Abstract

Orbiter is a computer algebra system devoted to the classification problem of combinatorial and algebraic-geometric objects. It uses computational methods from the theory of permutation groups to provide efficient algorithms for computing orbits. This guide explains how to use Orbiter through this command line interface. Programmers who want to use the Orbiter C++ class library directly in their own programs should consult the programmer’s guide.
1 Introduction

Orbiter is a computer algebra system for the classification of combinatorial objects. Orbiter contributes to the knowledge base of combinatorial structures, and to provide useful tools to investigate structures from various points of view, including their symmetry properties. Orbiter is optimized for efficiency in terms of memory and execution speed. Orbiter is a library of C++ classes, together with a command line driven front end. There is no graphical user interface. The system offers two modes of use, programming or command line interface. This manual is about the command line interface. Readers who are interested in the Orbiter C++ class library should consult the programmer’s guide. A makefile with all commands used in this guide can be found in the examples subdirectory.
2 Installation

The installation of Orbiter requires the following steps:

(a) Ensure that git and the C++ development suite are installed (gnuc and make). Windows users may have to install cygwin (plus the extra packages git, make, gnuc). Macintosh users may have to install the xcode development tools from the appstore (it is free). Linux users may have to install the development packages. Orbiter often produces latex reports. In order to compile these files, make sure you have latex installed.

(b) Clone the Orbiter source tree from github (abetten/orbiter). The commands are:

\[
git\ clone\ <\text{github-orbiter-path}>\]

where \text{github-orbiter-path} has to be replaced by the actual address provided by github. To obtain this path, find Orbiter on github, then click on the green box that says “Code” and copy the address into the clipboard by clicking the clipboard symbol (see Figure 1). Back in the terminal, you can paste this text after the \text{git clone} command.

(c) Issue the following commands to compile Orbiter:

\[
make \\
make\ install
\]

These two commands compile the Orbiter source tree and copy the executables to the subdirectory bin inside the Orbiter source tree. The orbiter executable is called orbiter.out.

Figure 1: GitHub Orbiter Page
3 The Orbiter Session

Any orbiter session is invoked through the orbiter command `orbiter.out`, which is the name of the executable. Unless the executable resides in a directory contained in the search path of the shell, a path must be given. Several options apply to the orbiter session. They are listed in Table 1. Once started, the Orbiter session will produce a short welcome message:

Welcome to Orbiter! Your build number is 1072.

The build number is the version number of the Orbiter software, as defined by the number of submits to the Git repository. Higher numbers mean more recent versions. After this message, Orbiter will start parsing the command line arguments. Once this is done, the session will execute these commands. At the end of the session, a short message is given that specifies the processor time used up by the session. In Unix terminology, this is called the user time.

Orbiter session finished.
User time: 0:00
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-v</td>
<td>v</td>
<td>Set verbosity to v. Larger values of v lead to more text output of the Orbiter session. v = 0 gives minimal output.</td>
</tr>
<tr>
<td>-list_arguments</td>
<td></td>
<td>Prints the command line arguments.</td>
</tr>
<tr>
<td>-seed</td>
<td>s</td>
<td>Seed the pseudo random number generator with the integer value s.</td>
</tr>
<tr>
<td>-memory_debug</td>
<td></td>
<td>Turn on dynamic memory debugging.</td>
</tr>
<tr>
<td>-override_polynomial</td>
<td>poly</td>
<td>Set the override polynomial for finite fields to poly. This is used when making a cheat sheet of a finite field $\mathbb{F}_q$.</td>
</tr>
<tr>
<td>-orbiter_path</td>
<td>p</td>
<td>Set the orbiter path to p. This is useful in case the Orbiter session has to clone or fork new Orbiter sessions. In most cases, the orbiter path will end with a forward slash “/.”</td>
</tr>
<tr>
<td>-magma_path</td>
<td>p</td>
<td>Set the magma path to p. This is useful in case the Orbiter session has to create a magma process.</td>
</tr>
<tr>
<td>-fork</td>
<td>L M f t s</td>
<td>Fork new Orbiter sessions in parallel. The new sessions will be indexed by the values $i$ that result from a loop with start-value $f$ and increment $s$ bounded from above by $t$, equivalent to a $C$-loop of type “for (i=f; i &lt; t; i+= s).” Every occurrence of the string $L$ in the argument list is replaced by the resulting value of the loop variable $i$. The forked process will write to a file whose name is described through the mask $M$. The actual file name results from using the printf command from the $C$-library for $M$ with the integer value of the loop variable. All of the command line arguments after the fork command are passed through to the new Orbiter session, with all arguments $L$ replaced by the integer value of the loop counter. The number of Orbiter sessions forked is $(t - f)/s$. The orbiter path from -orbiter_path is used when starting the forked sessions.</td>
</tr>
</tbody>
</table>

Table 1: Orbiter session commands
4 Finite Fields

A finite field $\mathbb{F}_q$ of order $q$ can be constructed in two ways. For this, we write $q = p^e$ for some primes $p$ and a positive integer $e$. The prime $p$ is called the characteristic of the field. If $e = 1$, the field is constructed from the integers modulo $p$. If $e$ is bigger than one, the field is an extension field of a prime field $\mathbb{F}_p$. It is constructed as a polynomials factoring. The polynomial ring is the ring $\mathbb{F}_p[X]$, and the ideal is generated by an irreducible polynomial in $\mathbb{F}_p[X]$ of degree $e$. The isomorphism type of the resulting field only depends on the order $q$ of the field, and not on the choice of the polynomial. However, for practical computations, it is important to know which polynomial is used. This way, results can be compared between different computer algebra systems. Orbiter has a large collection of polynomials built in. Besides these, a polynomial can be specified. The polynomials that Orbiter offers are in fact special. They are primitive, which means that the root $\alpha$ is a primitive element for the field $\mathbb{F}_q$. This just means that it is a generator of the multiplicative group. So, any non-zero element in $\mathbb{F}_q$ is a suitable power of $\alpha$.

In Orbiter, the elements of finite fields are represented as integers. The elements of the field $\mathbb{F}_q$ are mapped to the interval $[0, q - 1]$. It is helpful to know which ordering is used to facilitate this mapping. For prime fields, the field elements are represented as the positive remainders modulo $p$. For instance, in Figure 2 the addition and multiplication table of $\mathbb{F}_7$ with respect to the natural ordering of the interval $[0, 6]$ is shown (integers are represented by colors to make for a more appealing visual). It is customary to restrict the multiplication table to the non-zero elements of the field. The following Orbiter command was used to create the tables:

```
orbiter.out -cheat_sheet_GF 7
```

If $\mathbb{F}_q$ is an extension of the prime field $\mathbb{F}_p$, we use a different labeling. This time, we exploit the fact that $\mathbb{F}_q$ is a vector space over $\mathbb{F}_p$. Let $\alpha$ be a root of the irreducible polynomial $m(X) \in \mathbb{F}_p[X]$ used to create the field. Suppose that $q = p^e$, i.e., the degree of $m(X)$ is $e$. An $\mathbb{F}_p$-basis for the vector space $\mathbb{F}_q$ over $\mathbb{F}_p$ is given by the powers $\alpha^i$, for $0 \leq i < e$. 

![Figure 2: Addition and multiplication table of $\mathbb{F}_7$ using the standard ordering of elements](image)
Figure 3: Addition and multiplication table of $\mathbb{F}_{49}$ using the lexicographic ordering of elements.

Therefore, any element $\gamma$ of $\mathbb{F}_q$ has a unique expression of the form

$$\gamma = \sum_{h=0}^{e-1} a_i \alpha^i, \quad 0 \leq a_i < p \text{ for all } i.$$ 

The associated integer rank of $\gamma$ is obtained by replacing $\alpha$ by $p$ in this expression and evaluating the expression over the integers. So, the rank of $\gamma$ is

$$\sum_{h=0}^{e-1} a_i p^i.$$ 

It is clear that these values exhaust the interval $[0, q - 1]$ if $\gamma$ ranges over all field element in $\mathbb{F}_q$. The ordering of elements of $\mathbb{F}_q$ according to these ranks is called the lexicographical ordering. The numerical rank of zero is 0 and the numerical rank of one is 1. The numerical rank of $\alpha$, the primitive element, is $p$. The numerical ranks of the elements of the prime subfield are exactly the elements of $[0, p - 1]$. This makes for an interesting effect when displaying the addition and multiplication tables of the extension field. The tables of the prime field appear in the upper left corner, as seen for instance in Figure 3 for $\mathbb{F}_{49}$. The following Orbiter command was used to create the tables:

orbiter.out -cheat_sheet_GF 49

There is a second way of labeling the elements which is sometimes used. This labeling exhibits the cyclic structure of the multiplicative group. The non-zero elements are arranged
Figure 4: Addition and multiplication table of $\mathbb{F}_7$ using the second ordering of elements

Figure 5: Addition and multiplication table of $\mathbb{F}_{49}$ using the second ordering of elements
<table>
<thead>
<tr>
<th>Subfield</th>
<th>Polynomial</th>
<th>Numerical rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{F}_4$</td>
<td>$X^2 + X + 1$</td>
<td>7</td>
</tr>
<tr>
<td>$\mathbb{F}_8$</td>
<td>$X^3 + X + 1$</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 2: The subfields of $\mathbb{F}_{64}$

according to powers of a primitive element, see Figure 4. In this figure, as in all the following ones, the colors are consistent and represent the same values as in the prime field. For the extension field $\mathbb{F}_{49}$, see Figure 5. In the second ordering, the addition table of the prime field no longer exhibits cyclic structure (the additive group of an extension field is never cyclic).

The subfield embedding is not as nice as in the first ordering. Within the multiplicative group, the elements of the prime field $\mathbb{F}_p$ appear spaced out in intervals of $p + 1$ apart.

Orbiter uses primitive polynomials for creating extension fields. Because of this, the element $\alpha$ is always primitive. Since the numerical rank of $\alpha$ is $p$, this means that the rank $p$ always represents a primitive element in an extension field.

The command

```
orbiter.out -cheat_sheet_GF 4
```

creates a report for the field $\mathbb{F}_4$.

Unlike other computer algebra systems (GAP [11] and Magma [8]), Orbiter does not use Conway polynomials. Instead, it provides the option to override the polynomial used to create the finite field. For subfield relationships, the cheat sheet will indicate the irreducible polynomials of all subfields for a given field. For instance, Table 2 shows the subfields of $\mathbb{F}_{64}$ generated by the polynomial $X^6 + X^5 + 1$ whose numerical rank is 97.
5 Finite Projective Spaces

The projective space PG($n, q$) is the set of non-zero subspaces of $\mathbb{F}_q^{n+1}$ ordered with respect to inclusion. The projective dimension of a subspace is always one less than the vector space dimension. So, a projective point is a vector subspace of dimension one. A projective line is a vector subspace of dimension two, etc. A point is written as $P(x)$ for some vector $x = (x_0, \ldots, x_n)$ with $x_0, \ldots, x_n \in \mathbb{F}_q$, not all zero. For any non-zero element $\lambda \in \mathbb{F}_q$, $P(\lambda x)$ is the same point as $P(x)$. For $a = (a_0, \ldots, a_n) \in \mathbb{F}_q$, not all zero, the symbol $[a]$ represents the line

$$\{P(x) \mid a \cdot x = \sum_{i=0}^{n} a_i x_i = 0\}.$$ 

For any non-zero element $\lambda \in \mathbb{F}_q$, $[\lambda a] = [a]$.

The command

```
orbiter.out -cheat_sheet_PG 3 2
```

creates a report for the projective geometry PG(3, 2). Orbiter has enumerators for points and subspaces of PG($n, q$). The point enumerator allows to represent the points using the integer interval $[0, \theta_n(q) - 1]$, where

$$\theta_n(q) = \frac{q^{n+1} - 1}{q - 1}.$$ 

The points in projective geometry are the one-dimensional subspaces.

The permutation representation of various groups acting on projective space is based on an enumeration of the elements of $\mathfrak{S}_{r_1}$. By enumerating $\mathfrak{S}_{r_1}$ we mean that we choose a fixed bijection between the set $\mathfrak{S}_{r_1}$ and the integer interval $\mathbb{Z}_N = \{0, 1, \ldots, N - 1\}$ where $N = \theta_n(q)$. In order to facilitate the bijection, we enumerate generating vectors. The conditions on the vectors are summarized below:

1. The vector is not the zero vector.
2. The rightmost nonzero entry in the vector is one. If it is not, we normalize the vector so that the rightmost nonzero vector is indeed one. This operation does not change the projective point which is associated with the vector.

The second condition ensures that we list each projective point exactly once. We require two functions, RANK and UNRANK. The function RANK takes a vector $x \in \mathbb{F}_q^n$, not zero, and maps it to the element in $\mathbb{Z}_N$ representing the projective point $P(x)$. A frame in PG($n, q$) is a set of $n + 2$ points, no $n + 1$ in a hyperplane. We assume that the coordinates of a vector are indexed by the elements of $\mathbb{Z}_n$. Also, we let $e_i$ be the $i$-th unit vector. A frame for PG($n, q$) is

$$e_0, \ldots, e_{n-1}, e_0 + \cdots + e_{n-1}.$$ 

This is the standard frame. We start the labeling of points with the standard frame. After these $n + 2$ points, we list the remaining points in lexicographic ordering (of the right-normalized representative). Thus, for PG(2, 2) the ordering is
\[ a = \text{RANK}(x) \quad x = \text{UNRANK}(a) \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(1, 0, 0)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(0, 1, 0)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(0, 0, 1)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(1, 1, 1)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(1, 1, 0)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(1, 0, 1)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(0, 1, 1)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Representatives of the points of PG(2, 2)

\[(1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 1), (1, 1, 0), (1, 0, 1), (0, 1, 1).\]

Let us describe the two functions rank and unrank to perform the actual mappings between PG\((n, q)\) and \(\mathbb{Z}_N\), where \(N = \theta_n(q)\). For this, assume that ranking and unranking functions have already been defined for the elements of the finite field \(\mathbb{F}_q\). Thus, we assume that for \(x \in \mathbb{F}_q\), \(\text{RANK}(\mathbb{F}_q, x)\) is a number \(b\) in \(\mathbb{Z}_q\). Also, for \(b \in \mathbb{Z}_q\), we assume that \(\text{UNRANK}(\mathbb{F}_q, b)\) is the corresponding \(x \in \mathbb{F}_q\). So, we assume that \(\text{RANK}\) and \(\text{UNRANK}\) are mutually inverse functions. Consider the group PGL(3, 2) acting on PG(2, 2), for instance. The points of \(\text{PG}(2, 2)\) are listed in 3.

Let \(V\) be a finite dimensional vector space and let \(\mathcal{G}r_k(V)\) be the Grassmannian of \(k\)-dimensional subspaces of \(V\). If \(\text{dim}(V) = n\), the notation \(\mathcal{G}r_{n,k}\) is used for \(\mathcal{G}r_k(V)\). The size

\begin{algorithm}
1: procedure \text{RANK}(vector : x, field : \mathbb{F}_q, int : n)
2: \quad \text{assert } x \text{ is a nonzero vector in } \mathbb{F}_q^n.
3: \quad \text{if } x = e_i \text{ then}
4: \quad \quad \text{return } i
5: \quad \text{if } x = \text{one} \text{ then}
6: \quad \quad \text{return } n
7: \quad i \leftarrow \max\{j \in \mathbb{Z}_n \mid x_j \neq 0\}
8: \quad x \leftarrow \frac{1}{x_i} x
9: \quad a := 0
10: \quad \text{for } j = i - 1, \ldots, 1, 0 \text{ do}
11: \quad \quad a \leftarrow a + \text{RANK}(\mathbb{F}_q, x_j)
12: \quad \quad \text{if } j > 0 \text{ then}
13: \quad \quad \quad a \leftarrow a \cdot q
14: \quad \quad \text{if } i = n - 1 \text{ and } a \geq \sum_{j=0}^{i-1} q^j \text{ then}
15: \quad \quad \quad a \leftarrow a - 1
16: \quad \quad a \leftarrow a + n - i + \sum_{j=0}^{i-1} q^j
17: \quad \quad \text{return } a
\end{algorithm}
Algorithm 2 Unrank

1: procedure Unrank (int : a, field : $\mathbb{F}_q$, int : n)
2:     assert $a \in \mathbb{Z}_N$ where $N = \theta_{n-1}(q)$.
3:     if $a < n$ then
4:         return $e_a$
5:     $a \leftarrow a - n$
6:     if $a = 0$ then
7:         return one
8:     $a \leftarrow a - 1$
9:     $x \leftarrow 0$
10:    for $i = 1, \ldots, n - 1$ do
11:        if $a \geq \sum_{j=1}^{i-1} q^j$ then
12:            $a \leftarrow a - \sum_{j=1}^{i-1} q^j$
13:        else
14:            $x_i \leftarrow 1$
15:        break
16:    for $k = i + 1, \ldots, n - 1$ do
17:        $x_k \leftarrow 0$
18:    $a \leftarrow a + 1$
19:    if $i = n - 1$ and $a \geq \sum_{j=0}^{i-1} q^j$ then
20:        $a \leftarrow a + 1$
21:    $j \leftarrow 0$
22:    while $a > 0$ do
23:        $r \leftarrow a \mod q$
24:        $x_j \leftarrow \text{Unrank}(\mathbb{F}_q, r)$
25:        $j \leftarrow j + 1$
26:        $a \leftarrow (a - r)/q$
27:    for $h = j, \ldots, i - 1$ do
28:        $x_h \leftarrow 0$
29:    return $x$
of the set $\mathcal{G}_{r_n,k}$ can be computed as
\[
\begin{bmatrix} n \\ k \end{bmatrix}_q = \prod_{i=0}^{k-1} \frac{q^{n-i} - 1}{q^{k-i} - 1}.
\]

using the $q$-binomial coefficient. A system of representatives of the $k$-dimensional subspaces is obtained from the $k \times n$ RREF (row-reduced echelon form) matrices of rank $k$. The elements of $\text{PG}(n-1, q)$ are the $k$-dimensional subspaces of $V = \mathbb{F}_q^n$. It is convenient to identify a subspace with a matrix whose rows contain a basis for it. In coding theory, such a matrix is called a generator matrix.
Interestingly, in PG(n, q) instead of x, the set V, which is often called the variety of a polynomial ideal and to compute a generator for the ideal of a set.

Table 4: The Orbiter ordering of monomials of degree 2, 3 and 4 in a plane

<table>
<thead>
<tr>
<th>h</th>
<th>monomial</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X_0^3</td>
<td>(3, 0, 0)</td>
</tr>
<tr>
<td>1</td>
<td>X_1^3</td>
<td>(0, 3, 0)</td>
</tr>
<tr>
<td>2</td>
<td>X_2^3</td>
<td>(0, 0, 3)</td>
</tr>
<tr>
<td>3</td>
<td>X_0X_1</td>
<td>(2, 1, 0)</td>
</tr>
<tr>
<td>4</td>
<td>X_0^2X_1</td>
<td>(0, 2, 1)</td>
</tr>
<tr>
<td>5</td>
<td>X_0X_1^2</td>
<td>(1, 2, 0)</td>
</tr>
<tr>
<td>6</td>
<td>X_0^2X_2</td>
<td>(0, 0, 2)</td>
</tr>
<tr>
<td>7</td>
<td>X_0X_1X_2</td>
<td>(1, 0, 2)</td>
</tr>
<tr>
<td>8</td>
<td>X_1X_2</td>
<td>(0, 1, 2)</td>
</tr>
<tr>
<td>9</td>
<td>X_0X_1X_2</td>
<td>(1, 1, 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h</th>
<th>monomial</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X_0^4</td>
<td>(4, 0, 0)</td>
</tr>
<tr>
<td>1</td>
<td>X_1^4</td>
<td>(0, 4, 0)</td>
</tr>
<tr>
<td>2</td>
<td>X_2^4</td>
<td>(0, 0, 4)</td>
</tr>
<tr>
<td>3</td>
<td>X_0X_1^3</td>
<td>(3, 1, 0)</td>
</tr>
<tr>
<td>4</td>
<td>X_0^3X_2</td>
<td>(3, 0, 1)</td>
</tr>
<tr>
<td>5</td>
<td>X_0X_1^3</td>
<td>(1, 3, 0)</td>
</tr>
<tr>
<td>6</td>
<td>X_1X_2^3</td>
<td>(0, 3, 1)</td>
</tr>
<tr>
<td>7</td>
<td>X_0X_1^3</td>
<td>(1, 0, 3)</td>
</tr>
<tr>
<td>8</td>
<td>X_1X_2^3</td>
<td>(0, 1, 3)</td>
</tr>
<tr>
<td>9</td>
<td>X_0^2X_2</td>
<td>(2, 2, 0)</td>
</tr>
<tr>
<td>10</td>
<td>X_0X_1X_2</td>
<td>(2, 0, 2)</td>
</tr>
<tr>
<td>11</td>
<td>X_0^2X_2</td>
<td>(2, 2, 0)</td>
</tr>
<tr>
<td>12</td>
<td>X_0X_1X_2</td>
<td>(2, 1, 1)</td>
</tr>
<tr>
<td>13</td>
<td>X_0^2X_2</td>
<td>(1, 2, 1)</td>
</tr>
<tr>
<td>14</td>
<td>X_0X_1X_2</td>
<td>(1, 1, 2)</td>
</tr>
</tbody>
</table>

Table 4: The Orbiter ordering of monomials of degree 2, 3 and 4 in a plane

6 Algebraic Sets

A set of points V in PG(n, q) is algebraic if there is a set of homogeneous polynomials p_1, ..., p_r whose roots over F_q are the given set. In this case, we write V = v(p_1, ..., p_r). The set V is often called the variety of p_1, ..., p_r.

Conversely, given a set of points V in PG(n, q), the ideal I(V) is the set of homogeneous polynomials in F_q[X_0, ..., X_n] which vanish on all of V. This set is an ideal in the polynomial ring. In fact, it is a principal ideal, meaning that it is generated by one element only. Orbiter has ways to compute the variety of a polynomial ideal and to compute a generator for the ideal of a set.

