Two matrix group algorithms with applications to computing the automorphism group of a finite p-group

by

Ruth Schwingel

Thesis submitted for the degree
of
Doctor of Philosophy

School of Mathematical Sciences
Queen Mary and Westfield College
University of London

Supported by a studentship from CAPES

January 2000

Abstract

A theoretical description of an algorithm to determine the automorphism group of a finite p-group P was first given by Newman. Implementations of this algorithm with substantial improvements by O'Brien are available in GAP and MAGMA.

The original algorithm, starting with the Frattini quotient $V = P/\Phi(P)$, computes recursively the automorphism group G of the quotient Q of P by successive terms of the lower p-central series of P. Thus the first step returns $G = GL(V)$.

The heart of the algorithm is the computation of the subgroup of G that normalises a certain subspace of the p-multiplicator M of Q. A refinement in the algorithm replaces G by a subgroup H that normalises certain subspaces of V corresponding to heuristically determined characteristic subgroups of P. In this thesis we describe and give the GAP3 code for two substantial improvements to the algorithm.

The first improvement is an algorithm that returns a generating set for the stabiliser in $GL(V)$ of any given sequence of subspaces of a finite dimensional vector space V over any finite field. This is an algorithm of independent interest, as the intersection problem for subgroups of $GL(d, p^n)$ is both important and hard. In the algorithm for computing the automorphism group of the p-group P this intersection algorithm is used to compute the precise subgroup K of $GL(V)$ that stabilises the given sequence of subspaces rather than the over-group H of K currently computed.

The theoretical basis for the intersection algorithm is a new Galois correspondence between lattices of subspaces of V and subgroups of $GL(V)$. The basic computational tool is the 'meataxe' algorithm.

As a second contribution, we give an efficient algorithm to compute a canonical form for a subspace U of M under the action of a p-subgroup G of $GL(M)$, and also to compute generators for the subgroup of G that normalises U. Here 'efficient' means 'polynomial in the size of the input', and M can be any finite dimensional vector space over $GF(p)$. This is important as the kernel of the action of G on V is a p-group; and G itself is often a p-group.

Contents

Acknowledgements

Many thanks are due to my supervisor, Prof. Charles Leedham-Green, for his advice, suggestions, encouragement and great pacience while this thesis was being undertaken.

I would also like to thank Dr. Leonard Soicher for his support and advice, and all the members of QMW maths department, in particular the students in office 201.

I must also thank Dr. Eamon O'Brien for his advice and help in testing the implementation of one of my algorithms.

Many thanks also to my family and friends for their support and prayers.

Finally I thank CAPES who made the last four years financially possible.

Chapter 1

Computing the automorphism group of a finite p-group

1.1 Introduction

In [12] E. A. O'Brien describes an algorithm to compute the automorphism group of a finite p-group. The algorithm constructs a standard presentation for the p-group using the *standard presentation algorithm* [11] and simultaneously constructs a generating set for its automorphism group using the *p-group generation algorithm* [10]. The first theoretical description of the p-group generation algorithm was given by M. F. Newman [9] in 1977 and a full theoretical description and implementation was given by O'Brien in 1990.

In this chapter we give in section 1.2 a brief description of O'Brien's algorithm to compute the automorphism group of a finite p-group. In sections 1.3 and 1.4 we describe some further improvements implemented by O'Brien and B. Eick in 1996.

1.2 The basic algorithm

The *lower exponent-p central series* of a group G is the sequence of subgroups

$$G = G_0 \geqslant \cdots \geqslant G_i \geqslant G_{i+1} \geqslant \cdots$$

where $G_{i+1} = [G_i, G]G_i^p$ for $i > 1$. If $G_c = \langle 1 \rangle$ and c is the smallest such integer, then we say that G has *exponent-p class c* or, in this thesis simply, *class c*.

Let P be a d-generator p-group of class c. Then $P_2 = \Phi(P)$[6, III 3.14] where $\Phi(P)$ is the Frattini subgroup of P. Let F be a free group of rank d generated by the set $X = \{a_1, \ldots, a_d\}$ and let R be the kernel of a homomorphism from F onto P, *i.e.*, $F/R \cong P$. Defining R^* to be $[R, F]R^p$ we now define $P^* = F/R^*$ to be the *p-covering group* of P, and the extension is independent of the surjection $F/R \longrightarrow P$ [10, Lemma 2.3]. Furthermore we define R/R^* to be the *p-multiplicator* and P_c^* the *nucleus* of P.

The group H is an *immediate descendant* of P if it is a d-generator group of class $c+1$ and $H/H_c \cong P$. Every immediate descendant of P is isomorphic to a quotient of P^* [10, Thm 2.2]. An *allowable subgroup* is a subgroup of the p-multiplicator which is the kernel of a homomorphism from P^* onto an immediate descendant of P.

Given $\alpha \in \mathrm{Aut}(F/R)$ every extension of α to $\alpha^* \in \mathrm{Aut}(F/R^*)$ can be constructed as follows. For each $i \in \{1, \ldots, d\}$ choose a representative $u_i \in F$ of the coset $a_i R\alpha$ and define $a_i R^* \alpha^* = u_i R^*$. For a proof that α^* is an automorphism of F/R^* see [10, Thm 2.5]. The automorphism α^* is called an *extended automorphism*.

The basic algorithm described by O'Brien in [12] to compute the automorphism group of P starts with a presentation for the rank d elementary abelian p-group $P/\Phi(P)$ and its automorphism group $GL(d,p)$ and iteratively constructs the immediate descendant P/P_{i+1} of P/P_i and a generating set for its automorphism group, eventually reaching $P = P/P_{c+1}$ and constructing a generating set for $\mathrm{Aut}(P)$. Given a presentation for P/P_i it determines the p-covering group $(P/P_i)^*$ and the p-multiplicator $M_p(P/P_i)$ of P/P_i. The immediate descendant P/P_{i+1} is the quotient of the p-covering group by an allowable subgroup $M < M_p(P/P_i)$. Now each generator α of $\mathrm{Aut}(P/P_i)$ is extended to an automorphism α^* of $(P/P_i)^*$. Each extended automorphism α^* induces a permutation of the allowable subgroups that depends only on α [10, Thm 2.7]. Two allowable subgroups M_1/R^* and M_2/R^* are said to be *equivalent* if and only if their quotients F/M_1 and F/M_2 are isomorphic. The orbits of the allowable subgroups under the action of the permutations induced by the α^* are exactly the equivalence classes of the allowable subgroups [10, Thm 2.8].

The *stabiliser* S_M of the allowable subgroup M is defined by

$$S_M = \langle\, \xi \in \mathrm{Aut}(P/P_i) \mid M\xi^* = M \,\rangle.$$

For $\xi \in S_M$ let ξ^* be an arbitrary extension to $\mathrm{Aut}((P/P_i)^*)$. Then ξ^* fixes M and therefore we can calculate its restriction to P/P_{i+1}. Now the automorphism group of P/P_{i+1} can be determined according to the following theorem.

Theorem 1.1. *Let S consist of the restriction to P/P_{i+1} of one ξ^* for each automorphism ξ in S_M and let V be the group of all automorphisms of P/P_{i+1} whose restriction to P/P_i is the identity. Then $Aut(P/P_{i+1}) = SV$.*

Proof. See [10, Thm 2.10]. ■

Assuming the orders of P/P_i and P/P_{i+1} are p^n and p^{n+s}, respectively, the group V is generated by the set $\{\theta_{jk}\}$ where θ_{jk} is defined by

$$\begin{aligned} \theta_{jk} \quad : \quad & a_j \longmapsto a_j a_{n+k} && \text{for } \ j \in \{1, \ldots, d\}, \ k \in \{1, \ldots, s\} \\ & a_r \longmapsto a_r && \text{for } \ r \in \{1, \ldots, d\} \setminus \{j\} \end{aligned}$$

where $a_{n+1}, \ldots, a_{n+s}$ are elements of a basis of the allowable subgroup M. The elements of V are called *central automorphisms* of P/P_{i+1}.

The method used by O'Brien to make the orbit-stabiliser calculation more efficient consists of picking a characteristic subgroup C of the p-covering group in the p-multiplicator and working within the intersection of the allowable subgroup and the nucleus with C. This splits the given orbit-stabiliser calculation into a number of easier orbit-stabiliser calculations. For more details see [10, §4].

1.3 First improvements

As pointed out before, the iteration of the algorithm starts with a presentation for the rank d elementary abelian p-group $P/\Phi(P)$ and its automorphism group $GL(d,p)$. In practice the order of $GL(d,p)$ is far too big to permit an efficient calculation. The following theorem of Bryant and Kovács [2, §1] shows that the restriction of $\mathrm{Aut}(P)$ to $P/\Phi(P)$ might be any subgroup of $GL(d,p)$.

Theorem 1.2. *For each linear group H of finite dimension d, with $d \geqslant 2$, over the field of order p, there exists a finite p-group P such that the restriction of $Aut(P)$ to $P/\Phi(P)$ is isomorphic, as linear group, to H.*

Proof. See [2, Theorem 1]. ■

The question then is, given a d-generator p-group P, how to find a proper subgroup of $H = GL(d, p)$ that can easily be proved to contain the image of $\mathrm{Aut}(P)$, if such exists. Also, given a number of such subgroups, how to find a generating set for their intersection.

From now on the expression *initialisation of the automorphism calculation* will always mean finding a suitable subgroup of $GL(V)$ to start the automorphism calculation.

Two methods were developed to solve this problem and in 1996 E. O'Brien and B. Eick implemented them in MAGMA [1] and GAP [14] respectively.

1.3.1 Characteristic subgroups

The characteristic subgroup method was developed by C. Leedham-Green, A. Niemeyer, E. O'Brien and M. Smith. It is an important improvement on the original algorithm, but as we will see in this section, it can still be improved.

The rank d elementary abelian p-group $P/\Phi(P)$ can be regarded as a d-dimensional vector space $V = F^d$ where F is the finite field of p elements. Hence subgroups of P containing $\Phi(P)$ can be regarded as subspaces of V. Let $C_1, \ldots, C_t$ be characteristic subgroups of P containing $\Phi(P)$. Then for each $\alpha \in \mathrm{Aut}(P)$ and $i = 1, \ldots, t$ we have $C_i^\alpha = C_i$. Now let $U_1, \ldots, U_t$ be the subspaces of V corresponding to $C_1, \ldots, C_t$. Then the restriction of α to V, *i.e.* to $P/\Phi(P)$, is a matrix $g \in GL(V)$ such that $U_i g = U_i$ for $i = 1, \ldots, t$.

Let G be the subgroup of $GL(V)$ stabilising the subspaces $U_1, \ldots, U_t$. Then G clearly contains the image $\mathrm{Aut}(P)$, but it might still be much bigger.

Clearly G depends on the choice of the characteristic subgroups and there is no standard "ideal choice".

The characteristic subgroups calculated in the GAP implementation of the characteristic subgroup method are the 2-step centralisers $\mathcal{C}_H(P_{i-2}/P_i)$ and omega subgroups $\Omega_j(H) = \langle h \in H | h^{p^j} = 1\rangle$ of factors $H = P/P_i$ of the lower exponent-p central series of P, the centre of P and the users can also include their own characteristic subgroups.

Once the subspaces U_i corresponding to the characteristic subgroups C_i, for $1 \leqslant i \leqslant t$, are determined, a chain of subspaces of V

$$V = W_m > W_{m-1} > \cdots > W_0 = \langle 0 \rangle$$

is set up by taking certain sums and intersections of the U_i. The stabiliser of this chain in $GL(V)$ is then determined and used in the initialisation of the automorphism calculation. This stabiliser contains the stabiliser of the subspaces $U_1, \ldots, U_t$ and is determined as follows.

The factors W_i/W_{i-1} for $i = 1, \ldots, m$, determine a block structure on $d \times d$ matrices such that with respect to an appropriate basis the elements of the group $G \leqslant GL(V)$ stabilising all W_i's have the form

$$\begin{pmatrix} \boxed{*} & & & 0 \\ & \boxed{*} & & \\ & & \ddots & \\ * & & & \boxed{*} \end{pmatrix}$$

where the i-th block contains the full general linear group $GL(W_i/W_{i-1})$. The group G obtained in this way is usually smaller than $GL(V)$ but may properly contain the subgroup of $GL(V)$ corresponding to the induced automorphism group.

The reasons why the method described above might not return the smallest subgroup of $GL(V)$ stabilising all the subspaces in the lattice L generated by the U_i's are:

- The lattice L generated by $U_1, \ldots, U_t$ is not in general upper/lower semi-modular. The chain of subspaces $\{W_j\}$ should be replaced by a maximal chain in a semi-modular lattice containing the U_i as described in Chapter 2.
- Let H be the intersection of the normalisers of the U_i and let $\{W_j\}$ be a maximal chain in the above lattice. Then some of the H-modules W_j/W_{j+1} may be isomorphic.
- H may act on some factor W_j/W_{j+1} as the general linear group (in a smaller dimension) over a larger field.
- There may be relations between entries below the blocks.

These problems will be addressed in Chapter 2, where we construct a generating set for $\bigcap_{i=1}^{t} \mathcal{N}_{GL(V)}(U_i)$.

1.3.2 Minimal overgroups

The minimal overgroup method was developed by E. O'Brien. It considers the minimal overgroups of $\Phi(P)$; these correspond to the subspaces of dimension 1 of the d-dimensional vector space $P/\Phi(P)$ over F. By the use of finger-print functions, invariants of these subspaces are determined which have to be respected by the automorphism group. These invariants determine a partition of the subspaces, and then the stabiliser of this partition in $GL(V)$ is determined.

One alternative to get a smaller stabiliser is to use maximal subspaces of $P/\Phi(P)$. Stabiliser calculations done by using maximal subspaces suggest this method is often much more time consuming than using the 1-dimensional subspaces.

1.4 Orbit and stabiliser calculations

The orbit and stabiliser calculations in Eick's and O'Brien's implementation of the automorphisms of a p-group algorithm are done as referred to in [10, 3.5]. It uses the algorithms described in [8, §3] and [3, Chapter 7] for the soluble and insoluble cases, respectively.

We developed an algorithm to determine a canonical form of a subspace of a vector space W under the action of a p-subgroup of $GL(W)$, together with a set of generators for the stabiliser of the canonical form. This algorithm is important in the context of the automorphisms of a p-group calculations because, using the same notation as in 1.3.1, the kernel of the action of G on V is a p-subgroup, and G itself is often a p-group. The algorithm is described in Chapter 3 and the commented code is printed out in Appendix B.

Chapter 2

The intersection of subspace normalisers in GL(V)

2.1 Introduction

The algorithm to determine the normaliser for a sequence of subspaces of a vector space was motivated by the automorphisms of a p-group problem. But as an independent algorithm it has a much broader range of applications. For instance, the problem of finding the intersection of a family of permutation groups is hard, and for matrix groups seems much worse. Our algorithm efficiently solves an important special case.

Given a set $U_1, \ldots, U_t$ of subspaces of the d-dimensional vector space $V = F^d$ over a field F with q elements where $q = p^m$ for some prime p, we find a generating set for $G = \bigcap_{i=1}^{t} \mathcal{N}_{GL(V)}(U_i)$.

In section 2.2 we prove the Galois correspondence which is the basis for the intersection of subspace normalisers algorithm. In sections 2.3 to 2.7 we describe the basic steps of the algorithm. Implementation issues are described in section 2.8 and in section 2.9 we provide some information on the performance of the implementation.

2.2 A Galois correspondence between algebras and lattices

With $G = \bigcap_{i=1}^{t} \mathcal{N}_{GL(V)}(U_i)$, clearly every subspace of V in the lattice L generated by the U_i is G-invariant, but the lattice of G-invariant subspaces of V is in general bigger than L, and it is this bigger lattice that we need to consider.

Let L be a lattice generated by subspaces $U_1, \ldots, U_t$ of $V = F^d$ and let A be an algebra of matrices in $M_d(F)$. We define $A(L)$ to be the algebra of matrices in $M_d(F)$ normalising every subspace in L and $L(A)$ to be the lattice of subspaces of V which are normalised by all elements of A. Hence $L(\,)$ is a map from the set $\mathcal{A}$ of all subalgebras of $M_d(F)$ into the set $\mathcal{L}$ of all sublattices of the full lattice of subspaces of V and $A(\,)$ is a map from $\mathcal{L}$ into $\mathcal{A}$. These algebras and lattices satisfy the following Galois correspondence.

Proposition 2.1. *Let A_1, A_2 be algebras of matrices in $M_d(F)$ and let L_1, L_2 be lattices of subspaces of $V = F^d$. Then*

(a) $A_1 \leqslant A_2 \implies L(A_1) \geqslant L(A_2)$

(b) $L_1 \leqslant L_2 \implies A(L_1) \geqslant A(L_2)$.

Proof. By definition we have for $i = 1, 2$

$$\begin{aligned} A(L_i) &= \{a \in M_d(F);\ Wa = W \ \text{ for all } \ W \in L_i\} \\ L(A_i) &= \{W \leqslant V;\ Wa = W \ \text{ for all } \ a \in A_i\}. \end{aligned}$$

(a) Suppose $W \in L(A_2)$. Then $Wb = W$ for all $b \in A_2$. From $A_1 \leqslant A_2$ then follows $Wb = W$ for all $a \in A_1$, hence $W \in L(A_1)$.

(b) Suppose $a \in A(L_2)$. Then $Wa = W$ for all $W \in L_2$. From $L_1 \leqslant L_2$ then follows $Ua = U$ for all $U \in L_1$, hence $a \in A(L_1)$. ■

Proposition 2.2. *Let A be an algebra of matrices in $M_d(F)$ and let L be a lattice of subspaces of $V = F^d$. Then*

(a) $A(L(A)) \geqslant A$

(b) $L(A(L)) \geqslant L$

Proof. (a) By definition we have

$$\begin{aligned} A(L(A)) &= \{\, a \in M_d(F)\,;\, Ua = U \text{ for all } U \in L(A)\,\} \\ L(A) &= \{\, W \leqslant V\,;\, Wa = W \text{ for all } a \in L\,\}. \end{aligned}$$

Suppose $b \in A$. Then $Wb = W$ for all $W \in L(A)$, hence $b \in A(L(A))$.

