

Orbiter User's Guide

Anton Betten

June 5, 2020

Abstract

Orbiter is a program package devoted to the field of Algebraic Combinatorics. Specifically, it is aimed at the problem of classifying combinatorial objects and orbit computations, hence the name. Orbiter provides algorithms for effective handling of finite permutation groups in various actions. This guide is targeted for command line interface usage. Programmers who want to use the Orbiter class library in their own C++ program should consult the programmer's guide.

1 Introduction

Orbiter is a computer algebra system for Algebraic Combinatorics, with an emphasis on the classification of combinatorial objects. The kinds of problems for which Orbiter was designed for lie at the interface between group theoretic computations and combinatorics. Classification and computer search are the main topics for which Orbiter is perhaps stronger than other systems. Orbiter hopes to contribute 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 code and Orbiter data structures are 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

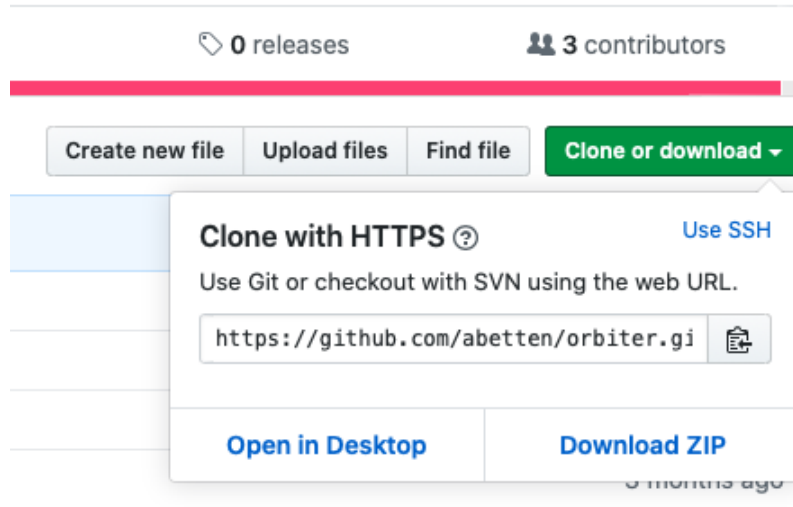


Figure 1: GitHub Clone or Download button

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

```
git clone <github-orbiter-path>
```

where `<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 “Clone or download” 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 `git clone` command.

- (c) Issue the following commands to complete the download of submodules:

```
cd orbiter
git submodule init
git submodule update
```

- (d) Issue the following commands to compile Orbiter using recursive makefiles:

```
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`.

3 Finite Fields

Finite fields and projective spaces over finite fields play an important role in Orbiter. The elements of the field \mathbb{F}_q are represented in different ways. Suppose that $q = p^e$ for some prime p and some integer $e \geq 1$. The elements of \mathbb{F}_q are mapped bijectively to the integers in the interval $[0, q - 1]$, using the base- p representation. If $e = 1$, the map takes the residue class $a \bmod p$ with $0 \leq a < p$ to the integer a . Otherwise, we write the field element as

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

where α is the root of some irreducible polynomial $m(X)$ of degree e over \mathbb{F}_p and $0 \leq a_i < p$ for all i . The associated integer is obtained as

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

This is the numerical rank of the polynomial. This representation takes 0 in \mathbb{F}_q to the integer 0. Likewise, $1 \in \mathbb{F}_q$ is mapped to the integer 1. Arithmetic is done by considering the polynomials over \mathbb{F}_p and modulo the irreducible polynomial $m(X)$ with root α . For instance, the field \mathbb{F}_4 is created using the polynomial $m(X) = X^2 + X + 1$. The elements are

$$0, \quad 1, \quad 2 = \alpha, \quad 3 = \alpha + 1.$$

Orbiter maintains a small database of primitive (irreducible) polynomials for the purposes of creating finite fields. This means that the residue class of α is a primitive element of the field, where α is a root of the polynomial.

The command

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

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

Unlike other computer algebra systems (GAP [7] and Magma [6]) Orbiter does not use Conway polynomials. However, Orbiter provides the option to override the polynomial used to create the finite field. For subfield relationships, the cheat sheet for the large field will indicate the irreducible polynomials of the subfield. For instance, Table 1 shows the subfields of \mathbb{F}_{64} generated by the polynomial $X^6 + X^5 + 1$ whose numerical rank is 97.

4 Finite Projective Spaces

Finite projective spaces and their groups are essential objects in Orbiter. The projective space $\text{PG}(n, q)$ 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.

Subfield	Polynomial	Numerical rank
\mathbb{F}_4	$X^2 + X + 1$	7
\mathbb{F}_8	$X^3 + X + 1$	11

Table 1: The subfields of \mathbb{F}_{64}

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 $P(\mathbf{x})$ for some vector $\mathbf{x} = (x_0, \dots, x_n)$ with $x_0, \dots, x_n \in \mathbb{F}_q$, not all zero. For any non-zero element $\lambda \in \mathbb{F}_q$, $P(\lambda\mathbf{x})$ is the same point as $P(\mathbf{x})$. For $\mathbf{a} = (a_0, \dots, a_n) \in \mathbb{F}_q$, not all zero, the symbol $[\mathbf{a}]$ represents the line

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

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

The command

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

creates a report for the projective geometry $\text{PG}(3, 2)$. Orbiter has enumerators for points and subspaces of $\text{PG}(n, q)$. 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{Gr}_1 . By enumerating \mathfrak{Gr}_1 we mean that we choose a fixed bijection between the set \mathfrak{Gr}_1 and the integer interval $\mathbb{Z}_N = \{0, 1, \dots, N - 1\}$ 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 $\mathbf{x} \in \mathbb{F}_q^n$, not zero, and maps it to the element in \mathbb{Z}_N representing the projective point $\mathbf{P}(\mathbf{x})$. A frame in $\text{PG}(n, q)$ 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 \mathbf{e}_i be the i -th unit vector. A frame for $\text{PG}(n, q)$ is

$$\mathbf{e}_0, \dots, \mathbf{e}_{n-1}, \mathbf{e}_0 + \dots + \mathbf{e}_{n-1}.$$

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 $\text{PG}(2, 2)$ the ordering is

$$(1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 1), (1, 1, 0), (1, 0, 1), (0, 1, 1).$$

Let us describe the two functions rank and unrank to perform the actual mappings between $\text{PG}(n, q)$ 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 RANK and UNRANK are mutually inverse functions. Consider the group $\text{PGL}(3, 2)$ acting on $\text{PG}(2, 2)$, for instance. The points of

Algorithm 1 Rank

```

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

```

$\text{PG}(2, 2)$ are listed in 2.

Let V be a finite dimensional vector space and let $\mathfrak{Gr}_k(V)$ be the Grassmannian of k -dimensional subspaces of V . If $\dim(V) = n$, the notation $\mathfrak{Gr}_{n,k}$ is used for $\mathfrak{Gr}_k(V)$. The size of the set $\mathfrak{Gr}_{n,k}$ 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

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  $\mathbf{e}_a$ 
5:    $a \leftarrow a - n$ 
6:   if  $a = 0$  then
7:     return one
8:    $a \leftarrow a - 1$ 
9:    $\mathbf{x} \leftarrow \mathbf{0}$ 
10:  for  $i = 1, \dots, n - 1$  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  $k = i + 1, \dots, n - 1$  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:     $r \leftarrow a \bmod q$ 
24:     $x_j \leftarrow \text{UNRANK}(\mathbb{F}_q, r)$ 
25:     $j \leftarrow j + 1$ 
26:     $a \leftarrow (a - r)/q$ 
27:  for  $h = j, \dots, i - 1$  do
28:     $x_h \leftarrow 0$ 
29:  return  $\mathbf{x}$ 

```

$a = \text{RANK}(x)$	$\mathbf{x} = \text{UNRANK}(a)$
0	(1, 0, 0)
1	(0, 1, 0)
2	(0, 0, 1)
3	(1, 1, 1)
4	(1, 1, 0)
5	(1, 0, 1)
6	(0, 1, 1)

Table 2: Representatives of the points of $\text{PG}(2, 2)$

$$\begin{array}{llll}
L_0 = \begin{bmatrix} 100 \\ 010 \end{bmatrix} & L_4 = \begin{bmatrix} 101 \\ 010 \end{bmatrix} & L_8 = \begin{bmatrix} 102 \\ 010 \end{bmatrix} & L_{12} = \begin{bmatrix} 010 \\ 001 \end{bmatrix} \\
L_1 = \begin{bmatrix} 100 \\ 011 \end{bmatrix} & L_5 = \begin{bmatrix} 101 \\ 011 \end{bmatrix} & L_9 = \begin{bmatrix} 102 \\ 011 \end{bmatrix} & \\
L_2 = \begin{bmatrix} 100 \\ 012 \end{bmatrix} & L_6 = \begin{bmatrix} 101 \\ 012 \end{bmatrix} & L_{10} = \begin{bmatrix} 102 \\ 012 \end{bmatrix} & \\
L_3 = \begin{bmatrix} 100 \\ 001 \end{bmatrix} & L_7 = \begin{bmatrix} 110 \\ 001 \end{bmatrix} & L_{11} = \begin{bmatrix} 120 \\ 001 \end{bmatrix} &
\end{array}$$

Table 3: The 13 lines of $\text{PG}(2, 3)$

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. For instance, the 13 lines of $\text{PG}(2, 3)$ are shown using 2×3 RREF generator matrices as shown in Table 3.

5 Algebraic Sets

A very important notion in projective geometry is that of algebraic sets. A set of points V in $\text{PG}(n, q)$ is algebraic if there is a set of homogeneous polynomials p_1, \dots, p_r whose roots over \mathbb{F}_q are the given set. In this case, we write $V = \mathbf{v}(p_1, \dots, p_r)$. The set V is often called the variety of p_1, \dots, p_r .

