Classification of Combinatorial Objects with Orbiter

Anton Betten (Colorado State University)

Anton.Betten@colostate.edu

Introduction

Orbiter [7] is a computer algebra system devoted to the classification of combinatorial objects. For our purposes, combinatorial object means element of a set with a fixed group action. Two objects are equivalent (or isomorphic) if they belong to the same orbit under the group. The classification problem is the problem of determining the orbits of the group action, for instance by listing one representative from each orbit. This is different from enumerative combinatorics, where the goal is to determine the number of orbits, for instance using enumerative formulae. In most cases, construction of orbit representatives includes determining the stabilizer groups of the orbit representatives. Using the orbit-stabilizer lemma (orbit length equals index of stabilizer in the group that acts), we can then determine the number of objects total. This is often useful as a way to double check the classification, for instance when an enumerative formula is known for the total number of objects.

Related problems that fall within the scope of this research are the following. The isomorphism problem: Given two objects of the same type, determine whether or not they belong to the same orbit. If so, determine a group element that maps one to the other. The automorphism group problem: Given one object, determine the automorphism group. The recognition problem: Given an object, identify the unique element in a previously constructed transversal that is isomorphic to it.

The purpose of this article is to introduce the computer algebra system Orbiter which is devoted to the classification of combinatorial objects. We wish to illustrate how Orbiter can be used to solve these problems for some types of objects. For the sake of space, we restrict our focus to a problem from algebraic geometry, namely the classification of cubic surfaces with 27 lines (cf. [6]). A further application is the classification of smooth quartic curves (cf. [10]). Many further applications can be found in the User's Guide. We will discuss how Orbiter is installed and used. After that, we will show how mathematical objects can be created, including groups, finite fields, and projective spaces. We will then turn to the problem of classifying cubic surfaces with 27 lines over a finite field. Finally, we will comment on some computer graphics capabilities, mathematical databases, and the use of Orbiter in parallel computing.

What is Orbiter?

Orbiter is an open source software package written in C++. It can be used as class library or as a standalone computer algebra system. Orbiter does not have an interactive shell interface. Instead, Orbiter takes commands from the command line. Command sequences can be collected in unix shell scripts or makefiles. Orbiter offers functionality in group theory, in particular the field of finite permutation groups. Based on group theoretic algorithms, combinatorial objects can be classified. This article will illustrate some of the capabilities of Orbiter by showing examples of what it can do. Many further examples can be found in the User's guide, which can be found at [4]. A survey paper about Orbiter is [7].

Many researchers are working on the construction of various types of combinatorial objects. McKay's canonical augmentation procedure from [21] is very popular, as is his software package Nauty [20] (see also [22]). The book [19] is devoted to the classification problem for codes and designs. Many other contributions cannot be listed here for reasons of space.

Orbiter uses partially ordered sets with group action to classify combinatorial objects. For background, see [23]. For the algorithmic aspects, the work of Schmalz [27] and [26] is relevant ("Leiterspiel algorithm"). Before Orbiter, there was DISCRETA [8]. Before DISCRETA, there was Bernd Schmalz's program DCC ("double coset constructor") to compute double cosets in groups.

The goal of this document is to show how Orbiter can be used from the command line. We will not address the use of Orbiter as a C++ class library.

Why do we need it?

Orbiter overlaps with some well-established computer algebra systems, in particular GAP [17] and Magma [12]. The goal is not to replace these systems but rather to offer something that cannot be done or that would be difficult to do in the existing systems. The classification of combinatorial objects is often difficult, and requires tedious computations, for which GAP or Magma may not be the best platform (see [7] for some case studies). Orbiter is trying to fill a gap (no pun intended!) where the existing systems become inefficient. We would also like to point out that there is considerable overlap with the GAP package fining [2]. Again, fining is not the system of choice when it comes to classification, so there is no conflict. In addition, Orbiter offers a way to collect mathematical data by means of a feedback loop. More on that in Section "Mathematical Data" below.