(b) By definition we have

$$\begin{aligned} L(A(L)) &= \{\, W \leqslant V\,;\, Wa' = W \text{ for all } a' \in A(L)\,\} \\ A(L) &= \{\, a \in M_d(F)\,;\, Ua = U \text{ for all } U \in L\,\} \end{aligned}$$

Suppose $W \in L$. Then for all $a \in A(L)$ we have $Wa = W$, hence $W \in L(A(L))$. ■

Corollary 2.1. $L(A(L(A))) = L(A)$ *and* $A(L(A(L))) = A(L)$. ■

We write $\overline{A} = A(L(A))$ and $\overline{L} = L(A(L))$ and call them the *closures* of A and L, respectively.

Corollary 2.2. $L(A)$ *and* $A(L)$ *are closed.* ■

Corollary 2.3. *Let $\mathcal{L}$ be the full lattice of subspaces of $V = F^d$. Then $L(\,)$ and $A(\,)$ are order reversing bijections between the set of all closed sublattices of $\mathcal{L}$ and the set of all closed subalgebras of $M_d(F)$.* ■

Once we have determined an algebra A normalising every subspace in L, Corollary 2.2 shows that A also normalises $\overline{L}$. Hence a composition series for V as an A-module is a chain of maximal length in $\overline{L}$. So the algorithm to determine the normaliser in $GL(V)$ of $\overline{L}$ has the following basic steps.

Step 1 Determine the algebra A normalising every U_i for $i = 1, \ldots, t$.

Step 2 Determine a composition series $V = V_1 > \cdots > V_n > V_{n+1} = \langle 0 \rangle$ of V as A-module.

Step 3 Let A_B be the image of A in $\prod_{i=1}^{n} \mathrm{End}(V_i/V_{i+1})$. We determine a generating set B for the group G_B of units of A_B. Complications arise from two sources:

(a) distinct composition factors may be isomorphic as A-modules;

(b) A need not act absolutely irreducibly on every composition factor.

Step 4 There is an exact sequence $1 \longrightarrow G_P \longrightarrow G \longrightarrow G_B \longrightarrow 1$ where $G = \bigcap_{i=1}^{t} \mathcal{N}_{GL(V)}(U_i)$, and G_P is the kernel of the action of G on $\sum_{i=1}^{t} V_i/V_{i+1}$. For each generator b of G_B, find an element g_b of G that maps to b.

Step 5 Find a generating set S for G_P (as normal subgroup of G).

Step 6 Then $S \cup \{g_b; b \in B\}$ is a generating set for G.

2.3 Determining the normalising algebra

The algebra A can be determined by solving a system of linear equations in d^2 indeterminates $x_{11}, x_{12}, \ldots, x_{dd}$ obtained from the relations $U_iX \leqslant U_i$ for $i = 1, \ldots, t$, where $X = (x_{jk})_{d\times d}$ is the indeterminate matrix. We take a basis for U_i and extend it to a basis for V. Working with respect to this basis, the condition $uX \in U_i$ for any $u \in U_i$ is a linear equation in the coefficients of X. Since the entries of X with respect to the original basis are linear combinations of the entries of X with respect to the new basis, the above linear equations give rise to linear equations in the x_{jk}. The equations are homogeneous since the 0 matrix satisfies the conditions. Taking the equations arising in this way for every u in a basis for U_i we obtain the required system. Each basis element (vector of length d^2) of the solution set of the system determines a $d \times d$ matrix as basis element for A.

2.4 The composition series

An A-module V is defined by the action of the algebra A, generated by a set of matrices, which in our case is the basis determined in section 2.3, on the vector space $V = F^d$.

As an A-module V has a composition series

$$V = V_1 > \cdots > V_n > V_{n+1} = \langle 0 \rangle.$$

If $d_i = \dim(V_i/V_{i+1})$ then with an appropriate change of basis each algebra element has the block form described in section 1.3.1 where the i-th block is a $d_i \times d_i$ matrix and entries corresponding to isomorphic composition factors are equal. The change of basis matrix to convert the matrices into block form is obtained from the composition series as follows. If $v_{i_1} + V_{i+1}, \ldots, v_{i_{k_i}} + V_{i+1}$ is the basis for V_i/V_{i+1} returned by the composition series calculation for $i = 1, \ldots, n$, then we obtain the inverse of the change of basis matrix by concatenating the lists of vectors $[v_{i_1}, \ldots, v_{i_{k_i}}]$ for $i = 1, \ldots, n$, such that each v_{i_j} becomes a row of the matrix.

In our implementation of the intersection of subspace normalisers algorithm a composition series of V is obtained by the algorithm of Holt and Rees [5] to test modules for irreducibility. This algorithm is a generalisation of the 'Meataxe' algorithm of Parker [13] which uses Norton's irreducibility test that goes as follows. Let the algebra A be generated by matrices $a_1, \ldots, a_r$ and let V^{tr} be the module defined by the transposes $a_1^{tr}, \ldots, a_r^{tr}$. Choose an element $a \in A$, determine its nullspace N and the nullspace N^{tr} of its transpose a^{tr}. Then V is proved to be irreducible if all the following occur

(a) N is non-zero;

(b) every non-zero vector $v \in N$ generates the whole of V as A-module;

(c) at least one non-zero vector $w \in N^{tr}$ generates the whole of V^{tr} as A-module.

If (a) is satisfied but (b) or (c) fails, then this gives an A-invariant subspace of V, either directly in (b) or indirectly in (c).

As part of our composition series calculation we test the composition factors for isomorphisms. The isomorphism information will be used in the next step of the algorithm.

2.5 The action of A on the composition factors

Let $V = V_1 > \cdots > V_n > V_{n+1} = \langle 0 \rangle$ be the composition series of V as an A-module determined by the algorithm in step 3. The algebra A acts irreducibly on the factors V_i/V_{i+1} of dimension d_i for $i = 1, \ldots, n$, and by Wedderburn's Theorem [4, 26.4] this action is isomorphic to $M_{d_i/e_i}(K_i)$ where $K_i = \mathrm{Hom}(V_i/V_{i+1}, V_i/V_{i+1}) \supseteq F$. Since F and K_i are finite we have $K_i \cong GF(q^{e_i})$ for some $e_i \geqslant 1$. For more details see [5, 2.3]. If the action is absolutely irreducible then $e_i = 1$, *i. e.*, $K_i = F$ [4, 29.13]; hence the action is isomorphic to $M_{d_i}(F)$.

V_i/V_{i+1} is an irreducible A_B-module for all i. So A_B is an Artin ring acting faithfully on the semi-simple module $\bigoplus V_i/V_{i+1}$. Hence A_B is semi-simple, and acts on V_i/V_{i+1} as $M_{d_i/e_i}(K_i)$ where $K_i = GF(q^{e_i})$ for some $e_i \geqslant 1$, for all i. It follows that A_B is isomorphic to $\prod_{j \in J} M_{d_j/e_j}(K_j)$ for some subset J of $\{1, \ldots, n\}$, where for some map θ from $\{1, \ldots, n\}$ onto J such that $e_{i\theta} = e_j$ and $d_{i\theta} = d_j$ for all i, A_B acts on V_i/V_{i+1} as $M_{d_j/e_j}(K_j)$ for some fixed isomorphism of V_i/V_{i+1} onto V_j/V_{j+1}.

We now consider V_i/V_{i+1} as an d_i-dimensional $F(G)$-module, where G is the group of units of the algebra A. Hence the action of G on V_i/V_{i+1} is isomorphic to $GL(d_i, F)$ if V_i/V_{i+1} is absolutely irreducible and isomorphic to $GL(d_i/e_i, K_i)$ if V_i/V_{i+1} is not absolutely irreducible and G_B is the direct product of $GL(d_j/e_j, K_j)$ for $j \in J$.

The algorithm tests every composition factor V_i/V_{i+1} for $i = 1, \ldots, n$ for absolute irreducibility.

2.5.1 Absolutely irreducible action

If V_i/V_{i+1} is absolutely irreducible then generators for $GL(d_i, F)$ are determined as described in Proposition 2.3.

Suppose V_i/V_{i+1} is isomorphic to composition factors $V_{j_1}/V_{j_1+1}, \ldots, V_{j_s}/V_{j_s+1}$ of V, the isomorphisms being given by $d_i \times d_i$ matrices $m_{j_1}, \ldots, m_{j_s}$. Then for each generator h of $GL(d_i, F)$ we determine a $d \times d$ matrix b having h as i-th diagonal block, $m_{j_k} h m_{j_k}^{-1}$ as j_k-th block for $k = 1, \ldots, s$, the identity in the remaining diagonal blocks and zero elsewhere.

2.5.2 Non-absolutely irreducible action

Suppose V_i/V_{i+1} is not absolutely irreducible. Then we want to determine a K-basis $\mathcal{B}$ for V_i/V_{i+1} such that the generators for $GL(d_i/e_i, K)$ with respect to this basis can be easily written down. First we use the Meataxe to find an $F(G)$-endomorphism α of V_i/V_{i+1} of order $q^{e_i} - 1$. Then $K = F\langle\alpha\rangle$. Now α is an $d_i \times d_i$ matrix over F which with respect to which $\mathcal{B}$ is a matrix with identical $e_i \times e_i$ blocks down the diagonal, *i. e.*, it acts on V_i/V_{i+1} as a diagonal K-matrix. Next we determine a composition series $V_i/V_{i+1} = W_1 > \cdots > W_n > W_{n+1} = \langle 0 \rangle$ for V_i/V_{i+1} as K-module. The composition factors W_j/W_{j+1} for $j = 1, \ldots, n$, are 1-dimensional K-spaces. Taking $v_j \in W_j \setminus W_{j+1}$ for $j = 1, \ldots, n$ and the basis $\{\alpha, \alpha^q, \ldots, \alpha^{q^{e_i-1}}\}$ of K over F we obtain the required basis

$$\mathcal{B} = \{v_1\alpha, \ldots, v_1\alpha^{q^{e_i-1}}, \ldots, v_n\alpha, \ldots, v_n\alpha^{q^{e_i-1}}\}.$$

Let β be one of the identical $e_i \times e_i$ blocks of α after changing basis to $\mathcal{B}$. Now we construct the generators for $GL(d_i/e_i, K)$ given in Proposition 2.3

as $d_i \times d_i$ matrices over F by interpreting 0 as an $e_i \times e_i$ block of zeros, 1 as the $e_i \times e_i$ identity and we take β to be the action of a primitive element of K on the required block.

For every generator of $GL(d_i/e_i, K)$ we now determine a $d \times d$ matrix b exactly as in the absolutely irreducible case.

In [16] Taylor gives pairs of generators for some matrix groups. The following proposition gives the generators for $GL(n, F)$ and we give an alternative proof in the case $F = GF(2)$.

Proposition 2.3. *a) Generators for $GL(n, GF(2))$ are*

$$\begin{pmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \quad \textit{and} \quad \begin{pmatrix} 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ & & & \ddots & \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}.$$

b) Let $p = 2$ and $m > 1$ or let $p > 2$ be a prime. Furthermore let x be a generator of the multiplicative group $GF(p^m)$. Then $GL(n, GF(p^m))$ is generated by the matrices

$$\begin{pmatrix} x & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ & & \ddots & \\ 0 & 0 & \cdots & 1 \end{pmatrix} \quad \textit{and} \quad \begin{pmatrix} 1 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}.$$

Proof of a). Let F be any field and define $n \times n$ matrices $B_{ij}(\lambda) = I_n + \lambda E_{ij}$. By [15, Chapter 1 Theorem 9.2] we have

$$SL(n, F) = \langle B_{ij}(\lambda) \mid i \neq j,\ \lambda \in F \rangle. \tag{$*$}$$

For $F = GF(2)$ clearly $GL(n, F) = SL(n, F)$. The first generating matrix is a permutation matrix which is clearly in $SL(n, F)$ and will be denoted P. Since F has two elements we only have to consider matrices $B_{ij}(\lambda)$ with $\lambda = 1$ which we will denote B_{ij}. Hence the second generating matrix is B_{12}. We want to prove that P and B_{12} generate all B_{ij}, $i \neq j$. It is easy to check that $P^r B_{1j} P^r = B_{r+1\,r+j}$ where suffixes are taken modulo n, for $j = 2, \ldots, n$, $r = 1, \ldots, n-1$, and that $(B_{1j} B_{j\,j+1})^2 = B_{1\,j+1}$ for $j = 2, \ldots, n-1$. Using these two relations we easily obtain all B_{ij}, and by $(*)$ our proof is completed. ■

2.6 Lifting generators of G_B to G

In the previous section we determined a generating set B for the group G_B. Considering the exact sequence $\langle 1 \rangle \longrightarrow G_P \longrightarrow G \longrightarrow G_B \longrightarrow \langle 1 \rangle$ as described in step 4 of our algorithm, we now want to lift the generators of G_B to G.

As described in section 2.3, we obtained generators for the algebra A by solving a certain system $\mathcal{S}$ of linear equations. In section 2.4 we obtained a change of basis matrix which enabled us to write the generators of A in block form. Using these generators in block form we can now rewrite the system $\mathcal{S}$ such that the solution of this rewritten system $\mathcal{S}_B$ is precisely the generating set of A in block form.

For each matrix $b \in B$ we determine a system of linear equations consisting of the system $\mathcal{S}_B$ to which we add equations fixing all block entries of b. This is a non-homogeneous system of linear equations and we determine one of its solutions. As a $d \times d$ matrix this solution is an element g_b of G that in the exact sequence maps to b.

2.7 Determining generators for G_P

With the algebra elements in block form we can easily recognise the 0-in-blocks ideal A_P of A consisting of the matrices with zero entries in the blocks. We obtain generators for A_P by solving a system of linear equations consisting of the system $\mathcal{S}_B$ as described in section 2.6, to which we add equations setting all block entries to zero.

The ideal A_P is clearly nilpotent, hence we obtain a generating set for G_P, which is unipotent, by adding the identity matrix to each generator of A_P.

2.8 Implementation issues

The commented code for the intersection of normalisers algorithm is printed out in Appendix A. It is written in GAP Version 3 and is planned to be translated to Version 4 in the near future.

The algorithm makes use of the 'matrix' package by D. Holt and others and of some code by A. Hulpke to determine the composition series of a G-module.

In this algorithm all vector spaces are row spaces and a row vector is a list of elements in a common field.

The intersection of the normalisers in $GL(V)$ of a list of subspaces of a finite dimensional vector space V over a finite field F is determined by a call to the function `IntersectionOfNormalisers` with input a list S of generators for the subspaces and a field F. The generators need not form bases for the subspaces. The output is a list containing the following elements:

1. G: a group record for the intersection of the normalisers in $GL(V)$ of the subspaces of V with generators in S; this record has a component 'size' containing the order of the group;

2. stab[1]: a list of $d \times d$ matrices over F which generate the block part of G (the lifted generators of G_B);

3. stab[2]: a list of $d \times d$ matrices over F which generate the 0-in-blocks part G_P of G.

The list 'solution' obtained in `IntersectionOfNormalisers` is a list of possibly singular $d \times d$ matrices over F and generates an algebra A, say. We want to consider the vector space $V = V(d, F)$ as an A-module, determine its composition series $V = V_n > V_{n-1} > \cdots > V_0 = \langle 0 \rangle$ and the isomorphisms between the composition factors. In GAP there are the functions `Module`, `NaturalModule` and `GModule` to define modules acted on by rings, algebras and matrix groups respectively. In the `GModule` case the group acts on a d-dimensional vector space over a finite field F.

When using the `GModule` structure there are functions available to determine the composition series, check for isomorphisms between modules and to check irreducibility and absolute irreducibility of modules. But such functions are not available for the `Module` and `NaturalModule` structures. Although the input for `GModule` is required to be either a matrix group or a list of non-singular matrices (*i. e.*, generators for a matrix group), most of the functions for G-modules do not make use of the non-singularity of the matrices. Hence in general these functions can also be used for A-modules. To be able to use these functions for our A-module we have to change one single line in the function `CompleteBasis` in the 'matrix' package. In line 43 of `CompleteBasis` we replace

```
while v[h] = zero do
```

by

```
while h <= d and v[h] = zero do
```

and this enables us to use the `GModule` structure for a finite dimensional vector space over a finite field acted on by a matrix algebra.

To determine the composition series of the A-module returned by

```
GModule( solution, F )
```

we use a modified version of A. Hulpke's function `CompositionSeriesGMod` which we call `CompositionSeriesAMod`. We replace the main `while` loop in `CompositionSereisGMod` by a recursion we call `CompositionSeriesRecursion`. In this recursion we introduce a function to check for isomorphisms between composition factors making use of the function `IsomorphismGModule`. Another modification is that we determine a change of basis matrix to reflect the composition series on the matrices of A. This means that the matrices of A when conjugated by this change of basis matrix become of the block form described in 1.3.1.

The composition series code is printed out in section 2 of Appendix A.

The function `IsAbsolutelyIrreducible` tests the irreducible module for absolute irreducibility. If the result is `false` then the dimension e of the centralising field K is determined. Also a matrix which centralises the module and has minimal polynomial of degree e over F is determined. The centralising matrix determined in GAP is not necessarily a primitive element of K, *i. e.*, it might not have order $q^e - 1$. To get a primitive element we have to call `FieldGenCentMat`.

In GAP it is very important to understand the difference between equality and identity of lists. Two lists are *equal* if their entries are equal. If we have a list A then the assignment `B := A;` does not create a new list but only creates a new name for the old list. In this case, if we change one element of B, then it is changed also in A. This is because A and B are not only equal but they are *identical*. These same definitions are valid also for records.

If we want to change a list with the same contents as A without changing A, then we have to make a copy of A. The functions `Copy` and `ShallowCopy` both return a new list that is equal but not identical to the old list. And the difference between `Copy` and `ShallowCopy` is that for

```
B := Copy( A );
```

the corresponding elements of A and B are equal whereas in the case of

```
B := ShallowCopy( A );
```

they are identical. This means that for making a copy of a vector over a field we can use `ShallowCopy` but for copying a matrix we have to use `Copy`.

Two important functions for lists which are used very often in our code are `Add` and `Append`. A call to these functions does not return any value.

They both take an existing list as first argument and a single new element or another list as second argument and change the first argument by respectively adding or appending the second argument to it.