Conversely, given a set of points V in $\text{PG}(n, q)$, the ideal $I(V)$ is the set of homogeneous polynomials in $\mathbb{F}_q[X_0, \dots, X_n]$ 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 $\text{PG}(n, q)$, every set is algebraic of degree at most $(n+1)(q-1)$ [8]. 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 \mathbb{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

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

Using the indexing of monomials from Table 4, we record the following pairs (a, i) where a is the coefficient and i is the index of the monomial

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

h	monomial	vector
0	X_0^2	(2, 0, 0)
1	X_1^2	(0, 2, 0)
2	X_2^2	(0, 0, 2)
3	X_0X_1	(1, 1, 0)
4	X_0X_2	(1, 0, 1)
5	X_1X_2	(0, 1, 1)

h	monomial	vector
0	X_0^3	(3, 0, 0)
1	X_1^3	(0, 3, 0)
2	X_2^3	(0, 0, 3)
3	$X_0^2X_1$	(2, 1, 0)
4	$X_0^2X_2$	(2, 0, 1)
5	$X_0X_1^2$	(1, 2, 0)
6	$X_1^2X_2$	(0, 2, 1)
7	$X_0X_2^2$	(1, 0, 2)
8	$X_1X_2^2$	(0, 1, 2)
9	$X_0X_1X_2$	(1, 1, 1)

h	monomial	vector
0	X_0^4	(4, 0, 0)
1	X_1^4	(0, 4, 0)
2	X_2^4	(0, 0, 4)
3	$X_0^3X_1$	(3, 1, 0)
4	$X_0^3X_2$	(3, 0, 1)
5	$X_0X_1^3$	(1, 3, 0)
6	$X_1^3X_2$	(0, 3, 1)
7	$X_0X_2^3$	(1, 0, 3)
8	$X_1X_2^3$	(0, 1, 3)
9	$X_0^2X_1^2$	(2, 2, 0)
10	$X_0^2X_2^2$	(2, 0, 2)
11	$X_1^2X_2^2$	(0, 2, 2)
12	$X_0^2X_1X_2$	(2, 1, 1)
13	$X_0X_1^2X_2$	(1, 2, 1)
14	$X_0X_1X_2^2$	(1, 1, 2)

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

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

```
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 2).

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

```
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" \
  -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 numbering of Table 5. Each has a coefficient of one. Therefore, the equation is encoded as 1, 6, 1, 8, 1, 11, 1, 13.

h	monomial	vector
0	X_0^3	$(3, 0, 0, 0)$
1	X_1^3	$(0, 3, 0, 0)$
2	X_2^3	$(0, 0, 3, 0)$
3	X_3^3	$(0, 0, 0, 3)$
4	$X_0^2 X_1$	$(2, 1, 0, 0)$
5	$X_0^2 X_2$	$(2, 0, 1, 0)$
6	$X_0^2 X_3$	$(2, 0, 0, 1)$
7	$X_0 X_1^2$	$(1, 2, 0, 0)$
8	$X_1^2 X_2$	$(0, 2, 1, 0)$
9	$X_1^2 X_3$	$(0, 2, 0, 1)$
10	$X_0 X_2^2$	$(1, 0, 2, 0)$
11	$X_1 X_2^2$	$(0, 1, 2, 0)$
12	$X_2^2 X_3$	$(0, 0, 2, 1)$
13	$X_0 X_3^2$	$(1, 0, 0, 2)$
14	$X_1 X_3^2$	$(0, 1, 0, 2)$
15	$X_2 X_3^2$	$(0, 0, 1, 2)$
16	$X_0 X_1 X_2$	$(1, 1, 1, 0)$
17	$X_0 X_1 X_3$	$(1, 1, 0, 1)$
18	$X_0 X_2 X_3$	$(1, 0, 1, 1)$
19	$X_1 X_2 X_3$	$(0, 1, 1, 1)$

Table 5: The Orbiter ordering of monomials of degree 2 and 3 in $\text{PG}(3, q)$

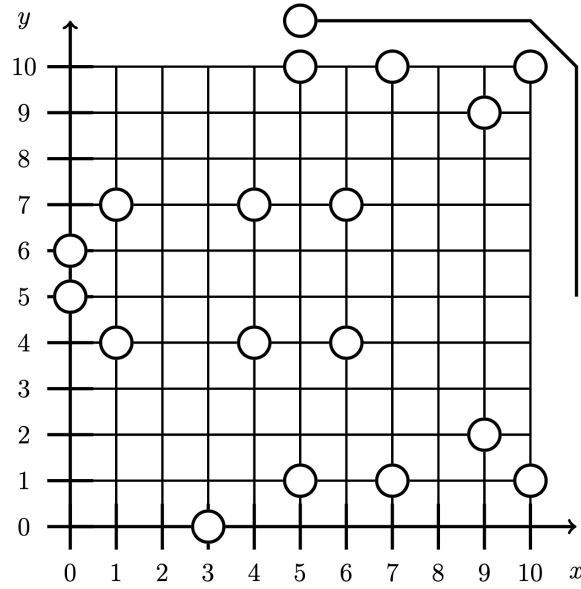


Figure 2: Elliptic curve $y^2 \equiv x^3 + x + 3 \pmod{11}$

Type	Perm. Domain	Degree
General linear $GL(n, q)$	all vectors of V	q^n
Affine $AGL(n, q)$	all vectors of V	q^n
Projective $PGL(n, q)$	$\mathfrak{S}r_1(V)$	$\frac{q^n-1}{q-1}$
Wreath product $GL(d, q) \wr \text{Sym}(n)$	$\mathfrak{S}r_1((\mathbb{F}_q^d)^{\otimes n})$ extended	$n + nq^d + \frac{q^{d^n}-1}{q-1}$

Table 6: Basic actions

6 Linear Groups

Groups in Orbiter are always permutation groups. This means that the group comes with a set on which it acts, called the permutation domain. One group can have many different actions. Three basic actions are defined for matrix groups: projective, affine, and general linear. In addition, there are direct product action and tensor product actions. Group theoretic algorithms require short orbits to be efficient. In order to create short orbits, Orbiter tags on an extra permutation domain for product and tensor actions. These extra permutation domains can be blended out later by forming a restricted action.

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 3. 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
```

Command	Arguments	Group
-GL	$n \ q$	$\mathrm{GL}(n, q)$
-GGL	$n \ q$	$\Gamma\mathrm{L}(n, q)$
-SL	$n \ q$	$\mathrm{SL}(n, q)$
-SSL	$n \ q$	$\Sigma\mathrm{L}(n, q)$
-PGL	$n \ q$	$\mathrm{PGL}(n, q)$
-PGGL	$n \ q$	$\mathrm{P}\Gamma\mathrm{L}(n, q)$
-PSL	$n \ q$	$\mathrm{PSL}(n, q)$
-PSSL	$n \ q$	$\mathrm{P}\Sigma\mathrm{L}(n, q)$
-AGL	$n \ q$	$\mathrm{AGL}(n, q)$
-AGGL	$n \ q$	$\mathrm{A}\Gamma\mathrm{L}(n, q)$
-ASL	$n \ q$	$\mathrm{ASL}(n, q)$
-ASSL	$n \ q$	$\mathrm{A}\Sigma\mathrm{L}(n, q)$
-GL_d_q_wr_Sym_n	$d \ q \ n$	$\mathrm{GL}(d, q) \wr \mathrm{Sym}(n)$

Table 7: Basic types of Orbiter matrix groups

creates the group $\mathrm{PGL}(4, 2)$ acting on the 15 elements of $\mathfrak{Gr}_1(\mathbb{F}_2^4)$. The basic types of groups are listed in Table 7.

For instance,

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

creates $\mathrm{P}\Gamma\mathrm{L}(3, 4)$ acting on the 21 points of $\mathrm{PG}(2, 4)$.

The command

```
orbiter.out -v 3 -linear_group -GL_d_q_wr_Sym_n 2 2 4 -end
```

creates the group $\mathrm{GL}(2, 2) \wr \mathrm{Sym}(4)$ acting on $\mathrm{PG}(15, 2)$ 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 $\mathrm{Sym}(4)$. An additional $16 = 4 \times 4$ points form four permutation domains of $\mathrm{GL}(2, 2)$, one for each factor.

A collineation of a projective space π is a bijective mapping from the points of π to themselves which preserves collinearity. That is, a collineation φ 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 Ψ_M is the induced map on projective space. By considering the homomorphism $M \mapsto \Psi_M$, the group $\mathrm{GL}(n+1, q)$ of invertible endomorphisms becomes a subgroup of the group of collineations of $\mathrm{PG}(n, q)$. This is the projectivity group $\mathrm{PGL}(n+1, q)$. It is isomorphic to $\mathrm{GL}(n+1, q)/\mathbb{F}_q^\times$. Another source of collineations is this: Let $\Phi \in \mathrm{Aut}(\mathbb{F}_q)$ be a field automorphism. Then Φ

acts on projective space by sending $P(\mathbf{x})$ to $P(\mathbf{x}\Phi)$. This map is another type of collineation, called automorphic collineation. This way, $\text{Aut}(\mathbb{F}_q)$ can be considered another subgroup of the group of collineations. If $q = p^h$ for some prime p and some integer h then

$$\Phi_0 : \mathbb{F}_q \rightarrow \mathbb{F}_q, \quad x \mapsto x^p$$

is a generator for the cyclic group $C_h \simeq \text{Aut}(\mathbb{F}_q)$. The collineation group of $\text{PG}(n, q)$ ($n \geq 2$) is isomorphic to the semidirect product of the projectivity group and the automorphism group of the field. The collineation group is $\text{P}\Gamma\text{L}(n+1, q) = \text{PGL}(n+1, q) \ltimes \text{Aut}(\mathbb{F}_q)$. We use the following notation for elements of $\text{P}\Gamma\text{L}(n+1, q)$. Let Φ_0 be a generator for $\text{Aut}(\mathbb{F}_q)$ and let $M \in \text{GL}(n+1, q)$. The map

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

is denoted as

$$M_k. \tag{1}$$

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

$$M_k \cdot N_l = \left(M \cdot N^{\Phi^{-k}} \right)_{k+l}, \tag{2}$$

$$\left(M_k \right)^{-1} = \left(\left(M^{-1} \right)^{\Phi^k} \right)_{-k}. \tag{3}$$

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

$$(M, \mathbf{a}, k) \in \text{GL}(n, q) \times \mathbb{F}_q^n \times \text{Aut}(\mathbb{F}_q),$$

which act on \mathbb{F}_q^n like so:

$$\left(\mathbf{x}, (M, \mathbf{a}, k) \right) \mapsto \left(\mathbf{x} \cdot M + \mathbf{a} \right)^{\Phi^k}.$$

We abbreviate the group elements as

$$M_{\mathbf{a}, k} = (M, \mathbf{a}, k).$$

The multiplication in $\text{A}\Gamma\text{L}(n, q)$ is

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

The inverse of an element is

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

A correlation is a one-to-one mapping between the set of points and the set of hyperplanes which reverses incidence. So, if ρ is a correlation and P is a point and ℓ is a hyperplane then P^ρ is a hyperplane and ℓ^ρ is a point and

$$\ell^\rho \in P^\rho \iff P \in \ell.$$

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

Table 8: Commands for creating subgroups

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

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

It is possible to create various subgroups of a group. Table 8 lists some commands to do so. For instance, the command

