,m))
```

```
                                        ,1):
# Create the indices representing the basis elements of the
# Clifford algebra.
Basis:=[[0],seq( op(MakeIndicies(n,kk)) ,kk=1..n)]:

# Create an array to store the multiplication rules.
MultTable:=Array(1..dim,1..dim):
# We know that v*e=e*v=v (where e is the multiplicative identity in
# the Clifford algebra).
for ii from 1 to dim do
  MultTable[ii,1] :=[[[ii,1 ,ii],1 ]]:
  MultTable[1,ii] :=[[[1 ,ii,ii],1 ]]:
od:
for ii from 2 to dim do
  for jj from 2 to dim do
    # Combine the indices representing the product of the two
    # elements.
    prod:=[ op(Basis[ii]) , op(Basis[jj]) ]:
    # If the product produces a basis element, then simply save this
    # in the multiplication table. Otherwise we need to simplify. We
    # use the 'member' function here because this also returns the
    # index that the product equates to.
    if member(prod,Basis,'NN') then
      MultTable[ii,jj]:=[[[ii,jj,NN],1]]:
    else
      # The initial coefficient of the product is 1.
      C:=1:
      V:=Array(1..1,i->[prod,C]):
      # Compute the product of the elements in the Clifford algebra.
      V:=ProdFunc(V):
      num:=ArrayTools:-NumElems(V):
      if num>0 then
        V:=convert(V,list):
        # Find the location of the reduced product in the list of
        # Basis elements. This index identifies what the product of
        # the two elements equals (along with the coefficient).
        ind:=map(x->ListTools:-SearchAll(x[1],Basis),V):
        MultTable[ii,jj]:=[seq([[ii,jj,ind[kk]], V[kk][2]],
```

```
                                    kk=1..num)]:
        else
          MultTable[ii,jj]:=[[[ii,jj,1],0]]:
        fi:
      fi:
    od:
  od:
  unassign('QF');
  MultTable:=map(x->x[],MultTable):
  # Convert the multiplication table array to a list and return the
  # table representing the multiplication rules.
  return _DG([["Algebra",alg,[dim,table( [] )]],
              convert(MultTable,list)]):
end proc:
#--------------------------end ModuleApply----------------------------#
end module:
```

B.2 Representations Code

B.2.1 Fundamental Weights Code

```
#########################################################################
#                                                                       #
#                  FundamentalWeightsProcedure                          #
#                                                                       #
#########################################################################
FundamentalWeights:=proc({algType:=""})
  description "Compute the fundamental weights of a Lie algebra for a
               given system of simple roots or a given Cartan
               subalgebra.";
  local CM, CMinv, SR, rank, FW, FWi, ind, CMS, checkOrder, Props,
        typeTable, endFW, checkSort, NumArgs, argType;
  global RSD:
  # Input arguments:
  #   FundamentalWeights(SR,RSD) - list of Simple Roots and a Root
  #                                Space Decomposition table
  #   FundamentalWeights(Props)  - a table created using
  #                                 SimpleLieAlgebraProperties
  #   FundamentalWeights(CM,SR)  - a Cartan matrix with the simple roots
  #                                used to determine the Cartan matrix
  #   FundamentalWeights(CSA)    - a list of vectors defining a Cartan
  #                                 subalgebra

  NumArgs:=nargs:
  if algType<>"" then
    NumArgs:=NumArgs-1:
  fi:
  if NumArgs=1 then
    if type(eval(args[1]),'table') then
      Props:=eval(args[1]):
      SR:=Props["SimpleRoots"]:
      RSD:=Props["RootSpaceDecomposition"]:
      if verify("CartanMatrix",map(x->x[],[indices(Props)]),'member')
          then
        CM:=Props["CartanMatrix"]:
      else
```

```
      CM:=DifferentialGeometry:-LieAlgebras:-CartanMatrix(SR,RSD):
    fi:
  elif type(eval(args[1]),'list') then
    # Compute the root space decomposition for the given cartan
    # subalgebra.

    RSD:=DifferentialGeometry:-LieAlgebras:-RootSpaceDecomposition(
            args[1]):
    # Compute a system of simple roots using the given cartan
    # subalgebra.
    SR:=DifferentialGeometry:-LieAlgebras:-SimpleRoots(
            DifferentialGeometry:-LieAlgebras:-PositiveRoots(RSD)):
    # Compute the Cartan matrix.
    CM:=DifferentialGeometry:-LieAlgebras:-CartanMatrix(SR,RSD):
  else
    error "Expected a table of properties or a list of vectors.
           Instead received %1.", args[1];
  fi:
elif NumArgs=2 then
  argType:=map(whattype,map(eval,[args])):
  SR:=args[ListTools:-Search('list',argType)];
  if not(SR::('list')('Vector')) then
    error "Expected one of the argument to be a list of Vectors.";
  fi:
  if member('table',argType) then
    RSD:=eval(args[ListTools:-Search('table',argType)]):
    CM:=DifferentialGeometry:-LieAlgebras:-CartanMatrix(SR,RSD):
  elif member('Matrix',argType) then
    CM:=args[ListTools:-Search('Matrix',argType)]:
  else
    error "Expected one of the arguments to be a table or a Matrix";
  fi:
else
  error "Expected 1 or 2 arguments. Instead received %1.", nargs;
fi:
checkOrder := false:
typeTable:=table(["A"=false,"B"=false,"C"=false,"D"=false]):
# The procedure 'CartanMatrixToStandardForm' does not recognize the
```

```
# cartan matrix for so(4). Therefore, we hardcode the order in which
# the simple roots are stored. Otherwise, we convert the cartan matrix
# to standard form and sort the simple roots accordingly. This ensures
# that the fundamental weights will be properly ordered such that the
# values identifying irreducible reps are consistent with
# Fulton & Harris
if LinearAlgebra:-Equal(CM,<<2,0>|<0,2>>) then
  if algType="D" then
    SR:=[<I,I>,<I,-I>]:
  else
    SR:=[<1,1>,<1,-1>]:
  fi:
  typeTable["D"]:=true:
else
  CMS:=[DifferentialGeometry:-LieAlgebras:-
          CartanMatrixToStandardForm(CM,SR)]:
  CM:=CMS[1]:
  SR:=CMS[2]:
  typeTable["A"]:= member("A",CMS[3..-1]):
  typeTable["B"]:= member("B",CMS[3..-1]):
  typeTable["C"]:= member("C",CMS[3..-1]):
  typeTable["D"]:= member("D",CMS[3..-1]):
fi:
# Compute the inverse of the cartan matrix.
CMinv:=CM^(-1):
# Get the rank of the algebra
rank:=nops(SR):
# Compute the fundamental weights
FW := [seq( add( CMinv[n,k]*SR[k] , k=1..rank ) , n=1..rank )]:
# Because the C.M. is symmetric for type "A" algebras, the ordering of
# the F.W. might be exactly reversed. Check this by computing the 1-norm,
# and reverse the order if necessary.
if typeTable["A"] then
  if rank=3 then
    checkSort:=map(x->abs((x^+).CM.x),FW):
    if checkSort<>[2,2,2] then
      ind:=map(attributes,
               sort([
```

```
                    seq(setattribute(evalf(checkSort[i]),i),i = 1..nops(SR))
                        ],`>`)
                  );
          FW:=map(n->FW[n],ind):
          typeTable["A"] := false:
        else
          typeTable["D"] := false:
        fi:
      fi:
      if typeTable["A"] then
        if not ( ListTools:-Sorted(map(x->evalf(norm(x,2)),FW)) ) then
          FW := ListTools:-Reverse(FW):
        fi:
      fi:
    elif typeTable["C"] then
      if rank=2 then
        checkSort:=map(x->piecewise(x[-1]=0,0,1),FW);
        ind:=[ListTools:-SearchAll(1,checkSort)];
        endFW:=1/2*add(FW[ii],
                    ii=[seq(k,k=1..ind[1]-1),seq(k,k=ind[1]+1..rank)]):
        if LinearAlgebra:-Equal(FW[ind[1]][1..-2],endFW[1..-2]) then
          typeTable["B"]:=false:
        else
          FW:=ListTools:-Reverse(FW):
          typeTable["C"]:=false:
        fi:
      fi:
    fi:
    # If a properties table was given as the input, then save the F.W. in
    # the table by adding a new index and return the table. Otherwise,
    # return the list of fundamental weights.
    if type(args[1],'table') then
      Props["FundamentalWeights"]:=FW:
      return eval(Props):
    else
      return FW:
    fi:
  end proc:
```

```
#----------------end procedure "FundamentalWeights"-----------------#
```

B.2.2 Highest Weight Vector Code

```
#########################################################################
#                                                                       #
#                    HighestWeightVectorProcedure                       #
#                                                                       #
#########################################################################
HighestWeightVector:=proc(rho::`DGrepresentation`,
                          csa::`list`,srv::`list`,{output:="all"})
  description "Computes the highest weight vector of a given Lie
               algebra representation with the corresponding weight.";

  local CSA, props, alg, SRV, Properties, SR, numX, M, KX, RepX, V,
        E, N, ii, var, GetWeight, Compute_H_Action, inc_v, inc_p, dim,
        ComputeSimpleRootVectors, K2, HWVector, Weights,
        ProcessVectors, ValidVectors;
  global H, Num;

  # Input:
  #  Rep   - initialized Lie algebra representation
  #   CSA   - Cartan subalgebra (list)
  #   Delta - vectors in Lie algebra corresponding to the Simple roots
  #   output- (optional) "vector" returns the highest weight vector (HWV),
  #           "all" returns the HWV with the corresponding weight

  # Output:
  #  HW -Highest weight (Array of Vectors)
  #  HWV-Highest weight vector (Array of Vectors)
#-----------Define the local procedure "Compute_H_Action"------------#
  Compute_H_Action:=proc(HWVector,KMatrix)
    description "Find the linear combination of vectors in the kernel
                 of the positive root space such that each vector is
                 also a simultaneous eigenvector for the Cartan
                 subalgebra.";
    local nn, EM, newK, ev, V2, ii, comb, HWV, K2, testDiagonal;
    global H, Num;
```

```
    # reassign the input arguments to new variables to save any changes
    # that are made
    HWV:=HWVector:
    K2:=Matrix(convert(HWVector,list)):
    # perform the process for each element in the Cartan subalgebra.
    # If a new linear combination is made, then the process is
    # recursively passed back into this function. The loop is reset
    # (to ensure that the new vectors are also eigenvalues of the
    # previously checked CSA elements).
    for nn from 1 to nops(H) do
      # Compute the action of H[n] on the kernel
      EM:=LinearAlgebra:-LinearSolve(K2,H[nn].K2):
      ev,V2:=LinearAlgebra:-Eigenvectors(EM):
      newK:=Array(1..Num):
      # break the ouput representing the eigenvectors into a list of
      # vectors.
      for ii from 1 to Num do
        comb:=V2[1..-1,ii]:
        newK(ii):=add(HWV[kk]*comb[kk],kk=1..Num):
      od:
      HWV:=newK:
      K2:=Matrix(convert(HWV,list)):
    od:
    # Check that the action of H on the kernel is diagonal. If not,
    # then reprocess the vectors in the kernel.
    for nn from 1 to nops(H) do
      EM:=LinearAlgebra:-LinearSolve(K2,H[nn].K2):
      testDiagonal:=LinearAlgebra:-DiagonalMatrix(
                      LinearAlgebra:-Diagonal(EM)):
      if not(LinearAlgebra:-Equal(EM,testDiagonal)) then
        HWV:=Compute_H_Action(HWV,K2):
      fi:
    od:
    return HWV:
  end proc:
#-----------------end procedure "Compute_H_Action"-------------------#

#------------------Define "ComputeSimpleRootVectors"-----------------#
```

```
# This is used only if a cartan subalgebra is given, but NOT the vectors
# corresponding to the simple roots.
ComputeSimpleRootVectors:=proc(CSA::list)
  description "Computes the vectors in the Lie algebra that correspond
               to the simple roots for a given cartan subalgebra.";
  local RSD, R, E, ind, isImaginary, isReal, ImCheck, ReCheck, PR, SR,
        SRV, k;
  # Compute the root space decomposition
  RSD:=LieAlgebras:-RootSpaceDecomposition(CSA):
  # Retrieve the roots and store them as vectors
  R:=LieAlgebras:-LieAlgebraRoots(RSD):
  # Create the basis vectors used to identify the Positive roots
  R:=convert(R,Array):
  # Create the storage array for the basis vectors.
  E:=Array(1..2*nops(CSA)):
  ind:=1:
  # Create the functions that will identify if an entry if purely real
  # or purely imaginary. (1/0 are returned instead of true/false)
  isImaginary:=x->piecewise(type(x,imaginary),1,0):
  isReal:=x->piecewise(type(x,imaginary) or x=0,0,1):
  for k from 1 to nops(CSA) do
    ImCheck:=map(isImaginary,map(x->x[k],R)):
    ReCheck:=map(isReal,map(x->x[k],R)):
    # If there exists a root were the kth entry is purely imaginary,
    # create a vector with an 'I' in the kth spot.
    if ArrayTools:-HasNonZero(ImCheck) then
      E(ind):=Vector(nops(CSA),i->piecewise(i=k,I,0)):
      ind:=ind+1:
    fi:
    # If there exists a root were the kth entry is real, create a
    # vector with a '1' in the kth spot
    if ArrayTools:-HasNonZero(ReCheck) then
      E(ind):=Vector(nops(CSA),i->piecewise(i=k,1,0)):
      ind:=ind+1:
    fi:
  od:
  # Remove any of the excess places in the array, and convert to a
  # list. This list is the basis that can be used to determine the
```

```
  # positive roots
  E:=convert(map(E,ArrayTools:-SearchArray(E)^+),list):
  R:=convert(R,list):
  # Obtain the positive roots,
  PR:=LieAlgebras:-PositiveRoots(R,E):
  # Compute the simple roots using the system of positive roots.
  SR:=LieAlgebras:-SimpleRoots(PR):
  SRV:=map(x->RSD[convert(x,list)],SR):
  return SRV;
end proc:
#--------------end procedure "ComputeSimpleRootVectors"--------------#

#--------------------begin "HighestWeightVector"---------------------#
# Check the input arguments
if (nargs<1 or nargs>4) then
  error "Expected 1-4 input arguments. Instead received %d",nargs;
fi:
if _params['csa']=NULL then
  # If the cartan subalgebra is missing, then use
  # 'SimpleLieAlgebraProperties' to obtain it along with the vectors
  # corresponding to the Simple Roots.
  props:=LieAlgebras:-SimpleLieAlgebraProperties(op(rho)[1][2,1][1]):
  CSA:=props["CartanSubalgebra"]:
  SR:=map(convert,props["SimpleRoots"],list):
  SRV:=map(x->props["SimpleRootSpaces"][x],SR):
elif _params['csa']<>NULL and _params['srv']=NULL then
  # If the cartan subalgebra is given but not the vectors corresponding
  # to the simple roots, then compute the simple roots using the given
  # cartan subalgebra.
  CSA:=csa:
  SRV:=ComputeSimpleRootVectors(csa):
else
  CSA:=csa:
  SRV:=srv:
fi:

numX:=ArrayTools:-NumElems(X):
# Compute the kernel of the representation of the positive root space
```

```
# (i.e. the collection of vectors that are killed by the representation
# of every element in the positive roots space)
RepX:=Matrix(map(x->LieAlgebras:-ApplyRepresentation(rho,x)^+,SRV))^+:
# For representations with sparse matrices, this method of finding the
# kernel is MUCH faster (using "solve" as opposed to "LA:-NullSpace")
N:=ArrayTools:-Size(RepX):
V:=Vector(N[2],i->x[i]):
V:=eval(V,solve(convert(RepX.V,set))):
var:=indets(V):

HWVector:=Array(1..nops(var)):
for ii from 1 to nops(var) do
  HWVector(ii):=[seq(var[k]=0,k=1..(ii-1)),var[ii]=1,
                 seq(var[k]=0,k=(ii+1)..nops(var))
                ]:
od:
HWVector:=map[2](eval,V,HWVector):



# The dimension of the kernel (dimK) is the number of irreducible
# representations into which 'rho' decomposes.
dim:=max(ArrayTools:-Size(HWVector)):
# Retrieve the representation matrices of the Cartan subalgebra
H:=map[2](LieAlgebras:-ApplyRepresentation,rho,CSA):
# Define a procedure to compute the weight of a given vector in the
# representation space.
GetWeight:=V->LinearAlgebra:-LinearSolve(
               Matrix(V),Matrix(map(h->h.V,H))
                                      )^+:
# Allocate space for the arrays to hold the weights of the vectors,
# the vectors that do not have weights (require further processing),
# and the vectors that are eigenvectors of the cartan subalgebra.
Weights:=Array(1..dim):
ProcessVectors:=Array(1..dim):
ValidVectors:=Array(1..dim):
inc_v:=1:
inc_p:=2:
```

```
# The vectors that do not have a weight need to be processed to find
# the linear combinations which do have weights (the combinations may
# contain complex coefficients).
if inc_p>1 then
  # If there is only one vector that is not an eigenvector for H, then
  # compute the action of H on all the vectors in the null space of X.
  if inc_p=2 then
    ProcessVectors:=HWVector:
    inc_p:=dim+1:
  fi:
  # Trim the array and convert it to a matrix.
  ProcessVectors:=ProcessVectors[1..inc_p-1]:
  K2:=Matrix(convert(ProcessVectors,list)):
  Num:=inc_p-1:
  # Compute the action of H on the null space vectors to find the proper
  # linear combinations such that the vectors passed back have weights.

  ProcessVectors:=Compute_H_Action(ProcessVectors,K2):

  # Save the vectors returned and compute their weights.
  ValidVectors:=ArrayTools:-Concatenate(2,ValidVectors[1..inc_v-1],
                                        ProcessVectors):
  # If there are still issues with a vector not being an eigenvector
  # for H, then compute the action of H on all the null space vectors
  # of X to find the proper linear combination of vectors to produce
  # eigenvectors.
  try
    Weights:=ArrayTools:-Concatenate(2,Weights[1..inc_v-1],
                                     map(GetWeight,ProcessVectors)
                                    ):
  catch:
    K2:=Matrix(convert(HWVector,list)):
    ValidVectors:=Compute_H_Action(HWVector,K2):
    Weights:=map(GetWeight,ValidVectors):
  end try:
fi:
if (output="vector" or output="Vector") then
  return ValidVectors;
```

```
else
  return ValidVectors, Weights;
fi:
end proc:
#----------------------end "HighestWeightVector"---------------------#
```

B.2.3 Decompose Representation Code

```
#########################################################################
#                                                                       #
#                    DecomposeRepresentationModule                      #
#                                                                       #
#########################################################################
DecomposeRepresentation:=module()
  description "Decomposes the representation and identifies the
               irreducible representation.";
  local ModuleApply, ComputeSimpleRoots, ComputeFundamentalWeights,
        printDecomp, GroupAndSortVectors, MaxComplexEntries,
        MaxNonIntegerEntries,
        BasisTransformationForIrreducibleRepresentation;

# DecomposeRepresentation(rho)
# DecomposeRepresentation(rho,Props)
# DecomposeRepresentation(rho,CSA)
# DecomposeRepresentation(...,print=true/false);
# DecomposeRepresentation(...,output=[])

# The values for output can be "Weights", "Vectors", "Marks",
# "Transformation". The order in which they are given in the list is
# the order in which they will be output. If ouput is omitted, then
# all values will be output and they will be given in the order
# "Marks", "Weights", "Vectors", "Transform".

#-----------------------Begin "ModuleApply"-------------------------#
ModuleApply:=proc(Rep::DGrepresentation,{output:="all"})
  local alg, FW, Mark, HWVectors, HighestWeights, HighestWeightMarks,
        FundamentalWeightMatrix, test1, test2, ind, arg, narg, valid,
        L, outargs, ii, indArg, temp, printValue, PR, SC, Iind, Cind,
        nn, OrderIndex, HWM, FindInd, Q, OutArgFcn, Output;
  global Props, RSD, CSA, SR;

  arg:=[args]:
  narg:=nargs:
  # Get the index for the arguments that are of type 'name'.
```

```
ind:=map(x->piecewise(type(op(x)[1],name),1,0),arg):
# Get the index for all the other arguments.
indArg:=[ListTools:-SearchAll(0,ind)]:
ind :=[ListTools:-SearchAll(1,ind)]:
if nops(ind)>0 then
  # reassign the arguments that are of type 'name' to a new
  # variable 'temp'. Then remove those arguments from the
  # original list.
  temp:=map(n->arg[n],ind):
  arg :=map(n->arg[n],indArg);
  narg:=nops(indArg):
  # test for the user specification for 'output'.
  test1:=map(x->piecewise(member('output',{op(x)}),1,0),temp);
  ind:=ArrayTools:-SearchArray(Vector(test1));
  if ArrayTools:-NumElems(ind)>0 then
    valid:=[seq(i,i=1..ind[1]-1),seq(i,i=ind[1]+1..nops(temp))]:
    temp:=map(n->temp[n],valid);
  fi:
  if nops(temp)=0 then
    printValue:=false;
  else
    if nops([op(temp[1])])=2 then
      printValue:=op(temp[1])[2]:
    else printValue:=true:
    fi:
  fi:
fi:
# Get the domain of the representation.
alg:=DifferentialGeometry:-Tools:-DGinfo(Rep,"DomainFrame"):
if narg=1 then
  # Compute the cartan subalgebra, the root space decomposition,
  # the simple roots, and finally the fundamental roots.
  CSA:=DifferentialGeometry:-LieAlgebras:-CartanSubalgebra(alg);
  RSD:=DifferentialGeometry:-LieAlgebras:-
           RootSpaceDecomposition(CSA):
  SR:=DifferentialGeometry:-LieAlgebras:-SimpleRoots(
           DifferentialGeometry:-LieAlgebras:-PositiveRoots(RSD)):
  FW:=DifferentialGeometry:-LieAlgebras:-FundamentalWeights(SR,RSD);
```

```
elif narg=2 then
  if type(arg[2],'table') then
    # If a properties table was given, then retrieve the cartan
    # subalgebra. If the fundamental weights are included in the
    # table, retrieve them, otherwise they must be computed.
    Props:=eval(arg[2]):
    CSA:=Props["CartanSubalgebra"]:
    RSD:=eval(Props["RootSpaceDecomposition"]):
    if verify("FundamentalWeights",map(x->x[],[indices(Props)]),
              'member')  then
      FW:=Props["FundamentalWeights"]:
    else
      SR:=DifferentialGeometry:-LieAlgebras:-SimpleRoots(
            DifferentialGeometry:-LieAlgebras:-PositiveRoots(RSD)):
      Props:=DifferentialGeometry:-LieAlgebras:-
                FundamentalWeights(SR,RSD);
      Props:=FundamentalWeights(eval(Props)):
      FW:=Props["FundamentalWeights"]:
    fi:
    SR:=Props["SimpleRoots"]:
  elif type(arg[2],'list') then
    # With a cartan subalgebra given, compute the root space
    # decomposition, the simple roots, and finally the fundamental
    # roots.
    if type(arg[2,1],'function') then
      CSA:=arg[2]:
      RSD:=DifferentialGeometry:-LieAlgebras:-
                RootSpaceDecomposition(CSA):
      SR:=DifferentialGeometry:-LieAlgebras:-SimpleRoots(
            DifferentialGeometry:-LieAlgebras:-PositiveRoots(RSD)):
      FW:=DifferentialGeometry:-LieAlgebras:-
               FundamentalWeights(SR,RSD);
    else
      error "Expected 2nd argument to be a table or list of
             DGvectors.";
    fi:
  else
    error "Expected 2nd argument to be a table or list of
```

```
          DGvectors.";
  fi:
else
  error "Expected 1 or 2 arguments. Instead received %1.", narg;
fi:

# Compute the highest weight vector(s) of the representation with
# the corresponding weight(s)
HWVectors,HighestWeights:=DifferentialGeometry:-
                          LieAlgebras:-HighestWeightVector(Rep,CSA):

# Convert the set of fundamental weights to matrix form (the jth
# weight becomes the jth column in the matrix)
FundamentalWeightMatrix:=Matrix(FW):
# Any weight can be written as a linear combination of the fundamental
# weights. The coefficients of this linear combinations are positive
# integers, which when stored as an k-tuple (m1,m2,...,mk) is called
# a mark. Create a routine to compute the mark of a given weight.
Mark:=W->LinearAlgebra:-LinearSolve(FundamentalWeightMatrix,W):


# Compute the marks of the highest weights. These marks identify the
# irreducible representations into which the representation decomposes.
HighestWeightMarks:=map(Mark,HighestWeights):

# First we check that the marks are real valued.
Cind:=MaxComplexEntries(HighestWeightMarks,HighestWeights):
if ArrayTools:-NumElems(Cind)>0 then
  for nn from 1 to nops(FW) do
    FW[nn](Cind):=FW[nn](Cind)*I:
  od:
  FundamentalWeightMatrix:=Matrix(FW):
  Mark:=W->LinearAlgebra:-LinearSolve(FundamentalWeightMatrix,W):
  HighestWeightMarks:=map(Mark,HighestWeights):
fi:

# Now we need to check that the marks corresponding to the highest
# weights are both real valued and that the highest weights can be
```

```
# written as a linear combination of the fundamental weights. If
# this is not the case, then the fundamental weights can be adjusted
# accordingly so that they are.

