The Implementation of Various Algorithms for Permutation Groups in the Computer Algebra System: AXIOM.

N.J.Doye

Submitted for the degree of M.Sc.
in Symbolic Computation at
The University of Bath
1993

Copyright

Signature of Author ....

N.J.Doye

Acknowledgements

The author gratefully acknowledges the help and assistance of many, many people during the writing of this report. I would most like to thank Geoff Smith for supervising the work and for answering my various questions. Thanks must also go to Volkmar Felsch (Aachen) for his help with extending the subgroup lattice algorithm. I would like to thank Mike Dewar for helping me with the AXIOM programming language and compiler. Thanks must also go to the following people for their intellectual and/or spiritual guidance: Christian Eisele, Jet Kang, Bill Naylor, Paul Moseley, Dave Chapman, Mark Owen, Simon Scott, Angela Cobban, John ffitch, Geoff Doye (my father), and most importantly, my fiancée, Deborah Wicks.

Contents

Chapter 1

Introduction.

1.1 Computer Algebra Systems and Group Theory.

Since 1953 there has been an ever growing interest in how to solve various algebraic problems using a computer. In the main, this work has been in the areas of real and complex polynomials, other "standard" functions (eg. sin, log) over $\mathbb{R}$ and $\mathbb{C}$, and matrix manipulation. This has lead to the development of computer algebra systems such as MACSYMA, Maple, and REDUCE. However, there has also been a significant amount of research into computational methods for modern algebra, most of which has been done into group theory.

In 1958, J. Neubüser (then at Kiel) started the first project to create a number of programs for group theory, including the "cyclic extension method" (cf. Section 5.2) for finding the lattice of subgroups of a given permutation group. In 1969, Neubüser moved to Aachen, and since then, has continued working in the field. These many years of experience culminating in 1988 with the official release of the specialist computer algebra system: GAP (an acronym for Groups, Algorithms and Programming).

GAP is designed in a similar way to Maple, in that it has a small (and hence fast) kernel written in C++, (Maple's is written in C,) with an interpreted library of routines written in GAP's own language. The GAP language has been designed to be compilable, but as yet, no compiler exists. This, however, does not stop GAP being very fast indeed. Also, the size of the kernel, and the choice of implementation language, mean that GAP can run on many machines, even as small as an Atari ST or Commodore Amiga.

In 1965, J. Cannon commenced work on a system of programs (written in assembler) for calculating all or part of the subgroup structure of a given group. Then, in 1969-70 Cannon developed a prototype of a more general purpose group theory system, written in BCPL (the fore-runner of C). This was followed in 1971-72 by a collaboration with the Aachen team, which resulted in GROUP: a large collection of routines to deal with group theoretical problems, with a simple command interface.

GROUP really showed the way that dedicated computer algebra systems for group theory should go. This was due to the fact that to gain the most benefit from all the various routines written, a simple command interface was not enough, and that a full scale algebraic programming language was needed. To this end, development of the algebraic language Cayley commenced in 1975, and was released in 1982.

Cayley version 3.8 is a fast system, dealing in many areas of modern algebra, whose only disadvantage is its old fashioned user interface, which is much the same as REDUCE's. (Considering the high popularity of both Cayley and REDUCE, this is obviously cannot be that much of a disadvantage.) Cayley version 4 is released this year, and is renamed Magma.

One area where the Cayley and GAP, REDUCE and Maple analogy breaks down, however, is in the way that the systems are distributed. Here, GAP is more akin to REDUCE, being distributed with full sources for free, whereas Cayley and Maple are not. Also, Cayley is supplied as a "black box system", but GAP (like REDUCE and Maple,) allows the user to have full access to the inner workings of the system.

1.2 AXIOM.

AXIOM started life in the mid-1970's as an IBM research project called Scratchpad, but unlike Cayley and GAP, is a general purpose computer algebra system. Another major difference between AXIOM and the group theory systems is that AXIOM is not built on C or C++, but like REDUCE, AXIOM is built on a dialect of LISP. It has all the capabilities of traditional computer algebra systems such as REDUCE and Maple, but is far more easily extended. Indeed, in the user's manual, [2], AXIOM is referred to as a "scientific computation system", as not only can it handle symbolic computation, (the realm of the traditional computer algebra system,) various types of numerical computation, (also included in most modern computer algebra systems,) but also has very sophisticated two-dimensional and three dimensional drawing packages.

This, however, is *not* what makes AXIOM different from most computer algebra systems. It is the programming language which makes AXIOM so versatile. In AXIOM, strict type-checking is performed to make sure that operations are well-defined. Every *object* is an element of a unique *domain* or *type*, (e.g. 23 is an element of the domain `Integer`,) and there exist coercion functions between many of the types.

Types are also objects in AXIOM, and hence have their own types, which are called *categories*. Categories ensure the type-correctness (e.g. the definition of matrices states `Matrix(R: Ring)`, meaning that matrices take one argument, (the type of the entries in the matrix,) which must form a ring. Thus we may form matrices of integers, but not matrices of hash tables).

Of course, categories are objects, too, and thus have a type. This type is the distinguished symbol `Category`.

In virtually every computer algebra system, the data types defined when the system is first compiled are the only data types available to the user. In REDUCE (versions 3.3 and 3.4), the addition of a new data type is possible, but the new domain must be either a ring or a field (so that one may form polynomials over this new domain).

What makes AXIOM so different from other computer algebra systems is it's extensibility. The user may define their own types, domains and categories. The user may also write their own polymorphic algorithms, either interactively or in a *package*. All packages, domains and categories are compiled for speed. For more detail see [2], [6] and [7].