```
orbiter.out -v 3 -linear_group -PGL 7 11 -Janko1 -end
```

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

The following command

```
orbiter.out -v 3 -linear_group -PGL 3 11 -singer 1 -end
```

Modifier	Arguments	Meaning
<code>-wedge</code>		action on the exterior square
<code>-PGL2OnConic</code>		induced action of $\text{PGL}(2, q)$ on the conic in the plane $\text{PG}(2, q)$
<code>-subfield_ structure_action</code>	s	action by field reduction to the subfield of index s
<code>-on_k_subspaces</code>	k	induced action on k dimensional subspaces
<code>-on_tensors</code>		induced action of $\text{GL}(d, q) \wr \text{Sym}(n)$ on the tensor space
<code>-on_rank_one_ tensors</code>		induced action of $\text{GL}(d, q) \wr \text{Sym}(n)$ on the tensor space
<code>-restricted_action</code>	s	restricted action on the set s

Table 9: Commands for creating induced or restricted group actions

can be used to create a subgroup known as the Singer cycle. The Singer cycle in $\text{GL}(d, q)$ 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 $\text{PG}(d - 1, q)$. The additional integer parameter k after the `-singer` command can be used to create a the subgroup of index k of the Singer cycle.

The command

```
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 $\text{PG}(2, 11)$, 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

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

creates the group $\text{GL}(2, 2) \wr \text{Sym}(4)$ acting on the 65535 elements of $\text{PG}(15, 2)$. The auxiliary points that result from the wreath product action are blended out. This is done by restricting the action to the points of $\text{PG}(15, 2)$.

It is possible to perform group theoretic tasks using the `-group_theoretic_activities` option. Tables 10 and 11 list the possible commands.

The command

```
orbiter.out -v 5 -linear_group -PGL 3 4 -end \
-group_theoretic_activities -find_singer_cycle
```

Modifier	Arguments	Meaning
<code>-orbits_on_subsets</code>	k	Compute orbits on k -subsets
<code>-orbits_on_points</code>		Compute orbits in the action that was created
<code>-orbits_of</code>	i	Compute orbit of point i in the action that was created
<code>-stabilizer</code>		Compute the stabilizer of the orbit representative (needs <code>-orbits_on_points</code>)
<code>-draw_poset</code>		Draw the poset of orbits (needs <code>-orbits_on_subsets</code>)
<code>-classes</code>		Compute a report of the conjugacy classes of elements (needs Magma [6])
<code>-normalizer</code>		Compute the normalizer (needs Magma [6]; needs a group with a subgroup)
<code>-report</code>		Produce a latex report about the group
<code>-syLOW</code>		Include Sylow subgroups in the report (needs <code>-report</code>)
<code>-print_elements</code>		Produce a printout of all group elements
<code>-print_elements_tex</code>		Produce a latex report of all group elements
<code>-group_table</code>		Produce the group table (needs <code>-report</code>)
<code>-orbits_on_set_system_from_file</code>	fname f l	reads the csv file “fname” and extract sets from columns $[f, \dots, f + l - 1]$
<code>-orbit_of_set_from_file</code>	fname	reads a set from the text file “fname” and computes orbits on the elements of the set
<code>-multiply</code>	s_1 s_2	Creates group elements from s_1 and s_2 and multiplies
<code>-inverse</code>	s	Creates a group element from s and computes its inverse
<code>-search_element_of_order</code>	s	Finds all elements of order s in the group
<code>-find_singer_cycle</code>		Finds all Singer cycles whose matrix is a companion matrix

Table 10: Group theoretic activities (Part 1)

Modifier	Arguments	Meaning
<code>-tensor_classify</code>	d	Classify tensors of tensor-rank at most d .
<code>-tensor_permutations</code>		Compute the permutation representation of generators of wreath product
<code>-classify_arcs</code>	$s \ r$	Classify (s, r) -arcs in $\text{PG}(2, q)$. If $r = 0$, classify all subsets of size s in $\text{PG}(2, q)$. See Section 11.
<code>-classify_nonconical_arcs</code>	$s \ r$	Classify nonconical (s, r) -arcs in $\text{PG}(2, q)$. If $r = 0$, classify all subsets of size s in $\text{PG}(2, q)$. See Section 11.

Table 11: Group theoretic activities (Part 2)

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 ω and 3 is $\omega + 1$.

The command

```
orbiter.out -v 4 \
  -linear_group -GL_d_q_wr_Sym_n 2 2 4 -on_tensors -end \
  -group_theoretic_activities \
  -orbits_on_points
pdflatex GL_2_2_wreath_Sym4_res65535_orbits.tex
open GL_2_2_wreath_Sym4_res65535_orbits.pdf
```

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 `GL_2_2_wreath_Sym4_res65535_orbits.tex` is created. It contains a list of the generators of the group and the orbits.

Through its interface to Magma [6], Orbiter can be used to compute the conjugacy classes of groups. For instance, the command

```
orbiter.out -v 3 -linear_group -PSL 3 2 -end \
  -group_theoretic_activities -classes
```

can be used to create a report about the conjugacy classes of the simple group $\text{PSL}(3, 2)$.

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 $SL(2, 3)$. 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 3 that Orbiter field elements are integers in the interval $[0, \dots, q - 1]$, 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" \
    "0,0,1,2,0,0,0,2,0" \
    -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 is 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 [18] lists generators for $U_3(3)$ as 3×3 matrices over the field \mathbb{F}_9 using the following short Magma [6] program:

```

F<w>:=GF(9);
x:=CambridgeMatrix(1,F,3,[
"164",
"506",
"851"]);
y:=CambridgeMatrix(1,F,3,[
"621",
"784",
"066"]);
G<x,y>:=MatrixGroup<3,F|x,y>;

```

The generators are given using the Magma command `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 \mathbb{F}_9 . Magma uses Conway polynomials to generate finite fields which are not of prime order. The Conway polynomial for \mathbb{F}_9 can be determined using the following Magma command (which can be typed into the Magma online calculator at [17])

```

F<w>:=GF(9);
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 `-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

```

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.

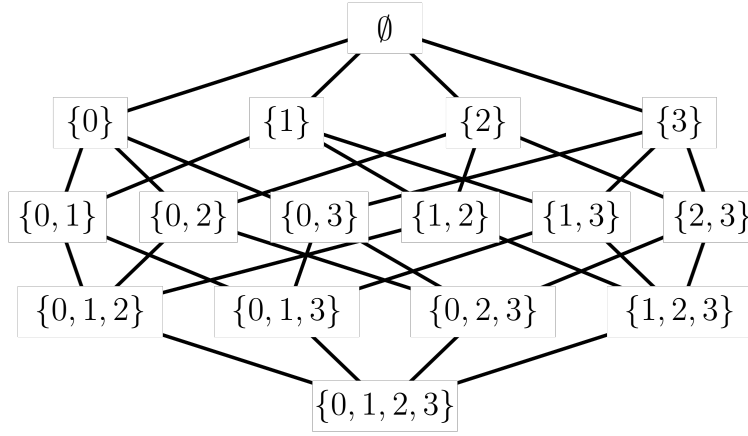


Figure 3: The lattice of subsets of a 4-element set

7 Orbits on Subsets

The lattice of subsets of a set X is $\mathfrak{P}(X)$, the set of all subsets of X , ordered with respect to inclusion. For instance, Figure 3 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 $\mathfrak{P}(X)$ but rather in a subposet of it. We require that the subposet is closed under the group action and that the following property holds:

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

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

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

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 4. All nodes except for the root node are labeled by elements of $X = \{0, 1, 2, 3\}$. In order to

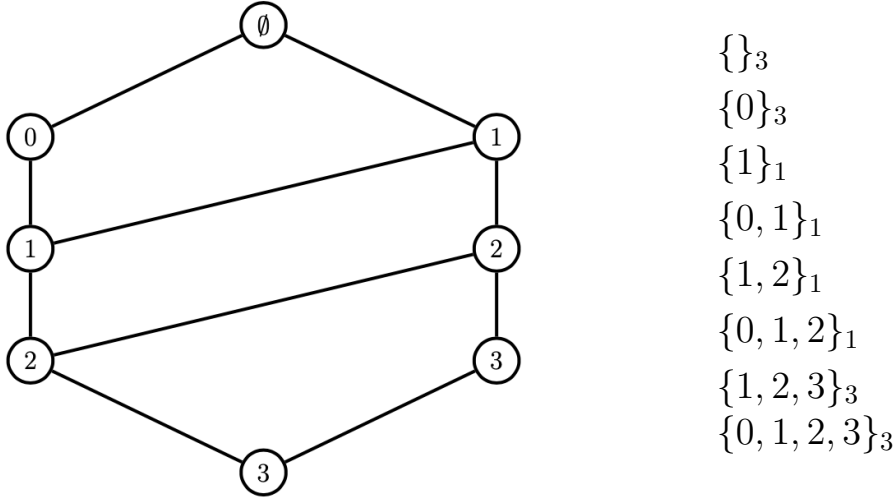


Figure 4: The poset of orbits under the Singer group

determine the set that is associated to a node, we follow the unique leftmost path to the root node and collect the node labels. This produces the set associated to the node. The orbit representatives are indicated next to the diagram. By convention, the order of the stabilizer is written as a subscript. Since this example is small enough, it is possible to show the complete orbits, as in Figure 5. Elements belonging to an orbit are grouped together. The leftmost element in each group is the orbit representative.