# First we check that the highest weights can be written as a
# positive integer combination of the fundamental weights.

SC:=map(v->Vector(Tools:-GetComponents(convert(v,Vector),FW)),
                                        HighestWeights):
# Find the indices in which the components are not integer.
Iind:=MaxNonIntegerEntries(SC):
if ArrayTools:-NumElems(Iind)>0 then
  # Identify the unique coefficients for each weight.
  SC:=map(x->ListTools:-MakeUnique(convert(x,list),1),SC);
  # Identitfy the unique coefficients overall.
  SC:=ListTools:-MakeUnique(convert(map(op,SC),list),1);
  # Retrieve only the coefficients which are not integers.
  SC:=map(n->SC[n],[ListTools:-SearchAll(false,
                                          map(type,SC,integer))]);
  # Find the least commmon multiple of these coefficients.
  SC:=lcm(op(SC)):
  # Scale the fundamental weights by this value. This will ensure
  # that the marks have been scaled properly.
  FW:=map(x->x*SC,FW):
  # Recompute the marks.
  FundamentalWeightMatrix:=Matrix(FW):
  Mark:=W->LinearAlgebra:-LinearSolve(FundamentalWeightMatrix,W):
  HighestWeightMarks:=map(Mark,HighestWeights):
fi:
# Keep the old order of the marks and create a function that will
# identify the index of the mark located in the array.
HWM:=map(convert,HighestWeightMarks,Vector):
FindInd:=V->convert(ArrayTools:-SearchArray(
              map(x->piecewise(LinearAlgebra:-Equal(x,V),1,0),HWM)
                                          ),list):
# Sort the marks in decreasing order w.r.t. increasing index
HighestWeightMarks:=GroupAndSortVectors(
                       map(convert,HighestWeightMarks,Vector),`>`):
```

```
# Because the sorting algorithms used to sort the marks is recursive,
# we need to wait until the sorting it completed. Once the marks are
# sorted, we obtain the index for the sorting to allow us to sort the
# weights and vectors likewise.

OrderIndex:=map(FindInd,HighestWeightMarks):
OrderIndex:=map(op,ListTools:-MakeUnique(convert(OrderIndex,list),1)):
OrderIndex:=convert(OrderIndex,Array):

HighestWeights:=map(n->HighestWeights[n],OrderIndex):
HWVectors:=map(n->HWVectors[n],OrderIndex):
# Return the solutions in the order that they were requested, or the
# default order if no specification was made. If the user makes an
# unidentifiable request, NULL is returned.
if evalb(printValue=true or printValue=1) then
  printDecomp(HighestWeightMarks);
fi:
if output="all" then
  Q:=BasisTransformationForIrreducibleRepresentation(Rep,HWVectors):
  return HighestWeightMarks, HighestWeights, HWVectors, Q;
else
  if type(output,string) then
    Output:=[output]:
  else
    Output:=output:
  fi:
  L:=map(x->x[1],Output):
  if member("t",L) or member("T",L) then
  Q:=BasisTransformationForIrreducibleRepresentation(Rep,HWVectors):
  fi:



  OutArgFcn:=x->piecewise(x="M" or x="m",HighestWeightMarks,
                          x="W" or x="w", HighestWeights,
                          x="V" or x="v",HWVectors,
                          x="T" or x="t",Q,
                          NULL
```

```
                          ):
    outargs:=map(OutArgFcn,L):
    return outargs[];
  fi:
end proc:
#------------------------End "ModuleApply"-------------------------#

#--------------------Define "MaxComplexEntries"---------------------#
MaxComplexEntries:=proc(VV,WW)::Vector;
  local Num, IND, test1, test2, test, NN, ii:
  Num:=ArrayTools:-NumElems(VV):
  IND:={}:
  NN:=nops(IND):
  for ii from 1 to Num do
    test1:=map(x->piecewise(type(x,nonreal),1,0),VV[ii]):
    test2:=map(x->piecewise(type(x,nonreal),1,0),WW[ii]):
    test:=Vector(ArrayTools:-NumElems(test1),kk->test1(kk)*test2(kk)):
    test:=ArrayTools:-SearchArray(test):
    if ArrayTools:-NumElems(test)>=NN then
      IND:=IND union convert(test,set):
      NN:=nops(IND):
    fi:
  od:
  return convert(convert(IND,list),Vector);
end proc:
#----------------------End "MaxComplexEntries"-----------------------#

#------------------Define "MaxNonIntegerEntries"--------------------#
MaxNonIntegerEntries:=proc(VV)::Vector;
  local Num, IND, test, NN, ii:
  Num:=ArrayTools:-NumElems(VV):
  IND:={}:
  NN:=nops(IND):
  for ii from 1 to Num do
    test:=ArrayTools:-SearchArray(
                  map(x->piecewise(type(x,integer),0,1),VV[ii])
                                ):
    if ArrayTools:-NumElems(test)>=NN then
```

```
      IND:=IND union convert(test,set):
      NN:=nops(IND):
    fi:
  od:
  return convert(convert(IND,list),Vector);
end proc:
#--------------------End "MaxNonIntegerEntries"---------------------#

#-----------------------Define "printDecomp"-------------------------#
printDecomp:=proc(Marks)
  local oplus, G, num, sz, Gamma;
  num := ArrayTools:-NumElems(Marks):
  if num>1 then
    oplus := convert(`&oplus;`,name):
    G := map(x->'Gamma'[convert(x,list)[]],Marks):
    G := [op(add(G[i],i=1..num))]:
    num := nops(G):
    sz := interface(rtablesize):
    interface(rtablesize=2*num):
    print(Vector[row](2*num-1,i->piecewise(modp(i,2)=0,
                                           oplus,G[(i+1)/2])));
    interface(rtablesize=sz):
  else
    G := map(x->'Gamma'[convert(x,list)[]],Marks):
    print(G);
  fi:
  ``:
end proc:
#-------------------------End "printDecomp"--------------------------#

#--------------------Define "GroupAndSortVectors"--------------------#
GroupAndSortVectors:=proc(V::Array,sortorder)::Array;
  global sz,ord;
  local SortVectors;

  SortVectors:=proc(V::Array,indx)::Array;
    global sz,ord;
    local val, num, uniqueVal, temp, ii, ind, testSort, OrderIndex;
```

```
    OrderIndex:=[seq(ii,ii=1..ArrayTools:-NumElems(V))]:
    val := convert(map(x->abs(x[indx]),V),list):
    uniqueVal := sort(ListTools:-MakeUnique(val),ord):
    temp := Array(1..nops(uniqueVal)):
    for ii from 1 to nops(uniqueVal) do
      ind := [ListTools:-SearchAll(uniqueVal[ii],val)]:
      num := nops(ind):

      temp[ii]:=Array(1..num,i->V[ind[i]]):
      if indx+1<=sz then
        testSort:=convert(map(x->abs(x[indx+1]),temp[ii]),list):
        if num>1 and
            (not(ListTools:-Sorted(testSort,ord)) and indx<sz) then
          temp[ii]:=SortVectors(temp[ii],indx+1):
        elif num>1 and
            (not(ListTools:-Sorted(testSort,ord)) and
            indx+1=sz) then
          ind := map(attributes,sort([
                 seq(setattribute(evalf(testSort[i]),i),i = 1..num)
                                    ],ord)):
          temp[ii]:= map(k->temp[ii][k],convert(ind,Array)):
        fi:
      fi:
    od:
    return ArrayTools:-Concatenate(2,seq(temp[k],k=1..nops(uniqueVal))):
  end proc:

  ord:=sortorder:
  sz:=ArrayTools:-NumElems(V[1]):
  return SortVectors(V,1):
end proc:
#----------------------End "GroupAndSortVectors"---------------------#

#------Define "BasisTransformationForIrreducibleRepresentation"------#
BasisTransformationForIrreducibleRepresentation:=proc(Rep,HWV)
  description "Computes a change of basis transformation that will
               decompose a given representation into a direct sum of
               irreducible representations. This is a brute force
```

```
            creation method. In the future, a faster method may be
            used by working with the Weight Lattice of the
            Representation.";
local Delta, NegativeRootSpace, LinearIndependenceTest, Y, dim, ZV,
      Q, tracker, kk ,inc, NN, nn, QQ;
global RSD, CSA, SR;
# This function tests if a given vector is linearly independent
# from a given set of vectors.
LinearIndependenceTest:=proc(V::Vector,Vset::list)
  if GetComponents(V,Vset)=[] then return true:  fi:
  return false:
end proc:
# Determine the negative root spaces
NegativeRootSpace:=map(x->RSD[convert(-x,list)],SR):

# Get the representation matrices for the basis elements of the NRS.
Y:=map[2](LieAlgebras:-ApplyRepresentation,Rep,NegativeRootSpace):
# Get the dimension of the representation space.

dim:=ArrayTools:-Size(Y[1],1):
# Create a zero vector
ZV:=Vector(dim):
# Preallocate a list to contain the created vectors.
Q:=[seq(0,ii=1..dim)]:
# Preallocate a list to distinguish the vectors which span the
# different irreducible representations.
tracker:=[seq(0,ii=1..dim)]:
# Initialize the counters
kk:=1:  # Identifies the vector that is to be acted upon by Y.
inc:=1: # Identifies the next available slot for storage.
for NN from 1 to ArrayTools:-NumElems(HWV) do
  # Store the highest weight vector in the next available slot.
  Q[inc]:=HWV[NN]:
  # Identity the irred. rep. this vector corresponds to
  tracker[inc]:=NN:
  inc:=inc+1:
  # Proceed through the list of created vectors to test the action
  # of every Y. If the newly created vectors are nonzero and is not
```

```
    # a linear combination of the previous vectors, then keep it.
    while kk<inc do
      # Loop through each of the vectors in the NRS.
      for nn from 1 to nops(Y) do
        QQ:=Y[nn].Q[kk]:
        # For a given Y, repeatedly test higher and higher powers of
        # Y until the action kills the vector or produces a vector
        # which is a linear combination of previously created vectors.
        while not(LinearAlgebra:-Equal(ZV,QQ)) and
              LinearIndependenceTest(QQ,Q[1..inc-1]) do
          Q[inc]:=QQ:
          tracker[inc]:=NN:
          inc:=inc+1:
          QQ:=Y[nn].QQ:
        od:
      od:
      # Step up to the newest created vector which has not been
      # acted upon by all the Y.
      kk:=kk+1:
    od:
  od:
  # The set of linearly independent vectors created span the
  # representation space. These vectors provide a change of basis
  # transformation. Return the matrix corresponding to this
  # transformation.
  return Matrix(Q);
end proc:
#--------End "BasisTransformationForIrreducibleRepresentation"-------#
end module:
```

B.3 Magic Square

```
#############################################################################
#                                                                           #
#                        MagicSquareModule                                  #
#                                                                           #
#############################################################################
MagicSquare:=module()
  description "Compute the magic square Lie algebras using the Vinberg
```

```
              symmetric formula.";
  local  ModuleApply, StructureTable, DerivationsTable,
         DerivationAction, FindLinearCombination;
  export `&*`,`&.`,conj, prod, herm, sgn, delta, TrFree, saBasisIndex,
         BasisIndex;

#----------------------Define "StructureTable"----------------------#
StructureTable:=proc(alg)
  description  "Computes the structure constants and stores the rules
               for multiplying basis elements in a table.";
  local Basis, N , StuctEqs, CoefficientsTable, ii, jj, V, NZ, kk, obj;
  global `index/sign`;

  Basis:=DifferentialGeometry:-Tools:-DGinfo(alg,"FrameBaseVectors"):
  N:=nops(Basis):

  # To save on memory and future processing, if one of the input
  # values are negative, then the output is negative. Otherwise, it
  # is positive.
  `index/sign` := proc(Idx::list,Tbl::table,Entry::list)
    if (nargs =2) then
      if assigned(Tbl[ Idx[1],Idx[2] ]) then Tbl[ Idx[1],Idx[2] ];
      elif assigned(Tbl[ -Idx[1],Idx[2] ]) then -Tbl[ -Idx[1],Idx[2] ];
      elif assigned(Tbl[ Idx[1],-Idx[2] ]) then -Tbl[ Idx[1],-Idx[2] ];
      elif assigned(Tbl[ -Idx[1],-Idx[2] ]) then Tbl[ Idx[1],Idx[2] ];
      else  0;
      fi:
    elif Entry = [sign] then  0;
    else  Tbl[op(Idx)] := op(Entry);
    fi:
  end proc:

  # Create the lookup table to identify the product of two elements
  CoefficientsTable:=table(sign):
  # Retrive the structure constants from the DG algebra framework.
  StuctEqs:=DifferentialGeometry:-Tools:-DGinfo(alg,
                                  "LieBracketStructureEquations")[1];
  for ii from 1 to N do
```

```
    for jj from 1 to N do
      obj:=Vector[row](DifferentialGeometry:-GetComponents(
                                                StuctEqs[ii,jj],Basis)):
      CoefficientsTable[ii,jj]:=ArrayTools:-SearchArray(obj)[1]*
                                  obj[ArrayTools:-SearchArray(obj)[1]];
    od:
  od:
  return eval(CoefficientsTable):
end proc:
#------------------------End "StructureTable"------------------------#

#---------------------Define "DerivationsTable"----------------------#
DerivationsTable:=module()
  description "Create a lookup table to recall the derivations for
               D_{a,b} where ‘a’ and ‘b’ are basis elements.";
  local MakeTable, ModuleApply;

  MakeTable:=proc(dim,{Matrixtype:=1})
    global ‘index/ZeroMatrix1‘,‘index/ZeroMatrix2‘;

    # indexing procedure(s) for the table. If an index is entered
    # which does not exist, a zero matrix is returned. Two procedures
    # are required to ensure that the proper sized zero matrices are
    # returned for each algebra.
    if Matrixtype=1 then

      ‘index/ZeroMatrix1‘ := proc(Idx::list,Tbl::table,Entry::list)
        if (nargs = 2) then
          if assigned(Tbl[Idx[1],Idx[2]]) then  Tbl[Idx[1],Idx[2]];
          elif assigned(Tbl[Idx[2],Idx[1]]) then -Tbl[Idx[2],Idx[1]];
          else  Matrix(dim,dim);
          fi:
        elif Entry = [ZeroMatrix1] then  Matrix(dim,dim);
        else  Tbl[op(Idx)] := op(Entry);
        fi:
      end proc:

      return eval(table(ZeroMatrix1));
```

```
  elif Matrixtype=2 then

    `index/ZeroMatrix2` := proc(Idx::list,Tbl::table,Entry::list)
      if (nargs = 2) then
        if assigned(Tbl[Idx[1],Idx[2]]) then  Tbl[Idx[1],Idx[2]];
        elif assigned(Tbl[Idx[2],Idx[1]]) then -Tbl[Idx[2],Idx[1]];
        else  Matrix(dim,dim);
        fi:
      elif Entry = [ZeroMatrix2] then  Matrix(dim,dim);
      else  Tbl[op(Idx)] := op(Entry);
      fi:
    end proc:

    return eval(table(ZeroMatrix2));
  fi:
end proc:

ModuleApply:=proc(CT::table,alg,{type:=1})
  local Associator, Commutator, D, Der, dim, ZM,  xi, yi, A, r,
        MakeObject;
  # Get the dimension of the algebra.
  dim:=DifferentialGeometry:-Tools:-DGinfo(alg,"FrameBaseDimension"):
  MakeObject:=obj->Vector(dim,{abs(obj)=piecewise(obj>0,1,obj<0,-1,0)
                               }):

  # Define the procedure to compute the Associator term: (xy)a-x(ya)
  Associator:=(i,j,m)->MakeObject(CT[CT[i,j],m])
                       -MakeObject(CT[i,CT[j,m]]);

  # Define the procedure to compute the Commutator term: [[x,y],a]
  Commutator:=(i,j,m)->MakeObject(CT[CT[i,j],m])
                       -MakeObject(CT[m,CT[i,j]])
                       -MakeObject(CT[CT[j,i],m])
                       +MakeObject(CT[m,CT[j,i]]):

  # Define the rule of assignment for the derivation.
  D:=(x,y,a)->Commutator(x,y,a)-3*Associator(x,y,a):
```

```
    # Initialize a table with default values set to a zero matrix
    Der:=MakeTable(dim,Matrixtype=type):

    # Create a zero matrix
    ZM:=Matrix(dim,dim):

    for xi from 2 to dim do
      for yi from 2 to dim do
        if xi<>yi then
          A:=Matrix([ seq(D(xi,yi,r),r=1..dim) ]):
          if not(LinearAlgebra:-Equal(A,ZM)) then
            Der[(xi,yi)]:=A:
          fi:
        fi:
      od:
    od:
    return eval(Der):
  end proc:
end module:
#-----------------------End "DerivationsTable"-----------------------#

#---------------------Define "DerivationAction"----------------------#
DerivationAction:=proc(A,B,X)
  description "Compute the action of A+B on an element in the matrix
               algebra of the tensor algebra KxM.";
  local DerAction, Z, ind, mm, xx, cc, indx;
  global BasisIndex;

  DerAction:=proc(A,B,ind)
    local r,c,v, action;
    global BasisKM, BasisIndex;

    r,c,v:=ArrayTools:-SearchArray(A[1..-1,ind[1]]):
    if (op(1,r)>0) then
      action:=add(v[k]*BasisKM[BasisIndex[r[k],ind[2]]],k=1..op(1,r)):
    else action:=0:
    fi:
    r,c,v:=ArrayTools:-SearchArray(B[1..-1,ind[2]]):
```

```
    if op(r)[1]=0 then
      return action;
    fi:
    return action +
            add(v[k]*BasisKM[BasisIndex[ind[1],r[k]]],k=1..op(1,r)):
  end proc:

  Z:=Matrix(3,3):
  # Return the index to the nonzero entries of the matrix. This will
  # reduce the processing time.
  ind:=ArrayTools:-SearchArray(X):
  for mm in ind do
    xx:=X(mm):
    # Get the coefficients on the tensor elements
    cc:=map(n->xx[n],ArrayTools:-SearchArray(xx)):
    # Get the index of the nonzero tensor elements.
    indx:=map(n->BasisIndex[n],ArrayTools:-SearchArray(xx)):
    # Compute the Derivation action on the basis elements of the tensor
    # algebra, multiply by the appropriate coefficient, and add them all
    # together to complete the action on the entire matrix entry.
    Z(mm):=add( cc[kk]*DerAction(A,B,indx[kk]) ,
                kk=1..ArrayTools:-NumElems(indx)):
  od:
  return Z;
end proc:
#-----------------------End "DerivationAction"-----------------------#

#-------------------Define "FindLinearCombination"-------------------#
FindLinearCombination:=proc(Q,alpha,beta)
  description "Given the result of the bracket, 'Q', of the basis
               elements 'alpha' and 'beta' (identified by their
               indices), finds the linear combination of basis
               elements of der(K)+der(M)+sa3(KxM).";
  local R, C, Val, vv, IND, pp, r, c, v, nn, ind, count,
        GeneralDer_KElement, ElementCoeffs_K, GeneralDer_MElement,
        ElementCoeffs_M, nzIndx,obj;
  global IsZero, DerK_dim, DerK_Basis, DerM_dim, DerM_Basis, DerK,
         DerM, saBasisIndex, LieBasisIndex;
```

```
if IsZero(Q) then
  return NULL;
fi:
GeneralDer_KElement:=add(_a[i]*DerK_Basis[i],i=1..DerK_dim):
ElementCoeffs_K:=Vector(nops(DerK_Basis),i->_a[i]):
GeneralDer_MElement:=add(_b[i]*DerM_Basis[i],i=1..DerM_dim):
ElementCoeffs_M:=Vector(nops(DerM_Basis),i->_b[i]):

# The object Q is a list where ss[1] comes from der(K), ss[2] is in
# der(M), and ss[3] comes from sa3(KxM).

# We will first find the linear combination of basis elements of the
# algebra sa3(KxM).
R,C,Val:=ArrayTools:-SearchArray(Q[3]):
# Initialize storage arrays and counters.
vv:=Array(1..0):
IND:=Array(1..0):
pp:=Array(1..0):
count:=0:

# Find the linear combination of basis elements of the algebra Der(K).
if not(ArrayTools:-IsEqual(GeneralDer_KElement-Q[1],DerK[1,1])) then
  r,c,v:=ArrayTools:-SearchArray(
               subs(solve(convert(GeneralDer_KElement-Q[1],list)),
               ElementCoeffs_K)):
  if ArrayTools:-NumElems(r)>0 then
    count:=count+1:
    obj:=Array(1..op(1,r),{
               seq((k)=[
                   [alpha,beta,LieBasisIndex[r[k],0,0]],v[k]
                       ],k=1..op(1,r))
                         }):
    pp:=ArrayTools:-Concatenate(2,pp,obj):
  fi:
fi:

# Find the linear combination of basis elements of the algebra Der(M).
```

```
if not(ArrayTools:-IsEqual(GeneralDer_MElement-Q[2],DerM[1,1])) then
  r,c,v:=ArrayTools:-SearchArray(
              subs(solve(convert(GeneralDer_MElement-Q[2],list)),
              ElementCoeffs_M)):
  if ArrayTools:-NumElems(r)>0 then
    count:=count+1:
    obj:=Array(1..op(1,r),{
                seq((k)=[
                    [alpha,beta,LieBasisIndex[0,r[k],0]],v[k]
                        ],k=1..op(1,r))
                          }):
    pp:=ArrayTools:-Concatenate(2,pp,obj):
  fi:
fi:

for nn from 1 to op(1,R) do
  if R[nn]<C[nn] then
    r,c,v:=ArrayTools:-SearchArray(Val[nn]):
    ind:=map(r->saBasisIndex[R[nn],C[nn],r],r):
    vv:=ArrayTools:-Concatenate(2,vv,v):
    IND:=ArrayTools:-Concatenate(2,IND,ind):
  elif (R[nn]=C[nn] and R[nn]<3) then
    r,c,v:=ArrayTools:-SearchArray(Val[nn]):
    ind:=map(r->saBasisIndex[R[nn],C[nn],r],r):
    if R[nn]=1 then
      vv:=ArrayTools:-Concatenate(2,vv,v):
      IND:=ArrayTools:-Concatenate(2,IND,ind):
    else
      vv:=ArrayTools:-Concatenate(2,vv,v):
      IND:=ArrayTools:-Concatenate(2,IND,ind):
    fi:
  fi:
od:
nzIndx:=ArrayTools:-SearchArray(IND):
if op(1,ArrayTools:-SearchArray(IND))>0 then
  pp:=ArrayTools:-Concatenate(2,pp,
                    Array(1..op(1,nzIndx),{
                    seq( k=[
```

```
                        [alpha,beta,LieBasisIndex[0,0,IND[nzIndx[k]]]],
                          vv[nzIndx[k]]],k=1..op(1,nzIndx) )
                                        })):
  fi:
  IND:=ArrayTools:-SearchArray(pp):
  return convert(map(n->pp[n],IND),list);
end proc:
#--------------------End "FindLinearCombination"--------------------#

#------------------------Define "ModuleApply"------------------------#
ModuleApply:=proc(Name1, Name2, newalg)
   global DerK, DerM, dimK, dimM, dimKM, BasisKM, zT, BasisIndex,
          ConjugateSign, KTable, MTable, saBasisIndex, IsZero,
          DerK_dim, DerK_Basis, DerM_dim, DerM_Basis, LieBasisIndex;
   local  conj, prod, `&*`, `&.`, sa3_Basis, sa3, dN, AlgebraBasis,
          inc, ii, jj, sgn, delta, TrFree, herm, Bracket,
          TensorDerivation, kk, MultTable, Dimension, ni, nj, bk, PP,
          obj, Nbk;



  # Determine the appropriate Cayley algebra to load for the first
  # input algebra. This algebra will be denoted by "K".
  if Name1[1]="O" or Name1[1]="o" then
    if StringTools:-FirstFromLeft("S",Name1)>0 or
         StringTools:-FirstFromLeft("s",Name1)>0 then
      DGsetup(AlgebraLibraryData("Octonions",_tempalg1,type="Split")):
    else
      DGsetup(AlgebraLibraryData("Octonions",_tempalg1)):
    fi:
elif Name1[1]="Q" or Name1[1]="q" or Name1[1]="H" or Name1[1]="h" then
    if StringTools:-FirstFromLeft("S",Name1)>0 or
          StringTools:-FirstFromLeft("s",Name1)>0 then
      DGsetup(AlgebraLibraryData("Quaternions",_tempalg1,type="Split")):
    else
      DGsetup(AlgebraLibraryData("Quaternions",_tempalg1)):
    fi:
elif Name1[1]="C" or Name1[1]="c" then
```

```
  if StringTools:-FirstFromLeft("S",Name1)>0 or
        StringTools:-FirstFromLeft("s",Name1)>0 then
    DGsetup(AlgebraLibraryData("Complex",_tempalg1,type="Split")):
  else
    DGsetup(AlgebraLibraryData("Complex",_tempalg1)):
  fi:
else
  DGsetup(AlgebraLibraryData("Real",_tempalg1)):
fi:
# Determine the dimension of the algebra.
dimK:=DifferentialGeometry:-Tools:-DGinfo(_tempalg1,
                                    "FrameBaseDimension");
# Determine the structure constants for the algebra.
KTable:=StructureTable(_tempalg1):
# Create the lookup table for the basis elements of the Derivations
# algebra of K.
DerK:=DerivationsTable(KTable,_tempalg1,type=1):
# Remove the temporary frame. All further computations regarding K
# will be done without use of the frame.
RemoveFrame(_tempalg1);

# Determine the appropriate Cayley algebra to load for the second
# input algebra. This algebra will be denoted by "M".
if Name2[1]="O" or Name2[1]="o" then
  if StringTools:-FirstFromLeft("S",Name2)>0 or
        StringTools:-FirstFromLeft("s",Name2)>0 then
    DGsetup(AlgebraLibraryData("Octonions",_tempalg2,type="Split")):
  else
    DGsetup(AlgebraLibraryData("Octonions",_tempalg2)):
  fi:
elif Name2[1]="Q" or Name2[1]="q" or Name2[1]="H" or Name2[1]="h" then
  if StringTools:-FirstFromLeft("S",Name2)>0 or
        StringTools:-FirstFromLeft("s",Name2)>0 then
    DGsetup(AlgebraLibraryData("Quaternions",_tempalg2,type="Split")):
  else
    DGsetup(AlgebraLibraryData("Quaternions",_tempalg2)):
  fi:
elif Name2[1]="C" or Name2[1]="c" then
```

```
  if StringTools:-FirstFromLeft("S",Name2)>0 or
        StringTools:-FirstFromLeft("s",Name2)>0 then
    DGsetup(AlgebraLibraryData("Complex",_tempalg2,type="Split")):
  else
    DGsetup(AlgebraLibraryData("Complex",_tempalg2)):
  fi:
else
  DGsetup(AlgebraLibraryData("Real",_tempalg2)):
fi:
# Determine the dimension of the algebra.
dimM:=DifferentialGeometry:-Tools:-DGinfo(_tempalg2,
                                          "FrameBaseDimension");
# Determine the structure constants for the algebra.
MTable:=StructureTable(_tempalg2):
# Create the lookup table for the basis elements of the Derivations
# algebra of M.
DerM:=DerivationsTable(MTable,_tempalg2,type=2):
# Remove the temporary frame. All further computations regarding M
# will be done without use of the frame.
RemoveFrame(_tempalg2);

# The dimension of the tensor algebra is dim(K)*dim(M).
dimKM:=dimK*dimM:
# Initialize an array to store the basis elements of the tensor algebra.
BasisKM:=Array(1..dimKM):
# Initialize a lookup table to identify/retrieve basis elements based
# on their index.
BasisIndex:=table([]):
# Create a lookup table to compute the sign of the conjugate of an object
# in the tensor algebra. Creating such a table costs memory, but the
# computations only need to be done once.
ConjugateSign:=table([]):
inc:=0:
for ii from 1 to dimK do
  for jj from 1 to dimM do
    inc:=inc+1:
    # An element in the tensor algebra is given by an array of zeros,
    # and any nonzero entries in the array are the coefficients in
```

```
    # front of that object.
    BasisKM[inc]:=Array(1..dimKM,{(inc)=1}):
    # Due to the nature in which the basis elements of KxM are stored,
    # we need to know which objects in the lower algebras are used to
    # create the object in the tensor algebra. Therefore, we have a look
    # up table that tells us [i,j]->n  and that n->[i,j]. Therefore, we
    # can insert two indices into the table to return the index of the
    # corresponding object in the tensor algebra, or we can insert one
    # index into the table (identifying the tensor object) and it will
    # return two indices, each one identifying the basis elements in
    # K and M respectively that were used.
    BasisIndex[ii,jj]:=inc:
    BasisIndex[inc]:=[ii,jj]:
    # Note that conjugation in the tensor algebra is simply the tensor
    # of the conjugate of each basis element.
    ConjugateSign[inc]:=piecewise( (ii=1 and jj=1) or
                                   (ii>1 and jj>1),1,-1
                                 ):
  od:
od:

## Define procedures and shortcuts. ##

# Determine the sign of a coefficient.
sgn:=x->piecewise(x>0,1,x<0,-1,0):
# Define the procedure for multiplying two basis elements in the
# tensor product algebra KxM.
prod:=(u,v)->sgn(KTable[u[1],v[1]])*
             sgn(MTable[u[2],v[2]])*
             BasisIndex[abs(KTable[u[1],v[1]]),abs(MTable[u[2],v[2]])]:

# Define the procedure for computing the conjugate of an object
# (both in the algebras and the tensor algebra).
conj:=U->Array(1..dimKM,i->ConjugateSign[i]*U[i]):

# Define the Kronecker delta function (used for computing the inner
# product of basis elements).
delta:=(i,j)->piecewise(i=j,1,0):
```

```
# Define the procedure for computing the product of two elements in
# the tensor algebra.
`&*`:=proc(X,Y)
  local nx, ny, xi, yi, Z, nn;
  global dimKM;
  # Zero elements in the tensor algebra will not be stored as arrays
  # (to save space). This is easy to check.
  if not(evalb(X::Array)) then
    return 0;
  fi:
  if not(evalb(Y::Array)) then
    return 0;
  fi:
  # Identify the nonzero entries in X and Y.
  nx:=ArrayTools:-SearchArray(X):
  ny:=ArrayTools:-SearchArray(Y):
  # initialize a zero object
  Z:=Array(1..dimKM):
  for xi in nx do
    for yi in ny do
      # Compute the product of the elements in the tensor algebra.
      nn:=prod(BasisIndex[xi],BasisIndex[yi]):
      Z(abs(nn)):=Z(abs(nn))+sgn(nn)*X[xi]*Y[yi]:
    od:
  od:
  return Z:
end proc:

# Define multiplication of square matrices over the tensor algebra.
# Because we are only working with 3x3, we can save time by
# eliminating checks on the proper dimensions of the matrices.
`&.`:=proc(U,V)
  local Z, ii, jj:
  # Initialize a zero matrix. Any zero entries after the computation
  # will remain as zero, while nonzero entries will be arrays.
  Z:=Matrix(3,3):
  for ii from 1 to 3 do
```

```
    for jj from 1 to 3 do
      Z[ii,jj]:=add( U[ii,k]&*V[k,jj] ,k=1..3):
    od:
  od:
  return Z:
end proc:

# Define the procedure for computing the trace-free part of a matrix.
TrFree:=proc(U)
  local Tr;
  Tr:=add(U[k,k],k=1..3)/3:
  return Matrix(3,3,(i,j)->piecewise(i=j,U[i,j]-Tr,U[i,j])):
end proc:

# Define the procedure for creating a basis of the trace-free
# skew-hermition NxN matrices over KxM.
sa3_Basis:=proc()
  description "Create a basis for the 3x3 trace-free skew Hermition
               matrices over the algebra KxM.";
  global BasisKM, zT;
  local HermMatrix, TrFreeMatrix, Test, RealInd, ImInd, num, HM,
        inc, ii, jj, kk, numR, numI, saBasisIndex;
  # Define a procedure for creating a skew hermition matrix with
  # a tensor object 'x' in the [m,n] and [n,m] entries.
  HermMatrix:=(m,n,x)->Matrix(3,3,{(m,n)=x,(n,m)=conj(x)},fill=0):
  # Define a procedure for creating a trace free matrix with tensor
  # objects in 2 of the 3 diagonal entries.
  TrFreeMatrix:=(m,n,x)->Matrix(3,3,{(m,m)=x,(n,n)=-x},fill=0):
  # Find the objects in the basis of the tensor algebra that are
  # equal to their conjugate. This will be used to identity the
  # 'Real' and the 'Imaginary' elements in the tensor algebra.
  Test:=map(x->piecewise(ArrayTools:-IsEqual(x,conj(x)),1,0),BasisKM):
  RealInd:=ArrayTools:-SearchArray(Test):
  numR:=ArrayTools:-NumElems(RealInd):
  ImInd:=ArrayTools:-SearchArray(1-Test):
  numI:=ArrayTools:-NumElems(ImInd):
  # Compute the number of basis objects for the algebra sa3(KxM).
  num:=3*dimKM+2*numI:
```

```
# Initialize an array to store the matrix tensor objects.
HM:=Array(1..num):
inc:=0:
# Initialize an index table to identify/retrieve basis elements of
# sa3(KxM).
saBasisIndex:=table([]):
# Compute the matrices that have zero entries on the main diagonal.
for ii from 1 to 2 do
  for jj from (ii+1) to 3 do
    # Compute the matrices that have real entries.
    for kk from 1 to numR do
      inc:=inc+1:
      HM[inc] := Matrix(3,3,{(ii,jj)=BasisKM[RealInd[kk]],
                             (jj,ii)=-BasisKM[RealInd[kk]]},
                            fill=0):
      saBasisIndex[ii,jj,RealInd[kk]]:=inc:
      saBasisIndex[inc]:=[ii,jj,RealInd[kk]]:
    od:
    # Compute the matrices that have imaginary entries.
    for kk from 1 to numI do
      inc:=inc+1:
      HM[inc] := Matrix(3,3,{(ii,jj)=BasisKM[ImInd[kk]],
                             (jj,ii)=BasisKM[ImInd[kk]]},
                            fill=0):
      saBasisIndex[ii,jj,ImInd[kk]]:=inc:
      saBasisIndex[inc]:=[ii,jj,ImInd[kk]]:
    od:
  od:
od:
# These will be the trace free matrices with non zero elements
# on the main diagonal.
for ii from 1 to 2 do
  for kk from 1 to numI do
    inc:=inc+1:
    HM[inc] := Matrix(3,3,{(ii,ii)=BasisKM[ImInd[kk]],
                           (3,3)=-BasisKM[ImInd[kk]]},
                          fill=0):
    saBasisIndex[ii,ii,ImInd[kk]]:=inc:
```

```
      saBasisIndex[inc]:=[ii,ii,ImInd[kk]]:
    od:
  od:
  return HM, eval(saBasisIndex), inc:
end proc:


TensorDerivation:=proc(X,Y,NonZeroIndex)
  local indx, indy, DD, xi, yi, ii, jj, N;
  # Create the zero element in the derivation algebra.
  DD:=[DerK[1,1],DerM[1,1]]:
  for N in NonZeroIndex do
    # Find the nonzero coefficients of the entry in X.
    indx:=ArrayTools:-SearchArray(X(N)):
    # Find the nonzero coefficients of the entry in Y.
    indy:=ArrayTools:-SearchArray(Y(N)):



    for xi in indx do
      for yi in indy do
        # convert the running index to the tuple index to identify
        # the basis elements from the underlying algebras K and M.
        DD:=DD+[
               X(N)[xi]*Y(N)[yi]*
                 delta(BasisIndex[xi][2],BasisIndex[yi][2])*
                 DerK[BasisIndex[xi][1],BasisIndex[yi][1]],
               X(N)[xi]*Y(N)[yi]*
                 delta(BasisIndex[xi][1],BasisIndex[yi][1])*
                 DerM[BasisIndex[xi][2],BasisIndex[yi][2]]
               ]:
      od:
    od:
  od:
  return eval(DD);
end proc:

Bracket:=proc(UU,VV)
```

```
  description "Compute the Lie bracket of elements from
               der(K)+der(M)+sa3(KxM).";
  local NonZeroIndex, DD, W1, W2, W3, Bxy, Tr;
  # Find the indices for which BOTH matrices are nonzero.
  NonZeroIndex:=ArrayTools:-SearchArray(
                 Matrix(3,3,(i,j)->piecewise(UU[3][i,j]<>0 and
                                             VV[3][i,j]<>0,1,0)
                                      )):

  # If there are nonzero entries then compute the Tensor Derivation
  # for those entries. Otherwise, the derivation is zero.
  if op(NonZeroIndex)[1]<>0 then
    DD:=TensorDerivation(UU[3],VV[3],NonZeroIndex):
  else
    DD:=[DerK[1,1],DerM[1,1]]:
  fi:
  # Compute the commutator of the derivations + one third of the
  # Tensor derivation for each derivation algebra.
  W1:=UU[1].VV[1]-VV[1].UU[1]+DD[1]/3:
  W2:=UU[2].VV[2]-VV[2].UU[2]+DD[2]/3:
  # Compute the commutator for the matrices in sa3(KxM), compute the
  # trace to find the trace-free part of the commutator. Then combine
  # this result with the Derivation action on the 3x3 matrices.
  Bxy:=UU[3]&.VV[3]-VV[3]&.UU[3]:
  Tr:=add(Bxy[i,i],i=1..3)/3:
  W3:=Bxy-Matrix(3,3,{(1,1)=Tr,(2,2)=Tr,(3,3)=Tr})
      +DerivationAction(
             UU[1],UU[2],VV[3]-DerivationAction(VV[1],VV[2],UU[3])
                       ):
  return [W1,W2,W3]:
end proc:

# This nasty thing is used to identify if an object in
# der(K)+der(M)+sa3(KxM) is a zero object (zero in all three
# components). Determine if each matrix is is a zero matrix. Then
# count how many return 'true'. If there are 3, then return 'true'.
IsZero:=S->evalb(nops(
                 [ListTools:-SearchAll(true,
```

```
                    [ArrayTools:-IsEqual(S[1],DerK[1,1]),
                     ArrayTools:-IsEqual(S[2],DerM[1,1]),
                     ArrayTools:-IsEqual(S[3],Matrix(3,3))]
                                        )
                ])=3
              ):

# Create a list of the basis elements of der(K).
DerK_Basis:=DGbasis(map(x->x[1],[entries(DerK)])):
# Get the dimension of der(K).
DerK_dim:=nops(DerK_Basis):
# If der(K)=[], then reassign a zero matrix to DerK_Basis to be used
# in the calculations, but keep the dimension=0.
if DerK_Basis=[] then
  DerK_Basis:=[DerK[1,1]]:
  DerK_dim:=0:
fi:
# Create a list of the basis elements of der(M).
DerM_Basis:=DGbasis(map(x->x[1],[entries(DerM)])):
# Get the dimension of der(M).
DerM_dim:=nops(DerM_Basis):
# If der(M)=[], then reassign a zero matrix to DerM_Basis to be used
# in the calculations, but keep the dimension=0.
if DerM_Basis=[] then
  DerM_Basis:=[DerM[1,1]]:
  DerM_dim:=0:
fi:
# Compute the basis elements of sa3(KxM). Note also that the indexing
# table is returned and the dimension of sa3(KxM).
sa3,saBasisIndex,dN:=sa3_Basis():

# Create an indexing table to identify the basis elements of
# der(K)+der(M)+sa3(KxM)
LieBasisIndex:=table([]):
inc:=0:
for ii from 1 to DerK_dim do
  inc:=inc+1:
  LieBasisIndex[inc]:=[ii,0,0]:
```

```
  LieBasisIndex[ii,0,0]:=inc:
od:
for jj from 1 to DerM_dim do
  inc:=inc+1:
  LieBasisIndex[inc]:=[0,jj,0]:
  LieBasisIndex[0,jj,0]:=inc:
od:
for kk from 1 to ArrayTools:-NumElems(sa3) do
  inc:=inc+1:
  LieBasisIndex[inc]:=[0,0,kk]:
  LieBasisIndex[0,0,kk]:=inc:
od:

# Define a procedure to quickly construct a basis element of the
# algebra der(K)+der(M)+sa3(KxM).
AlgebraBasis:=(i,j,k)->piecewise(i>0 and j=0 and k=0,
                        [DerK_Basis[i],DerM[-1,-1],Matrix(3,3)],
                                 i=0 and j>0 and k=0,
                        [DerK[-1,-1],DerM_Basis[j],Matrix(3,3)],
                                 i=0 and j=0 and k>0,
                        [DerK[-1,-1],DerM[-1,-1],sa3[k]],
                        [DerK[-1,-1],DerM[-1,-1],Matrix(3,3)]
                                 ):
# Initialize an array to store to bracket rules. Because the
# number of elements computed for each bracket (and thus for
# ALL brackets) is unknown beforehand, an empty array is created
# and additional results will be concatednated with each
# calculation.
MultTable:=Array(1..0):
# Compute the dimension of the Lie algebra.
Dimension:=DerK_dim+DerM_dim+ArrayTools:-NumElems(sa3):
inc:=0:
for ni from 1 to Dimension-1 do
  for nj from (ni+1) to Dimension do
    inc:=inc+1:


    # Compute the Lie bracket of basis elements, [ei,ej].
```

```
      obj:=Bracket(AlgebraBasis(op(LieBasisIndex[ni])),
                   AlgebraBasis(op(LieBasisIndex[nj]))
                  ):
      # Find the linear combination of basis elements. Essentially,
      # we are finding the structure constants for
      # der(K)+der(M)+sa3(KxM).
      bk:=FindLinearCombination( obj, ni, nj):
      if evalb(bk<>NULL) then
        Nbk:=nops(bk);
        # Store the coefficients for the product as seperate list
        # elements. These will be used to reconstruct the algebra in
        # the DG framework
        PP:=Array(1..2*Nbk,
                  {seq((k)=bk[k],k=1..Nbk),
                   seq((Nbk+k)=[[bk[k,1,2],bk[k,1,1],bk[k,1,3]],-bk[k,2]],
                               k=1..Nbk)
                  }):
        MultTable:=ArrayTools:-Concatenate(2,MultTable,PP):
      fi:
    od:
  od:
  # Store the structure constants in the framework to build the algebra.
  return _DG([["LieAlgebra",newalg,[Dimension,table( [] )]],
              convert(MultTable,list)]):
end proc:
#--------------------------End "ModuleApply"-------------------------#
end module:
```

C Maple Tutorials

C.1 Building a Quaternion Algebra

How To Create A Quaternion Algebra

Synopsis

• We show how to create a Quaternion algebra in Maple using the AlgebraLibraryData command.

Commands Illustrated

• LieAlgebras, DGsetup, AlgebraLibraryData, MultiplicationTable

Examples

Load in the required packages.

```
> with(DifferentialGeometry): with(LieAlgebras):
```

Example 1.

The output of the command AlgebraLibraryData is a list of nonzero products, where by default *e1* is the multiplicative identity, and *e2*, *e3*, *e4* correspond to the pure imaginary elements.