1.3 Magma.

Magma is to be released in November 1993, and like its predecessor, Cayley, will deal with many areas of modern algebra. Like Cayley, Magma is implemented in C, but like AXIOM, is based on the principles of category theory and universal algebra. Thus the user will be able to implement polymorphic algorithms.

User defined domains are expected to be available in 1994. At the time of writing, it was unclear to the author whether the equivalent of an AXIOM package will be a feature of the system.

1.4 Permutations and Permutation Groups in AXIOM.

To create a permutation in AXIOM, it is first necessary to create a list, or list of lists, and then coerce it to be a permutation. For example:

(1) ->list1 := [1,2,3,4]

(1) [1, 2, 3, 4]

Type: List PositiveInteger

(1) ->perm1 := cycle list1

(1) (1 2 3 4)

Type: Permutation PositiveInteger

(1) ->perm2 := cycle [1,2]

(1) (1 2)

Type: Permutation PositiveInteger

(1) ->perm3 := cycles([list1,[5,6]])

(1) (5 6)(1 2 3 4)

Type: Permutation PositiveInteger

To create a permutation group in AXIOM, one must coerce a list of permutations, (all of the same type, of course,) which will be the generators of the group. For example:

(1) –>s4 := permutationGroup([perm1,perm2])

(1) $< (1\ 2\ 3\ 4), (1\ 2) >$

Type: PermutationGroup PositiveInteger

For more information on how to deal with permutations and permutation groups in AXIOM, see [2] and the on-line help system, HyperDoc. One important fact which should be noted about the way AXIOM deals with permutations, is that the permutations are considered to be maps acting from the *left*.

Chapter 2

Permutation Groups.

2.1 Initial Definitions.

Definition 2.1 *Let* Ω *be any set, then we define* $\mathrm{Sym}(\Omega)$*, the* symmetric group *on* Ω *to be set of all bijections from* Ω *to* Ω*. That is to say,*

$$\mathrm{Sym}(\Omega) := \{\alpha | \alpha : \Omega \to \Omega \text{ is bijective}\}$$

For $n \in \mathsf{N}$*, we also write* S_n *to mean any group isomorphic to* $\mathrm{Sym}(\Omega_n)$*, where* $|\Omega_n| = n$*. (Usually,* $\Omega_n = \{1, \ldots, n\} \subset \mathsf{Z}$*.)*

The fact that $\mathrm{Sym}(\Omega)$ forms a group is trivial, but a proof can be found on pp.66-68 of [4]. It is more usual to refer to the bijections from Ω to Ω as *permutations.*

Definition 2.2 *A group* G*, is called a* permutation group *iff* $\exists$ *a set* Ω *such that* $G \leq \mathrm{Sym}(\Omega)$*.*

From hereon, we shall concern ourselves only with the case of finite permutation groups. This does mean that, for G a finite permutation group, Ω need be finite in the above definition (2.2), (for example, S_4 is a subgroup of $\mathrm{Sym}(\mathsf{Z})$,) but clearly, there must exist a finite subset of Ω, Φ say, such that $G \leq \mathrm{Sym}(\Phi)$.

Definition 2.3 *For any permutation group* G*,* $\exists S \subseteq G$ *such that* $\forall g \in G$*,* g *may be written as a finite product of elements of* S *and their inverses. Any* *such a set S is said to* generate G*, and is called a* set of generators *for G. We also write $G = \langle S \rangle$.*

For example, if $S = \{(1\ 2), (1\ 2\ 3\ 4)\}$ then S generates S_4.

2.2 Conjugacy Classes.

Definition 2.4 *For $g, h \in G$, a group, h* conjugated by g*, written h^g is defined to be $g^{-1}hg$. Similarly, for $S \subseteq G$, S* conjugated by g*, written S^g is defined to be $g^{-1}Sg$, that is $\{h^g | h \in S\}$.*

A useful lemma for the subgroup lattice algorithm (specifically, see section 5.5) is the following:

Lemma 2.5 *For H a subgroup of a group G, if $H = \langle S \rangle$, then for $g \in G$, $H^g \leq G$ and furthermore $H^g = \langle S^g \rangle$.*

Proof: Suppose $k_1, k_2 \in H^g$, then $\exists h_1, h_2 \in H$ such that $h_1^g = k_1$, $h_2^g = k_2$. $H \leq G \Rightarrow h_1 h_2^{-1} \in H \Rightarrow (h_1 h_2^{-1})^g \in H^g$. But, $(h_1 h_2^{-1})^g = h_1^g (h_2^{-1})^g = h_1^g (h_2^g)^{-1}$, thus $H^g \leq G$.

If $k \in H^g$, then $\exists h \in H$ such that $h^g = k$. Now, $H = \langle S \rangle$, thus h is equal to some finite product of elements of S and their inverses, $w(s_1, \ldots, s_n)$, say. But conjugation by a group element distributes over multiplication in the group, and conjugation by a group element commutes with inversion, thus $k = h^g = (w(s_1, \ldots, s_n))^g = w(s_1^g, \ldots, s_n^g)$ a finite product of elements of S^g and their inverses. □

Definition 2.6 *Let X be a set and let G be a group. A* (right) action of G on *X is a map*

$$\mu : X \times G \longrightarrow X$$

such that both the following two conditions hold:

1. $(x, \mathrm{id}_G)\mu = x, \ \forall x \in X$
2. $(x, g_1 g_2)\mu = ((x, g_1)\mu, g_2)\mu, \ \forall x \in X, \forall g_1, g_2 \in G$

Clearly, if G is a permutation group, (i.e. $\exists\Omega$ such that $G \leq \mathrm{Sym}(\Omega)$,) then G acts on Ω. Also G acts on G via the map:

$$\begin{aligned} \mu &: G \times G \longrightarrow G \\ \mu &: (g_1, g_2) \longmapsto g_1^{g_2} \end{aligned} \tag{2.1}$$

Also, if we define $\mathcal{L}$ to be $\{H|H \leq G\}$ the following defines a group action:

$$\begin{aligned} \mu &: \mathcal{L} \times G \longrightarrow G \\ \mu &: (H, g) \longmapsto H^g \end{aligned} \tag{2.2}$$

This can be shown to be a group action using lemma 2.5.