As the poset of orbits can get quite busy, it is often desired to replace it by the lex-least spanning tree. This is the tree that results by keeping only the links between a node and its lex-least ancestor, see Figure 6.

8 Orbits on Subspaces

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

Suppose we want to classify the subspaces in $\text{PG}(3, 2)$ under the action of the orthogonal group. The orthogonal group is the stabilizer of a quadric. In $\text{PG}(3, 2)$ there are two distinct nondegenerate quadrics, $\mathcal{Q}^+(3, 2)$ and $\mathcal{Q}^-(3, 2)$. The $\mathcal{Q}^+(3, 2)$ 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 7. $\text{PG}(3, 2)$ has 15 points:

$P_0 = (1, 0, 0, 0)$	$P_3 = (0, 0, 0, 1)$	$P_6 = (1, 0, 1, 0)$	$P_9 = (1, 0, 0, 1)$
$P_1 = (0, 1, 0, 0)$	$P_4 = (1, 1, 1, 1)$	$P_7 = (0, 1, 1, 0)$	$P_{10} = (0, 1, 0, 1)$
$P_2 = (0, 0, 1, 0)$	$P_5 = (1, 1, 0, 0)$	$P_8 = (1, 1, 1, 0)$	$P_{11} = (1, 1, 0, 1)$

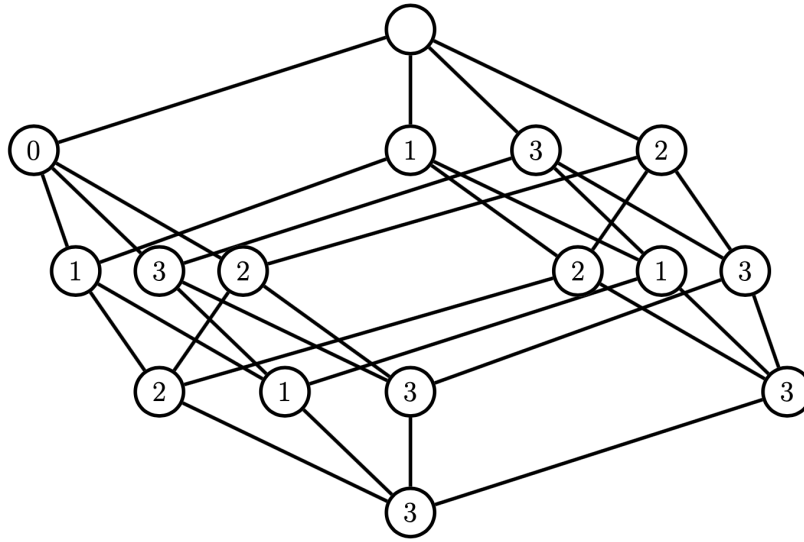


Figure 5: The poset with orbits indicated by grouping

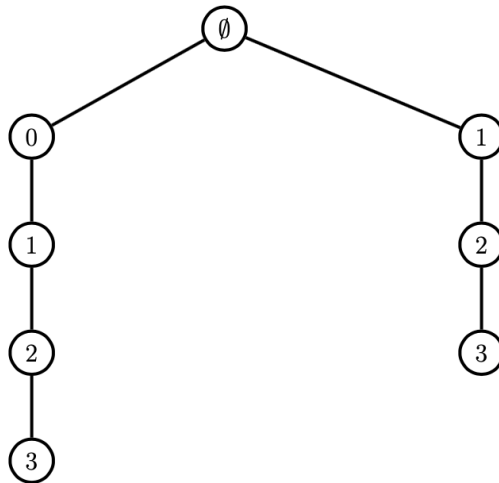


Figure 6: The lex-least spanning tree for the poset of orbits

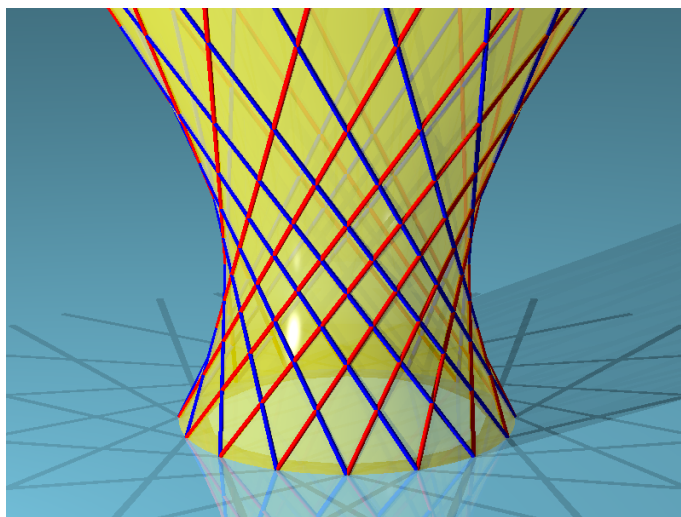


Figure 7: The hyperbolic quadric in affine space \mathbb{R}^3

$$P_{12} = (0, 0, 1, 1) \quad P_{13} = (1, 0, 1, 1) \quad P_{14} = (0, 1, 1, 1)$$

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

$$P_0, P_1, P_2, P_3, P_4, P_6, P_7, P_9, P_{10}.$$

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

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

produces a classification of all subspaces of $\text{PG}(3, 2)$ under $\text{PGO}^+(4, 2)$. The option `-draw_poset` creates a Hasse diagram of the classification as shown Figure 8. The Hasse diagram is the poset of orbits of the group on the subspace lattice.

9 Graph Theory

Orbiter is able to work with algebraically defined graphs. It can also construct and classify graphs up to isomorphism. In Table 12, command line arguments are shown for some of the graphs that Orbiter can create. 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. Regarding the problem of classifying small graphs, Table 13 lists the command line options that are available. Here is an example:

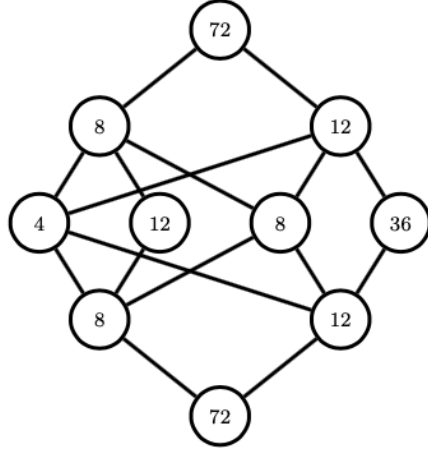


Figure 8: Hasse-diagram for the orbits of the orthogonal group $O^+(4, 2)$ on subspaces of $\text{PG}(3, 2)$

Key	Arguments	Meaning
-Johnson	$n \ k \ s$	Johnson graph
-Paley	q	Paley graph
-Sarnak	$p \ q$	Lubotzky-Phillips-Sarnak graph [11]
-Schlaefli	q	Schlaefli graph
-Shrikhande		Shrikhande graph
-Winnie_Li	$q \ i$	Winnie-Li graph [10]
-Grassmann	$n \ k \ q \ r$	Grassmann graph
-coll_orthogonal	$\epsilon \ d \ q$	Collinearity graph of $O^\epsilon(d, q)$

Table 12: Types of graphs

Option	Arguments	Meaning
-girth	d	Girth at least d
-regular	r	Regular of degree r
-no_transmitter		Tournament without transmitter (requires -tournament)

Table 13: Options for classifying graphs

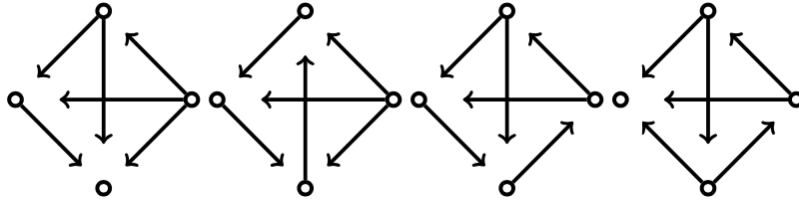


Figure 9: The four isomorphism types of tournaments on 4 vertices

```
orbiter.out -v 2 -graph_classify -n 4
```

classifies all graphs with 4 vertices. For this, the set $X = \binom{V}{2}$ is considered, where $V = \{0, 1, \dots, n-1\}$. The poset \mathcal{P} is the lattice of subsets of X . The group action of $\text{Sym}(V)$ induces an action on the lattice. For tournaments, the option `-tournament` can be added. In this case, $X = V^{[2]}$ is the set of ordered pairs of elements from V . The ordered pair (a, b) represents the fact that a beats b in the tournament. Again, the poset \mathcal{P} is the lattice of subsets of X and the group $\text{Sym}(V)$ acts in the induced action on ordered pairs. For example,

```
orbiter.out -v 2 -graph_classify -n 4 -v 2 -tournament -draw_graphs_at_level 6
```

classifies the tournaments on 6 vertices. The `-draw_graphs_at_level 6` option instructs Orbiter to draw all representatives at level 6. Figure 9 shows the resulting list of 4 tournaments.