```
> AD1 := AlgebraLibraryData("Quaternions",H);
```

$$AD1 := [e1^2 = e1, e1.e2 = e2, e1.e3 = e3, e1.e4 = e4, e2.e1 = e2, e2^2 = -e1, e2.e3 = e4, e2.e4 = -e3, e3.e1 = e3, e3.e2 = -e4, e3^2 = -e1, e3.e4 = e2, e4.e1 = e4, e4.e2 = e3, e4.e3 = -e2, e4^2 = -e1] \quad (1)$$

We use the command DGsetup to store these structure equations in memory.

```
> DGsetup(AD1);
```

algebra name: H (2)

At this point one can now invoke many of the commands in the LieAlgebras package. For example, we can display the multiplication table for the basis elements:

```
H > MultiplicationTable(H);
```

$$\begin{bmatrix} & | & e1 & e2 & e3 & e4 \\ & ---- & ---- & ---- & ---- & ---- \\ e1 & | & e1 & e2 & e3 & e4 \\ e2 & | & e2 & -e1 & e4 & -e3 \\ e3 & | & e3 & -e4 & -e1 & e2 \\ e4 & | & e4 & e3 & -e2 & -e1 \end{bmatrix} \quad (3)$$

We know that the quaternions are an associative algebra, so we can consider a matrix representation of an element in *H*. This can be accomplished by using the Adjoint representation.

```
H > ad:=Adjoint(H);
```

$$ad := \left[\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}\right] \quad \textbf{(4)}$$

Note that multiplication in the algebra is preserved by the adjoin representation. For example,

```
H > ad[2].ad[3];
```

$$\begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \textbf{(5)}$$

Example 2.

We can also use the command AlgebraLibraryData to create the algebra of the split quaternions, that is, quaternions with the signature (1,3)

```
> AD2 := AlgebraLibraryData("Quaternions",Hs,type="Split");
```

$$AD2 := [e1^2 = e1, e1.e2 = e2, e1.e3 = e3, e1.e4 = e4, e2.e1 = e2, e2^2 = -e1, e2.e3 = e4, e2.e4 = -e3, e3.e1 = e3, e3.e2 = -e4, e3^2 = e1, e3.e4 = -e2, e4.e1 = e4, e4.e2 = e3, e4.e3 = e2, e4^2 = e1] \quad \textbf{(6)}$$

```
> DGsetup(AD2);
```

algebra name: Hs **(7)**

```
Hs > MultiplicationTable(Hs);
```

		$e1$	$e2$	$e3$	$e4$
	----	----	----	----	----
$e1$	|	$e1$	$e2$	$e3$	$e4$
$e2$	|	$e2$	$-e1$	$e4$	$-e3$
$e3$	|	$e3$	$-e4$	$e1$	$-e2$
$e4$	|	$e4$	$e3$	$e2$	$e1$

(8)

C.2 Building an Octonion Algebra

How To Create An Octonion Algebra

Synopsis

• We show how to create a Octonion algebra in Maple using the AlgebraLibraryData command.

Commands Illustrated

• LieAlgebras, DGsetup, AlgebraLibraryData, MultiplicationTable

Examples

Load in the required packages.

```
> with(DifferentialGeometry): with(LieAlgebras): with(Tools):
```

Example 1.

The output of the command AlgebraLibraryData is a list of nonzero products, where by default *e1* is the multiplicative identity, and the remaining objects correspond to the pure imaginary basis elements.

```
> AD1 := AlgebraLibraryData("Octonions",O);
```

$$AD1 := [e1^2 = e1, e1.e2 = e2, e1.e3 = e3, e1.e4 = e4, e1.e5 = e5, e1.e6 = e6, e1.e7 = e7, e1.e8 = e8, e2.e1 = e2, e2^2 = -e1, e2.e3 = e4, e2.e4 = -e3, e2.e5 = e6, e2.e6 = -e5, e2.e7 = -e8, e2.e8 = e7, e3.e1 = e3, e3.e2 = -e4, e3^2 = -e1, e3.e4 = e2, e3.e5 = e7, e3.e6 = e8, e3.e7 = -e5, e3.e8 = -e6, e4.e1 = e4, e4.e2 = e3, e4.e3 = -e2, e4^2 = -e1, e4.e5 = e8, e4.e6 = -e7, e4.e7 = e6, e4.e8 = -e5, e5.e1 = e5, e5.e2 = -e6, e5.e3 = -e7, e5.e4 = -e8, e5^2 = -e1, e5.e6 = e2, e5.e7 = e3, e5.e8 = e4, e6.e1 = e6, e6.e2 = e5, e6.e3 = -e8, e6.e4 = e7, e6.e5 = -e2, e6^2 = -e1, e6.e7 = -e4, e6.e8 = e3, e7.e1 = e7, e7.e2 = e8, e7.e3 = e5, e7.e4 = -e6, e7.e5 = -e3, e7.e6 = e4, e7^2 = -e1, e7.e8 = -e2, e8.e1 = e8, e8.e2 = -e7, e8.e3 = e6, e8.e4 = e5, e8.e5 = -e4, e8.e6 = -e3, e8.e7 = e2, e8^2 = -e1] \quad (1)$$

We use the command DGsetup to store these structure equations in memory.

```
> DGsetup(AD1);
```

$$\text{algebra name: } O \quad (2)$$

At this point one can now invoke many of the commands in the LieAlgebras package. For example, we can display the multiplication table for the basis elements:

```
H > MultiplicationTable(O);
```

		e1	e2	e3	e4	e5	e6	e7	e8
	----	----	----	----	----	----	----	----	----
e1	|	e1	e2	e3	e4	e5	e6	e7	e8
e2	|	e2	-e1	e4	-e3	e6	-e5	-e8	e7
e3	|	e3	-e4	-e1	e2	e7	e8	-e5	-e6
e4	|	e4	e3	-e2	-e1	e8	-e7	e6	-e5
e5	|	e5	-e6	-e7	-e8	-e1	e2	e3	e4
e6	|	e6	e5	-e8	e7	-e2	-e1	-e4	e3
e7	|	e7	e8	e5	-e6	-e3	e4	-e1	-e2
e8	|	e8	-e7	e6	e5	-e4	-e3	e2	-e1

(3)

Example 2.

We can also use the command AlgebraLibraryData to create the algebra of the split quaternions, that is, quaternions with the signature (1,3)

```
> AD2 := AlgebraLibraryData("Octonions",Os,type="Split");
```

$$AD2 := [e1^2 = e1, e1.e2 = e2, e1.e3 = e3, e1.e4 = e4, e1.e5 = e5, e1.e6 = e6, e1.e7 = e7, e1.e8 = e8, e2.e1 = e2, e2^2 = -e1, e2.e3 = e4, e2.e4 = -e3, e2.e5 = -e6, e2.e6 = e5, e2.e7 = -e8, e2.e8 = e7, e3.e1 = e3, e3.e2 = -e4, e3^2 = -e1, e3.e4 = e2, e3.e5 = -e7, e3.e6 = e8, e3.e7 = e5, e3.e8 = -e6, e4.e1 = e4, e4.e2 = e3, e4.e3 = -e2, e4^2 = -e1, e4.e5 = -e8, e4.e6 = -e7, e4.e7 = e6, e4.e8 = e5, e5.e1 = e5, e5.e2 = e6, e5.e3 = e7, e5.e4 = e8, e5^2 = e1, e5.e6 = e2, e5.e7 = e3, e5.e8 = e4, e6.e1 = e6, e6.e2 = -e5, e6.e3 = -e8, e6.e4 = e7, e6.e5 = -e2, e6^2 = e1, e6.e7 = e4, e6.e8 = -e3, e7.e1 = e7, e7.e2 = e8, e7.e3 = -e5, e7.e4 = -e6, e7.e5 = -e3, e7.e6 = -e4, e7^2 = e1, e7.e8 = e2, e8.e1 = e8, e8.e2 = -e7, e8.e3 = e6, e8.e4 = -e5, e8.e5 = -e4, e8.e6 = e3, e8.e7 = -e2, e8^2 = e1]$$ **(4)**

```
> DGsetup(AD2);
```

algebra name: Os **(5)**

```
Os > MultiplicationTable(Os);
```

		e1	e2	e3	e4	e5	e6	e7	e8
	----	----	----	----	----	----	----	----	----
e1	|	e1	e2	e3	e4	e5	e6	e7	e8
e2	|	e2	-e1	e4	-e3	-e6	e5	-e8	e7
e3	|	e3	-e4	-e1	e2	-e7	e8	e5	-e6
e4	|	e4	e3	-e2	-e1	-e8	-e7	e6	e5
e5	|	e5	e6	e7	e8	e1	e2	e3	e4
e6	|	e6	-e5	-e8	e7	-e2	e1	e4	-e3
e7	|	e7	e8	-e5	-e6	-e3	-e4	e1	e2
e8	|	e8	-e7	e6	-e5	-e4	e3	-e2	e1

(6)

We know that the derivations algebra of the octonions is the exceptional semisimple Lie algebra g_2. We can use the command Derivations to compute the set of linear transformations.

```
Os > Der := Derivations(Os);
```

$$Der := \left[\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 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 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 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 & -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 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 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 & 1 & 0 \\ 0 & 1 & 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 \end{bmatrix}, \begin{bmatrix} 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 & 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 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 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 & 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 & -1 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 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 & -1 & 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 & 0 \end{bmatrix}, \begin{bmatrix} 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 & 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 & 0 \end{bmatrix}\right] \quad \textbf{(7)}$$

$$\left[\begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&-1&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&-1&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&-1&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&-1&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{bmatrix},$$

$$\begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&-1\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&-1&0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&1&0&0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&-1&0&0&0\\ 0&0&0&0&0&0&0&-1\\ 0&0&0&0&0&0&1&0 \end{bmatrix},$$

$$\left[\begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&-1&0&0&0\\ 0&0&0&0&0&-1&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&-1&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&-1&0&0&0 \end{bmatrix}\right]$$

With these matrices, we can use the command LieAlgebraData to create the algebra structure equations for the Lie algebra.

```
Os > LA := LieAlgebraData(Der,g2):
Os > DGsetup(LA);
```

Lie algebra: g2 **(8)**

We can check a few properties to see if we have indeed created g_2. First we use the command DGinfo to return the dimension of the Lie algebra.

```
g2 > DGinfo("FrameBaseDimension");
```

14 **(9)**

C.3 Building a Jordan Algebra

How To Create A Jordan Algebra

Synopsis

• We show how to create a Jordan algebra in Maple using the AlgebraLibraryData command.

Commands Illustrated

• LieAlgebras, DGsetup, AlgebraLibraryData, MultiplicationTable

Examples

Load in the required packages.

```
> with(DifferentialGeometry): with(LieAlgebras): with(Tools): with
  (Tensor):
  interface(rtablesize=20):
```

Example 1.

The output of the command AlgebraLibraryData is a list of nonzero products defining the structure equations for the basis elements of the algebra. One of the most commonly known Jordan algebras is that created by 3 × 3 skew hermitian matrices over the octonions. We specify the dimension of the matrices desired and the algebra over which the matrices are constructed. The output has been supressed.

```
> JD := AlgebraLibraryData("Jordan(3,Octonions)",J3O):
```

We use the command DGsetup to store these structure equations in memory.

```
> DGsetup(JD);
```

algebra name: J3O **(1)**

At this point one can now invoke many of the commands in the LieAlgebras package. For example, retrieve the basis elements:

```
J3O > Basis:=Tools:-DGinfo(J3O,"FrameBaseVectors"):
```

We can now start combining basis elements using evalDG command to create more general elements of the algebra.

```
J3O > x:=evalDG(3*Basis[3]-2*Basis[20]);
```

$$x := 3\,e3 - 2\,e20 \quad \textbf{(2)}$$

```
J3O > y:=evalDG(Basis[15]+2*Basis[27]);
```

$$y := e15 + 2\,e27 \quad \textbf{(3)}$$

We also use the evalDG command to compute the product of two elements in the algebra.

```
J3O > z:=evalDG(x.y);
```

(4)

$$z := -2\,e4 + \frac{3}{2}\,e15 + 3\,e27 \tag{4}$$

```
J3O > Der := Derivations(J3O):
J3O > LA := LieAlgebraData(Der,f4):
J3O > DGsetup(LA);
```

Lie algebra: f4 (5)

```
f4 > B:=KillingForm(f4):
C4 > QF:=QuadraticFormSignature(B);
```

$$QF := [[\,], [e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12, e13, e14, e15, e16, e17, e18, e19, e20, e21, e22, e23, e24, e25, e25 - 2\,e50, e25 - 3\,e45 + e50, e25 + 2\,e32 + e45 + e50, e26, e26 - 2\,e51, e26 + 3\,e40 + e51, e26 + 2\,e33 - e40 + e51, e27, e27 - 2\,e52, e27 - 3\,e39 + e52, e27 + 2\,e34 + e39 + e52, e28, e28 + 2\,e48, e28 + 3\,e42 - e48, e28 + 2\,e35 - e42 - e48, e29, e29 + 2\,e49, e29 - 3\,e41 - e49, e29 + 2\,e36 + e41 - e49, e30, e30 - 2\,e46, e30 - 3\,e44 + e46, e30 + 2\,e37 + e44 + e46, e31, e31 - 2\,e47, e31 + 3\,e43 + e47, e31 + 2\,e38 - e43 + e47], [\,]] \tag{6}$$

```
C4 > map(nops,QF);
```

$$[0, 52, 0] \tag{7}$$

```
f4 > CSA:=CartanSubalgebra(f4);
```

$$CSA := [e1, e32, e33 - 2\,e40, e37 + 2\,e46] \tag{8}$$

```
f4 > nops(CSA);
```

$$4 \tag{9}$$

```
f4 > Query(f4,"Semisimple");
```

true (10)

Example 2.

We can also use the command AlgebraLibraryData to create a Clifford algebra over a 3-dimensional vector space with "unusual" quadratic forms. For example,

```
> Q:=Matrix([[4,0,-2],[0,-1,-1/2],[-2,-1/2,1]]);
```

$$Q := \begin{bmatrix} 4 & 0 & -2 \\ 0 & -1 & -\frac{1}{2} \\ -2 & -\frac{1}{2} & 1 \end{bmatrix} \tag{11}$$

```
> CD2 := AlgebraLibraryData("Clifford(3)",C3,quadraticform=Q);
```

$$CD2 := [e1^2 = e1, e1.e2 = e2, e1.e3 = e3, e1.e4 = e4, e1.e5 = e5, e1.e6 = e6, e1.e7 = e7, e1.e8 = e8, e2.e1 = e2, e2^2 = -4\,e1, e2.e3 = e5, e2.e4 = e6, e2.e5 = -4\,e3, e2.e6 = -4\,e4, e2.e7 = e8, e2.e8 = -4\,e7, e3.e1 = e3, e3.e2 = -e5, e3^2 = e1, e3.e4 = e7, e3.e5 = -e2, e3.e6 = -e8, e3.e7 = e4, e3.e8 = -e6, e4.e1 = e4, e4.e2 = -e6 + 4\,e1, e4.e3 = -e7 + e1, e4^2 = -e1, e4.e5 = e8 + 4\,e3 - e2, e4.e6 = 4\,e4 + e2, e4.e7 = e4 + e3, e4.e8 = 4\,e7 - e6 - e5, e5.e1 \tag{12}$$

$$= e5,\ e5.e2 = 4\ e3,\ e5.e3 = e2,\ e5.e4 = e8,\ e5^2 = 4\ e1,\ e5.e6 = 4\ e7,\ e5.e7 = e6,\ e5.e8 = 4\ e4,$$
$$e6.e1 = e6,\ e6.e2 = 4\ e4 + 4\ e2,\ e6.e3 = -e8 + e2,\ e6.e4 = -e2,\ e6.e5 = -4\ e7 + 4\ e5$$
$$+ 4\ e1,\ e6^2 = 4\ e6 - 4\ e1,\ e6.e7 = e6 + e5,\ e6.e8 = 4\ e8 + 4\ e4 + 4\ e3,\ e7.e1 = e7,\ e7.e2$$
$$= e8 + 4\ e3,\ e7.e3 = -e4 + e3,\ e7.e4 = -e3,\ e7.e5 = -e6 + e5 + 4\ e1,\ e7.e6 = 4\ e7 - e5,$$
$$e7^2 = e7 + e1,\ e7.e8 = e8 + 4\ e4 + e2,\ e8.e1 = e8,\ e8.e2 = -4\ e7 + 4\ e5,\ e8.e3 = -e6$$
$$+ e5,\ e8.e4 = -e5,\ e8.e5 = 4\ e4 - 4\ e3 + 4\ e2,\ e8.e6 = 4\ e8 + 4\ e3,\ e8.e7 = e8 + e2,\ e8^2 =$$
$$-4\ e7 + 4\ e6 - 4\ e1\ ]$$

We use the command DGsetup to store these structure equations in memory. We can also specify the names of the vectors and forms in the algebra in such a way as to indicate the manner in which the basis elements were constructed.

```
> DGsetup(CD2,[x||0,x||1,x||2,x||3,x||12,x||13,x||23,x||123],
                [d||0,d||1,d||2,d||3,d||12,d||13,d||23,d||123]);
```

algebra name: C3 **(13)**

Display the multiplication table. We see that the product of the basis elements in $C(Q)$ is not a trivial matter.

```
C3 > MultiplicationTable(C3);
```

$$[[, |, x0, x1, x2, x3, x12, x13, x23, x123\ ],$$ **(14)**
$$[, ----, ----, ----, ----, ----, ----, ----, ----, ----\ ],$$
$$[x0, |, x0, x1, x2, x3, x12, x13, x23, x123\ ],$$
$$[x1, |, x1, -4\ x0, x12, x13, -4\ x2, -4\ x3, x123, -4\ x23\ ],$$
$$[x2, |, x2, -x12, x0, x23, -x1, -x123, x3, -x13\ ],$$
$$[x3, |, x3, 4\ x0 - x13, x0 - x23, -x0, -x1 + 4\ x2 + x123, x1 + 4\ x3, x2 + x3, -x12 - x13$$
$$+ 4\ x23\ ],$$
$$[x12, |, x12, 4\ x2, x1, x123, 4\ x0, 4\ x23, x13, 4\ x3\ ],$$
$$[x13, |, x13, 4\ x1 + 4\ x3, x1 - x123, -x1, 4\ x0 + 4\ x12 - 4\ x23, -4\ x0 + 4\ x13, x12$$
$$+ x13, 4\ x2 + 4\ x3 + 4\ x123\ ],$$
$$[x23, |, x23, 4\ x2 + x123, x2 - x3, -x2, 4\ x0 + x12 - x13, -x12 + 4\ x23, x0 + x23, x1$$
$$+ 4\ x3 + x123\ ],$$
$$[x123, |, x123, 4\ x12 - 4\ x23, x12 - x13, -x12, 4\ x1 - 4\ x2 + 4\ x3, 4\ x2 + 4\ x123, x1$$
$$+ x123, -4\ x0 + 4\ x13 - 4\ x23\ ]]$$

```
C3 > KB := KillingForm(C3);
```

$$KB := 16\ d1\ d3 - 16\ d1\ d123 - 32\ d1\ d1 - 16\ d123\ d2 + 16\ d23\ d12 + 4\ d2\ d3 + 8\ d0\ d0$$ **(15)**
$$+ 16\ d13\ d0 + 4\ d23\ d0 + 4\ d3\ d2 + 16\ d13\ d23 + 16\ d3\ d1 + 32\ d13\ d13 + 4\ d0\ d23$$

$$-16\,d123\,d1+16\,d12\,d13+16\,d23\,d13+16\,d123\,d123-8\,d3\,d3+8\,d2\,d2$$
$$+12\,d23\,d23+16\,d13\,d12+16\,d12\,d23+32\,d12\,d12+16\,d0\,d13-16\,d2\,d123$$

```
C3 > QF:=QuadraticFormSignature(KB);
```

$$QF := \left[\left[x0, x0-2\,x23, x1-2\,x123, x12+\frac{1}{2}\,x13-2\,x23\right], \left[x0-\frac{5}{8}\,x13+\frac{1}{2}\,x23, x1, x1\right.\right. \tag{16}$$
$$\left.\left.+3\,x3+x123, x1-2\,x2+x3-x123\right], [\,]\right]$$

```
C3 > map(nops,QF);
```

$$[4, 4, 0] \tag{17}$$

C.4 Building a Clifford Algebra

How To Create A Clifford Algebra

Synopsis

• We show how to create a Clifford algebra in Maple using the AlgebraLibraryData command.

Commands Illustrated

• LieAlgebras, DGsetup, AlgebraLibraryData, MultiplicationTable

Examples

Load in the required packages.

```
> with(DifferentialGeometry):
  with(LieAlgebras):
  with(Tools):
  with(Tensor):
  interface(rtablesize=20):
```

Example 1.

The output of the command AlgebraLibraryData is a list of nonzero products, where by default $e1$ is the multiplicative identity in the Clifford algebra. The remaining objects represent tensors of vectors from the underlying vector space. We must specify the dimension of the underlying vector space V. If a symmetric quadratic form Q is not given, then the identity matrix is used by default.

```
> CD := AlgebraLibraryData("Clifford(3)",C3);
```

$$CD := [e1^2 = e1, e1.e2 = e2, e1.e3 = e3, e1.e4 = e4, e1.e5 = e5, e1.e6 = e6, e1.e7 = e7, e1.e8 = e8, e2.e1 = e2, e2^2 = -e1, e2.e3 = e5, e2.e4 = e6, e2.e5 = -e3, e2.e6 = -e4, e2.e7 = e8, e2.e8 = -e7, e3.e1 = e3, e3.e2 = -e5, e3^2 = -e1, e3.e4 = e7, e3.e5 = e2, e3.e6 = -e8, e3.e7 = -e4, e3.e8 = e6, e4.e1 = e4, e4.e2 = -e6, e4.e3 = -e7, e4^2 = -e1, e4.e5 = e8, e4.e6 = e2, e4.e7 = e3, e4.e8 = -e5, e5.e1 = e5, e5.e2 = e3, e5.e3 = -e2, e5.e4 = e8, e5^2 = -e1, e5.e6 = e7, e5.e7 = -e6, e5.e8 = -e4, e6.e1 = e6, e6.e2 = e4, e6.e3 = -e8, e6.e4 = -e2, e6.e5 = -e7, e6^2 = -e1, e6.e7 = e5, e6.e8 = e3, e7.e1 = e7, e7.e2 = e8, e7.e3 = e4, e7.e4 = -e3, e7.e5 = e6, e7.e6 = -e5, e7^2 = -e1, e7.e8 = -e2, e8.e1 = e8, e8.e2 = -e7, e8.e3 = e6, e8.e4 = -e5, e8.e5 = -e4, e8.e6 = e3, e8.e7 = -e2, e8^2 = e1] \quad (1)$$

We use the command DGsetup to store these structure equations in memory.