Interestingly, in PG(n, q), every set is algebraic of degree at most (n + 1)(q - 1) [12]. The associated polynomial is unique and known as the algebraic normal form of the set.

Table 4 shows the Orbiter monomial orderings for degrees 2, 3 and 4 in a plane. Suppose we are interested in F_{11} rational points of the elliptic curve y^2 = x^3 + x + 3. We write x^3 + 3 - y^2 + x = 0. Homogenizing yields X^3 + 3Z^3 - Y^2Z + XZ = 0. Using X_0, X_1, X_2 instead of X, Y, Z yields

X_0^3 + 3X_3^3 + 10X_1^2X_2 + X_0X_2^2 = 0.

Using the indexing of monomials from Table 4, we record the following pairs (a, i) where a
Figure 6: Elliptic curve $y^2 \equiv x^3 + x + 3 \mod 11$

is the coefficient and $i$ is the index of the monomial

$$(1, 0), (3, 2), (10, 6), (1, 7).$$

This is concatenated to the sequence $1, 0, 3, 2, 10, 6, 1, 7$. The Orbiter command

```bash
orbiter.out -v 2 -create_combinatorial_object -q 11 -n 2 \
    -projective_variety "EC" 3 "1,0,3,2,10,6,1,7"
```

creates the algebraic set associated to the cubic curve $y^2 = x^3 + x + 3$ in $\text{PG}(2, 11)$. It turns out that there are exactly $18$ points over $\mathbb{F}_{11}$ (cf. Figure 6).

Table 5 shows the Orbiter monomial orderings for degrees $2$ and $3$ in $\text{PG}(3, q)$. The command

```bash
orbiter.out -v 2 -create_combinatorial_object -n 3 -q 4 \
    -projective_variety "Hirschfeld_surface_q4" 3 "1,6,1,8,1,11,1,13" \
    -monomial_type_PART -end \
    -save ""
```

computes the points of the Hirschfeld surface in $\text{PG}(3, 4)$. The equation

$$X_0^2X_3 + X_1^2X_2 + X_1X_2^2 + X_0X_3^2 = 0$$

consists of the monomials $6, 8, 11, 13$ in the partition-based numbering of Table 5. Each has a coefficient of one. Therefore, the equation is encoded as $1, 6, 1, 8, 1, 11, 1, 13$.
<table>
<thead>
<tr>
<th>$h$</th>
<th>monomial</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$X_0^3$</td>
<td>(3, 0, 0, 0)</td>
</tr>
<tr>
<td>1</td>
<td>$X_1^3$</td>
<td>(0, 3, 0, 0)</td>
</tr>
<tr>
<td>2</td>
<td>$X_2^3$</td>
<td>(0, 0, 3, 0)</td>
</tr>
<tr>
<td>3</td>
<td>$X_3^3$</td>
<td>(0, 0, 0, 3)</td>
</tr>
<tr>
<td>4</td>
<td>$X_0^2 X_1$</td>
<td>(2, 1, 0, 0)</td>
</tr>
<tr>
<td>5</td>
<td>$X_0^2 X_2$</td>
<td>(2, 0, 1, 0)</td>
</tr>
<tr>
<td>6</td>
<td>$X_0^2 X_3$</td>
<td>(2, 0, 0, 1)</td>
</tr>
<tr>
<td>7</td>
<td>$X_0 X_1^2$</td>
<td>(1, 2, 0, 0)</td>
</tr>
<tr>
<td>8</td>
<td>$X_1^2 X_2$</td>
<td>(0, 2, 1, 0)</td>
</tr>
<tr>
<td>9</td>
<td>$X_1^2 X_3$</td>
<td>(0, 0, 1, 2)</td>
</tr>
<tr>
<td>10</td>
<td>$X_0 X_2^2$</td>
<td>(1, 0, 2, 0)</td>
</tr>
<tr>
<td>11</td>
<td>$X_1 X_2^2$</td>
<td>(0, 1, 2, 0)</td>
</tr>
<tr>
<td>12</td>
<td>$X_3^2 X_3$</td>
<td>(0, 0, 2, 1)</td>
</tr>
<tr>
<td>13</td>
<td>$X_0 X_3^2$</td>
<td>(1, 0, 0, 2)</td>
</tr>
<tr>
<td>14</td>
<td>$X_1 X_3^2$</td>
<td>(0, 1, 0, 2)</td>
</tr>
<tr>
<td>15</td>
<td>$X_2 X_3^2$</td>
<td>(0, 0, 1, 2)</td>
</tr>
<tr>
<td>16</td>
<td>$X_0 X_1 X_2$</td>
<td>(1, 1, 1, 0)</td>
</tr>
<tr>
<td>17</td>
<td>$X_0 X_1 X_3$</td>
<td>(1, 1, 0, 1)</td>
</tr>
<tr>
<td>18</td>
<td>$X_0 X_2 X_3$</td>
<td>(1, 0, 1, 1)</td>
</tr>
<tr>
<td>19</td>
<td>$X_1 X_2 X_3$</td>
<td>(0, 1, 1, 1)</td>
</tr>
</tbody>
</table>

Table 5: The Orbiter ordering of monomials of degree 2 and 3 in PG(3, $q$)
<table>
<thead>
<tr>
<th>Type</th>
<th>Perm. Domain</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>General linear GL(n, q)</td>
<td>all vectors of (V)</td>
<td>(q^n)</td>
</tr>
<tr>
<td>Affine AGL(n, q)</td>
<td>all vectors of (V)</td>
<td>(q^n)</td>
</tr>
<tr>
<td>Projective PGL(n, q)</td>
<td>(\mathfrak{S}_r(V))</td>
<td>(\frac{q^{n-1}}{q-1})</td>
</tr>
<tr>
<td>Wreath product GL(d, q)≀Sym(n)</td>
<td>(\mathfrak{S}_r((\mathbb{F}_q^d)\otimes^n)) extended</td>
<td>(n + nq^d + \frac{q^{n-1}}{q-1})</td>
</tr>
</tbody>
</table>

Table 6: Basic actions

7 Linear Groups

Orbiter provides support for the classical groups of matrices over finite fields (cf. [28]). Any group is associated with a permutation action. There can be multiple actions for the same group though. New actions can be formed from old actions. Basic group actions are projective, affine, and general linear, as well as orthogonal, unitary and tensor product. Product actions can be defined also. In order to establish a permutation representation, the elements (aka points) of the permutation domain need to be made available. One way would be to make a table of all elements in the permutation domain. However, this would be time and memory intensive. For this reason, a different technique is used that creates points only when needed. The way this works is that the permutation domain is encoded implicitly, using a fixed bijection to a suitable integer interval (zero based), called the domain. Whenever we want the \(i\)th point in the domain, we can call a function that produces it. Conversely, whenever we have a point, we can call a function that tells us what the associated index in the domain. This is facilitated by two mutually inverse functions. The rank function turns a point into an index. The unrank function turns an index in the domain into a point. This way, we avoid storing tables and we make it easier for the computer to store points (namely by storing the associated indices in the domain instead).

Let \(V \simeq \mathbb{F}_q^n\) be a finite dimensional vector space over \(\mathbb{F}_q\). A group \(G\) can act on \(V\) in one of the types listed in Table 6. The elements of finite fields are represented as integers as described in Section 4. The elements of the various sets on which the group acts are encoded as integers. For instance,

```
orbiter.out -linear_group -PGL 4 2 -end
```

creates the group PGL(4, 2) acting on the 15 elements of \(\mathfrak{S}_r(\mathbb{F}_2^4)\). The basic types of groups are listed in Table 7.

For instance,

```
orbiter.out -v 3 -linear_group -PGGL 3 4 -end
```

creates PGL(3, 4) acting on the 21 points of PG(2, 4).

The command

```
orbiter.out -v 3 -linear_group -GL_d_q_wr_Sym_n 2 2 4 -end
```
Table 7: Basic types of Orbiter matrix groups

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>-GL</td>
<td>$n \ q$</td>
<td>GL($n, q$)</td>
</tr>
<tr>
<td>-GGL</td>
<td>$n \ q$</td>
<td>ΓL($n, q$)</td>
</tr>
<tr>
<td>-SL</td>
<td>$n \ q$</td>
<td>SL($n, q$)</td>
</tr>
<tr>
<td>-SSL</td>
<td>$n \ q$</td>
<td>ΣL($n, q$)</td>
</tr>
<tr>
<td>-PGL</td>
<td>$n \ q$</td>
<td>PGL($n, q$)</td>
</tr>
<tr>
<td>-PGGL</td>
<td>$n \ q$</td>
<td>PΓL($n, q$)</td>
</tr>
<tr>
<td>-PSL</td>
<td>$n \ q$</td>
<td>PSL($n, q$)</td>
</tr>
<tr>
<td>-PSSL</td>
<td>$n \ q$</td>
<td>PΣL($n, q$)</td>
</tr>
<tr>
<td>-AGL</td>
<td>$n \ q$</td>
<td>AGL($n, q$)</td>
</tr>
<tr>
<td>-AGGL</td>
<td>$n \ q$</td>
<td>AΓL($n, q$)</td>
</tr>
<tr>
<td>-ASL</td>
<td>$n \ q$</td>
<td>ASL($n, q$)</td>
</tr>
<tr>
<td>-ASSL</td>
<td>$n \ q$</td>
<td>AΣL($n, q$)</td>
</tr>
<tr>
<td>-GL_d_q_wr_Sym_n</td>
<td>$d \ q \ n$</td>
<td>GL($d, q$) $\wr$ Sym($n$)</td>
</tr>
</tbody>
</table>

creates the group GL(2, 2) $\wr$ Sym(4) acting on PG(15, 2) extended by a set of 20 extra points. The extra points are associated with the actions of the components of the wreath product: Four points form a permutation domain for the permutation part Sym(4). An additional $16 = 4 \times 4$ points form four permutation domains of GL(2, 2), one for each factor.

A collineation of a projective space $\pi$ is a bijective mapping from the points of $\pi$ to themselves which preserves collinearity. That is, a collineation $\varphi$ maps any three collinear points $P, Q, R$ to another collinear triple $\varphi(P), \varphi(Q), \varphi(R)$. The collineations form a group with respect to composition, the collineation group. If $M$ is the matrix of an endomorphism, then $\Psi_M$ is the induced map on projective space. By considering the homomorphism $M \mapsto \Psi_M$, the group GL($n+1, q$) of invertible endomorphisms becomes a subgroup of the group of collineations of PG($n, q$). This is the projectivity group PGL($n+1, q$). It is isomorphic to GL($n+1, q$)/$F_q^\times$. Another source of collineations is this: Let $\Phi \in$ Aut($F_q$) be a field automorphism. Then $\Phi$ acts on projective space by sending $P(x)$ to $P(x\Phi)$. This map is another type of collineation, called automorphic collineation. This way, Aut($F_q$) can be considered another subgroup of the group of collineations. If $q = p^h$ for some prime $p$ and some integer $h$ then

$$\Phi_0 : F_q \rightarrow F_q, \ x \mapsto x^p$$

is a generator for the cyclic group $C_h \simeq$ Aut($F_q$). The collineation group of PG($n, q$) ($n \geq 2$) is isomorphic to the semidirect product of the projectivity group and the automorphism group of the field. The collineation group is PGL($n+1, q$) = PGL($n+1, q$) $\rtimes$ Aut($F_q$). We use the following notation for elements of PGL($n+1, q$). Let $\Phi_0$ be a generator for Aut($F_q$) and let $M \in$ GL($n+1, q$). The map

$$(\Psi_M, \Phi_0^k) : \text{PG}(n, q) \rightarrow \text{PG}(n, q), \ P(x) \mapsto P(y), \ y = (x \cdot M)^{\Phi_0^k}$$
is denoted as

\[ M_k. \]  

(1)

The identity element is \( I_0 \), where \( I \) is the identity matrix and 0 is the residue class modulo \( h \). The rules for multiplication and inversion in the collineation group are given as

\[ M_k \cdot N_l = \left( M \cdot N^{\Phi^{-k}} \right)_{k+l}, \]  

(2)

\[ (M_k)^{-1} = \left( \left( M^{-1} \right)^{\Phi^k} \right)_{-k}. \]  

(3)

The affine group \( AGL(n,q) \) is the semidirect product of \( GL(n,q) \) with \( \mathbb{F}_q^n \). The affine semilinear group \( A\Gamma L(n,q) \) is the semidirect product of \( AGL(n,q) \) with \( \text{Aut}(\mathbb{F}_q) \). The elements of \( A\Gamma L(n,q) \) are triples

\[ M_{a,k} := (M, a, k) \in GL(n,q) \times \mathbb{F}_q^n \times \text{Aut}(\mathbb{F}_q), \]

which act on \( \mathbb{F}_q^n \):

\[ (x, (M, a, k)) \mapsto (x \cdot M + a)^{\Phi^k}. \]

The multiplication in \( A\Gamma L(n,q) \) is

\[ M_{a,k} \cdot N_{b,l} = (MN)_{a \Phi^{-k} + b \Phi^{-k}, k+l}. \]

The inverse of an element is

\[ (M_{a,k})^{-1} = \left( \left( M^{-1} \right)^{\Phi^k} \right)_{a \Phi^k, M^{-1}, -k}. \]

A correlation is a one-to-one mapping between the set of points and the set of hyperplanes which reverses incidence. So, if \( \rho \) is a correlation and \( P \) is a point and \( \ell \) is a hyperplane then \( P^{\rho} \) is a hyperplane and \( \ell^{\rho} \) is a point and

\[ \ell^{\rho} \in P^{\rho} \iff P \in \ell. \]

A correlation of order two is called polarity. The standard polarity is the map

\[ \rho : \mathcal{P} \leftrightarrow \mathcal{L}, \ P(x) \leftrightarrow [x]. \]

There are many ways to create subgroups of a group. Table 8 lists some commands to do so. For instance, the command

\texttt{ orbiter.out -v 3 -linear_group -PGL 7 11 -Janko1 -end } \]

creates the first Janko group as a subgroup of \( \text{PGL}(7,11) \).

The command

\texttt{ orbiter.out -v 3 -linear_group -PGL 3 11 -singer 1 -end } \]

19
<table>
<thead>
<tr>
<th>Modifier</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Janko1</td>
<td></td>
<td>first Janko group, needs PGL(7, 11)</td>
</tr>
<tr>
<td>-monomial</td>
<td></td>
<td>subgroup of monomial matrices</td>
</tr>
<tr>
<td>-diagonal</td>
<td></td>
<td>subgroup of diagonal matrices</td>
</tr>
<tr>
<td>-null_polarity_group</td>
<td></td>
<td>null polarity group</td>
</tr>
<tr>
<td>-symplectic_group</td>
<td></td>
<td>symplectic group</td>
</tr>
<tr>
<td>-singer</td>
<td>k</td>
<td>subgroup of index $k$ in the Singer cycle</td>
</tr>
<tr>
<td>-singer_and_frobenius</td>
<td>$k$</td>
<td>subgroup of index $k$ in the Singer cycle, extended by the Frobenius automorphism of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$</td>
</tr>
<tr>
<td>-borel_upper</td>
<td></td>
<td>Borel subgroup of upper triangular matrices</td>
</tr>
<tr>
<td>-borel_lower</td>
<td></td>
<td>Borel subgroup of lower triangular matrices</td>
</tr>
<tr>
<td>-identity_group</td>
<td></td>
<td>identity subgroup</td>
</tr>
<tr>
<td>-subgroup_from_file</td>
<td>$f \ l$</td>
<td>read subgroup from file $f$ and give it the label $l$</td>
</tr>
<tr>
<td>-orthogonal</td>
<td>$\epsilon$</td>
<td>orthogonal group $O^\epsilon(n, q)$, with $\epsilon \in {\pm 1}$ when $n$ is even</td>
</tr>
<tr>
<td>-subgroup_by_generators</td>
<td>$o \ n \ s_1 \ldots s_n$</td>
<td>Generate a subgroup from generators. The label &quot;l&quot; is used to denote the subgroup; $o$ is the order of the subgroup; $n$ is the number of generators and $s_1, \ldots, s_n$ are the generators for the subgroup in string representation.</td>
</tr>
</tbody>
</table>

Table 8: Commands for creating subgroups
<table>
<thead>
<tr>
<th>Modifier</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-wedge</td>
<td></td>
<td>action on the exterior square</td>
</tr>
<tr>
<td>-PGL2OnConic</td>
<td></td>
<td>induced action of PGL(2, q) on the conic in the plane PG(2, q)</td>
</tr>
<tr>
<td>-subfield_structure_action</td>
<td>s</td>
<td>action by field reduction to the subfield of index s</td>
</tr>
<tr>
<td>-on_k_subspaces</td>
<td>k</td>
<td>induced action on k dimensional subspaces</td>
</tr>
<tr>
<td>-on_tensors</td>
<td></td>
<td>induced action of GL(d, q) \text{≀} \text{Sym}(n) on the tensor space</td>
</tr>
<tr>
<td>-on_rank_one_tensors</td>
<td></td>
<td>induced action of GL(d, q) \text{≀} \text{Sym}(n) on the tensor space</td>
</tr>
<tr>
<td>-restricted_action</td>
<td>s</td>
<td>restricted action on the set s</td>
</tr>
</tbody>
</table>

Table 9: Commands for creating induced or restricted group actions

can be used to create the Singer cycle. The Singer cycle in GL(d, q) is a generator for a subgroup of order \( q^d - 1 \). It induces an element of order \( \frac{q^d-1}{q-1} \) on the associated projective geometry PG(d − 1, q). The additional integer parameter \( k \) after the -singer command can be used to create the subgroup of index \( k \) of the Singer cycle.

The command

\[
\text{orbiter.out -v 3 -linear_group -PGL 3 11 -singer_and_frobenius 19 -end}
\]

can be used to create a subgroup of index 19 of the Singer cycle of PG(2, 11), extended by a group of order 3 that arises from the field extension \( \mathbb{F}_{11^3} \) over \( \mathbb{F}_{11} \). The group created by this command has order 21.

It is possible to create new group actions from old. Table 9 lists some commands to do so. For instance, the command

\[
\text{orbiter.out -v 4 \ -linear_group -GL_d_q_wr_Sym_n 2 2 4 -on_tensors -end}
\]

creates the group GL(2, 2) \text{≀} \text{Sym}(4) acting on the 65535 elements of PG(15, 2). By restricting the action to the points of PG(15, 2), the auxiliary points in the permutation domain are hidden.

It is possible to perform group theoretic tasks using the -group_theoretic_activities option. Tables 10-12 list the possible commands.

The command

\[
\text{orbiter.out -v 5 -linear_group -PGL 3 4 -end \ -group_theoretic_activities -find_singer_cycle}
\]
<table>
<thead>
<tr>
<th>Modifier</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-export_gap</td>
<td></td>
<td>Export the group to GAP [11].</td>
</tr>
<tr>
<td>-export_magma</td>
<td></td>
<td>Export the group to Magma [8].</td>
</tr>
<tr>
<td>-orbits_on_subsets</td>
<td>$k$</td>
<td>Compute orbits on $k$-subsets.</td>
</tr>
<tr>
<td>-orbits_on_points</td>
<td></td>
<td>Compute orbits in the action that was created.</td>
</tr>
<tr>
<td>-orbits_of</td>
<td>$i$</td>
<td>Compute orbit of point $i$ in the action that was created.</td>
</tr>
<tr>
<td>-stabilizer</td>
<td></td>
<td>Compute the stabilizer of the orbit representative (needs -orbits_on_points).</td>
</tr>
<tr>
<td>-draw_poset</td>
<td></td>
<td>Draw the poset of orbits (needs -orbits_on_subsets).</td>
</tr>
<tr>
<td>-classes</td>
<td></td>
<td>Compute a report of the conjugacy classes of elements (this command relies on Magma [8]).</td>
</tr>
<tr>
<td>-group_table</td>
<td></td>
<td>Stores the group table as csv-file.</td>
</tr>
<tr>
<td>-centralizer_of_element</td>
<td>label coding</td>
<td>Compute the centralizer of the coded group element, using label to create file names (this command relies on Magma [8]).</td>
</tr>
<tr>
<td>-normalizer</td>
<td></td>
<td>Compute the normalizer (this command relies on Magma [8]; needs a group with a subgroup).</td>
</tr>
<tr>
<td>-report</td>
<td></td>
<td>Produce a latex report about the group.</td>
</tr>
<tr>
<td>-sylow</td>
<td></td>
<td>Include Sylow subgroups in the report (needs -report).</td>
</tr>
<tr>
<td>-print_elements</td>
<td></td>
<td>Produce a printout of all group elements.</td>
</tr>
<tr>
<td>-print_elements_tex</td>
<td></td>
<td>Produce a latex report of all group elements.</td>
</tr>
<tr>
<td>-orbits_on_set_system_from_file</td>
<td>fname $f$ $l$</td>
<td>reads the csv file “fname” and extract sets from columns $[f, ..., f + l - 1]$.</td>
</tr>
<tr>
<td>-orbit_of_set_from_file</td>
<td>fname</td>
<td>reads a set from the text file “fname” and computes orbits on the elements of the set.</td>
</tr>
<tr>
<td>-order_of_products</td>
<td>$g_1 \ldots g_n$</td>
<td>Creates a table of the orders of all products $g_i g_j$, $1 \leq i, j \leq n$.</td>
</tr>
<tr>
<td>-multiply</td>
<td>$s_1$ $s_2$</td>
<td>Creates group elements from $s_1$ and $s_2$ and multiplies.</td>
</tr>
<tr>
<td>-inverse</td>
<td>$s$</td>
<td>Creates a group element from $s$ and computes its inverse.</td>
</tr>
</tbody>
</table>

Table 10: Group theoretic activities (Part 1)
<table>
<thead>
<tr>
<th>Modifier</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-search_element_of_order</td>
<td>$s$</td>
<td>Finds all elements of order $s$ in the group.</td>
</tr>
<tr>
<td>-find_singer_cycle</td>
<td></td>
<td>Finds all Singer cycles whose matrix is a companion matrix.</td>
</tr>
<tr>
<td>-classify_arcs</td>
<td>description</td>
<td>Classify arcs in geometries. See Section 13.</td>
</tr>
<tr>
<td>-linear_codes</td>
<td>$d$</td>
<td>Classify linear codes with redundancy $r$, minimum distance at least $d$ and length at most $n$ using an $r$-dimensional linear group. See Section 17.</td>
</tr>
<tr>
<td>-surface_classify</td>
<td></td>
<td>Classify cubic surfaces. The group must be $\text{PGL}(4,q)$ (or any subgroup) in the wedge action. See Section 14.</td>
</tr>
<tr>
<td>-surface_identify_HCV</td>
<td></td>
<td>Identifies the isomorphism type of the Hilbert Cohn-Vossen surface with parameter $a$. See Section 14.</td>
</tr>
<tr>
<td>-surface_isomorphism_testing</td>
<td>surface-descr-1 surface-descr-2</td>
<td>Computes an isomorphism between two given surfaces or concludes that none exists. See Section 14.</td>
</tr>
<tr>
<td>-surface_recognize</td>
<td>surface-descr</td>
<td>Identifies the isomorphism type of the given surface. See Section 14.</td>
</tr>
<tr>
<td>-classify_surfaces_through_arcs_and_trihedral_pairs</td>
<td></td>
<td>Classifies the surfaces using the associated arcs. The group must be $\text{PGL}(4,q)$ (or any subgroup) in the standard action. See Section 14.</td>
</tr>
<tr>
<td>-create_surface</td>
<td>surface-descr</td>
<td>Creates a surface from a description. See Section 14.</td>
</tr>
</tbody>
</table>

Table 11: Group theoretic activities (Part 2)
<table>
<thead>
<tr>
<th>Modifier</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| -spread_classify         | $k$         | Classifies spreads of $\text{PG}(k-1,q)$ in $\text{PG}(n-1,q)$. The group must be $\text{PGL}(n,q)$ of any subgroup. See Section ??.
| -packing_classify        | $k$         | Classifies packings in $\text{PG}(3,q)$ consisting of spreads whose isomorphism type belongs to the given list. The spreads are stored in files with prefix $\text{path}$. See Section ??.
| -packing_with_assumed_symmetry | description | Classifies packings in $\text{PG}(3,q)$ consisting of spreads whose isomorphism type belongs to the given list. A group of symmetries $H$ is assumed. The normalizer $N$ of $H$ is used to classify the packings. See Section ??.
| -tensor_classify         | $d$         | Classifies tensors of tensor-rank at most $d$.                          |
| -tensor_permutations     |             | Compute the permutation representation of generators of wreath product. |

Table 12: Group theoretic activities (Part 3)

finds all Singer cycles in $\text{PGL}(3,4)$ whose matrix is in the companion matrix form. The first one found is the matrix

$$
\begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
2 & 3 & 2
\end{bmatrix}
$$

whose projective order is 21. Here, we are using the numeric form of field elements, so 2 is $\omega$ and 3 is $\omega + 1$.