10 Projective Geometry

Orbiter can be used to classify sets in projective space and to compute their collineation stabilizer. Here is an example. We consider the following three Orbiter commands:

```
orbiter.out -v 5 -create_combinatorial_object -q 11 \
  -elliptic_curve 1 3 -end \
  -save "./"
orbiter.out -v 2 -process_combinatorial_objects \
  -draw_points_in_plane EC_11_1_3 -q 11 -n 2 \
  -fname_base_out EC_11_1_3 -embedded \
  -input -file_of_points elliptic_curve_b1_c3_q11.txt \
  -end \
  -end
orbiter.out -v 2 -canonical_form_PG 2 11 \
  -input \
  -file_of_points elliptic_curve_b1_c3_q11.txt \
  -end \
  -prefix PG_2_11_EC \
  -save elliptic_curve_b1_c3_q11_classified \
  -report
```


The first command creates an elliptic curve over a finite field. Specifically, it creates the curve whose affine equation is

$$y^2 \equiv x^3 + x + 3 \pmod{11}.$$

The curve turns out to have exactly 18 points in $\text{PG}(2, 11)$. The second command produces a picture of the point set in $\text{PG}(2, 11)$, shown in Figure 2. The third command computes the collineation stabilizer. This is done by using the techniques of canonical forms in graphs, using the Nauty [12] package which is included in Orbiter. Orbiter shows that the curve has a collineation stabilizer of order 6, generated by

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 8 \\ 5 & 9 & 5 \\ 8 & 1 & 1 \end{bmatrix}.$$

The types of objects that can be created using `-create_combinatorial_object` are listed in Tables 14 and 15. Modifier options that apply to multiple options are listed in Table 16.

The purpose of the `-process_combinatorial_objects` option is to perform various jobs for data streams. A data stream represents a possibly large number of integer data sets, stored in files. Using the rank and unrank encoding from Section 4, integer data can encode sets of points in projective space, for instance. The types of jobs are listed in Table 17.

11 Arcs in Projective Planes

A (k, d) -arc in a projective plane π 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 Φ 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. 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 $\text{P}\Gamma\text{L}(3, 2^e)$. The command

```
orbiter.out -v 4 \
  -linear_group -PGGL 3 16 -end \
  -group_theoretic_activities \
    -poset_classification_control -problem_label arcs_q16_2 -W -end \
    -classify_arcs 18 2 \
  -end
```

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

Key	Arguments	Purpose
-hyperoval		To create a hyperoval
-subiaco_oval	f_short	Create points of the Subiaco oval
-subiaco_hyperoval		Create points of the Subiaco hyperoval
-adelaide_hyperoval		Create points of the Adelaide hyperoval
-translation	i	Create points of the translation hyperoval with exponent i
-Segre		Create points of the Segre hyperoval
-Payne		Create points of the Payne hyperoval
-Cherowitzo		Create points of the Cherowitzo hyperoval
-O’Keefe_Penttila		Create points of the O’Keefe, Penttila hyperoval
-BLT_database	k	The k th BLT-set of order q from the database ($k = 0, 1, \dots$)
-ovoid		Create points of an ovoid
-Baer		Create points of the (standard) Baer subgeometry
-orthogonal	ϵ	Create points of the $Q^\epsilon(n, q)$ quadric
-hermitian		Create points of Hermitian variety given by $\sum_{i=0}^n X_i^{\sqrt{q}+1} = 0$
-cubic		Create points of a cubic
-twisted_cubic		Create points of the twisted cubic
-elliptic_curve	$a \ b$	Create points of the elliptic curve $y^2 = x^3 + ax + b$
-ttp_construction_A		Create points of the twisted tensor product code of type A [2]
-ttp_construction_A_hyperoval		Create points of the twisted tensor product code of type A
-ttp_construction_B		Create points of the twisted tensor product code of type B

Table 14: Orbiter Objects (Part 1)

Key	Arguments	Purpose
-unital_ XXq_YZq_ZYq		Create points of the unital with equation $XX^q + YZ^q + ZY^q = 0$
-desarguesian_ line_spread_ in_PG_3_q		Create the desarguesian line spread in $\text{PG}(3, q)$ as a set of 2-subspaces
-Buekenhout_Metz		Create points of the Buekenhout Metz unital
-Uab	$a \ b$	Create points of the Buekenhout Metz unital in the form of Barwick and Ebert [1]
-whole_space		Create points of the whole space
-hyperplane	pt	Create points of the hyperplane given by dual coordinates associated with the given point
-segre_variety	$a \ b$	Create points of the Segre variety
-Maruta_Hamada_arc		Create points of the Maruta Hamada arc
-projective_variety	$l \ d \ \mathcal{C}$	Create points of the projective variety of degree d with label l , with coefficient vector \mathcal{C}
-projective_curve	$l \ r \ d \ \mathcal{C}$	Create points of the projective curve of degree d with label l , with coefficient vector \mathcal{C} in r variables

Table 15: Orbiter Objects (Part 2)

Key	Arguments	Purpose
-q	q	The size of the finite field \mathbb{F}_q
-Q	Q	The field size of the extension field \mathbb{F}_Q
-n	n	The projective dimension
-poly	r	Use polynomial with rank r to create the field \mathbb{F}_q
-poly_Q	r	Use polynomial with rank r to create the field \mathbb{F}_Q
-embedded_in_PG_4_q		
-BLT_in_PG		Create points of the BLT-set with ranks in $\text{PG}(n, q)$ instead of orthogonal point ranks

Table 16: Orbiter Objects: Modifiers

Job key	Purpose
-dualize_hyperplanes_to_points	Turns ranks of hyperplanes into ranks of points
-dualize_points_to_hyperplanes	Turns ranks of points into ranks of hyperplanes
-ideal	Compute the ideal of a set of points
-homogeneous_polynomials	Prints the equation whose coefficient vector is the input vector
-canonical_form	Computes the canonical form of a set
-draw_points_in_plane	Produces a drawing of a set of points in a projective plane
-klein	Applies the Klein correspondence
-line_type	Computes the line type
-plane_type	Computes the plane type
-conic_type	Computes the conic type
-hyperplane_type	Computes the hyperplane type
-intersect_with_set_from_file	Computes the intersection with a set specified in a file
-arc_with_given_set_as_s_lines_after_dualizing	Finds arcs with the given set as s -lines
-arc_with_two_given_sets_of_lines_after_dualizing	Finds arcs with the two given sets as s -lines and t -lines, respectively
-arc_with_three_given_sets_of_lines_after_dualizing	Finds arcs with the three given sets as s -lines and t -lines and u -lines, respectively

Table 17: Orbiter Jobs in $\text{PG}(n, q)$

12 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 $\text{PTL}(4, q)$. One approach is described in [5] and relies on Schlaefli's notion of a double six as a substructure [15]. Another approach is through non-conical six-arcs in a plane, as described in [9]. 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 surfaces but a surface may have several double sixes associated to it. The classification algorithms 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. Also, 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, the five-plus-one configurations of lines correspond to certain configurations of points on the Klein quadric.

The command

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

to perform the same classification with respect to the collineation group $\text{PTL}(4, 4)$. The `-report` option can be used to create a report of the classified surfaces. So, for instance

```
orbiter.out -v 3 -linear_group -PGGL 4 4 -wedge -end \
  -group_theoretic_activities -surface_classify -report
```

produces a latex report of the surface in $\text{PG}(3, 4)$. 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)

$$X_0^2 X_3 + X_1^2 X_2 + X_1 X_2^2 + X_0 X_3^2 + X_1 X_2 X_3 = 0 \quad (4)$$

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:

$$\begin{bmatrix} 1 & 4 & 4 & 0 \\ 6 & 0 & 0 & 0 \\ 6 & 2 & 0 & 1 \\ 7 & 0 & 4 & 0 \end{bmatrix}_0. \quad (5)$$

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

$$\begin{aligned} 0 &= \alpha^3 X_0^2 X_2 + \alpha^3 X_1^2 X_2 + \alpha^3 X_1^2 X_3 + \alpha^3 X_0 X_2^2 + \alpha^3 X_1 X_2^2 + \alpha^3 X_2^2 X_3 \\ &\quad + \alpha^2 X_1 X_3^2 + \alpha^2 X_2 X_3^2 + X_0 X_2 X_3 + X_1 X_2 X_3, \\ 0 &= X_0^2 X_3 + X_1^2 X_2 + X_1 X_2^2 + X_0 X_3^2 + X_1 X_2 X_3. \end{aligned}$$

The isomorphism is given as a collineation:

$$\begin{bmatrix} 2 & 3 & 0 & 0 \\ 7 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 2 & 3 & 2 & 4 \end{bmatrix}_2.$$

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.

A second algorithm to classify cubic surfaces has been described in [4] and in [9]. This algorithm is available in Orbiter also. For instance, the command

```
orbiter.out -v 3 -linear_group -PGL 4 13 -end \
  -group_theoretic_activities \
  -classify_surfaces_through_arcs_and_triangular_pairs 13
```

classifies all cubic surfaces with 27 lines over the field \mathbb{F}_{13} using this algorithm. A report of the classification is produced in the file `arc_lifting_q13.tex`.

Besides classification, there are two further ways to create surfaces in Orbiter. The first is a built-in catalogue of cubic surfaces with 27 lines for small finite fields \mathbb{F}_q (at the moment, $q \leq 101$ is required). The second is a way of creating members of known infinite families. Both are facilitated using the `-create_surface` option. For instance,

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

creates the member of the Hilbert, Cohn-Vossen surface described in [5] with parameter $a = 3$ and $b = 1$ over the field \mathbb{F}_{13} . The command

```
orbiter.out -v 3 -linear_group -PGGL 4 8 -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. 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

```
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.

