Filomat 39:35 (2025), 12461–12483
https://doi.org/10.2298/FIL2535461A

Published by Faculty of Sciences and Mathematics,
University of Niš, Serbia
Available at: http://www.pmf.ni.ac.rs/filomat

Groups of Lie type in C++

Yousif Alanazi[a,*], **Anton Betten**[a]

[a]*Department of Mathematics, College of Science, Kuwait University, Sabah Al Salem University City, P.O. Box 5969, Safat 13060, Shadadiya, Kuwait*

Abstract. We are looking at groups of Lie type from the point of view of efficient algorithms and efficient data structures. We compute the automorphism group of a Lie group and convert the representation of elements from matrices to words in the generators and back. Our work is based on our own computer algebra system Orbiter, which is a C++ class library with a small front end. The language C++ encourages abstraction and object oriented programming, which are well-suited for mathematical data types. We illustrate this by showing how orbit algorithms for very different groups can be run through the same software interface.

1. Introduction

Working with groups of Lie type offers many computational challenges, which can benefit from computer algebra systems. Many software projects focus on the representations of these groups. Some of which are Lie [45], the GAP package LIE [19],[21], atlasofliegroups [1], and Chevie [30]. Problems that require computing orbits of Lie groups require a somewhat different approach, focusing more on the group elements themselves. For instance, orbits on sets or orbits on subspaces require computing the orbits of a group on a partially ordered set ("poset"), such as the set of subsets or the set of subspaces. In these, we assume that the group G acts on the poset, and we compute orbits by using the G-invariant relation between different levels in the poset. Many problems in combinatorics relate to this, as well as problems in finite geometry. The vector spaces often carry a form, and so groups of Lie type arise naturally in this setting. In previous work, we discussed the classification of geometric objects in projective spaces such as BLT-sets [5], cubic surfaces [11], packings [6], unitals in projective planes [4] and others. In each case, the computational bottleneck is the computation of orbits of a certain group of Lie type. For this reason, we wish to discuss some of the relevant techniques and tools that are relevant. The goal is to build a universal orbit algorithm that can work with different groups and different actions, in particular for orbits on partially ordered sets.

We propose an approach in which the representation and the permutation action are tightly coupled. A group element may be stored in the representation, but the action may be on a permutation action. The permutation action may be the natural action associated with the vector space, or with the associated

2020 *Mathematics Subject Classification.* Primary 20G40; Secondary 20G05, 11T06, 68W30.
Keywords. Group of Lie Type, Finite Field, Dynkin Diagram, Classical Groups, Computer Algebra, C++
Received: 09 July 2025; Accepted: 22 September 2025
Communicated by Dragana Cvetković-Ilić
* Corresponding author: Yousif Alanazi
Email addresses: yousif.alanazi@ku.edu.kw (Yousif Alanazi), anton.betten@ku.edu.kw (Anton Betten)
ORCID iDs: https://orcid.org/0009-0009-6156-5682 (Yousif Alanazi), https://orcid.org/0000-0003-0194-0777 (Anton Betten)

projective space. But it is possible that the permutation action is derived in some way from this "basic" action. For instance, we may have an action on the exterior square of the underlying vector space. The matrix representation offers a compact way of storing the group element (provided the dimension is low). The permutation representation offers the ability to store the group in the form of a stabilizer chain. Thus, both representations are needed, and it is very important to be able to choose the best representation for the problem at hand. We are suggesting to piggy-back the permutation representation on top of the matrix representation. The present paper will highlight Orbiter [8] and its functionality for groups of Lie type. Related problems are that of creating groups of Lie type in the first place. To do so, we need to construct a base and stabilizer chain for a suitable permutation action. This way, we can avoid storing long arrays of images associated with permutations. Whenever we need to evaluate an image, we compute the image from scratch using the matrix representation.

While it would be possible to solve these problems in GAP [28] and Magma [13], writing code in C++ offers an advantage in efficiency. We are not trying to replace these general-purpose systems, rather we are offering highly efficient solutions to orbit problems that go beyond what a general-purpose system can offer. We already touched upon the problem of computing orbits in previous work [7].

Another computational approach to groups of Lie type is [17]. Further approaches to the representatione of Lie algebras are the Mathematica package called LieART [26] as well as the software RealLie [12]. There are probably many other software packages that should be mentioned here. Another approach to Computational Group Theory in C++ is PermLib [38], see also [39].

For background on finite simple groups, see [46] or [32]. For an account of the geometry of the classical groups, see [44]. For computational aspects in the theory of linear algebraic groups, see [22]. For background on representation theory, see [27]. For a background on computational group theory, see [34] or [41] or [14]. For background on Lie theory, with an emphasis on computational methods, see [20].

2. Algebra and Geometry

We start with some definitions and notation. Let G be a finite group with identity element e. An element $g \in G$, $g \neq e$, is called an involution if $g^2 = e$. In general, the order of an element $g \in G$ is the least positive integer h such that $g^h = e$. We denote the order of g as $|g|$. So, an involution is an element g with $|g| = 2$. If g and h are two elements of a group G, we can form a new element $h^{-1}gh$ which is called the h-conjugate of g. The set $\{h^{-1}gh \mid h \in G\}$ is called the conjugacy class of g in G. The group G is partitioned into conjugacy classes. The elements of a given class share many properties. In particular, they have the same order. For this reason, looking at the conjugacy classes of a group is a good way to get to know a given group. The class of g is the orbit of g under the action of G on itself by conjugation. We use ATLAS [18] notation for conjugacy classes. So, we will be using the order of elements in the class and a capital letter to distinguish between classes of elements of the same order. For instance, the classes of involutions are denoted 2A, 2B, 2C and so on.

The orthogonal groups are the stabilizers of nondegenerate quadrics. They come in two versions, as they can be considered in affine and in projective space. Here, we consider these quadrics over finite fields only. Over a finite field, there are two different types of quadrics if the number of variables is even. Accordingly, there are two different types of orthogonal groups in even dimension. After a possible change of basis, one can assume a quadratic form to be in a canonical form. Once the quadratic form is chosen, the associated bilinear form is fixed. There are some complications in even characteristic. Namely, in odd characteristic the bilinear form determines the quadratic form. This is not true in characteristic two.

Let $V = V(m, q) \simeq \mathbb{F}_q^m$ be the m-dimensional vector space over the field $\mathbb{F}_q$. Let $\mathrm{PG}(m-1, q)$ be the associated projective space. Its elements are the subspaces of V, apart from the zero subspace, ordered with respect to inclusion. Let $\mathbf{x} = (X_0, \ldots, X_{m-1})$ be a coordinate vector, and let $P(\mathbf{x})$ be the corresponding projective point. Note that the coordinate vector of a point in $\mathrm{PG}(n, q)$ has $n + 1$ entries.

Let $Q(\mathbf{x})$ be a quadratic form, i.e. a mapping $V \to \mathbb{F}_q$ with

$$Q(\lambda\mathbf{x}) = \lambda^2 Q(\mathbf{x}), \text{ for all } \mathbf{x} \in V, \ \lambda \in \mathbb{F}_q.$$

The associated bilinear form is

$$\beta(\mathbf{x}, \mathbf{y}) := Q(\mathbf{x} + \mathbf{y}) - Q(\mathbf{x}) - Q(\mathbf{y}).$$

By definition,

$$\beta(\mathbf{x}, \mathbf{x}) = 2Q(\mathbf{x}).$$

This means that β defines Q, unless the characteristic of the field is even. In a finite dimensional vector space, and after choosing a basis, we can define the form matrix of a bilinear form. Using row vector convention, the form matrix of β is the matrix B such that

$$\beta(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot B \cdot \mathbf{y}^\top \tag{1}$$

for any two vectors $\mathbf{x}, \mathbf{y}$.

A point $\mathbf{P}(\mathbf{x})$ with $Q(\mathbf{x}) = 0$ is called singular (otherwise non-singular). We will also say that a singular point lies on the quadric. Likewise, a non-singular point is a point off the quadric. A subspace U consisting exclusively of singular points is called totally singular. The bilinear form allows to introduce a notion of perpendicularity. For a point $\mathbf{P}(\mathbf{x})$, the perp is defined as

$$\mathbf{P}(\mathbf{x})^\perp = \{\mathbf{P}(\mathbf{y}) \mid \beta(\mathbf{x}, \mathbf{y}) = 0\}.$$

More generally, for a subspace U we put

$$U^\perp = \{\mathbf{P}(\mathbf{y}) \mid \beta(\mathbf{x}, \mathbf{y}) = 0 \text{ for all } \mathbf{x} \in U\}.$$

The set $V^\perp$ is of interest when it comes to determine whether Q is non-degenerate. Namely, we say that Q is non-degenerate if $V^\perp$ contains no singular vector. When the characteristic is not two, this corresponds to β being non-degenerate, i.e. $V^\perp = 0$.