Definition 2.7 *Let G act on the set X. For $x, y \in X$, let $x \sim y$ iff $\exists g \in G$ such that $(x, g)\mu = y$. (It is a trivial exercise to show that $\sim$ is an equivalence relation.) Each cell of the partition of X under the equivalence relation $\sim$ is called an* orbit in X under G*, and for $x \in X$, the cell containing x is called the* orbit of x*.*

Under the map defined in equation (2.1), for $g \in G$, the orbit of g is called the conjugacy class of g*. Similarly, under the map defined in equation (2.2), for $H \leq G$, the orbit of H is called the* conjugacy class of H*.*

2.3 Normal Subgroups and the Normalizer of a Subgroup.

Definition 2.8 *Let G be a group and $H \leq G$, if the cardinality of the conjugacy class of H is 1, then H is said to be* normal *in G, and we may write $H \trianglelefteq G$.*

Definition 2.9 *Let G be a group and $H \leq G$, $g \in G$ is said to* normalize H *if $H^g = H$. The* normalizer of H in G*, denoted $N_G(H)$, is defined to be the set: $\{g|g \in G, H^g = H\}$. (Clearly, $N_G(H) \leq G$.)*

Theorem 2.10 *Let G be a group and $H \leq G$, then $H \trianglelefteq N_G(H)$ and $H \trianglelefteq G \Leftrightarrow G = N_G(H)$.*

This is almost tautological, the proof being merely an unravelling of the definitions.

2.4 Invariant Partitions and Primitivity.

Definition 2.11 *Let G be a permutation group acting on the set of points Ω and let $B \subseteq \Omega$. We denote the image of B under the action of g, an element of G, by B^g.*

Definition 2.12 *A partition of Ω into disjoint subsets $B_1, \ldots, B_r$ is said to be* invariant under G *if*

$$\forall g \in G,\ \forall i \in \{1, \ldots, r\},\ \exists j \in \{1, \ldots, r\} \text{ such that} B_i^g = B_j.$$

We often refer to the B_i's as blocks.

Definition 2.13 *A* trivial invariant partition of Ω *is (at least) one of the following:*

- *the* discrete partition *where each block is a singleton;*
- *the* complete partition *where $r = 1$ and $B_1 = \Omega$;*
- *any partition in which each block is a union of orbits of G.*

Definition 2.14 *A group G acting on a set Ω is said to be* primitive *if there does not exist a non-trivial partition of Ω that is invariant under G, else, the group is said to be* imprimitive.

Definition 2.15 *For G a group acting on the set Ω, and $\omega_1, \omega_2 \in \Omega$, the* finest invariant partition with respect to ω_1, ω_2 *is the invariant partition of Ω with ω_1 and ω_2 in the same block, with all blocks of minimal cardinality.*

2.5 The Lattice of Subgroups.

Definition 2.16 *A* lattice *is a partially ordered set $\mathcal{L}$, with ordering $\leq$, where $\forall H, K \in \mathcal{L}$, $\exists P \in \mathcal{L}$ such that $\forall J \in \{J | J \leq H, J \leq K\}, J \leq P$, and similarly $\exists Q \in \mathcal{L}$ such that $\forall J \in \{J | J \geq H, J \geq K\}, J \geq Q$.*

A definition of a partial ordering may be found on p.394 of [4].

Theorem 2.17 *The set of all subgroups $\mathcal{L}$ forms a lattice when ordered by "$\leq$", the subgroup inclusion operator.*

Proof: Clearly, $\leq$ is reflexive, antisymmetric, and transitive, and thus partially orders $\mathcal{L}$. Notice also that for $H, K \in \mathcal{L}$, $H \cap K$ and $\text{join}(H, K)$ are both in $\mathcal{L}$ and satisfy the necessary conditions. $\square$

For a permutation group G, the lattice of subgroups can be considered to be partitioned into distinct *layers*. By taking a prime decomposition of the order of any particular subgroup H of G, and then summing the exponents of all the prime factors, the subgroup may then be placed in the layer identified with the sum so obtained. i.e. If $|H| = \prod_{i=1}^{r} p_i^{t_i}$, (where the p_i are distinct primes, and $\forall i \in \{1, \ldots, r\}$, $t_i \in \mathbb{N}$,) then H is in layer $\sum_{i=1}^{r} t_i$.

For example, the subgroup lattice for S_3 has $\{\text{id}_{S_3}\}$ in layer 0, whereas in layer 1, it has $\langle(1\ 2\ 3)\rangle, \langle(1\ 2)\rangle, \langle(2\ 3)\rangle$ and $\langle(1\ 3)\rangle$, and in layer 2 (the top layer) it has just S_3 itself. This is obviously just a very simple example since each subgroup in the first layer has only one overgroup and one subgroup. Also, since $|S_3| = 6 = 2^1.3^1$ there are only three layers.