13 The Povray Interface

Orbiter can be used to create high quality raytracing graphics. Orbiter serves as a front end for 3D graphics processed through Povray [14]. 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. Tables 18 and 19 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 20 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 19 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 \
```

Command	Arguments	Purpose
-cubic_lex	coeffs	cubic surface given by 20 coefficients in lexicographic ordering
-cubic_orbiter	coeffs	cubic surface given by 20 coefficients in Orbiter ordering
-cubic_Goursat	$A B C$	cubic surface with tetrahedral symmetry given by 3 Goursat coefficients as $Axyz + B(x^2 + y^2 + z^2) + C = 0$
-quadric_lex_10	coeffs	quadric surface given by 10 coefficients in lexicographic ordering
-quartic_lex_35	coeffs	quartic surface given by 35 coefficients in lexicographic ordering
-point	coeffs	point given by three coordinates
-point_list_from_csv_file	fname	List of points with coordinates given in a csv file
-line_through_two_points_recentered_from_csv_file	fname	List of lines through two points with point coordinates given in a csv file
-line_through_two_points_from_csv_file	fname	List of lines through two points with point coordinates given in a csv file
-point_as_intersection_of_two_lines	$i_1 i_2$	Create a point from the intersection of two lines i_1 and i_2
-edge	$i_1 i_2$	Create an edge (line segment) between points i_1 and i_2
-text	$i_1 s$	Create a label s located at the point i
-triangular_face_given_by_three_lines	$i_1 i_2 i_3$	Create a triangular face give by three lines i_1, i_2, i_3
-face	pts	Create a face through the vertices pts, ordered cyclically
-quadric_through_three_skew_lines	$i_1 i_2 i_3$	Create a quadric through three skew lines
-plane_defined_by_three_points	$i_1 i_2 i_3$	Create a plane through three noncollinear points
-line_through_two_points_recentered	pt-coords	Create a line through two points given by 6 coordinates, recentered
-line_through_two_points	pt-coords	Create a line through two points given by 6 coordinates

Table 18: Scene definition commands (part 1)

Command	Arguments	Purpose
-line_through_two_existing_points	$i_1 i_2$	Create a line through two points
-line_through_point_with_direction	coeffs6	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
-plane_by_dual_coordinates	coeffs4	Create a plane $ax + by + cz + d = 0$ give four dual coordinates as coeff4 = a, b, c, d
-dodecahedron		Create a Dodecahedron centered at the origin (20 points, 30 edges, 12 faces)
-Hilbert_Cohn_Vossen_surface		Create the Hilbert, Cohn-Vossen surface (1 cubic surface, 45 tritangent planes, 27 lines)
-obj_file	fname	Read points and faces from the given .obj file
-group_of_things	list	Create a group of things from the given list
-group_of_things_with_offset	list offset	Create a group of things from the given list, each value is increase by offset
-group_of_things_as_interval	$a b$	Create a group of things from the interval $a, \dots, a + b - 1$
-group_of_things_as_interval_with_exceptions	$a b$ ex	Create a group of things from the interval $a, \dots, a + b - 1$ with the exceptional elements in the list ex removed
-group_of_all_points		Create a group of things from all points currently defined
-group_of_all_faces		Create a group of things from all faces currently defined
-group_subset_at_random	$i f$	Create a group of things from the existing group i by picking a random subset with probability f
-create_regulus	$i N$	Create a regulus for quadric i with N lines

Table 19: Scene definition commands (part 2)

Command	Arguments	Purpose
-spheres	i r prop	For each element in point group i , create a sphere of radius r with given Povray properties.
-cylinders	i r prop	For each element in edge group i , create a cylinder of radius r with given Povray properties.
-prisms	i d prop	For each element in face group i , create a prism of half-thickness d with given Povray properties.
-planes	i prop	For each element in plane group i , create a plane with given Povray properties.
-lines	i r prop	For each element in line group i , create a line of radius r with given Povray properties.
-cubics	i prop	For each element in group i of cubics, create a surface with given Povray properties.
-quadrics	i prop	For each element in group i of quadrics, create a surface with given Povray properties.
-quartics	i prop	For each element in group i of quartics, create a surface with given Povray properties.
-texts	i d s prop	For each element in group i of labels, create a text element with half-thickness d and size s with given Povray properties.

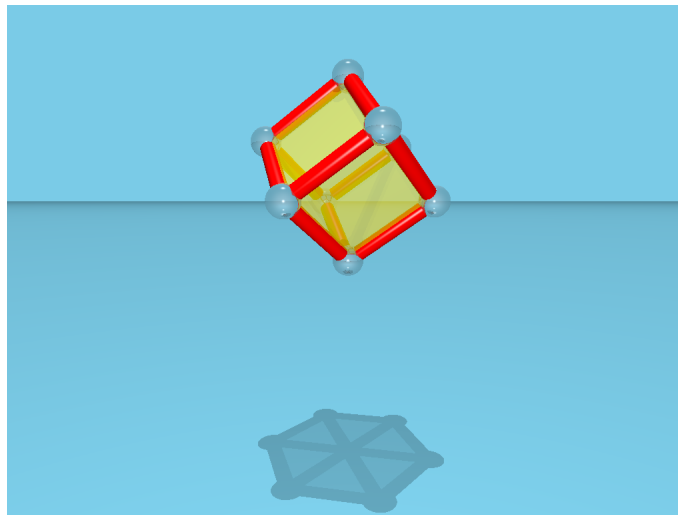
Table 20: Graphical output commands

```

6      -scene_objects \
7      -obj_file cube_centered.obj \
8      -edge "0, 1" \
9      -edge "0, 2" \
10     -edge "0, 4" \
11     -edge "1, 3" \
12     -edge "1, 5" \
13     -edge "2, 3" \
14     -edge "2, 6" \
15     -edge "3, 7" \
16     -edge "4, 5" \
17     -edge "4, 6" \
18     -edge "5, 7" \
19     -edge "6, 7" \
20     -group_of_things_as_interval 0 8 \
21     -spheres 0 0.3 "texture{ PolishedChrome pigment{quick.color
    White} }" \
22     -group_of_things_as_interval 0 6 \
23     -prisms 1 0.05 "texture{ pigment{ color Yellow transmit 0.7
    } finish {diffuse 0.9 phong 0.6} }" \
24     -group_of_things_as_interval 0 12 \
25     -cylinders 2 0.15 "texture{ pigment{ color Red } finish {di
    ffuse 0.9 phong 0.6} }" \
26     -scene_objects_end \
27     -povray_end
28

```

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:



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.

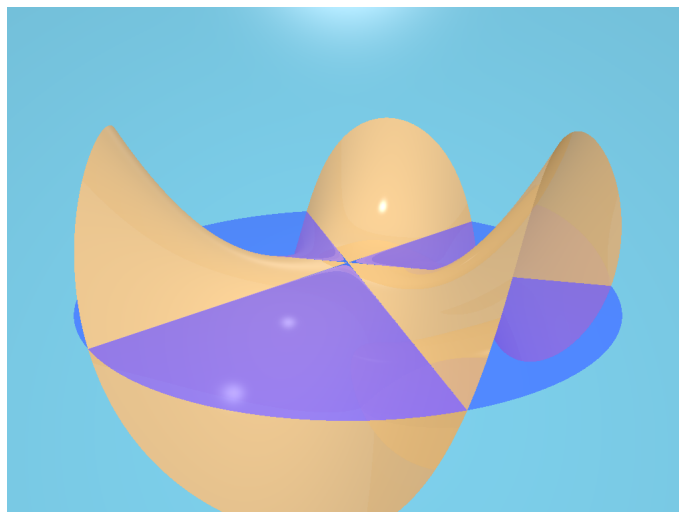
```

1      orbiter.out -v 2 -povray \
2      -round 0 -nb_frames_default 30 -output_mask monkey_%d.%03d.pov \

3      -video_options -W 1024 -H 768 -global_picture_scale 0.8 \
4      -default_angle 75 -clipping_radius 0.8 \
5      -camera 0 "0,0,1" "1,1,0.5" "0,0,0" \
6      -rotate_about_z_axis \
7      -end \
8      -scene_objects \
9      -cubic_lex "1,0,0,0,-3,0,0,0,0,0,0,0,0,0,0,0,0,-1,0" \
10     -plane_by_dual_coordinates "0,0,1,0" \
11     -group_of_things "0" \
12     -group_of_things "0" \
13     -cubics 0 "texture{ pigment{ Gold } finish {ambient 0.4 diffus
14     e 0.5 roughness 0.001 reflection 0.1 specular .8} }" \
15     -planes 1 "texture{ pigment{ color Blue transmit 0.5 } finish
16     { diffuse 0.9 phong 0.2} }" \
17     -scene_objects_end \
18     -povray_end

```

Here is one of the frames that are created:



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

$$\frac{5}{2}xyz - (x^2 + y^2 + z^2) + 1 = 0.$$

```

1      orbiter.out -v 2 -povray \
2          -round 0 -nb_frames_default 30 -output_mask HCV_%d_%03d.pov \
3          -video_options -W 1024 -H 768 -global_picture_scale 0.9 \
4          -default_angle 75 -clipping_radius 2.4 \
5          -camera 0 "1,1,1" "-3,1,3" "0.12,0.12,0.12" \
6          -end \
7          -scene_objects \
8              -Hilbert_Cohn_Vossen_surface \
9              -group_of_things "0" \
10                 -cubics 0 "texture{ pigment{ White*0.5 transmit 0.5 } finish
{ambient 0.4 diffuse 0.5 roughness 0.001 reflection 0.1 specular
.8} }" \
11                 -group_of_things_as_interval 0 6 \
12                 -group_of_things_as_interval 6 6 \
13                 -group_of_things_as_interval_with_exceptions 12 15 "14,19,23
" \
14                 -lines 1 0.02 "texture{ pigment{ color Red } finish { diffus
e 0.9 phong 1}}" \
15                 -lines 2 0.02 "texture{ pigment{ color Blue } finish { diffu
se 0.9 phong 1}}" \
16                 -lines 3 0.02 "texture{ pigment{ color Yellow } finish { dif
fuse 0.9 phong 1}}" \
17                 -label 0 "a1" -label 2 "a2" -label 4 "a3" \
18                 -label 6 "a4" -label 8 "a5" -label 10 "a6" \
19                 -label 12 "b1" -label 14 "b2" -label 16 "b3" \
20                 -label 18 "b4" -label 20 "b5" -label 22 "b6" \
21                 -label 24 "c12" -label 26 "c13" -label 30 "c15" \

```

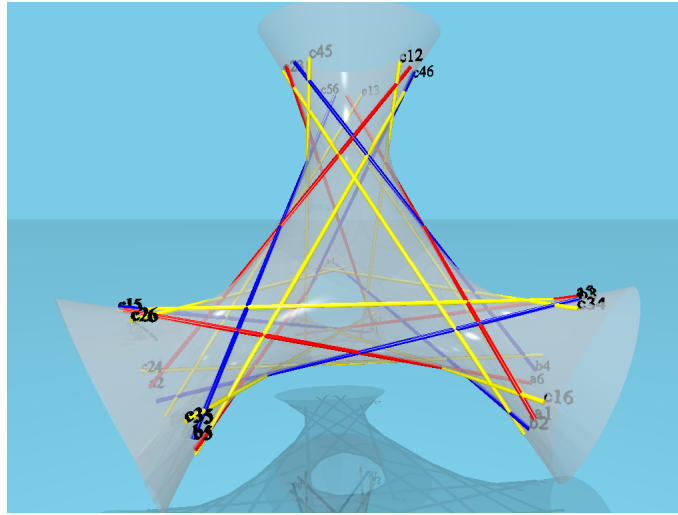


Figure 10: The Hilbert Cohn-Vossen surface