Running Orbiter

There are two ways to run Orbiter: Native and Docker. Native means that Orbiter is compiled from scratch, using the source code from the github repository (cf. [5]). Docker [14] is a system to run preconfigured software in an encapsulated way on various platform, including Windows. We describe using Orbiter through unix *makefiles*, which are run through the tool *make* (cf. [16]). This is a software tool that allows collecting short command snipplets in the form of text files that can easily be handled. However, the conventions in the tool involve some subtleties regarding the use of whitespace, which can cause problems to novice users. We will point out possible pitfalls along the way. Note that it is not necessary to use makefiles. Another possibility would be to use shell scripts. Ultimately, it would be possible to type out all commands into a terminal window. This could be a little tedious though, considering the fact that most Orbiter commands expect lengthy parameters from the command line.

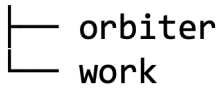
Let us start by discussing how to run Orbiter as a native application. To do so, a unix-like compile environment is required, including a modern C++ compiler and the tools *git* and *make*. Windows users may need to install Cygwin [13]. The following steps are required: Using git, *clone* the repository. Then enter the directory *orbiter* and type

```
make
```

Once compiled, the Orbiter executable is

```
src/apps/orbiter/orbiter.out
```

within the Orbiter directory. We then recommend creating a separate work directory *not within the orbiter directory*. For the following, we assume the following directory tree structure:



In the work directory, create a small makefile like so:

```
OP=../orbiter
ORBITER_PATH=$(OP)/src/apps/orbiter/

test:
  $(ORBITER_PATH)orbiter.out
```

Different directory structures can be accommodated by changing the first line. Next, typing

```
make test
```

within the work directory will invoke Orbiter. Here, *test* is the makefile "target." The makefile target must appear in the makefile. In the example above, the block

```
test:
  $(ORBITER_PATH)orbiter.out
```

is the makefile target "test." It is important that the indentation after the makefile target is done using tab characters (no spaces). There can be multiple targets in one makefile, as long as they are separated by an empty line. for more information about the syntax of makefiles, see [16].

A second way to run Orbiter is through Docker [14]. This does not require a compile environment. However, it comes at a small performance cost when running Orbiter commands that are computationally heavy. Orbiter has already been precompiled (by the Orbiter developer) into an *image*, which is a completely self-sustained copy of a unix-environment that can run by the user under the docker front-end. The image is stored on a docker server under the name `abetten/orbiter`. Docker will receive the name of the image from the command line, pull a local copy of the image, and run the image in an encapsulated environment called a *container*. A copy of the image is stored locally, so that subsequent calls to Orbiter can be satisfied using the local copy, which increases turnaround speed. For instance, the following bare-bones makefile sets up Orbiter for use through Docker:

```
DOCKER_OPTIONS=run -it \
  --volume ${PWD}:/mnt -w \
  /mnt abetten/orbiter
ORBITER_PATH=docker $(DOCKER_OPTIONS)

test:
  $(ORBITER_PATH)orbiter.out
```

In this file, there is a space character in line three after `abetten/orbiter` which is important (and unfortunately cannot be seen). By typing

```
make test
```

into a terminal window, Docker starts up and pulls a copy of Orbiter to the local machine, which is then executed. Orbiter will start up, produce a few messages and then shut down. Interestingly, this will work on a Windows machine also (using *supershell* as terminal). The *make* command is passed through to the container, which contains the unix-like software environment, including make. The associated *makefile* resides on the local machine, as do input and output files.

Orbiter comes with a version numbering system called a build number. The build number should match the commit number on the github tree, shown in Figure 1.