```
> DGsetup(CD);
```

$$\text{algebra name: } C3 \quad (2)$$

At this point one can now invoke many of the commands in the LieAlgebras package. For example, we can display the multiplication table for the basis elements:

```
C3 > MultiplicationTable(C3,"AlgebraTable");
```

		e1	e2	e3	e4	e5	e6	e7	e8
	----	----	----	----	----	----	----	----	----
e1	|	e1	e2	e3	e4	e5	e6	e7	e8
e2	|	e2	-e1	e5	e6	-e3	-e4	e8	-e7
e3	|	e3	-e5	-e1	e7	e2	-e8	-e4	e6
e4	|	e4	-e6	-e7	-e1	e8	e2	e3	-e5
e5	|	e5	e3	-e2	e8	-e1	e7	-e6	-e4
e6	|	e6	e4	-e8	-e2	-e7	-e1	e5	e3
e7	|	e7	e8	e4	-e3	e6	-e5	-e1	-e2
e8	|	e8	-e7	e6	-e5	-e4	e3	-e2	e1

(3)

One of the important properties to determine is the signature of the algebra. To find this, we first compute the Killing form.

```
C3 > B:=KillingForm(C3);
```

$$B := 8\,\theta 1\,\theta 1 - 8\,\theta 2\,\theta 2 - 8\,\theta 3\,\theta 3 - 8\,\theta 4\,\theta 4 - 8\,\theta 5\,\theta 5 - 8\,\theta 6\,\theta 6 - 8\,\theta 7\,\theta 7 + 8\,\theta 8\,\theta 8 \quad \textbf{(4)}$$

Once we have the Killing form, we can compute the quadratic form signature.

```
C3 > QF:=QuadraticFormSignature(B);
```

$$QF := [[e1, e8], [e2, e3, e4, e5, e6, e7], [\,]] \quad \textbf{(5)}$$

```
C3 > map(nops,QF);
```

$$[2, 6, 0] \quad \textbf{(6)}$$

Example 2.

We can also use the command AlgebraLibraryData to create a Clifford algebra over a 3-dimensional vector space with "unusual" quadratic forms. For example,

```
> Q:=Matrix([[2,0,1],[0,-1,0],[1,0,2]]);
```

$$Q := \begin{bmatrix} 2 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 2 \end{bmatrix} \quad \textbf{(7)}$$

```
> CD2 := AlgebraLibraryData("Clifford(3)",C3,quadraticform=Q);
```

$$CD2 := [e1^2 = e1, e1.e2 = e2, e1.e3 = e3, e1.e4 = e4, e1.e5 = e5, e1.e6 = e6, e1.e7 = e7, e1.e8 = e8, e2.e1 = e2, e2^2 = -2\,e1, e2.e3 = e5, e2.e4 = e6, e2.e5 = -2\,e3, e2.e6 = -2\,e4, e2.e7 = e8, e2.e8 = -2\,e7, e3.e1 = e3, e3.e2 = -e5, e3^2 = e1, e3.e4 = e7, e3.e5 = -e2, e3.e6 = -e8, e3.e7 = e4, e3.e8 = -e6, e4.e1 = e4, e4.e2 = -e6 - 2\,e1, e4.e3 = -e7, e4^2 = -2\,e1, e4 \quad \textbf{(8)}$$

$$.e5 = e8 - 2\,e3,\ e4.e6 = -2\,e4 + 2\,e2,\ e4.e7 = 2\,e3,\ e4.e8 = -2\,e7 - 2\,e5,\ e5.e1 = e5,\ e5.e2 = 2\,e3,\ e5.e3 = e2,\ e5.e4 = e8,\ e5^2 = 2\,e1,\ e5.e6 = 2\,e7,\ e5.e7 = e6,\ e5.e8 = 2\,e4,\ e6.e1 = e6,\ e6.e2 = 2\,e4 - 2\,e2,\ e6.e3 = -e8,\ e6.e4 = -2\,e2,\ e6.e5 = -2\,e7 - 2\,e5,\ e6^2 = -2\,e6 - 4\,e1,\ e6.e7 = 2\,e5,\ e6.e8 = -2\,e8 + 4\,e3,\ e7.e1 = e7,\ e7.e2 = e8 - 2\,e3,\ e7.e3 = -e4,\ e7.e4 = -2\,e3,\ e7.e5 = -e6 - 2\,e1,\ e7.e6 = -2\,e7 - 2\,e5,\ e7^2 = 2\,e1,\ e7.e8 = -2\,e4 + 2\,e2,\ e8.e1 = e8,\ e8.e2 = -2\,e7 - 2\,e5,\ e8.e3 = -e6,\ e8.e4 = -2\,e5,\ e8.e5 = 2\,e4 - 2\,e2,\ e8.e6 = -2\,e8 + 4\,e3,\ e8.e7 = 2\,e2,\ e8^2 = -2\,e6 - 4\,e1\ ]$$

We use the command DGsetup to store these structure equations in memory. We can also specify the names of the vectors and forms in the algebra in such a way as to indicate the manner in which the basis elements were constructed.

```
> DGsetup(CD2,[x[0],x[1],x[2],x[3],x[12],x[13],x[23],x[123]],
                [d[0],d[1],d[2],d[3],d[12],d[13],d[23],d[123]]);
```

algebra name: C3 **(9)**

Display the multiplication table. We see that the product of the basis elements in $C(Q)$ is not a trivial matter.

```
C3 > MultiplicationTable(C3);
```

$$\left[\left[, |, x_0, x_1, x_2, x_3, x_{12}, x_{13}, x_{23}, x_{123}\right],\right.$$
$$\left[, ----, ----, ----, ----, ----, ----, ----, ----, ----\right],$$
$$\left[x_0, |, x_0, x_1, x_2, x_3, x_{12}, x_{13}, x_{23}, x_{123}\right],$$
$$\left[x_1, |, x_1, -2\,x_0, x_{12}, x_{13}, -2\,x_2, -2\,x_3, x_{123}, -2\,x_{23}\right],$$
$$\left[x_2, |, x_2, -x_{12}, x_0, x_{23}, -x_1, -x_{123}, x_3, -x_{13}\right],$$
$$\left[x_3, |, x_3, -2\,x_0 - x_{13}, -x_{23}, -2\,x_0, -2\,x_2 + x_{123}, 2\,x_1 - 2\,x_3, 2\,x_2, -2\,x_{12} - 2\,x_{23}\right],$$
$$\left[x_{12}, |, x_{12}, 2\,x_2, x_1, x_{123}, 2\,x_0, 2\,x_{23}, x_{13}, 2\,x_3\right],$$
$$\left[x_{13}, |, x_{13}, -2\,x_1 + 2\,x_3, -x_{123}, -2\,x_1, -2\,x_{12} - 2\,x_{23}, -4\,x_0 - 2\,x_{13}, 2\,x_{12}, 4\,x_2 - 2\,x_{123}\right],$$
$$\left[x_{23}, |, x_{23}, -2\,x_2 + x_{123}, -x_3, -2\,x_2, -2\,x_0 - x_{13}, -2\,x_{12} - 2\,x_{23}, 2\,x_0, 2\,x_1 - 2\,x_3\right],$$
$$\left.\left[x_{123}, |, x_{123}, -2\,x_{12} - 2\,x_{23}, -x_{13}, -2\,x_{12}, -2\,x_1 + 2\,x_3, 4\,x_2 - 2\,x_{123}, 2\,x_1, -4\,x_0 - 2\,x_{13}\right]\right] \quad \textbf{(10)}$$

```
C3 > KB := KillingForm(C3);
```

$$KB := 8\,d_0\,d_0 - 8\,d_0\,d_{13} - 16\,d_1\,d_1 - 8\,d_1\,d_3 + 8\,d_2\,d_2 + 8\,d_2\,d_{123} - 8\,d_3\,d_1 - 16\,d_3\,d_3 \quad \textbf{(11)}$$

$$+ 16\, d_{12}\, d_{12} - 8\, d_{12}\, d_{23} - 8\, d_{13}\, d_0 - 16\, d_{13}\, d_{13} - 8\, d_{23}\, d_{12} + 16\, d_{23}\, d_{23} + 8\, d_{123}\, d_2 - 16\, d_{123}\, d_{123}$$

```
C3 > QF:=QuadraticFormSignature(KB);
```

$$QF := \left[\left[x_0, x_2, x_{12}, x_{12} + 2\, x_{23}\right], \left[x_0 + x_{13}, x_1, x_1 - 2\, x_3, x_2 - x_{123}\right], [\,]\right] \quad (12)$$

```
C3 > map(nops,QF);
```

$$[4, 4, 0] \quad (13)$$

```
C3 > R:=Matrix([[3,0,0],[0,1,0],[0,0,-1]]);
```

$$R := \begin{bmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad (14)$$

```
C3 > CD3 := AlgebraLibraryData("Clifford(3)",C3b,quadraticform=R);
```

$$CD3 := [e1^2 = e1, e1.e2 = e2, e1.e3 = e3, e1.e4 = e4, e1.e5 = e5, e1.e6 = e6, e1.e7 = e7, e1.e8 = e8, e2.e1 = e2, e2^2 = -3\, e1, e2.e3 = e5, e2.e4 = e6, e2.e5 = -3\, e3, e2.e6 = -3\, e4, e2.e7 = e8, e2.e8 = -3\, e7, e3.e1 = e3, e3.e2 = -e5, e3^2 = -e1, e3.e4 = e7, e3.e5 = e2, e3.e6 = -e8, e3.e7 = -e4, e3.e8 = e6, e4.e1 = e4, e4.e2 = -e6, e4.e3 = -e7, e4^2 = e1, e4.e5 = e8, e4.e6 = -e2, e4.e7 = -e3, e4.e8 = e5, e5.e1 = e5, e5.e2 = 3\, e3, e5.e3 = -e2, e5.e4 = e8, e5^2 = -3\, e1, e5.e6 = 3\, e7, e5.e7 = -e6, e5.e8 = -3\, e4, e6.e1 = e6, e6.e2 = 3\, e4, e6.e3 = -e8, e6.e4 = e2, e6.e5 = -3\, e7, e6^2 = 3\, e1, e6.e7 = -e5, e6.e8 = -3\, e3, e7.e1 = e7, e7.e2 = e8, e7.e3 = e4, e7.e4 = e3, e7.e5 = e6, e7.e6 = e5, e7^2 = e1, e7.e8 = e2, e8.e1 = e8, e8.e2 = -3\, e7, e8.e3 = e6, e8.e4 = e5, e8.e5 = -3\, e4, e8.e6 = -3\, e3, e8.e7 = e2, e8^2 = -3\, e1] \quad (15)$$

```
C3 > DGsetup(CD3,[y[0],y[1],y[2],y[3],y[12],y[13],y[23],y[123]],
                 [f[0],f[1],f[2],f[3],f[12],f[13],f[23],f[123]]);
```

algebra name: C3b (16)

```
C3b > KB := KillingForm(C3b);
```

$$KB := 8\, f_0 f_0 - 24\, f_1 f_1 - 8\, f_2 f_2 + 8\, f_3 f_3 - 24\, f_{12} f_{12} + 24\, f_{13} f_{13} + 8\, f_{23} f_{23} - 24\, f_{123} f_{123} \quad (17)$$

```
C3b > QF:=QuadraticFormSignature(KB);
```

$$QF := \left[\left[y_0, y_3, y_{13}, y_{23}\right], \left[y_1, y_2, y_{12}, y_{123}\right], [\,]\right] \quad (18)$$

How To Decompose A Lie Algebra Representation

Synopsis

• We show how to decompose a representation of a semi-simple Lie algebra in Maple using the DecomposeRepresentation command.

Commands Illustrated

• DecomposeRepresentation, DGsetup, Representation, SimpleLieAlgebraData, StandardRepresentation, TensorProductOfRepresentations,

Examples

Load in the required packages.

```
> with(DifferentialGeometry):
  with(LieAlgebras):
  with(Tools):
  with(Tensor):
```

Example 1.

Use the command SimpleLieAlgebraData to read in the structure constants for the Lie algebra sl(4).

```
> LA := SimpleLieAlgebraData("sl(4)",alg):
```

We use the command DGsetup to store these structure equations in memory.

```
> DGsetup(LA);
```

Lie algebra: alg **(1)**

At this point one can now invoke many of the commands in the LieAlgebras package. Now, we use the command StandardRepresentation to retrieve the standard representation of sl(4).

```
alg > M:=StandardRepresentation(alg);
```

$$M := \left[\begin{bmatrix} 1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&-1 \end{bmatrix}, \begin{bmatrix} 0&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&-1 \end{bmatrix}, \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&1&0\\0&0&0&-1 \end{bmatrix}, \begin{bmatrix} 0&1&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&1&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&1\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix}, \right.$$
$$\begin{bmatrix} 0&0&0&0\\1&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0\\0&0&1&0\\0&0&0&0\\0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0\\0&0&0&1\\0&0&0&0\\0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0\\0&0&0&0\\1&0&0&0\\0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&1&0&0\\0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0\\0&0&0&1\\0&0&0&0\\0&0&0&0 \end{bmatrix},$$
$$\left.\begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\1&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&1&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&1&0 \end{bmatrix}\right] \quad \textbf{(2)}$$

Next we initialize a frame for the representation space. Note that the dimension of the representation space is the same as the dimension of the matrices above.

```
alg > DGsetup([seq(v||i,i=1..4)],V);
```
frame name: V **(3)**

Build the standard representation.

```
V > rho:=Representation(alg,V,M):
```

Because the standard representation is irreducible, to make things a little more interesting let's build a new representation by taking a tensor product of the standard representation. Before we do, we need to initialize a frame for the range space.

```
V > DGsetup([seq(w||i,i=1..4^3)],W);
```
frame name: W **(4)**

Next, we build the tensor product of the standard representation.

```
W > phi:=TensorProductOfRepresentations([rho,rho,rho],W):
```

This new representation may not be irreducible, but we know that it can be decomposed into a sum of irreducible representations. Using the command DecomposeRepresentation, we can identify the irreducible components of the decomposition.

```
W > HWMark,HighestWeight,HWVector:=DecomposeRepresentation(phi,
                    output=["Mark","Weight","Vector"],print=true):
```
$$\left[\ \Gamma_{3,0,0} \ \oplus\ 2\,\Gamma_{1,1,0} \ \oplus\ \Gamma_{0,0,1}\ \right] \qquad \textbf{(5)}$$

Example 2.

We can create the Lie algebra by specifying the structure constants.

```
> structEqs:=[[x1,x2]=x2, [x1,x3]=x3, [x1,x4]=-x4, [x1,x5]=-x5,
                  [x2,x4]=-x1,[x2,x5]=x6, [x2,x6]=-x3, [x3,x4]=-x6,
                  [x3,x5]=-x1,[x3,x6]=x2, [x4,x6]=-x5, [x5,x6]=x4 ];
```
structEqs := [[*x1*, *x2*] = *x2*, [*x1*, *x3*] = *x3*, [*x1*, *x4*] = −*x4*, [*x1*, *x5*] = −*x5*, [*x2*, *x4*] = −*x1*, [*x2*, *x5*] = *x6*, [*x2*, *x6*] = −*x3*, [*x3*, *x4*] = −*x6*, [*x3*, *x5*] = −*x1*, [*x3*, *x6*] = *x2*, [*x4*, *x6*] = −*x5*, [*x5*, *x6*] = *x4*] **(6)**

Next, read in the structure constants using the LieAlgebraData, and initialize the frame.

```
> LA:=LieAlgebraData(structEqs,[x1,x2,x3,x4,x5,x6],so31);
```
LA := [[*e1*, *e2*] = *e2*, [*e1*, *e3*] = *e3*, [*e1*, *e4*] = −*e4*, [*e1*, *e5*] = −*e5*, [*e2*, *e4*] = −*e1*, [*e2*, *e5*] = *e6*, [*e2*, *e6*] = −*e3*, [*e3*, *e4*] = −*e6*, [*e3*, *e5*] = −*e1*, [*e3*, *e6*] = *e2*, [*e4*, *e6*] = −*e5*, [*e5*, *e6*] = *e4*] **(7)**

```
> DGsetup(LA):
```

Because we did not use the command SimpleLieAlgebraData to obtain the algebra data, we cannot use the

command StandardRepresentation to obtain a matrix representation. However, the Adjoint is always available for use.

```
so31 > AD:=Adjoint(so31):
```

Now that we have a matrix representation for our algebra, create a frame for the range space and create the representation.

```
so31 > DGsetup([seq(v||i,i=1..6)],V);
```

frame name: V **(8)**

```
V > rho:=Representation(so31,V,AD);
```

$$\rho := \left[e1, \begin{bmatrix} 0&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&-1&0&0\\ 0&0&0&0&-1&0\\ 0&0&0&0&0&0 \end{bmatrix}, e2, \begin{bmatrix} 0&0&0&-1&0&0\\ -1&0&0&0&0&0\\ 0&0&0&0&0&-1\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&1&0 \end{bmatrix}, e3, \begin{bmatrix} 0&0&0&0&-1&0\\ 0&0&0&0&0&1\\ -1&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&-1&0&0 \end{bmatrix}, e4, \right. \quad \textbf{(9)}$$

$$\left. \begin{bmatrix} 0&1&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 1&0&0&0&0&0\\ 0&0&0&0&0&-1\\ 0&0&1&0&0&0 \end{bmatrix}, e5, \begin{bmatrix} 0&0&1&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&1\\ 1&0&0&0&0&0\\ 0&-1&0&0&0&0 \end{bmatrix}, e6, \begin{bmatrix} 0&0&0&0&0&0\\ 0&0&-1&0&0&0\\ 0&1&0&0&0&0\\ 0&0&0&0&-1&0\\ 0&0&0&1&0&0\\ 0&0&0&0&0&0 \end{bmatrix} \right] ,$$

Again, it will be more interesting to look at how a more complex representation decomposes. Therefore, we will create a new representation by taking the tensor product of the adjoint representation. So we will look at how the representation $\varphi = Ad(V) \otimes Ad(V)$ decomposes.

```
V > DGsetup([seq(w||k,k=1..6^2)],W);
```

frame name: W **(10)**

```
W > phi:=TensorProductOfRepresentations([rho,rho],W):
so31 > CSA:=CartanSubalgebra(so31);
```

$$CSA := [e1, e6] \quad \textbf{(11)}$$

```
W > M,H,V,Q:=DecomposeRepresentation(phi,CSA,print=true):
```

$$\left[\Gamma_{4,0} \oplus 2\,\Gamma_{2,2} \oplus \Gamma_{2,0} \oplus \Gamma_{0,4} \oplus \Gamma_{0,2} \oplus 2\,\Gamma_{0,0} \right] \quad \textbf{(12)}$$

Example 3.

We can also create a Lie algebra by using a matrix representation. Then we can test this representation to see if it is irreducible or whether it decomposes. A representation has been provided and is contained in the section "Matrix Representation".

Matrix Representation