```

22      -label 32 "c16" -label 34 "c23" -label 36 "c24" \
23      -label 40 "c26" -label 42 "c34" -label 44 "c35" \
24      -label 48 "c45" -label 50 "c46" -label 52 "c56" \
25      -group_of_things_as_interval 0 6 \
26      -texts 4 0.2 0.15 "texture{ pigment{Black} } no_shadow" \
27      -group_of_things_as_interval 6 6 \
28      -texts 5 0.2 0.15 "texture{ pigment{Black} } no_shadow" \
29      -group_of_things_as_interval 12 12 \
30      -texts 6 0.2 0.15 "texture{ pigment{Black} } no_shadow" \
31      -scene_objects_end \
32      -povray_end
33

```

Figure 10 shows the final product. The Schläefli 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:

$$\begin{aligned}\sin(x) &\mapsto \text{push } x \text{ sin,} \\ a + b &\mapsto \text{push } a \text{ push } b \text{ add,} \\ a \cdot b &\mapsto \text{push } a \text{ push } b \text{ mult.}\end{aligned}$$

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

```
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 ϵ apart. The code to produce the plot is

```
1 orbiter.out -v 2 -povray \
2   -round 0 -nb_frames_default 1 -output_mask lissajous_%d_%03d.pov
   \
3   -video_options -W 1024 -H 768 -global_picture_scale 0.40 \
4   -default_angle 45 -clipping_radius 5 -omit_bottom_plane \
5   -camera 0 "0,-1,0" "0,0,12" "0,0,0" \
```

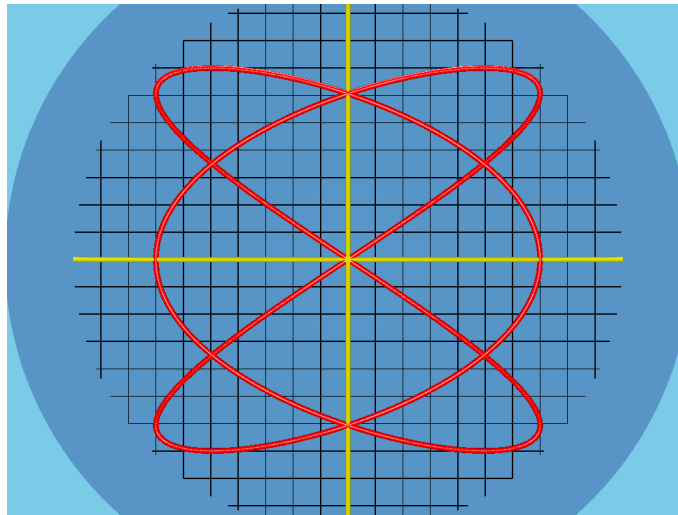


Figure 11: Lissajous figure

```

6      -rotate_about_z_axis \
7      -end \
8      -scene_objects \
9      -line_through_two_points_recentered_from_csv_file coordinate_g
rid.csv \
10     -group_of_things "0" \
11     -group_of_things "1" \
12     -group_of_things "2" \
13     -lines 0 0.09 "texture{ pigment{ color Yellow } }" \
14     -lines 1 0.09 "texture{ pigment{ color Yellow } }" \
15     -lines 2 0.09 "texture{ pigment{ color Yellow } }" \
16     -group_of_things_as_interval 3 39 \
17     -lines 3 0.02 "texture{ pigment{ color Black } }" \
18     -point_list_from_csv_file function_lissajous_N2000_points.csv
\
19     -group_of_things_as_interval 0 6524\
20     -spheres 4 0.1 "texture{ pigment{ color Red } finish { diffuse
0.9 phong 1}}" \
21     -plane_by_dual_coordinates "0,0,1,0" \
22     -group_of_things "0" \
23     -planes 5 "texture{ pigment{ color Blue*0.5 transmit 0.5 } }"
\
24     -scene_objects_end \
25     -povray_end
26

```

The plot is shown in Figure 11. 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


```

1      orbiter.out -v 2 -povray \
2          -round 0 -nb_frames_default 30 -output_mask lissajous_3d_%d_%03d
        .pov \
3          -video_options -W 1024 -H 768 -global_picture_scale 0.40 \
4          -default_angle 45 -clipping_radius 5 -omit_bottom_plane \
5          -camera 0 "0,0,1" "7,7,5" "0,0,1" \
6          -rotate_about_z_axis \
7          -end \
8          -scene_objects \
9              -line_through_two_points_recentered_from_csv_file coordinate_g
rid.csv \
10             -group_of_things "0" \
11             -group_of_things "1" \
12             -group_of_things "2" \
13             -lines 0 0.09 "texture{ pigment{ color Yellow } }" \
14             -lines 1 0.09 "texture{ pigment{ color Yellow } }" \
15             -lines 2 0.09 "texture{ pigment{ color Yellow } }" \
16             -group_of_things_as_interval 3 39 \
17             -lines 3 0.02 "texture{ pigment{ color Black } }" \
18             -point_list_from_csv_file function_lissajous_3d.N2000.points.c
sv \
19             -group_of_things_as_interval 0 6538\
20             -spheres 4 0.1 "texture{ pigment{ color Red } finish { diffuse
0.9 phong 1}}" \
21             -plane_by_dual_coordinates "0,0,1,0" \
22             -group_of_things "0" \
23             -scene_objects_end \
24             -povray_end
25

```

The function is computed using the command

```

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 \

```

The 3D curve is shown in Figure 12.

14 Cryptography and Number Theory

In Table 21, some number theoretic commands are shown. For instance,

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

Table 21: Number Theoretic Commmands

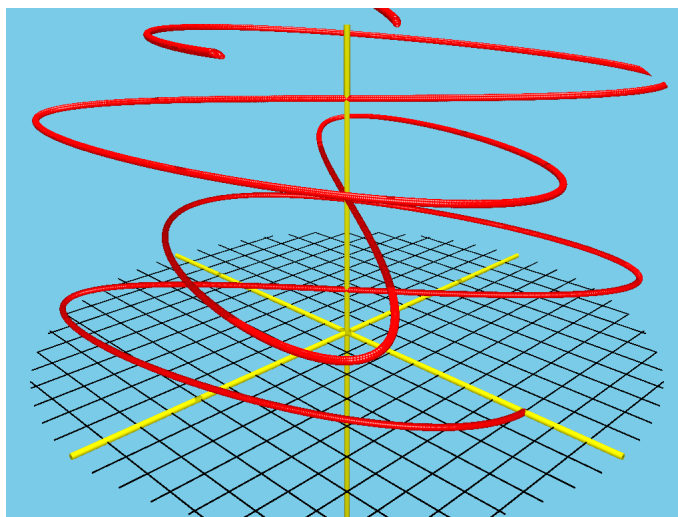


Figure 12: Lissajous Spacecurve

```
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.

In Table 22, 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 $y^2 = x^3 + x + 3 \pmod{11}$ 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 $y^2 = x^3 + x + 3 \pmod{11}$. The command

```
orbiter.out -v 2 -EC_points 199 5 7
```

computes all points on the curve $y^2 = x^3 + 5x + 7 \pmod{199}$. 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 $y^2 = x^3 + 5x + 7 \pmod{199}$ 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

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

Table 22: Cryptographic Commmands

```
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 $y^2 = x^3 + 5x + 7 \pmod{199}$, 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 $y^2 = x^3 + 5x + 7 \pmod{199}$, and assuming a group order of 212.

15 Coding Theory

Orbiter can classify linear codes with bounded minimum distance. To do so, Orbiter establishes a certain matroid in a suitable projective geometry and computes the orbits under the projective group. Recall that a linear $[n, k]$ -code \mathcal{C} over \mathbb{F}_q is a k -dimensional subspace of \mathbb{F}_q^n . The code has minimum distance d if

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

where $d(\mathbf{x}, \mathbf{y})$ is the Hamming metric on \mathbb{F}_q^n , which counts the number of positions where \mathbf{x} and \mathbf{y} differ. It is desirable to have both k and d large with respect to n . However, these are contradicting aims: A large value of k limits the quantity d . Likewise, a large value of d limits k . The notion of isometry with respect to the Hamming metric 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 23, some coding theoretic commands of Orbiter are shown. For instance, the command

Command	Arguments	Description
-RREF	$q \ m \ n$ list-of-integers	Compute the RREF of the $m \times n$ matrix over \mathbb{F}_q
-nullspace	$q \ m \ n$ list-of-integers	Compute a basis for the right nullspace of the given $m \times n$ matrix
-normalize_from_the_right		Normalizes the result of -RREF or nullspace from the right
-weight_enumerator	$q \ m \ n$ list-of-integers	Computes the weight enumerator of the linear code generated by the given $m \times n$ matrix
-BCH	$n \ q \ t$	Creates the BCH-code of length n over the field \mathbb{F}_q with designed distance t

Table 23: Coding Theoretic Commmands

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

creates a binary BCH-code of length 15 with minimum distance at least 3. The generator matrix produced by this command is

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

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 with a lower bound on d . We wish to use Orbiter for solving this problem for small instances.