2.6 Series and Solubility.

Definition 2.18 *A* subnormal series *of a group G, is a finite sequence $(H_i)_{i=0}^{n}$ of subgroups of G, such that $\forall i \in \{0, \ldots, n-1\}, H_{i+1} \lhd H_i$, with $H_n = \{\text{id}_G\}$ and $H_0 = G$.*

Definition 2.19 *A non-trivial group is said to be* simple *if it contains no proper, non-trivial normal subgroups.*

Definition 2.20 *A subnormal series $(H_i)_{i=0}^{n}$ of a group G is called a* composition series, *if $\forall i \in \{0, \ldots, n-1\}, H_i/H_{i+1}$ is simple.*

Definition 2.21 *A group G is said to be* soluble *if $\exists (H_i)_{i=0}^{n}$, a subnormal series for G, such that $\forall i \in \{0, \ldots, n-1\}, H_i/H_{i+1}$ is Abelian.*

2.7 Perfect Groups.

Definition 2.22 *Let G be a group, and $g, h \in G$, then the* commutator of g and h *is defined to be $g^{-1}h^{-1}gh$ and is denoted $[g, h]$. We define G', the* commutator subgroup *to be $\langle [g, h] | g, h \in G \rangle$*

Theorem 2.23 *Let G be a group, then $G' \trianglelefteq G$ and G/G' is Abelian. Furthermore, for $N \trianglelefteq G$, G/N is Abelian, iff $G' \leq N$.*

For a proof of this theorem, see p.157 of [4]

Definition 2.24 *A non-trivial group G is called* perfect *if $G = G'$.*

Definition 2.25 *The* derived series $(H_i)_{i=0}^n$, *for a group G is defined via the following,*

$$H_0 := G; \; H_{i+1} := H_i{}'$$

and we shall restrict ourselves having $\forall i \in \{0, \ldots, n-1\}, H_i \neq H_{i+1}$, and $H_n = H_n{}'$. The last term in the derived series, H_n, is called the soluble residuum *of the group.*

The following theorem is most important in the soluble residuum method (cf. section 6.3).

Theorem 2.26 *A group, G, is soluble if the soluble residuum of the group is trivial. If, however, the soluble residuum is non-trivial, then it is perfect.*

Proof: Suppose that the soluble residuum of G is trivial, then the derived series $(H_i)_{i=0}^n$ of G has $H_0 = G$ and $H_n = \{\mathrm{id}_G\}$. Then, by theorem 2.23, $\forall i \in \{0, \ldots, n-1\}, H_{i+1} \lhd H_i$, thus the derived series is also a subnormal series. Also, by theorem 2.23, every factor H_i/H_{i+1}, is Abelian.

If the soluble residuum is non-trivial, then by definition, it is perfect. $\square$

Chapter 3

Two Simple Algorithms.

3.1 Introduction.

In this chapter, I shall discuss my implementation of Dimino's algorithm to find all the elements of a finite permutation group, in AXIOM. I shall also discuss my implementation of a function to find the normalizer of a subgroup in a permutation group.

Throughout this chapter, G is a finite permutation group generated by $S = \{s_1, \ldots, s_n\}$.

3.2 Dimino's Algorithm.

Dimino's algorithm is a fairly simple method of calculating every element of a group, given a list of generators for the group. In Dimino's algorithm each element of the group is calculated one after the other, so the data structure containing all the elements of the group would have to be chosen so that increasing the number of entries was efficient. AXIOM is built on AKCL (a dialect of LISP) and thus finds lists very easy to deal with. Thus the obvious choice for the return type of Dimino's algorithm was a list of permutations. The signature being:

```
listOfElements :  PermutationGroup D -> List Permutation D
```

where `D` is any type which has `SetCategory`.

To see how Dimino's algorithm works one does not need any sophisticated mathematics. The following facts suffice:

1. G has an identity element, id_G.
2. $\forall g \in G,\ g = w(s_1, \ldots, s_n)$ a finite product of elements of S.
3. $H \leq G \Rightarrow H$ is itself a group.
4. For $H \leq G,\ \forall g \in G,\ |Hg| = |H|$ and the cosets partition the group.
5. For $H \leq G,\ g, s \in G,\ H(gs) = (Hg)s$.

In the algorithm, a "." refers to an aggregate accessor function, and "·" is the group binary operation. . Comments are preceded by ";;" and are written in italics.

Algorithm 3.1 (Simple Dimino's Algorithm.)

Input: S = $[\mathrm{s}_1, \ldots, \mathrm{s}_n]$, a list of generators for G.
Output: A list of elements of G.

;; *First, check if the list is empty, so that we don't try to extract the first*
;; *element of the empty list.*

If S = [ ], then
  return $[\mathrm{id}_G]$

;; *Throughout the algorithm,* elementList *is the list of elements so far*
;; *calculated, and* order *is the number of elements in the* elementList.
;; *We construct the cyclic subgroup of G generated by* s_1.