For n even, there are two types of nondegenerate quadratic forms. They are called hyperbolic (or plus type) and elliptic (or minus type). The canonical form of a nondegenerate quadric of plus type in $\mathrm{PG}(n-1, q)$ may be chosen to be

$$X_0X_1 + X_2X_3 + \cdots + X_{n-2}X_{n-1} = 0$$

For n odd, there is only one type called parabolic. We write $\mathcal{Q}^\epsilon(n, q)$ for the nondegenerate quadric in $\mathrm{PG}(n, q)$. Here $\epsilon = 1, -1, 0$ indicates a plus, minus, and parabolic type, respectively.

Next, we discuss the groups. The orthogonal group associated with V and Q is

$$O(V, Q) := \{f \in \mathrm{GL}(n, q) \mid Q(f(\mathbf{x})) = Q(\mathbf{x}) \text{ for all } \mathbf{x} \in V\}.$$

The full orthogonal group, denoted $\Gamma\mathrm{O}(V, Q)$ consists of all σ-semilinear transformations f such that

$$Q(f(\mathbf{x})) = a\sigma(Q(\mathbf{x})) \text{ for all } \mathbf{x} \in V$$

for some $a \in \mathbb{F}_q \setminus \{0\}$. The general orthogonal group is

$$\mathrm{GO}(V, Q) := \Gamma\mathrm{O}(V, Q) \cap \mathrm{GL}(V).$$

If Q in nondegenerate on V, we indicate the type of the group using a triple (ϵ, n, q). ϵ is in $\{0, \pm 1\}$, and indicates the type of quadratic form 0 means parabolic, $+1$ is hyperbolic, and -1 is elliptic. The parameters n and q are the dimension of the vector space and the order of the field. So, for instance, $\mathrm{P\Gamma O}^{+}(n, q)$ is the orthogonal group on a nondegenerate form in n variables of hyperbolic type and n is even.

The group acts on the equation by (semi-)linear substitution of the variables. The collineation group $\mathrm{P\Gamma O}^{+}(n, q)$ contains as a normal subgroup the projective stabilizer $\mathrm{PGO}^{+}(n, q)$. If q is prime, the two groups coincide. Otherwise, the factor group is isomorphic to the group of field automorphisms of $\mathbb{F}_q$. If $q = p^e$, then $\mathrm{Aut}(\mathbb{F}_q)$ is a cyclic group of order e, generated by the Frobenius automorphism which sends x to x^p. The Frobenius automorphism induces an automorphism of the projective space, namely the map which sends $\mathbf{P}(x_0, \ldots, x_n)$ to $\mathbf{P}(x_0^p, \ldots, x_n^p)$. This map is called the automorphic collineation. A collineation is an automorphism of the underlying projective space. The automorphism group of the projective space is called the collineation group.

We let $\mathcal{Q}^{\epsilon}(n-1, q)$ be the orthogonal geometry defined by a non-degenerate quadratic form in $\mathrm{PG}(n-1, q)$. We let $\mathrm{P\Gamma L}^{\epsilon}(n, q)$ be the associated collineation group. This is the stabilizer of the quadratic form in the collineation group $\mathrm{P\Gamma L}(n, q)$.

Next, we introduce group elements that will be useful to generate these groups. We start with orthogonal reflections and then move on to Siegel transformations.

To every point $P(\mathbf{z})$ not on the quadric, we associate an orthogonal reflection $\sigma_{\mathbf{z}}$ by putting

$$\sigma_{\mathbf{z}}(\mathbf{x}) = \mathbf{x} - \frac{\beta(\mathbf{z}, \mathbf{x})}{Q(\mathbf{z})}\mathbf{z}. \tag{2}$$

Following [44, Corollary 11.42], we know that the orthogonal reflections generate the orthogonal group $O^{\epsilon}(n, q)$ (with the exception of $\mathrm{GO}^{+}(4, 2)$). By taking quotients modulo the center, we find that the same is true for the projective group.

Let $\mathbf{u}$ be a singular vector and $\mathbf{v} \in \langle \mathbf{u} \rangle^{\perp}$. The Siegel transformation associated with $\mathbf{u}$ and $\mathbf{v}$ is

$$\rho_{\mathbf{u},\mathbf{v}}(\mathbf{x}) := \mathbf{x} + \beta(\mathbf{x}, \mathbf{v})\mathbf{u} - \beta(\mathbf{x}, \mathbf{u})\mathbf{v} - Q(\mathbf{v})\beta(\mathbf{x}, \mathbf{u})\mathbf{u},$$

see [44, Equation (11.17)].

3. Groups of Lie Type

What is a group of Lie type? Chevalley in [16] constructed a special basis for all the complex simple Lie algebras. Using this basis, the corresponding algebraic groups can be defined over the integers. Using the reduction modulo a prime number p, this gives finite groups of Lie type in a unified context. While some of these groups were known before, several new ones were found specifically in this context. These groups play an important role in the classification of finite simple groups. In fact, it is now known (using the classification of finite simple groups) that every finite simple group is a group of Lie type, except for the cyclic groups of prime order, the alternating groups, the Tits group, and the 26 sporadic simple groups. The groups of Lie type are also known as the Chevalley groups. For a survey, see [15].

Dynkin diagrams are a tool to classify finite dimensional simple Lie algebras. They characterize root systems, which are sets of vectors in Euclidean space admitting only angles of 90 and 120 (and 180) degrees between any two. The root system comes with a group of Euclidean reflections, and these reflections generate a group that is called the the Weyl group. Groups generated by reflections have then been studied by Coxeter, and the Dynkin diagrams describe the finite Coxeter groups. When we study Lie groups over finite field, we no longer have an Euclidean reflection group. However, we have some finite field analogues, such as points off quadrics, as we shall see. Dynkin diagrams are named after [23] and [24] (the former is only available in Russian). According to the AMS MathSciNet review by Freudenthal, the types have already been known to Cartan.

The groups of Lie type fall into several types or families, some finite and some infinite. These types are parametrized with the Dynkin diagrams, which have names such an A_n, B_n, C_n and so forth. The types A_n, B_n, C_n and D_n, are infinite families and correspond to the classical groups. The classical groups are the groups of matrices that arise from the general linear group when a form (bilinear, alternating or hermitian) is preserved. The exceptional types are E_6, E_7, E_8, F_4, and G_2. Further groups of Lie type (twisted type) were found by Steinberg and Ree. For more details, we refer to [46].

Figure 1 shows the relevant types of Dynkin diagrams.

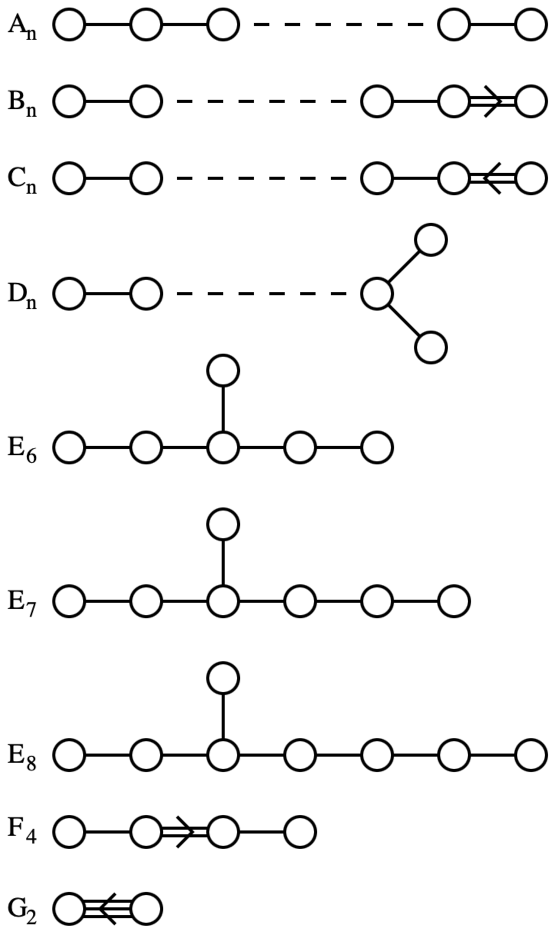

Figure 1: Dynkin Diagrams. The subscript is the number of nodes.

Once we select a finite field, we indicate its order after the Lie group symbol in parenthesis. So, for instance, we write $A_n(q)$ for the group of Lie type A_n over the field $\mathbb{F}_q$.

For more background on Lie algebras and Chevalley groups, see the notes by Geck [29].