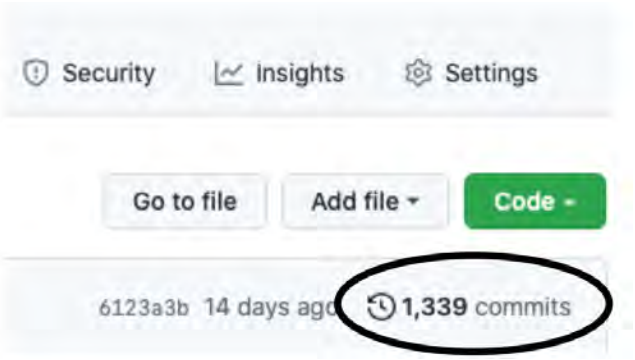


Figure 1: *The commit number*

When Orbiter starts up, the build number is displayed. In order to update to a more recent version of Orbiter, Docker needs to be instructed to discard the local image. To do so, the command

```
docker rmi -f abetten/orbiter
```

is used. After that, any new invocation of Orbiter will cause Docker to pull the latest Orbiter *image* from the Docker repository. It is convenient to combine the Docker and Native compile environment into a single makefile and use the comment symbol (hash #) to switch between the two modes (the line numbers are not part of the file).

```
1  OP=~/orbiter
2  OP2=$(OP)/src/apps/orbiter/
3  DOCKER_OPTS=run·-it·\
4  ··--volume·${PWD}:/mnt·-w·\
5  ··/mnt·abetten/orbiter·
6  #ORBITER_PATH=docker·$(DOCKER_OPTS)
7  ORBITER_PATH=$(OP2)
```

Here, whitespace characters can be seen: (spaces are shown as dots, and tab is a little triangle pointing to the right). Please observe the space at the end of line 5 and that the line(s) after the target(s) must start with a tab symbol (and no spaces). Also, the backslash signs are used to break long lines. Please make sure that there are no spaces after the backslash sign. In order to switch to Docker mode, the hash symbol can be removed in line 6 and instead put at the beginning of line 7. In the following examples, we assume that the 7 lines just shown are present at the beginning of the makefile. For brevity, we will only show the commands and their labels. These snipplets must come after the top part.

Group Theory

Orbiter distinguishes between a group and the group action. One group can have different actions. Every group must have at least one action. Let us look at an example: The following makefile command is used to create the group $\mathrm{P\Gamma L}(3,8)$.

```
PGGL_3_8:
▷  $(ORBITER_PATH)orbiter.out·-v·3·\
▷  ▷  -define·G·-linear_group·\
▷  ▷  -PGGL·3·8·-end
```

The command `PGGL` refers to the group $\mathrm{P\Gamma L}$, and the two parameters afterwards are the dimension and the order of the field of coefficients. Not much happens! This is because though we told Orbiter to create the group, we did not ask that anything be done with it. Orbiter simply creates the group and exits. Note that Orbiter creates an object of type group under the name G. Using this label, it is possible to do something with the group later. The syntax is

```
-with LABEL -do ...-end
```

where LABEL is the label of any Orbiter object (such as G in case of the group $\mathrm{P\Gamma L}(3,8)$). Suppose we want to do something with the group. To this end, we will use a group theoretic activity. One possibility is to print a latex report. The next command does just that:

```
PGGL_3_8_report:
▷  $(ORBITER_PATH)orbiter.out·-v·3·\
▷  ▷  -define·G·-linear_group·\
▷  ▷  ▷  -PGGL·3·8·-end·\
▷  ▷  -with·G·-do·\
▷  ▷  -group_theoretic_activities·\
▷  ▷  ▷  -report·\
▷  ▷  -end
▷  pdflatex·PGGL_3_8_report.tex
▷  open·PGGL_3_8_report.pdf
```

The Orbiter command produces a report in the form of a latex file. The file name is generated automatically based on the group that was created. Next, the pdflatex command is used to translate the latex file. The open command is Macintosh specific and opens the pdf file in this case. The last two commands may have to be replaces by the appropriate local latex specific commands.