order := 1
elementList := $[\mathrm{id}_G]$
g := s_1
While (g $\neq$ 1)
  order := order + 1

```
  elementList := cons(g,elementList)
  g := g·s_1

;; This next loop calculates elementList to be ⟨s_1, ..., s_i⟩ each time
;; it completes.

For i in 2..n
  If s_i ∉ elementList then
    ;; We have ⟨s_1, ..., s_{i−1}⟩ < ⟨s_1, ..., s_i⟩.
    ;; previousOrder denotes the order of ⟨s_1, ..., s_{i−1}⟩, it is needed because
    ;; of fact 4, above.
    previousOrder := order
    ;; We now add on the coset ⟨s_1, ..., s_{i−1}⟩ · s_i.
    order := order + 1
    elementList := cons(s_i,elementList)
    For j in 2..previousOrder
      order := order + 1
      elementList := cons(elementList.previousOrder·s_i,elementList)

    ;; We now look for and calculate other cosets using facts 2 and 5, above.
    ;; At the start of each iteration of the repeat loop, repPos is the
    ;; position of the previous coset representative, when repPos ≤ 0,
    ;; we are finished.

    repPos := previousOrder
    Repeat
      ;; rep is the previous coset representative.
      rep := elementList.repPos
      For j in 1..i
        elt := rep·s_j
        If elt ∉ elementList then
          order := order + 1
          repPos := repPos + 1
          elementList := cons(elt,elementList)
          ;; multPos is the position of each of the elements of
          ;; ⟨s_1, ..., s_{i−1}⟩ except id_G.
          multPos := repPos − 1 + counter * previousOrder
          For k in 2..previousOrder
            order := order + 1
            elementList := cons(elementList.multPos·elt,elementList)
            repPos := repPos +1
      repPos := repPos − previousOrder
      counter := counter + 1
```

```
    if repPos < 1 then
      leave    ;; The repeat loop.
return elementList
```

This algorithm can be compared with Butler's on p.20 of [1], which I also implemented. The sole difference between the two, is that Butler (effectively) uses "append" statements to put new elements onto the end of the list, whereas, in algorithm 3.1, "cons" statements are used. Appending elements to the list is an efficient way of programming in many languages (e.g. C,Pascal), but not in LISP, and hence AXIOM. In AXIOM, append is a more expensive function to call, since really, the call: `append(list,[elt])` is in fact the same as the call: `reverse(cons(elt,reverse(list)))`. Reversing a list is not a quick process.

In actual evaluation time, the "append" version was far slower than the "cons" version, the difference in speeds becoming far more apparent, the greater the order of the group. For most groups, the "append" version also spent far more time garbage collecting (due to the calls to reverse, which creates many lists each time it is called).

For example, $S_6 := \langle (1\ 2\ 3\ 4\ 5\ 6), (1\ 2) \rangle$, the "append" version normally takes 14.0 to 14.2 seconds to calculate the answer, plus another 2.0 to 3.1 seconds garbage collecting. The "cons" version normally takes 9.6 to 10.0 seconds evaluating the answer, and between 0.0 and 1.0 seconds garbage collecting. As stated previously, the difference in times became far more marked for larger groups. (It should be noted that all readings were taken in the same session, and that in other sessions — obviously when the computer was under a smaller load — the performance of both algorithms, was seen to be roughly 14% better.)

3.3 Finding the Normalizer of a subgroup.

In this section I shall describe two different approaches to implement an algorithm to find a the normalizer of a subgroup in a group (cf. definition 2.9). The algorithm relies on the following theorem.

Theorem 3.2 *Let G be a group, $H = \langle t_1, \ldots, t_n \rangle \leq G$. Then g normalizes H iff $\{t_1^g, \ldots, t_n^g\} \subseteq H$.*

Proof: Immediate from lemma 2.5.

The algorithm, taken from p.28 of [1] is as follows:

Algorithm 3.3 (Normalizer of H, a Subgroup of G, a Group.)

Input: S = [$\mathrm{s}_1, \ldots, \mathrm{s}_n$], a list of generators for G;
 T = [$\mathrm{t}_1, \ldots, \mathrm{t}_m$], a list of generators for H.
Output: A list of generators for $N_G(H)$.

;; *First assign some variables some values*

answer := T
Γ := listOfElements(S)
subgroupElts := listOfElements(T)

;; Γ *is the discard set we shall be using. It is initialised as* $G - H$ *in the*
;; *following* for *loop, which also checks that* $H \leq G$

```
For elt in subgroupElts
   oldGammaSize := length(Γ)
   Γ := Γ − {elt}
   If length(Γ) = oldGammaSize, then
      error(H ≰ G)
```

;; *Now, the algorithm follows using theorem 3.2*

```
For g in Γ
   flag := true
   For t in T
      If t^g ∉ H, then
         flag := false
         leave     ;; The For t in T loop.
   If flag, then
      answer := cons(g, answer)
      For i in 1..order(g)
         Γ := Γ − g^i
return answer
```

The difference between the two implementations of the algorithms was that in one version Γ was represented as a bitstring of length equal to the order of G, where bit number n equal to "true" signified that the nth element of the list of elements of G was in Γ. In the other version Γ was represented as a list of permutations.

Surprisingly, it was the more naïve "list" version of the algorithm that proved to be the faster of the two. I would have thought that the low-level machine operations that take place on bitstrings would be quicker, but then again, AXIOM is a Lisp-based system and hence deals with lists very efficiently.

Chapter 4

The Finest Invariant Partition.

Let G be a permutation group acting on a set Ω. The algorithm to find the minimal partition with respect to two points is taken from p.75 of [1]. It depends on the following theorem.

Theorem 4.1 *Let $G = \langle S \rangle$ (where $S = \{s_1, \ldots, s_n\}$,) be a permutation group acting on a set Ω and let $\Omega = \bigcup_{i=1}^{r} B_i$ be a partition of Ω into disjoint subsets. The partition is invariant under G iff*

$$\forall \omega_1, \omega_2 \in \Omega, \text{ such that } \exists j \in \{1, \ldots, r\} \text{ with } \omega_1, \omega_2 \in B_j \Longrightarrow$$

$$\forall s \in S, \exists k \in \{1, \ldots, r\} \text{ such that } \omega_1^s, \omega_2^s \in B_k$$

Proof: Suppose that $\Omega = \bigcup_{i=1}^{r} B_i$ is an invariant partition under G. Suppose $\omega_1, \omega_2 \in B_j$ a block, then for any $g \in G$, $\omega_1^g, \omega_2^g \in B_j^g$, which is also a block, by definition 2.12. Thus taking g to be a generator of G, we see that we have proved the theorem one way round.