4. Piggy-backing the permutation representation

The matrix representation is a very efficient way to work with groups. However, for most of the common group algorithms, we need a stabilizer chain with respect to a permutation action. A stabilizer chain is defined by a base, which is a set of points which when stabilized leave only the trivial subgroup. Constructing a base and stabilizer chain for a given permutation group is a main requirement for computational group theory.

What do we mean by piggy-backing the permutation representation? Traditionally, a permutation representation is realized by storing the images of the elements in the domain under the group element in question. The data structure is a vector (or list) of integers, essentially a table. In case two group elements need to be multiplied, we concatenate the mappings and produce another vector of images. When the element needs to be inverted, we read the mapping backwards. This kind of premutation representation has one disadvantage: Multiplication and inversion of group elements is expensive because an array of length N needs to be examined completely. Here, N is the permutation degree. The point is that finite matrix groups often have large permutation domains, which renders this approach to permutation groups ineffective.

Let us consider a finite matrix group G. it has a representation on a vector space $\mathbb{F}_q^n$ of degree $N = q^n$. The group action is given by vector-matrix multiplication. The group multiplication is multiplication of matrices, or perhaps slightly more complicated including field automorphisms. The permutation representation is based on an enumeration of the vectors. An enumeration is a fixed bijection between the set of vectors and a suitable interval of integers of size N called the domain. We may assume that the domain is $D = \{0, 1, \ldots, N-1\}$. The bijection is implicitly defined by means of two conversion algorithms, called **Rank** and **Unrank**. Ranking and unranking is a technique that is very common in computer algebra system, as it helps alleviate the need for tables. **Rank** takes a vector and returns the associated integer in D. **Unrank** takes an integer in D and returns the associated vector. The process of ranking and unranking a set of objects is called enumeration. Piggy-backing and enumerations are available among others in GAP [28] and Magma [13], and also in Orbiter and Fining [3]. In Magma, enumerations are provided by indexing objects. Tables take time to be set up and cost memory. Enumerations help to avoid this at the price of a small slowdown in evaluating the permutation action. Namely, whenever we want to know what the group element g does to the integer i, we first turn i into a vector using the enumeration, we then multiply the matrix to the vector to get a new vector, and we then convert that second vector back into an integer j using the enumeration.

In Figure 2, we show the schematic approach to piggy-backing the permutation action.

The top row shows the user interfacing input and output numbers. Below that is the matrix representation. The group element is stored as a matrix, possibly with a field automorphism (in case of semilinear group actions). The advantage is the efficient storage of group elements as matrices. Piggy-backing is also helpful when exploiting the representation of the group. Things like computing eigenvectors are impossible in the permutation representation. On the other hand, stabilizer chains are necessary to represent groups on a computer, so the permutation representation is needed. So, piggy-backing serves both goals at the same time.

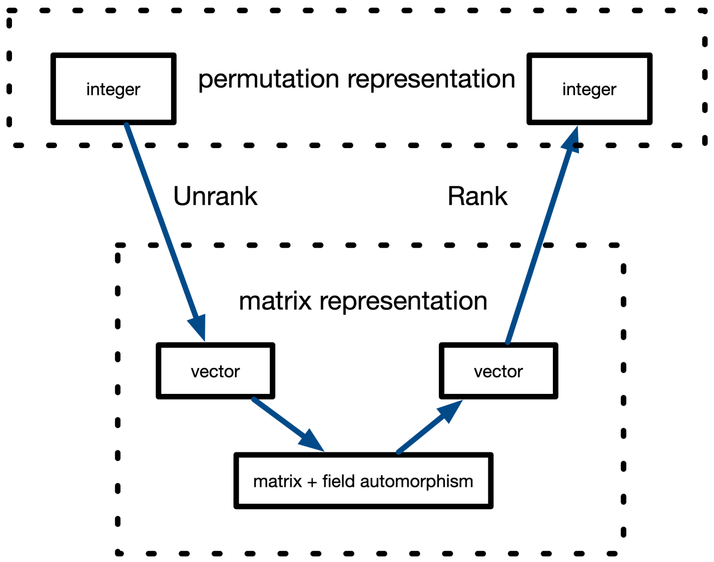

Figure 2: Computing the image in a permutation representation of a matrix group.

5. Generators for the Groups of Lie Type

How to create these groups of Lie type explicitly? It depends a bit on what kind of generating system we want and which of the versions of a given group we want. Do we want the simple group of do we want one of the various extension? It was shown by Steinberg [42] that all nonabelian simple groups can be generated by two elements (the case of abelian simple groups is trivial, and corresponds to cyclic groups which are generated by one element). Generating systems for specific types of groups were given by in [40] and [35].

However, in most cases, a generating system is not good enough, as we need to establish a stabilizer chain (also known as a Sims chain) for the group with respect to a chosen permutation representation. This means we are looking for a base and strong generating set. This can be done using the Schreier-Sims algorithm, see [34] of [41]. But them it seems reasonable to describe a base and strong generating set directly. In [10, Chapter 9], we find a description of a base and strong generating set for the general linear group over a finite field (and the associated collineation group). This has been implemented in Orbiter. For the orthogonal groups, we can use Siegel transformations to generate the simple group and orthogonal reflections for the full group.

A different approach of creating the orthogonal groups or the unitary groups is to create the associated variety first and then to use an algorithm to compute its group (see [2]). Of course, this may be much more difficult, but it is good to know that such an algorithm exists. This work relies on indexing of points and lines of the finite projective geometry. Using these, the incidence matrix of the geometry is established: the indexing provides a fixed ordering of the rows and columns of the incidence matrix. Next, the decorated matrix is computed to include information about the combinatorial object (the variety). The incidence matrix is extended by one additional row and one additional column. This additional row and column is used to distinguish the points of the geometry. After that, a bipartite graph, known as Levi graph, is defined from the decorated incidence matrix. One class of vertices corresponding to rows of the decorated incidence matrix, and the other class corresponding to the columns. The automorphism group of the Levi graph is computed using Nauty [36], a graph canonization software. Nauty also computes a base and strong generators for the automorphism group of the graph. The generators obtained from Nauty are permutations of the vertices of the Levi graph. They can be lifted to semilinear mappings of the underlying projective space. Once this is done for the original strong generating set, we perform a randomized Schreier Sims to get a new strong generating set with respect to the standard base of the appropriate (semilinear) general linear group.

5.1. *The Klein Quadric*

The Klein quadric defines the $Q^+(5,q)$ variety. For our purposes, the canonical equation may be taken as

$$X_0X_1 + X_2X_3 + X_4X_5 = 0. \tag{3}$$

With respect to the standard basis, the associated form matrix (1) is given by

$$B = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \tag{4}$$

The associated orthogonal group is $\mathrm{PGO}^+(6,q)$. Under the Klein correspondence (Plücker coordinates) the points on the quadric correspond one-to-one to the lines of $\mathrm{PG}(3,q)$. The group $\mathrm{PGL}(4,q)$ associated with projective space $\mathrm{PG}(3,q)$ embeds as a subgroup of index two into $\mathrm{PGO}^+(6,q)$. The coset of this subgroup in the full group $\mathrm{PGO}^+(6,q)$ correspond to the polarities of $\mathrm{PG}(3,q)$. A polarity is a inclusion-reversing automorphism of the underlying projective space of order two.

Let us look at the group and the geometry for $q = 2$. In this case, the variety has 35 singular points and 28 points which are non-singular. The group $\mathrm{PGO}^+(6,2)$ has order 40320. A list of the 22 conjugacy classes of elements is shown in Table 1.

The group is generated by the 28 elements of class 2A, which are the orthogonal reflections defined by the 28 points off the quadric. Table 2 shows the coordinate vectors of the 28 points off the quadric. Table 3 shows representatives for the two conjugacy classes of Siegel transformations:

$$\rho_{\mathbf{u},\mathbf{v}_1}, \quad \rho_{\mathbf{u},\mathbf{v}_2}$$

with $\mathbf{v}_1$ singular and $\mathbf{v}_2$ non-singular. In Table 1, the class containing $\rho_{\mathbf{u},\mathbf{v}_i}$ is called Siegel i. There are two classes of Siegel transformations in this group.

5.2. *The Study Quadric*

Consider next the orthogonal group $\mathrm{PGO}^+(8,q)$ and its action on the Study quadric, given by the equation

$$X_0X_1 + X_2X_3 + X_4X_5 + X_6X_7 = 0. \tag{5}$$

Study [43] uses the quadric to represent Euclidean motions in three dimensional Euclidean space. This work is still relevant in the theory of Kinematic (see [31],[25],[37]). For more on quadrics over a finite field, see [33].