The command

```
orbiter.out -v 4 \n   -linear_group -GL_d_q_wr_Sym_n 2 2 4 -on_tensors -end \n   -group_theoretic_activities \n   -orbits_on_points
```

create the group $\text{GL}(2,2) \wr \text{Sym}(4)$ acting on the 65535 elements of $\text{PG}(15,2)$. The command also computes the orbit on points. The latex file $\text{GL}_2_2_\text{wreath}_\text{Sym4}_\text{res65535}_\text{orbits}.tex$ is created. It contains a list of the generators of the group and the orbits.

The quaternion group is a group of order 8 generated by the following matrices over $\mathbb{R}$:

$$
i = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad j = \begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix}, \quad k = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}.
$$
It is isomorphic to a subgroup of $\text{SL}(2,3)$. To perform the embedding, we need to replace the real number $-1$ by the corresponding field element in $\mathbb{F}_3$. Recalling the convention from Section 4 that Orbiter field elements are integers in the interval $[0, \ldots, q - 1]$, we replace $-1$ by 2. The Orbiter command

```
orbiter.out -v 3 -linear_group -SL 2 3 \
    -subgroup_by_generators "quaternion" "8" 3 \
        "1,1,1,2" \ 
        "2,1,1,1" \ 
        "0,2,1,0" \
    -end \ 
    -group_theoretic_activities \ 
    -print_elements_tex \ 
    -group_table \ 
    -report \ 
    -end
```

creates the group. The command produces a list of group elements.

The group of the cube can be created over the field $\mathbb{F}_3$ like so:

```
orbiter.out -v 3 -linear_group -GL 3 3 \
    -subgroup_by_generators "cube" "48" 3 \
        "0,1,0,2,0,0,0,0,1" \ 
        "0,0,1,0,1,0,2,0,0" \ 
        "2,0,0,0,1,0,0,0,1" \
    -end \ 
    -group_theoretic_activities \ 
    -print_elements_tex \ 
    -report
```

The tetrahedral subgroup can be created like so:

```
orbiter.out -v 3 -linear_group -GL 3 3 \
    -subgroup_by_generators "tetra" "12" 2 \
        "0,1,0,0,0,1,1,0,0" \ 
        "0,0,1,2,0,0,2,0,0" \ 
    -end \ 
    -group_theoretic_activities \ 
    -print_elements_tex \ 
    -report
```

Sometimes, the generators depend on specific choices made for the finite field. For instance, if the field if a true extension field over its prime field, the choice of the polynomial matters. This is particularly relevant if generators are taken from other sources. For instance, the electronic Atlas of finite simple groups [29] lists generators for $U_3(3)$ as $3 \times 3$ matrices over the field $\mathbb{F}_9$ using the following short Magma [8] program:
\[ F<\omega> := \text{GF}(9); \]
\[ x := \text{CambridgeMatrix}(1, F, 3, ["164", "506", "851"]); \]
\[ y := \text{CambridgeMatrix}(1, F, 3, ["621", "784", "066"]); \]
\[ G<x, y> := \text{MatrixGroup}<3, F|x, y>; \]

The generators are given using the Magma command \text{CambridgeMatrix}, which allows for more efficient coding of field elements. The field elements are coded as base-3 integers (like in Orbiter) with respect to the Magma version of \( F_9 \). Magma uses Conway polynomials to generate finite fields which are not of prime order. The Conway polynomial for \( F_9 \) can be determined using the following Magma command (which can be typed into the Magma online calculator at [27])

\[
F<\omega> := \text{GF}(9);
\text{print DefiningPolynomial}(F);
\]

which results in

\[ $.1^2 + 2*$.1 + 2 \]

which is the Magma way of printing the polynomial \( X^2 + 2X + 2 \). To have Orbiter use this polynomial, the \texttt{-override_polynomial} option can be used. First, the polynomial is identified with the vector of coefficients \((1, 2, 2)\) which is then read as base-3 representation of an integer as

\[ (1, 2, 2) = 1 \cdot 3^2 + 2 \cdot 3 + 2 = 17. \]

The Orbiter command

\[
\texttt{orbiter.out -v 3 -linear_group -override_polynomial "17" -PGL 3 9 \ -subgroup_by_generators "U_3_3" "6048" 2 \ "1,6,4, 5,0,6, 8,5,1" \ "6,2,1, 7,8,4, 0,6,6" \ -end \ -group_theoretic_activities \ -report -end}
\]

can then be used to create the group. Notice how the generators are encoded almost like in the Magma command, except that commas are used to separate entries.

Through its interface to Magma [8], Orbiter can perform group theoretic computations. For instance, the command sequence
computes the classes of elements in $\text{PGL}(4,4)$ together with the normalizers of the associated cyclic subgroups generated by the class representatives. Note that the orbiter command is repeated twice, with one Magma command in between. Upon the first invocation, the file \texttt{PGGL\_4\_4\_classes.magma} is written. The magma command produces the file \texttt{PGGL\_4\_4\_classes\_out.txt}, which is used during the second invocation of the Orbiter command to create the latex file containing the conjugacy classes in the group. This command sequence may have to be changed depending on the location of magma on the system. Also, the open command to display a pdf file is Macintosh specific.

The command sequence

```
orbiter.out -v 6 \
  -linear_group -PGGL 4 4 -end \
  -group_theoretic_activities "centralizer_of_element" "2A" "1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1, 1" \
/usr/local/magma/magma element_2A_centralizer.magma
```

computes the centralizer of the element

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

which is a Baer involution. The command finds the centralizer to be a group of order 40320.
Orbiter produces a list of generators in coded form, shown below:

```
1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,1,1,
1,0,0,0,1,0,0,0,0,1,0,1,1,0,1,1,1,1,
1,0,0,0,1,0,0,0,0,1,0,0,1,1,1,1,1,1,
1,0,0,0,0,1,0,0,0,0,1,0,1,1,1,1,1,1,0,
1,0,0,0,0,1,0,0,1,1,1,1,1,1,1,1,1,0,
1,0,0,0,0,1,0,0,0,0,1,0,1,1,1,1,1,0,0,
1,0,0,0,0,1,1,0,0,0,1,1,1,1,1,1,0,0,0,
1,0,0,0,0,1,1,0,0,0,1,1,1,1,1,1,0,1,0,
```

The group is isomorphic to $\text{PGL}(4,2).Z_2$, though this structure description cannot be obtained from Orbiter. The generators computed previously can be used to recreate the group using the following command:

```
orbiter.out -v 6 \
- linear_group -PGGL 4 4 \
- subgroup_by_generators "centralizer_2A" "40320" 10 \
  "1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1," \
  "1,0,0,0,0,1,0,0,0,0,1,0,1,1,0,1,1," \
  "1,0,0,0,0,1,0,0,0,0,1,0,1,1,1,1,1," \
  "1,0,0,0,0,1,0,0,0,0,1,0,1,1,1,1,0," \
  "1,0,0,0,0,1,0,0,0,1,0,1,1,1,1,0,0," \
  "1,0,0,0,0,1,1,0,0,0,1,1,1,0,1,0,0," \
  "1,0,0,0,1,1,1,1,1,0,1,0,1,1,1,0,0," \
  "1,0,0,0,0,1,1,0,0,0,1,1,1,0,1,0,0," \
  "0,1,0,0,0,1,0,1,1,1,0,1,0,1,1,1,1," \
- end
```
8 Poset Classification

According to Plesken [22], a group $G$ acts on a poset $\mathcal{P}$ if for all $x, y \in \mathcal{P}$ and all $g \in G$,

$$x \leq y \Rightarrow xg \leq yg.$$  

The orbits of $G$ on $\mathcal{P}$ form another poset, the poset of orbits. Many problems in Combinatorics rely on understanding how certain groups act on certain posets. The information is often contained in structure constants which can be computed from the poset of orbits. Applications lie in the theory of combinatorial designs, where the poset is a part of the subset lattice of a set. In finite geometry, the poset could be an incidence geometry, such as the projective space. For finite posets, Orbiter can help compute things like the poset of orbits and the structure constants associated with it. In order to do so, we must specify the group action and the poset as well as the action on it. Orbiter will apply an algorithm to compute the orbits which is based on earlier work of Schmalz [25]. The algorithm is a bit related to the work of McKay [21] but different. The way in which isomorph checking is handled in the two algorithms is very different. McKay relies on the notion of a function that can compute a canonical form of an object. Such a function is difficult to provide and in most cases relies on expensive backtracking. The Schmalz variant does not require such a function. Instead, it can do isomorphism testing based on precomputed isomorphisms at lower levels in the poset. By induction on the level, the complete orbit or posets is created one level at a time, in a breadth-first manner. On the other hand, McKay’s algorithm proceeds depth-first, and never needs to revisit a node that has been dealt with, resulting in a smaller memory-footprint. A more in-depth comparison of the two algorithms has yet to be done (See [4] for a brief attempt). It seems that there are problems for which the Schmalz variant has an advantage, for instance problems in the finite geometry. On the other hand, McKay’s algorithm performs very well on graphs. The need for a canonical form algorithm somewhat limits the scope of problems for which McKay’s algorithm can be applied. At present, canonical form algorithms are available for very limited classes of objects. McKay’s software package Nauty [19] deals with graphs. Leon’s software Part [15] is devoted to codes and designs. On the other hand, the applications of Orbiter’s classification algorithm are potentially unlimited. At the moment, two types of posets are available. The poset can be a subposet of either the subset-lattice and the subspace-lattice. The way that the poset is defined is by means of a test function. This test function is provided by the user. The purpose of this test function is to determine if an element in the corresponding lattice belongs to the poset or not.

There are many places in Orbiter where the poset classification algorithm is applied. In order to control the behavior of the algorithm, options can be set according to Tables 13-14.
<table>
<thead>
<tr>
<th>Modifier</th>
<th>Args</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-problem_label</td>
<td>str</td>
<td>Use str as a prefix for files that are created.</td>
</tr>
<tr>
<td>-depth</td>
<td>d</td>
<td>Set search depth to d.</td>
</tr>
<tr>
<td>-v</td>
<td>v</td>
<td>Set verbosity to v. Larger numbers mean more output.</td>
</tr>
<tr>
<td>-lex</td>
<td></td>
<td>Use filter based on lexicographic ordering.</td>
</tr>
<tr>
<td>-w</td>
<td></td>
<td>Save orbits at level d.</td>
</tr>
<tr>
<td>-W</td>
<td></td>
<td>Save orbits at all levels.</td>
</tr>
<tr>
<td>-write_data_files</td>
<td></td>
<td>Save data to files.</td>
</tr>
<tr>
<td>-t</td>
<td></td>
<td>Write a file containing the search tree at level d.</td>
</tr>
<tr>
<td>-T</td>
<td></td>
<td>Write a file containing the search tree at all levels.</td>
</tr>
<tr>
<td>-log</td>
<td></td>
<td>Write a log file file at level d.</td>
</tr>
<tr>
<td>-Log</td>
<td></td>
<td>Write a log file file at all levels.</td>
</tr>
<tr>
<td>-x</td>
<td>x</td>
<td>Set x-dimension for drawing.</td>
</tr>
<tr>
<td>-y</td>
<td>y</td>
<td>Set y-dimension for drawing.</td>
</tr>
<tr>
<td>-rad</td>
<td>r</td>
<td>Set radius for orbit nodes in the drawing.</td>
</tr>
<tr>
<td>-findgroup</td>
<td>o</td>
<td>Find all nodes whose stabilizer has order o.</td>
</tr>
<tr>
<td>-draw_poset</td>
<td></td>
<td>Produce a drawing of the poset of orbits.</td>
</tr>
<tr>
<td>-draw_full_poset</td>
<td></td>
<td>Produce a drawing of the full poset with elements grouped by orbits.</td>
</tr>
<tr>
<td>-make_relations_with_flag_orbits</td>
<td></td>
<td>Produce a bitmap drawing of the neighboring relations in the poset with flag orbits.</td>
</tr>
<tr>
<td>-recognize</td>
<td>L</td>
<td>Recognize the given object in the classified list and compute a transporter that maps the given object to the canonical form. Here, L must be a list of integers (comma separated and enclosed in double quotes) encoding an object.</td>
</tr>
</tbody>
</table>

Table 13: Options to influence the poset classification algorithm (Part 1)
<table>
<thead>
<tr>
<th>Modifier</th>
<th>Args</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-plesken</code></td>
<td></td>
<td>Compute Plesken matrices $A_{sup}$ and $A_{inf}$.</td>
</tr>
<tr>
<td><code>-print_data_structure</code></td>
<td></td>
<td>Print the data structure.</td>
</tr>
<tr>
<td><code>-list</code></td>
<td></td>
<td>List orbits at level $d$.</td>
</tr>
<tr>
<td><code>-list_all</code></td>
<td></td>
<td>List orbits at all levels.</td>
</tr>
<tr>
<td><code>-table_of_nodes</code></td>
<td></td>
<td>Produce a spreadsheet of all orbits.</td>
</tr>
<tr>
<td><code>-export_schreier_trees</code></td>
<td></td>
<td>Export all Schreier trees.</td>
</tr>
<tr>
<td><code>-draw_schreier_trees</code></td>
<td><code>args</code></td>
<td>Draw all Schreier trees.</td>
</tr>
<tr>
<td><code>-tools_path</code></td>
<td><code>path</code></td>
<td>Set the tools path.</td>
</tr>
<tr>
<td><code>-scale</code></td>
<td><code>s</code></td>
<td>Set the scale factor for all graphics.</td>
</tr>
<tr>
<td><code>-embedded</code></td>
<td></td>
<td>Produce a fully latexable tikz graphics file. By default, the tikz graphics file will have to be included to another latex file.</td>
</tr>
<tr>
<td><code>-sideways</code></td>
<td></td>
<td>Produce tikz graphics sideways.</td>
</tr>
<tr>
<td><code>-path</code></td>
<td><code>str</code></td>
<td>Use <code>str</code> as a path for files that are created.</td>
</tr>
</tbody>
</table>

Table 14: Options to influence the poset classification algorithm (Part 2)
9 Orbits on Subsets

The lattice of subsets of a set $X$ is $\mathcal{P}(X)$, the set of all subsets of $X$, ordered with respect to inclusion. For instance, Figure 7 shows the lattice of subsets of a 4-element set. Assume that a group $G$ acts on $X$, and hence on the lattice by means of the induced action on subsets. The orbits of $G$ on subsets clump together nodes in the lattice. The set of $G$-orbits form a new poset, the poset of orbits. Poset classification is the process of computing the poset of orbits. Orbiter has an algorithm to perform poset classification. In many cases, we are not interested in the full lattice of subsets $\mathcal{P}(X)$ but rather in a subposet of it. We require that the subposet is closed under the group action and that the following property holds:

$$x, y \in \mathcal{P}(X) \text{ and } x \leq y \Rightarrow (y \in \mathcal{P} \rightarrow x \in \mathcal{P}).$$

The join of two subsets in the poset may or may not belong to the poset. Let us consider the poset of subsets of the 4-element set under the action of a group of order 3. We take the 4 points to be the vectors of $X = \mathbb{F}_2^2$. Let $G$ be the group generated by the Singer cycle in $\text{GL}(2, 2)$, so

$$G = \langle \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \rangle \simeq \langle (0)(1, 3, 2) \rangle,$$

the latter being the permutation representation on the set $X$. Thus, $G$ is a group of order 3 acting with one fixed point. The command

```
orbiter.out -v 3 -linear_group -GL 2 2 -singer 1 -end -group_theoretic_activities -orbits_on_subsets 4 -draw_poset -report
```

computes the orbits of $G$ on the poset of subsets. The poset of orbits is shown in Figure 8. All nodes except for the root node are labeled by elements of $X = \{0, 1, 2, 3\}$. In order to
Figure 8: The poset of orbits under the Singer group

The lex-least spanning tree of a poset is obtained by joining each non-root node to its unique lex-least ancestor. Figure 10 shows the spanning tree for the poset of subsets of a 4 element set.
Figure 9: The poset with orbits indicated by grouping

Figure 10: The lex-least spanning tree for the poset of orbits
Orbits on Subspaces

Orbiter can compute the orbits of a group on the lattice of subspaces of a finite vector space.

The orthogonal group is the stabilizer of a non-degenerate quadric. Suppose we want to classify the subspaces in PG(3, 2) under the action of the orthogonal group. In PG(3, 2) there are two distinct nondegenerate quadrics, \( Q^+(3, 2) \) and \( Q^-(3, 2) \). The \( Q^+(3, 2) \) quadric is a finite version of the quadric given by the equation

\[ x_0x_1 + x_2x_3 = 0, \]

and depicted over the real numbers in Figure 11. PG(3, 2) has 15 points:

- \( P_0 = (1, 0, 0, 0) \)
- \( P_1 = (0, 1, 0, 0) \)
- \( P_2 = (0, 0, 1, 0) \)
- \( P_3 = (0, 0, 0, 1) \)
- \( P_4 = (1, 1, 1, 1) \)
- \( P_5 = (1, 1, 0, 0) \)
- \( P_6 = (1, 0, 1, 0) \)
- \( P_7 = (0, 1, 1, 0) \)
- \( P_8 = (1, 1, 1, 0) \)
- \( P_9 = (1, 0, 0, 1) \)
- \( P_{10} = (0, 1, 0, 1) \)
- \( P_{11} = (1, 1, 0, 1) \)
- \( P_{12} = (0, 0, 1, 1) \)
- \( P_{13} = (1, 0, 1, 1) \)
- \( P_{14} = (0, 1, 1, 1) \)

The \( Q^+(3, 2) \) quadric given by the equation above consists of the nine points

\[ P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_{10}. \]

The quadric is stabilized by the group PGO\(^+(4, 2)\) of order 72. The command

```
orbiter.out -v 5 -linear_group -PGL 4 2 -orthogonal 1 -end \
-groupe_theoretic_activities -orbits_on_subspaces 4 \
-draw_poset
```

produces a classification of all subspaces of PG(3, 2) under PGO\(^+(4, 2)\). The option \(-draw_poset\) creates a Hasse diagram of the classification as shown Figure 12. The Hasse diagram is the poset of orbits of the group on the subspace lattice.
Figure 12: Hasse-diagram for the orbits of the orthogonal group $O^\pm(4, 2)$ on subspaces of PG(3, 2)
<table>
<thead>
<tr>
<th>Key</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-load_from_file</td>
<td>filename</td>
<td>Read a graph from file</td>
</tr>
<tr>
<td>-edge_list</td>
<td>n list-of-edges</td>
<td>Create a graph on n vertices from a list of edges as ranked pairs.</td>
</tr>
<tr>
<td>-edges_as_pairs</td>
<td>n edges-as-pairs</td>
<td>Create a graph on n vertices from a list of edges as pairs.</td>
</tr>
<tr>
<td>-Johnson</td>
<td>n k s</td>
<td>Johnson graph</td>
</tr>
<tr>
<td>-Paley</td>
<td>q</td>
<td>Paley graph</td>
</tr>
<tr>
<td>-Sarnak</td>
<td>p q</td>
<td>Lubotzky-Phillips-Sarnak graph [17]</td>
</tr>
<tr>
<td>-Schlaefli</td>
<td>q</td>
<td>Schlaefli graph</td>
</tr>
<tr>
<td>-Shrikhande</td>
<td></td>
<td>Shrikhande graph</td>
</tr>
<tr>
<td>-Winnie_Li</td>
<td>q i</td>
<td>Winnie-Li graph [16]</td>
</tr>
<tr>
<td>-Grassmann</td>
<td>n k q r</td>
<td>Grassmann graph</td>
</tr>
<tr>
<td>-coll_orthogonal</td>
<td>ϵ d q</td>
<td>Collinearity graph of $O^*(d, q)$</td>
</tr>
</tbody>
</table>

Table 15: Types of graphs

11 Graph Theory

Orbiter can construct certain algebraically defined graphs. It can also construct and classify small graphs and tournaments up to isomorphism. Table 15 shows some Orbiter commands to create graphs. For instance,

```
orbiter.out -v 2 -create_graph -Johnson 5 2 0 -end -save graph_J520.bin
```

creates $J(5, 2, 0)$, also known as the Petersen graph.

```
orbiter.out -v 2 -create_graph -Paley 13 -end -save graph_P13.bin
```

creates the Paley graph of order 13. Very small graphs can be encoded manually. For instance, the graph

![Graph](image)

can be created using the command

```
orbiter.out -v 2 -create_graph -coll_orthogonal 3 4 5 -end -save graph_colOrtho.bin
```
<table>
<thead>
<tr>
<th>Key</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-find_cliques</td>
<td>options</td>
<td>Find all cliques.</td>
</tr>
<tr>
<td>-export_magma</td>
<td></td>
<td>Export to Magma [8].</td>
</tr>
<tr>
<td>-export_maple</td>
<td></td>
<td>Export to Maple [18].</td>
</tr>
<tr>
<td>-export_csv</td>
<td></td>
<td>Export to csv-file.</td>
</tr>
<tr>
<td>-print</td>
<td></td>
<td>Print the graph.</td>
</tr>
<tr>
<td>-sort_by_colors</td>
<td>filename</td>
<td>Sort the vertices by color classes.</td>
</tr>
<tr>
<td>-split</td>
<td>filename</td>
<td>Split the graph.</td>
</tr>
</tbody>
</table>

Table 16: Graph Theoretic Activities

![Petersen graph](image)

Figure 13: Adjacency Matrix of the Petersen graph

orbiter.out -v 2 -create_graph -edges_as_pairs 5 \ 
"0,1,0,2,0,3,0,4,1,3,1,4,2,4" -end

The graph is stored as file `graph_v5_e7.colored_graph`.