Suppose now that $\Omega = \bigcup_{i=1}^{r} B_i$ is a partition with the property that for all pairs of points ω_1, ω_2 in the same block B_j, say, that for all $s \in S$ there exists a block B_k, say, with $\omega_1^s, \omega_2^s \in B_k$. Thus the following is true:

$$\forall s \in S, \ \forall B_j \in \{B_1, \ldots, B_m\}, \ \exists B_k \in \{B_1, \ldots, B_m\} \text{ such that } B_j^s \subseteq B_k$$

and hence by symmetry,

$$\forall s \in S,\ \forall B_j \in \{B_1, \dots, B_m\},\ \exists B_k \in \{B_1, \dots, B_m\} \text{ such that } B_j^{s^{\pm 1}} = B_k$$

Thus, since every element $g \in G$ is equal to a finite product of elements of $S^{\pm}$, it is clear that the partition is invariant. □

Thus, if we discover that two points α and β are in the same block, then we must set the images of α and β under each of the generators to be in the same block, and hence their images, and so on and so forth.

In the algorithm, points are considered to be in the block labelled by the "least" element, with respect to some ordering. So the block a point is in, only changes when the representative of that block discovers itself to be in a different block.

So if β is in the block labelled by α, and then α discovers itself to be in the block labelled by $\bar{\alpha} < \alpha$. Then once the images of $\bar{\alpha}$ and α under the group generators have been reconciled, and we know that the images of α and β under the generators must be equal, we may alter β's images to be the same as α's. Thus the only thing that needs to be remembered each time the algorithm loops, is which (ex-)block representatives, have ceased to be block representatives.

One minor difficulty which arose during the implementation of the algorithm, was that, in AXIOM, the set Ω may be infinite (for example, `Z`). This would have made implementing the algorithm as it stood impossible. Thus, it was decided that the algorithm would be implemented on the *support* of G.

AXIOM contains a built in function to find the support of a permutation group, which returns a `Set`. `Set`s in AXIOM are unwieldy objects as they cannot easily be iterated over (without coercion), thus it was necessary to coerce this to another type. Since the size of the support is fixed, the ability to add or remove members was not needed, whereas fast access to individual elements, was considered important. Thus the support of G was coerced to being a `Vector`. Any of the data structures which did require their lengths to be changed, were implemented as lists.

The algorithm, as implemented, is given below. It does not depend on the existence of an ordering on Ω, but instead, one point is considered to be "less than" another if it's position in the support vector is less than the other's.

Algorithm 4.2 (Minimal partition with respect to two points.)

Input: $\mathrm{S} = [\mathrm{s}_1, \dots, \mathrm{s}_n]$, a list of generators for G.
ω_1, ω_2 two different points in the support of G

Output: The finest invariant partition of the support of G.

```
;; First get the support of G
;; The nth element of partitionVector stores the
;; "least" point known to be in the same block as the nth element of
;; the supportVector at any given time.

supportVector := support(G)
partitionVector := supportVector

;; Now find the position of ω1, ω2 in supportVector.

position1 := position(ω1, supportVector)
position2 := position(ω2, supportVector)

;; Place them in the same block.
;; criticalList stores which block representatives, are no longer
;; block representatives.

If position1 < position2 then
  partitionVector.position2 := ω1
  criticalList := [ω2]
Else
  partitionVector.position1 := ω2
  criticalList := [ω1]

;; The main loop of the algorithm.
;; criticalImage is the new representative point of the block containing
;; criticalPoint.

While criticalList ≠ []
  criticalPoint := criticalList.1
  criticalList := tail(criticalList)
  criticalImage := partitionVector.position(criticalPoint, supportVector)

  ;; Check the images of criticalPoint and criticalImage
  ;; under the S and reconcile, if necessary.

  For s in S
    position1 := position(criticalImage^s, supportVector)
    position2 := position(criticalPoint^s, supportVector)
    image1 := partitionVector.min(position1, position2)
    image2 := partitionVector.max(position1, position2)
```

```
    If image1 ≠ image2, then
      For position in 1..length(supportVector) such that
              partitionVector.position = image2
        partitionVector.position := image1
      criticalList := cons(image2, criticalList)

;; Now reorganise the information to gain readable output.

answer := []
blockRepresentativesList := removeDuplicates(partitionVector)
For point in blockRepresentativesList
  sameBlockAsPointList := []
  For position in 1..length(supportVector) such that
          partitionVector.position = point
    sameBlockAsPointList := cons(supportVector.position,
                          sameBlockAsPointList)
  answer := cons(sameBlockAsPointList,answer)
return answer
```

Chapter 5

The Subgroup Lattice of a Soluble Group.

5.1 Introduction.

In this chapter I shall discuss the various problems and decisions that have to be made when implementing the cyclic extension method for determining all the subgroups of a given soluble finite permutation group.

5.2 The Cyclic Extension Method.

The cyclic extension method is the inductive step in the subgroup lattice algorithm. Let G be a group, and $\mathcal{L}$ the subgroup lattice for G. Supposing we have knowledge of all the subgroups that are in layer i, and indeed suppose that $H \leq G$ is a subgroup in that layer. If $K \geq H$ is in layer $i+1$, then the index of H in K must be prime. Conversely, the following is true.

Theorem 5.1 *If $K \ncong C_1$ is a soluble group, then K has a normal subgroup H of prime index in K.*

Proof: The proof of this theorem follows from the fundamental theorem of Abelian groups. □

Suppose $H = \langle S \rangle$. Now since $|K : H|$ is prime, we must only need one more element, g, say, to generate K. What conditions may we place on this g?