0	1	1A	1	40320	40320	
1	2	2A	28	1440	1440	orth. refl.
2	2	2B	105	384	384	Siegel 1
3	2	2C	210	192	192	Siegel 2
4	2	2D	420	96	96	
5	3	3A	112	360	720	
6	3	3B	1120	36	72	
7	4	4A	420	96	192	
8	4	4B	1260	32	64	
9	4	4C	1260	32	64	
10	4	4D	2520	16	32	
11	5	5A	1344	30	120	
12	6	6A	1120	36	72	
13	6	6B	1120	36	72	
14	6	6C	1680	24	48	
15	6	6D	3360	12	24	
16	6	6E	3360	12	24	
17	7	7A	5760	7	42	
18	8	8A	5040	8	32	
19	10	10A	4032	10	40	
20	12	12A	3360	12	48	
21	15	15A	2688	15	120	

Table 1: Classes of elements of $\mathrm{PGO}^{+}(6,2)$

111111	011100	011110	011101	011011
110000	110010	110001	000011	000111
111000	111010	111001	100011	100111
110100	110110	110101	010011	010111
001100	001110	001101	001011	
101100	101110	101101	101011	

Table 2: The 28 points off the Klein quadric $Q^{+}(5,2)$

$$\mathbf{u} = (1,0,0,0,0,0), \quad \mathbf{v}_1 = (0,0,1,0,0,0), \quad \mathbf{v}_2 = (0,0,1,1,0,0),$$

$$\rho_{\mathbf{u},\mathbf{v}_1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \rho_{\mathbf{u},\mathbf{v}_2} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Table 3: Some associated Siegel transformations

Let us consider the group $\mathrm{PGO}^{+}(8,2)$ of order 348364800. The group $\mathrm{PGO}^{+}(8,2)$ stabilizes the quadric defined by (5) inside $\mathrm{PG}(7,2)$. The 255 points of $\mathrm{PG}(7,2)$ fall into two classes. The quadric defines 135 singular points and 120 non-singular points. For reference, Table 4 shows the coordinate vectors of the 120 points off the quadric.

11000000	00101100	11010110	11101001	00000011	10000111
11100000	10101100	00110110	11011001	10000011	01000111
11010000	01101100	10110110	00111001	01000011	00100111
00110000	00011100	01110110	10111001	00100011	10100111
10110000	10011100	00001110	01111001	10100011	01100111
01110000	01011100	10001110	11000101	01100011	00010111
11001000	11111100	01001110	11100101	00010011	10010111
11101000	11000010	00101110	11010101	10010011	01010111
11011000	11100010	10101110	00110101	01010011	11110111
00111000	11010010	01101110	10110101	11110011	11001111
10111000	00110010	00011110	01110101	00001011	11101111
01111000	10110010	10011110	00001101	10001011	11011111
11000100	01110010	01011110	10001101	01001011	00111111
11100100	11001010	11111110	01001101	00101011	10111111
11010100	11101010	11000001	00101101	10101011	01111111
00110100	11011010	11100001	10101101	01101011	
10110100	00111010	11010001	01101101	00011011	
01110100	10111010	00110001	00011101	10011011	
00001100	01111010	10110001	10011101	01011011	
10001100	11000110	01110001	01011101	11111011	
01001100	11100110	11001001	11111101	00000111	

Table 4: The 120 points off the Study quadric $\mathbf{Q}^+(7,2)$

The Orbiter command to create the group will be discussed in Section 8.1.

Next, we turn to conjugacy classes of elements in the orthogonal group. We find 67 conjugacy classes. The classes of elements of order at most 4 are listed in Table 5. The columns are the number of the class, the order of elements in the class, an ATLAS style label for the class, the size of the class, the order of the centralizer of an element of the class, the order of the normalizer of the group generated by an element of the class.

0	1	1A	1	348364800	348364800
1	2	2A	120	2903040	2903040
2	2	2B	1575	221184	221184
3	2	2C	3780	92160	92160
4	2	2D	7560	46080	46080
5	2	2E	37800	9216	9216
6	2	2F	56700	6144	6144
7	3	3A	2240	155520	311040
8	3	3B	4480	77760	155520
9	3	3C	89600	3888	7776
10	3	3D	268800	1296	2592

Table 5: Classes of elements of order at most 3 of $\mathrm{PGO}^+(8,2)$

The 120 elements in conjugacy class 2A are in one-to-one correspondence to the 120 orthogonal reflections as defined in 2, where $\mathbf{P}(\mathbf{z})$ is any of the 120 elements off the quadric as shown in Table 4.

0	1	1A	1	174182400	174182400
1	2	2A	1575	110592	110592
2	2	2B	3780	46080	46080
3	2	2C	3780	46080	46080
4	2	2D	3780	46080	46080
5	2	2E	56700	3072	3072
6	3	3A	2240	77760	155520
7	3	3B	2240	77760	155520
8	3	3C	2240	77760	155520
9	3	3D	89600	1944	3888
10	3	3E	268800	648	1296

Table 6: Classes of elements of order at most 4 of $\mathrm{P\Omega}^+(8,2)$

5.3. *The Simple Group*

In general, the groups $\mathrm{PGO}^\epsilon(n,q)$ are not simple. However, in most cases they contain a simple subgroup $\mathrm{P\Omega}^\epsilon(n,q)$. In some cases, the simple group can be obtained by taking the derived group. Recall that for a group G, and for g and h elements of G, the commutator is

$$[g,h] = g^{-1}h^{-1}gh.$$

The commutator subgroup of G, denoted as G', is the group generated by all commutators in G. It is known to be the smallest normal subgroup of G such that the factor group is abelian.

The derived group of $\mathrm{PGO}^+(8,2)$ is the group $\mathrm{PGO}^+(8,2)'$ also known as $\mathrm{P\Omega}^+(8,2)$. It has index two in $\mathrm{PGO}^+(8,2)$ as has order 174182400. As we pass to the derived subgroup, some of the conjugacy classes of $\mathrm{PGO}^+(8,2)$ disappear. Others split into two. The classes of elements of order at most 3 are listed in Table 6.

5.4. *The Graph on the Set of Involutions*

The graph Γ_i has as vertices the elements of class 2A (120 elements). Two vertices are adjacent in Γ_i if the product of the associated elements has order i (for $i = 2,3$). Each of these relations defines a strongly regular graph (srg). The parameters of a strongly regular graph are denoted as $\mathrm{srg}(v,k,\lambda,\mu)$. We have the following result, which can be easily verified by Orbiter, for instance. Of course, this result is well-known. We will need these graphs in Section 6.

Theorem 5.1. 1. *The graph* Γ_2 *is a* $\mathrm{srg}(120,63,30,36)$.

2. *The graph* Γ_3 *is a* $\mathrm{srg}(120,56,28,24)$.

3. *The graphs* Γ_2 *and* Γ_3 *are complements of each other.*

4. *The automorphism group of* Γ_i*, for each of* $i = 2,3$*, is* $\mathrm{PGO}^+(8,2)$ *of order* 348364800.

6. The Dynkin Diagram of type D_8

Let G be a group with associated Dynkin diagram $\mathbb{X}$. Temporarily label the nodes of the Dynkin diagram with the integers $1,\dots,n$. A valid labeling of $\mathbb{X}$ with labels in G is an assignment of involutions from G to the nodes of $\mathbb{X}$, $i \mapsto g_i$, such that the order of g_ig_j is equal to the order implied by the Dynkin diagram. Now, regarding the order implied by the diagram, two adjacent nodes imply that the order is three. Two non-adjacent nodes imply that the order is two. If there is a number written along the edge, then the order implied is this number. The Dynkin diagram of type D_8 implies that the order of products of pairs is either two or three.

The Dynkin diagram of type D_8 is shown in Figure 3.

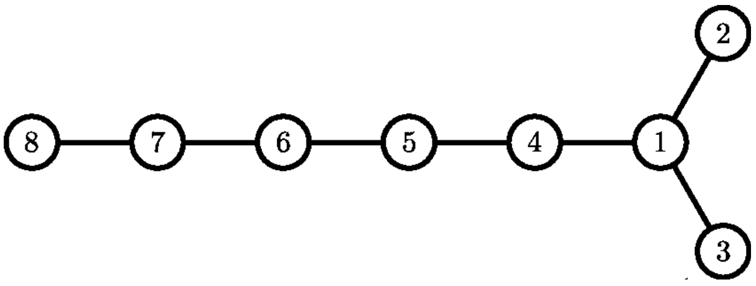

Figure 3: The Dynkin diagram of type D_8

In order to make it easier to talk about possible labelings, a numbering of nodes has been chosen.