Indexing

Orbiter can create many types of permutation groups. Each of these groups can be considered in many actions. Starting from basic groups with their standard actions, new actions can be induced or combined or otherwise

modified to create new actions. Basic groups often come with a natural permutation action. This could be the action on points of an underlying projective space, or an affine space, or an orthogonal space, a hermitian space etc. In order to create these basic actions, the permutation representation needs to be established. To this end, Orbiter uses fixed bijections between the space on which we act and the integer interval $0, \ldots, N-1$ where N is the degree of the action. In GAP [17], this technique is realized by enumerators. In Magma [12], it is called indexing. The bijections selected for indexing are fixed once and for all. For objects in projective spaces over finite fields which are not prime fields, this requires the choice of a polynomial to create the field. Orbiter has built-in polynomials, stored in the form of tables. This guarantees that the results from one computation can be repeated at any time later and do not change. It is however possible to override the polynomial used to create extension fields. This allows compatibility with other computer algebra systems. Indexing exists for many types of objects. Besides points in projective space, Orbiter has indexing for the grassmannian, orthogonal spaces, tensor spaces, direct products, the wedge product, and many other objects.

Let us consider the example from above. In order to create the group $\mathrm{P\Gamma L}(3,8)$, Orbiter first establishes indexing for the 73 points of $\mathrm{PG}(2,8)$. It is possible to produce a latex report which shows the bijection. Because the field $\mathbb{F}_8$ is an extension field, we first need to enumerate the elements in the field $\mathbb{F}_8$. To this end, we pick an irreducible polynomial and represent each field element as polynomial with coefficients over $\mathbb{F}_2$. For instance, Orbiter picks $X^3 + X^2 + 1$ over $\mathbb{F}_2$ to create $\mathbb{F}_8$. Next, we use the binary representation of numbers to map field elements to integers in the interval $0, \ldots, 7$ like so:

0	000	0
1	001	1
2	010	α
3	011	$\alpha + 1$
4	100	α^2
5	101	$\alpha^2 + 1$
6	110	$\alpha^2 + \alpha$
7	111	$\alpha^2 + \alpha + 1$

In order to index the points of projective space, Orbiter uses a variant of the lexicographic ordering. Namely, the points of the standard frame are given preference over all other points. In $\mathrm{PG}(n,q)$, a frame has size $n+2$ and is given the ranks $0, \ldots, n$. The remaining elements are grouped by the number of trailing zeros and then enumerated lexicographically, assuming that the last nonzero element is one.

Algebraic Geometry

A cubic surface is an algebraic variety of degree three in $\mathrm{PG}(3,\mathbb{F})$ (the three-dimensional projective space over the field $\mathbb{F}$). One of the first observations about cubic surfaces is that over an algebraically closed field, a smooth surface has exactly 27 lines on them. It is possible that the 27 lines can be obtained over fields that are not algebraically closed, but this is the exception and not the rule. An example is shown in Figure 2, which shows the Clebsch surface over the real numbers in an affine chart.