1. $g \notin H$, since otherwise $\langle S \cup \{g\} \rangle = H$
2. $g \in N_G(H)$, since otherwise $g \notin N_K(H)$, then since $g \in K$, theorem 2.10 $\Rightarrow H \ntrianglelefteq K$.
3. $\exists p$, a prime, such that $g^p \in H$, else $|K : H|$ would not be prime.

5.3 The Initial Subgroups.

So the cyclic extension method gives our inductive step, in finding all the subgroups of a given soluble group. But where do we start from? We could start with just layer 0 (which contains just the trivial subgroup) and proceed from there, but this is not a very intelligent place to start.

Since the method relies on constructing extensions of our current subgroups, by elements of prime order in G/H, then to construct any cyclic subgroup, C, say, of order p^n, where p is prime, and $n \in \mathbb{N} - \{1\}$, involves the construction of all the subgroups of C of order p up to p^{n-1}. This is clearly a gross waste of effort.

It is far better to find all cyclic subgroups of order p^n (p, n as above,) first, and then apply the cyclic extension method to this initial set of subgroups.

Indeed this method gives us other advantages, in that we will then have a list of all elements of prime power order, before we start the cyclic extension method. In condition 3, above, we required that Hg be of order p in G/H. Using the following theorem, however, we see that, we only need g to be of order p^n, in G.

Theorem 5.2 *Let $H = \langle S \rangle$ be a subgroup in the ith layer of $\mathcal{L}$, and $K = \langle S \cup \{g\} \rangle$ be in the $(i+1)$th layer. Suppose furthermore that $H \trianglelefteq K$, $\exists p$, a prime, such that $g^p \in H$, and that the order of g in G is $p^n r$, where $\gcd(p, r) = 1$. Then*

1. *the order of g^r in G is p^n,*

2. $(g^r)^p \in H$,
3. $\langle S \cup \{g^r\}\rangle = K$.

Proof: Firstly, 1 is trivial. Next, $g^p \in H \Rightarrow (g^p)^r \in H$, but $(g^p)^r = (g^r)^p$, and 2 is proven. Finally, as a consequence of the Euclidean Algorithm (see p.348 of [4],) we have that,

$$\gcd(p, r) = 1 \Rightarrow \exists a, b \in \mathsf{Z} \text{ such that } ap + br = 1$$

and so, $g = g^{ap+br} = (g^p)^a(g^r)^b$. But $g^p \in H \subset K$ and $g^r \in \langle S \cup \{g^r\}\rangle \Rightarrow g \in \langle S \cup \{g^r\}\rangle$. Thus, $K = \langle S \cup \{g\}\rangle \subseteq \langle S \cup \{g^r\}\rangle$, and clearly, $\langle S \cup \{g^r\}\rangle \subseteq \langle S \cup \{g\}\rangle = K$. □

Now suppose that g is an element of prime power order (but not an involution), then there exist other elements in $C = \langle\{g\}\rangle$ that generate C. Let $\hat{g}$ be one such element. Then clearly, for $\langle S\rangle = H \in \mathcal{L}$, we have that $\langle S \cup \{g\}\rangle = \langle S \cup \{\hat{g}\}\rangle$. Thus for the cyclic extension method, we need only consider one generator from each of the cyclic subgroups of prime power order, to extend our current subgroups.

This gives us the following algorithm.

Algorithm 5.3 (Subgroup Lattice Algorithm for Soluble Groups.)

Input: $\mathrm{S} = \{\mathrm{s}_1, \ldots, \mathrm{s}_\mathrm{n}\}$, a list of generators for G.
Output: $\mathcal{L}$, the lattice of subgroups of G.

;; *First, do some initial calculations.*

elementList := listOfElements(G)
placeInLattice($\langle \mathrm{id}_\mathrm{G}\rangle$)
placeInLattice(G)

;; *Now find all the subgroups of prime power order.*
;; Γ *is the set through which to search for the generators of such.*
;; cyclicGenerators *is the set of cyclic subgroup of prime power*
;; *order generator representatives.*

For g in Γ
  If order(g) is a prime power, then ;; *Test done by factoring order.*
    placeInLattice($\langle \mathrm{g}\rangle$)
    cyclicGenerators := cons(g, cyclicGenerators)

```
    For i in 1..order(g)
      If gcd(i-1,order(g)) = 1, then
        remove(g^{i-1}, Γ)

For each layer in ℒ, except the bottom layer and top two layers
  For H in this layer

    ;; Find Γ, the set of all candidate subgroup extenders.

    calculate N_G(H)
    Γ := cyclicGenerators ∩ N_G(H) ∩ G − H
    For K in the next layer
      If H ≤ K, then
        Γ := Γ − K

    ;; Now search through Γ.

    For g in Γ
      If ∃ p, a prime, such that g^p ∈ H, then

        ;; Now check to see if ⟨H ∪ {g}⟩ is a new subgroup.

        flag := true
        For K in the next layer
          If H ⊂ K and g ∈ K, then
            flag := false
            leave  ;; the For K in the next layer loop
        If flag, then
          placeInLattice(⟨H ∪ {g}⟩)
          Γ := Γ − ⟨H ∪ {g}⟩
        Else
          Γ := Γ − {g}
return ℒ
```

5.4 Representation I.

All subsets (e.g. Γ) and subgroups were represented as bitstrings, as this makes membership testing a trivial function, compared with traversing an entire list of permutations to see if one element is in there. Indeed, to test whether the subgroup $\langle H \cup \{g\}\rangle$ is new, all that is needed is to take the bitstring representation of H, set the bit corresponding to g to be "true", (to obtain a bitstring representation for $H \cup g$,) and then for K in the next layer, check whether

$$H \cup g \texttt{ and not(}K\texttt{) = "000...000"}.$$

Now, since each subgroup has a unique representation as a bitstring, instead of each layer in the lattice being a `List` of `Records`, each containing information about a particular subgroup, I decided to represent each layer as an `AssociationList` whose keys were the bitstrings representing the subgroups. This appeared to have little effect on the operating speed of the implementation, but made the coding far more elegant.