Next, we will use the graphs from Section 5.4 to find all valid labelings of the Dynkin diagram of type D_8. To search for all labelings, we use the following algorithm:

Algorithm 6.1.

```
Let $C$ be the conjugacy class 2A of size 120 in $\mathrm{PGO}^+(8,2)$.
Let $C_1 := C$
For each $g_1 \in C_1$
  Let $C_2 := \{g \in C \setminus \{g_1\} \mid |g_1 g| = 3\}$.
  For each $g_2 \in C_2$.
    Let $C_3 := \{g \in C \setminus \{g_1, g_2\} \mid |g_1 g| = 3, |g_2 g| = 2\}$.
    For each $g_3 \in C_3$.
      Let $C_4 := \{g \in C \setminus \{g_1, g_2, g_3\} \mid |g_1 g| = 3, |g_2 g| = 2,$
        $|g_3 g| = 2\}$.
      For each $g_4 \in C_4$.
        Let $C_5 := \{g \in C \setminus \{g_1, g_2, g_3, g_4\} \mid |g_1 g| = 2,$
          $|g_2 g| = 2, |g_3 g| = 2, |g_4 g| = 3\}$.
        For each $g_5 \in C_5$.
          Let $C_6 := \{g \in C \setminus \{g_1, g_2, g_3, g_4, g_5\} \mid$
            $|g_1 g| = 2, |g_2 g| = 2, |g_3 g| = 2, |g_4 g| = 2, |g_5 g| = 3\}$.
          For each $g_6 \in C_6$.
            Let $C_7 := \{g \in C \setminus \{g_1, g_2, g_3, g_4, g_5, g_6\} \mid$
              $|g_1 g| = 2, |g_2 g| = 2, |g_3 g| = 2, |g_4 g| = 2, |g_5 g| = 2,$
              $|g_6 g| = 3\}$.
            For each $g_7 \in C_7$.
              Let $C_8 := \{g \in C \setminus \{g_1, g_2, g_3, g_4, g_5, g_6, g_7\} \mid$
                $|g_1 g| = 2, |g_2 g| = 2, |g_3 g| = 2, |g_4 g| = 2,$
                $|g_5 g| = 2, |g_6 g| = 2, |g_7 g| = 3\}$.
              For each $g_8 \in C_8$.
                Found a valid labeling
                  $(g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_8)$.
              end for.
            end for.
          end for.
        end for.
      end for.
    end for.
  end for.
end for.
```

The large number of loops makes us worry about the running time of the algorithm. As we will see in the proof of the following theorem, we can use the symmetry in the conjugacy class to our benefit. Namely, we can make one choice arbitrary, which eliminates one loop. The remaining search turns out to be doable. Here is the result:

Theorem 6.2. *There are 696729600 valid labelings of the D_8 Dynkin diagram with respect to generators from the conjugacy class of orthogonal reflections of* $\mathrm{PGO}^+(8,2)$ *of size 120. These labelings correspond to the elements of the automorphism group of* $\mathrm{PGO}^+(8,2)$.

Proof. The group $G = \mathrm{PGO}^+(8,2)$ of order 348364800 has one conjugacy class of orthogonal reflections of size 120 (class 2A in Table 6). Because the group acts transitively on the conjugacy class, we may pick one representative σ and construct the valid labelings of the Dynkin diagram of type D_8 with $\sigma_{\mathbf{z}}$ the label of a fixed node of the diagram. We pick element 10101011 from the list of external points of the quadric in Table 4, so

$$\mathbf{z} = \mathbf{P}(1,0,1,0,1,0,1,1)$$

This element is assigned to node 1. The labeling algorithm 6.1 produces a list of 5806080 completions. Since the element $\mathbf{z}$ is one of 120 different choices, each of which would lead to the same number of completions, we can estimate the total number of solutions. Namely, we find that the number of valid labelings is

$$696729600 = 120 \cdot 5806080.$$

Since

$$696729600 = 2 \cdot 348364800,$$

we have created a group which is twice bigger than the group $\mathrm{PGO}^+(8,2)$. The elements arise from the labelings in such a way: Fix one labeling, say

$$\sigma_{\mathbf{z}_1}, \ldots, \sigma_{\mathbf{z}_8}.$$

Any second labeling

$$\sigma_{\mathbf{w}_1}, \ldots, \sigma_{\mathbf{w}_8},$$

defines an automorphism of G. Namely, take any element g of G. Since G is generated by orthogonal reflections, we can find an expression

$$g = \sigma_{a_1} \cdots \sigma_{a_\ell}.$$

with a_i indices into the 120 elements of class 2A. We may assume that

$$a_i = \sigma_{\mathbf{z}_{\epsilon_i}}, \quad i = 1, \ldots, 8.$$

Simply replace g by

$$g = \sigma_{\mathbf{w}_{\epsilon_1}} \cdots \sigma_{\mathbf{w}_{\epsilon_\ell}}$$

to define an automorphism of $\mathrm{PGO}^+(8,2)$. The automorphism group of $\mathrm{PGO}^+(8,2)$ has order twice the order of $\mathrm{PGO}^+(8,2)$. This is because of the presence of a graph automorphism of the Dynkin diagram (nodes 2 and 3 can be exchanged). □

The order of all products of pairs can be precomputed and the data can be stored in the form of graphs. We use the graphs graphs Γ_2 and Γ_3 from Section 5.4.

As an example, let us look at one labeling from the list that was produced by the algorithm. Using σ_i to denote the orthogonal reflection associated with the external point P_i from Table 4, we find that the following assignment from nodes to involutions is valid:

$$(1,2,3,4,5,6,7,8) \mapsto \begin{pmatrix} \sigma_{10101011}, \sigma_{11000000}, \sigma_{00110000}, \sigma_{00001100}, \\ \sigma_{11111011}, \sigma_{11001111}, \sigma_{11111110}, \sigma_{00000011} \end{pmatrix}$$

For completeness, the external points involved in this labeling are (the subscript to P in the first column indicates the Orbiter number of the point)

$$\begin{aligned}
P_{112} &= \mathbf{P}(1,0,1,0,1,0,1,1), \\
P_{0} &= \mathbf{P}(1,1,0,0,0,0,0,0), \\
P_{1} &= \mathbf{P}(0,0,1,1,0,0,0,0), \\
P_{2} &= \mathbf{P}(0,0,0,0,1,1,0,0), \\
P_{12} &= \mathbf{P}(1,1,1,1,1,0,1,1), \\
P_{5} &= \mathbf{P}(1,1,0,0,1,1,1,1), \\
P_{8} &= \mathbf{P}(1,1,1,1,1,1,1,0), \\
P_{3} &= \mathbf{P}(0,0,0,0,0,0,1,1).
\end{aligned}$$

The matrices of the associated orthogonal reflections are given next:

$$\sigma_{10101011} = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 1&1&1&0&1&0&1&1 \\ 0&0&1&0&0&0&0&0 \\ 1&0&1&1&1&0&1&1 \\ 0&0&0&0&1&0&0&0 \\ 1&0&1&0&1&1&1&1 \\ 1&0&1&0&1&0&0&1 \\ 1&0&1&0&1&0&1&0 \end{bmatrix}, \; \sigma_{11000000} = \begin{bmatrix} 0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \end{bmatrix},$$

$$\sigma_{00110000} = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \end{bmatrix}, \; \sigma_{00001100} = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \end{bmatrix},$$

$$\sigma_{11111011} = \begin{bmatrix} 0&1&1&1&1&0&1&1 \\ 1&0&1&1&1&0&1&1 \\ 1&1&0&1&1&0&1&1 \\ 1&1&1&0&1&0&1&1 \\ 0&0&0&0&1&0&0&0 \\ 1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&0&0&1 \\ 1&1&1&1&1&0&1&0 \end{bmatrix}, \; \sigma_{11001111} = \begin{bmatrix} 0&1&0&0&1&1&1&1 \\ 1&0&0&0&1&1&1&1 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 1&1&0&0&0&1&1&1 \\ 1&1&0&0&1&0&1&1 \\ 1&1&0&0&1&1&0&1 \\ 1&1&0&0&1&1&1&0 \end{bmatrix},$$

$$\sigma_{11111110} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \sigma_{00000011} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

7. The Word Tree

The main problem we wish to address is that of converting from the matrix representation to the representation as word in a chosen set of generators. As already mentioned, the orthogonal reflections generate the orthogonal group in most cases. We will now express the elements of G as words in these generators. A word is an expression of the form

$$s_{i_1}^{\epsilon_1} \cdots s_{i_k}^{\epsilon_k} \tag{6}$$

with s_{i_j} an element of the chosen generating set and $\epsilon_{i_j} \in \{\pm 1\}$. In the case of involutorial generators, we may dispense with the ϵi_j as every element is its own inverse. The length of a word is the integer k in the expression (6). A word is reduced if there are no possible cancellations among consecutive terms. The length of a group element g (w.r.t. the chosen generating set) is the shortest length of a word that expresses g in terms of the given generators. We are interested in the word length of the elements of an orthogonal group in terms of specific generating sets.