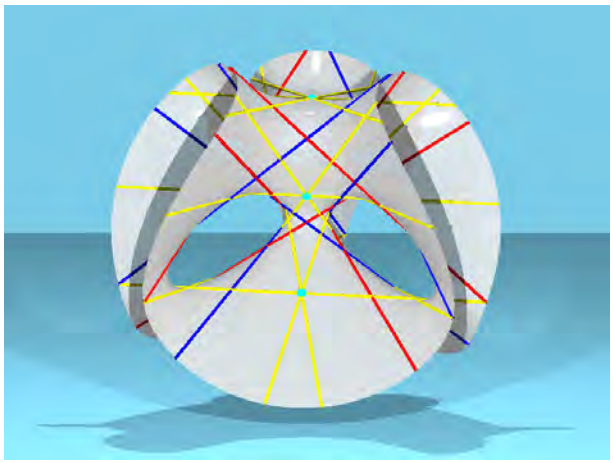


Figure 2: *The Clebsch surface*

In this example, all 27 lines are real, but only 24 are visible. This is because three lines lie in the plane at infinity. Another example with all 27 lines real is the Eckardt surface. The coloring of the lines is done according to a Schläfli double six. The six red lines and the six blue lines represent the double six. The 15 yellow lines (3 are at infinity) represent the 15 "diagonal" lines. It is possible that three of the lines are concurrent in a single point. Such a point is called an Eckardt point. The Clebsch surface is characterized by the fact that it has exactly 10 Eckardt points (shown in turquoise in the picture, but not all are visible in the affine chart). The number of Eckardt points is between 0 and 45, but not every number in the interval is possible. Besides the Clebsch surface, there are many other surfaces. Two surfaces are equivalent (or isomorphic) if there is a collineation of the projective space which maps one to the other. It is of interest to determine how many nonisomorphic cubic surfaces with 27 lines exist over a given field, in particular over any finite field. Let us consider an example. In order to classify the cubic surfaces with 27 lines over $\mathbb{F}_{13}$, we use Orbiter with the following makefile command:

```
surface_classify_q13:
▷   $(ORBITER_PATH)orbiter.out -v 5 \
▷   ▷   -define F -finite_field \
▷   ▷   ▷   -q 13 -end \
▷   ▷   -define P -projective_space \
▷   ▷   ▷   3 F -end \
▷   ▷   -with P -do \
▷   ▷   -projective_space_activity \
▷   ▷   ▷   -classify_surfaces_with \
double_sixes Surf27 -W -end \
▷   ▷   -end \
▷   ▷   -with Surf27 -do \
```

```
▷  ▷  -classification_of_cubic_\
surfaces_with_double_sixes_\
activity·\
▷  ▷  ▷  -report·-end·\
▷  ▷  -end·\
▷  ▷  -print_symbols
▷  pdflatex·Surfaces_q13.tex
▷  open·Surfaces_q13.pdf
```

The computations show that there are exactly 4 isomorphism types of cubic surfaces with 27 lines over $\mathbb{F}_{13}$. They are the Eckardt surface with 6 Eckardt points, the Fermat surface with 18 Eckardt points, and two further surfaces with 9 and 4 Eckardt points, respectively. Detailed information about each surface can be found in the latex report (which has 265 pages). Let us take a quick look at the command. The command sequence defines three objects, called F, P and $Surf27$. The objects are maintained in a symbol table. The line

```
-define F -finite_field -q 13 -end
```

creates the field $\mathbb{F}_{13}$ as object F. The line

```
-define P -projective_space 3 F -end
```

creates the three-dimensional projective space over the field F and defines the object P. The sequence

```
-with P -do \
  -projective_space_activity \
    -classify_surfaces_with_\
double_sixes Surf27 -W -end \
-end
```

classifies all cubic surfaces with 27 lines in the projective space P and defines the object $Surf27$. The sequence

```
-with Surf27 -do \
  -classification_of_cubic_\
surfaces_with_double_sixes_\
activity \
  -report -end \
-end
```

creates the latex report from the classification. The algorithm from [11] is used to classify cubic surfaces with 27 lines. This algorithm relies on the well-known relation between cubic surfaces with 27 lines and Schläfli double sixes (cf. [25]). The double sixes in turn are classified using smaller structures called five-plus-ones. These are five pairwise skew lines in $\mathrm{PG}(3,\mathbb{F})$ with a common transversal and with the property that none of the five lies in the regulus determined by three of the other four.

Consider the four surfaces over $\mathbb{F}_{13}$. The automorphism group orders and the number of Eckardt points are summarized below, together with the family name:

Surface	#Eckardt	Stab. Order	Name
0	4	12	
1	6	24	Eckardt [15]
2	9	108	
3	18	648	Fermat

The group $\mathrm{P\Gamma L}(4,13)$ has order 50858076935877120. The orbit stabilizer lemma applied to the classification gives that the number of cubic surfaces with 27 lines over $\mathbb{F}_{13}$ is

$$50858076935877120\Big(\frac{1}{12}+\frac{1}{24}+\frac{1}{108}+\frac{1}{648}\Big)$$

$= 6906652423390720$. On the other hand, according to [18], the number of cubic surfaces with 27 lines over $\mathbb{F}_q$ is

$$\frac{q^6(q^2-1)(q^3-1)(q^4-1)(q-2)(q-3)(q-5)^2}{51840}.$$

For $q=13$, this confirms the result from the classification.