Table 16 shows the commands for graph theoretic activities. For instance,

orbiter.out -v 2 -create_graph -Johnson 5 2 0 -end \  
-graph_theoretic_activity -export_csv graph_J520.bin -end
orbiter.out -v 2 -draw_matrix Johnson_5_2_0.csv 20

creates the Petersen graph and writes the adjacency matrix to file. The second command creates a bitmap drawing of the adjacency matrix, shown in Figure 13.

The clique finding command allows for additional commands as shown in Table 17. For instance, the cliques of size 3 in the graph `graph_v5_e7.colored_graph` can be found using

orbiter.out -v 2 -create_graph -load_from_file graph_v5_e7.colored_graph \  
-end -graph_theoretic_activity -find_cliques -target_size 3 -end -end

This command finds three cliques of size 3.

Table 18 lists the commands to classify small graphs and tournaments. For instance,
<table>
<thead>
<tr>
<th>Key</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-rainbow</td>
<td></td>
<td>Find all rainbow cliques. The size of the cliques is the number of vertex colors.</td>
</tr>
<tr>
<td>-target_size</td>
<td>$s$</td>
<td>Find all cliques of size $s$.</td>
</tr>
<tr>
<td>-weighted</td>
<td>$s$</td>
<td>Find weighted cliques.</td>
</tr>
<tr>
<td>-Sajeeb</td>
<td></td>
<td>Use the implementation by Sajeeb Chowdhury.</td>
</tr>
<tr>
<td>-output_file</td>
<td>fname</td>
<td>Write cliques to the named file.</td>
</tr>
<tr>
<td>-restrictions</td>
<td>$l$ $r$ $m$</td>
<td>Restricted search: At level $l$, restrict to all branches congruent to $r$ modulo $m$. Here, $0 \leq r &lt; m$.</td>
</tr>
</tbody>
</table>

Table 17: Clique Finding Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-girth</td>
<td>$d$</td>
<td>Girth at least $d$</td>
</tr>
<tr>
<td>-regular</td>
<td>$r$</td>
<td>Regular of degree $r$</td>
</tr>
<tr>
<td>-no_transmitter</td>
<td></td>
<td>Tournament without transmitter (requires -tournament)</td>
</tr>
</tbody>
</table>

Table 18: Options for classifying graphs
orbiter.out -v 2 -graph_classify -n 4

classifies all graphs with 4 vertices. For tournaments, the option \texttt{-tournament} can be added. For example,

\begin{verbatim}
orbiter.out -v 2 -graph_classify -n 4 -v 2 -tournament -draw_graphs_at_level 6
\end{verbatim}

classifies the tournaments on 6 vertices. The \texttt{-draw_graphs_at_level 6} command instructs Orbiter to draw all representatives at level 6. Figure 14 shows the resulting list of 4 tournaments.
12 Objects in Projective Geometries

Orbiter can create and process objects in projective space. An object can be represented as a set of integers. The integers describe the elements of the object. For instance, an elliptic curve could be represented by the list of ranks of $\mathbb{F}_q$-rational points on it. A spread could be represented by the list of ranks of the subspaces that define it. The `-create_combinatorial_object` command can be used to create objects. The possible secondary commands are shown in Tables 19 and 20. Modifier options can be used to specify certain parameters. They are listed in Table 21. For instance, the following command sequence creates the elliptic curve

$$y^2 \equiv x^3 + x + 3 \mod 11.$$ 

over the field $\mathbb{F}_{11}$:

```
orbiter.out -v 5 -create_combinatorial_object -q 11 \
  -elliptic_curve 1 3 -end \
  -save "./"
```

The curve has 18 points, saved to the file `elliptic_curve_b1_c3_q11.txt`.

Objects in projective geometries can be processed. There are various ways in which this can be done. The available commands are listed in Table 22. The objects that are processed must have been created earlier. For instance, they may be stored in files, or they may be read from the command line. Here is an example. Recall that we have created an elliptic curve and stored the set of $\mathbb{F}_q$-points in a file `elliptic_curve_b1_c3_q11.txt`. Suppose we want to draw the set of points in a graphical representation of the plane. The following command produces the picture shown earlier in Figure 6.

```
orbiter.out -v 2 -process_combinatorial_objects -q 11 -n 2 \
  -draw_points_in_plane EC_11_1_3 \
  -fname_base_out EC_11_1_3 -embedded \
  -input -file_of_points elliptic_curve_b1_c3_q11.txt -end \
  -end
```

In this command, the elliptic curve is read in through the `-input` option. This option defines an input stream. An input stream is a collection of objects defined in files or on the command line that will be processed one-by-one. Table 23 list the commands available to define input streams. For instance, the next command creates an input stream from the single file `elliptic_curve_b1_c3_q11.txt` and then computes the collineation stabilizer of the elliptic curve.

```
orbiter.out -v 2 -canonical_form_PG 2 11 \
  -input -file_of_points elliptic_curve_b1_c3_q11.txt -end \
  -classification_prefix elliptic_curve_b1_c3_q11 \
  -report \
  -end
```
<table>
<thead>
<tr>
<th>Command</th>
<th>Args</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-hyperoval</td>
<td></td>
<td>To create a hyperoval</td>
</tr>
<tr>
<td>-subiaco_oval</td>
<td>f_short</td>
<td>Create the Subiaco oval</td>
</tr>
<tr>
<td>-subiaco_hyperoval</td>
<td></td>
<td>Create the Subiaco hyperoval</td>
</tr>
<tr>
<td>-adelaide_hyperoval</td>
<td></td>
<td>Create the Adalaide hyperoval</td>
</tr>
<tr>
<td>-translation</td>
<td>i</td>
<td>Create the translation hyperoval with exponent $i$</td>
</tr>
<tr>
<td>-Segre</td>
<td></td>
<td>Create the Segre hyperoval</td>
</tr>
<tr>
<td>-Payne</td>
<td></td>
<td>Create the Payne hyperoval</td>
</tr>
<tr>
<td>-Cherowitzo</td>
<td></td>
<td>Create the Cherowitzo hyperoval</td>
</tr>
<tr>
<td>-OKeefe_Penttila</td>
<td></td>
<td>Create the O’Keefe, Penttila hyperoval</td>
</tr>
<tr>
<td>-BLT_database</td>
<td>$k$</td>
<td>Create the $k$th BLT-set of order $q$ from the database ($k = 0, 1, \ldots$)</td>
</tr>
<tr>
<td>-ovoid</td>
<td></td>
<td>Create an ovoid</td>
</tr>
<tr>
<td>-Baer</td>
<td></td>
<td>Create the (standard) Baer subgeometry</td>
</tr>
<tr>
<td>-orthogonal</td>
<td>$\epsilon$</td>
<td>Create the $Q^\epsilon(n,q)$ quadric</td>
</tr>
<tr>
<td>-hermitian</td>
<td></td>
<td>Create the Hermitian variety given by $\sum_{i=0}^{n} X_i^{q+1} = 0$</td>
</tr>
<tr>
<td>-cubic</td>
<td></td>
<td>Create a cubic</td>
</tr>
<tr>
<td>-twisted_cubic</td>
<td></td>
<td>Create a twisted cubic</td>
</tr>
<tr>
<td>-elliptic_curve</td>
<td>$a \ b$</td>
<td>Create the elliptic curve $y^2 = x^3 + ax + b$</td>
</tr>
<tr>
<td>-ttp_construction_A</td>
<td></td>
<td>Create the twisted tensor product code of type A [3]</td>
</tr>
<tr>
<td>-ttp_construction_A_hyperoval</td>
<td></td>
<td>Create the twisted tensor product code of type A [3]</td>
</tr>
<tr>
<td>-ttp_construction_B</td>
<td></td>
<td>Create the twisted tensor product code of type B [3]</td>
</tr>
</tbody>
</table>

Table 19: Orbiter Objects (Part 1)
<table>
<thead>
<tr>
<th>Command</th>
<th>Args</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-unital_XXq_YZq_ZYq</td>
<td></td>
<td>Create the unital with equation $XX^q + YZ^q + ZY^q = 0$</td>
</tr>
<tr>
<td>-desarguesian_line_spread_in_PG_3_q</td>
<td></td>
<td>Create the desarguesian line spread in PG(3, q) as a set of 2-subspaces</td>
</tr>
<tr>
<td>-Buekenhout_Metz</td>
<td></td>
<td>Create the Buekenhout Metz unital</td>
</tr>
<tr>
<td>-Uab</td>
<td>a b</td>
<td>Create the Buekenhout Metz unital in the form of Barwick and Ebert [2]</td>
</tr>
<tr>
<td>-whole_space</td>
<td></td>
<td>Create the whole space</td>
</tr>
<tr>
<td>-hyperplane</td>
<td>pt</td>
<td>Create the hyperplane given by dual coordinates associated with the given point</td>
</tr>
<tr>
<td>-segre_variety</td>
<td>a b</td>
<td>Create the Segre variety</td>
</tr>
<tr>
<td>-Maruta_Hamada_arc</td>
<td></td>
<td>Create the Maruta Hamada arc</td>
</tr>
<tr>
<td>-projective_variety</td>
<td>l d C</td>
<td>Create the projective variety of degree $d$ with label $l$, with coefficient vector $C$</td>
</tr>
<tr>
<td>-projective_curve</td>
<td>l r d C</td>
<td>Create the projective curve of degree $d$ with label $l$, with coefficient vector $C$ in $r$ variables</td>
</tr>
</tbody>
</table>

Table 20: Orbiter Objects (Part 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Args</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-q</td>
<td>q</td>
<td>The size of the finite field $F_q$</td>
</tr>
<tr>
<td>-Q</td>
<td>Q</td>
<td>The field size of the extension field $F_Q$</td>
</tr>
<tr>
<td>-n</td>
<td>n</td>
<td>The projective dimension</td>
</tr>
<tr>
<td>-poly</td>
<td>r</td>
<td>Use polynomial with rank $r$ to create the field $F_q$</td>
</tr>
<tr>
<td>-poly_Q</td>
<td>r</td>
<td>Use polynomial with rank $r$ to create the field $F_Q$</td>
</tr>
<tr>
<td>-embedded_in_PG_4_q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-BLT_in_PG</td>
<td></td>
<td>Create the BLT-set with ranks in PG($n, q$) instead of orthogonal point ranks</td>
</tr>
<tr>
<td>-monomial_type_LEX</td>
<td></td>
<td>Monomials are in lexicographical ordering.</td>
</tr>
<tr>
<td>-monomial_type_PART</td>
<td></td>
<td>Monomials are in partition ordering.</td>
</tr>
</tbody>
</table>

Table 21: Orbiter Objects: Modifiers

43
<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-dualize_hyperplanes_to_points</td>
<td>Turns ranks of hyperplanes into ranks of points</td>
</tr>
<tr>
<td>-dualize_points_to_hyperplanes</td>
<td>Turns ranks of points into ranks of hyperplanes</td>
</tr>
<tr>
<td>-ideal_LEX</td>
<td>Compute the ideal of a set of points, using lexicographic ordering of monomials</td>
</tr>
<tr>
<td>-ideal_PART</td>
<td>Compute the ideal of a set of points, using partition ordering of monomials</td>
</tr>
<tr>
<td>-homogeneous_polynomials.LEX</td>
<td>Prints the equation whose coefficient vector is the input vector using lexicographic ordering of monomials.</td>
</tr>
<tr>
<td>-homogeneous_polynomials.PART</td>
<td>Prints the equation whose coefficient vector is the input vector using partition ordering of monomials.</td>
</tr>
<tr>
<td>-canonical_form</td>
<td>Computes the canonical form of a set</td>
</tr>
<tr>
<td>-draw_points_in_plane</td>
<td>Produces a drawing of a set of points in a projective plane</td>
</tr>
<tr>
<td>-klein</td>
<td>Applies the Klein correspondence</td>
</tr>
<tr>
<td>-line_type</td>
<td>Computes the line type</td>
</tr>
<tr>
<td>-plane_type</td>
<td>Computes the plane type</td>
</tr>
<tr>
<td>-conic_type</td>
<td>Computes the conic type</td>
</tr>
<tr>
<td>-hyperplane_type</td>
<td>Computes the hyperplane type</td>
</tr>
<tr>
<td>-intersect_with_set_from_file</td>
<td>Computes the intersection with a set specified in a file</td>
</tr>
<tr>
<td>-arc_with_given_set_as_s_lines_after_dualizing</td>
<td>Finds arcs with the given set as s-lines</td>
</tr>
<tr>
<td>-arc_with_two_given_sets_of_lines_after_dualizing</td>
<td>Finds arcs with the two given sets as s-lines and t-lines, respectively</td>
</tr>
<tr>
<td>-arc_with_three_given_sets_of_lines_after_dualizing</td>
<td>Finds arcs with the three given sets as s-lines and t-lines and u-lines, respectively</td>
</tr>
</tbody>
</table>

Table 22: Orbiter Jobs in PG(n, q)
<table>
<thead>
<tr>
<th>Command</th>
<th>Args</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-set_of_points</td>
<td>set</td>
<td>A set of point from the command line</td>
</tr>
<tr>
<td>-set_of_lines</td>
<td>set</td>
<td>A set of lines from the command line</td>
</tr>
<tr>
<td>-set_of_packing</td>
<td>set</td>
<td>A set of spreads forming a packing. Each spread is encoded as one integer.</td>
</tr>
<tr>
<td>-file_of_points</td>
<td>fname</td>
<td>A file containing point sets.</td>
</tr>
<tr>
<td>-file_of_lines</td>
<td>fname</td>
<td>A file containing line sets.</td>
</tr>
<tr>
<td>-file_of_packings</td>
<td>fname</td>
<td>A file containing packings. Each packing is given as a sets of spreads.</td>
</tr>
<tr>
<td>-file_of_packings_through_</td>
<td>fname1</td>
<td>A file containing packings. Each packing is given as a sets of spreads.</td>
</tr>
<tr>
<td>spread_table</td>
<td>fname2</td>
<td>Each spread is encoded as one integer. The spreads are stored in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>second file.</td>
</tr>
<tr>
<td>-file_of_point_set</td>
<td>fname</td>
<td>A file containing a point set</td>
</tr>
<tr>
<td>-file_of_designs</td>
<td>fname $N$ $b$ $k$ $s$</td>
<td>A file containing large sets of designs on $N$ points, with $b$ blocks overall, blocks consist of $k$-subset, and with $s$ blocks per design.</td>
</tr>
</tbody>
</table>

Table 23: Orbiter input streams
Orbiter shows that the curve has a collineation stabilizer of order 6, generated by

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}, \quad \begin{bmatrix}
1 & 1 & 8 \\
5 & 9 & 5 \\
8 & 1 & 1 \\
\end{bmatrix}.
\]

This command relies on the Nauty [19] package.
<table>
<thead>
<tr>
<th>Command</th>
<th>Args</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-q</td>
<td>q</td>
<td>Specify the size of the field $\mathbb{F}_q$.</td>
</tr>
<tr>
<td>-d</td>
<td>d</td>
<td>Require that no more than $d$ points lie on a line.</td>
</tr>
<tr>
<td>-n</td>
<td>n</td>
<td>The size of the matrix group.</td>
</tr>
<tr>
<td>-target_size</td>
<td>t</td>
<td>Specify the size of the arc to be $t$.</td>
</tr>
<tr>
<td>-conic_test</td>
<td></td>
<td>Require that no 6 points of the arc lie on a conic.</td>
</tr>
<tr>
<td>-affine</td>
<td></td>
<td>Classify arcs in the affine geometry, assuming that $x_0 = 0$ is the hyperplane at infinity. The condition that no more that $d$ point lie on a line applies to affine lines only.</td>
</tr>
<tr>
<td>-no_arc_testing</td>
<td></td>
<td>Do not test the at most $d$ points per line condition.</td>
</tr>
<tr>
<td>-forbidden_point_set</td>
<td>set</td>
<td>The arc must not contain any of the given points.</td>
</tr>
</tbody>
</table>

Table 24: Commands for Classifying Arcs

13 Arcs in Projective Planes

A $(k, d)$-arc in a projective plane $\pi$ is a set $S$ of $k$ points such that every line intersects $S$ in at most $d$ points. Arcs are related to linear codes and other structures. Two arcs $S_1$ and $S_2$ are equivalent if there is a projectivity $\Phi$ such that $\Phi(A) = B$. The problem of classifying arcs is the problem of determining the orbits of the projectivity group on arcs. At times, we consider the larger group of collineations. In that case, the problem of classifying arcs is the problem of determining the orbits of the collineation group on arcs. Orbiter can solve such classification problems, at least for small parameter cases. Table 24 list the commands available to classify arcs. Here is an example. A hyperoval in a plane $\text{PG}(2, 2^e)$ is a $(2^e + 2, 2)$-arc. It is interesting to classify the hyperovals up to collineation equivalence under the group $\text{PGL}(3, 2^e)$. The command

```
orbiter.out -v 4 \n  -linear_group -PGGL 3 16 -end \n  -group_theoretic_activities \n  -poset_classification_control -problem_label arcs_q16_d2 \n    -W -depth 18 -end \n  -classify_arcs \n    -target_size 18 \n    -q 16 \n    -n 3 \n    -d 2 \n```

47
performs the classification of hyperovals in PG(2, 16). There are exactly two hyperovals in this plane. Orbiter also finds the stabilizers of these arcs. They have orders 16320 and 144, respectively.

It is also possible to classify arcs in affine spaces, though this requires a bit more effort. Here is an example. Suppose we want to classify large arcs in AG(4, 3). To this end, we rely on the embedding of AG(4, 3) in PG(4, 3). The easiest embedding in terms of Orbiter labels is the one where $X_0 = 0$ is the hyperplane at infinity. In this case, the labels of affine lines (i.e. lines that do not lie completely in the hyperplane) are all consecutive and they are listed first in the list of lines of PG(4, 3). We create a cheat sheet about PG(4, 3) using

```
orbiter.out -cheat_sheet_PG 4 3
pdflatex PG_4_3.tex
```

We now have to create the group and forbid the points of the hyperplane. The command

```
orbiter.out -v 5 \
-linear_group -PGL 5 3 \n-subgroup_by_generators "AGL_4_3" "1965150720" 14 \n "1,1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1," \n "1,0,1,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1," \n "1,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1," \n "1,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1," \n "1,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1," \n "1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1," \n "1,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,1," \n "1,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,1," \n "1,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,1," \n "1,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,1," \n "1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1," \n "1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,1," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,1," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,2," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,2," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,2," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,2," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,2," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,2," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1," \n "1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1," \n -end \n-group_theoretic_activities \n-report \n-end
```

does exactly that. Finally, the command

```
orbiter.out -v 5 \
-linear_group -PGL 5 3 \n-subgroup_by_generators "AGL_4_3" "1965150720" 14 \
48
classifies all arcs in the affine space AG(4, 3). The unique largest arc has size 20 [13]. To investigate this arc more closely, the command

```
orbiter.out -v 2 -canonical_form_PG 4 3
  -input -set_of_points
    "0,5,6,8,11,16,19,25,28,43,45,51,57,63,70,99,102,105,107,114"
  -end
  -classification_prefix arc_AG_4_3
  -report
  -end
```

pdflatex arc_AG_4_3_classification.tex

can be used. It shows that the arc has a stabilizer of order 2880. This command also shows a tactical decomposition of the geometry and the arc within which gives structural information about the arc. The output contains a set of generators for the stabilizer. This group can be recreated and explored using the command

```
orbiter.out -v 5
```
This command also exports the group to GAP and Magma.
14 Cubic Surfaces

Orbiter can classify cubic surfaces with 27 lines over finite fields. There are several different approaches to classify cubic surfaces over finite fields with 27 lines under the collineation group $\text{PGL}(4, q)$. One approach is described in [7] and relies on Schlaefli’s notion of a double six as a substructure [24]. Another approach is through non-conical six-arcs in a plane, as described in [14]. Both approaches have been implemented in Orbiter. The purpose of the construction algorithm is to produce the equations of surfaces. In order to do so, the notion of a double six of lines in $\text{PG}(3, q)$ is used. A double six determines a unique surface but a surface may have several double sixes associated to it. The classification algorithm sorts out the relationship between the isomorphism types of double sixes and the isomorphism types of cubic surfaces. In order to classify all double sixes, yet another substructure is considered. These are the five-plus-ones. They consist of 5 lines with a common transversal. The poset classification algorithm is used to classify the five-plus-ones. Orbiter will sort out the isomorphism classes of double sixes based on their relation to the five-plus-ones. In order to classify the five-plus-ones, the related Klein quadric is considered. Lines in $\text{PG}(3, q)$ correspond to points on the Klein quadric. Thus, a five-plus-one configuration of lines corresponds to a certain configuration of points on the Klein quadric. The command

```
orbiter.out -v 3 -linear_group -PGL 4 7 -wedge -end \
  -group_theoretic_activities \
  -poset_classification_control -W -end \n  -surface_classify -end
```

classifies all cubic surfaces over the field $\mathbb{F}_7$ under the projective linear group. If desired, it is possible to use

```
orbiter.out -v 3 -linear_group -PGGL 4 4 -wedge -end \
  -group_theoretic_activities \
  -poset_classification_control -W -end \n  -surface_classify -end
```

to perform the same classification with respect to the collineation group $\text{PGL}(4, 4)$.