```
> MatEntries:=[
      {(1 ,2 )= 2,(1 ,7 )= 1,(2 ,2 )= 2,(3 ,1 )= 1,(3 ,2 )= 2,(3 ,3 )= 1,(3 ,7 )=-1,
       (3 ,9 )=-3,(4 ,2 )= 2,(5 ,5 )=-1,(6 ,2 )= 2,(7 ,7 )=-1,(8 ,2 )= 1,(8 ,8 )= 1,
       (9 ,2 )= 4,(9 ,9 )=-2,(10,1 )= 1,(10,2 )= 7,(10,3 )= 1,(10,5 )= 1,(10,8 )= 1,
       (10,9 )=-3},
      {(1 ,1 )= 1,(1 ,2 )= 1,(1 ,4 )=-1,(1 ,6 )=-1,(1 ,7 )= 2,(3 ,2 )= 1,(3 ,3 )= 1,
       (3 ,4 )= 1,(3 ,6 )= 1,(3 ,7 )=-2,(3 ,9 )=-3,(4 ,2 )= 1,(4 ,4 )=-1,(6 ,2 )= 1,
       (6 ,4 )=-1,(7 ,7 )=-1,(9 ,2 )= 2,(9 ,9 )=-2,(10,1 )=-1,(10,3 )=-1,(10,5 )= 2,
       (10,8 )=-2,(10,9 )=-3,(10,10)= 2},
      {(1 ,1 )= 1,(1 ,2 )= 1,(1 ,4 )=-3,(1 ,6 )= 1,(1 ,7 )= 1,(3 ,1 )=-1,(3 ,2 )= 1,
       (3 ,4 )= 3,(3 ,6 )=-1,(3 ,7 )=-1,(3 ,9 )=-2,(4 ,2 )= 1,(4 ,4 )=-1,(5 ,5 )=-1,
       (6 ,2 )= 1,(6 ,4 )=-3,(6 ,6 )= 2,(8 ,2 )=-1,(8 ,8 )= 1,(9 ,2 )= 2,(9 ,9 )=-2,
       (10,2 )= 1,(10,5 )= 1,(10,8 )= 1,(10,9 )=-2},
      {(1 ,1 )=-1,(1 ,3 )=-1,(1 ,5 )=-1,(1 ,9 )= 1,(2 ,1 )=-1,(2 ,3 )=-1,(2 ,9 )= 1,
       (3 ,1 )= 2,(3 ,2 )= 2,(3 ,3 )= 2,(3 ,5 )=-1,(3 ,8 )= 2,(3 ,10)=-2,(4 ,1 )=-1,
       (4 ,3 )=-1,(4 ,5 )=-1,(4 ,9 )= 1,(6 ,1 )=-1,(6 ,3 )=-1,(6 ,5 )=-1,(6 ,9 )= 1,
       (8 ,1 )=-2,(8 ,3 )=-1,(8 ,6 )= 1,(8 ,7 )=-1,(8 ,9 )= 1,(9 ,1 )=-1,(9 ,3 )=-1,
       (9 ,9 )= 1,(10,1 )=-2,(10,2 )= 2,(10,3 )=-1,(10,5 )=-2,(10,6 )= 1,(10,7 )=-1,
       (10,8 )= 2,(10,9 )= 3,(10,10)=-2},
      {(1 ,2 )=-1,(1 ,7 )= 1,(1 ,8 )= 1,(2 ,2 )=-1,(2 ,8 )= 1,(3 ,1 )= 1,(3 ,6 )=-1,
       (4 ,2 )=-1,(4 ,7 )= 1,(4 ,8 )= 1,(6 ,2 )=-1,(6 ,7 )= 1,(6 ,8 )= 1,(8 ,2 )=-1,
       (8 ,4 )=-2,(8 ,6 )= 2,(8 ,8 )= 1,(9 ,2 )=-1,(9 ,8 )= 1,(10,1 )= 1,(10,2 )=-3,
       (10,4 )=-2,(10,6 )= 1,(10,7 )= 1,(10,8 )= 3},
      {(1 ,2 )=-3,(1 ,4 )= 1,(1 ,9 )= 2,(2 ,2 )=-1,(2 ,4 )= 1,(3 ,2 )= 2,(3 ,5 )= 1,
       (3 ,9 )=-2,(4 ,2 )=-3,(4 ,4 )= 1,(4 ,9 )= 2,(6 ,2 )=-3,(6 ,4 )= 1,(6 ,9 )= 2,
       (8 ,2 )=-1,(8 ,4 )= 1,(8 ,7 )= 1,(9 ,2 )=-1,(9 ,4 )= 1,(10,2 )=-3,(10,4 )= 3,
       (10,5 )= 1,(10,7 )= 1},
      {(1 ,2 )= 1,(1 ,8 )=-1,(3 ,2 )=-3,(3 ,8 )= 1,(5 ,2 )= 1,(5 ,4 )=-1,(10,1 )=-1,
       (10,2 )=-3,(10,3 )=-1,(10,4 )= 1,(10,9 )= 1},
      {(1 ,4 )= 2,(1 ,6 )=-2,(3 ,2 )= 1,(3 ,4 )=-2,(3 ,6 )= 2,(3 ,8 )=-1,(5 ,7 )=-1,
       (10,1 )=-1,(10,2 )= 1,(10,6 )= 1,(10,8 )=-1},
      {(1 ,7 )=-1,(3 ,2 )= 1,(3 ,4 )=-1,(3 ,7 )= 1,(5 ,2 )= 2,(5 ,9 )=-2,(10,2 )=-1,
       (10,4 )=-1,(10,5 )=-1,(10,9 )= 2},
      {(1 ,1 )= 1,(1 ,3 )= 1,(1 ,4 )=-1,(1 ,8 )= 1,(1 ,9 )=-1,(3 ,1 )=-1,(3 ,3 )=-1,
       (3 ,4 )= 1,(3 ,8 )=-1,(3 ,9 )= 1,(6 ,2 )=-1,(6 ,8 )= 1,(7 ,2 )=-1,(7 ,4 )= 1,
       (8 ,2 )= 2,(10,2 )= 2},
      {(1 ,1 )= 1,(1 ,2 )= 2,(1 ,3 )= 2,(1 ,5 )=-1,(1 ,6 )= 1,(1 ,7 )=-1,(1 ,8 )= 2,
       (1 ,10)=-2,(3 ,1 )=-1,(3 ,2 )=-2,(3 ,3 )=-2,(3 ,5 )= 1,(3 ,6 )=-1,(3 ,7 )= 1,
       (3 ,8 )=-2,(3 ,10)= 2,(6 ,1 )=-1,(6 ,6 )= 1,(6 , 7)=-1,(7 , 5)=-1,(8 ,1 )=-1,
       (8 ,3 )=-1,(8 ,9 )= 1,(10,1 )=-1,(10,3 )=-1,(10,9 )= 1},
      {(1 ,2 )= 2,(1 ,5 )= 1,(1 ,7 )= 1,(1 ,9 )=-2,(3 ,2 )=-2,(3 ,5 )=-1,(3 ,7 )=-1,
       (3 ,9 )= 2,(6 ,7 )= 1,(7 ,2 )=-2,(7 ,9 )= 2,(8 ,2 )=-1,(8 ,4 )= 1,(10, 2)=-1,
       (10, 4)= 1},
      {(1 ,2 )= 3,(1 ,8 )=-1,(3 ,2 )=-4,(3 ,4 )= 1,(3 ,8 )= 1,(4 ,2 )= 2,(5 ,1 )= 1,
       (5 ,3 )= 1,(5 ,9 )=-1,(6 ,2 )= 2,(7 ,2 )=-1,(7 ,8 )= 1,(9 ,2 )=-1,(9 ,4 )= 1,
       (10,1 )=-1,(10,2 )=-1,(10,3 )=-1,(10,4 )= 1,(10,9 )= 1},
      {(1 ,3 )=-1,(1 ,6 )=-1,(1 ,7 )= 1,(1 ,9 )= 1,(3 ,3 )= 1,(3 ,5 )=-1,(3 ,6 )= 1,
       (3 ,7 )=-1,(3 ,9 )=-1,(4 ,1 )=-1,(4 ,3 )=-1,(4 ,9 )= 1,(5 ,1 )= 2,(5 ,2 )= 2,
       (5 ,3 )= 2,(5 ,5 )=-2,(5 ,8 )= 2,(5 ,10)=-2,(6 ,1 )=-1,(6 ,3 )=-1,(6 ,9 )= 1,
       (7 ,1 )=-1,(7 ,6 )= 1,(7 ,7 )=-1,(9 ,5 )=-1,(10,1 )=-2,(10,2 )=-2,(10,3 )=-2,
       (10,5 )= 1,(10,8 )=-2,(10,10)=2},
      {(1 ,2 )=-1,(1 ,4 )= 2,(1 ,6 )=-2,(1 ,8 )= 1,(3 ,2 )= 1,(3 ,4 )=-2,(3 ,6 )= 2,
       (3 ,7 )= 1,(3 ,8 )=-1,(4 ,2 )=-1,(4 ,8 )= 1,(5 ,1 )= 1,(5 ,6 )=-1,(5 ,7 )= 1,
       (6 ,2 )=-1,(6 ,8 )= 1,(7 ,4 )=-2,(7 ,6 )= 2,(9 ,7 )= 1,(10,1 )=-1,(10,6 )=1}
      ]:
> M:=map(ind->Matrix(10,10,ind),MatEntries):
```

Using the predefined matrices, we can build the Lie algebra using the command LieAlgebraData.

```
> LA := LieAlgebraData(M,alg);
```

```
LA := [[e1, e4] = e4, [e1, e5] = e5, [e1, e6] = 2 e6, [e1, e7] = -e7, [e1, e9] = e9, [e1, e10] = -e10, [e1, e12]
    = e12, [e1, e13] = -2 e13, [e1, e14] = -e14, [e1, e15] = -e15, [e2, e4] = -e4, [e2, e6] = e6, [e2, e7] = e7,
    [e2, e8] = e8, [e2, e9] = 2 e9, [e2, e11] = -e11, [e2, e12] = e12, [e2, e13] = -e13, [e2, e14] = -2 e14, [e2,
    e15] = -e15, [e3, e5] = -e5, [e3, e6] = e6, [e3, e8] = -e8, [e3, e9] = e9, [e3, e10] = e10, [e3, e11] = e11,
    [e3, e12] = 2 e12, [e3, e13] = -e13, [e3, e14] = -e14, [e3, e15] = -2 e15, [e4, e7] = -e2 + e1, [e4, e8]
    = e5, [e4, e9] = e6, [e4, e10] = -e11, [e4, e13] = -e14, [e5, e7] = -e8, [e5, e10] = -e3 + e1, [e5, e11]
    = e4, [e5, e12] = e6, [e5, e13] = -e15, [e6, e7] = -e9, [e6, e10] = -e12, [e6, e13] = e1, [e6, e14] = e4, [e6,
    e15] = e5, [e7, e11] = -e10, [e7, e14] = -e13, [e8, e10] = e7, [e8, e11] = -e3 + e2, [e8, e12] = e9, [e8, e14]
    = -e15, [e9, e11] = -e12, [e9, e13] = e7, [e9, e14] = e2, [e9, e15] = e8, [e10, e15] = -e13, [e11, e15] =
    -e14, [e12, e13] = e10, [e12, e14] = e11, [e12, e15] = e3]
> DGsetup(LA);
```

(13)

The dimension of the representation space is 10, so we can initialize a frame for the range space.

```
so31 > DGsetup([seq(u||k,k=1..10)],U);
                                    frame name: U
```

(14)

Next, we build the representation and then decompose it.

```
so31 > rho:=Representation(alg,U,M):
W > DecomposeRepresentation(rho,print=true);
```

$$\left[\Gamma_{2,0,0} \right]$$

(15)

Decomposing the Representation Der(O)

Synopsis

• We show how to build the Lie algebra $\mathfrak{g}_2$ from its representation Der($\mathbb{O}$) (the derivation algebra of the octonions). Then we decompose this representation using the DecomposeRepresentation command and show how to transform the representation into a direct (matrix) sum of irreducible representations.

Commands Illustrated

• AlgebraLibraryData, DecomposeRepresentation, Derivations, DGsetup, LieAlgebraData, Representation,

Examples

Load in the required packages.

```
> with(DifferentialGeometry):
  with(LieAlgebras):
```

Example 1.

Use the command AlgebraLibraryData to read in the structure constants for the algebra of octonions.

```
> AD := AlgebraLibraryData("Octonions",O);
```

$$AD := [e1^2 = e1,\ e1.e2 = e2,\ e1.e3 = e3,\ e1.e4 = e4,\ e1.e5 = e5,\ e1.e6 = e6,\ e1.e7 = e7,\ e1.e8 = e8,\ e2.e1 = e2,\ e2^2 = -e1,\ e2.e3 = e4,\ e2.e4 = -e3,\ e2.e5 = e6,\ e2.e6 = -e5,\ e2.e7 = -e8,\ e2.e8 = e7,\ e3.e1 = e3,\ e3.e2 = -e4,\ e3^2 = -e1,\ e3.e4 = e2,\ e3.e5 = e7,\ e3.e6 = e8,\ e3.e7 = -e5,\ e3.e8 = -e6,\ e4.e1 = e4,\ e4.e2 = e3,\ e4.e3 = -e2,\ e4^2 = -e1,\ e4.e5 = e8,\ e4.e6 = -e7,\ e4.e7 = e6,\ e4.e8 = -e5,\ e5.e1 = e5,\ e5.e2 = -e6,\ e5.e3 = -e7,\ e5.e4 = -e8,\ e5^2 = -e1,\ e5.e6 = e2,\ e5.e7 = e3,\ e5.e8 = e4,\ e6.e1 = e6,\ e6.e2 = e5,\ e6.e3 = -e8,\ e6.e4 = e7,\ e6.e5 = -e2,\ e6^2 = -e1,\ e6.e7 = -e4,\ e6.e8 = e3,\ e7.e1 = e7,\ e7.e2 = e8,\ e7.e3 = e5,\ e7.e4 = -e6,\ e7.e5 = -e3,\ e7.e6 = e4,\ e7^2 = -e1,\ e7.e8 = -e2,\ e8.e1 = e8,\ e8.e2 = -e7,\ e8.e3 = e6,\ e8.e4 = e5,\ e8.e5 = -e4,\ e8.e6 = -e3,\ e8.e7 = e2,\ e8^2 = -e1] \quad (1)$$

We use the command DGsetup to store these structure equations in memory.

```
> DGsetup(AD);
```

algebra name: O (2)

At this point one can now invoke many of the commands in the LieAlgebras package. We use the command Derivations to obtain the derivations algebra over the octonions.

```
O > Der:=Derivations(O);
```

$$Der := \left[\begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&-1&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&-1&0&0\\0&0&0&0&0&0&0&0\end{bmatrix}, \begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0&0&0&0\\0&-1&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&0&0\\0&0&0&0&0&-1&0&0\end{bmatrix}, \begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&-1&0\\0&-1&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0&0&0&0\end{bmatrix},\right.$$

$$\begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&-1\\0&0&0&0&0&0&0&0\\0&-1&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\end{bmatrix}, \begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&-1&0&0&0&0\\0&0&0&0&0&0&0&0\\0&-1&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{bmatrix}, \begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&0\\0&0&0&-1&0&0&0&0\\0&0&0&0&0&0&0&0\\0&-1&0&0&0&0&0&0\end{bmatrix},$$

$$\begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&-1&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&-1&0\end{bmatrix}, \begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&-1&0&0&0&0&0\\0&0&0&-1&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{bmatrix}, \begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&-1&0&0&0\\0&0&0&1&0&0&0&0\\0&0&-1&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{bmatrix},$$

$$\begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&-1\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&-1&0&0&0&0&0\\0&0&0&1&0&0&0&0\end{bmatrix}, \begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&-1&0&0&0&0\\0&0&-1&0&0&0&0&0\end{bmatrix}, \begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&-1&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&-1&0\end{bmatrix},$$

$$\left.\begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&-1\\0&0&0&0&-1&0&0&0\\0&0&0&0&0&1&0&0\end{bmatrix}, \begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\\0&0&0&0&0&-1&0&0\\0&0&0&0&-1&0&0&0\end{bmatrix}\right] \quad \textbf{(3)}$$

We know that the derivation algebra of the octonions is the Lie algebra $\mathfrak{g}_2$. Therefore, we can create and initialize the structure for the Lie algebra using the matrices in Der($\mathbb{O}$).

```
> LD:=LieAlgebraData(Der,g2):
> DGsetup(LD);
```

Lie algebra: g2 **(4)**

To make things interesting, let's change the basis of the matrix representation. The following matrix given below, S, is a matrix such that for each matrix X in Der($\mathbb{O}$), $R = S \cdot X \cdot S^{-1}$ is also a derivation on $\mathbb{O}$. Note that the action on Der($\mathbb{O}$) by S does not change the bracket rules for the Lie algebra. This can be seen by

$$[SXS^{-1}, SYS^{-1}] = SXS^{-1}SYS^{-1} - SYS^{-1}SXS^{-1} = SXYS^{-1} - SYXS^{-1} = S[X, Y]S^{-1}.$$

```
> S:=Matrix(8,8,{(1,1)= 3,(1,4)= 1,(1,6)=-3,(1,7)=-1,(2,1)=-2,
                 (2,2)=-2,(2,5)= 2,(2,6)= 3,(2,7)=-2,(2,8)=-1,
                 (3,1)=-3,(3,5)=-3,(3,6)=-3,(3,8)=-1,(4,3)=-3,
                 (4,4)=-3,(4,5)=-2,(4,6)= 1,(4,7)=-2,(4,8)= 1,
                 (5,1)= 3,(5,2)=-1,(5,3)= 3,(5,7)= 1,(6,2)= 1,
                 (6,4)=-1,(6,6)=-1,(6,7)=-1,(7,4)=-1,(7,7)=-3,
                 (8,2)=-1,(8,8)= 1});
```

$$S := \begin{bmatrix} 3 & 0 & 0 & 1 & 0 & -3 & -1 & 0 \\ -2 & -2 & 0 & 0 & 2 & 3 & -2 & -1 \\ -3 & 0 & 0 & 0 & -3 & -3 & 0 & -1 \\ 0 & 0 & -3 & -3 & -2 & 1 & -2 & 1 \\ 3 & -1 & 3 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & -3 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(5)}$$

```
> SI:=S^(-1):
> R:=map(D->S.D.SI,Der):
```

To see what we've done, let's compare a specific derivation before and after the transformation,

```
g2 > Der[8],R[8];
```

(6)

$$\left[\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} -\frac{49}{200} & -\frac{411}{400} & -\frac{47}{200} & -\frac{27}{40} & -\frac{27}{40} & -\frac{1327}{400} & \frac{839}{400} & -\frac{47}{80} \\ \frac{281}{600} & \frac{453}{400} & \frac{43}{600} & \frac{41}{40} & \frac{43}{120} & \frac{1121}{400} & -\frac{2891}{1200} & \frac{43}{240} \\ -\frac{17}{25} & -\frac{63}{50} & -\frac{1}{25} & -\frac{6}{5} & -\frac{1}{5} & -\frac{141}{50} & \frac{137}{50} & -\frac{1}{10} \\ -\frac{119}{600} & -\frac{147}{400} & \frac{443}{600} & \frac{1}{40} & \frac{83}{120} & \frac{121}{400} & \frac{509}{1200} & \frac{83}{240} \\ \frac{39}{100} & \frac{171}{200} & -\frac{33}{100} & -\frac{3}{20} & -\frac{3}{20} & \frac{447}{200} & -\frac{279}{200} & \frac{27}{40} \\ \frac{37}{200} & -\frac{57}{400} & \frac{11}{200} & -\frac{9}{40} & -\frac{9}{40} & -\frac{149}{400} & \frac{93}{400} & \frac{11}{80} \\ \frac{1}{5} & \frac{3}{20} & \frac{1}{10} & 0 & 0 & \frac{11}{20} & -\frac{7}{20} & \frac{1}{4} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right] \quad \textbf{(6)}$$

Next we initialize a frame for the representation space. Note that the dimension of the representation space is the same as the dimension of the matrices above.

```
g2 > DGsetup([seq(v||i,i=1..8)],V):
```

Then we build the standard representation.

```
V > rho:=Representation(g2,V,R):
```

This representation may not be irreducible, but we know that it can be decomposed into a sum of irreducible representations. Using the command DecomposeRepresentation, we can identify the irreducible components of the decomposition and return a change of basis matrix that will allow us to transform the representation "rho" into block sums of irreducible representations. We return the marks which identify the irreducible representations. We also return the transformation matrix which allows us the change the basis of the representation so that the matrices are block diagonal matrices.

```
g2 > Marks,Q:=DecomposeRepresentation(rho,
                output=["Marks","Transform"],print=true):
```

$$\left[\Gamma_{1,0} \oplus \Gamma_{0,0}\right] \quad \textbf{(7)}$$

This shows that the representation Der(𝕆) for $\mathfrak{g}_2$ is not irreducible. However, using the matrix Q, we can perform another change of basis procedure on the basis elements to transform them into block sums of irreducible representations.

```
g2 > QI:=Q^(-1):
g2 > T:=map(r->QI.r.Q,R);
```

$$T := \left[\begin{bmatrix} I & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & I & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -I & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & -32\,I & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 32\,I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{64}\,I & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{64}\,I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad \textbf{(8)}$$

$$\begin{bmatrix} 0 & 0 & 2\,I & 8 & 16\,I & 0 & 0 & 0 \\ 0 & 0 & -1 & -4\,I & -8 & 0 & 0 & 0 \\ \frac{1}{16}\,I & \frac{1}{8} & 0 & 0 & 0 & -8 & -16\,I & 0 \\ -\frac{1}{32} & -\frac{1}{16}\,I & 0 & 0 & 0 & 4\,I & 8 & 0 \\ \frac{1}{128}\,I & \frac{1}{64} & 0 & 0 & 0 & -1 & -2\,I & 0 \\ 0 & 0 & \frac{1}{64} & \frac{1}{16}\,I & \frac{1}{8} & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{128}\,I & -\frac{1}{32} & -\frac{1}{16}\,I & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} \frac{1}{2}\,I & 0 & 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & -\frac{1}{2}\,I & 0 & 0 & 0 & 0 & 64 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -I & 0 & 0 & 0 \\ -\frac{1}{256} & 0 & 0 & 0 & 0 & \frac{1}{2}\,I & 0 & 0 \\ 0 & -\frac{1}{256} & 0 & 0 & 0 & 0 & -\frac{1}{2}\,I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\begin{bmatrix} 0 & I & 0 & 0 & 0 & -64\,I & 0 & 0 \\ \frac{1}{4}\,I & 0 & 0 & 0 & 0 & 0 & -64\,I & 0 \\ 0 & 0 & 0 & 2\,I & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{4}\,I & 0 & 2\,I & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{4}\,I & 0 & 0 & 0 & 0 \\ -\frac{1}{256}\,I & 0 & 0 & 0 & 0 & 0 & I & 0 \\ 0 & -\frac{1}{256}\,I & 0 & 0 & 0 & \frac{1}{4}\,I & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 32 & 0 & 0 & 0 \\ 0 & 0 & 2\,I & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4}\,I & 0 & 0 & 0 & 0 & -32 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{64} & 0 & 0 & 0 & 0 & 2\,I & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4}\,I & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{64} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & -32 & 0 & 0 & 0 \\ 0 & 0 & 2\,I & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} I & 0 & 0 & 0 & 0 & 32 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{64} & 0 & 0 & 0 & 0 & 2\,I & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} I & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{64} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 2 & -8\,I & 16 & 0 & 0 & 0 \\ 0 & 0 & I & -4 & 8\,I & 0 & 0 & 0 \\ -\frac{1}{16} & \frac{1}{8} I & 0 & 0 & 0 & 8\,I & -16 & 0 \\ -\frac{1}{32} I & \frac{1}{16} & 0 & 0 & 0 & 4 & -8\,I & 0 \\ -\frac{1}{128} & \frac{1}{64} I & 0 & 0 & 0 & I & -2 & 0 \\ 0 & 0 & \frac{1}{64} I & -\frac{1}{16} & \frac{1}{8} I & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{128} & -\frac{1}{32} I & \frac{1}{16} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\begin{bmatrix} 0 & -I & 0 & 0 & 0 & -64\,I & 0 & 0 \\ -\frac{1}{4} I & 0 & 0 & 0 & 0 & 0 & -64\,I & 0 \\ 0 & 0 & 0 & -2\,I & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{4} I & 0 & -2\,I & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{4} I & 0 & 0 & 0 & 0 \\ -\frac{1}{256} I & 0 & 0 & 0 & 0 & 0 & -I & 0 \\ 0 & -\frac{1}{256} I & 0 & 0 & 0 & -\frac{1}{4} I & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} \frac{1}{2} I & 0 & 0 & 0 & 0 & -64 & 0 & 0 \\ 0 & -\frac{1}{2} I & 0 & 0 & 0 & 0 & -64 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -I & 0 & 0 & 0 \\ \frac{1}{256} & 0 & 0 & 0 & 0 & \frac{1}{2} I & 0 & 0 \\ 0 & \frac{1}{256} & 0 & 0 & 0 & 0 & -\frac{1}{2} I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & -32\,I & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{4} & 0 & 0 & 0 & 0 & 32\,I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{64} I & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{64} I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 2 & 8\,I & -16 & 0 & 0 & 0 \\ 0 & 0 & I & -4 & -8\,I & 0 & 0 & 0 \\ -\frac{1}{16} & \frac{1}{8} I & 0 & 0 & 0 & -8\,I & 16 & 0 \\ \frac{1}{32} I & \frac{1}{16} & 0 & 0 & 0 & 4 & 8\,I & 0 \\ \frac{1}{128} & -\frac{1}{64} I & 0 & 0 & 0 & I & -2 & 0 \\ 0 & 0 & -\frac{1}{64} I & -\frac{1}{16} & \frac{1}{8} I & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{128} & \frac{1}{32} I & \frac{1}{16} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\left[\begin{bmatrix} 0 & 0 & -2\,I & 8 & 16\,I & 0 & 0 & 0 \\ 0 & 0 & 1 & 4\,I & -8 & 0 & 0 & 0 \\ -\frac{1}{16}\,I & -\frac{1}{8} & 0 & 0 & 0 & -8 & -16\,I & 0 \\ -\frac{1}{32} & \frac{1}{16}\,I & 0 & 0 & 0 & -4\,I & 8 & 0 \\ \frac{1}{128}\,I & \frac{1}{64} & 0 & 0 & 0 & 1 & 2\,I & 0 \\ 0 & 0 & \frac{1}{64} & -\frac{1}{16}\,I & -\frac{1}{8} & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{128}\,I & -\frac{1}{32} & \frac{1}{16}\,I & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} \frac{1}{2}\,I & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{4} & \frac{1}{2}\,I & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{4} & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{2}\,I & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{4} & -\frac{1}{2}\,I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right]$$

Notice that the last row and last column are zeros. We see that the irreducible representation $\Gamma_{1,0}$ is a 7-dimensional representation.