The following result gives the word length of all elements of $\mathrm{PGO}^+(6,2)$ in terms of the generating set of orthogonal reflections. In the case of $G = \mathrm{PGO}^+(6,2)$, there are 28 orthogonal reflection, and they form conjugacy class 2A in Table 1.

Theorem 7.1. *The word length is a class property if the generating set is a union of conjugacy classes. For* $\mathrm{PGO}^+(6,2)$, *the possible word lengths are listed in Table 7, using the labeling of classes given in Table 1.*

length	classes
0	1A
1	2A
2	2C, 3A
3	2D, 4A, 6B
4	2B, 3B, 4D, 5A, 6C
5	4B, 6A, 6E, 10A, 12A
6	4C, 6D, 7A, 15A
7	8A

Table 7: Word length in $\mathrm{PGO}^+(6,2)$ by classes

Proof. Suppose that

$$g = s_{i_1}^{\epsilon_1} \cdots s_{i_k}^{\epsilon_k}$$

is an expression for g as a word in the generators. Conjugation by h yields

$$g^h = (s_{i_1}^h)^{\epsilon_1} \cdots (s_{i_k}^h)^{\epsilon_k}.$$

By assumption, the generating set is invariant under conjugation, so the

$$s_{i_j}^h$$

are themselves elements in the generating set. So, the expression for g^h is a word in the given set of generators, obviously of the same length as the original word. This means that the distance from the identity is a well-defined function on the conjugacy classes.

We can compute the orbit of a group on a set by using the technique of Schreier trees. The nodes of the tree are the elements of the group. The edges are labeled by generators. The root of the tree corresponds to the identity element. From any node, a set of successors is generated by right multiplication of the group element by exactly one generator at a time. So, if g represents a node, and s a generator, then gs considered for possible addition. If gs is already part of the tree, it is discarded. Otherwise, gs is added as a new node and an edge is created from g to gs, labeled by the generator s. This process is repeated by looping over the generators in a fixed order and by looping over all nodes as they are discovered. A FIFO queue is used to keep track of the nodes that have been discovered by not yet extended. By following the unique path from the root to any node, we can recover a word in the generators that expresses the group element. Because of the way that Schreier trees are built, this word is reduced. This means that the Schreier tree is in fact the word tree of the group.

Once the word tree is computed, we form the level sets. The level set L_i contains all elements g which are listed at distance i from the root in the tree. This means that L_i consists of those group elements whose word length is i. Once the level sets L_i are computed, we intersect them with the conjugacy classes. The data is summarized in Table 8. Row i represents level set L_i. The column # is the number of group elements in L_i. The columns to the right are sizes of the intersection of L_i with each of the conjugacy classes. The

length	#	1A	2A	2B	2C	2D	3A	3B	4A	4B	4C	4D
		1	28	105	210	420	112	1120	420	1260	1260	2520
0	1	1	0	0	0	0	0	0	0	0	0	0
1	28	0	28	0	0	0	0	0	0	0	0	0
2	322	0	0	0	210	0	112	0	0	0	0	0
3	1960	0	0	0	0	420	0	0	420	0	0	0
4	6769	0	0	105	0	0	0	1120	0	0	0	2520
5	13132	0	0	0	0	0	0	0	0	1260	0	0
6	13068	0	0	0	0	0	0	0	0	0	1260	0
7	5040	0	0	0	0	0	0	0	0	0	0	0

length	5A	6A	6B	6C	6D	6E	7A	8A	10A	12A	15A
	1344	1120	1120	1680	3360	3360	5760	5040	4032	3360	2688
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1120	0	0	0	0	0	0	0	0
4	1344	0	0	1680	0	0	0	0	0	0	0
5	0	1120	0	0	0	3360	0	0	4032	3360	0
6	0	0	0	0	3360	0	5760	0	0	0	2688
7	0	0	0	0	0	0	0	5040	0	0	0

Table 8: Shortest words in $\mathrm{PGO}^+(6,2)$ by classes

theorem is immediate from this table. □

8. Why C++?

There are many advantages to using C++. For starters, the language is has stood the test of time and offers a robust and stable environment. Development tools are plentiful. Support is excellent. Compatibility to old versions is outstanding. Many libraries exist, and the language itself supports large software projects, efficient data structures, abstraction, namespaces and most of all, classes. Classes embed into larger concepts like abstraction, data encapsulation, and code decoupling. Namespaces address the problem of symbol space pollution, which is one big disadvantage of C. Namespaces allow to write modular code, which is important in large software projects. In Orbiter, we utilize namespace nesting (up to four levels deep) to organize a huge code base of over one million lines of code. For comparison, in 2020, the Linux project had about 28 million lines of code. The ability to write very efficient code supported by efficient data structures is a big draw. The latter is important for problems like that of working with groups of Lie type. Memory efficiency is particularly important for large orbit computations. This is especially true for poset classification, which is an algorithm to compute orbits of a group on partially ordered sets. We have already talked about piggy-backing permutation representations on matrix representations. The language C++ supports abstract data types, so techniques like piggy-backing can be implemented and offer the benefit of a simpler interface to permutation groups.

In Computer Algebra, computations are often time critical and memory intense. Dynamic memory allocation is important for this. In C++, the memory is managed by the programmer. Other systems try to hide the memory management from the programmer. The problem is not the memory allocation but the release of unused memory back to the system. Garbage collection is used by many systems, but it comes at a cost. The running times becomes unpredictable, as the system may interrupt to do garbage collection at any time. Having the programmer responsible for returning unused memory eliminates the need for garbage collection, but it demands more attention on the side of the programmer. A drawback is the potential to create memory leaks, which arise when data is not returned to the system by mistake. Memory leaks are often difficult to detect and fix, though there are tools to support leak detection.

In Orbiter, abstraction is used to provide a uniform interface to groups and group actions. There is a set of standard functions that is offered for any group object. The permutation action is decoupled from the group implementation, and one group can appear with multiple actions. New actions can be created from old. The orbit algorithms build on this uniform interface. One could say that the orbit algorithm computes orbits of groups without knowing what kind of group it is. For a discussion of the orbit algorithms, see [7]. Orbit algorithms tie into posets with group action. Available posets are the poset of subsets of a set and the poset of subspaces of a vector space. The poset classification algorithm is memory hungry. It needs to store a large about of group elements either as generators of subgroups or as isomorphisms of substructures. Orbiter offers several representations of group elements. On the one hand, there is the matrix representation, where a matrix is a two dimensional array, which is linearized in the row-major form (which is also the default in C and C++). On the other hand, there is the compact representation, where a group element (matrix) is coded as a bit-vector. Here, we utilize the fact that we are working over a finite field $\mathbb{F}_q$. We need approximately $n^2 \log_2(q)$ bits to represent a group element in a matrix representation of degree n. The compact representation comes at a cost. Namely, the group element must be converted back and from the compact state to be used for computations. Nevertheless, the benefit in memory savings outweighs this additional level of complication. More importantly, the conversion to and from the compact representation happens inside the orbit algorithms, which is inside the Orbiter library. So, the user of the Orbiter library does not have to worry about it.

Orbiter also has a command-processor user interface, so it can be used as a computer algebra system with its own small language (called Orbiter dash code). Thus, Orbiter is in fact two things in one. On the one hand, it is a class library that can be used to write code. On the other hand, Orbiter can be used as a computer algebra system using the existing commands. For a description of the Orbiter dash code, see the user's guide [9] or Section 8.1 below. Describing the features of the Orbiter library is beyond the scope of this paper, unfortunately.