A second algorithm to classify cubic surfaces has been described in [6] and in [14]. For instance, the command

```
orbiter.out -v 4 -linear_group -PGGL 4 4 -end \
  -group_theoretic_activities \
  -trihedral_control -problem_label tri1_q4 -end \n  -trihedra2_control -problem_label tri2_q4 -end \n  -control_six_arcs -problem_label sixarcs_q4 -end \n  -classify_surfaces_through_arcs_and_trihedral_pairs \n  -end
```

classifies all cubic surfaces with 27 lines over the field $\mathbb{F}_4$ (there is just one, the Hirschfeld surface). A report of the classification is produced in the file `arc_lifting_q4.tex`. 

51
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-q</code></td>
<td><code>q</code></td>
<td>Specify the order of the field. The surface will be defined in PG(3,q).</td>
</tr>
<tr>
<td><code>-catalogue</code></td>
<td><code>i</code></td>
<td>Create the $i$th surface in the catalogue. Here, $i$ is an index variable used to index all surfaces in PG(3,q). The index $i$ is zero-based.</td>
</tr>
<tr>
<td><code>-by_coefficients</code></td>
<td>list-of-coeff-pairs</td>
<td>Create a surface from a list of coefficient-monomial pairs.</td>
</tr>
<tr>
<td><code>-family_HCV</code></td>
<td><code>a</code></td>
<td>Create the Hilbert, Cohn-Vossen surface with parameter $a$, see [7] (use $b = 1$). The equation is $X_3^3 - b^2(X_0^2 + X_1^2 + X_2^2)X_3 + \frac{b^3}{3}(a^2 + 1)X_0X_1X_2 = 0$.</td>
</tr>
<tr>
<td><code>-arc_lifting</code></td>
<td><code>A</code></td>
<td>Create the surface associated with the arc $a_1, \ldots, a_6$ in PG(2,q) by means of the Clebsch map. Each of the $a_i$ is the rank of a point in PG(2,q). Use the trihedral pair algorithm. Here, $A$ is a comma-separated string containing the numerical ranks of the $P_i$ in PG(2,q).</td>
</tr>
<tr>
<td><code>-arc_lifting_with_two_lines</code></td>
<td><code>A L</code></td>
<td>Create the surface associated with the arc $a_1, \ldots, a_6$ in PG(2,q) by means of the Clebsch map. Each of the $a_i$ is the rank of a point in PG(2,q). Use the two-lines algorithm. Here, $A$ is a comma-separated string containing the numerical ranks of the $P_i$ in PG(3,q) and $L$ is a comma-separated string of the numerical ranks of two lines in PG(3,q). If both of the lines are given as 0, the program will pick a suitable set of lines automatically.</td>
</tr>
<tr>
<td><code>-select_double_six</code></td>
<td><code>L</code></td>
<td>Relabel the lines by choosing the 12 lines in $L$ as new double six. The entries in $L$ are line indices with respect to the old double six. They are integers in the interval $[0, 26]$. This command can be repeated. In each application, the double six refers to the previous labeling.</td>
</tr>
</tbody>
</table>

Table 25: Commands to create a known cubic surface
Besides classification, there are many other ways to create cubic surfaces in Orbiter directly. For this purpose, the commands in Table 25 can be used. There is a built-in catalogue of cubic surfaces with 27 lines for small finite fields \( \mathbb{F}_q \) (all surfaces in fields \( \mathbb{F}_q, q \leq 97 \) are built-in, plus some for larger fields). The command

```bash
orbiter.out -v 3 -linear_group -PGGL 4 4 -wedge -end \
   -group_theoretic_activities \ 
   -create_surface -q 4 -catalogue 0 -end
```

creates the unique cubic surface with 27 lines over the field \( \mathbb{F}_4 \) which is stored under the index 0 in the catalogue.

Another way of creating surfaces is as members of known infinite families. For instance,

```bash
orbiter.out -v 3 -linear_group -PGL 4 13 -wedge -end \
   -group_theoretic_activities \
   -create_surface -family_HCV 3 -q 13 -end
```

creates the member of the Hilbert, Cohn-Vossen surface described in [7] with parameter \( a = 3 \) and \( b = 1 \) over the field \( \mathbb{F}_{13} \).

It is possible to apply a transformation to the surface. Suppose we are interested in the surface over \( \mathbb{F}_8 \) created in (4). The command

```bash
orbiter.out -v 3 -linear_group -PGGL 4 8 -wedge -end \
   -group_theoretic_activities \
   -create_surface -q 8 -catalogue 0 -end \
   -transform_inverse "1,4,4,0,6,0,0,0,6,2,0,1,7,0,4,0,0"
```

creates surface 0 over \( \mathbb{F}_8 \) and applies the inverse transformation to recover the surface whose equation was given in (4). The surface number 0 over \( \mathbb{F}_8 \) is created, and the transformation (5) is applied in inverse. The commands -transform and -transform_inverse accept the transformation matrix in row-major ordering, with the field automorphism as additional element. It is possible to give a sequence of transformations. In this case, the transformations are applied in the order in which the commands are given on the command line. The command

```bash
orbiter.out -v 3 -linear_group -PGGL 4 8 -wedge -end \
   -group_theoretic_activities \
   -create_surface -q 8 -catalogue 0 \ 
   -select_double_six "15,11,22,19,24,5,16,10,23,20,25,4" \ 
   -select_double_six "3,2,1,0,5,4,9,8,7,6,11,10" \ 
   -end \ 
   -transform_inverse "1,4,4,0,6,0,0,0,6,2,0,1,7,0,4,0,0" \ 
   -transform "4,4,0,0, 0,0,1,0, 1,0,0,0, 0,0,0,1, 0"
```

can be used to create the surface number 0 over \( \mathbb{F}_8 \), so that the 13 Eckardt points lie in the plane \( \pi = [0,0,0,1]^\perp \). The double six has been relabeled in such a way that the Clebsch
map $\Phi_{b_1,b_2,\pi}$ sends the lines $a_i$ to $P_i$, for $i = 1, \ldots, 6$. A latex report is written. A closer inspection of the latex report shows that

$$P_1 = (1,0,0), \ P_2 = (0,1,0), \ P_3 = (0,0,1), \ P_4 = (1,1,1), \ P_5 = (2,3,1), \ P_6 = (3,2,1).$$

The command

```
orbiter.out -v 3 -linear_group -PGGL 4 8 -end \ 
-group_theoretic_activities \ 
-control_six_arcs -end \ 
-create_surface -q 8 -arc_lifting "0,1,2,3,28,35" \ 
-end
```

creates the cubic surface associated with the non-conical 6-arc 0, 1, 2, 3, 28, 35 over the field $\mathbb{F}_8$. The arc has been computed in Section 13. This command used trihedral pairs to create the surface from the arc. The automorphism group of the surface is created as well. A latex report about the surface is written.

The command

```
orbiter.out -v 3 -linear_group -PGGL 4 8 -end \ 
-group_theoretic_activities \ 
-control_six_arcs -end \ 
-create_surface -q 8 -arc_lifting_with_two_lines "0,1,2,20,30,37" "24,1898" \ 
-end
```

creates the same cubic surface, using a different algorithm. The arc is now given as elements in $\text{PG}(3,8)$, and the ranks of two lines in $\text{PG}(3,8)$ are given as well. These two lines, $\ell_1$ and $\ell_2$, pass through the first two points of the arc, are not contained in the plane containing the arc, and are skew. This command does not create the automorphism group of the surface and does not write a latex report file.

The `-surface_recognize` option can be used to identify a given surface in the list produced by the classification. For instance,

```
orbiter.out -v 3 -linear_group -PGGL 4 8 -wedge -end \ 
-group_theoretic_activities -surface_recognize -q 8 \ 
-by_coefficients "1,6,1,8,1,11,1,13,1,19" -end -end
```

identifies the surface (cf. Table 5)

$$X_0^2X_3 + X_1^2X_2 + X_1X_2^2 + X_0X_3^2 + X_1X_2X_3 = 0$$

(4)
in the classification of surfaces over the field $\mathbb{F}_8$. This means that an isomorphism from the given surface to the surface in the list is computed. Also, the generators of the automorphism group of the given surface are computed, using the known generators for the automorphism
group of the surface in the classification. For instance, executing the command above creates an isomorphism between the given surface and the surface in the catalogue:

\[
\begin{pmatrix}
1 & 4 & 4 & 0 \\
6 & 0 & 0 & 0 \\
6 & 2 & 0 & 1 \\
7 & 0 & 4 & 0 \\
\end{pmatrix}_0.
\]

Orbiter can compute isomorphism between two given surfaces. The surfaces must have 27 lines. For instance, the command

```
orbiter.out -v 3 -linear_group -PGGL 4 8 -wedge -end \ 
 -group_theoretic_activities -surface_isomorphism_testing \ 
 -q 8 -by_coefficients \ 
 "5,5,5,8,5,9,5,10,5,11,5,12,4,14,4,15,1,18,1,19" -end \ 
 -q 8 -by_coefficients "1,6,1,8,1,11,1,13,1,19" -end
```

computes an isomorphism between the two \( \mathbb{F}_8 \)-surfaces

\[
0 = \alpha^3 X_0^2 X_2 + \alpha X_1^2 X_2 + \alpha^3 X_1^2 X_3 + \alpha^3 X_0^2 X_2 + \alpha^3 X_1 X_2^2 + \alpha^3 X_2 X_3 \\
+ \alpha^2 X_1 X_3^2 + \alpha^2 X_2 X_3^2 + X_0 X_2 X_3 + X_1 X_2 X_3, \\
0 = X_0^2 X_3 + X_1^2 X_2 + X_1 X_2^2 + X_0 X_3^2 + X_1 X_2 X_3.
\]

The isomorphism is given as a collineation:

\[
\begin{pmatrix}
2 & 3 & 0 & 0 \\
7 & 3 & 0 & 0 \\
0 & 0 & 3 & 0 \\
2 & 3 & 2 & 4 \\
\end{pmatrix}_2.
\]

In here, the numerical representation of elements of \( \mathbb{F}_8 \) as integers in the interval \([0, 7]\) is used. The exponent of the Frobenius automorphism is listed as a subscript.
15 Number Theory

In Table 26, some number theoretic commands are shown. For instance,

```
orbiter.out -v 2 -inverse_mod 18059241 58014043
```

computes the inverse of 18059241 modulo 58014043. The command

```
orbiter.out -v 5 -jacobi 2221 7817
```

computes the Jacobi symbol

\[
\left( \frac{2221}{7817} \right).
\]

The denominator \( p \) has to be a positive odd integer.
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-trace</td>
<td>$q$</td>
<td>Computes the absolute trace function for all elements in $\mathbb{F}_q$</td>
</tr>
<tr>
<td>-norm</td>
<td>$q$</td>
<td>Computes the absolute norm function for all elements in $\mathbb{F}_q$</td>
</tr>
<tr>
<td>-jacobi</td>
<td>$a \ p$</td>
<td>Computes the Jacobi symbol $\left( \frac{a}{p} \right)$</td>
</tr>
<tr>
<td>-power_mod</td>
<td>$a \ n \ p$</td>
<td>Raises $a$ to the power $n$ modulo $p$</td>
</tr>
<tr>
<td>-primitive_root</td>
<td>$p$</td>
<td>Computes a primitive root modulo $p$</td>
</tr>
<tr>
<td>-discrete_log</td>
<td>$b \ a \ p$</td>
<td>Computes $n$ such that $a^n \equiv n \mod p$</td>
</tr>
<tr>
<td>-square_root_mod</td>
<td>$a \ p$</td>
<td>Computes a square root of $a$ modulo $p$</td>
</tr>
<tr>
<td>-inverse_mod</td>
<td>$a \ p$</td>
<td>Computes the modular inverse of $a$ modulo $p$</td>
</tr>
<tr>
<td>-sift_smooth</td>
<td>$a \ n$ primes</td>
<td>Computes all smooth numbers in the interval $[a, a + n - 1]$. Smooth means that they factor completely over the list of primes given.</td>
</tr>
<tr>
<td>-solovay_strassen</td>
<td>$a \ n$</td>
<td>Performs $n$ Solovay / Strassen tests on the number $a$</td>
</tr>
<tr>
<td>-miller_rabin</td>
<td>$a \ n$</td>
<td>Performs $n$ Miller / Rabin tests on the number $a$</td>
</tr>
<tr>
<td>-fermat</td>
<td>$a \ n$</td>
<td>Performs $n$ Fermat tests on the number $a$</td>
</tr>
<tr>
<td>-find_pseudoprime</td>
<td>$a \ n_1 \ n_2 \ n_3$</td>
<td>Computes a pseudoprime which survives $n_1$ Fermat tests, $n_2$ Miller Rabin tests, $n_3$ Solovay Strassen tests</td>
</tr>
<tr>
<td>-find_strong_pseudoprime</td>
<td>$a \ n_1 \ n_2$</td>
<td>Computes a pseudoprime which survives $n_1$ Fermat tests and $n_2$ Miller Rabin tests</td>
</tr>
<tr>
<td>-random</td>
<td>$n \ f$ name</td>
<td>Creates $n$ random numbers and writes them to the csv file $f$ name</td>
</tr>
<tr>
<td>-random_last</td>
<td>$n$</td>
<td>Creates $n$ random numbers prints the last one</td>
</tr>
<tr>
<td>-affine_sequence</td>
<td>$a \ b \ p$</td>
<td>Splits the interval $[0, p - 1]$ into affine sequences of the form $x_{n+1} = ax_n + b \mod p$</td>
</tr>
</tbody>
</table>

Table 26: Number Theoretic Commands
16 Cryptography

In Table 27, some cryptographic commands are shown. For instance,

orbiter.out -v 2 -EC_add 11 1 3 "1,4" "1,4"

adds the point (1, 4) on the curve $y^2 = x^3 + x + 3 \mod 11$ to itself. The command

orbiter.out -v 2 -EC_cyclic_subgroup 11 1 3 "1,4"

computes the cyclic subgroup generated by the point (1, 4) on the curve $y^2 = x^3 + x + 3 \mod 11$. The command

orbiter.out -v 2 -EC_points 199 5 7

computes all points on the curve $y^2 = x^3 + 5x + 7 \mod 199$. The command

orbiter.out -v 6 -seed 17 -EC_Koblitz_encoding 199 5 7 67 "147,164" "DEADBEEF"

encode the message “DEADBEEF” on the curve $y^2 = x^3 + 5x + 7 \mod 199$ using the base point (147, 164) and the secret key 67. The $i$th input character is encoded as two points $(R_i, T_i)$ on the curve using the Elgamal scheme. A random round key is generated for each plaintext symbol. As seen in this example, the -seed command can be used to seed the random number generator with an arbitrary integer (here 17). The command

orbiter.out -v 2 -EC_bsgs 199 5 7 "147,164" 212 \ "172,158,45,195,50,22,10,103,55,33,50,22,145,105,31,74,73,155,67,60,25,6"

performs a baby-step-giant-step brute force attack on the ciphertext sequence

$$R_i = (172, 158), (45, 195), (50, 22), (10, 103), (55, 33),$$

$$ (50, 22), (145, 105), (31, 74), (73, 155), (67, 60), (25, 6),$$

using the base point (147, 164) on the curve $y^2 = x^3 + 5x + 7 \mod 199$, assuming a group order of 212. The command

orbiter.out -v 2 -EC_bsgs_decode 199 5 7 "129,176" 212 \ "127,188,51,141,85,29,106,90,41,105,179,71,171,2,16,197,183,72,27,129,37,10" \ "50,179,169,13,153,169,115,116,188,110,176"

performs a decoding of the ciphertext sequence

$$T_i = (127, 188), (51, 141), (85, 29), (106, 90), (41, 105), (179, 71),$$

$$ (171, 2), (16, 197), (183, 72), (27, 129), (37, 10),$$

assuming round keys

$$k_i = 50, 179, 169, 13, 153, 169, 115, 116, 188, 110, 176,$$

using the base point (147, 164) on the curve $y^2 = x^3 + 5x + 7 \mod 199$, and assuming a group order of 212.
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-RSA_encrypt_text</td>
<td>d n b text</td>
<td>Using blocks of b letters at a time, encrypt “text” using RSA with exponent d modulo n</td>
</tr>
<tr>
<td>-RSA</td>
<td>d n</td>
<td>encrypt the given sequence of integers using RSA with exponent d modulo n</td>
</tr>
<tr>
<td>-EC_add</td>
<td>p a b i_1 i_2</td>
<td>On the elliptic curve ( y^2 \equiv x^3 + ax + b \mod p ), add the points with indices ( i_1 ) and ( i_2 ), each given as a pair ( x, y )</td>
</tr>
<tr>
<td>-EC_points</td>
<td>p a b</td>
<td>Computes all points over ( \mathbb{F}_p ) of the elliptic curve ( y^2 \equiv x^3 + ax + b \mod p )</td>
</tr>
<tr>
<td>-EC_multiple_of</td>
<td>p a b pt n</td>
<td>Computes the ( n ) fold multiple of the given point ( pt ) on the elliptic curve ( y^2 \equiv x^3 + ax + b \mod p )</td>
</tr>
<tr>
<td>-EC_cyclic_subgroup</td>
<td>p a b pt</td>
<td>Computes the cyclic subgroup generated by the given point ( pt ) on the elliptic curve ( y^2 \equiv x^3 + ax + b \mod p )</td>
</tr>
<tr>
<td>-EC_Koblitz_encoding</td>
<td>p a b s pt plain</td>
<td>Computes the Koblitz encoding of “plain” (all caps) on the elliptic curve ( y^2 \equiv x^3 + ax + b \mod p ) using the base point ( pt ) and the secret exponent ( s )</td>
</tr>
<tr>
<td>-EC_bsgs</td>
<td>p a b pt n cipher</td>
<td>Prepare the baby-step giant-step tables for the ciphertext “cipher” on the elliptic curve ( y^2 \equiv x^3 + ax + b \mod p ) using the base point ( pt ) of order ( n )</td>
</tr>
<tr>
<td>-EC_bsgs_decode</td>
<td>p a b pt n cipher round-keys</td>
<td>Decodes the ciphertext “cipher” on the elliptic curve ( y^2 \equiv x^3 + ax + b \mod p ) using the base point ( pt ) of order ( n ) and the round keys “keys”</td>
</tr>
<tr>
<td>-EC_discrete_log</td>
<td>p a b pt base-pt</td>
<td>Computes the elliptic curve discrete log analogue of ( pt ) with respect to ( base-pt ) on the elliptic curve ( y^2 \equiv x^3 + ax + b \mod p )</td>
</tr>
</tbody>
</table>

Table 27: Cryptographic Commands
17 Coding Theory

Orbiter can classify linear codes with prescribed minimum distance. Recall that a linear $[n, k; q]$-code $C$ is a $k$-dimensional subspace of $\mathbb{F}_q^n$. The Hamming space $H(n, q)$ is the space $\mathbb{F}_q^n$ equipped with the Hamming metric. For two vectors $x, y \in \mathbb{F}_q^n$, the Hamming distance is

$$d(x, y) = \# i : x_i \neq y_i.$$ 

The weight of a vector is

$$\text{wt}(x) = d(x, 0) = \# i : x_i \neq 0.$$ 

The minimum distance of a subset $C$ of $H(n, q)$ is

$$d(C) = \min_{c, c' \in C, c \neq c'} d(c, c').$$

We say that $C$ is a linear $[n, k, d; q]$-code if $C$ is $[n, k; q]$ with $d(C) = d$. The minimum weight of a subset $C$ of $H(n, q)$ is

$$\text{wt}(C) = \min_{c \in C \setminus \{0\}} \text{wt}(c).$$

For a linear code $C$, the two quantities agree, and we have

$$d(C) = \text{wt}(C) = \min_{c \in C \setminus \{0\}} \text{wt}(c).$$

It is desirable to have linear codes for which both $k$ and $d$ are reasonably large with respect to $n$. However, these are contradicting aims: A large value of $k$ limits $d$ from above. Likewise, a large value of $d$ limits $k$. It is interesting to have construction methods for linear codes which bound $d$ from below. This is one of the main topics of coding theory.