We included a testing function `TestStab` in the code. This function tests if the $d \times d$ matrices over F given as first argument stabilise the subspaces of F^d whose bases are given as second argument. The user can turn off the testing function by setting `TestStabFlag` to `false`.

A few times throughout the algorithm the semi-echelon form of a matrix is determined. We say that a matrix is in *semi-echelon* form if the first nonzero element or leading term in every row is one, and all entries below these elements are zero. A matrix is in *full echelon* or *triangular* form if it is in semi-echelon form with the additional properties that for $j > i$ the leading term position of row j is bigger than that of row i, and that the columns of row leading term positions contain exactly one nonzero entry.

2.9 Performance

In order to give some indication of the performance of the GAP Version 3 implementation of the algorithm to determine the intersection of subspace normalisers we give in the table below some results and timings obtained by running the algorithm on a Pentium III PC. In the table we are using the following notation: F is the field, d the dimension of the full vector space, n the number of subspaces, $|G|$ the size of the intersection and t the time in seconds.

F	d	n	$\|G\|$	t
$GF(3)$	6	4	$2^7 \cdot 3$	0.1
$GF(2)$	15	4	$2^{31} \cdot 3^2 \cdot 5 \cdot 7 \cdot 31$	6
$GF(3)$	15	7	2	3.7
$GF(3)$	15	5	$2^2 \cdot 3^2$	3.4
$GF(5^3)$	15	6	$2^2 \cdot 31$	4.3
$GF(5^3)$	15	4	$2^{23} \cdot 3^6 \cdot 5^{138} \cdot 7^3 \cdot 19^2 \cdot 31^{10} \cdot 829^2$	7.8
$GF(2)$	25	9	1	156
$GF(2)$	25	6	2^5	35
$GF(3)$	25	7	2	330.9

Chapter 3

The canonical form of a subspace of V under the action of a p-subgroup of $GL(V)$

3.1 Introduction

Let $V = F^d$ be a d-dimensional vector space over a finite field F of characteristic p and let P be an upper uni-triangular subgroup of the matrix group $GL(V)$. In this chapter we will describe an algorithm to determine a canonical form of a subspace U of V under the action of P. This canonical form will be defined in terms of an order relation $\otimes$ on the orbit of U under P and will be proven to be unique. Hence we can decide whether two subspaces of V lie in the same orbit by determining and comparing their canonical forms. Together with the canonical form U_c of U the algorithm returns a list of generators for the stabiliser of U_c in P. Canonical form and stabiliser are determined without constructing the orbit of U under P.

Our canonical form algorithm requires the generators of P to form a special generating set called a base. The canonical form depends on the choice of basis for V, but not on the choice of base for P. If P is an arbitrary p-subgroup of $GL(V)$, an appropriate change of basis has to be performed before starting the canonical form calculation. Algorithms to determine the change of basis matrix and a base for P are described in section 3.2.

The first step in determining the canonical form of a subspace of V in its orbit under P is to determine the canonical form of a vector of V in its orbit under P. In section 3.3 we describe the algorithm to determine the canonical form of a vector. The algorithm to determine the canonical form of a subspace is described in section 3.4.

In section 3.5 are given the implementation issues and in section 3.6 we give some information about the performance of the algorithms.

3.2 Preparing the input

An important aspect to consider when doing computations with vector spaces is the choice of bases. The right choice of basis may allow us to use more efficient algorithms to solve the given problems. In our problem we have a p-group P acting on a d-dimensional vector space. Hence we can choose a basis $e_1, \ldots, e_d$ for V such that for $i = 1, \ldots, d$ the subspaces $V_i = \langle e_i, \ldots e_d \rangle$ of V satisfy $V_i g = V_i$ for all $g \in P$.

Definition 3.1. A chain of subspaces $V = V_1 > \cdots > V_d > 0$ satisfying the condition $V_i g = V_i$ for all $g \in P$ and $i = 1, \ldots, d$ is called a *P-invariant flag* for V.

A P-invariant flag for V can be determined as follows.

Algorithm: `PInvariantFlag`

Input: a vector space $V = F^d$;

a list $[x_1, \ldots, x_t]$ of matrices that generate a p-subgroup P of $GL(V)$

Output: a list $flag = [e_1, \ldots, e_d]$ of vectors such that the subspaces $V_i = \langle e_i, \ldots, e_d \rangle$ for $i = 1, \ldots, d$ form a P-invariant flag for V

begin

  $W_1 := V$;

  $k := 1$;

  while $W_k \neq \{0\}$ **do**

    $k := k + 1$;

    $W_k = \sum_{j=1}^{t} W_{k-1}(x_j - 1_d)$;

  end while; /* the while loop terminates as P is unipotent */

  $flag := [\ ]$;

  for i from 1 to k **do**

    add a factor basis for W_{i+1} in W_i to $flag$;

  end for;

  return $flag$;

end

Note: If U is a subspace of W and $w_1 + U, \ldots, w_k + U$ is a basis for W/U, then $w_1, \ldots, w_k$ is a factor basis for U in W.

In this chapter the vectors $e_1, e_2, \ldots, e_d$ will always be such that the subspaces $V_i = \langle e_i, \ldots e_d \rangle$ for $i = 1, \ldots, d$ form a P-invariant flag for V. Once the matrices in P are in upper uni-triangular form, $e_1, \ldots, e_d$ will always be the standard basis of V. But if the matrices in P are arbitrary, then we may use $[e_1, \ldots, e_d]^{-1}$ as change of basis matrix to get the generators of P into upper uni-triangular form.

Our algorithm to determine the canonical form of a subspace of V in its orbit under P requires the generating set of P to be a base.

Definition 3.2. A *base* for a p-group P of order p^n is a sequence of generators $g_1, g_2, \ldots, g_n$ of P such that defining $P_i = \langle g_i, \ldots, g_n \rangle$ for $i = 1, \ldots n$ the series

$$P = P_1 > P_2 > \cdots > P_{n+1} = \langle 1 \rangle$$

is a chief series of P.

By [15, Chapter 2 Theorem 1.12] we have $|P_i : P_{i+1}| = p$ for $i = 1, \ldots, n$.

Having the generators of P in upper uni-triangular form, a base for P is obtained by the algorithm `pGroupBase` given below.

Algorithm: `pGroupBase`

Input: a list X of $d \times d$ upper uni-triangular matrices over F that generate P

Output: a list *base* of $d \times d$ matrices over F that form a base for P

begin

eliminate 1_d from X;

$Y := X$;

$base := \emptyset$;

$row := 1$;

$col := 2$;

```
while Y ≠ ∅ do
   search for h ∈ Y with h[row, col] ≠ 0;
   if such h exists then
      a := h[row, col];
      add h to base;
      remove h from Y;
      for y ∈ Y with y[row, col] ≠ 0 do
         b := y[row, col];
         g := yh^(-a/b);  (a/b as integer in the range [1, ..., p - 1])
         if g ≠ 1_d then
            replace y by g;
         else
            remove y from Y;
         end if;
      end for;
      if h^p ≠ 1_d then
         add h^p to Y;
      end if;
      A := [h];
      B_h := [];
      while A ≠ [] do
         pick k in A;
         remove k from A;
         for x ∈ X do
            if [k, x] ≠ 1_d then        /* Commutator */
```

add $[k,x]$ to A;

add $[k,x]$ to B_h;

end if;

end for;

end while;

append B_h to Y;

end if;

if $col < d$ **then**

$row := row + 1$;

$col := col + 1$;

else

$col := col - row + 2$;

$row := 1$;

end if;

end while;

return $base$;

end

Our aim is to prove that the algorithm `pGroupBase` is correct. We start by establishing some notation.

Let P be a finite p-group with generating set X and let $h \in X$. Recursively we define sets B_i as follows.

$$\begin{aligned} B_1 &= [h, X] = \{[h,x];\ x \in X\} \\ B_i &= [h, \underbrace{X, \ldots, X}_{i}] = \{[b,x];\ b \in B_{i-1},\ x \in X\} \end{aligned}$$

Then

$$B_1 \subseteq B_1 \cup B_2 \subseteq \cdots \subseteq \cup_{i=1}^{k} B_i \subseteq \cup_{i=1}^{k+1} B_i \subseteq \cdots$$

and there is a least n such that $\cup_{i=1}^{n} B_i = \cup_{i=1}^{n+1} B_i$. For this least n we denote $B_h = \cup_{i=1}^{n} B_i$.

Lemma 3.1. *Let G be a p-group with generating set X and let $P = \langle Y \rangle$, $P \lhd G$ for some subset Y of G. If $h \in Y$ is such that for $Y_0 = Y \setminus \{h\}$ and $Q = \langle\, Y_0,\, h^p,\, B_h \,\rangle$ we have $h \notin Q$, then $|P : Q| = p$ and $Q \lhd G$.*

Proof. Let B be the subgroup of G generated by B_h. Then $B \lhd G$, hence we may divide out by this subgroup. In this new setup h is central, so we may divide out by $\langle h^p \rangle$ reducing to the case when h is central of order p.

By hypothesis $h \notin Q$, hence Q is a proper subgroup of P, implying that P is the direct product of Q and $\langle h \rangle$ and it follows that $|P : Q| = p$.

Furthermore $h \notin [P, G]$ and since we reduced to the case in which h is central of order p in P and consequently P is the direct product of Q and $\langle h \rangle$, it follows that $Q \lhd G$. ■

Next we define the depth of an upper uni-triangular matrix. This definition relies on an ordering of the pairs (i, j) with $1 \leqslant i < j \leqslant d$ given by

$$(i_1, j_1) \prec (i_2, j_2) \quad \text{if} \quad \begin{cases} j_1 - i_1 < j_2 - i_2 \ \text{ or} \\ j_1 - i_1 = j_2 - i_2 \ \text{ and } \ i_1 < i_2. \end{cases}$$

Definition 3.3. Let $g \neq 1_d$ be an upper uni-triangular $d \times d$ matrix. The *depth* of g is k, denoted $\mathrm{d}(g) = k$, if with respect to $\prec$ the first pair (i, j) with $g[i, j] \neq 0$ is the k-th pair. And we define $\mathrm{d}(1_d) = d(d-1)/2 + 1$.

Definition 3.4. A *loop invariant* for a while-loop is an assertion which is true when the while-loop first starts execution, and which is true after each complete execution of the statement sequence of that while-loop.

Theorem 3.1. *The algorithm* `pGroupBase` *having as input a list X of upper uni-triangular $d \times d$ matrices over a field F determines a base for the p-group $\langle X \rangle$.*

Proof. The algorithm starts by removing all copies of the identity matrix 1_d from X. Then we set Y to X, *base* to the empty list and initialise the row and column counter by setting *row* to 1 and *col* to 2. Next we enter the while-loop. We want to prove that this while-loop terminates after finitely many iterations and that a loop invariant for this while-loop is:

(a) $P = \langle\, base \cup Y\,\rangle$;

(b) if (row, col) is the k-th pair with respect to $\prec$ then $\langle Y \rangle = \{\, g \in P\,;\, \mathrm{d}(g) \geqslant k\,\}$;

(c) if (row, col) is the k-th pair with respect to $\prec$ and $k > 1$ then either old $Y = Y$ or $|\,\langle \text{old } Y\,\rangle : \langle Y \rangle\,| = p$, where old Y is the Y we had at the beginning of the previous iteration.

When starting the first iteration of the while-loop we have $P = \langle\, base \cup Y\,\rangle$, hence (a) is true. Pair $(row, col) = (1, 2)$ is first with respect to $\prec$, hence (c) is true. By definition of depth all upper uni-triangular matrices g satisfy $\mathrm{d}(g) \geqslant 1$ and since $P = \langle Y \rangle$ it follows that also (b) is true.

Suppose we are starting an iteration of the while-loop with *base*, Y, *row*, and *col* such that (a), (b) and (c) are true and (row, col) is the k-th pair with respect to $\prec$. First we look for $h \in Y$ with $h[row, col] \neq 0$, which means we are looking for $h \in Y$ with $\mathrm{d}(h) = k$. If no such h exists then $\mathrm{d}(y) \geqslant k+1$ for all $y \in Y$. Then we update *row* and *col*, but Y and *base* remain the same, hence (a) is true. Since old $Y = Y$ also (c) is true and clearly $\langle Y \rangle = \{\, g \in P\,;\, \mathrm{d}(g) \geqslant k \,\}$.

If there exists $h \in Y$ with $h[row, col] \neq 0$ then we set $a = h[row, col]$, add h to *base* and remove h from Y. Hence we still have $P = \langle\, base \cup Y \rangle$. Now we look for all remaining $y \in Y$ with $y[row, col] \neq 0$, *i.e.*, all $y \in Y$ with $\mathrm{d}(y) = k$. For each of them we set $b = y[row, col]$ and $g = yh^{-a/b}$ taking a/b as integer in the range $[1, \ldots, p]$. Then $g[row, col] = 0$ and since $\mathrm{d}(h) = \mathrm{d}(y) = k$ we clearly have $\mathrm{d}(g) \geqslant k+1$.

If $g = 1_d$ then we remove y from Y, else we replace y by g in Y such that eventually $\mathrm{d}(y) \geqslant k+1$ for all $y \in \langle\, Y \rangle$.

Next we determine h^p and if different from 1_d we add it to Y, noting that $h^p[row, col] = 0$. Then we determine the list of commutators B_h and add it to Y, noting that $b[row, col] = 0$ for each $b \in B_k$. Hence $\langle\, Y \rangle \leqslant \{\, g \in P\,;\, \mathrm{d}(g) \geqslant k+1 \,\}$ and as B_h is the set of all commutators $[h, z]$ for $z \in G$ it follows that $\{\, g \in P\,;\, \mathrm{d}(g) \geqslant k+1 \,\} \leqslant \langle Y \rangle$, so that equality holds.

Next we update *row* and *col*. For the new lists Y and *base* assertion (a) clearly remains true and by Lemma 3.1 also (c) remains true.

If $col < d$ then we increase *row* and *col* both by 1. Then old $(col - row) = col - row$ and $col =$ old $col + 1$ such that new (row, col) is the $(k+1)$-th pair with respect to our pair ordering. If $col = d$ then we replace *col* by $col - row + 2$ and *row* by 1. Then $col - row =$ old $(col - row) + 1$ and from old $col = d$, $row = 1$ follows that (row, col) is the $(k+1)$-th pair with respect to $\prec$. Hence in all cases (b) is true at the end of the iteration.

The list Y will never contain the identity matrix 1_d which has depth $d(d-1)/2+1$, hence at the end of iteration $d(d-1)/2$ the list Y will be empty, terminating the while-loop. Furthermore loop invariant (c) assures that at the end of the while-loop *base* will be a base for P. ■

3.3 Canonical form of a vector under a p-group

The canonical form of a vector is defined in terms of an order relation on the vectors in V and this order relation is defined in terms of the set

$$Z_v = \{\, i \mid v = a_1e_1 + \cdots + a_de_e \text{ and } a_i = 0\}, \quad \text{for } v \in V.$$

It is important to notice that Z_v and consequently $\otimes$ depend on the ordered basis $e_1, \ldots, e_d$ chosen for V. In our case this basis is chosen such that the subspaces $V_i = \langle\, e_i \ldots, e_d \,\rangle$ for $i = 1, \ldots, d$ form a P-invariant flag for V.

For the factor space V/V_i we choose the basis $\{e_1 + V_i, \ldots, e_{i-1} + V_i\}$ and define the sets

$$Z_{v+V_i} = \{\, j \mid v + V_i = a_1e_1 + \cdots + a_{i-1}e_{i-1} + V_i \text{ and } a_j = 0\,\}$$

for $i = 2, \ldots, d$.

Definition 3.5. Let $X, Y \subseteq \{1, \ldots, d\}$. We say that $X < Y$ if one of the following occur:

(a) $X \neq \emptyset$ and $Y = \emptyset$;

(b) $X \neq \emptyset$, $Y \neq \emptyset$ and $\min X < \min Y$;

(c) $X \neq \emptyset,\ \ Y \neq \emptyset,\ \ \min X = \min Y = k$ and $X \setminus \{k\} < Y \setminus \{k\}$.

The relation defined above is a total order on the subsets of $\{1, \dots, d\}$.

Definition 3.6. Given vectors v and w in V we define the relations $\olessthan$ and $\circledequal$ as follows:

$$
\begin{aligned}
v \olessthan w \quad & \text{if} \quad Z_v < Z_w, \\
v \circledequal w \quad & \text{if} \quad Z_v = Z_w, \\
v + V_i \olessthan w + V_i \quad & \text{if} \quad Z_{v+V_i} < Z_{w+V_i}, \\
v + V_i \circledequal w + V_i \quad & \text{if} \quad Z_{v+V_i} = Z_{w+V_i}.
\end{aligned}
$$

The relation $\olessthan$ is a partial order on the vectors in V.

Definition 3.7. The *canonical form* of a vector $v \in V$ in its orbit under P is a vector v_c in this orbit which is minimal with respect to $\olessthan$.

We will prove in Theorem 3.2 that this canonical form is unique in the orbit of v under the action of P.

Our algorithm to determine the canonical form of a vector in its orbit under P relies on the concept of weight of an element of P with respect to a given vector.

Definition 3.8. For $g \in P$ and $v \in V$ the *weight* of g with respect to v is given by

$$
\mathrm{wt}_v(g) = \begin{cases} d+1, & \text{if } v = vg \\ \max\{\, j \mid v = vg \bmod \langle e_j, \dots, e_d \rangle \}, & \text{otherwise.} \end{cases}
$$

In the next section we will extend the definition of weight with respect to a vector to weight with respect to a subspace and for the latter it will be convenient being able to express weight in terms of depth.

Definition 3.9. The *depth* of a vector v is given by

$$\mathrm{d}(v) = \begin{cases} d+1, & \text{if } v = 0 \\ \min\{\, j \mid v = a_1e_1 + \cdots + a_de_d \text{ and } a_j \neq 0\}, & \text{otherwise.} \end{cases}$$

It follows clearly from the definitions that $\mathrm{wt}_v(g) = \mathrm{d}(v - vg)$.

We are using the same notation $\mathrm{d}(\ )$ to represent the depth of a matrix and the depth of a vector. This should cause no confusion because the context always makes clear if we are referring to matrices or vectors.