Computer Graphics

Through its interface to the ray-tracing software Povray [24], Orbiter can create three-dimensional graphics and animations. For instance, the code below creates the Kummer surface:

```
Kummer_surface:
▷  $(ORBITER_PATH)orbiter.out·-v·2·\
▷  ▷  -povray·\
▷  ▷  -round·0·-nb_frames_default·30·\
▷  ▷  -output_mask·Kummer_%d_%03d.pov·\
▷  ▷  -video_options·-W·1024·-H·768·\
▷  ▷  -global_picture_scale·0.9·\
▷  ▷  -default_angle·75·\
▷  ▷  -clipping_radius·2.4·\
▷  ▷  -camera·0·"1,1,1"·"-3,1,3"·\
▷  ▷  ▷  "0.12,0.12,0.12"··\
▷  ▷  -end·\
▷  ▷  -scene_objects·\
▷  ▷  ▷  -quartic_lex_35·"-2,0,0,0,2,\
0,0,2,0,2,0,0,0,0,0,0,0,0,0,0,-2,\
0,0,2,0,2,0,0,0,0,-2,0,2,0,-2"·\
▷  ▷  ▷  -group_of_things·"0"·\
▷  ▷  ▷  -quartics·0·"texture{·\
pigment{·White*0.5·transmit·0.5·}·\
finish·{ambient·0.4·diffuse·0.5·\
roughness·0.001·reflection·0.1·\
specular·.8}·}"·\
▷  ▷  -scene_objects_end·\
▷  ▷  -povray_end
```

The code prepares 30 different frames, showing the surface as it rotates along a vertical axis. The frames can be assembled to an animation using tools like ffmpeg [3]. The surface is defined using the coefficient vector. The monomials of degree 4 are arranged in lexicographic order. Exactly 35 coefficients are given after the `-quartic_lex_35` option. The remaining options are used to specify things like camera viewpoint

and surface color. An image created from this command is shown in Figure 3.

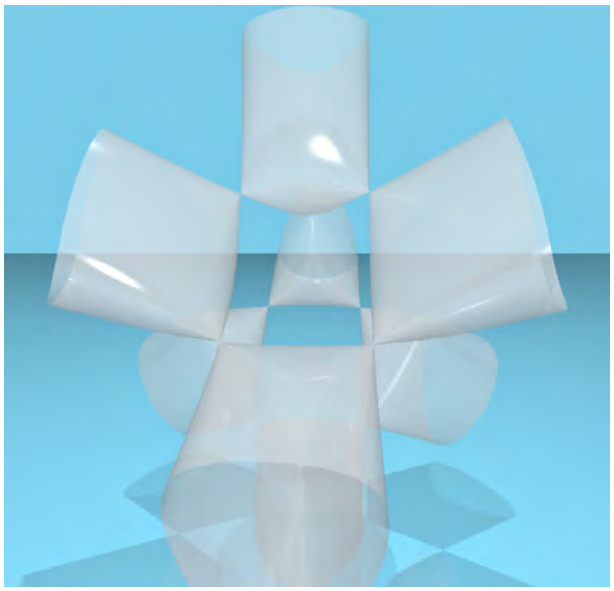


Figure 3: *The Kummer surface*

Mathematical Data

The computations necessary to perform a classification involve potentially large numbers of CPU-cycles, For this reason, the mathematical data produced by Orbiter should be conserved for later use. One way to do so is by means of the feedback loop. The output of a classification is written to a C++ source file, which can then be compiled into a later version of Orbiter. This way, the classification becomes available instantaneous to later users of Orbiter. It is also possible to create web-databases of mathematical data. The mathdata project [9] is underway, which will allow access to the data by clicking through tables. At present, Orbiter has databases for cubic surfaces, quartic curves, BLT-sets, spreads and translation planes, packings, dual hyperovals, arcs, linear codes, irreducible polynomials over finite fields and others. For a more in-depth discussion of the feedback loop, see [7].