```
g2 > DGsetup([seq(u||i,i=1..7)],U):
U > phi:=Representation(g2,U,map(x->x[1..7,1..7],T));
```

$$\phi := \left[\left[\left[e1, \begin{bmatrix} I & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -I \end{bmatrix}\right]\right], \left[\left[e2, \begin{bmatrix} 0 & 0 & 0 & 0 & -32\,I & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 32\,I \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{I}{64} & 0 & 0 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 \\ 0 & 0 & \frac{I}{64} & 0 & 0 & 0 & 0 \end{bmatrix}\right]\right], \left[e3, \right]\right] \quad \textbf{(9)}$$

$$\left[\left[\begin{matrix} 0 & 0 & 2\,I & 8 & 16\,I & 0 & 0 \\ 0 & 0 & -1 & -4\,I & -8 & 0 & 0 \\ \frac{I}{16} & \frac{1}{8} & 0 & 0 & 0 & -8 & -16\,I \\ -\frac{1}{32} & -\frac{I}{16} & 0 & 0 & 0 & 4\,I & 8 \\ \frac{I}{128} & \frac{1}{64} & 0 & 0 & 0 & -1 & -2\,I \\ 0 & 0 & \frac{1}{64} & \frac{I}{16} & \frac{1}{8} & 0 & 0 \\ 0 & 0 & -\frac{I}{128} & -\frac{1}{32} & -\frac{I}{16} & 0 & 0 \end{matrix}\right]\right], e4,$$

$$\left[\left[\begin{matrix} \frac{I}{2} & 0 & 0 & 0 & 0 & 64 & 0 \\ 0 & -\frac{I}{2} & 0 & 0 & 0 & 0 & 64 \\ 0 & 0 & I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -I & 0 & 0 \\ -\frac{1}{256} & 0 & 0 & 0 & 0 & \frac{I}{2} & 0 \\ 0 & -\frac{1}{256} & 0 & 0 & 0 & 0 & -\frac{I}{2} \end{matrix}\right]\right], e5, \left[\left[\begin{matrix} 0 & I & 0 & 0 & 0 & -64\,I & 0 \\ \frac{I}{4} & 0 & 0 & 0 & 0 & 0 & -64\,I \\ 0 & 0 & 0 & 2\,I & 0 & 0 & 0 \\ 0 & 0 & \frac{I}{4} & 0 & 2\,I & 0 & 0 \\ 0 & 0 & 0 & \frac{I}{4} & 0 & 0 & 0 \\ -\frac{I}{256} & 0 & 0 & 0 & 0 & 0 & I \\ 0 & -\frac{I}{256} & 0 & 0 & 0 & \frac{I}{4} & 0 \end{matrix}\right]\right],$$

$$e6, \left[\left[\begin{matrix} 0 & 0 & 0 & 0 & 32 & 0 & 0 \\ 0 & 0 & 2\,I & 0 & 0 & 0 & 0 \\ 0 & \frac{I}{4} & 0 & 0 & 0 & 0 & -32 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{64} & 0 & 0 & 0 & 0 & 2\,I & 0 \\ 0 & 0 & 0 & 0 & \frac{I}{4} & 0 & 0 \\ 0 & 0 & \frac{1}{64} & 0 & 0 & 0 & 0 \end{matrix}\right]\right], e7, \left[\left[\begin{matrix} 0 & 0 & 0 & 0 & -32 & 0 & 0 \\ 0 & 0 & 2\,I & 0 & 0 & 0 & 0 \\ 0 & \frac{I}{4} & 0 & 0 & 0 & 0 & 32 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{64} & 0 & 0 & 0 & 0 & 2\,I & 0 \\ 0 & 0 & 0 & 0 & \frac{I}{4} & 0 & 0 \\ 0 & 0 & -\frac{1}{64} & 0 & 0 & 0 & 0 \end{matrix}\right]\right], e8,$$

$$\begin{bmatrix} 0 & 0 & 2 & -8\,I & 16 & 0 & 0 \\ 0 & 0 & I & -4 & 8\,I & 0 & 0 \\ -\frac{1}{16} & \frac{I}{8} & 0 & 0 & 0 & 8\,I & -16 \\ -\frac{I}{32} & \frac{1}{16} & 0 & 0 & 0 & 4 & -8\,I \\ -\frac{1}{128} & \frac{I}{64} & 0 & 0 & 0 & I & -2 \\ 0 & 0 & \frac{I}{64} & -\frac{1}{16} & \frac{I}{8} & 0 & 0 \\ 0 & 0 & \frac{1}{128} & -\frac{I}{32} & \frac{1}{16} & 0 & 0 \end{bmatrix}\Bigg\rbrace, \Bigg\lbrace e9,$$

$$\begin{bmatrix} 0 & -I & 0 & 0 & 0 & -64\,I & 0 \\ -\frac{I}{4} & 0 & 0 & 0 & 0 & 0 & -64\,I \\ 0 & 0 & 0 & -2\,I & 0 & 0 & 0 \\ 0 & 0 & -\frac{I}{4} & 0 & -2\,I & 0 & 0 \\ 0 & 0 & 0 & -\frac{I}{4} & 0 & 0 & 0 \\ -\frac{I}{256} & 0 & 0 & 0 & 0 & 0 & -I \\ 0 & -\frac{I}{256} & 0 & 0 & 0 & -\frac{I}{4} & 0 \end{bmatrix}\Bigg\rbrace, \Bigg\lbrace e10,$$

$$\begin{bmatrix} \frac{I}{2} & 0 & 0 & 0 & 0 & -64 & 0 \\ 0 & -\frac{I}{2} & 0 & 0 & 0 & 0 & -64 \\ 0 & 0 & I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -I & 0 & 0 \\ \frac{1}{256} & 0 & 0 & 0 & 0 & \frac{I}{2} & 0 \\ 0 & \frac{1}{256} & 0 & 0 & 0 & 0 & -\frac{I}{2} \end{bmatrix}\Bigg\rbrace, \Bigg\lbrace e11, \begin{bmatrix} 0 & 0 & 0 & 0 & -32\,I & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{4} & 0 & 0 & 0 & 0 & 32\,I \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{I}{64} & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{4} & 0 & 0 \\ 0 & 0 & \frac{I}{64} & 0 & 0 & 0 & 0 \end{bmatrix}\Bigg\rbrace, \Bigg\lbrace e12,$$

$$\left[\begin{bmatrix} 0 & 0 & 2 & 8\,I & -16 & 0 & 0 \\ 0 & 0 & I & -4 & -8\,I & 0 & 0 \\ -\frac{1}{16} & \frac{I}{8} & 0 & 0 & 0 & -8\,I & 16 \\ \frac{I}{32} & \frac{1}{16} & 0 & 0 & 0 & 4 & 8\,I \\ \frac{1}{128} & -\frac{I}{64} & 0 & 0 & 0 & I & -2 \\ 0 & 0 & -\frac{I}{64} & -\frac{1}{16} & \frac{I}{8} & 0 & 0 \\ 0 & 0 & -\frac{1}{128} & \frac{I}{32} & \frac{1}{16} & 0 & 0 \end{bmatrix}\right], \left[e13,\right.$$

$$\left[\begin{bmatrix} 0 & 0 & -2\,I & 8 & 16\,I & 0 & 0 \\ 0 & 0 & 1 & 4\,I & -8 & 0 & 0 \\ -\frac{I}{16} & -\frac{1}{8} & 0 & 0 & 0 & -8 & -16\,I \\ -\frac{1}{32} & \frac{I}{16} & 0 & 0 & 0 & -4\,I & 8 \\ \frac{I}{128} & \frac{1}{64} & 0 & 0 & 0 & 1 & 2\,I \\ 0 & 0 & \frac{1}{64} & -\frac{I}{16} & -\frac{1}{8} & 0 & 0 \\ 0 & 0 & -\frac{I}{128} & -\frac{1}{32} & \frac{I}{16} & 0 & 0 \end{bmatrix}\right], \left[e14,,\begin{bmatrix} \frac{I}{2} & 1 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{4} & \frac{I}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{4} & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{I}{2} & 1 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{4} & -\frac{I}{2} \end{bmatrix}\right]$$

C.6 Derivations of the Octonions

The Derivations Algebra of the Octonions.

Synopsis

• We show how to create a derivations algebra of the octonions and use this to create the Lie algebra g_2.

Commands Illustrated

• LieAlgebras, DGsetup, AlgebraLibraryData, MultiplicationTable , LieAlgebraData, CartanMatrix, RootSpaceDecomposition, PositiveRoots, SimpleRoots,

Load in the required packages.

```
> with(DifferentialGeometry): with(LieAlgebras): with(Tools):
```

Example 1.

The output of the command AlgebraLibraryData is a list of nonzero products, where by default *e1* is the multiplicative identity, and the remaining objects correspond to the pure imaginary basis elements.

```
> AD := AlgebraLibraryData("Octonions",O);
```

$$AD := [e1^2 = e1,\ e1.e2 = e2,\ e1.e3 = e3,\ e1.e4 = e4,\ e1.e5 = e5,\ e1.e6 = e6,\ e1.e7 = e7,\ e1.e8 = e8,\ e2.e1 = e2,\ e2^2 = -e1,\ e2.e3 = e4,\ e2.e4 = -e3,\ e2.e5 = e6,\ e2.e6 = -e5,\ e2.e7 = -e8,\ e2.e8 = e7,\ e3.e1 = e3,\ e3.e2 = -e4,\ e3^2 = -e1,\ e3.e4 = e2,\ e3.e5 = e7,\ e3.e6 = e8,\ e3.e7 = -e5,\ e3.e8 = -e6,\ e4.e1 = e4,\ e4.e2 = e3,\ e4.e3 = -e2,\ e4^2 = -e1,\ e4.e5 = e8,\ e4.e6 = -e7,\ e4.e7 = e6,\ e4.e8 = -e5,\ e5.e1 = e5,\ e5.e2 = -e6,\ e5.e3 = -e7,\ e5.e4 = -e8,\ e5^2 = -e1,\ e5.e6 = e2,\ e5.e7 = e3,\ e5.e8 = e4,\ e6.e1 = e6,\ e6.e2 = e5,\ e6.e3 = -e8,\ e6.e4 = e7,\ e6.e5 = -e2,\ e6^2 = -e1,\ e6.e7 = -e4,\ e6.e8 = e3,\ e7.e1 = e7,\ e7.e2 = e8,\ e7.e3 = e5,\ e7.e4 = -e6,\ e7.e5 = -e3,\ e7.e6 = e4,\ e7^2 = -e1,\ e7.e8 = -e2,\ e8.e1 = e8,\ e8.e2 = -e7,\ e8.e3 = e6,\ e8.e4 = e5,\ e8.e5 = -e4,\ e8.e6 = -e3,\ e8.e7 = e2,\ e8^2 = -e1] \quad (1)$$

We use the command DGsetup to store these structure equations in memory.

```
> DGsetup(AD);
```

algebra name: O (2)

At this point one can now invoke many of the commands in the LieAlgebras package. For example, we can display the multiplication table for the basis elements:

```
O > MultiplicationTable(O);
```

		e1	e2	e3	e4	e5	e6	e7	e8
	----	----	----	----	----	----	----	----	----
e1	|	e1	e2	e3	e4	e5	e6	e7	e8
e2	|	e2	-e1	e4	-e3	e6	-e5	-e8	e7
e3	|	e3	-e4	-e1	e2	e7	e8	-e5	-e6
e4	|	e4	e3	-e2	-e1	e8	-e7	e6	-e5
e5	|	e5	-e6	-e7	-e8	-e1	e2	e3	e4
e6	|	e6	e5	-e8	e7	-e2	-e1	-e4	e3
e7	|	e7	e8	e5	-e6	-e3	e4	-e1	-e2
e8	|	e8	-e7	e6	e5	-e4	-e3	e2	-e1

(3)

We can also create a list of the basis elements.

```
O > Basis := DGinfo(O,"FrameBaseVectors");
```

$$Basis := [e1, e2, e3, e4, e5, e6, e7, e8] \quad \textbf{(4)}$$

We want to show that the formula $D_{a,b}(x) = [[a, b], x] - 3(a, b, x)$ allows us to generate a basis of the derivation algebra over the octonions. To do so, we first define procedures that will allow us to compute this. We begin by defining a map to compute the commutator.

```
O > Commutator := (x,y)->evalDG(x.y-y.x):
```

Next we create a function which will compute the associator of three arguments, (x,y,z)=(xy)z-x(yz). Note that we must use the command evalDG to specify the correct order of operations.

```
O > Associator:=(x,y,z)->evalDG( evalDG(x.y).z-x.evalDG(y.z) ):
```

Now we are ready to create a function for the derivation formula. We will want to include the condition that if the formula is zero, then we return a zero vector. The reason for this will become apparent shortly.

```
O > Der := proc(a,b,x)
      local obj;
      obj := evalDG( Commutator( Commutator(a,b) ,x)-3*Associator
    (a,b,x) ):
      if obj=0 then
        return DGzero("vector");
      else
        return obj;
      fi:
    end proc:
```

Here is an example of how this formula works using the basis elements *e3* and *e4* as the indices and *e5* as the argument.

```
O > Der(Basis[3],Basis[4],Basis[5]);
```

$$-2\,e6 \quad \textbf{(5)}$$

It will be much more convenient if we can represent this linear mapping as a matrix. The next few steps will illustrate how this can be accomplished. We can use the command GetComponents to write the result as a linear combination of the basis elements.

```
> GetComponents( Der(Basis[3],Basis[4],Basis[5]), Basis);
```

$$[0, 0, 0, 0, 0, -2, 0, 0] \tag{6}$$

This result will be the fifth column in the matrix representing this derivation. Using the map command, we can create a list of coefficients. Then, using the Matrix command, we convert the list of lists to a matrix representing the derivation.

```
> Matrix(map(x->GetComponents(Der(Basis[3],Basis[4],x),Basis),
  Basis));
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & -4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 \end{bmatrix} \tag{7}$$

We can use the map command to create yet another function; this one allowing us to specify the index elements for the map, and the result being the matrix representation of the derivation.

```
> MatrixDer:=(i,j)->Matrix(map(x->GetComponents(Der(Basis[i],
  Basis[j],x),Basis),  Basis)):
```

We can compare this with our previous result.

```
> MatrixDer(3,4);
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & -4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 \end{bmatrix} \tag{8}$$

Now we are ready to begin computing a spanning set for the derivation algebra. To do this, we need only cycle through the imaginary octonions. Furthermore, $D_{a,b}(x) = -D_{b,a}(x)$, so we don't need to check all possible combinations either. First, let's initialize a list for containg all the matrices we create.

```
> DerBasis:=[seq(0,ii=1..21)]:
```

Now we start making the matrices by cycling over the imaginary octonions.

```
> inc:=1:
  for ii from 2 to 8 do
    for jj from (ii+1) to 8 do
      DerBasis[inc]:=MatrixDer(ii,jj):
      inc:=inc+1:
    od:
  od:
```

However, this process may have created too many elements than is needed for a basis. We are only interested in the elements which are linearly independent. We can use the command DGbasis to create a linearly independent set.

```
> DerBasis := DGbasis( DerBasis );
```

$$DerBasis := \left[\begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&4&0&0&0&0&0\\ 0&-4&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&-2\\ 0&0&0&0&0&0&2&0\\ 0&0&0&0&0&-2&0&0\\ 0&0&0&0&2&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&4&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&-4&0&0&0&0&0&0\\ 0&0&0&0&0&0&2&0\\ 0&0&0&0&0&0&0&2\\ 0&0&0&0&-2&0&0&0\\ 0&0&0&0&0&-2&0&0 \end{bmatrix},\right.$$

$$\begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&4&0&0&0\\ 0&0&0&0&0&0&0&2\\ 0&0&0&0&0&0&-2&0\\ 0&-4&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&2&0&0&0&0\\ 0&0&-2&0&0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&4&0&0\\ 0&0&0&0&0&0&-2&0\\ 0&0&0&0&0&0&0&-2\\ 0&0&0&0&0&0&0&0\\ 0&-4&0&0&0&0&0&0\\ 0&0&2&0&0&0&0&0\\ 0&0&0&2&0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&4&0\\ 0&0&0&0&0&2&0&0\\ 0&0&0&0&2&0&0&0\\ 0&0&0&-2&0&0&0&0\\ 0&0&-2&0&0&0&0&0\\ 0&-4&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{bmatrix},$$

$$\begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&4\\ 0&0&0&0&-2&0&0&0\\ 0&0&0&0&0&2&0&0\\ 0&0&2&0&0&0&0&0\\ 0&0&0&-2&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&-4&0&0&0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&4&0&0&0&0\\ 0&0&-4&0&0&0&0&0\\ 0&0&0&0&0&-2&0&0\\ 0&0&0&0&2&0&0&0\\ 0&0&0&0&0&0&0&2\\ 0&0&0&0&0&0&-2&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&-2\\ 0&0&0&0&4&0&0&0\\ 0&0&0&0&0&2&0&0\\ 0&0&-4&0&0&0&0&0\\ 0&0&0&-2&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&2&0&0&0&0&0&0 \end{bmatrix},$$

$$\begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&2&0\\ 0&0&0&0&0&4&0&0\\ 0&0&0&0&-2&0&0&0\\ 0&0&0&2&0&0&0&0\\ 0&0&-4&0&0&0&0&0\\ 0&-2&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&-2&0&0\\ 0&0&0&0&0&0&4&0\\ 0&0&0&0&0&0&0&-2\\ 0&0&0&0&0&0&0&0\\ 0&2&0&0&0&0&0&0\\ 0&0&-4&0&0&0&0&0\\ 0&0&0&2&0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&2&0&0&0\\ 0&0&0&0&0&0&0&4\\ 0&0&0&0&0&0&2&0\\ 0&-2&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&-2&0&0&0&0\\ 0&0&-4&0&0&0&0&0 \end{bmatrix},$$

$$\left.\begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&-2&0&0&0&0\\ 0&0&2&0&0&0&0&0\\ 0&0&0&0&0&4&0&0\\ 0&0&0&0&-4&0&0&0\\ 0&0&0&0&0&0&0&2\\ 0&0&0&0&0&0&-2&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&2&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&-2&0&0&0&0&0&0\\ 0&0&0&0&0&0&4&0\\ 0&0&0&0&0&0&0&-2\\ 0&0&0&0&-4&0&0&0\\ 0&0&0&0&0&2&0&0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&-2&0&0&0&0&0\\ 0&2&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&4\\ 0&0&0&0&0&0&2&0\\ 0&0&0&0&0&-2&0&0\\ 0&0&0&0&-4&0&0&0 \end{bmatrix}\right] \quad \textbf{(9)}$$

With these matrices, we can use the command LieAlgebraData to create the algebra structure equations for the Lie algebra.

```
> LA := LieAlgebraData(DerBasis,g2):
> DGsetup(LA);
```

Lie algebra: g2 **(10)**

We can check a few properties to see if we have indeed created g_2. First we use the command DGinfo to return the dimension of the Lie algebra.

```
g2 > DGinfo("FrameBaseDimension");
```

14 **(11)**

Next we can identify the rank by computing a Cartan subalgebra. Notice that the number of elements in the list is the dimension of the Cartan subalgebra.

```
g2 > CSA := CartanSubalgebra(g2);
```

$$CSA := [e1, e2 - 2\,e13]$$ **(12)**

We can also use the command Query to dertermine if the Lie algebra is semisimple.

```
g2 > Query(g2,"Semisimple");
```

true **(13)**

Finally, to really ensure that we have created the exceptional Lie algebra g_2, we can compute the Cartan Matrix. To do this, we need a root space decomposition and a set of simple roots.

```
g2 > RSD := RootSpaceDecomposition(CSA):
g2 > PosRoots := PositiveRoots(RSD);
```

$$PosRoots := \left[\left[\begin{matrix} 2\,I \\ 6\,I \end{matrix}\right], \left[\begin{matrix} 2\,I \\ -6\,I \end{matrix}\right], \left[\begin{matrix} 6\,I \\ -6\,I \end{matrix}\right], \left[\begin{matrix} 4\,I \\ 0 \end{matrix}\right], \left[\begin{matrix} 0 \\ 12\,I \end{matrix}\right], \left[\begin{matrix} 6\,I \\ 6\,I \end{matrix}\right]\right]$$ **(14)**

```
g2 > SimRoots := SimpleRoots(PosRoots);
```

$$SimRoots := \left[\left[\begin{matrix} 2\,I \\ -6\,I \end{matrix}\right], \left[\begin{matrix} 0 \\ 12\,I \end{matrix}\right]\right]$$ **(15)**

```
g2 > CM := CartanMatrix(SimRoots, RSD);
```

$$CM := \begin{bmatrix} 2 & -1 \\ -3 & 2 \end{bmatrix}$$ **(16)**

Now, we can compute the Cartan matrix of g_2 directly by using the command CartanMatrix.

```
g2 > CartanMatrix("G",2);
```

$$\begin{bmatrix} 2 & -1 \\ -3 & 2 \end{bmatrix}$$ **(17)**

Therefore, since the Cartan matrices are equivalent, we know that the Lie algebra that we have made is in fact g_2.

Verifying the Magic Square Lie Algbras

Synopsis

• We will verify that the Lie algebras created using Vinberg's construction are the Lie algebras shown in the magic square.

Commands Illustrated

• LieAlgebras, DGsetup, LieAlgebraData, CartanMatrix, RootSpaceDecomposition, PositiveRoots, SimpleRoots, CartanMatrixToStandardForm

Load in the required packages.

```
> with(DifferentialGeometry): with(LieAlgebras):
```

The format for verifying the created Lie algebras will be to identify the Cartan matrix. This will be done by initializing the structure constants given by the procedure MagicSquare, then creating a Cartan subalgebra. We will then compute a root space decomposition and a set of simple roots. Using these, we can create a Cartan matrix. Then we will create the standard form Cartan matrix and compare the results.

Example 1: $M(\mathbb{R},\mathbb{R}) \cong \mathfrak{so}_3$.

Create the structure constants for the Lie algebra.

```
> LD := MagicSquare("Real","Real",A1);
```

$$LD := [[e1, e2] = -e3, [e1, e3] = e2, [e2, e3] = -e1] \quad (1)$$

Initialize the frame.

```
> DGsetup(LD);
```

Lie algebra: A1 (2)

Compute a Cartan subalgebra.

```
A1 > CSA := CartanSubalgebra(A1);
```

$$CSA := [e1] \quad (3)$$

Compute a root space decomposition.