Orbiter reduces the problem of classifying $[n, k, \geq d]$ codes over \mathbb{F}_q to an equivalent problem in finite geometry. It is well known (see, for instance [3]) that the equivalence classes of $[n, k, \geq d]$ codes over \mathbb{F}_q for $d \geq 3$ are in canonical one-to-one correspondence to the sets of size n in $\text{PG}(n, k-1, q)$ with the property that any set of size at most $d-1$ is linearly dependent. Let $\Lambda_{m,s}(q)$ be the the poset of subsets of $\text{PG}(m, q)$ such that any set of s or less

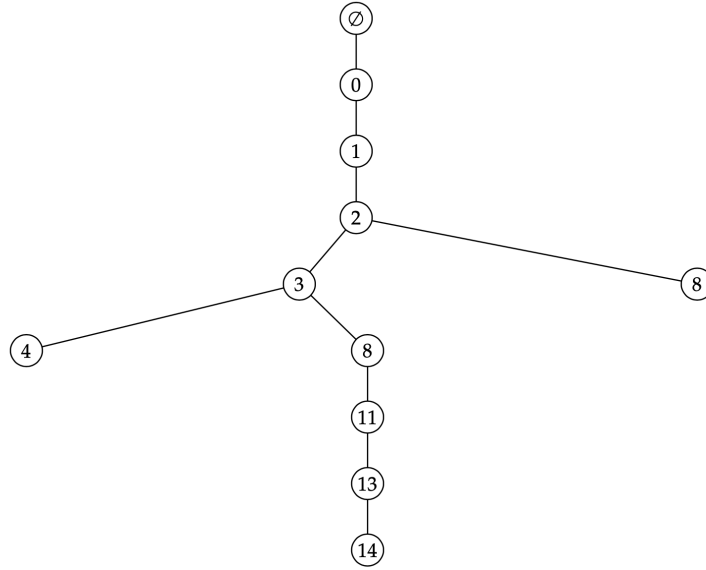


Figure 13: Orbits of $\text{PGL}(4, 2)$ on the poset $\Lambda_{3,3}(2)$,

points is independent. The group $G = \text{PGL}(m+1, q)$ acts on this poset. For $m = n - k - 1$ and $s = d - 1$, the orbits of G on sets in $\Lambda_{m,s}(q)$ of size n are in canonical one-to-one correspondence to the $[n, k, \geq d]$ codes over \mathbb{F}_q .

The Orbiter command

```
orbiter.out -v 2 -code_classify -n 8 -k 4 -q 2 -d 4 -lex
```

can be used to classify the $[8, 4, 4]$ codes over \mathbb{F}_2 . It turns out that there is exactly one such code, the $[8, 4, 4]$ extended Hamming code. Using the group $\text{PGL}(4, 2)$ acting on the poset $\Lambda_{3,3}(2)$, Orbiter produces the poset of orbits shown in Figure 13. In this diagram, the numbers stand for Orbiter numbers 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 $[8, 4, 4]$ -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 like so:

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

This matrix is the parity check matrix H of the code \mathcal{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.

16 Diophantine Systems

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

$$n = a_1 + a_2 + \dots + a_k, \quad a_1 \geq a_2 \geq \dots \geq a_k \geq 1,$$

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 described 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

$$[10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \mid 10].$$

Orbiter creates this diophantine system using the command

```
orbiter.out -v 4 -diophant -from_scratch part10 1 10 \
  -coefficient_matrix "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 -from_scratch part10 1 10 \
  -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 [16].

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, \dots, a_n) is the line type of (S, \mathcal{L}) . The equation

$$\binom{n}{2} = \sum_{j=2}^n a_j \binom{j}{2} \quad (7)$$

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, \dots, 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

$$[15 \ 10 \ 6 \ 3 \ 1 \mid 15].$$

The Orbiter command

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

creates this system and stores it to 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` [13]. The program finds 15 solutions, written to the file `linsp6.sol`. In Table 24, Orbiter commands for creating diophantine systems are shown. In Table 25, Orbiter activities for diophantine systems are shown.

17 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

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

Table 24: Orbiter Commmands to create Diophantine systems

Command	Arguments	Description
<code>-input_file</code>	file	Specify the input file
<code>-print</code>		Print the system
<code>-solve_mckay</code>		Solve the system using McKay's pos-solve
<code>-solve_standard</code>		Solve the system using the standard solver
<code>-draw</code>		Produce a metapost drawing of the coefficient matrix of the system
<code>-perform_column_reductions</code>		Eliminate variables which must be zero and write a reduced system

Table 25: Orbiter activities for Diophantine systems

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:

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

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

The program also prints out the tactical decomposition schemes of the design, which are

$$\begin{array}{c|c} \rightarrow & 13_1 \\ \hline 13_0 & 4 \end{array} \quad \begin{array}{c|c} \downarrow & 13_1 \\ \hline 13_0 & 4 \end{array}$$

18 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 $\text{PG}(2, 16)$, 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}
=====
52_{ 0} |      17      0
221_{ 3} |      13      4

col_scheme at level 3 :
is 1 x 2
      | 221_{ 1} 52_{ 2}
=====
273_{ 0} |      17      17
```

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 \
  -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} |

row_scheme at level 4 :
is 2 x 2
```

	221_{ 1}	52_{ 2}	
=====			
52_{ 0}		17	0
221_{ 3}		13	4

col_scheme at level 4 :
is 2 x 2

	221_{ 1}	52_{ 2}	
=====			
52_{ 0}		4	0
221_{ 3}		13	17

extra_col_scheme at level 3 :
is 1 x 2

	221_{ 1}	52_{ 2}	
=====			
273_{ 0}		17	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:

```
<HTDO type=bt ptanz=2 btanz=2 fuse=simple>
```

	221	52	
52	4	0	
221	13	17	

```
1 1
</HTDO>
```

This decomposition is block-tactical (note the **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 `max_arc_q16_r4bd.stack`:

```
<HTDO type=bt ptanz=2 btanz=3 fuse=simple>
```

	221	51	1	
52	4	0	0	
221	13	17	17	

```
3
</HTDO>
```

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:

```
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}
=====
52_{ 0} |      17      0
221_{ 3} |      13      4
```

```
col_scheme at level 5 :
```

```
is 2 x 3
      | 221_{ 1} 51_{ 2} 1_{ 4}
=====
52_{ 0} |      4      0      0
221_{ 3} |      13     17     17
```

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				0.1		
\rightarrow	221_1	51_2	1_4	\downarrow	221_1	51_2	1_4
52_0	17	0	0	52_0	4	0	0
204_3	13	4	0	221_3	13	17	17
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_r4bdrd.tdo
```

yields the tactical decomposition

0.1.1				0.1.1			
\rightarrow	221_1	51_2	1_4	\downarrow	221_1	51_2	1_4
52_0	17	0	0	52_0	4	0	0
204_3	13	4	0	204_3	12	16	0
17_5	13	3	1	17_5	1	1	17

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.

References

- [1] Susan Barwick and Gary Ebert. *Unitals in projective planes*. Springer Monographs in Mathematics. Springer, New York, 2008.
- [2] Anton Betten. Twisted tensor product codes. *Des. Codes Cryptogr.*, 47(1-3):191–219, 2008.
- [3] Anton Betten, Michael Braun, Harald Friertinger, Adalbert Kerber, Axel Kohnert, and Alfred Wassermann. *Error-correcting linear codes*, volume 18 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Berlin, 2006. Classification by isometry and applications, With 1 CD-ROM (Windows and Linux).
- [4] Anton Betten, James W. P. Hirschfeld, and Fatma Karaoğlu. Classification of cubic surfaces with twenty-seven lines over the finite field of order thirteen. *Eur. J. Math.*, 4(1):37–50, 2018.
- [5] Anton Betten and Fatma Karaoğlu. Cubic surfaces over small finite fields. *Des. Codes Cryptogr.*, 87(4):931–953, 2019.
- [6] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).
- [7] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.8.7*, 2017.

- [8] D. G. Glynn and J. W. P. Hirschfeld. On the classification of geometric codes by polynomial functions. *Des. Codes Cryptogr.*, 6(3):189–204, 1995.
- [9] Fatma Karaoğlu. The cubic surfaces with twenty-seven lines over finite fields, Ph.D. thesis, University of Sussex, 2018.
- [10] Wen-Ch'ing Winnie Li. Character sums and abelian Ramanujan graphs. *J. Number Theory*, 41(2):199–217, 1992.
- [11] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan graphs. *Combinatorica*, 8(3):261–277, 1988.
- [12] Brendan McKay. Nauty and Traces (Version 2.7r1), Australian National University, 2020.
- [13] Brendan McKay. Possolve, A program to solve positive integer systems, Australian National University, private communication, around 1997.
- [14] POV-RAY Developers. POV-RAY, Persistence of Vision Raytracer Pty. Ltd. (2003-2008) <http://povray.org>, accessed 4/2/2017.
- [15] 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.
- [16] N. J. A. Sloane. *A handbook of integer sequences*. Academic Press, New York-London, 1973.
- [17] Magma system. Magma Calculator <http://magma.maths.usyd.edu.au/calc/>, accessed 11/24/2019.
- [18] Robert Wilson, Peter Walsh, Jonathan Tripp, Ibrahim Suleiman, Richard Parker, Simon Norton, Simon Nickerson, Steve Linton, John Bray, and Rachel Abbott. ATLAS of Finite Group Representations - Version 3, <http://brauer.maths.qmul.ac.uk/Atlas/v3/>, accessed 11/24/2019.