The canonical form of a vector $v \in V$ in its orbit under P is obtained by the algorithm `VectorCanonicalForm` given below. The algorithm basically consists of a while loop in which at each iteration the element of minimal weight is removed from the set of generators of P. It is essential for the correctness of our result that the algorithm goes through all possible weights for $g \in P$. This is achieved by using a base as generating set for P, as we will see in the proof of Theorem 3.2.

Algorithm: `VectorCanonicalForm`

Input:	a base X for P;
	a vector v_0;
Output:	v_0 is replaced by its canonical form v;
	an element x of P such that $v_0x = v$;
	X is replaced by a base for the stabiliser of v in P

begin

$v := v_0$;

$j_0 := \min\{\mathrm{wt}_v(g) \mid g \in X\}$;

$x := 1_{d\times d}$;

while $j_0 < d+1$ **do**

pick some $g \in X$ with $\mathrm{wt}_v(g) = j_0$;

$v := vg^{\alpha}$, α such that $v = \sum_{i=1}^{d} \lambda_i e_i$ with $\lambda_{j_0} = 0$;

$x := xg^{\alpha}$;

for $h \in X \setminus \{g\}$ **do**

if $\mathrm{wt}_v(h) = j_0$ **then**

$h := hg^{\beta}$, β such that $\mathrm{wt}_v(h) > j_0$;

end if;

end for;

$X := X \setminus \{g\}$;

$j_0 := \min\{\mathrm{wt}_v(g) \mid g \in X\}$;

end while;

return $v,\ x,\ X$;

end

The correctness of the algorithm `VectorCanonicalForm` will be proved in Theorem 3.2 and this requires the following lemmas.

Lemma 3.2. *Let v be a vector in a finite dimensional vector space V over a finite field F of characteristic p and let X be a generating set for a p-subgroup P of the matrix group $GL(V)$. Then*

$$\min\{\mathrm{wt}_v(g)\,;\ g \in X\} = \min\{\mathrm{wt}_v(g)\,;\ g \in P\}.$$

Proof. Clearly $\min\{\mathrm{wt}_v(g) \mid g \in X\} \geqslant \min\{\mathrm{wt}_v(g) \mid g \in P\}$. Suppose $g_1, g_2 \in X$ with $\mathrm{wt}_v(g_1) = i$, $\mathrm{wt}_v(g_2) = j$ and let $V = V_1 > \cdots > V_d > \langle 0 \rangle$

with $V_i = \langle e_i, \ldots, e_d \rangle$ for $i = 1, \ldots, d$ be a P-invariant flag for V. Then

$$\begin{aligned} v(1-g_1) &\in V_i, \\ v(1-g_2) &\in V_j, \\ v(1-g_1)(1-g_2) &\in V_{\min\{i,j\}}. \end{aligned}$$

Therefore

$$v(1-g_1g_2) = -v(1-g_1)(1-g_2) + v(1-g_1) + v(1-g_2) \in V_{\min\{i,j\}}.$$

Hence $\mathrm{wt}_v(g_1g_2) \geqslant \min\{\mathrm{wt}_v(g_1), \mathrm{wt}_v(g_2)\}$, completing the proof. ■

Lemma 3.3. *Let $v, w \in V$. Then $v \olessthan w$ if and only if $v + V_i \ominus w + V_i$ for $i = 2, \ldots t \leqslant d$ and $v + V_i \olessthan w + V_i$ for $i = t+1, \ldots, d+1$.* ■

Theorem 3.2. *Let V be a d-dimensional vector space over a finite field F of characteristic p and let $v_0 \in V$. Let X be a base for a p-subgroup P of the matrix group $GL(V)$ and let*

$$V = \langle e_1, \ldots, e_d \rangle > \cdots > \langle e_{d-1}, e_d \rangle > \langle e_d \rangle > 0$$

be a P-invariant flag for V. Then the algorithm `VectorCanonicalForm` *replaces v_0 by the unique canonical form v of v_0 in its orbit under P, determines an element $x \in P$ such that $v_0 x = v$ and replaces X by a base for the stabiliser of v in P.*

Proof. The algorithm starts by setting v to v_0, determining

$$j_0 = \min\{\mathrm{wt}_v(h) \mid h \in X\}$$

and setting x to the $d \times d$ identity matrix.

If $j_0 = d+1$ then $v = vh$ for all $h \in X$, hence $v = vh$ for all $h \in P$. Then $v = v_0$ which is clearly the unique minimal element with respect to $\lessdot$ in its orbit under P.

In case $j_0 < d+1$ we enter a while-loop with j_0, a vector $v = a_1e_1 + \cdots + a_de_d$, a matrix x and a list X of matrices which is a base for P. We want to prove that the while-loop terminates after finitely many iterations and that a loop invariant for this while-loop is:

(a) $v_0x = v$;

(b) X is a base for the stabiliser in P of $v + V_{j_0}$;

(c) $v + V_{j_0} \lessdot v_0h + V_{j_0}$ or $v + V_{j_0} = v_0h + V_{j_0}$ for $h \in P$.

When starting the first iteration of the while-loop we have $v = v_0$ and $x = 1_d$, hence (a) is true. By definition of j_0 we have $v + V_{j_0} = vh + V_{j_0}$ for all $h \in X$ and by Lemma 3.2 for all $h \in \langle X \rangle = P$. Hence (c) is true, $\langle X \rangle$ is the stabiliser of $v + V_{j_0}$ in P and $v + V_{j_0} = v_0 + V_{j_0}$ is minimal with respect to $\lessdot$ in its orbit under P, such that (b) is true.

We start an iteration of the while-loop by picking a matrix $g \in X$ with $\mathrm{wt}_v(g) = j_0$. Such g exists by construction of j_0. Now we determine the least $\alpha > 0$ such that $vg^\alpha = \sum_{i=1}^d \lambda_i e_i$ with $\lambda_{j_0} = 0$. Then

$$vg^\alpha + V_{j_0+1} \lessdot vh + V_{j_0+1} \quad \text{for all} \quad h \in \langle X \rangle. \tag{1}$$

Next we set $v = vg^\alpha$ and $x = xg^\alpha$. Then clearly $v = v_0x$, hence (a) remains true.

In the proof of Lemma 3.2 we saw that $\mathrm{wt}_v(g_1, g_2) \geqslant \min\{\mathrm{wt}_v(g_1), \mathrm{wt}_v(g_2)\}$ and as j_0 is the least weight of elements in X it follows that $\mathrm{wt}_v(hg^\beta) \geqslant j_0$

for any β. Let $h \in X$, $h \neq g$ be such that $\mathrm{wt}_v(h) = j_0$. Then

$$\begin{aligned} v &= \sum_{i=1}^{d} \lambda_i e_i, \quad \lambda_{j_0} = 0 \\ vh &= \sum_{i=1}^{d} \mu_i e_i, \quad \mu_{j_0} \neq 0, \quad \mu_i = \lambda_i \text{ for } i < j_0 \\ vg &= \sum_{i=1}^{d} \nu_i e_i, \quad \nu_{j_0} \neq 0, \quad \nu_i = \lambda_i \text{ for } i < j_0. \end{aligned}$$

Then

$$vhg^\beta = \sum_{i=1}^{d} \xi_i e_i, \quad \xi_{j_0} = \mu_{j_0} + \beta\nu_{j_0}, \quad \xi_i = \lambda_i \text{ for } i < j_0,$$

hence we can find β such that $\mu_{j_0} + \beta\nu_{j_0} = 0$, *i. e.*, we can find β such that $\mathrm{wt}_v(hg^\beta) > j_0$. Now we replace all $h \in X$, $h \neq g$ with $\mathrm{wt}_v(h) = j_0$ by hg^β for convenient integers β such that $\mathrm{wt}_v(hg^\beta) > j_0$. Then we remove g from X and determine a new j_0. This j_0 is strictly bigger than the previous one, proving that the while-loop terminates after at most $d + 1 - j_0$ (the first j_0) iterations. The new list X clearly remains a base for $\langle X \rangle$ and since $j_0 = \min\{\mathrm{wt}_v(x) \mid x \in X\}$ it follows that

$$v + V_{j_0} = vh + V_{j_0} \text{ for all } h \in \langle X \rangle. \tag{2}$$

The groups $\langle \mathrm{old}\, X \rangle$ and $\langle X \rangle$ are consecutive terms in a chief series of P, hence $\langle X \rangle$ is maximal among normal subgroups of P which are properly contained in $\langle \mathrm{old}\, X \rangle$. Hence, if $\{j_1, \ldots, j_k\} = \{\mathrm{wt}_v(h) \mid h \in \mathrm{old}\, X\}$ has minimal term j_0', then $j_0 = \min\{\mathrm{wt}_v(h) \mid h \in X\} = \min\{j_1, \ldots, j_k\} \setminus \{j_0'\}$. This means that we do not miss out any $j \in \{\mathrm{wt}_v(h) \mid h \in P\}$ in between j_0' and j_0. Therefore

$$v + V_{j_0} \neq vh + V_{j_0} \text{ for all } h \in P \setminus \langle X \rangle. \tag{3}$$

Now it follows from (2) and (3) that $\langle X \rangle$ is the stabiliser of $v + V_{j_0}$ in P, proving that (b) remains true. Furthermore it follows from (1) that

$$v + V_{j_0} \otimes vh + V_{j_0} \text{ for all } h \in P \setminus \langle X \rangle,$$

proving that (c) remains true.

When we reach $j_0 = d + 1$ we have $v + V_{j_0} = v$, hence X is a base for the stabiliser of v in P. From Lemma 3.3 follows $v \otimes vh$ for all $h \in P$ with $v \neq vh$. Hence v is the canonical form of v_0 in its orbit under P. ■

3.3.1 Example

In this section we determine the canonical form of the vector $v_0 = (0, 1, 1)$ over $GF(2)$ under the action of a p-group P generated by a list of matrices $X = [g_1, g_2, g_3]$ where

$$g_1 = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, g_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}, g_3 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

The matrices in X are upper uni-triangular and form a base for P. Following the algorithm `VectorCanonicalForm` we set $v = (0, 1, 1)$ and determine $j_0 = \{\mathrm{wt}_v(g) \mid g \in X\}$.

$$\left.\begin{array}{lcl} vg_1 = (0,1,1) & \Longrightarrow & \mathrm{wt}_v(g_1) = 4 \\ vg_2 = (0,1,0) & \Longrightarrow & \mathrm{wt}_v(g_2) = 3 \\ vg_3 = (0,1,1) & \Longrightarrow & \mathrm{wt}_v(g_3) = 4 \end{array}\right\} \Longrightarrow j_0 = 3$$

Furthermore we set $x = 1_{3\times 3}$. Now $j_0 < 4$ and as $\mathrm{wt}_v(g_2) = 3$ we set $g = g_2$. Next we determine α to be 1 as $vg_2 = (0, 1, 0)$ has coefficient 0 for e_3. Then we set $v = vg_2$ and $x = g_2$. There is no further $h \in X$ having weight 3 hence we now set $X = [g_1, g_3]$ and determine a new j_0.

$$\left.\begin{array}{rcl} vg_1 = (0,1,0) & \Longrightarrow & \mathrm{wt}_v(g_1) = 4 \\ vg_3 = (0,1,0) & \Longrightarrow & \mathrm{wt}_v(g_3) = 4 \end{array}\right\} \Longrightarrow \quad j_0 = 4$$

This completes the calculations, hence the canonical form of $(0,1,1)$ under P is $(0,1,0)$, a base for the stabiliser in P of this canonical form is $[g_1, g_3]$ and g_2 is an element of P which transforms $(0,1,1)$ into its canonical form.

3.4 Canonical form for a subspace of V under a p-group

Let $V = V_1 > \cdots > V_d > 0$ be a P-invariant flag for V and let U be a subspace of V. By intersecting the P-invariant flag of V with U and deleting repeated subspaces we obtain a Q-invariant flag for U

$$U = U_1 > \cdots > U_m > 0$$

with $U_i = U \cap V_{f(i)} = \langle u_i, \ldots, u_m \rangle$ for $i = 1, \ldots, m$ where the function $f : \{1, \ldots, d\} \to \{1, \ldots, m\}$ reflects the fact of repeated subspaces having been deleted and where Q is the normaliser of U in P. Hence $u_1, \ldots, u_m$ is the appropriate basis to be used for U when determining the canonical form of U under the action of P. In this section the vectors $u_1, \ldots, u_m$ will always be such that the subspaces $U_i = \langle u_i, \ldots, u_m \rangle$ for $i = 1, \ldots, m$ form a Q-invariant flag for U. As noted in section 3.2, since we require the matrices in P to be upper uni-triangular, we will have $V_i = \langle e_i, \ldots, e_d \rangle$ for $i = 1, \ldots, d$ where $e_1, \ldots, e_d$ is the standard basis for V. Hence $u_1, \ldots, u_m$ will always be the echelon form of the basis for U given as input.

If for $g \in P \setminus Q$ we have $Ug = W$, then $U_i g = W_i$, where $W_i = W \cap V_{f(i)}$ for $i = 1, \ldots, m$.

Now we extend the definitions of $\olessthan$, canonical form and weight given in the previous section for a vector in V to a definition for a subspace of V.

Definition 3.10. Given two m-dimensional subspaces U and W of V with invariant flags $U = U_1 > \cdots > U_m > 0$ and $W = W_1 > \cdots > W_m > 0$, respectively, we say that $U_i \olessthan W_i$ if one of the following occurs:

(a) $i = m,\ \ U_m = \langle u \rangle,\ \ W_m = \langle w \rangle$ and $u \olessthan w$;

(b) $i < m$ and $U_{i+1} \olessthan W_{i+1}$;

(c) $i < m,\ \ U_{i+1} = W_{i+1},\ \ U_i = \langle U_{i+1}, u \rangle,\ \ W_i = \langle W_{i+1}, w \rangle$ and $\min_{\olessthan}\{u + x \mid x \in U_{i+1}\} \ \olessthan\ \min_{\olessthan}\{w + x \mid x \in W_{i+1}\}$.

The relation $\olessthan$ is a partial order on the subspaces of V.

We are using the same symbol $\olessthan$ to represent the order relation for vectors and subspaces. Again this should cause no confusion because the context always makes clear if we are comparing vectors or subspaces.

Definition 3.11. The *canonical form* of a subspace $U < V$ in its orbit under P is a subspace U_c in its orbit which is minimal with respect to $\olessthan$.

Definition 3.12. Let U be a subspace of V with basis $\mathcal{B} = \{v, u_{m-k}, \ldots, u_m\}$, where $\langle u_{m-k}, \ldots, u_m \rangle$ is in canonical form under the action of P and let $g \in P$. The *weight* of g with respect to $\mathcal{B}$ is given by

$$\mathrm{wt}_{\mathcal{B}}(g) = \begin{cases} \mathrm{d}(v - vg), & \text{if } \ \mathrm{d}(v - vg) \notin \{\mathrm{d}(u_{m-k}), \ldots, \mathrm{d}(u_m)\} \\ \mathrm{d}(v - vg - \lambda_{i_1} u_{i_1} - \cdots - \lambda_{i_r} u_{i_r}), & \text{if the following occurs} \end{cases}$$

$$
\begin{aligned}
&\mathrm{d}(v - vg) = \mathrm{d}(u_{i_1}),\\
&\mathrm{d}(v - vg - \lambda_{i_1} u_{i_1}) = \mathrm{d}(u_{i_2}),\\
&\vdots\\
&\mathrm{d}(v - vg - \lambda_{i_1} u_{i_1} - \cdots - \lambda_{i_r - 1} u_{i_r - 1}) = \mathrm{d}(u_{i_r}),\\
&\mathrm{d}(v - vg - \lambda_{i_1} u_{i_1} - \cdots - \lambda_{i_r} u_{i_r}) \notin \{\mathrm{d}(u_{m-k}), \ldots, \mathrm{d}(u_m)\}
\end{aligned}
$$

where λ_{i_j} is such that the coefficient of u_{i_j} in $v - vg - \lambda_{i_1} u_{i_1} - \cdots - \lambda_{i_j} u_{i_j}$ is zero for $j = 1, \ldots, r$.

Note that the definition of depth remains precisely the same we had in section 3.3, being given in terms of the basis $e_1, \ldots, e_d$ of V.

The canonical form of a subspace $U = \langle u_1, \ldots, u_m \rangle$ of V in its orbit under P is determined by stepping up the invariant flag $U = U_1 > \cdots > U_m > 0$. Starting with $U_m = \langle u_m \rangle$ whose canonical form is determined by the algorithm `VectorCanonicalForm`, our algorithm takes as input the canonical form of U_i and determines the canonical form of U_{i-1}, until we reach the full subspace U. This algorithm is called `NextSubspCanonicalForm` and is basically the same as the algorithm `VectorCanonicalForm` differing only in two points. The first difference is that we replace the function that determines the weight with respect to a vector by a function that determines the weight with respect to a subspace. The second difference is that we determine and store the depths of the vectors already dealt with since they are needed to determine the weights with respect to subspaces.

3.4.1 Example

In this section we calculate the canonical form of the 2-dimensional subspace $U = \langle (1,0,1), (0,1,1) \rangle$ of $V = GF(2)^3$ under the action of the same group $P = \langle X \rangle$, $X = [g_1, g_2, g_3]$ as in example 3.3.1.

The matrices in X are in upper uni-triangular form, hence the P-invariant flag for V is given by the standard basis $e_1 = (1,0,0)$, $e_2 = (0,1,0)$, $e_3 = (0,0,1)$. The list X is a base for P and the basis given for U is in triangular form, hence we start by determining the canonical form of the vector $(0,1,1)$ under P. This was already done in example 3.3.1 where we obtained

$$u = (0,1,0), \quad x = g_2, \quad X = [g_1, g_3].$$

Now we multiply each basis element of U by x, obtaining

$$U = \langle (1,0,1), (0,1,0) \rangle,$$

where the last vector is in canonical form. Then we set up a list *depths* of length dimension of U, containing at its last position the depth of u: $depths = [\ \ , 2]$.

The next step is to determine the canonical form under $\langle g_1, g_3 \rangle$ of the subspace generated by the next vector in the basis of U which is $v = (1,0,1)$ and the vectors already dealt with. Since U in our example has dimension 2, this is the last step in our calculation.