Consider an example. We create the group $\mathrm{PGO}^+(6,2)$. The stabilizer chain is shown in Table 9. The

Level	Base pt	Orbit length	Subgroup order
0	0	35	40320
1	1	16	1152
2	2	9	72
3	3	1	8
4	4	1	8
5	5	4	8
6	9	2	2
7			1

Table 9: Stabilizer chain for the group $\mathrm{PGO}^+(6,2)$ in the permutation action on the Klein quadric.

group order is 40320. The degree of the action is 35, the number of singular points (i.e., points on the Klein quadric). The base points are

$$\begin{aligned} 0 &= \mathbf{P}(1,0,0,0,0,0),\\ 1 &= \mathbf{P}(0,1,0,0,0,0),\\ 2 &= \mathbf{P}(0,0,1,0,0,0),\\ 3 &= \mathbf{P}(1,0,1,0,0,0),\\ 4 &= \mathbf{P}(0,1,1,0,0,0),\\ 5 &= \mathbf{P}(0,0,0,1,0,0),\\ 9 &= \mathbf{P}(0,0,0,0,1,0). \end{aligned}$$

A strong generating set is

$$\begin{bmatrix}100000\\010000\\001000\\000100\\000001\\000010\end{bmatrix}, \begin{bmatrix}100000\\010000\\001000\\001111\\001001\\001010\end{bmatrix}, \begin{bmatrix}100000\\010000\\001000\\000110\\001001\\000010\end{bmatrix}, \begin{bmatrix}100000\\010000\\000110\\000001\\001001\\000100\end{bmatrix}, \begin{bmatrix}100000\\010000\\000100\\001010\\000010\\000101\end{bmatrix}, \begin{bmatrix}100000\\011111\\100010\\100001\\101000\\100100\end{bmatrix}, \begin{bmatrix}100000\\010100\\101001\\000100\\000110\\000001\end{bmatrix}, \begin{bmatrix}010110\\001000\\010000\\101010\\000010\\011001\end{bmatrix}, \begin{bmatrix}001111\\101000\\011111\\111100\\010101\\010110\end{bmatrix}, \begin{bmatrix}000010\\101111\\101010\\100110\\100000\\111110\end{bmatrix}.$$

8.1. *The Orbiter Dash Code Language*

The Orbiter command line processor offers its own very basic language for symbolic algebra. It has a symbol table for storing objects. The syntax of the Orbiter dash code language is very easy. There are only 4 types of instructions:

1. Definitions: A new object can be defined and put in the symbol table. Various parameters and options can be set. Example: We can define a linear group or a projective space, a polynomial ring, an algebraic variety or a combinatorial object.

2. Activity: Given one or more objects, an activity is a command that applies to the given objects. Example: Given a variety, we can compute the automorphism group

3. Assignment: Some activities return new objects. These new objects can be put in the symbol table under a new name.

4. Global command: A global command is a command that does not need any input object.

Orbiter commands resemble Unix command line arguments, with a dash preceding keywords (hence the name dash code). Let us look at an example. The following Orbiter command sequence can be used to create the group $PGO^{+}(6, 2)$ from the equation given in (3). To begin with, we encode the equation of the variety as makefile variable in algebraic form. The equation is written in algebraic form in plain text. All multiplication symbols must be written as star symbols.

```
Op_6_eqn="X0*X1+X2*X3+X4*X5"
```

The next command creates the variety and the group:

```
Op_6_2_group:
▹  $(ORBITER) -v 6 \
▹  ▹  -define F -finite_field -q 2 -end \
▹  ▹  -define P -projective_space -n 5 -field F -v 0 -end \
▹  ▹  -define R -polynomial_ring \
▹  ▹  ▹  -field F \
▹  ▹  ▹  -number_of_variables 6 \
▹  ▹  ▹  -homogeneous_of_degree 2 \
▹  ▹  ▹  -monomial_ordering_partition \
▹  ▹  ▹  -variables "X0,X1,X2,X3,X4,X5" "X_0,X_1,X_2,X_3,X_4,X_5
" \
▹  ▹  -end \
▹  ▹  -define V -variety \
▹  ▹  ▹  -projective_space P \
▹  ▹  ▹  -ring R \
▹  ▹  ▹  -equation_in_algebraic_form \
▹  ▹  ▹  ▹  $(Op_6_eqn) \
▹  ▹  ▹  -label_txt Op_6_2 \
▹  ▹  ▹  -label_tex "{\\rm Op\_6\_2}" \
▹  ▹  -end \
▹  ▹  -with V -do -variety_activity -compute_group \
▹  ▹  ▹  -nauty_control -save_orbit_of_equations eqn_ -end \
▹  ▹  -end \
▹  ▹  -with V -do -variety_activity -singular_points -end \
▹  ▹  -with V -do -variety_activity -report -end
▹  pdflatex variety_Op_6_2_report.tex
▹  open variety_Op_6_2_report.pdf
```

The command consists of the following sequence of Orbiter commands:

1. An object of type `-draw_options` is defined to control the graphical output in the reporting functions later.

2. An object of type `-finite_field` is defined to represent the field $\mathbb{F}_2$. The order of the field is specified using the `-q` option.

3. An object of type `-projective_space` is defined to represent the projective space PG(5, 2). The projective dimension is specified using the `-n` option. The field is specified using the `-field` option. The label F refers to the field object defined in step 2.

4. An object of type `-polynomial_ring` is defined to represent the quadratic component of the polynomial ring $\mathbb{F}_2[X_0, X_1, \ldots, X_5]$. An ordering of the monomials is chosen.

5. An object of type `-variety` is defined to represent the algebraic variety cut out by equation (3). The equation is pulled from the makefile variable using the `$(NAME)` syntax, where `NAME` is the name of a makefile variable defined earlier. This syntax is the syntax of unix shell commands. Here, we use `$(Op_6_eqn)` to refer to the makefile variable defined previously.

6. An object of type `-modified_group` is defined to represent the stabilizer of the variety. The group is computed using the algorithm described in [2]. The command `-nauty_control` can be used to set options for Nauty. Nauty is a C-library which is a toolbox for graph canonization, see [36]. Nauty is required to compute the automorphism group of a variety.

7. A group theoretic activity is invoked for the group of the variety, which has been stored in the variable G. The activity asks to produce a report of the variety. The report is a latex file which summarizes known properties of the variety. Known properties include the automorphism group order, the number of points, the points, possible singularities, and other things.

Once the Orbiter command finishes, the makefile invokes `pdflatex` to translate the latex command and `open` to show the document on the screen. This finishes the session.

8.2. Abstraction

How does abstraction look like in Orbiter? Let is look at the code to compute the orbits of a group on a set. We will look at two different group actions and compare the Orbiter dash code. Here is the first code example. We create the group $\mathrm{PGO}^+(6, 2)$ and produce a latex report for it:

```
Op_6_2_report:
▹  $(ORBITER) -v 10 \
▹  ▹  -define Do -draw_options \
▹  ▹  ▹  -radius 200 -line_width 0.3 \
▹  ▹  -end \
▹  ▹  -define F -finite_field -q 2 -end \
▹  ▹  -define G -linear_group -PGOp 6 F -end \
▹  ▹  -with G -do \
▹  ▹  -group_theoretic_activity \
▹  ▹  ▹  -report Do \
▹  ▹  -end \
▹  ▹  -print_symbols
▹  pdflatex PGOp_6_2_report.tex
▹  open PGOp_6_2_report.pdf
```

The second code example creates the same group, but then we also create the generating set formed by the orthogonal reflections associated with the 28 points off the quadric. At first, we store the Orbiter index values of these 28 points as points in PG(5, 2) inside a makefile variable. These points have been computed by the code example `Op_6_2_group` above.

```
ORTHOGONAL_OP_6_2_POINTS_OFF="6,7,10,13,14,15,16,20,24,28,29,
30,31,35,39,43, 44, 45, 46, 48, 49, 50, 52, 53, 54, 56, 57, 5
8"
```

Next, we create the group and the action on itself by right-multiplication. This is done by creating a new group called R. This group R is not a completely new group. Instead, it is a modification of an existing group. The internal representation of the group remains, but the action changes. We then create the orbit tree using a Schreier algorithm, based on the generating set consisting of the orthogonal reflections.

```
Op_6_2_orbits:
▷  $(ORBITER) -v 4 \
▷  ▷  -define F -finite_field -q 2 -end \
▷  ▷  -define G -linear_group -PGOp 6 F -end \
▷  ▷  -define Pts -vector \
▷  ▷  ▷  -dense $(ORTHOGONAL_OP_6_2_POINTS_OFF) \
▷  ▷  -end \
▷  ▷  -define O -orthogonal_space 1 6 F -end \
▷  ▷  -with O -do -orthogonal_space_activity \
▷  ▷  ▷  -create_orthogonal_reflection Pts \
▷  ▷  -end \
▷  ▷  -define gens -vector_ge \
▷  ▷  ▷  -action G \
▷  ▷  ▷  -read_csv O_1_6_2_orthogonal_reflections.csv Element \
▷  ▷  -end \
▷  ▷  -define R -modified_group -from G \
▷  ▷  ▷  -action_on_self_by_right_multiplication \
▷  ▷  -end \
▷  ▷  -define Orb -orbits -group R \
▷  ▷  ▷  -on_points_with_generators gens \
▷  ▷  ▷  -print_interval 100 \
▷  ▷  -end
```

The computation shown in this example was the basis for the example presented in Section 7. Namely, from the group orbit, we can compute the reduced words of all elements, from which Table 7 follows. Note that the word length is a class function, because our generating set is the whole class of involutions 2A.