There is a notion of isometry with respect to the Hamming metric. This leads to a notion of equivalence of codes. Two codes are equivalent if the coordinates of the vectors in one code can be computed (simultaneously) so as to obtain the second code. The automorphism group is the set of isometry maps from one code to itself.

In Table 28, some coding theoretic commands of Orbiter are shown. For instance, the command

```
orbiter.out -BCH 15 2 3
```

creates a binary BCH-code of length 15 with minimum distance at least 3. The generator
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-RREF</td>
<td>$q \ m \ n$ list-of-integers</td>
<td>Compute the RREF of the $m \times n$ matrix over $\mathbb{F}_q$</td>
</tr>
<tr>
<td>-nullspace</td>
<td>$q \ m \ n$ list-of-integers</td>
<td>Compute a basis for the right nullspace of the given $m \times n$ matrix</td>
</tr>
<tr>
<td>-normalize_from_the_right</td>
<td>$q \ m \ n$ list-of-integers</td>
<td>Normalizes the result of -RREF or nullspace from the right</td>
</tr>
<tr>
<td>-weight_enumerator</td>
<td>$q \ m \ n$ list-of-integers</td>
<td>Computes the weight enumerator of the linear code generated by the given $m \times n$ matrix</td>
</tr>
<tr>
<td>-BCH</td>
<td>$n \ q \ t$</td>
<td>Creates the BCH-code of length $n$ over the field $\mathbb{F}_q$ with designed distance $t$</td>
</tr>
<tr>
<td>-Hamming_graph</td>
<td>$n \ q$</td>
<td>Creates the distance matrix of the Hamming graph $H(n, q)$. The vertices are the elements of $\mathbb{F}_q^n$, and the $i,j$-entry is the distance between the vectors whose affine ranks are $i$ and $j$, respectively. The matrix is written as csv-file.</td>
</tr>
<tr>
<td>-draw_matrix</td>
<td>csv-file $w$</td>
<td>Creates a colored bitmap drawing of the matrix in the csv file, using $w$ pixels per entry. The color indicates the value of the matrix entry.</td>
</tr>
</tbody>
</table>

Table 28: Coding Theoretic Commands
matrix produced by this command is

\[
\begin{bmatrix}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1
\end{bmatrix}
\]

The commands

```
orbiter.out -Hamming_graph 4 2
orbiter.out -draw_matrix Hamming_n4_q2.csv 20
```

create the csv-file `Hamming_n4_q2.csv` and produce the bitmap file `Hamming_n4_q2_draw.bmp` shown in Figure 15.

The classification problem of optimal codes in coding theory is the problem of determining the equivalence classes of codes for a given set of values of \( n \) and \( k \) and \( q \) with a lower bound on \( d \). Orbiter can be used for solving this problem for small instances.

Orbiter can be used to classify linear codes with given redundancy and bounded minimum distance. The redundancy of a linear \([n,k]\) code is the parameter \( r = n - k \). Codes with redundancy \( r \) can be identified with subsets of \( \text{PG}(r-1, q) \). Under this correspondence, a code with minimum distance at least \( d \) corresponds to a subset such that any \( d-1 \) elements are independent. We use the notation \( \Lambda_{r-1,s}(q) \) to denote the poset of subsets of \( \text{PG}(r-1, q) \) for which any \( d-1 \)-subset (if any) is independent. Under the correspondence, the action of \( \text{PGL}(r, q) \) on \( \Lambda_{r-1,s}(q) \) corresponds to the orbits of equivalent linear codes. For this reason, we are interested in determining the orbits of \( \text{PGL}(r, q) \) on \( \Lambda_{r-1,s}(q) \). An orbit of size \( n \) represents an isometry class of \([n, n - r, d; q]\) codes with \( d \geq s + 1 \). The projective stabilizer of the subset is the automorphism group of the code. The Orbiter command

```
orbiter.out -v 3 -linear_group -PGL 4 2 -end \
  -group_theoretic_activities -poset_classification_control \
  -W -problem_label codes_r4_d4 -end -linear_codes 4 100 -end
```

can be used to classify linear codes with redundancy 4 and minimum distance at least 4. The extremal code that satisfies these conditions is the \([8, 4, 4]\) codes over \( \mathbb{F}_2 \). The Orbiter program confirms that there is exactly one such code. Orbiter computes the code together with the projective stabilizer. To do so, we first create the group \( \text{PGL}(4, 2) \) acting on the
Figure 15: The color-coded distance matrix of the Hamming graph $H(4,2)$

Figure 16: Orbits of PGL$(4,2)$ on the poset $\Lambda_{3,3}(2)$
poset $\Lambda_{3,3}(2)$. Orbiter then produces the poset of orbits shown in Figure 16. In this diagram, the numbers stand for Orbiter ranks of points in $\text{PG}(3, 2)$. All nodes except for the root node have a number attached to it. The nodes represent subsets. In order to determine the set associated to a node, follow the path from the root node to the node and collect the points according to their labels. The root node represents the empty set. The $[8, 4, 4; 2]$-code is represented by the set \{0, 1, 2, 3, 8, 11, 13, 14\}. The fact that there is only one node at level 8 in the poset of orbits tells us that the code is unique up to equivalence. Let us look at the code. The elements of the set \{0, 1, 2, 3, 8, 11, 13, 14\} are points in $\text{PG}(3, 2)$. We write the coordinate vectors in the columns of a matrix $H$:

$$H = \begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 
\end{bmatrix}.$$ 

This matrix is the parity check matrix $H$ of the code $C$. This means that the words of the code are the vectors $c$ such that $c \cdot H^\top = 0$. Observe that the vectors that we put in the columns of $H$ all have odd weight. They are in fact the points of the hyperplane $x + y + z + w = 0$. This shows that the stabilizer of the code which is the stabilizer of the set is equal to $\text{AGL}(3, 2)$, a group of order 1344.
18 Diophantine Systems

Diophantine systems of equations and inequalities arise frequently in Combinatorics. Suppose we want all partitions of an integer \( n \) as

\[
n = a_1 + a_2 + \ldots + a_k, \quad a_1 \geq a_2 \geq \cdots \geq a_k \geq 1,
\]

with \( a_i \in \mathbb{Z}_{>0} \). For \( 1 \leq j \leq n \), let

\[
c_j = \# \{ i \mid a_i = j \}.
\]

The following diophantine equation holds for any partition

\[
\sum_{j=1}^{n} j c_j = n \tag{6}
\]

Conversely, any partition is uniquely determined by a solution of this equation. Therefore, counting partitions of \( n \) is the same as counting nonnegative integer solutions of (6). Let \( p_n \) be the number of partitions of \( n \). Suppose we wish to compute \( p_{10} \). In this case, the extended coefficient matrix of the system is

\[
\begin{bmatrix}
10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\
\end{bmatrix},
\]

Orbiter creates this diophantine system using the command

```
orbiter.out -v 4 -diophant -label part10 \ 
  -coefficient_matrix 1 10 "10,9,8,7,6,5,4,3,2,1" \ 
  -RHS "10,10,1" -x_min_global 0 -x_max_global 10
```

It also creates three csv-files: one for the coefficient matrix, one for the RHS information, and one for the bounds on the variables. With these files, it is possible to recreate the diophantine system using the command

```
orbiter.out -v 4 -diophant -label part10 \ 
  -coefficient_matrix_csv part10_coeff_matrix.csv \ 
  -RHS_csv part10_RHS.csv \ 
  -x_bounds_csv part10_x_bounds.csv
```

The command

```
orbiter.out -v 4 -diophant_activity -input_file part10.diophant -solve_mckay
```

solves the system and finds that \( p_{10} = 42 \). The sequence \( p_n \) is recorded under the key A000041 in Sloane’s Handbook of integer sequences [26].

A linear space is a pair \((S, \mathcal{L})\) where \( S \) is a set and \( \mathcal{L} \) is a set of subsets of \( S \) such that each set \( L \in \mathcal{L} \) satisfies \(|L| \geq 2\) and moreover for any two \( a, b \in S \) there is exactly one element \( L \in \mathcal{L} \) such that both \( a \) and \( b \) belong to \( L \). The usual notions of isomorphism and automorphism apply. For finite linear spaces, a first combinatorial property is the number \( a_i \) which counts
the number of sets $L \in \mathcal{L}$ of size $i$. The vector $(a_2, \ldots, a_n)$ is the line type of $(S, \mathcal{L})$. The equation

$$\binom{n}{2} = \sum_{j=2}^{n} a_j \binom{j}{2}$$

(7)

is satisfied. The equation can be used to generate all possible line types of a putative linear space. Here is an example. For $|S| = 6$, (7) becomes

$$x_0 \binom{6}{2} + x_1 \binom{5}{2} + x_2 \binom{4}{2} + x_3 \binom{3}{2} + x_4 \binom{2}{2} = \binom{6}{2}.$$

Here, $(x_0, x_1, \ldots, x_4)$ is the line type of a putative linear space on 6 points. That is, $x_i = a_{6-i}$ is the number of lines of size $6 - i$. The extended coefficient matrix of the system is

$$\begin{bmatrix} 15 & 10 & 6 & 3 & 1 & | & 15 \end{bmatrix}.$$  

The Orbiter command

```
orbiter.out -v 4 -diophant -label linsp6 \n    -coefficient_matrix 1 5 "15,10,6,3,1" -RHS "15,15,1" \n    -x_min_global 0 -x_max_global 15
```

creates this system and stores it in the file `linsp6.diophant` using the name specified. The command

```
orbiter.out -v 3 -diophant_activity -input_file linsp6.diophant -solve_mckay
```

solves the system using McKay’s program `possolve` [20]. The program finds 15 solutions, written to the file `linsp6.sol`. In Table 29, Orbiter commands for creating diophantine systems are shown. In Table 30, Orbiter activities for diophantine systems are shown.

Let us consider a problem from [5]. Suppose we are interested in linear spaces on 30 points with line type $(7, 5^{27}, 4^{24})$. This notation means that we assume one 7-lines, 27 5-lines and 24 4-lines. The type of a point $P$ is the set of integers

$$p_j = \#j\text{-lines through } P.$$  

We are trying to precompute the matrix of point types

$$(p_{ij})$$

where $j = 7, 5, 4$ and $i$ belongs to an index set of all possible point types. Fixing a point $P$, counting points $Q \neq P$ collinear with $P$ yields

$$6p_7 + 4p_5 + 3p_4 = 29, \quad p_7 \leq 1, \ p_5 \leq 27, \ p_4 \leq 24.$$ 

Using the Orbiter commands
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-label</td>
<td>label</td>
<td>Use the given name as file name.</td>
</tr>
<tr>
<td>-coefficient_matrix</td>
<td>m n list-of-integers</td>
<td>Set the $m \times n$ coefficient matrix.</td>
</tr>
<tr>
<td>-coefficient_matrix_csv</td>
<td>fname</td>
<td>Read the coefficient matrix from the given csv-file.</td>
</tr>
<tr>
<td>-RHS</td>
<td>list-of-integers</td>
<td>3n values: (RHS-low, RHS-high, RHS-type) for each row of the system.</td>
</tr>
<tr>
<td>-RHS_csv</td>
<td>fname</td>
<td>Read the RHS information from the given csv file.</td>
</tr>
<tr>
<td>-RHS_constant</td>
<td>low,high,type</td>
<td>Set the RHS according to low,high,type.</td>
</tr>
<tr>
<td>-x_max_global</td>
<td>a</td>
<td>Set the upper bound for all variables to $a$</td>
</tr>
<tr>
<td>-x_min_global</td>
<td>a</td>
<td>Set the lower bound for all variables to $a$</td>
</tr>
<tr>
<td>-x_bounds</td>
<td>list-of-values</td>
<td>Set the lower and upper bounds for all variables.</td>
</tr>
<tr>
<td>-x_bounds_csv</td>
<td>fname</td>
<td>Read the lower and upper bounds for all variables from the given file.</td>
</tr>
<tr>
<td>-has_sum</td>
<td>s</td>
<td>For the sum of the variables to be $s$.</td>
</tr>
<tr>
<td>-maximal_arc</td>
<td>s d secants subset</td>
<td>Create system for a maximal arc of size $s$ and degree $d$ in $\mathrm{PG}(2,q)$. Use the given set of two pencil lines. The subset picks the lines from the given pencils which are external.</td>
</tr>
<tr>
<td>-q</td>
<td>q</td>
<td>Use $\mathrm{PG}(2,q)$ for maximal arcs.</td>
</tr>
<tr>
<td>-override_polynomial</td>
<td>a</td>
<td>Use polynomial numerically coded as $a$ for creating $F_q$.</td>
</tr>
</tbody>
</table>

Table 29: Orbiter Commands to create Diophantine systems
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-input_file</td>
<td>file</td>
<td>Specify the input file</td>
</tr>
<tr>
<td>-print</td>
<td></td>
<td>Print the system</td>
</tr>
<tr>
<td>-solve_mckay</td>
<td></td>
<td>Solve the system using McKay’s pos-solve</td>
</tr>
<tr>
<td>-solve_standard</td>
<td></td>
<td>Solve the system using the standard solver</td>
</tr>
<tr>
<td>-draw</td>
<td></td>
<td>Produce a metapost drawing of the coefficient matrix of the system</td>
</tr>
<tr>
<td>-draw_as_bitmap</td>
<td>w b</td>
<td>Produce a bitmap drawing of the coefficient matrix of the system, using boxes of w pixels with. Set the color bit-depth to b (b = 8 or b = 24). The output is a bmp-file.</td>
</tr>
<tr>
<td>-perform_column_reductions</td>
<td></td>
<td>Eliminate variables which must be zero and write a reduced system</td>
</tr>
<tr>
<td>-test_single_equation</td>
<td></td>
<td>For each row of the system, compute the number of solutions of the system restricted to the nonzero coefficients.</td>
</tr>
<tr>
<td>-project_to_single_equation_and_solve</td>
<td>i j</td>
<td>Solve the system assuming the jth solution to the restricted system consisting of the ith row.</td>
</tr>
<tr>
<td>-project_to_two_equations_and_solve</td>
<td>i j r m</td>
<td>Solve the system assuming any solution to the restricted system consisting of the ith and the j-th row whose number is congruent to r mod m.</td>
</tr>
</tbody>
</table>

Table 30: Orbiter activities for Diophantine systems
we determine the possibilities

\[ (p_7, p_5, p_4) = \begin{cases} 
1 & 5 & 1 \\
1 & 2 & 5 \\
0 & 5 & 3 \\
0 & 2 & 7 
\end{cases} \]

The rows in this matrix are called the point types \((i = 0, 1, 2, 3)\). Let \(b_i\) be the number of points of type \(i\). By counting points, incident (point,line) pairs by \(j\)-lines and pairs of intersecting \(j\)-lines, we arrive at the following system:

\[
\begin{align*}
    b_0 + b_1 + b_2 + b_3 &= 30 \\
    b_0 + b_1 &= 7 \\
    5b_0 + 2b_1 + 5b_2 + 2b_3 &= 135 = 27 \cdot 5 \\
    b_0 + 5b_1 + 3b_2 + 7b_3 &= 96 = 24 \cdot 4 \\
    10b_0 + b_1 + 10b_2 + b_3 &\leq 351 = \binom{27}{2} \\
    10b_1 + 3b_2 + 21b_3 &\leq 276 = \binom{24}{2}
\end{align*}
\]

Using the Orbiter commands

\[
\begin{align*}
\text{orbiter.out -v 4 -diophant -label linsp30_pt_distribution} \ \\
\text{-coefficient_matrix 6 4 "1,1,1,1,1,0,0,0,5,2,5,2,1,5,3,7,10,1,10,1,0,10,3,21"} \ \\
\text{-RHS "30,30,1,7,7,1,135,135,1,96,96,1,0,351,2,0,276,2"} \ \\
\text{-x_min_global 0 -x_max_global 30}
\end{align*}
\]

we determine the possibilities

\[
(b_0, b_1, b_2, b_3) = \begin{cases} 
2 & 5 & 23 & 0 \\
3 & 4 & 22 & 1 \\
4 & 3 & 21 & 2 \\
5 & 2 & 20 & 3 \\
6 & 1 & 19 & 4 \\
7 & 0 & 18 & 5 
\end{cases} \]
19 Design Theory

We use the convention of design theory that the incidence matrix of a design has rows indexed by points and columns indexed by blocks (also called lines). A decomposition is a partition of the points and blocks of the geometry such that each class consists either exclusively of points or exclusively of blocks.

A decomposition is point-tactical if for all points, the number of incident lines in the \( j \)th block class depends only on the class of the point. If the point belongs to class \( i \), this number is denoted as \( a_{ij} \). A decomposition is block-tactical if for all blocks, the number of incident points in the \( i \)th point class depends only on the class of the block. If the block belongs to class \( j \), this number is denoted as \( b_{ij} \).

A projective plane of order \( n \) is a design with \( n^2 + n + 1 \) points and equally many blocks (also called lines), each of size \( n + 1 \) such that any two points lie in exactly one block and any two blocks have exactly one point in common. Projective planes are known to exist for all \( n = q \) which are a power of a prime. This follows from a construction which utilizes the projective geometry \( \text{PG}(2, q) \). Points are the one-dimensional subspaces of \( \mathbb{F}_q^3 \), blocks are the two-dimensional subspaces of \( \mathbb{F}_q^3 \), and incidence is natural (inclusion of subspaces). The automorphism group of this design is the collineation group of the projective space. Projective planes other than these exist, though none are known when \( n \) is not a prime power. The number of lines through a point equals the number of points on a line. The fact that these numbers exist imply that there is a tactical decomposition. Namely, the trivial decomposition with two classes, one containing all points and one containing all lines. The structure constants of the decomposition are the numbers just described.

The command

```
orbiter.out -v 8 -create_design -q 3 -family PG_2_q -end
```

creates the design \( \text{PG}(2, 3) \). The blocks of the design are encoded in the lexicographic ordering of \( k \)-subsets (here \( k = 4 \)) as

\[
\{19, 79, 126, 219, 256, 284, 371, 392, 465, 541, 619, 627, 653\}.
\]

It also creates the automorphism group of the design (of order 5616). The group is created as matrix group \( \text{PGL}(3, 3) \), defined using strong generators:

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 2
\end{bmatrix},
\begin{bmatrix}
1 & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 1
\end{bmatrix},
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 1 & 1
\end{bmatrix},
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 1 & 1
\end{bmatrix},
\begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 1
\end{bmatrix}
\]
The program also prints out the tactical decomposition schemes of the design, which are

\[
\begin{array}{c|c}
\rightarrow & 13_1 \\ \hline
13_0 & 4
\end{array}
\quad \begin{array}{c|c}
\downarrow & 13_1 \\ \hline
13_0 & 4
\end{array}
\]
20 Linear Spaces and Tactical Decompositions

Suppose we want to study the projective plane of order 16. So, the geometry is a linear space with $16^2 + 16 + 1 = 273$ points and equally many lines. Each point lies on 17 lines and each line contains 17 points. Any two points lie on exactly one line and any two lines intersect in exactly one point. Of course, the linear space could be the desarguesian plane $PG(2, 16)$, but it could also be any of the other projective planes of order 16. At this point, we are only working with the parameters of the geometry as a linear space, and the isomorphism type of the plane is as yet undecided.