We have $\mathcal{B} = \{(1,0,1), (0,1,0)\}$ and determine the weight of g_1 and g_2 with respect to $\mathcal{B}$.

$$\begin{aligned} \mathrm{d}(v - vg_1) &= \mathrm{d}((0,1,0)) = 2 \in depths \\ \mathrm{d}(v - vg_1 - u) &= \mathrm{d}((0,0,0)) = 4 \\ \mathrm{d}(v - vg_3) &= \mathrm{d}((0,0,1)) = 3 \notin depths. \end{aligned}$$

Hence $\mathrm{wt}_{\mathcal{B}}(g_1) = 4$ and $\mathrm{wt}_{\mathcal{B}}(g_3) = 3$. The vector v is already in echelon form with respect to u.

Next we determine α to be 1 as $vg_3 = (1,0,0)$ has coefficient 0 for e_3. Then we set $v = vg_3$ which is in echelon form with respect to u and set $x = xg_3 = g_2g_3$. There are no more matrices of weight 3, hence we set $X = [g_1]$. But 3 is the dimension of V, hence we are done.

So the canonical form of $U = \langle (1,0,1),(0,1,1) \rangle$ under $P = \langle g_1, g_2, g_3 \rangle$ is $\langle (1,0,0),(0,1,0) \rangle$, the normaliser of this canonical form under P is the group $\langle g_1 \rangle$ and the matrix in P transforming U into its canonical form is $x = g_2g_3$.

3.5 Implementation issues

The commented GAP Version 3 code for the canonical form of a subspace under the action of a p-group is printed out in Appendix B. The code for the three functions `FullEchelonBase`, `SemiEchelonBase` and `IntersectionMat` which are also used in the intersection of subspace normalisers algoritm is printed out in Appendix A.

The canonical form of a subspace U of V under the action of a p-subgroup P of the matrix group $GL(V)$ is obtained by a call to the function

`SubspaceCanonicalForm(X,U,F)`.

In case P is an arbitrary p-subgroup of $GL(V)$ we first have to determine a P-invariant flag for V. This is done by a call to the function `PInvariantFlag(M,d,F)`. It is important to notice that the matrices in M are not generators of P, but generators of the corresponding nilpotent algebra, obtained by subtracting the identity from the matrices in X. Then we change basis of the matrices in X to get them into upper uni-triangular form. Next we determine a base for P by a call to the function `pGroupBase(X)`.

There is a function in GAP3 called `SumIntersectionMat` which performs a Zassenhaus algorithm to compute bases for the sum and the intersection of spaces generated by the vectors in two lists $M1$ and $M2$. In the intersection of subspace normalisers algoritm we only need to determine intersections of subspaces, while in the canonical form algoritm we need sums and intersections, but for different subspaces. When computing sums of subspaces of a vector space of large dimension it is more efficient not to perform the whole Zassenhaus algorithm, but only the part concerning the sum. In this case, instead of semi-echelonising a matrix with $2m$ columns, we semi-echelonise a matrix with m columns, where m is the length of the generating vectors. Therefore we do not use the function `SumIntersectionMat`, but two functions `SumMat` and `IntersectionMat` which perform only the parts of the Zassenhaus algoritms required in each case. Furthermore there was a small bug in the `SumIntersectionMat` function leading to a wrong result in the special case when $M1$ is an empty list and $M2$ contains only the zero vector. The very straightforward fix was done in the function `SumMat`.

3.6 Performance

In order to give some indication on the performance of the GAP Version 3 implementation of the algorithm to determine the canonical form of a subspace under the action of a p-group we give in the table below some results and timings obtained by running the algorithm on a Pentium III PC. In all examples we use the field $GF(2)$. The notation used in the table is the following: d is the dimension of the full vector space, dim is the dimension of the subspace whose canonical form is being determined, n is the number of generators given for the p-group acting on the subspace, $|P|$ is the size of the p-group, $|S|$ is the size of the stabiliser of the canonical form determined by the algorithm, t_B is the time taken to determine a base for P and t is the total time in seconds.

d	dim	n	$\|P\|$	$\|S\|$	t_B	t
17	9	3	2^{86}	2^{32}	7409.8	7410.62
17	7	2	2^{39}	2^{10}	14.73	15.36
17	7	1	2^{3}	1	0.01	0.019
21	7	2	2^{87}	2^{26}	1034.96	1036.39
20	4	2	2^{77}	2^{44}	359.86	360.71