9. Conclusion

We have discussed the C++ library Orbiter for algebraic computations, in particular orbits of groups of Lie type under finite group actions. We discussed the specific needs of an orbit algorithm for orbits on a partially orcdered set. We looked at two applications. The first is computing the labelings of a Dynkin diagram of type D_n. This allows us to compute the automorphism group of an orthogonal group over of plus type which is defined over a finite field. A second application was the computation of word trees in Coxeter groups. We looked at the orthogonal group $\mathrm{PGO}^+(6,2)$ of order 40320 and its set of 28 orthogonal reflections. Based on this generating set, which is fully invariant under automorphisms, we found that the word tree is invariant also, and we were able to express the elements of each level as union of conjugacy classes of the group. Finally, we discussed the role of abstraction in computer algebra systems. We illustrated this by orbit algorithms for various types of groups and group actions in Orbiter, using a unified command syntax. Thoughout, we discussed the basic ideas of C++ programming, though much more would remain to be said.

Acknowledgement

The second author thanks Prof. Latif Hanna for stimulating discussions.

References

[1] Jeffrey Adams, Marc van Leeuwen, and Fokko du Cloux, 2025. *atlasofliegroups*, `https://github.com/jeffreyadams/atlasofliegroups.git`.

[2] Abdullah Alazemi and Anton Betten, 2024. Computing the Group of an Algebraic Variety over a Finite Field, In: K. Buzzard, et al. (ed): ICMS 2024, Springer Lecture Notes in Computer Science, 14749, pp 89–96 (2024).

[3] John Bamberg, Anton Betten, Philippe Cara, Jan De Beule, Michel Lavrauw, and Max Neunhöffer. *FinInG – Finite Incidence Geometry, Version 1.4.1*, 2018.

[4] John Bamberg, Anton Betten, Cheryl E. Praeger, and Alfred Wassermann. Unitals in the Desarguesian projective plane of order 16. *J. Statist. Plann. Inference*, 144:110–122, 2014.

[5] Anton Betten. Rainbow cliques and the classification of small BLT-sets. In *ISSAC 2013—Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation*, pages 53–60. ACM, New York, 2013.

[6] Anton Betten. The packings of PG$(3, 3)$. *Des. Codes Cryptogr.*, 79(3):583–595, 2016.

[7] Anton Betten. How fast can we compute orbits of groups? In *ICMS 2018—Proceedings of the International Congress on Mathematical Software; James H. Davenport, Manuel Kauers, George Labahn, Josef Urban (ed.)*, pages 62–70. Springer, 2018.

[8] Anton Betten, 2022. Orbiter – A program to classify discrete objects, 2022, `https://github.com/abetten/orbiter`.

[9] Anton Betten, 2022. Orbiter User's Guide. 2022. `https://www.math.colostate.edu/~betten/orbiter/users_guide.pdf`.

[10] Anton Betten, Michael Braun, Harald Fripertinger, Adalbert Kerber, Axel Kohnert, and Alfred Wassermann. *Error-correcting linear codes*, volume 18 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Berlin, 2006. Classification by isometry and applications, With 1 CD-ROM (Windows and Linux).

[11] Anton Betten and Fatma Karaoglu. Cubic surfaces over small finite fields. *Des. Codes Cryptogr.*, 87(4):931–953, 2019.

[12] Richard Bödi and Michael Joswig, 1998. *RealLie*, A software package for real representations of quasi-simple Lie groups `https://page.math.tu-berlin.de/~joswig/software/RealLie/index.html`.

[13] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[14] G. Butler. *Fundamental algorithms for permutation groups*, volume 559 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1991.

[15] Roger W. Carter. *Simple groups of Lie type*, volume Vol. 28 of *Pure and Applied Mathematics*. John Wiley & Sons, London-New York-Sydney, 1972.

[16] C. Chevalley. Sur certains groupes simples. *Tohoku Math. J. (2)*, 7:14–66, 1955.

[17] Arjeh M. Cohen, Scott H. Murray, and D. E. Taylor. Computing in groups of Lie type. *Math. Comp.*, 73(247):1477–1498, 2004.

[18] J. H. Conway, R. T. Curtis, S. P. Norton, R. A. Parker, and R. A. Wilson. *Atlas of finite groups*. Oxford University Press, Eynsham, 1985. Maximal subgroups and ordinary characters for simple groups, With computational assistance from J. G. Thackray.

[19] W. A. de Graaf and GAP Team, 2024. *SLA, Computing with simple Lie algebras*, Version 1.6.2 (2024) (GAP package), https://gap-packages.github.io/sla/.

[20] Willem A. de Graaf. *Lie algebras: theory and algorithms*, volume 56 of *North-Holland Mathematical Library*. North-Holland Publishing Co., Amsterdam, 2000.

[21] Willem A. de Graaf. Exploring Lie theory with gap. In *Computational Aspects of Discrete Subgroups of Lie Groups*, volume 783 of *Contemp. Math.*, pages 27–46. Amer. Math. Soc., [Providence], RI, [2023] ©2023.

[22] Willem Adriaan de Graaf. *Computation with linear algebraic groups*. Monographs and Research Notes in Mathematics. CRC Press, Boca Raton, FL, 2017.

[23] E. Dynkin. Classification of the simple Lie groups. *Rec. Math. [Mat. Sbornik] N.S.*, 18/60:347–352, 1946.

[24] E. B. Dynkin. The structure of semi-simple algebras. *Uspehi Matem. Nauk (N.S.)*, 2(4(20)):59–127, 1947.

[25] Johannes Karl Eberharter and Bahram Ravani. Local metrics for rigid body displacements. *Journal of Mechanical Design*, 126:805–812, 2004.

[26] Robert Feger, Thomas W. Kephart, and Robert J. Saskowski. Lieart 2.0 – a mathematica application for lie algebras and representation theory. *Computer Physics Communications*, 257:107490, 2020.

[27] William Fulton and Joe Harris. *Representation theory*, volume 129 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1991. A first course, Readings in Mathematics.

[28] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.11.1*, 2021. `http://www.gap-system.org`.

[29] Meinolf Geck, 2024. *A Course on Lie algebras and Chevalley groups*, Preprint on arXiv:2404.11472.

[30] Meinolf Geck, Gerhard Hiß, Frank Lübeck, Gunter Malle, Jean Michel, and Götz Pfeiffer, 2025. *the CHEVIE project*, `https://www.math.rwth-aachen.de/~CHEVIE/`.

[31] Anton Gfrerrer, 2000. *Study's Kinematic Mapping — A Tool for Motion Design*, Notes, University of Graz, 2000.

[32] J.I. Hall, 2014. *Classical Groups and Geometry*, Notes, Michigan State University, 2014.

[33] J. W. P. Hirschfeld. *Projective geometries over finite fields.* The Clarendon Press, Oxford University Press, New York,,, 1979.

[34] Derek F. Holt, Bettina Eick, and Eamonn A. O'Brien. *Handbook of computational group theory*. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2005.

[35] R. B. Howlett, L. J. Rylands, and D. E. Taylor. Matrix generators for exceptional groups of Lie type. *J. Symbolic Comput.*, 31(4):429–445, 2001.

[36] Brendan McKay, 2020. Nauty and Traces (Version 2.7r1), Australian National University, 2020.

[37] Boris Odehnal, Hellmuth Stachel, and Georg Glaeser. *The universe of quadrics*. Springer, Berlin, [2020] ©2020.

[38] Thomas Rehn, 2012. PermLib `https://github.com/tremlin/PermLib`.

[39] Thomas Rehn and Achill Schürmann, 2010. C++ Tools for Exploiting Polyhedral Symmetries. Lecture Notes in Computer Science, 2010, Volume 6327/2010.

[40] L. J. Rylands and D. E. Taylor. Matrix generators for the orthogonal groups. *J. Symbolic Comput.*, 25(3):351–360, 1998.
[41] Ákos Seress. *Permutation group algorithms*, volume 152 of *Cambridge Tracts in Mathematics*. Cambridge University Press, Cambridge, 2003.
[42] Robert Steinberg. Generators for simple groups. *Canadian J. Math.*, 14:277–283, 1962.
[43] Eduard Study, 1903. *Geometrie der Dynamen – Die Zusammensetzung von Kräften und verwandte Gegenstände der Geometrie*, Leipzig, Verlag und Druck von B.G. Teubner, 1903.
[44] D.E. Taylor. *The geometry of the classical groups*, volume 9 of *Sigma Series in Pure Mathematics*. Heldermann Verlag, Berlin, 1992.
[45] M. A. A. van Leeuwen, A. M. Cohen, and B. Lisser, 1992. *LiE, A Package for Lie Group Computations*, Computer Algebra Nederland, Amsterdam, ISBN 90-74116-02-7, 1992.
[46] Robert A. Wilson. *The finite simple groups*, volume 251 of *Graduate Texts in Mathematics*. Springer-Verlag London, Ltd., London, 2009.