We decide to study maximal arcs of degree 4 in this plane (the degree has to divide the order of the plane). A maximal arc of degree $d$ is a set of points so that each line intersects in either $d$ or zero points. A line which intersects in $d$ points is called a secant. A line which intersects in no point is called an external line. The command

```
orbiter.out -v 4 -maximal_arc_parameters 16 4
```

creates a decomposition stack for the parameters of the arc and writes the file `max_arc_q16_r4.stack`

```
<HTDO type=pt ptanz=2 btanz=2 fuse=simple>
  221 52
  52 17 0
  221 13 4

  1 1
</HTDO>
```

This is a point-tactical decomposition with 2 point-classes and 2 block-classes. The point classes are associated with the rows. The block-classes are associated with the columns. The first row and column indicates the size of the classes. The entries $a_{ij}$ count the number of blocks in the column class $j$ that are incident with a given point in the $i$th row class. The fuse information at the bottom (1 1) is a partition of the row classes which indicates the ancestor decomposition which was column tactical. The next step is to convert the stack file to a tdo file. The command

```
orbiter.out -v 4 -convert_stack_to_tdo max_arc_q16_r4.stack
```

does that. It creates the file `max_arc_q16_r4.tdo`. It also prints the decomposition stack:

```
lambda_scheme at level 2 :
is 1 x 1
  | 273{ 1}
=============
273{ 0} |

row_scheme at level 4 :
is 2 x 2
  | 221{ 1} 52{ 2}
```
Next, we can compute all coarsest column-tactical refinements of the decomposition. To this end, the command

```
orbiter.out -v 4 -tdo_refinement -input_file max_arc_q16_r4.tdo \n    -dual_is_linear_space -end
```

is used. Because the incidence structure is a projective plane, the dual is a linear space also. Hence the option `-dual_is_linear_space` can be used, which is helpful to reduce possibilities. As it turns out, there is exactly one refinement, and it is tactical. The file `max_arc_q16_r4r.tdo` is produced. Note the added letter `r` at the end of the file name (r for refinement). We can use the following command to display the decomposition stack in the file:

```
orbiter.out -v 4 -tdo_print max_arc_q16_r4r.tdo
```

This produces the following output:

```
decomposition 0.1:
lambda_scheme at level 2 :
  is 1 x 1
    | 273_{ 1}
---------------------
273_{ 0} | 17 17

row_scheme at level 4 :
  is 2 x 2
    | 221_{ 1} 52_{ 2}
---------------------
52_{ 0} | 4 0
221_{ 3} | 13 17

col_scheme at level 4 :
  is 2 x 2
    | 221_{ 1} 52_{ 2}
---------------------
52_{ 0} | 4 0
221_{ 3} | 13 17
```
At the moment, the classes of the partitions are too large to be useful for generation of these objects. It would be helpful to investigate the block derived maximal arc. This would lead to finer partitions with more classes of smaller size. We start with the block tactical decomposition in the following stack file:

\[ \begin{array}{c|cc}
221 & 52 & \\
273 & 17 & 17 \\
\end{array} \]

This decomposition is block-tactical (note the \texttt{type=bt} entry). The matrix entries are \( b_{ij} \) which is the number of points in point class \( i \) which lie on a line in block class \( j \). The fuse partition is a partition of the block classes. And now for the block derived decomposition. We use the stack file \texttt{max_arc_q16_r4bd.stack}:

\[ \begin{array}{c|ccc}
221 & 51 & 1 & \\
52 & 4 & 0 & 0 \\
221 & 13 & 17 & 17 \\
\end{array} \]

This time, we split off one line from the class of 52 external lines, to yield two classes (of size 51 and 1, respectively). The block partition is three, indicating that all three block classes need to be fused in order to arrive at the row-tactical ancestor decomposition in the stack. We are using the command

```
orbiter.out -v 4 -convert_stack_to_tdo max_arc_q16_r4bd.stack
```

to convert the stack file into a tdo file. This produces the following output:

\[ \begin{array}{c|c}
273 & 1 \\
\end{array} \]

\[ \begin{array}{c|c}
273 & 0 \\
\end{array} \]
row_scheme at level 4:
is 2 x 2

```
<table>
<thead>
<tr>
<th>221_{ 1} 52_{ 2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>52_{ 0}</td>
</tr>
<tr>
<td>221_{ 3}</td>
</tr>
</tbody>
</table>
```

col_scheme at level 5:
is 2 x 3

```
<table>
<thead>
<tr>
<th>221_{ 1} 51_{ 2} 1_{ 4}</th>
</tr>
</thead>
<tbody>
<tr>
<td>52_{ 0}</td>
</tr>
<tr>
<td>221_{ 3}</td>
</tr>
</tbody>
</table>
```

The command

```
orbiter.out -v 4 -tdo_refinement -input_file max_arc_q16_r4bd.tdo \
-dual_is_linear_space -end
```

is used to compute the coarsest row-tactical refinements. It turns out that there is exactly one, obtained by splitting the point-class of size 221 into two, one of size 204 and one of size 17. The command

```
orbiter.out -v 4 -tdo_print max_arc_q16_r4bdr.tdo
```

can be used to print this decomposition as

```
0.1 → 221_1 51_2 1_4
52_0 | 17 0 0
204_3 | 13 4 0
17_5 | 13 3 1
```

Using the command

```
orbiter.out -v 4 -tdo_refinement -input_file max_arc_q16_r4bdr.tdo \
-dual_is_linear_space -end
```

we compute the coarsest column tactical refinements yet again. As it turns out, the decomposition does not split because it is already tactical. Another print command

```
orbiter.out -v 4 -tdo_print max_arc_q16_r4bdrr.tdo
```

yields the tactical decomposition

```
0.1.1 → 221_1 51_2 1_4
52_0 | 17 0 0
204_3 | 13 4 0
17_5 | 13 3 1
```

```
0.1.1 ↓ 221_1 51_2 1_4
52_0 | 4 0 0
221_3 | 13 17 17
```

```
75
```
How do we read this decomposition? We see that there is one external line of 17 points. We may call this the line at infinity. Through each of these 17 points at infinity, there are 16 further lines, coming in two types: Always 3 lines have the property that they are external, and the remaining 13 lines are secants. Taken together, all 51 external lines (distinct from the very first line) arises in this way and cover 204 points. Likewise, all 221 secants arise from the second type of lines. Each of the secants has the following type with respect to points: it intersects 4 points of the maximal arc, 12 points from the set of 204, and exactly one point at infinity.
21 Spreads

A $t$-spread of $\text{PG}(n, q)$ is a set of disjoint $\text{PG}(t, q)$ that cover all of $\text{PG}(n, q)$ pointwise. $t$-spreads in $\text{PG}(n, q)$ exist if $t + 1$ divides $n + 1$. Two $t$-spreads are isomorphic if there is a collineation of $\text{PG}(n, q)$ which maps one to the other. The classification problem for $t$-spreads is the problem of determining a complete set of pairwise non-isomorphic $t$-spreads. Orbiter can be used to classify spreads for small parameters. For instance, the command

```
orbiter.out -v 6 \
    -linear_group -PGGL 4 4 -end \n    -group_theoretic_activities \n    -poset_classification_control \n    -W \n    -problem_label spreads_16_4 \n    -end \n    -spread_classify 2 \n    -end
```

classifies the line-spreads of $\text{PG}(3, 4)$ under the action of $\text{PGL}(4, 4)$. Under the André Bruck-Bose construction [1, 9], these spreads correspond to translation planes of order 16 with kernel $\mathbb{F}_4$. Up to isomorphism, there are exactly three line-spreads in $\text{PG}(3, 4)$. They are the dearguesian spread, the Hall spread, and the semifield spread. Spreads of $\text{PG}(k - 1, q)$ exist in $\text{PG}(n - 1, q)$ if and only if $k$ divides $n$. However, constructing and classifying all spreads is difficult and possible only for very small parameters.

22 Packings

A packing of $\text{PG}(3, q)$ is a set of pairwise line-disjoint line-spreads of $\text{PG}(3, q)$ of size $q^2 + q + 1$. Each spread contains $q^2 - 1$ lines. A simple counting argument shows that every line is contained in exactly one spread of the packing. The classification problem for packings is the problem of determining a complete set of pairwise non-isomorphic packings. Orbiter can be used to classify packings for small parameters. It is sometimes useful to make a symmetry assumption. This means that only those packings will be found that satisfy the symmetry assumption. The reason for making such an assumption is that the problem becomes easier and hence more tractable. Often, an assumption is made that the packings are invariant under a (nontrivial) group $H$. This section describes various ways in which Orbiter can help find and classify packings, with or without symmetry assumption.

Suppose we want to find packings in $\text{PG}(3, 4)$. To this end, we can use the three types of spreads that we just discussed in the previous example. They are the Hall spread, the Desarguesian spread, and the semifield spread. Suppose we are interested in packings which consist solely of spreads of Hall type. A packing which consists of isomorphic spreads is called uniform. Suppose we want to make a symmetry assumption. Specifically, we pick a group $H$ and assume that the packing is invariant under $H$ (which means that $H$ is a subgroup of the automorphism group of the packing). Let $N = N_G(H)$ be the normalizer of $H$ in $G$, with $G$
the collineation group $\text{PGL}(4, q)$. At first, the $H$-orbits on spreads are computed and tallied by length. The orbits of length one (i.e., fixed spreads) are selected and a graph is defined. The vertices of the graph are the spreads that are fixed. Two vertices are adjacent if the associated spreads are line-disjoint. A packing must be related to a clique of a certain size in this graph. In order to find these cliques, poset classification is applied to classify all cliques in this graph under the action of $N$. Note that because $N \leq N_G(H)$, $N$ is contained in the automorphism group of the graph and hence induces an action on cliques of any size in the graph. Once the cliques on fixed spreads are found and classified, the remaining spreads of the packing are assembled from the set of non-trivial orbits of $H$ on spreads. The orbit type of the packing is the combinatorial information about the number of $H$-orbits on spreads of a given length that make up the packing. In many cases, the orbit type contains only few orbit lengths. In order to pick the non-trivial orbits, we proceed by orbit-length. That is, all orbits of a given length are chosen in one step. Let us consider an example.

Suppose we are interested in packings invariant under a group $H$ of order two. The Orbiter command

```
$(\text{ORBITER}\_\text{PATH})/\text{orbiter.out} \ \ \
-\text{magma_path} /\text{usr/local/magma/} \ \ \
-v 6 \ \
-linear\_\text{group} -\text{PGGL} 4 4 -\text{end} \ \
-group\_\text{theoretic}\_\text{activities} \ \
-classes
```
pdflatex PGGL_4_4_classes_out.tex
open PGGL_4_4_classes_out.pdf

can be used to create a list of conjugacy classes of elements of $\text{PGL}(4, 4)$, using the help of Magma. The option `-magma_path` provides the path of the magma executable and is installation dependent. The command also computes the normalizers of the associated cyclic subgroups generated by the representatives of the conjugacy classes. We find that the group $\text{PGL}(4, 4)$ contains three conjugacy classes of involutions, which we may call $2A$, $2B$ and $2C$, respectively. The elements that Magma picks as class representatives are not particularly nice. In the following list, we replace the class representatives by their Jordan canonical forms. The class $2A$ contains the Baer involution

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

with 15 fixed points (a Baer subgeometry). The centralizer and the normalizer have order 40320. The class size is 48960. The class $2B$ contains the involution

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

with 78 fixed points.
with 21 fixed points (a hyperplane). The centralizer and the normalizer have order 368640. The class size is 5355. The class $2C$ contains the involution

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

with 5 fixed points (a line). The centralizer and the normalizer have order 30720. The class size is 64260. Elements of class $2B$ cannot stabilize a packing. For this reason, we are left with two classes, $2A$ and $2C$. Suppose we want to assume a stabilizer of type $2C$. So, we let $H$ be the group of order 2 generated by the representative of class $2C$ shown above. In order to compute the normalizer of $H$, we use the command

```
$(ORBITER_PATH)/orbiter.out -v 6 \\
  -linear_group -PGGL 4 4 -end \\
  -group_theoretic_activities \\
  -centralizer_of_element "2B" "1,0,0,0, 1,1,0,0, 0,0,1,0, 0,0,1,1, 0"
```

This gives us generators for the normalizer $N = N_G(H)$ as matrices, which we can use later. Next, we first have to create the complete list of Hall-spreads in $PG(3,4)$. To this end, we use the Orbiter command

```
- mkdir SPREAD_TABLES_4_HALL
$(ORBITER_PATH)/orbiter.out -v 6 \\
  -linear_group -PGGL 4 4 -end \\
  -group_theoretic_activities \\
  -poset_classification_control \\
  -W \\
  -problem_label packing_4 \\
  -end \\
  -packing_classify 2 "0" "SPREAD_TABLES_4_HALL/" \\
  -end
```

This command takes just under 10 minutes to execute. Next, we need to act with $N$ on the $H$-orbits. Only orbits which are line-disjoint can be considered. The following command can be used to do so. We assume that the orbit structure of $H$ on spreads consists of 5 fixed spreads and 8 orbits of length two. The following command computes the orbits of $H$ on spreads and eliminates orbits which are not line-disjoint, and hence cannot appear in a packing:

```
$(ORBITER_PATH)/orbiter.out -v 5 \\
  -orbiter_path $(ORBITER_PATH)/ \\
  -linear_group -PGGL 4 4 -end \\
  -group_theoretic_activities \\
  -poset_classification_control \\
```

79
The option -H is used to specify the assumed symmetry group $H$. The option -N is used to specify the normalizer of $H$ in $G = PΓL(4, 4)$. The orbits of $H$ on spreads are computed first and then sorted according to their length. It turns out that there are 2432 fixed spreads and 821312 $H$-orbits of length two. Of the orbits of length two, 213760 are line disjoint and hence can be used to build a packing. The other orbits are discarded. The next step is to compute the orbits of $N$ on 5-subsets of the set of 2432 fixed spreads. The group $N$ has been computed earlier. It has order 30720, and generators are given as part of the -N command. Orbiter finds exactly 8268 $N$-orbits on cliques of size 5 of the compatibility graph of fixed spreads. The compatibility graph has as vertices the 2432 fixed spreads. Two vertices are joined by an edge if the associated spreads are line-disjoint. Since we assume 5 fixed spreads, we need to search for all cliques of size 5 in this graph. We consider these cliques under the action of $N$, which leads us to 8268 orbits of fixed point cliques. For each of these cliques, we need to consider the orbits of length two which are compatible with the choice of the fixed point clique. As always, compatible means line-disjoint. The orbits that survive this test can be used to build a packing from the selected set of 5 fixed spreads. A graph is defined with the orbits which survive this test as vertices (the number of such orbits is about 10000). Two vertices are joined by an edge if the associated orbits are line-disjoint. We then find all cliques of size 8 in this graph. Since this step is computationally
demanding, we use parallel computing. First, we create the graphs in 8 batches $B_1, \ldots, B_7$. Then we solve the clique finding problem in 32 batches for each of the 8 batches. In total, there are 256 second order batches, $C_1, \ldots, C_{255}$. They are defined by the fixed point clique cases modulo 256. Batch $B_i$ is associated with second order batches $C_{i+8k}$, where $k = 0, \ldots, 31$. The following Orbiter command uses 8 jobs to compute the graphs associated with the first order batches:

```bash
$(ORBITER_PATH)/orbiter.out \
  -fork CASE_NUMBER log_%d 0 8 1\n  -v 5 \
  -orbiter_path $(ORBITER_PATH)/ \
  -linear_group -PGGL 4 4 -end \n  -group_theoretic_activities \n    -poset_classification_control \n    -W \n    -problem_label packing_4 \n  -end \n  -packing_classify 2 "0" "SPREAD_TABLES_4_HALL/" \n  -packing_with_assumed_symmetry \n    -problem_label _uniform_Hall \n  -H -PGGL 4 4 -subgroup_by_generators "2C" 2 1 \n    "1,0,0,0, 1,1,0,0, 0,0,1,0, 0,0,1,1, 0" \n    -end \n  -N -PGGL 4 4 -subgroup_by_generators "centralizer_2C" "30720" 9 \n    "1,0,0,0,0,1,0,0,0,1,0,0,0,1,1," \n    "1,0,0,0,0,1,0,0,0,2,0,0,0,0,2,1," \n    "1,0,0,0,0,1,0,0,0,1,0,2,0,3,1,0," \n    "1,0,0,0,0,1,0,0,0,1,0,1,0,3,1,0," \n    "1,0,0,0,0,1,0,0,0,1,0,1,1,1,1,1," \n    "1,0,0,0,1,1,0,0,0,0,1,0,0,1,1,0," \n    "1,0,0,0,2,1,0,0,0,0,1,0,1,0,1,0," \n    "1,0,0,0,1,1,2,0,0,0,0,1,0,0,0,1,0," \n    "1,0,3,0,1,1,1,3,0,2,0,0,0,0,2,1," \n    -end \n  -cliques_on_fixpoint_graph 5 \n  -cliques_on_fixpoint_graph_control -W -problem_label \n  PGGL_4_4_Subgroup_2C_2_uniform_Hall_fixp_graph_cliques -end \n  -process_long_orbits \n    -create_graphs \n    -split CASE_NUMBER 8 \n    -list_of_cases_from_file PGGL_4_4_Subgroup_2C_2_uniform_Hall_graph_cliques_by_type_500.csv \n    -clique_size 8 \n    -orbit_length 2 \n  -end \n -end
```
The following Orbiter command uses 32 jobs to perform the clique search for the graphs in batch $B_3$ (for example):

```
$(ORBITER_PATH)/orbiter.out \
  -fork CASE_NUMBER log_%d 3 256 8\ 
  -v 5 \ 
  -orbiter_path $(ORBITER_PATH)/ \ 
  -linear_group -PGGL 4 4 -end \ 
  -group_theoretic_activities \ 
  -poset_classification_control \ 
  -W \ 
  -problem_label packing_4 \ 
  -end \ 
  -packing_classify 2 "0" "SPREAD_TABLES_4_HALL/" \
  -packing_with_assumed_symmetry \ 
  -problem_label _uniform_Hall \ 
  -H -PGGL 4 4 -subgroup_by_generators "2C" 2 1 \ 
    "1,0,0,0, 1,1,0,0, 0,0,1,0, 0,0,1,1, 0" \ 
    -end \ 
  -N -PGGL 4 4 -subgroup_by_generators "centralizer_2C" "30720" 9 \ 
    "1,0,0,0,0,1,0,0,0,0,1,0,0,0,1,0,1,1," \ 
    "1,0,0,0,0,1,0,0,0,2,0,0,0,0,2,1," \ 
    "1,0,0,0,0,1,0,0,0,1,0,2,0,3,1,0," \ 
    "1,0,0,0,0,1,0,0,0,1,0,1,0,3,1,0," \ 
    "1,0,0,0,0,1,0,0,1,1,1,1,1,1," \ 
    "1,0,0,0,1,1,0,0,0,0,1,0,0,0,1,1,0," \ 
    "1,0,0,0,2,1,0,0,0,0,1,0,1,0,0,1,0," \ 
    "1,0,0,0,1,1,2,0,0,0,0,1,0,0,0,1,0," \ 
    "1,0,3,0,1,1,1,3,0,0,2,0,0,0,0,2,1," \ 
    -end \ 
  -cliques_on_fixpoint_graph 5 \ 
  -cliques_on_fixpoint_graph_control -W -problem_label PGGL_4_4_Subgroup_2C_2_uniform_Hall \ 
  -process_long_orbits \ 
  -solve \ 
  -split CASE_NUMBER 256 \ 
  -list_of_cases_from_file PGGL_4_4_Subgroup_2C_2_uniform_Hall_graph_cliques_by_type_500.csv \ 
  -clique_size 8 \ 
  -orbit_length 2 \ 
  -end \ 
  -end \ 
  -end
```

The number of jobs is $32 = 256/8$. The clique search jobs are Orbiter jobs that are created by this command. Notice the command line option `orbiter_path` to set the path to the
orbiter executable to $\texttt{(ORBITER\_PATH)}$ (which is a makefile variable). Using two computers in parallel, one with 32 cores and one with 8 cores, we are able to perform the clique search for all 8268 cases within 24 hours. Once the cliques have been computed, we read them all in and save them in one file. The command to do this is very similar to the previous command. Since there is no point in doing this in parallel, we remove the \texttt{-fork} command. Also, we replace \texttt{solve} by \texttt{-read_solutions}, which is aimed at reading the solutions from the clique finder. We add the extra command \texttt{-solution_path SOLUTIONS/} to specify the path to the directory where we collect the solution files. If all is done, it turns out that we have 8374 packings in total, all copied over into the single file

\texttt{PGGL\_4\_4\_Subgroup\_2C\_2\_uniform\_Hall\_graph\_cliques\_by\_type\_500\_packings.csv}

The next command performs the isomorph rejection for these packings:

\texttt{$(ORBITER\_PATH)/orbiter.out -v 2 -canonical_form_PG 3 4 \ -input -file_of_packings_through_spread_table \ PGGL\_4\_4\_Subgroup\_2C\_2\_uniform\_Hall\_graph\_cliques\_by\_type\_500\_packings.csv \ SPREAD\_TABLES\_4\_HALL/spread\_16\_spreads.csv -end \ -classification_prefix packings\_2C \ -report \ -end}$

This command takes about 11 hours.

A drawing of the adjacency matrix of the graph can be produced using the following command sequence:

\texttt{orbiter.out -v 2 -create_graph -load_from_file \ PGGL\_4\_4\_Subgroup\_2A\_2\_fixp\_graph.bin -end \ -graph_theoretic_activity -export_csv \ PGGL\_4\_4\_Subgroup\_2A\_2\_fixp\_graph.csv -end}

\texttt{orbiter.out -v 2 -draw_matrix PGGL\_4\_4\_Subgroup\_2A\_2\_fixp\_graph.bin.csv 2}
23 The Povray Interface

Orbiter can be used to create raytracing 3D-graphics. Orbiter serves as a front end for the raytracing software Povray [23]. This is a multi step process: A 3D scene is defined through orbiter commands. Next, Orbiter produces Povray files. After that, the povray files are processed through povray, and turned into graphics files (png), called frames. The frames can be turned into a video by using tools like ffmpeg. By default, an rotational animation is produced. Tables 31-32 list the commands to control the 3D-povray frontend. Tables 33 and 34 summarizes the Orbiter commands to build objects of a 3D scene. Building the scene itself does not create any graphical output. To this end, the commands in Table 35 are used. Each of these commands applies to a group of objects of the same kind. Groups of objects are created using the commands in Table 34 which start with group_of. Here is a simple example which combines scene building and graphical output. The example creates a cube with vertices, edges and faces:

```
1     orbiter.out -v 2 -povray
2         -round 0 -nb_frames_default 30 -output_mask cube_%d%03d.pov \
3         -video_options -W 1024 -H 768 -global_picture_scale 0.5 \
4         -default_angle 75 -clipping_radius 2.7 \
5         -end \n6         -scene_objects \n7         -obj_file cube_centered.obj \n8         -edge "0, 1" \n9         -edge "0, 2" \n10        -edge "0, 4" \n11        -edge "1, 3" \n12        -edge "1, 5" \n13        -edge "2, 3" \n14        -edge "2, 6" \n15        -edge "3, 7" \n16        -edge "4, 5" \n17        -edge "4, 6" \n18        -edge "5, 7" \n19        -edge "6, 7" \n20        -group_of_things_as_interval 0 8 \n21        -spheres 0 0.3 "texture{ Polished_Chrome pigment{quick_color White} }
22                " \n23        -group_of_things_as_interval 0 6 \n24        -prisms 1 0.05 "texture{ pigment{ color Yellow transmit 0.7 
25                          } finish {diffuse 0.9 phong 0.6} }
26                " \n27        -group_of_things_as_interval 0 12 \n28        -cylinders 2 0.15 "texture{ pigment{ color Red } finish {diffuse 0.9 phong 0.6} }
29                " \n30        -scene_objects_end \n31        -povray_end
```