Appendix A

Stabiliser code

A.1 The main code

```
TestStabFlag := true;
TestSizeFlag := true;
RequirePackage( "matrix" );
################################################################
# FullEchelonFactorBase( V, U ) . . . . computes a full echelon
#                   factor basis for U in V, where U and V are
#                   subspaces of F^d satisfying:
#                   - V and U in full echelon form
#                   - U is subspace of V
# DANGER!!! The program doesn't check if V and U satisfy the two
#           conditions
# Definition: If U is a subspace of V and v_1+U,...,v_k+U is a
#             basis for V/U, then v_1,...,v_k is a factor basis
#             for U in V
FullEchelonFactorBase := function( V, U )
   local fac, dimV, dimU, Vrow, Urow, col, zero;
   zero := 0 * V[1][1];
   fac := [];
   dimV := Length( V );
   dimU := Length( U );
```

```
   Urow := 1;
   col := 1;
   for Vrow in [ 1 .. dimV ] do
      while V[Vrow][col] = zero do
         col := col + 1;
      od;
      if Urow > dimU or U[Urow][col] = zero then
         Add( fac, V[Vrow] );
      else
         Urow := Urow + 1;
      fi;
   od;
   if Length( fac ) + dimU <> dimV then
      Error( "U is not a subspace of V \n" );
   fi;
   return fac;
end;
#################################################################
# SemiEchelonFactorBase( V, U ) . . . computes a basis in semi
#                echelon form for the complement of U in V, where
#                U and V are subspaces of F^d satisfying:
#                - U and V in semi-echelon form
#                - U is subspace of V
# DANGER!! The program doesn't check if conditions are satisfied
SemiEchelonFactorBase := function( V, U )
   local F, fac, L1, L2, dimV, i;
   F := Field( V[1][1] );
   fac := [];
   L1 := LeadingTermPositions( V, F );
   L2 := LeadingTermPositions( U, F );
   dimV := Length( V );
   for i in [ 1 .. dimV ] do
      if not ( L1[i] in L2 ) then
         Add( fac, V[i] );
      fi;
   od;
   if Length( U ) + Length( fac ) <> dimV then
      Error ( "U is not a subspace of V \n" );
```

```
   fi;
   return fac;
end;
##################################################################
# LeadingTermPositions( mat, F )
# INPUT - mat: semi-echelonised matrix over F with no zero rows
#       - F: field
# OUTPUT - a list ‘heads’ with heads[i] = position of first
#          nonzero entry in the i-th row of ‘mat’
# NOTE: output might be wrong if first element in each row of
#       ‘mat’ is not 1
LeadingTermPosition := function( mat, F )
   local heads, row;
   heads := [];
   for row in [ 1 .. Length( mat ) ] do
      heads[ row ] := Position( mat[ row ], F.one );
   od;
   return heads;
end;
##################################################################
# Belong ( sub, list, subsp_list ) . checks if ‘sub’ is in ‘list’
# USE: only in CleanUpAndSort
# INPUT - sub: echelonised basis for subsp. (elt of ‘subsp_list’)
#       - list: list of integers indicating the position in
#         ‘subsp_list’ of processed subspaces of dim. dim(sub)
#       - subsp_list: list of generating sets for subspaces of
#                     V(d,F)(some already processed) given by user
# OUTPUT - true if the integer giving the position of ‘sub’ in
#          ‘subsp_list’ is already in ‘list’ and false otherwise
Belong := function( sub, list, subsp_list )
   local j, t, found;
   j := 1;
   t := Length( list );
   found := false;
   while not found and j <= t do
      if sub = subsp_list[ list[j] ] then
         found := true;
      else
```

```
        j := j + 1;
      fi;
   od;
   return found;
end;
#################################################################
# CleanUpAndSort ( pos, Subsp, subsp_list, d, F, keep ) . . . if
#         subsp_list[pos] is not trivial or V and is not already
#         in ‘Subsp’, inserts it there according to its dimension
# INPUT - pos: the position in ‘subsp_list’ of the subspace that
#              is being processed
#       - Subsp : Subsp[i] is a list containing the positions of
#                the subspaces of dimension d-i in ‘subsp_list’
#       - subsp_list: list of generating sets for subspaces of
#                     F^d (some already processed)
#       - d: dimension of full vector space
#       - F: field
#       - keep: list with pos. of non-repeated, non-trivial and
#               already processed subspaces in ‘subsp_list’
CleanUpAndSort := function( pos, Subsp, subsp_list, d, F, keep )
   local dim, t, sub, zero, ls;
   sub := subsp_list[ pos ];
   ls := Length( sub );
   if ls > 0 then
      # check
      if not IsMat( sub ) then
         Error("subspace[",pos,"] has to be a matrix\n");
      elif Length( sub[1] ) <> d then
         Error("subspaces must have same parent space\n");
      fi;
      # determine dimension of subspace
      TriangulizeMat( sub );
      zero := List( [1 .. d ], x -> F.zero );
      dim := ls;
      while dim > 0 and sub[ dim ] = zero do
         dim := dim - 1;
      od;
      # delete the zero rows
```

```
      if dim < ls then
         sub := sub{ [ 1 .. dim ] };
      fi;
      if 0 < dim and dim < d then
         t := d - dim;    # position in ‘Subsp’ of sublist that
                          # shall contain ‘sub’
         # check if ‘sub’ is already in Subsp[t]
         if not Belong( sub, Subsp[t], subsp_list ) then
            subsp_list[ pos ] := sub;
            Add( keep, pos );
            Add( Subsp[t], pos );
         fi;
      fi;
   fi;
end;
#################################################################
# SysLinEqn( U, F, d ) . . determines system of linear equations
#                  in indeterminates x_1, ..., x_d^2 satisfying
#                  U * X = U, where X is the indeterminate matrix
# INPUT - U: semi-echelonised basis of subspace for which linear
#            equations are being determined
#       - F: field
#       - d: dimension of parent vector space
# NOTE: output might be wrong if U is not in semi-echelon form
SysLinEqn := function( U, F, d )
   local zeroeqn, heads, sys, dimU, i, row, col, eqn, c;
   zeroeqn := List( [ 1 .. d^2 ], x -> F.zero );
   heads := LeadingTermPositions( U, F );
   dimU := Length( U );
   sys := [];
   for i in [ 1 .. dimU ] do
      # determine equations for U[i]*X = (y_1, ..., y_d ) in U
      for col in [ 1 .. d ] do
         eqn := ShallowCopy( zeroeqn );
         # equation for y_col = U[1][col]*y_heads[1] + ...
         #                          + U[dimU][col]*y_heads[dimU]
         for row in [ 1 .. dimU ] do
            for c in [ 1 .. d ] do
```

```
              eqn[(c-1)*d+heads[row]] := U[row][col] * U[i][c];
           od;
        od;
        for c in [ 1 .. d ] do
           eqn[(c-1)*d+col] := eqn[(c-1)*d+col] - U[i][c];
        od;
        if eqn <> zeroeqn then
           Add( sys, eqn );
        fi;
     od;
   od;
   return sys;
end;
#################################################################
# TransformVecToMat ( vecs, d ) . . converts rows of ‘vecs’ into
#                                   dxd matrices
# INPUT - vecs: list containing vectors of length d^2
#       - d: integer
# OUTPUT - M: list of dxd matrices
TransformVecToMat := function( vecs, d )
   local M, k, i, c, m;
   M := [];
   m := Length( vecs );
   for k in [ 1 .. m ] do
      M[k] := [];
      c := 1;
      for i in [ 1 .. d ] do
         M[k][i] := vecs[k]{[c .. c+d-1]};
         c := c + d;
      od;
   od;
   return M ;
end;
#################################################################
# TransformMatToVec( M, d ) . . . converts dxd matrices in M into
#                                 vectors of length d^2
# INPUT - M: list of dxd matrices
#       - d: integer
```

```
# OUTPUT - vecs: a list of vectors of length d^2
TransformMatToVec := function( M, d )
   local i, j, m, vecs, v;
   vecs := [];
   m := Length( M );
   for i in [ 1 .. m ] do
      v := [];
      for j in [ 1 .. d ] do
         Append( v, M[i][j] );
      od;
      Add( vecs, v );
   od;
   return vecs;
end;
#################################################################
# IntersectionMat( M1, M2 ) . . . . determines a basis for the
#                      intersection of the spaces with generating
#                      sets M1 and M2
# NOTE: Taken from the GAP function SumIntersectionMat
IntersectionMat := function( M1, M2 )
   local n, mat, zero, v, heads, i, int;
   if Length( M1 ) = 0 then
      return [ ];
   elif Length( M2 ) = 0 then
      return [ ];
   elif Length( M1[1] ) <> Length( M2[1] ) then
      Error( "dimensions of matrices are not compatible" );
   elif 0 * M1[1][1] <> 0 * M2[1][1] then
      Error( "fields of matrices are not compatible" );
   fi;
   n := Length( M1[1] );
   zero := 0 * M1[1];
   mat := [];
   for v in M1 do
      v := ShallowCopy( v );
      Append( v, v );
      Add( mat, v );
   od;
```

```
   for v in M2 do
      v := ShallowCopy( v );
      Append( v, zero );
      Add( mat, v );
   od;
   mat := SemiEchelonMat( mat );
   heads := mat.heads;
   mat := mat.vectors;
   int := [];
   for i in [ n + 1 .. Length( heads ) ] do
      if IsBound( heads[i] ) then
         Add( int, mat[ heads[i] ]{[ n + 1 .. 2 * n ]} );
      fi;
   od;
   return int;
end;
#################################################################
# BlockInfo( dims, d )
# INPUT - dims: list of dimensions of blocks
#       - d: dimension of matrices
# OUTPUT - init: list of integers s.t. i-th block starts at
#                position ( init[i]+1, init[i]+1 )
#        - blocks: list of integers containing the positions
#                  of the block entries in vector of length d^2
BlockInfo := function( dims, d )
   local b, i, j, blocks, start, init;
   # determine positions in row vector of block entries
   b := Length( dims );  # number of blocks
   blocks := [];         # positions of block entries in vector
   init := [ 0 ];        # i-th block starts at position init[i]+1
   start := 0;
   for i in [ 1 .. b ] do
      for j in [ 1 .. dims[i] ] do
         Append( blocks, [ start+1 .. start+dims[i] ] );
         start := start + d;
      od;
      start := start + dims[i];
      if i > 1 then
```

```
         init[i] := init[i-1] + dims[i-1];
      fi;
   od;
   return [ init, blocks ];
end;
#################################################################
# TestStab( M, slinst, F, d ) . . . tests if all subspaces with
#                                  bases in ‘slist’ are stabilised
#                                  by the matrices in ‘M’
# INPUT - M: list of dxd matrices
#       - slist: list of bases for subspaces of F^d
#       - F: field
#       - d: dimension of matrices
TestStab := function( M, slist, F, d )
   local i, j, k, V, W, vec, s, si, m;
   V := F^d;
   s := Length( slist );
   m := Length( M );
   for i in [ 2 .. m ] do
      W := Subspace( V, slist[i] );
      for j in [ 1 .. m ] do
         si := Length( slist[i] );
         for k in [ 1 .. si ] do
            vec := slist[i][k] * M[j];
            if not ( vec in W ) then
               Error( "subspace is not stabilised\n" );
            fi;
         od;
      od;
   od;
   return true;
end;
#################################################################
# OrderGL( n, q ) . . . . determines order of group GL(n,q)
#                       |GL(n,q)|=(q^n-1)(q^n-q)...(q^n-q^(n-1))
OrderGL := function( n, q )
   local factor, i, order;
   if n = 0 then
```

```
      return 1;
   fi;
   order := 1;
   factor := q^n;
   for i in [ 0 .. n-1 ] do
      order := order * ( factor - q^i );
   od;
   return order;
end;
#################################################################
# MatrixBlock( mat, e )
# INPUT - mat: mxm matrix over F
#       - e: positive divisor of m
# OUTPUT - B: first exe block of 'mat'
MatrixBlock := function( mat, e )
   local B, i;
   B := List( [ 1 .. e ], i -> [] );
   for i in [ 1 .. e ] do
      B[i] := ShallowCopy( mat[i]{ [ 1 .. e ] } );
   od;
   return B;
end;
#################################################################
# SmallOverLargerField( block, m, F )
# INPUT - block: exe matrix over F
#       - m: positive multiple of b
#       - F: field
# OUTPUT - gens: list of mxm matrices that generate the group
#                GL(m/e,K) where K is an extension of F
SmallOverLargerField := function( block, m, F )
   local m1block, zblock, e, q, id, gens, mat, i, j;
   m1block := - block;
   zblock := 0 * block;
   e := Length( block );
   q := QuoInt( m, e );
   id := IdentityMat( m, F );
   gens := [];
   mat := Copy( id );
```

```
for i in [ 1 .. e ] do
   mat[i]{[1..e]} := ShallowCopy( block[i] );
od;
Add( gens, mat );
if e = 1 or e = m then
   return gens;
fi;
if F = GF(2) then
   mat := Copy( id );
   for i in [ 1 .. e ] do
      mat[i]{[1..e]} := ShallowCopy( zblock[i] );
      mat[i]{[ (q-1)*e+1 .. m ]} := ShallowCopy( block[i] );
   od;
   for i in [ 2 .. q ] do
      for j in [ 1 .. e ] do
         mat[ (i-1)*e+j ]{[ (i-2)*e+1 .. (i-2)*e+e ]}
                                 := ShallowCopy( block[j] );
         mat[ (i-1)*e+j ]{[ (i-1)*e+1 .. (i-1)*e+e ]}
                                := ShallowCopy( zblock[j] );
      od;
   od;
   AddSet( gens, mat );
   mat := Copy( id );
   for i in [ 1 .. e ] do
      mat[i]{[ e+1 .. 2*e ]} := ShallowCopy( block[i] );
   od;
   AddSet( gens, mat );
else
   mat := Copy( id );
   for i in [ 1 .. e ] do
      mat[i]{[1..e]} := ShallowCopy( m1block[i] );
      mat[i]{[ (q-1)*e+1 .. m ]} := ShallowCopy( block[i] );
   od;
   for i in [ 2 .. q ] do
      for j in [ 1 .. e ] do
         mat[ (i-1)*e+j ]{[ (i-2)*e+1 .. (i-2)*e+e ]}
                               := ShallowCopy( m1block[j] );
         mat[ (i-1)*e+j ]{[ (i-1)*e+1 .. (i-1)*e+e ]}
```

```
                                    := ShallowCopy( zblock[j] );
         od;
      od;
      AddSet( gens, mat );
   fi;
   return gens;
end;
#################################################################
# ConstructBlockGenerators( M )
# INPUT - M: irreducible but not absolutely irreducible compos.
#            factor of G-module
# OUTPUT - gens: list of generators for GL( m, K )
ConstructBlockGenerators := function( M )
   local CS, e, J, B, block, gens, inv, i, D, k,
         fac, prim, size, m, j, bK;
   FieldGenCentMat( M );
   prim := M.centMat;    # primitive element
   M :=GModule( [ prim ] );
   CS := PlainCompositionSeriesAMod( M );
   # assure that all composition factors have same dimension
   D := [];
   for fac in CS[2] do
      AddSet( D, fac.dimension );
   od;
   if Length( D ) <> 1 then
      Error( "all compos. factors must have same dimension" );
   fi;
   e := CS[2][1].dimension;
   m := QuoInt( M.dimension, e );
   # determine a basis for field extension K over field F
   bK := [ prim ];
   for i in [ 2 .. e ] do
      bK[i] := bK[i-1]^M.field.size;
   od;
   # determine basis over which 'prim' acts as scalar matrix
   B := [];
   for i in [ 1 .. m ] do
      k := i * e;
```

```
      for j in [ 1 .. e ] do
         Add( B, CS[3][k] * bK[j] );
      od;
   od;
   inv := B^-1;
   # change basis to get scalar matrix over K
   J := B * prim * inv;
   block := MatrixBlock( J, e );
   size := OrderGL( m, M.field.size^e );
   gens := SmallOverLargerField( block, M.dimension, M.field );
   # change basis back to original block form
   for i in [ 1 .. Length( gens ) ] do
      gens[i] := inv * gens[i] * B;
   od;
   return [ gens, size ];
end;
#################################################################
# GLGenerators( n, F )
# INPUT - n: dimension of block
#       - F: field
# OUTPUT - gens: list of nxn matrices that generate GL(n,F)
GLGenerators := function( n, F )
   local id, gens, mat, i;
   id := IdentityMat( n, F );
   gens := [];
   mat := Copy( id );
   mat[1][1] := F.root;
   Add( gens, mat );
   if n = 1 then
      return gens;
   fi;
   if F = GF(2) then
      mat := Copy( id );
      mat[1][1] := F.zero;
      mat[1][n] := F.one;
      for i in [ 2 .. n ] do
         mat[i][i-1] := F.one;
         mat[i][i] := F.zero;
```

```
      od;
      Add( gens, mat );
      mat := Copy( id );
      mat[1][2] := F.one;
      Add( gens, mat );
   else
      mat := Copy( id );
      mat[1][1] := -F.one;
      mat[1][n] := F.one;
      for i in [ 2 .. n ] do
         mat[i][i-1] := -F.one;
         mat[i][i] := F.zero;
      od;
      Add( gens, mat );
   fi;
   return gens;
end;
#################################################################
# BlockGenerators( gens, d, F, r, blocks ) . for each nxn matrix
#                 B in ‘blocks’ constructs a dxd identity matrix,
#                 inserts B in this matrix starting at position
#                 ( r+1, r+1 ) and appends this new matrix to
#                 ‘gens’
# Used in case there is no block isomorphic to B.
# INPUT - gens: list of dxd gen. matrices already determined
#       - d: dimension of matrices
#       - F: field
#       - r: block starts at position ( r+1, r+1 )
#       - blocks: list of generators for GL(n,F)
BlockGenerators := function( gens, d, F, r, blocks )
   local mat, i, j, id, n;
   n := Length( blocks[1][1] );
   id := IdentityMat( d, F );
   for i in [ 1 .. Length( blocks ) ] do
      mat := Copy( id );
      for j in [ 1 .. n ] do
         mat[r+j]{[r+1 .. r+n]} := ShallowCopy( blocks[i][j] );
      od;
```

```
      Add( gens, mat );
   od;
end;
#################################################################
# IsoBlocks( mat, block, n, iso, init ) . . . . determines blocks
#                        that are isomorphic to ‘block’ according
#                        to ‘iso’ and iserts them in ‘mat’ at
#                        positions given by ‘init’
# INPUT - mat: dxd matrix containing one nontrivial block
#       - block: the nontrivial block of ‘mat’ (nxn matrix)
#       - n: dimension of ‘block’
#       - iso: list of positions of isomorphic blocks and
#               the actual isomorphisms
#               [ b_1, b_2, iso_2, b_3, iso_3, ..., b_t, iso_t ]
#               => iso_i^-1 * M_1 * iso_i = M_i
#       - init: i-th block starts at position init[i]+1
# OUTPUT - matrix ‘mat’ with isomorphic blocks according to ‘iso’
IsoBlocks := function( mat, block, n, iso, init )
   local i, j, s, B, c;
   c := Length( iso );
   for i in [ 2, 4 .. c-1 ] do
      B := iso[i+1]^-1 * block * iso[i+1];   # isomorphic block
      s := init[ iso[i] ];          # block starts at position s+1
      for j in [ 1 .. n ] do
         mat[s+j]{[ s+1 .. s+n ]} := ShallowCopy( B[j] );
      od;
   od;
end;
#################################################################
# IsoGenerators( gens, iso, init, d, F, r, blocks ) . determines
#                    generators satisfying isomorphism conditions
#                    given by ‘iso’ and adds them to ‘mats’
# INPUT - gens: list of matrices already determined
#       - iso: [ b_1, b_2, iso_2, b_3, iso_3, ..., b_t, iso_t ]
#               b_i-th block ( i = 2, ..., t ) is isomorphic to
#               b_1-st block via isomorphism iso_i, i.e.,
#               iso_i^-1 * M_1 * iso_i = M_i
#       - init: i-th block starts at pos.( init[i]+1, init[i]+1 )
```

```
#       - d: dimension of matrices
#       - F: field
#       - r: block being dealt with starts at pos. ( r+1, r+1 )
#       - blocks: list of generators for GL(n,F)
IsoGenerators := function( gens, iso, init, d, F, r, blocks )
   local mat, n, i, j, id, n;
   id := IdentityMat( d, F );
   n := Length( blocks[1] );
   for i in [ 1 .. Length( blocks ) ] do
      mat := Copy( id );
      # first block
      for j in [ 1 .. n ] do
         mat[r+j]{[r+1 .. r+n]} := ShallowCopy( blocks[i][j] );
      od;
      # insert isomorphic blocks and append generator to `gens'
      IsoBlocks( mat, blocks[i], n, iso, init );
      Add( gens, mat );
   od;
end;
#################################################################
# GLBlockGenerators( dims, isom, factors, F, d, init )
# INPUT - dims: list containing dimensions of the blocks
#       - isom: isom[i] = [a] => a-th block forms single iso class
#               isom[i] = [ a, b, [iso_b], c, [iso_c], ... ]
#               => i-th block is isomorphic to a-th block and
#                  isomorphism is iso_i, i.e.,
#                  iso_i^-1 * M_a * iso_i = M_i
#       - factors: list of composition factors
#       - F: field
#       - d: dimension of stabilising matrices
#       - init: i-th block starts at position init[i]+1
# OUTPUT - a list `gens' of vectors of length d^2 which as dxd
#          matrices are in block form and generate the general
#          linear groups in the blocks satisfying the
#          isomorphism conditions
GLBlockGenerators := function( dims, isom, factors, F, d, init )
   local li, i, gens, c, n, r, index, blocks, size;
   li := Length( isom );     # number of isomorphism classes
```

```
   gens := [];
   size := 1;
   for i in [ 1 .. li ] do
      c := Length( isom[i] );   # length of i-th isom. info
      n := dims[ isom[i][1] ];  # dimension of block
      r := init[ isom[i][1] ];  # block starts at position r+1
      index := isom[i][1];
      if IsAbsolutelyIrreducibleAMod( factors[index] ) then
         blocks := GLGenerators( n, F );
         size := size * OrderGL( n, F.size );
      else
         blocks := ConstructBlockGenerators( factors[index] );
         size := size * blocks[2];
         blocks := Copy( blocks[1] );
      fi;
      if c = 1 then
         BlockGenerators( gens, d, F, r, blocks );
      else
         IsoGenerators( gens, isom[i], init, d, F, r, blocks );
      fi;
   od;
   gens := TransformMatToVec( gens, d );
   return [ gens, size ];
end;
#################################################################
# BlockPartGenerators( blockSol, sys, blocks, F, d )
# INPUT - blockSol: list of vectors which as dxd matrices gen.
#                   the linear groups in the blocks satisfying
#                   isomorphism conditions
#       - sys: list of vectors representing the system of linear
#              eqns whose solution is the non-p-part (in block
#              form) of the algebra normalising the lattice
#       - blocks: list of positions in a vector of length d^2 of
#                 the block entries in the corresp. dxd matrix
#       - F: field
#       - d: dimension of the parent vector space
# OUTPUT - a list `blockPart' containing dxd matrices generating
#          the non p-part of the subgroup of GL(d,F) normalising
```

```
#          the lattice
BlockPartGenerators := function( blockSol, sys, blocks, F, d )
   local zero, b, newsys, i, c, h, nh, eqn, blockPart, s, v;
   h := d^2;
   nh := h + 1;
   zero := List( [ 1 .. nh ], i -> F.zero );   # zero vector
   blockPart := [];
   for b in blockSol do
      # substitute block entries of generator `b' in the system
      newsys :=  Copy( sys );
      for i in [ 1 .. Length( newsys ) ] do
         newsys[i][nh] := F.zero;
      od;
      for i in blocks do
         eqn := ShallowCopy( zero );
         eqn[i] := F.one;
         eqn[nh] := b[i];
         Add( newsys, eqn );
      od;
      newsys := SemiEchelonMat( newsys ).vectors;
      # determine one sol. for the non-homog. system obtained
      c := Length( newsys );
      if c > h then
         Error( "there is no solution for equations \n" );
      else
         v := List( [ 1 .. c ], i -> newsys[i][nh] );
         newsys := newsys{[1..c]}{[1..h]};
         s := SolutionMat( TransposedMat( newsys ), v );
         if IsList( s ) then
            Add( blockPart, s );
         else
            Error("system is not consistent \n" );
         fi;
      fi;
   od;
   if blockPart <> [] then
      blockPart := TransformVecToMat( blockPart, d );
   fi;
```

```
  return blockPart;
end;
#################################################################
# UnitsGenerators( solution, dims, isom, factors, F, d )
# INPUT - solution: list of solutions for system of linear
#                   equations after changing basis to block form
#       - dims: list containing dimensions of blocks
#       - isom: list containing isomorphism info for blocks
#       - factors: list containing composition factors
#       - F: field
#       - d: dimension of matrices and parent vector space
# OUTPUT - pPart: list of dxd invertible matrices generating the
#                p-part of the stabiliser
#        - blockPart: list of dxd invertible matrices generating
#                    the non-p-part of the stabiliser
#        - size: order of the subgrp of GL(d,F) generated by the
#                matrices in 'pPart' and 'blockPart'
UnitsGenerators := function( solution, factors, dims, isom, F, d )
  local info, sys, zero, newsys, i, j, eqn, pPart,
        lp, blockPart, blockSol, size;
  # get some information on the blocks
  # - init = i-th block starts at row and column init[i]+1
  # - blocks = list of positions in a vector of length d^2
  #            of the block entries in the corresp. dxd matrix
  info := BlockInfo( dims, d );    #   = [ init, blocks ]
  sys := NullspaceMat( TransposedMat( solution ) );
  zero := List( [ 1 .. d^2 ], x -> F.zero );
  # determine p-part
  newsys := Copy( sys );
  for i in info[2] do
     eqn := ShallowCopy( zero );
     eqn[i] := F.one;
     Add( newsys, eqn );
  od;
  pPart := NullspaceMat( TransposedMat( newsys ) );
  pPart := TransformVecToMat( pPart, d );
  lp := Length( pPart );
  # go over to group elements by inserting 1's in the diagonal
```

```
   for i in [ 1 .. lp ] do
      for j in [ 1 .. d ] do
         pPart[i][j][j] := F.one;
      od;
   od;
   size := F.size^lp;
   # determine non-p-part generators as group elements
   blockSol := GLBlockGenerators(dims,isom,factors,F,d,info[1]);
   blockPart := BlockPartGenerators(blockSol[1],sys,info[2],F,d);
   size := size * blockSol[2];
   # check trivial case
   if pPart = [] and blockPart = [] then
      blockPart := [ IdentityMat( d, F ) ];
   fi;
   return [ blockPart, pPart, size ];
end;
#################################################################
# IntersectionOfNormalisers ( S, F )
# INPUT - S: list containing generators for subspaces of
#            V=V(d,F), the full vector space of dimension d over
#            the finite field F
#       - F: field
# OUTPUT - list containing the following elements:
#          - G: group record for the intersection of the
#               normalisers in GL(V) of the subspaces if V with
#               generators in S
#          - stab[1]: generating matrices for block part of G
#          - stab[2]: generating matrices for below-blocks part
#                     of G
IntersectionOfNormalisers := function( S, F )
   local elt, d, Subsp, i, keep, U, J, cs, k, full, size,
         solution, module, syslineqn, stab, G;
   elt := First( S, i -> Length( i ) <> 0 );
                                  # first non-empty elt in ‘S’
   d := Length( elt[1] );     # rank
   Subsp := List( [ 1 .. d - 1 ], i -> [] );
   keep := [];          # positions in ‘S’ of elts to be kept
   k := Length( S );
```

```
# determine echelonised basis for each subspace in ‘S’ and
# eliminate repetitions and trivial subspaces
for i in [ 1 .. k ] do
   CleanUpAndSort( i, Subsp, S, d, F, keep );
od;
S := S{ keep };
k := Length( keep );     # number of subspaces kept in ‘S’
if k = 0 then
   return GeneralLinearGroup( d, F.size );
fi;
# set up system of linear equations to determine algebra
# stabilising every subspace in ‘S’
syslineqn := [];
for U in S do
   # U must be in semi-echelon form otherwise SysLinEqn
   # returns the wrong result
   Append( syslineqn, SysLinEqn( U, F, d ) );
od;
# solve system (get basis for solution space)
solution := NullspaceMat( TransposedMat(syslineqn) );
# check trivial case
if solution = [] then
   return NullMat( d, d, F );
fi;
# go back to dxd matrices
solution := TransformVecToMat( solution, d );
# check if solution really stabilises all subspaces
if TestStabFlag then
   TestStab( solution, S, F, d );
fi;
# get module acted on by solution and corresponding
# composition series with isomorphism info and change of
# basis matrix to reflect composition series
module := GModule( solution, F );
cs := CompositionSeriesAMod(module);
J := cs[4]^-1;  # inverse of change of basis matrix
# get solution in block form
for i in [ 1 .. Length( solution ) ] do
```

```
      solution[i] := cs[4] * solution[i] * J;
   od;
   solution := TransformMatToVec( solution, d );
   # determine units of block and 0-in-blocks part of algebra
   stab := UnitsGenerators( solution, cs[5], cs[2], cs[3], F, d );
               #  = [ blockPart, pPart, size ]
   # go back to standard basis
   for i in [ 1 .. Length( stab[1] ) ] do
      stab[1][i] := J * stab[1][i] * cs[4];
   od;
   for i in [ 1 .. Length( stab[2] ) ] do
      stab[2][i] := J * stab[2][i] * cs[4];
   od;
   # test if pPart and blockPart stabilise original list of
   # subspaces and composition series
   if TestStabFlag then
      TestStab( stab[1], S, F, d );
      TestStab( stab[2], S, F, d );
      TestStab( stab[1], cs[1], F, d );
      TestStab( stab[2], cs[1], F, d );
   fi;
   full := Concatenation( stab[1], stab[2] );
   G := Group( full, IdentityMat( d, F ) );
   if TestSizeFlag then
      size := Size(G);
      if size <> stab[3] then
         Error( "wrong size for normaliser\n" );
      fi;
   fi;
   G.size := stab[3];
   return [ G, stab[1], stab[2] ];
end;
```