High Performance Computing

Orbiter is well-suited for high performance computing using compute clusters. The Orbiter workflow is file based: There is a makefile, and there are input and output files to every job. This kind of workflow fits well with parallel processing on compute clusters, where little command scripts are used to start up jobs. These command scripts have information about the desired hardware configuration and about the job demands. The orbiter makefile commands go into the command scripts. We acknowledge use of the NSF funded machine Summit which is run jointly by Colorado State University and CU Boulder [1].

References

[1] Jonathon Anderson, Patric J. Burns, Daniel Milroy, Peter Ruprecht, Thomas Hauser, and Howard Jay Siegel. Deploying RMACCS Summit: An HPC Resource for the Rocky Mountain Region. In *Proceedings of PEARC* 17*, New Orleans, LA, USA, July 09-13, 2017,* 7 pages.

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

[3] Fabrice Bellard. FFmpeg `https://www.ffmpeg.org`.

[4] A. Betten. Orbiter: `https://www.math.colostate.edu/~betten/orbiter/`.

[5] Anton Betten. Orbiter – A program to classify discrete objects, 2019, `https://github.com/abetten/orbiter`.

[6] Anton Betten. Classifying cubic surfaces over finite fields using orbiter. In *ICMS 2018—Proceedings of the International Congress on Mathematical Software; James H. Davenport, Manuel Kauers, George Labahn, Josef Urban (ed.)*, pages 55–61. Springer, 2018.

[7] Anton Betten. The orbiter ecosystem for combinatorial objects. In *ISSAC 2020—Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation*, pages 30–37. ACM, New York, 2020.

[8] Anton Betten, Evi Haberberger, Reinhard Laue, and Alfred Wassermann. *DISCRETA – A program system for the construction t-designs*. Lehrstuhl II für Mathematik, Universität Bayreuth, 1999. `http://www.mathe2.uni-bayreuth.de/~discreta`.

[9] Anton Betten, Nathan Kaplan, Fatma Karaoglu, and Oznur Oztunc. MATHDATA `https://www.math.colostate.edu/~betten/mathdata/`.

[10] Anton Betten and Fatma Karaoglu. Isomorphism Testing of Algebraic Varieties using Canonical Forms, In: Computer Algebra in Scientific Computing, CASC 2021, September 13-17, 2021, Sochi, Russia, Sirius Mathematics Center, `https://siriusmathcenter.ru/pr_img/1918100371/20210914/13241784/Program_010w.pdf`, pp 14-27.

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

[12] 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).

[13] Cygwin. https://www.cygwin.com.

[14] Docker. https://www.docker.com.

[15] F.E. Eckardt. Ueber diejenigen Flächen dritten Grades, auf denen sich drei gerade Linien in einem Punkte schneiden, Math. Ann. 10 (1876), 227-272.

[16] Free Software Foundation. GNU make https://www.gnu.org/software/make/manual/make.html.

[17] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.8.7*, 2017.

[18] Fatma Karaoglu and Anton Betten. The number of cubic surfaces with 27 lines over a finite field. To appear in Journal of Algebraic Combinatorics.

[19] P. Kaski and P. Östergård. *Classification algorithms for codes and designs*, volume 15 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Berlin, 2006.

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

[21] Brendan D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26(2):306–324, 1998.

[22] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symbolic Comput.*, 60:94–112, 2014.

[23] Wilhelm Plesken. Counting with groups and rings. *J. Reine Angew. Math.*, 334:40–68, 1982.

[24] POV-RAY Developers. POV-RAY, Persistence of Vision Raytracer Pty. Ltd. http://povray.org, accessed 1/23/2021.

[25] 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.

[26] B. Schmalz. t-Designs zu vorgegebener Automorphismengruppe. *Bayreuth. Math. Schr.*, 41:1–164, 1992. Dissertation, Universität Bayreuth, Bayreuth, 1992.

[27] Bernd Schmalz. Verwendung von Untergruppenleitern zur Bestimmung von Doppelnebenklassen. *Bayreuth. Math. Schr.*, (31):109–143, 1990.