```
A1 > RSD := RootSpaceDecomposition(CSA);
```

$$RSD := table([[I] = e2 + I\,e3, [-I] = e2 - I\,e3]) \quad (4)$$

Next compute a set of simple roots. Because the set of positive roots only contains a single element, it is also a simple root.

```
A1 > PR := PositiveRoots(RSD);
```

$$PR := [[\ I\ ]] \quad (5)$$

Now we can compute a Cartan matrix for this Lie algebra.

```
A1 > CM := CartanMatrix(PR,RSD);
```

$$CM := [\ 2\ ] \quad (6)$$

Here, it is useful to note that $sl_2(\mathbb{C})$ is isomorphic to $so_3(\mathbb{C})$. Thus we can create the standard form Cartan matrix for the Lie algebra A_1.

```
A1 > CartanMatrix("A",1);
```

$$[\ 2\ ] \quad (7)$$

Because the Cartan matrices are the same, we know that the two Lie algebras are isomorphic.

Example 2: $M(\mathbb{C},\mathbb{R}) \cong \mathfrak{su}_3$.

Create the structure constants for the Lie algebra.

```
> LD := MagicSquare("Complex","Real",A2):
```

Initialize the frame.

```
> DGsetup(LD);
```

Lie algebra: A2 **(8)**

Compute a Cartan subalgebra.

```
A2 > CSA := CartanSubalgebra(A2);
```

$$CSA := [e1,\ e7 + e8] \quad \textbf{(9)}$$

Compute a root space decomposition.

```
A2 > RSD := RootSpaceDecomposition(CSA):
```

Next compute a set of simple roots by first computing a set of positive roots.

```
A2 > PR := PositiveRoots(RSD);
```

$$PR := \left[\begin{bmatrix} \mathrm{I} \\ -3\,\mathrm{I} \end{bmatrix}, \begin{bmatrix} \mathrm{I} \\ 3\,\mathrm{I} \end{bmatrix}, \begin{bmatrix} 2\,\mathrm{I} \\ 0 \end{bmatrix}\right] \quad \textbf{(10)}$$

```
A2 > SR := SimpleRoots(PR);
```

$$SR := \left[\begin{bmatrix} \mathrm{I} \\ -3\,\mathrm{I} \end{bmatrix}, \begin{bmatrix} \mathrm{I} \\ 3\,\mathrm{I} \end{bmatrix}\right] \quad \textbf{(11)}$$

Now we can compute a Cartan matrix for this Lie algebra.

```
A2 > CM := CartanMatrix(SR,RSD);
```

$$CM := \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \quad \textbf{(12)}$$

Here, it is useful to note that $A_2 = sl_3(\mathbb{C})$ is isomorphic to $su_3(\mathbb{C})$. Thus we can create the standard form Cartan matrix for the Lie algebra A_2.

```
A2 > CartanMatrix("A",2);
```

$$\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \quad \textbf{(13)}$$

Because the Cartan matrices are the same, we know that the two Lie algebras are isomorphic.

Example 3: $M(\mathbb{H},\mathbb{R}) \cong \mathfrak{sp}_6$.

Create the structure constants for the Lie algebra. The result will be suppressed.

```
> LD := MagicSquare("Quaternion","Real",C3):
```

Initialize the frame.

```
> DGsetup(LD);
```

Lie algebra: C3 **(14)**

Compute a Cartan subalgebra.

```
C3 > CSA := CartanSubalgebra(C3);
```

$$CSA := [e1,\ e4,\ e16 + e19] \quad \textbf{(15)}$$

Compute a root space decomposition. Again, the output will be suppressed.

```
C3 > RSD := RootSpaceDecomposition(CSA):
```

Next compute a set of simple roots by first computing a set of positive roots.

```
C3 > PR := PositiveRoots(RSD);
```

$$PR := \left[\begin{bmatrix} 4\,I \\ 0 \\ -2\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 2\,I \\ 0 \end{bmatrix}, \begin{bmatrix} 4\,I \\ -I \\ I \end{bmatrix}, \begin{bmatrix} 4\,I \\ 2\,I \\ -2\,I \end{bmatrix}, \begin{bmatrix} 4\,I \\ I \\ I \end{bmatrix}, \begin{bmatrix} 4\,I \\ -2\,I \\ -2\,I \end{bmatrix}, \begin{bmatrix} 0 \\ I \\ -3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ I \\ 3\,I \end{bmatrix}, \begin{bmatrix} 4\,I \\ 0 \\ 4\,I \end{bmatrix}\right] \quad \textbf{(16)}$$

```
C3 > SR := SimpleRoots(PR);
```

$$SR := \left[\begin{bmatrix} 4\,I \\ -2\,I \\ -2\,I \end{bmatrix}, \begin{bmatrix} 0 \\ I \\ -3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ I \\ 3\,I \end{bmatrix}\right] \quad \textbf{(17)}$$

Now we can compute a Cartan matrix for this Lie algebra.

```
C3 > CM := CartanMatrix(SR,RSD);
```

$$CM := \begin{bmatrix} 2 & 0 & -2 \\ 0 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \quad \textbf{(18)}$$

We will need to put this Cartan Matrix in standard form.

```
C3 > CartanMatrixToStandardForm(CM);
```

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -2 & 2 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \text{"C"} \quad \textbf{(19)}$$

Now we can create the standard form Cartan matrix for the Lie algebra C_3.

```
C3 > CartanMatrix("C",3);
```

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -2 & 2 \end{bmatrix} \quad \textbf{(20)}$$

Because the standard form of the Cartan matrices are the same, we know that the two Lie algebras are isomorphic.

Example 4: $M(\mathbb{O},\mathbb{R}) \cong \mathfrak{f}_4$.

Create the structure constants for the Lie algebra. The result will be suppressed.

```
> LD := MagicSquare("Octonion","Real",F4):
```

Initialize the frame.

```
> DGsetup(LD);
```

Lie algebra: F4 **(21)**

Compute a Cartan subalgebra.

```
F4 > CSA := CartanSubalgebra(F4);
```

$$CSA := \left[e1,\ e2 + \frac{1}{2}\, e4,\ e15,\ e39 + e46 \right] \quad \textbf{(22)}$$

Compute a root space decomposition. Again, the output will be suppressed.

```
F4 > RSD := RootSpaceDecomposition(CSA):
```

Next compute a set of simple roots by first computing a set of positive roots.

```
F4 > PR := PositiveRoots(RSD):
```

Now we compute the set of simple roots.

```
F4 > SR := SimpleRoots(PR);
```

$$SR := \left[\begin{bmatrix} 0 \\ 0 \\ I \\ -3\,I \end{bmatrix}, \begin{bmatrix} 2\,I \\ -3\,I \\ -2\,I \\ 2\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 6\,I \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ 3\,I \end{bmatrix} \right] \quad \textbf{(23)}$$

Now we can compute a Cartan matrix for this Lie algebra.

```
F4 > CM := CartanMatrix(SR,RSD);
```

$$CM := \begin{bmatrix} 2 & -1 & 0 & -1 \\ -2 & 2 & -1 & 0 \\ 0 & -1 & 2 & 0 \\ -1 & 0 & 0 & 2 \end{bmatrix} \quad \textbf{(24)}$$

We will need to put this Cartan Matrix in standard form.

```
F4 > CartanMatrixToStandardForm(CM);
```

$$\left[\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -2 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \text{"F"} \right] \quad \textbf{(25)}$$

Now we can create the standard form Cartan matrix for the Lie algebra $\mathfrak{f}_4$.

```
F4 > CartanMatrix("F",4);
```

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -2 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \quad \textbf{(26)}$$

Because the standard form of the Cartan matrices are the same, we know that the two Lie algebras are isomorphic.

Example 5: $M(\mathbb{C},\mathbb{C}) \cong \mathfrak{su}3 \oplus \mathfrak{su}3$.

Create the structure constants for the Lie algebra. The result will be suppressed.

```
> LD := MagicSquare("Complex","Complex",alg):
```

Initialize the frame.

```
> DGsetup(LD);
```

Lie algebra: alg **(27)**

Compute a Cartan subalgebra.

```
alg > CSA := CartanSubalgebra(alg);
```

$$CSA := [e1,\ e2,\ e13 + e15,\ e14 + e16] \quad \textbf{(28)}$$

Compute a root space decomposition. Again, the output will be suppressed.

```
alg > RSD := RootSpaceDecomposition(CSA):
```

Next compute a set of simple roots by first computing a set of positive roots.

```
alg > PR := PositiveRoots(RSD):
```

Now we compute the set of simple roots.

```
alg > SR := SimpleRoots(PR);
```

$$SR := \left[\begin{bmatrix} I \\ I \\ -3\,I \\ 3\,I \end{bmatrix}, \begin{bmatrix} I \\ -I \\ 3\,I \\ 3\,I \end{bmatrix}, \begin{bmatrix} I \\ -I \\ -3\,I \\ -3\,I \end{bmatrix}, \begin{bmatrix} I \\ I \\ 3\,I \\ -3\,I \end{bmatrix}\right] \quad \textbf{(29)}$$

Now we can compute a Cartan matrix for this Lie algebra.

```
alg > CM := CartanMatrix(SR,RSD);
```

$$CM := \begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 2 & -1 & 0 \\ 0 & -1 & 2 & 0 \\ -1 & 0 & 0 & 2 \end{bmatrix} \quad \textbf{(30)}$$

In fact, this matrix is not a Cartan matrix, but is a direct sum of Cartan matrices. This can be seen using the transformation matrix given below:

```
alg > P:=Matrix(4,4,{(1,3)=1,(2,2)=1,(3,1)=1,(4,4)=1});
```

$$P := \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(31)}$$

```
alg > P.CM.P^(-1);
```

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & 0 & 0 \\ 0 & 0 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \quad \textbf{(32)}$$

Now we can create the standard form Cartan matrix for the Lie algebra $\mathfrak{su}_3$.

```
alg > CartanMatrix("A",2);
```

$$\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \tag{33}$$

By inspection of the Cartan matrices, it is clear that the algebra created by the magic square is isomorphic to $\mathfrak{su}_3 \oplus \mathfrak{su}_3$.

Example 6: $M(\mathbb{H},\mathbb{C}) \cong \mathfrak{su}_6$.

Create the structure constants for the Lie algebra. The result will be suppressed.

```
> LD := MagicSquare("Quaternion","Complex",A5):
```

Initialize the frame.

```
> DGsetup(LD);
```

Lie algebra: A5 (34)

Compute a Cartan subalgebra.

```
A5 > CSA := CartanSubalgebra(A5);
```

$$CSA := [e1,\ e4,\ e6,\ e28 + e32,\ e30 + e34] \tag{35}$$

Compute a root space decomposition. Again, the output will be suppressed.

```
A5 > RSD := RootSpaceDecomposition(CSA):
```

Next compute a set of simple roots by first computing a set of positive roots.

```
A5 > PR := PositiveRoots(RSD):
```

Now we compute the set of simple roots.

```
A5 > SR := SimpleRoots(PR);
```

$$SR := \left[\begin{bmatrix} 0 \\ I \\ -I \\ 3\,I \\ 3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ I \\ I \\ 3\,I \\ -3\,I \end{bmatrix}, \begin{bmatrix} 4\,I \\ -2\,I \\ 0 \\ 0 \\ 2\,I \end{bmatrix}, \begin{bmatrix} 0 \\ I \\ I \\ -3\,I \\ 3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ I \\ -I \\ -3\,I \\ -3\,I \end{bmatrix}\right] \tag{36}$$

Now we can compute a Cartan matrix for this Lie algebra.

```
A5 > CM := CartanMatrix(SR,RSD);
```

$$CM := \begin{bmatrix} 2 & 0 & 0 & 0 & -1 \\ 0 & 2 & -1 & -1 & 0 \\ 0 & -1 & 2 & 0 & -1 \\ 0 & -1 & 0 & 2 & 0 \\ -1 & 0 & -1 & 0 & 2 \end{bmatrix} \tag{37}$$

We will need to put this Cartan Matrix in standard form.

```
A5 > CartanMatrixToStandardForm(CM);
```

$$\left[\begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \text{"A"}\right] \tag{38}$$

Now we can create the standard form Cartan matrix for the Lie algebra A_5.

```
A5 > CartanMatrix("A",5);
```

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \tag{39}$$

Because the standard form of the Cartan matrices are the same, we know that the two Lie algebras are isomorphic.

Example 7: $M(\mathbb{O},\mathbb{C}) \cong e_6$.

Create the structure constants for the Lie algebra. The result will be suppressed.

```
> LD := MagicSquare("Octonion","Complex",E6):
```

Initialize the frame.

```
> DGsetup(LD);
```

Lie algebra: E6 **(40)**

Compute a Cartan subalgebra.

```
E6 > CSA := CartanSubalgebra(E6);
```

$$CSA := [e1, e2 + 2\,e4, e15, e19, e63 + e71, e67 + e75] \tag{41}$$

Compute a root space decomposition. Again, the output will be suppressed.

```
E6 > RSD := RootSpaceDecomposition(CSA):
```

Next compute a set of simple roots by first computing a set of positive roots.

```
E6 > PR := PositiveRoots(RSD):
```

Now we compute the set of simple roots.

```
E6 > SR := SimpleRoots(PR);
```

$$SR := \left[\begin{bmatrix} 0 \\ 0 \\ I \\ I \\ -3\,I \\ 3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 12\,I \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 2\,I \\ -6\,I \\ -2\,I \\ 0 \\ 0 \\ -2\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ I \\ 3\,I \\ -3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ -I \\ -3\,I \\ -3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ -I \\ 3\,I \\ 3\,I \end{bmatrix}\right] \tag{42}$$

Now we can compute a Cartan matrix for this Lie algebra.

```
E6 > CM := CartanMatrix(SR,RSD);
```

$$CM := \begin{bmatrix} 2 & 0 & -1 & -1 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & -1 \\ -1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -1 \\ 0 & 0 & -1 & 0 & -1 & 2 \end{bmatrix} \quad \textbf{(43)}$$

We will need to put this Cartan Matrix in standard form.

```
E6 > CartanMatrixToStandardForm(CM);
```

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & -1 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & 0 \\ 0 & 0 & -1 & 0 & 0 & 2 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \text{"E"} \quad \textbf{(44)}$$

Now we can create the standard form Cartan matrix for the Lie algebra e_6.

```
E6 > CartanMatrix("E",6);
```

$$\begin{bmatrix} 2 & 0 & -1 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 0 & 0 \\ -1 & 0 & 2 & -1 & 0 & 0 \\ 0 & -1 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix} \quad \textbf{(45)}$$

Because the standard form of the Cartan matrices are the same, we know that the two Lie algebras are isomorphic.

Example 8: $M(\mathbb{H},\mathbb{H}) \cong \mathfrak{so}_{12}$.

Create the structure constants for the Lie algebra. The result will be suppressed.

```
> LD := MagicSquare("Quaternion","Quaternion",D6):
```

Initialize the frame.

```
> DGsetup(LD);
```

Lie algebra: D6 **(46)**

Compute a Cartan subalgebra.

```
D6 > CSA := CartanSubalgebra(D6);
```

$$CSA := [e1, e4, e7, e12, e56 + e62, e59 + e65] \quad \textbf{(47)}$$

Compute a root space decomposition. Again, the output will be suppressed.

```
D6 > RSD := RootSpaceDecomposition(CSA):
```

Next compute a set of simple roots by first computing a set of positive roots.

```
D6 > PR := PositiveRoots(RSD):
```

Now we compute the set of simple roots.

```
D6 > SR := SimpleRoots(PR);
```

$$SR := \left[\begin{bmatrix} 0 \\ 0 \\ I \\ I \\ -3\,I \\ 3\,I \end{bmatrix}, \begin{bmatrix} 4\,I \\ -4\,I \\ -I \\ -I \\ I \\ -I \end{bmatrix}, \begin{bmatrix} 0 \\ 4\,I \\ -2\,I \\ 0 \\ 2\,I \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ I \\ 3\,I \\ -3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ -I \\ -3\,I \\ -3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ I \\ -I \\ 3\,I \\ 3\,I \end{bmatrix}\right] \tag{48}$$

Now we can compute a Cartan matrix for this Lie algebra.

```
D6 > CM := CartanMatrix(SR,RSD);
```

$$CM := \begin{bmatrix} 2 & -1 & -1 & -1 & 0 & 0 \\ -1 & 2 & 0 & 0 & 0 & 0 \\ -1 & 0 & 2 & 0 & -1 & 0 \\ -1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix} \tag{49}$$

We will need to put this Cartan Matrix in standard form.

```
D6 > CartanMatrixToStandardForm(CM);
```

$$\left[\begin{bmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & 0 \\ 0 & 0 & 0 & -1 & 0 & 2 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \text{"D"}\right] \tag{50}$$

Now we can create the standard form Cartan matrix for the Lie algebra D_6.

```
D6 > CartanMatrix("D",6);
```

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & 0 \\ 0 & 0 & 0 & -1 & 0 & 2 \end{bmatrix} \tag{51}$$

Because the standard form of the Cartan matrices are the same, we know that the two Lie algebras are isomorphic.

Example 9: $M(\mathbb{O},\mathbb{H}) \cong e_7$.

Create the structure constants for the Lie algebra. The result will be suppressed.

```
> LD := MagicSquare("Octonion","Quaternion",E7):
```

Initialize the frame.

```
> DGsetup(LD);
```

Lie algebra: E7 **(52)**

Compute a Cartan subalgebra.

```
E7 > CSA := CartanSubalgebra(E7);
```

$$CSA := \left[e1,\ e6 + \frac{1}{2}\,e10,\ e15,\ e18,\ e30,\ e116 + e126,\ e120 + e130\right] \quad \textbf{(53)}$$

Compute a root space decomposition. Again, the output will be suppressed.

```
E7 > RSD := RootSpaceDecomposition(CSA):
```

Next compute a set of simple roots by first computing a set of positive roots.

```
E7 > PR := PositiveRoots(RSD):
```

Now we compute the set of simple roots.

```
E7 > SR := SimpleRoots(PR);
```

$$SR := \left[\begin{bmatrix}0\\0\\0\\I\\-I\\3\,I\\3\,I\end{bmatrix},\begin{bmatrix}0\\0\\0\\I\\I\\-3\,I\\3\,I\end{bmatrix},\begin{bmatrix}0\\0\\4\,I\\-2\,I\\0\\-2\,I\\0\end{bmatrix},\begin{bmatrix}2\,I\\-3\,I\\-4\,I\\-I\\-I\\-I\\I\end{bmatrix},\begin{bmatrix}0\\0\\0\\I\\I\\3\,I\\-3\,I\end{bmatrix},\begin{bmatrix}0\\6\,I\\0\\0\\0\\0\\0\end{bmatrix},\begin{bmatrix}0\\0\\0\\I\\-I\\-3\,I\\-3\,I\end{bmatrix}\right] \quad \textbf{(54)}$$

Now we can compute a Cartan matrix for this Lie algebra.

```
E7 > CM := CartanMatrix(SR,RSD);
```

$$CM := \begin{bmatrix}2&0&-1&0&0&0&-1\\0&2&0&0&-1&0&0\\-1&0&2&0&-1&0&0\\0&0&0&2&-1&-1&0\\0&-1&-1&-1&2&0&0\\0&0&0&-1&0&2&0\\-1&0&0&0&0&0&2\end{bmatrix} \quad \textbf{(55)}$$

We will need to put this Cartan Matrix in standard form.

```
E7 > CartanMatrixToStandardForm(CM);
```

$$\left[\begin{bmatrix}2&-1&0&0&0&0&0\\-1&2&-1&0&0&0&0\\0&-1&2&-1&0&0&-1\\0&0&-1&2&-1&0&0\\0&0&0&-1&2&-1&0\\0&0&0&0&-1&2&0\\0&0&-1&0&0&0&2\end{bmatrix},\begin{bmatrix}0&0&0&0&1&0&0\\0&0&0&0&0&0&1\\0&0&0&1&0&0&0\\0&1&0&0&0&0&0\\0&0&1&0&0&0&0\\1&0&0&0&0&0&0\\0&0&0&0&0&1&0\end{bmatrix},\text{"E"}\right] \quad \textbf{(56)}$$

Now we can create the standard form Cartan matrix for the Lie algebra $\mathfrak{e}_7$.

```
E7 > CartanMatrix("E",7);
```

$$\begin{bmatrix} 2 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix} \quad \textbf{(57)}$$

Because the standard form of the Cartan matrices are the same, we know that the two Lie algebras are isomorphic.

Example 10: $M(\mathbb{O},\mathbb{O}) \cong \mathfrak{e}_8$.

Create the structure constants for the Lie algebra. The result will be suppressed.

```
> LD := MagicSquare("Octonion","Octonion",E8):
```

Initialize the frame.

```
> DGsetup(LD);
```

Lie algebra: E8 **(58)**

Compute a Cartan subalgebra.

```
E8 > CSA := CartanSubalgebra(E8);
```

$$CSA := \left[e1,\ e2 - \frac{1}{2}\, e14,\ e15,\ e16 - \frac{1}{2}\, e28,\ e29,\ e46,\ e223 + e237,\ e230 + e244 \right] \quad \textbf{(59)}$$

Compute a root space decomposition. Again, the output will be suppressed.

```
E8 > RSD := RootSpaceDecomposition(CSA):
```

Next compute a set of simple roots by first computing a set of positive roots.

```
E8 > PR := PositiveRoots(RSD):
```

Now we compute the set of simple roots.

```
E8 > SR := SimpleRoots(PR);
```

$$SR := \left[\begin{bmatrix} 0 \\ 0 \\ 0 \\ 6\,I \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 6\,I \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ I \\ -I \\ -3\,I \\ -3\,I \end{bmatrix}, \begin{bmatrix} 2\,I \\ -3\,I \\ -4\,I \\ 0 \\ -I \\ I \\ I \\ I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ I \\ -I \\ 3\,I \\ 3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ I \\ I \\ 3\,I \\ -3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ I \\ I \\ -3\,I \\ 3\,I \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 2\,I \\ -3\,I \\ -2\,I \\ 0 \\ -2\,I \\ 0 \end{bmatrix} \right] \quad \textbf{(60)}$$

Now we can compute a Cartan matrix for this Lie algebra.

```
E8 > CM := CartanMatrix(SR,RSD);
```

$$CM := \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 2 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -1 & -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 2 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 2 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & 0 \\ -1 & 0 & 0 & 0 & -1 & -1 & 0 & 2 \end{bmatrix} \quad \textbf{(61)}$$

If we try to put this in standard form, the program will reorder the simple roots, but it can't identify the root type.

```
E8 > CM,PM,RType := CartanMatrixToStandardForm(CM);
```

$$CM, PM, RType := \begin{bmatrix} 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 2 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, "" \quad \textbf{(62)}$$

However, the observant reader will notice that this Cartan matrix coincides with the following Dynkin diagram for e_8.

```
E8 > DynkinDiagram("E",8,version=2);
```