A.2 The composition series code

```
if not IsBound( GModule ) then
   RequirePackage( "matrix" );
```

```
fi;
#################################################################
# SubQuotGMod( module, sub ) . . generators of sub- and quotient-
#                          module and original module w.r.t. new
# basis as SubQuotGMod returns an additional component 'newbas',
# the basis corresponding to result[3] in terms of the old basis
SubQuotGMod := function( module, sub )
   local ans, dimension, subdim, leadpos, cfleadpos, w, i, j, k,
   m, ct, g, newg, newgn, smodule, qmodule, nmodule, matrices,
   smatrices, qmatrices, nmatrices, im, newim, F, zero, one;
   ans := [];
   subdim := Length( sub );
   if subdim = 0 then
      return ans;
   fi;
   dimension := DimensionFlag( modudle );
   if subdim = dimension then
      return ans;
   fi;
   matrices := GeneratorsFlag( module );
   F := FieldFlag( module );
   zero := F.zero;
   one := F.one;
   sub := ShallowCopy( sub );
   # As in SpinBasis, leadpos[i] gives the position of first
   # nonzero entry (which will always be 1) of sub[i].
   leadpos := [];
   cfleadpos := [];
   for i in [ 1 .. dimension ] do
      cfleadpos[i] := 0;
   od;
   for i in [ 1 .. subdim ] do
      j := 1;
      while j <= dimension and sub[i][j] = zero do
         j := j + 1;
      od;
      leadpos[i] := j;
      cfleadpos[j] := 1;
```

```
   for k in [ 1 .. i-1 ] do
      if leadpos[k] = j then
         Error( "Subbasis isn't normed.\n" );
      fi;
   od;
od;
# Now add a further dim-subdim vectors to the list sub,
# to comlete a basis.
k := subdim;
for i in [ 1 .. dimension ] do
   if cfleadpos[i] = 0 then
      k := k + 1;
      w := [];
      for m in [ 1 .. dimension ] do
         w[m] := zero;
      od;
      w[i] := one;
      leadpos[k] := i;
      Add( sub, w );
   fi;
od;
# Now work out action of generators on submodule
smatrices := [];
nmatrices := [];
for g in matrices do
   newg := [];
   newgn := [];
   for i in [ 1 .. subdim ] do
      im := sub[i] * g;
      newim := [];
      newimn := [];
      for j in [ 1 .. subdim ] do
         k := im[ leadpos[j] ];
         newim[j] := k;
         newimn[j] := k;
         if k <> zero then
            im := im - k * sub[j];
         fi;
```

```
      od;
      # Check that the vector is now zero. If not, then
      # sub was not the basis of a submodule at all.
      if im <> im * zero then
         return false;
      fi;
      for j in [ subdim + 1 .. dimension ] do
         newimn[j] := zero;
      od;
      Add( newg, newim );
      Add( newgn, newimn );
   od;
   Add( smatrices, newg );
   Add( nmatrices, newgn );
od;
smodule := GModule( smatrices, F );
# Now work out action of generators on quotient module
qmatrices := [];
ct := 0;
for g in matrices do
   ct := ct + 1;
   newg := [];
   newgn := nmatrices[ct];
   for i in [ subdim + 1 .. dimension ] do
      im := sub[i] * g;
      newim := [];
      newimn := [];
      for j in [ 1 .. dimension ] do
         k := im[ leadpos[j] ];
         if j > subdim then
            newim[ j - subdim ] := k;
         fi;
         newimn[j] := k;
         if k <> zero then
            im := im - k * sub[j];
         fi;
      od;
      Add( newg, newim );
```

```
         Add( newgn, newimn );
      od;
      Add( qmatrices, newg );
   od;
   qmodule := GModule( qmatrices, F );
   nmodule := GModule( nmatrices, F );
   ans := [ smodule, qmodule, nmodule, sub ];
   return ans;
end;
#################################################################
# LinearCombinationVecs( v, c )
# INPUT - v: list of 'len' vectors
#       - c: list of 'len' field elements
# OUTPUT - vector c[1]*v[1] + ... + c[len]*v[len]
LinearCombinationVecs := function( v, c )
   local len;
   len := Length( c );
   return Sum( [ 1 .. len ], i -> c[i] * v[i] );
end;
#################################################################
# CheckIsomorphisms( m, factors, isom ) . . checks if the irred.
#                    module 'm' is isomorphic to some module in
#                    'factors'; adds 'm' to 'factors' and the
#                    isomorphism information to 'isom'
CheckIsomorphisms := function( m, factors, isom )
   local notfound, i, phi, len, k;
   notfound := true;
   i := 1;
   len := Length( factors );
   while notfound and i <= len do
      if m.dimension = factors[i].dimension then
         phi := IsomorphismAModule( factors[i], m );
         if IsList( phi ) then
            notfound := false;
         fi;
      fi;
      i := i + 1;
   od;
```

```
   Add( factors, m );
   len := Length( factors );
   if notfound then
      Add( isom, [ len ] );
   else
      i := i - 1;
      k := 1;
      while not( i in isom[k] ) do
         k := k + 1;
      od;
      Append( isom[k], [ len, phi ] );
   fi;
end;
#################################################################
# CompositionSeriesRecursion( m, ser, facs, isom, dims )
# INPUT - m: module
#       - ser: already determined terms of composition series
#       - facs: already determined factors of comp. series
#       - isom: already determined isomorphism information
#       - dims: dimensions of already determined comp. factors
CompositionSeriesRecursion := function( m, ser, facs, isom, dims )
   local s, q, b, elt;
   if IsIrreducible( m ) then
      elt := Concatenation( m.denombasis, List( m.csbasis,
                 i -> LinearCombinationVecs( m.fakbasis, i ) ) );
      elt := SemiEchelonMat( elt ).vectors;
      Add( ser, elt );
      Add( dims, m.dimension );
      CheckIsomorphism( m, facs, isom );
   else
      s := SubQuotBasGMod( m, m.subbasis );
      q := s[2];
      b := s[4];
      s := s[1];
      s.denombasis := m.denombasis;
      s.csbasis := IdentityMat( s.dimension, s.field );
      s.fakbasis := List( b, i ->
                       LinearCombinationVecs( m.fakbasis, i ) );
```

```
      q.denombasis := Concatenation( m.denombasis,
                           s.fakbasis{ [ 1 .. s.dimension ] } );
      q.csbasis := IdentityMat( q.dimension, q.field );
      q.fakbasis := List( b{ [ s.dimension+1 .. Length(b) ] },
                    i -> LinearCombinationVecs( m.fakbasis, i ) );
      CompositionSeriesRecursion( s, ser, facs, isom, dims );
      CompositionSeriesRecursion( q, ser, facs, isom, dims );
   fi;
end;
#################################################################
# CompositionSeriesAMod( m ) . . . determines the composition
#                            series of the module ‘m’, the comp.
#                            factors, the isomorphisms between
#                            factors and the change of basis matrix
CompositionSeriesAMod := function( m )
   local b, s, ser, factors, isom, chbas, i, dims;
   b := IdentityMat( m.dimension, m.field );
   # denombasis: basis of kernel
   m.denombasis := [];
   # csbasis: basis of module
   m.csbasis := b;
   # fakbasis: preimage of basis, w.r.t. which csbasis is given
   m.fakbasis := b;
   ser := [];
   factors := [];
   isom := [];
   CompositionSeriesRecursion( m, ser, factors, isom, dims );
   # determine the change of basis matrix
   chbas := [];
   s := Length( ser );
   if s > 0 then
      ser[s] := b;
      Append( chbas, ser[1] );
      for i in [ 2 .. s ] do
         Append( chbas, SemiEchelonFactorBase(ser[i],ser[i-1]) );
      od;
   fi;
   return [ ser, dims, isom, chbas, factors ];
```

```
end;
#################################################################
# PlainComposSeriesRecursion( m, ser, factors ) . determines the
#                              composition series and composition
#                              factors of the module 'm'
PlainComposSeriesRecursion := function( m, ser, factors )
   local s, q, b, elt;
   if IsIrreducible( m ) then
      elt := Concatenation( m.denombasis, List( m.csbasis,
                i -> LinearCombinationVecs( m.fakbasis, i ) ) );
      elt := SemiEchelonMat( elt ).vectors;
      Add( ser, elt );
      Add( factors, m );
   else
      s := SubQutBasGMod( m, m.subbasis );
      q := s[2];
      b := s[4];
      s := s[1];
      s.denombasis := m.denombasis;
      s.csbasis := IdentityMat( s.dimension, s.field );
      s.fakbasis := List( b, i ->
                      LinearCombinationVecs( m.fakbasis, i ) );
      q.denombasis := Concatenation( m.denombasis,
                          s.fakbasis{ [ 1 .. s.dimension ] } );
      q.csbasis := IdentityMat( q.dimension, q.field );
      q.fakbasis := List( b{ [ s.dimension+1 .. Length(b) ] },
                 i -> LinearCombinationVecs( m.fakbasis, i ) );
      PlainCompositionSeriesRecursion( s, ser, factors );
      PlainCompositionSeriesRecursion( s, ser, factors );
   fi;
end;
#################################################################
# PlainCompositionSeriesAMod( m ) . . determines the composition
#                             series, comp. factors and change of
#                             basis matrix of the module 'm'
PlainCompositionSeriesAMod := function( m )
   local b, ser, factors, chbas, s, i;
   b := IdentityMat( m.dimension, m.field );
```

```
   m.denombasis := [];
   m.csbasis := b;
   m.fakbasis := b;
   ser := [];
   factors := [];
   PlainCompositionSeriesRecursion( m, ser, factors );
   chbas := [];
   s := Length( ser );
   if s > 0 then
      ser[s] := b;
      Append( chbas, ser[1] );
      for i in [ 2 .. s ] do
         Append( chbas, SemiEchelonFactorBase( ser[i], ser[i-1] ) );
      od;
   fi;
   return [ser, factors, chbas ];
end;
```

Appendix B

Canonical form code

```
TestSubspCanForm := true;
#################################################################
# IsUpperUniTriangular ( mat )
# INPUT - mat: matrix
# OUTPUT - the boolean ‘true’ in case the matrix ‘mat’ is upper
#          uni-triangular and ‘false’ otherwise
IsUpperUniTriangular := function( mat )
   local d, F, i, j;
   if mat = [] then
      return false;
   fi;
   d := Length( mat );
   F := Field( mat[1][1] );
   if Length( mat[1] ) <> d then
      return false;
   fi;
   if mat[1][1] <> F.one then
      return false;
   fi;
   for i in [ 2 .. d ] do
      if mat[i][i] <> F.one then
         return false;
      fi;
      for j in [ 1 .. i-1 ] do
         if mat[i][j] <> F.zero then
```

```
         return false;
      fi;
   od;
od;
return true;
end;
##################################################################
# Commutators  (X, h, id )
## INPUT - X: list of upper uni-triangular dxd matrices over F_p
##       - h: element of <X>
##       - id: dxd identity matrix over F_p
## OUTPUT - B: list of all non-trivial commutators in [h,X],
##             [h,X,X], ..., [h,X,...,X]
Commutators := function( X, h, id )
local A, B, lenA, x, y;
   A := [h];
   B := [];
   lenA := 1;
   while A <> [] do
      for x in X do
         y := Comm( A[1], x );
         if y <> id and not (y in A) then
            Add( A, y );
            Add( B, y );
            lenA := lenA + 1;
         fi;
      od;
      A := A{[2..lenA]};
      lenA := lenA - 1;
   od;
   return B;
end;
##################################################################
# pGroupBase ( X )
## INPUT - X: list of upper uni-triangular dxd matrices over F_p
## OUTPUT - base: list of upper uni-triangular dxd matrices over
##                 F_p that form a base for the group <X>
pGroupBase := function( X )
```

```
local base, d, F, row, col, id, Y, found, lenY, i, j, a, b,
      x, y, keep, newY, B, G;
# check trivial case
if X = [] then
   return X;
fi;
# check input and remove identity
d := Length( X[1] );
F := Field( X[1][1] );
id := IdentityMat( d, F );
if Length( X ) = 1 and X[1] = id then
   return [];
fi;
Y := [];
for x in X do
   if x <> id then
      if Length( x ) <> d then
         Error( "dimensions of matrices are not compatible" );
      elif not IsUpperUniTriangular( x ) then
         Error( "matrices are not upper uni-triangular" );
      fi;
      Add( Y, x );
   fi;
od;
# initialise
X := Copy( Y );
base := [];
row := 1;
col := 2;
while Y <> [] do
   found := false;
   lenY := Length( Y );
   newY := lenY;
   keep := [];
   i := 0;
   # look for g in Y with   g[row][col] <> 0
   while i < lenY and not found do
      i := i + 1;
```

```
    a := Y[i][row][col];
    if a <> F.zero then
       found := true;
    else
       Add( keep, i );
    fi;
 od;
 if found then
    Add( base, Y[i] );
    # process y in Y with y[row,col] <> 0
    for j in [ i+1 .. lenY ] do
       b := Y[j][row][col];
       if b <> F.zero then
          Y[j] := Y[j] * Y[i]^(-IntFFE(a/b));
          if Y[j] <> id then
             Add( keep, j );
          fi;
       else
          Add( keep, j );
       fi;
    od;
    # add p-th powers and commutators to Y
    y := Y[i]^F.char;
    if y <> id then
       Add( Y, y );
       newY := newY + 1;
       Add( keep, newY );
    fi;
    B := Commutators( X, Y[i], id );
    if B <> [] then
       Append( Y, B );
       Append( keep, [newY+1..newY+Length(B)] );
    fi;
 Y := Y{ keep };
 # update row, col
 if col < d then
    row := row + 1;
    col := col + 1;
```

```
      else
         col := col - row + 2;
         row := 1;
      fi;
   od;
   if TestBaseFlag then
      G := Group( X, id );
      if Size(G) <> F.char^Length(base) then
         Error( "is not a base\n" );
      fi;
   fi;
   return base;
end;
#################################################################
# SumMat ( M1, M2 )
# INPUT - M1: list of generators for vector space
#       - M2: list of generators for vector space
# OUTPUT - V: list of vectors that form a semi-echelonised
#             basis for < M1 > + < M2 >
SumMat := function ( M1, M2 )
   local V;
   if Length( M1 ) = 0 then
      if Length( M2 ) > 0 then
         return SemiEchelonMat( M2 ).vectors;
      else
         return M2;
      fi;
   elif Length( M2 ) = 0 then
      return SemiEchelonMat( M1 ).vectors;
   elif Length( M1[1] ) <> Length( M2[1] ) then
      Error( "dimensions of matrices are not compatible" );
   elif 0 * M1[1][1] <> 0 * M2[1][1] then
      Error( "fields of matrices are not compatible" );
   fi;
   V := Copy( M1 );
   Append( V, M2 );
   V := SemiEchelonMat( V ).vectors;
   return V;
```

```
end;
#################################################################
# PInvariantFlag( M, d, F )
# INPUT - M: list of matrices that generate a nilpotent algebra
#       - d: dimension of matrices & full vector space
#       - F: field
# OUTPUT - flag: list of vectors e_1, ..., e_d such that
#                0 < <e_1> < <e_1,e_2> < ... < <e_1,...,e_d> = V
#                (V = F^d) is an invariant flag for the vector
#                space V acted on by the matrices in M
PInvariantFlag := function( M, d, F )
   local V, t, i, j, flag, zero, n;
   V := [ IdentityMat( d, F ) ];
   zero := 0 * V[1][1];
   t := Length( M );
   i := 1;
   while Length( V[i] ) > 0 do
      if i > d+1 then
         Error( "M[i] are not nilpotent" );
      fi;
      i := i + 1;
      V[i] := [];
      for j in [ 1 .. t ] do
         V[i] := SumMat( V[i], V[i-1]*M[j] );
      od;
      TriangulizeMat( V[i] );
      if V[i] = [ zero ] then
        V[i] := [];
      fi;
   od;
   flag := [];
   n := Length( V );
   for i in [ 2 .. n ] do
      Append( flag, FullEchelonFactorBase( V[i-1], V[i] ) );
   od;
   return flag;
end;
#################################################################
```

```
# VectorWeight( v, F, g )
# INPUT - v: vector of length d
#       - F: field
#       - g: element of P, the p-group acting on V
# OUTPUT - wt: integer representing the weight of g
#              with respect to v
# Definition: The weight of g with respect to v is
#             wt_v(g) = max{ j | v = vg mod <e_j,...,e_d> }
#                     = depth( v - vg )
VectorWeight := function( v, F, g )
   local w, wt, d;
   d := Length( v );
   w := v - v * g;
   if w = 0 * v then
      wt := d + 1;
   else
      wt := PositionProperty( w, x -> x <> F.zero );
   fi;
   return wt;
end;
#################################################################
# SubspaceDepth( depths, w, U_k )
# INPUT - depths: list containing depths of vectors in U_k
#       - w: vector of length d
#       - U_k: basis { u_{i+1}, ..., u_t } for subspace in
#              canonical form
# OUTPUT - weight: the weight of g with respect to the vectors
#                  { v, u_{i+1}, ..., u_t }
SubspaceDepth := function( depths, w, U_k )
   local F, d, x, w, dw, pos, n;
   F := Field( w[1] );
   d := Length( w );
   n := Length( depths ) - Length( U_k );
   dw := PositionProperty( w, x -> x <> F.zero );
   while dw in depths do
      pos := Position( depths, dw ) - n;
      w := w - w[dw]/U_k[pos][dw] * U_k[pos];
      dw := PositionProperty( w, x -> x <> F.zero );
```

```
   od;
   if IsInt( dw ) then
      return dw;
   else
      return d + 1;
   fi;
end;
#################################################################
# VectorCanonicalForm( X, v, F )
# INPUT - X: list of dxd upper uni-triangular matrices over F
#            that form a base for the p-group < X >
#       - v: vector of length d whose canonical form we are
#            calculating
#       - F: field
# OUTPUT - v: the canonical form of the original vector v
#        - X: list of matrices that form a base for the stabiliser
#             of v in the original < X >
#        - transf: element of < X > that transforms the original
#                  v into its canonical form
VectorCanonicalForm := function( X, v, F )
   local searching, weights, len, min_wt, wt, found, H, d,
        lenH, i, done, transf;
   d := Length( v );
   transf := IdentityMat( d, F );
   searching := true;
   while searching do
      len := Length( X );
      min_wt := d + 1;
      weights := [];
      for i in [ 1 .. len ] do
         wt := VectorWeight( v, F, X[i] );
         min_wt := Minimum( min_wt, wt );
         Add( weights, wt );
      od;
      if min_wt = d + 1 then
         searching := false;
      else
         H := Filtered( [ 1 .. len ], i -> weights[i] = min_wt );
```

```
            # g = X[H[1]]
         lenH := Length( H );
         # determine v = v * g^alpha with new v having
         # coefficient 0 for e_{min_wt}
         found := false;
         while not found do
            v := v * X[H[1]];
            transf := transf * X[H[1]];
            if v[min_wt] = F.zero then
               found := true;
            fi;
         od;
         # for all h in X with wt_v(h) = min_wt determine
         # h = h * g^beta such that wt_v(h) > min_wt
         for i in [ 2 .. lenH ] do
            done := false;
            while not done do
               X[H[i]] := X[H[i]] * X[H[1]];
               wt := VectorWeight( v, F, X[H[i]] );
               if wt <> min_wt then
                  done := true;
               fi;
            od;
         od;
         X := Concatenation( X{[1..H[1]-1]}, X{[H[1]+1..len]} );
      fi;
      if min_wt = d then
         searching := false;
      fi;
   od;
   return [ v, transf, X ];
end;
################################################################
# EchelonisedVector( v, depths, U_k )
# INPUT - v: vector to be echelonised w.r.t. U_k
#       - depths: leading term positions of vectors in U_k
#       - U_k: basis of subspace already in canonical form
# OUTPUT - v: the original vector v echelonised w.r.t. U_k
```

```
EchelonisedVector := function( v, depths, U_k )
   local F, j, i;
   F := Field( v[1] );
   j := 0;
   for i in [ 1 .. Length( depths ) ] do
      if IsBound( depths[i] ) then
         j := j + 1;
         if v[depths[i]] <> F.zero then
            v := v - v[depths[i]] / U_k[j][depths[i]] * U_k[j];
         fi;
      fi;
   od;
   return v;
end;
#################################################################
# NextSubspCanonicalForm( X, U, depths, i, F )
# INPUT - X: list of matrices that form a base for the stabiliser
#            of the subspace < U[i+1],...,U[t] > in P
#       - U: list of vectors that forms a basis for a subspace of
#            F^d with U[i+1], ..., U[t] in canonical form
#       - depths: list having in position j the depth of the
#                 vector U[j], for j = i+1, ..., t
#       - i: position of vector in U whose canonical form is
#            going to be determined
#       - F: field
# OUTPUT - x: dxd matrix from  <X> such that
#             [U[i],...,U[t]] * x = cf( [U[i],...,U[t]] )
#        - U: list of vectors that form a basis for a subspace of
#             F^d such that the restriction to U[i],...,U[t] is
#             the canonical form of the original restricted
#             subspace under the action of P
#        - depths: same as input with depths[i] = depth(cf(U[i]))
#        - X: list of matrices that form a base for the
#             stabiliser of < U[i], ..., U[t] > in P
NextSubspCanonicalForm := function( X, U, depths, i, F )
   local d, v, searching, lenX, min_wt, j, wt, weights, H, lenU,
        lenH, found, done, x, y, count, dv;
   d := Length( U[1] );
```

```
x := IdentityMat( d, F );
lenU := Length( U );
v := ShallowCopy( U[i] );
searching := true;
while searching do
   lenX := Length( X );
   min_wt := d + 1;
   weights := [];
   for j in [ 1 .. lenX ] do
      wt := SubspaceDepth( depths, v-v*X[j], U{[i+1..lenU]});
      min_wt := Minimum( min_wt, wt );
      Add( weights, wt );
   od;
   if min_wt = d + 1 then
      searching := false;
   else
      H := Filtered( [1..lenX], j -> weights[j] = min_wt );
      lenH := Length( H );
      # determine v = v * g^a with new v having coefficient 0
      # for e_{min_wt}
      found := false;
      count := 0;
      v := EchelonisedVector( v, depths, U{[i+1 .. lenU]} );
      if v[min_wt] = F.zero then
         found := true;
      fi;
      while not found and count < F.char do
         v := v * X[H[1]];
         x := x * X[H[1]];
         v := EchelonisedVector( v, depths, U{[i+1..lenU]} );
         count := count + 1;
         if  v[min_wt] = F.zero then
            found := true;
         fi;
      od;
      if count = F.char then
         Error( "should do it <p times");
      fi;
```

```
        # for all h in X with wt(h) = min_wt determine
        # h = h * g^a s.t. wt(h) > min_wt
        for j in [ 2 .. lenH ] do
           done := false;
           while not done do
              X[H[j]] := X[H[j]] * X[H[1]];
              wt := SubspaceDepth( depths, v-v*X[H[j]],
                                          U{[i+1..lenU]} );
              if wt > min_wt then
                 done := true;
              elif wt < min_wt then
                 Error( "weight must not decrease" );
              fi;
           od;
        od;
        # remove g from X
        X := Concatenation( X{[1..H[1]-1]}, X{[H[1]+1..lenX]} );
     fi;
     if min_wt = d then
        searching := false;
     fi;
  od;
  U := U * x;
  dv := PositionProperty( U[i], y -> y <> F.zero );
  return [ x, U, dv, X ];
end;
#################################################################
# SubspaceCanonicalForm( X, U, F )
# INPUT - X: list of dxd matrices that generate p-group P
#       - U: list of vectors that form a basis for the subspace
#            of V whose canonical form under P we are determining
#       - F: field
# OUTPUT - Uflag: canonical form of < U >
#        - transf: matrix from P s.t. U * transf = Uflag
#        - base: list of matrices that form a base for the
#                stabiliser of Uflag in its orbit under P
#        - b: integer such that |P|=p^b
SubspaceCanonicalForm := function( X, U, F )
```

```
local d, Uflag, depths, lenU, cf, transf, i, r, id, flag,
     base, x, tU, V, W, b;
# check trivial case
if X = [] then
   return [ U, IdentityMat( Length(U[1]), F ), X ];
fi;
d := Length( X[1] );
id := IdentityMat( d, F );
flag := PInvariantFlag( List( X, x -> x - id ), d, F );
# put matrices into upper uni-triangular form
if flag <> id then
   for i in [ 1 .. Length( X ) ] do
      X[i] := X[i]^(flag^-1);
   od;
fi;
base := pGroupBase( X );
b := Length( base );
TriangulizeMat(U);
if TestSubspCanForm then
   tU := Copy( U );
fi;
lenU := Length( U );
if lenU = 0 then
   return [ U, id, base, b ];
fi;
cf := VectorCanonicalForm( base, U[lenU], F );
transf := cf[2];
base := cf[3];
U := U * transf;
depths := [];
depths[lenU] := PositionProperty( U[lenU], x -> x <> F.zero );
for i in [ lenU-1, lenU-2 .. 1 ] do
   r := NextSubspCanonicalForm( base, U, depths, i, F );
   transf := transf * r[1];
   U := Copy( r[2] );
   depths[i] := r[3];
   base := Copy( r[4] );
od;
```

```
   if TestSubspCanForm then
      V := VectorSpace( U, F );
      if tU * transf <> U then
         if VectorSpace( tU * transf ) <> V then
            Error( "U * transf <> cf( U )" );
         fi;
      fi;
      for i in [ 1 .. Length( base ) ] do
         if VectorSpace( U * base[i], F ) <> V then
            Error( "base must stabilise cf( U )" );
         fi;
      od;
   fi;
   return [ U, transf, base, b ];
end;
```