84
<table>
<thead>
<tr>
<th>Option</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-do_not_rotate</td>
<td></td>
<td>No rotation.</td>
</tr>
<tr>
<td>-rotate_about_z_axis</td>
<td></td>
<td>Rotate around z-axis.</td>
</tr>
<tr>
<td>-rotate_about_111</td>
<td></td>
<td>Rotate around (1,1,1)-axis (default).</td>
</tr>
<tr>
<td>-rotate_about_custom_axis</td>
<td>axis</td>
<td>Rotate around a custom axis. The axis is specified as a vector of length 3.</td>
</tr>
<tr>
<td>-boundary_none</td>
<td></td>
<td>Remove the clipping.</td>
</tr>
<tr>
<td>-boundary_box</td>
<td></td>
<td>Clip using a box shape.</td>
</tr>
<tr>
<td>-boundary_sphere</td>
<td></td>
<td>Clip using a sphere (default).</td>
</tr>
<tr>
<td>-font_size</td>
<td>s</td>
<td>Set font size to s.</td>
</tr>
<tr>
<td>-stroke_width</td>
<td>s</td>
<td>Set text depth to s.</td>
</tr>
<tr>
<td>-omit_bottom_plane</td>
<td></td>
<td>Remove the bottom plane.</td>
</tr>
<tr>
<td>-W</td>
<td>w</td>
<td>Set output dimension to w pixels wide.</td>
</tr>
<tr>
<td>-H</td>
<td>h</td>
<td>Set output dimension to h pixels height.</td>
</tr>
<tr>
<td>-nb_frames</td>
<td>n</td>
<td>Set number of frames to n. One revolution around the axis is split into n frames.</td>
</tr>
<tr>
<td>-zoom</td>
<td>r $a_s$ $a_t$ $c_s$ $c_t$</td>
<td>Set zoom angle and clipping with in round r to change from $a_s$ to $a_t$ and from $c_s$ to $c_t$, respectively.</td>
</tr>
<tr>
<td>-pan</td>
<td>r $F T C$</td>
<td>In round r, pan the camera from location $F$ to location $T$ in a rotational movement with center at $C$. Each of $F,T,C$ are three dimensional coordinates.</td>
</tr>
<tr>
<td>-pan_reverse</td>
<td>r $F T C$</td>
<td>Same as -pan, but camera movement is in opposite order.</td>
</tr>
<tr>
<td>-no_background</td>
<td></td>
<td>Remove background.</td>
</tr>
<tr>
<td>-no_bottom_plane</td>
<td>r</td>
<td>Remove bottom plane in round r.</td>
</tr>
<tr>
<td>-camera</td>
<td>r $S C L$</td>
<td>In round r, set camera location at $C$, sky at $S$ and pointing towards $L$. Each of $S,C,L$ are three-dimensional coordinate vectors.</td>
</tr>
<tr>
<td>-clipping</td>
<td>r $c$</td>
<td>In round r, set clipping radius to $c$.</td>
</tr>
</tbody>
</table>

Table 31: Options for Orbiter 3D-graphics (Part 1)
<table>
<thead>
<tr>
<th>Option</th>
<th>Arguments</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-text</td>
<td>r a text</td>
<td>In round r, produce running text text with sustain value a.</td>
</tr>
<tr>
<td>-label</td>
<td>r s a g text</td>
<td>In round r, produce running text text with start value s, sustain s and gravity g.</td>
</tr>
<tr>
<td>-latex</td>
<td>r s a preamble g text l fname</td>
<td>In round r, produce running latex text text with start value s, sustain s and gravity g. Put preamble in the latex source code. Use fname for the latex file names (no extension).</td>
</tr>
<tr>
<td>-global_picture_scale</td>
<td>d</td>
<td>Set scaling factor to d.</td>
</tr>
<tr>
<td>-picture</td>
<td>r d fname options</td>
<td>In round r, place picture fname scaled by d using options.</td>
</tr>
<tr>
<td>-picture</td>
<td>r d fname options</td>
<td>In round r, place picture fname scaled by d using options.</td>
</tr>
<tr>
<td>-look_at</td>
<td>L</td>
<td>Override camera look-at value to L. L is a three-dimensional vector.</td>
</tr>
<tr>
<td>-default_angle</td>
<td>a</td>
<td>Set default camera angle to a.</td>
</tr>
<tr>
<td>-clipping_radius</td>
<td>f</td>
<td>Set default clipping radius to f.</td>
</tr>
<tr>
<td>-scale_factor</td>
<td>s</td>
<td>Set default scale factor to s.</td>
</tr>
<tr>
<td>-line_radius</td>
<td>s</td>
<td>Set default line radius to s.</td>
</tr>
</tbody>
</table>

Table 32: Options for Orbiter 3D-graphics (Part 2)
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-cubic_lex</td>
<td>coeffs</td>
<td>Cubic surface given by 20 coefficients in lexicographic ordering</td>
</tr>
<tr>
<td>-cubic_orbiter</td>
<td>coeffs</td>
<td>Cubic surface given by 20 coefficients in Orbiter ordering</td>
</tr>
<tr>
<td>-cubic_Goursat</td>
<td>A B C</td>
<td>Cubic surface with tetrahedral symmetry given by 3 Goursat coefficients as $Ax^3 + B(x^2 + y^2 + z^2) + C = 0$</td>
</tr>
<tr>
<td>-quadric_lex_10</td>
<td>coeffs</td>
<td>Quadric surface given by 10 coefficients in lexicographic ordering</td>
</tr>
<tr>
<td>-quartic_lex_35</td>
<td>coeffs</td>
<td>Quartic surface given by 35 coefficients in lexicographic ordering</td>
</tr>
<tr>
<td>-octic_lex_165</td>
<td>coeffs</td>
<td>Octic surface given by 165 coefficients in lexicographic ordering</td>
</tr>
<tr>
<td>-point</td>
<td>coeffs</td>
<td>Point given by three coordinates</td>
</tr>
<tr>
<td>-point_list_from_csv_file</td>
<td>fname</td>
<td>List of points with coordinates given in a csv file</td>
</tr>
<tr>
<td>-line_through_two_points_recentered_from_csv_file</td>
<td>fname</td>
<td>List of lines through two points with point coordinates given in a csv file</td>
</tr>
<tr>
<td>-line_through_two_points_from_csv_file</td>
<td>fname</td>
<td>List of lines through two points with point coordinates given in a csv file</td>
</tr>
<tr>
<td>-point_as_intersection_of_two_lines</td>
<td>$i_1$ $i_2$</td>
<td>Create a point from the intersection of two lines $i_1$ and $i_2$</td>
</tr>
<tr>
<td>-edge</td>
<td>$i_1$ $i_2$</td>
<td>Create an edge (line segment) between points $i_1$ and $i_2$</td>
</tr>
<tr>
<td>-text</td>
<td>$i_1$ $s$</td>
<td>Create a label $s$ located at the point $i$</td>
</tr>
<tr>
<td>-triangular_face_given_by_three_lines</td>
<td>$i_1$ $i_2$ $i_3$</td>
<td>Create a triangular face give by three lines $i_1,i_2,i_3$</td>
</tr>
<tr>
<td>-face</td>
<td>pts</td>
<td>Create a face through the vertices pts, ordered cyclically</td>
</tr>
<tr>
<td>-quadric_through_three_skew_lines</td>
<td>$i_1$ $i_2$ $i_3$</td>
<td>Create a quadric through three skew lines</td>
</tr>
<tr>
<td>-plane_defined_by_three_points</td>
<td>$i_1$ $i_2$ $i_3$</td>
<td>Create a plane through three noncollinear points</td>
</tr>
<tr>
<td>-line_through_two_points_recentered</td>
<td>pt-coords</td>
<td>Create a line through two points given by 6 coordinates, recentered</td>
</tr>
</tbody>
</table>

Table 33: Scene definition commands (part 1)
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-line_through_two_points</code></td>
<td>pt-coords</td>
<td>Create a line through two points given by 6 coordinates</td>
</tr>
<tr>
<td><code>-line_through_two_existing_points</code></td>
<td>$i_1$ $i_2$</td>
<td>Create a line through two points</td>
</tr>
<tr>
<td><code>-line_through_point_with_direction</code></td>
<td>coeffs6</td>
<td>Create a line through a point $(x, y, z)$ with a given direction $(u_x, u_y, u_z)$, where coeffs6 $= x, y, z, u_x, u_y, u_z$</td>
</tr>
<tr>
<td><code>-plane_by_dual_coordinates</code></td>
<td>coeffs4</td>
<td>Create a plane $ax + by + cz + d = 0$ give four dual coordinates as coeff4 $= a, b, c, d$</td>
</tr>
<tr>
<td><code>-dodecahedron</code></td>
<td></td>
<td>Create a Dodecahedron centered at the origin (20 points, 30 edges, 12 faces)</td>
</tr>
<tr>
<td><code>-Hilbert_Cohn_Vossen_surface</code></td>
<td></td>
<td>Create the Hilbert, Cohn-Vossen surface (1 cubic surface, 45 tritangent planes, 27 lines)</td>
</tr>
<tr>
<td><code>-obj_file</code></td>
<td>fname</td>
<td>Read points and faces from the given .obj file</td>
</tr>
<tr>
<td><code>-group_of_things</code></td>
<td>list</td>
<td>Create a group of things from the given list</td>
</tr>
<tr>
<td><code>-group_of_things_with_offset</code></td>
<td>list offset</td>
<td>Create a group of things from the given list, each value is increase by offset</td>
</tr>
<tr>
<td><code>-group_of_things_as_interval</code></td>
<td>$a$ $b$</td>
<td>Create a group of things from the interval $a, \ldots, a + b - 1$</td>
</tr>
<tr>
<td><code>-group_of_things_as_interval_with_exceptions</code></td>
<td>$a$ $b$ $ex$</td>
<td>Create a group of things from the interval $a, \ldots, a + b - 1$ with the exceptional elements in the list $ex$ removed</td>
</tr>
<tr>
<td><code>-group_of_all_points</code></td>
<td></td>
<td>Create a group of things from all points currently defined</td>
</tr>
<tr>
<td><code>-group_of_all_faces</code></td>
<td></td>
<td>Create a group of things from all faces currently defined</td>
</tr>
<tr>
<td><code>-group_subset_at_random</code></td>
<td>$i$ $f$</td>
<td>Create a group of things from the existing group $i$ by picking a random subset with probability $f$</td>
</tr>
<tr>
<td><code>-create_regulus</code></td>
<td>$i$ $N$</td>
<td>Create a regulus for quadric $i$ with $N$ lines</td>
</tr>
</tbody>
</table>

Table 34: Scene definition commands (part 2)
<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-spheres</td>
<td>(i ) (r) (\text{prop})</td>
<td>For each element in point group (i), create a sphere of radius (r) with given Povray properties.</td>
</tr>
<tr>
<td>-cylinders</td>
<td>(i ) (r) (\text{prop})</td>
<td>For each element in edge group (i), create a cylinder of radius (r) with given Povray properties.</td>
</tr>
<tr>
<td>-prisms</td>
<td>(i ) (d) (\text{prop})</td>
<td>For each element in face group (i), create a prism of half-thickness (d) with given Povray properties.</td>
</tr>
<tr>
<td>-planes</td>
<td>(i) (\text{prop})</td>
<td>For each element in plane group (i), create a plane with given Povray properties.</td>
</tr>
<tr>
<td>-lines</td>
<td>(i ) (r) (\text{prop})</td>
<td>For each element in line group (i), create a line of radius (r) with given Povray properties.</td>
</tr>
<tr>
<td>-cubics</td>
<td>(i) (\text{prop})</td>
<td>For each element in group (i) of cubics, create a surface with given Povray properties.</td>
</tr>
<tr>
<td>-quadrics</td>
<td>(i) (\text{prop})</td>
<td>For each element in group (i) of quadrics, create a surface with given Povray properties.</td>
</tr>
<tr>
<td>-quartics</td>
<td>(i) (\text{prop})</td>
<td>For each element in group (i) of quartics, create a surface with given Povray properties.</td>
</tr>
<tr>
<td>-octics</td>
<td>(i) (\text{prop})</td>
<td>For each element in group (i) of octics, create a surface with given Povray properties.</td>
</tr>
<tr>
<td>-texts</td>
<td>(i ) (d) (s) (\text{prop})</td>
<td>For each element in group (i) of labels, create a text element with half-thickness (d) and size (s) with given Povray properties.</td>
</tr>
</tbody>
</table>

Table 35: Graphical output commands
This command will tell Orbiter to create 30 povray files (extension .pov), one for each frame of a rotating scene. The scene contains a cube whose vertices are shown in chrome, whose edges are in red, and whose faces are yellow and transparent. The cube turns around a vertical axis of symmetry. Here is the first frame of the result:

![Image of a cube with vertices, edges, and faces in different colors]

The coordinates of the cube are stored in an object file `cube_centered.obj`. The content of this file is:

```
v -1 -1 -1
v 1 -1 -1
v -1 1 -1
v 1 1 -1
v -1 -1 1
v 1 -1 1
v -1 1 1
v 1 1 1
f 1 2 4 3
f 1 2 6 5
f 1 3 7 5
f 2 4 8 6
f 3 4 8 7
f 5 6 8 7
```

Here is a simple example of a cubic surface, called the monkey saddle. The equation of the surface is

\[ z = x^3 - 3xy^2 \]

The example plots the surface together with the tangent plane at (0, 0, 0), rotated around the z-axis.

```
orbiter.out -v 2 -povray -round 0 -nb_frames_default 30 -output_mask monkey_%d_%03d.pov \
```
Here is one of the frames that are created:

Here is another cubic surface, called Hilbert Cohn-Vossen. The equation of the surface is

$$\frac{5}{2}xyz - (x^2 + y^2 + z^2) + 1 = 0.$$
Figure 17 shows the final product. The Schläfli labeling of lines can be seen. Orbiter can plot functions using a built-in function tracker. The functions must be continuous apart from a finite number of poles. The function can have multiple components, each described using an expression. Each expression is specified in Reverse Polish Notation (RPN). Consider an example. A Lissajous curve is defined using coordinate functions of the form

\[ x = r \sin(at + c), \quad y = r \sin(bt), \quad a, b, c, r \in \mathbb{R}. \]

The terms

\[ r \sin(at + c), \quad r \sin(bt) \]

are the expressions of the two coordinate functions. RPN means that the operator is listed after the operands. A stack data structure is used to hold temporary values. Operators are pushed to the top of the stack using the push commands. A binary operator pops the two elements from the stack, performs the operation, and pushes the resulting value back onto the stack. For a unary operator, only one element is popped and replaced by the result. Here
are some examples of expressions rewritten in RPN:

\[
\sin(x) \mapsto \text{push } x \text{ sin,}
\]

\[
a + b \mapsto \text{push } a \text{ push } b \text{ add,}
\]

\[
a \cdot b \mapsto \text{push } a \text{ push } b \text{ mult.}
\]

The coordinate functions are enclosed between -code and -code_end commands. Each coordinate function is described in RPN and terminated using a return keyword. By the time the return keyword is reached, the RPN expression must have exactly one value on the stack which is considered the value of the expression. Constants are declared between the -const and -const_end keywords. Likewise, variables are declared between the -var and -var_end keywords. Picking \(a = 3\), \(b = 2\), \(c = \pi/2\) and \(r = 7\), the function is computed using

```bash
orbiter.out -v 2 -smooth_curve "lissajous" 0.07 2000 15 0 18.85 \\
-const a 3 b 2 c 1.57 r 7 -const_end \\
-var t -var_end \\
-code \\
  push t push a mult push c add sin push r mult return \\
  push t push b mult sin push r mult return \\
-code_end
```

The sequence

```
push t push a mult push c add sin push r mult
```

is \(r \sin(at + c)\) expressed in RPN. The constants are defined in the line

```
-const a 3 b 2 c 1.57 r 7 -const_end
```

The input variable is defined using the line

```
-var t -var_end
```
The sequence

```
-smooth_curve "lissajous" 0.07 2000 15 0 18.85
```

defines the name of the output file, the fact that two consecutive points are never further
than $\epsilon = 0.07$ away, the fact that points that are 15 or more away from the origin should
be ignored, and the fact that the variable $t$ loops over the range $[0, 18.85]$ with a default of
2000 steps. The evaluator automatically reduces the step-size if consecutive image points
are more than $\epsilon$ apart. The code to produce the plot is

```
orbiter.out -v 2 -povray \
   -round 0 -nb_frames_default 1 -output_mask lissajous_%d_%03d.pov \
   -video_options -W 1024 -H 768 -global.picture_scale 0.40 \ 
   -default_angle 45 -clipping_radius 5 -omit_bottom_plane \
   -camera 0 "0,-1,0" "0,0,12" "0,0,0" \ 
   -rotate_about_z_axis \ 
   -end \ 
   -scene_objects \ 
   -line_through_two_points_recentered_from_csv_file coordinate_grid.csv \ 
   -group_of_things "0" \ 
   -group_of_things "1" \ 
   -group_of_things "2" \ 
   -lines 0 0.09 "texture{ pigment{ color Yellow } }" \ 
   -lines 1 0.09 "texture{ pigment{ color Yellow } }" \ 
   -lines 2 0.09 "texture{ pigment{ color Yellow } }" \ 
   -group_of_things_as_interval 3 39 \ 
   -lines 3 0.02 "texture{ pigment{ color Black } }" \ 
   -point_list_from_csv_file function_lissajous_N2000_points.csv \ 
   -group_of_things_as_interval 0 6524\ 
   -spheres 4 0.1 "texture{ pigment{ color Red } finish { diffuse 0.9 phong 1}}" \ 
   -plane_by_dual_coordinates "0,0,1,0" \ 
   -group_of_things "0" \ 
   -planes 5 "texture{ pigment{ color Blue*0.5 transmit 0.5 } }"
```

The plot is shown in Figure 18. We can turn it into a 3D plot by using the $t$ value for the $z$
coordinate. The code to produce the 3D plot is

```
orbiter.out -v 2 -povray \
   -round 0 -nb_frames_default 30 -output_mask lissajous_3d_%d_%03d
```
Figure 18: Lissajous figure

```
.pov

-video_options -W 1024 -H 768 -global_picture_scale 0.40 \\
-default_angle 45 -clipping_radius 5 -omit_bottom_plane \\
-camera 0 "0,0,1" "7,7,5" "0,0,1" \\
-rotate_about_z_axis \\
-end \\
-scene_objects \\
  -line_through_two_points_recentered_from_csv_file coordinate_grid.csv \\
  -group_of_things "0" \\
  -group_of_things "1" \\
  -group_of_things "2" \\
  -lines 0 0.09 "texture{ pigment{ color Yellow } }" \\
  -lines 1 0.09 "texture{ pigment{ color Yellow } }" \\
  -lines 2 0.09 "texture{ pigment{ color Yellow } }" \\
  -group_of_things_as_interval 3 39 \\
  -lines 3 0.02 "texture{ pigment{ color Black } }" \\
  -point_list_from_csv_file function_lissajous_3d_N2000_points.csv \\
-se \\
  -group_of_things_as_interval 0 6538 \\
-spheres 4 0.1 "texture{ pigment{ color Red } } finish { diffuse 0.9 phong 1}"
-plane_by_dual_coordinates "0,0,1,0" \\
-group_of_things "0"
-scene_objects_end \\
-povray_end
```

95
The function is computed using the command

```bash
orbiter.out -v 2 -smooth_curve "lissajous_3d" 0.07 2000 50 0 18.85 \ -const a 3 b 2 c 1.57 r 7 -const_end \ -var t -var_end \ -code \ push t push a mult push c add sin push r mult return \ push t push b mult sin push r mult return \ push t return \ -code_end \n```

The 3D curve is shown in Figure 19.

The Endrass octic \cite{10} is the algebraic surface given by the equation

\[
x^8 = 64 (-w^2 + x^2) (-w^2 + y^2) ((x + y)^2 - 2 w^2)((x - y)^2 - 2 w^2) - (-4 (1 + \sqrt{2}) (x^2 + y^2)^2 + (8 (2 + \sqrt{2}) x^2 + y^2)^2 + (2 + 7 \sqrt{2}) w^2) (x^2 + y^2) - 16 z^4 + 8 (1 - 2 \sqrt{2}) z^2 w^2 - (1 + 12 \sqrt{2}) w^4)^2
\]

The following Orbiter command creates a povray graphics of the octic, shown in Figure 20:

```bash
1 orbiter.out -v 2 -povray \
2 -round 0 -nb_frames_default 30 -output_mask endrass_octic_%d_%03d.pov \n3 -video_options -W 1024 -H 768 -global_picture_scale 0.75 -default \n4 -camera 0 "1,1,1" "6,6,3" "0,0,0" \n5 -rotate_about_111 \n6 -end \n7 -scene_objects \n```
This illustration includes coordinate axes and the $x,y$-plane.

It is possible to create a movie from orbiter povray graphics files using an external program such as ffmpeg. In order to do so, the files should have similar filenames. It is often easier to copy files into a temporary directory and changes the names along with it. The command prepare_frames can help, see Tables 36. For instance, the command

```bash
mkdir FRAMES
orbiter.out -prepare_frames \
-i 0 30 monkey_0_%03d.png \
-output_starts_at 0 \
-o FRAMES/frame%04d.png \
-end
```
Figure 20: The Endrass Octic

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-i</td>
<td>s l mask</td>
<td>Specify the input file names by running a printf command with the given mask applied to the index $i$ where $i$ goes from $s$ to $s+l-1$. This option can be repeated.</td>
</tr>
<tr>
<td>-step</td>
<td>s</td>
<td>Increment the index in steps of size $s$.</td>
</tr>
<tr>
<td>-o</td>
<td>mask</td>
<td>Create the output file using the given mask.</td>
</tr>
<tr>
<td>-output_starts_at</td>
<td>$i$</td>
<td>Start output file indices at $i$ (default is 0).</td>
</tr>
</tbody>
</table>

Table 36: Prepare frames commands
ffmpeg -r 5 -f image2 -i FRAMES/frame%04d.png -f mp4 -q:v 0 \ 
-vcodec mpeg4 "monkey.mp4"

creates a video \textit{monkey.mp4} from a set of 30 files. The individual filenames are created using \texttt{the printf format string \texttt{monkey_0_%03d.png}}, with an integer index that is drawn from the interval $[0, 29]$. The part that starts with a percent sign and ends with a ”d” character defines the way in which the integer is formatted. The number three before the “d” indicates that three characters will be printed. The zero indicates the use of leading zeros. So, the first file would be \texttt{monkey_0_000.png} and the very last file is \texttt{monkey_0_029.png}. The description of the printf format string can be found in the documentation of the C standard library.
References


[24] L. Schläfli. An attempt to determine the twenty-seven lines upon a surface of the third order and to divide such surfaces into species in reference to the reality of the lines upon the surface, *Quart. J. Math.* 2 (1858), 55–110.