Appendix C

Published paper

The paper below was accepted for publication by the journal Experimental Mathematics and is to appear in Volume 8(1999), No 4, pages 395-397.

The tensor product of polynomials

Ruth Schwingel
School of Mathematical Sciences, Queen Mary and Westfield College
University of London - Mile End Road, London E1 4NS, UK
R.Schwingel@qmw.ac.uk

Abstract
Using the Gröbner basis algorithm in MAGMAwe find necessary and sufficient conditions for a polynomial of degree 6 over any field to be the tensor product of two polynomials, one of degree 2 and one of degree 3.

1. Introduction

In order to determine whether or not there exists a tensor decomposition of the natural module for a matrix group G over a field K it proved to be useful to decide whether or not there exists a tensor decomposition of the characteristic polynomial of $g \in G$ [Leedham-Green and O'Brien 1997]. This latter problem was the motivation for the present work.

Let h be a univariate polynomial of degree d over an algebraically closed field K. If $d = m + n$ then clearly h is the product of two polynomials over K of degrees m and n. But if $d = mn$, with $m, n > 1$, then h is the tensor product (as defined below) of two polynomials, one of degree m and the other of degree n, if and only if the coefficients $c_1, \dots, c_d$ of h define an element $(c_1, \dots, c_d)$ in some $(m+n-1)$-dimensional variety $V \subset K^d$. This variety is determined by a prime ideal I_{mn} in the ring $K[c_1, \dots, c_d]$. The ideal I_{22} is easily computed by hand and the ideal I_{32} is just within the range of machine computation.

2. The tensor product

Given two monic polynomials $f(x) = x^m - a_1 x^{m-1} + \cdots + (-1)^m a_m$ with zeros $\alpha_1, \dots, \alpha_m$ and $g(x) = x^n - b_1 x^{n-1} + \cdots + (-1)^n b_n$ with zeros $\beta_1, \dots, \beta_n$ in $K[x]$, the *tensor product* of $f(x)$ and $g(x)$ is the monic polynomial $h(x)$ of degree mn with roots $\alpha_j \beta_k$ for $1 \le j \le m, 1 \le k \le n$; that is,

$$h(x) = x^{mn} - c_1 x^{mn-1} + \cdots + (-1)^{mn} c_{mn},$$

with c_i the i-th elementary symmetric function in $\alpha_j \beta_k$, for $1 \le j \le m$ for $1 \le k \le n$.

Let

$$\begin{aligned} p_i(f) &= \sum_{j=1}^{m} \alpha_j^i \\ p_i(g) &= \sum_{k=1}^{n} \beta_k^i \\ p_i(f \otimes g) &= \sum_{j,k} (\alpha_j \beta_k)^i = (\sum_{j=1}^{m} \alpha_j^i)(\sum_{k=1}^{n} \beta_k^i) = p_i(f) p_i(g) \end{aligned}$$

be the i-th power sums of α_j, β_k and $\alpha_j \beta_k$, $1 \le j \le m$, $1 \le k \le n$, respectively.

We can compute the i-th power sum p_i in terms of $\{e_1, \dots, e_i\}$ by using Newton's Formula [Macdonald 1995, p.23]

$$n e_n = \sum_{r=1}^{n} (-1)^{r-1} p_r e_{n-r},$$

where e_j is the j-th elementary symmetric function. Then by a simple algorithm we can compute the c_i's in terms of $\{a_j : 1 \le j \le m\}$ and $\{b_k : 1 \le k \le n\}$.

The *weight* in the x's of a monomial $x_1^{\varepsilon_1} \cdots x_m^{\varepsilon_m}$ is defined by $w = \sum_{i=1}^m i \cdot \varepsilon_i$. Each c_i is then a homogeneous polynomial of weight i in both the a_j's and the b_k's.

In general, the condition that the polynomial h should have a tensor factorisation with factors of degrees m and n is the condition that the coefficients of h define an element $(c_1, \ldots, c_{mn})$ in the variety $V \subset K^{mn}$ determined by an homogeneous ideal $I_{mn} \subset K[c_1, \ldots, c_{mn}]$. I_{mn} is the kernel of the homomorphism from $K[c_1, \ldots, c_{mn}]$ into $K[a_1, \ldots, a_m, b_1, \ldots, b_n]$ taking each c_i to the corresponding polynomial in the a_j's and b_k's. Being the kernel of an homomorphism into a domain, I_{mn} is a prime ideal, hence the variety V is irreducible.

To determine the dimension of V we consider the factorisation

$$h(x) = f(x) \otimes g(x) = \prod_{j,k} (x - \alpha_j \beta_k)$$

giving the polynomial functions $\varphi_{jk} : K^{m+n} \longrightarrow K$ defined by

$$\varphi_{jk}(\alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_n) = \alpha_j \beta_k.$$

It is easy to see that the $m + n - 1$ elements $\varphi_{11}, \ldots, \varphi_{m1}, \varphi_{12}, \ldots, \varphi_{1n}$ form a maximal set of algebraically independent elements over K, hence the dimension of V is $m + n - 1$. For more details on the theory of varieties see [Cox et al. 1997, Chapters 4, 5, 9].

3. Cases I_{22} and I_{32}

It is easy to prove that I_{22} is a principal ideal with generator of weight 6. The coefficients are

$$\begin{aligned} c_1 &= a_1 b_1 \\ c_2 &= a_2 b_1^2 + a_1^2 b_2 - 2 a_2 b_2 \\ c_3 &= a_1 a_2 b_1 b_2 \\ c_4 &= a_2^2 b_2^2 \end{aligned}$$

so that the generator $c_1^2 c_4 - c_3^2$ can be easily obtained.

The problem of finding a set of generators for I_{32} proved surprisingly harder. This is a classical Gröbner basis problem. Considering the polynomial parametrization

$$c_1 = q_1(a_1, \ldots, a_m, b_1, \ldots, b_n)$$

$$\vdots$$

$$c_d = q_d(a_1, \ldots, a_m, b_1, \ldots, b_n)$$

let I be the ideal

$$I = \langle c_1 - q_1, \ldots, c_d - q_d \rangle \subset K[a_1, \ldots, a_m, b_1, \ldots, b_n, c_1, \ldots, c_d].$$

Then the ideal I_{mn} is the $(m+n)$th elimination ideal $I_{mn} = I \cap K[c_1, \ldots, c_d]$, and the Elimination Theorem [Cox et al. 1997, §5.3, Theorem 1] proves that if B is a Gröbner basis for I with respect to lex order where $a_1 > \ldots > a_m > b_1 > \ldots > b_n > c_1 > \ldots > c_d$ then the set $B_{mn} = B \cap K[c_1, \ldots, c_d]$ is a Gröbner basis for I_{mn}.

We were unable to get the calculation to complete on any Gröbner basis package. Clearly I_{mn} is defined over $\mathbb{Q}$ (equivalently over $\mathbb{Z}$). Working over $GF(2)$ without using Gröbner techniques it was possible, using Magma[Bosma and Cannon 1993], to find homogeneous elements of I_{32} that we believed to form a generating set. The conjecture was later confirmed when Allan Steel showed us how to carry out the complete calculation using the Gröbner basis in Magma, working over $\mathbb{Q}$. This was done by defining the polynomial ring $P = \mathbb{Q}[a_1, a_2, a_3, b_1, b_2, c_1, \ldots, c_6]$ with elimination order [Cox et al. 1997, p.72], then defining the ideal $I = \langle c_1 - q_1, \ldots, c_6 - q_6 \rangle$ in P and determining its Gröbner basis B. A Gröbner basis D for the elimination ideal I_{32} is obtained by taking the images of the basis elements $b \in B$ under the homomorphism $\psi : P \longrightarrow K[c_1, \ldots, c_6]$ defined by $\psi(a_j) = \psi(b_k) = 0$, and $\psi(c_i) = c_i$. Eliminating redundancies in D a minimal generating set for I_{32} is obtained. The conclusion is that a minimal generating set for I_{32} contains 16 homogeneous polynomials of weights 19 to 30, each being the sum of at least 28 monomials.

It is hoped that new development of MagmaGröbner basis code will enable us to compute a free homogeneous resolution of the subring M of $K[a_1, a_2, a_3, b_1, b_2]$ generated by the images of $c_1, \ldots, c_6$. Preliminary calculations suggest a resolution of length five

$$0 \longrightarrow F_5 \longrightarrow F_4 \longrightarrow F_3 \longrightarrow F_2 \longrightarrow F_1 \longrightarrow F_0 \longrightarrow M \longrightarrow 0,$$

where the F_i are free modules over $K[c_1, \ldots, c_6]$ as follows: F_0 of rank 1 with a generator of weight 0, $F_1 = I_{32}$, F_2 generated by 34 polynomials of weights 24 to 35, F_3 by 29 polynomials of weights 28 to 38, F_4 by 12 polynomials of weights 33 to 40 and F_5 by 2 polynomials of weights 39 and 41.

The CPU time required for the calculation of the generators for I_{32} using MAGMA Version 2.3-1 on a Pentium II PC was 21 minutes. The polynomials are available from ftp://ftp.maths.qmw.ac. uk/pub/crlg/poly33.

We have been unable to produce any reasonable bound to the number of generators of I_{mn}, or to obtain any information about the weights of the elements of a minimal generating set, except for I_{22} and I_{32}, and have no theoretical explanation for the results obtained in these two particular cases. In particular it would be interesting to have some insight into the cohomological dimension of M.

References

Bosma, W., Cannon, J. J. (1993), *Handbook of* MAGMA *Functions.* Sydney: School of Mathematics and Statistics, University of Sydney.

Cox, D., Little, J., O'Shea, D. (1997), *Ideals, Varieties and Algorithms: An introduction to computational algebraic geometry and commutative algebra.* Springer, New York, 2nd edition.

Leedham-Green, C.R., O'Brien, E.A. (1997), *Recognising tensor products of matrix groups.* Journal of Algebra and Comptation (7) **5**, 541-559.

Macdonald, I.G. (1995), *Symmetric functions and Hall polynomials.* Oxford Mathematical Monographs, Clarendon Press, Oxford, 2nd edition.

Bibliography

[1] W. Bosma and J. Cannon. *Magma handbook*. Sydney, 1993.

[2] R. M. Bryant and L. G. Kovács. Lie representations and groups of prime power order. *Journal of the London Mathematical Society*, 17(2):415–421, 1978.

[3] G. Butler. *Fundamental Algorithms for Permutation Groups*. Springer Verlag, 1991.

[4] C. W. Curtis and I. Reiner. *Representation Theory of Finite Groups and Associative Algebras*. Interscience Publishers, New York, 1962.

[5] D. F. Holt and S. Rees. Testing modules for irreducibility. *Journal of the Australian Mathematical Society (Series A)*, 57:1–16, 1994.

[6] B. Huppert. *Endliche Gruppen I*. Springer Verlag, Berlin, 1967.

[7] N. Jacobson. *Structure of Rings*, volume XXXVII of *AMS Colloquium Publications*. American Mathematical Society, 1964.

[8] R. Laue, J. Neubüser, and U. Schoenwaelder. Algorithms for finite soluble groups and the SOGOS system. In M. Atkinson, editor, *Computational Group Theory*, pages 105–135, London, 1984. Academic Press.

[9] M. F. Newman. Determination of groups of prime-power order. In *Group Theory (Canberra, 1975)*, pages 73–84, Berlin, 1977. Springer Verlag.

[10] E. A. O'Brien. The p-group generation algorithm. *Journal of Symbolic Computation*, 9:677–698, 1990.

[11] E. A. O'Brien. Isomorphism testing for p-groups. *Journal of Symbolic Computation*, 17:133–147, 1994.

[12] E. A. O'Brien. Computing automorphism groups of p-groups. In W. Bosma and A. van der Poorten, editors, *Computational Algebra and Number Theory (Sydney, 1992)*, pages 83–90, Dordrecht, 1995. Kluwer Academic Publishers.

[13] R. Parker. The computer calculation of modular characters. In M. Atkinson, editor, *Computational Group Theory*, pages 267–274, London, 1984. Academic Press.

[14] M. Schönert et al. *GAP - Groups, Algorithms and Programming*. Lehrstuhl D für Mathematik, Rheinisch Westfälishe Technische Hochschule, Aachen, Germany, fifth edition, 1995.

[15] M. Suzuki. *Group Theory I*. Springer Verlag, New York, 1982.

[16] D. E. Taylor. Pairs of generators for matrix groups I. The Cayley Bulletin no 3, Department of Mathematics, University of Sydney, Sydney, October 1987. Pages 76–85.