For most of the `Records` which appeared in the code, a new AXIOM Domain was implemented. The main effects this had on the implementation were to make the code far more intelligible, and speeded up the compilation time.

5.5 Every Subgroup?

Given two groups, if they are isomorphic, then they have isomorphic subgroup lattices. So, suppose that G is a finite group, with subgroup lattice $\mathcal{L}$, and that $H_1, H_2 \leq G$, with $H_1 \cong H_2$. Then the portions of $\mathcal{L}$ contained below H_1 and H_2 are "the same". Furthermore, if H_1 and H_2 are conjugate subgroups then the portions above H_1 and H_2 are also "the same" (since $G/H_1 \cong G/H_2$ via conjugation).

So, if we have two conjugate subgroups of G, then there is no point in working out the subgroup lattice above and below both, since one is deducible from the other. Testing whether two subgroups are conjugate or not is easy to implement, but computationally, is quite expensive. A more efficient method is the following. We may, upon creation of a new subgroup, create all its conjugates, there and then.

To implement this change, it was decided that the conjugacy classes of the group should be calculated. This is in itself a useful piece of information. Again, internally, each conjugacy class was represented as a bitstring. Now, since conjugacy classes are really the orbits of G under the group action of conjugation by elements of G, an orbit finding algorithm was implemented. The associated Schreier vectors and backwards pointers were also stored as `Vectors` of `Union ( "start" , Permutation S)`s. (That is, a `Vector` whose elements are either `Permutations` or the distinguished symbol, `"start"`.)

The algorithm used is the same as that in [1] for finding all orbits, and is as follows.

Algorithm 5.4 (Conjugacy Classes Algorithm.)

Input: S, a list of generators for a group G;
L, a list of elements for G.
Output: The conjugacy classes, Schreier vector and backwards pointers for G acting on itself by conjugation.

```
;; Initialise the conjugacy classes, Schreier Vectors, and backwards pointers.

conjugacyClasses := []
schreierVector := [id_G, ..., id_G]
backwardsPointers := [id_G, ..., id_G]

;; Go through all the elements of the group, and check to see if they're
;; already in a conjugacy class (an easy test, not written here).

For i in 1..order(G)
  if not(alreadyInOrbit?(i,conjugacyClasses)), then

    ;; Create the entire conjugacy class containing L.i
    ;; Call this new class newClass, and set the Schreier
    ;; vector and backwards pointers vector to indicate that this element
    ;; is the start (or root) of a conjugacy class.

    newClass := "000...000"
    newClass.i := true
    schreierVector.i := "start"
    backwardsPointers.i := "start"

    ;; notDoneYet is a list of all elements, point, say,in the conjugacy
    ;; class which have not been checked to see whether for each
    ;; generator s, that point^s ∈ newClass

    notDoneYet := [pos]
    While notDoneYet ≠ []
      point := notDoneYet.1
      notDoneYet := notDoneYet − point
      For s in S
        conjPosition := position(point^s,L)
```

```
        ;; The following test stops us going into an infinite loop.

        if not(newClass.conjPosition), then
          newClass.conjPosition := true
          schreierVector.conjPosition := s
          backwardsPointers.conjPosition := point
          notDoneYet := cons(conjPosition,notDoneYet)

    ;; Once the while loop is finished notDoneYet is empty and
    ;; the conjugacy class is complete.

    conjugacyClasses := cons(newClass,conjugacyClasses)
return conjugacyClasses, schreierVector, backwardsPointers
```

Once all the conjugacy classes of elements are known, we need to be able to calculate the conjugacy class of a subgroup. To do this the following algorithm was written and implemented.

Algorithm 5.5 (Conjugacy Class of a Subgroup Algorithm.)

Input: T, a list of generators for H, a subgroup of G;
traceList, a list of all elements, $h \in G - id_G$ which map
(under conjugation) some $t \in T$ to any other element
of its conjugacy class;
Output: A list of all subgroups conjugate (but no equal) to H.

```
answer := []

;; For each element, g, of the traceList, we check to see whether
;; T^g generates a new subgroup in the conjugacy class.

For g in traceList

  ;; If we discover that T^g is already contained in one
  ;; of the other subgroups, then we set addFlag to "false".
  ;; conjugateList is T^g.

  addFlag := true
  conjugateList := []
  For t in T
    conjugateList := cons(t^g,conjugateList)

  ;; First check if T^g ⊆ H.
```

```
  For h in conjugateList
    If h ∉ H, then
      leave    ;; the For h in conjugateList loop.
    If h is the last element in conjugateList, then
      addFlag := false

  ;; Then check if T^g is contained in any of the other subgroups
  ;; in the conjugacy class.

  For K in answer
    If addFlag then

      ;; ⟨T^g⟩ could still be a new subgroup.

      For h in conjugateList
        If h ∉ K then
          leave    ;; the For h in conjugateList loop.
        If h is the last element in conjugateList, then
          addFlag := false

    Else    ;; From If addFlag then...
      leave    ;; the For K in answer loop.

  If addFlag, then
    answer := cons(conjugateList,answer)
return answer
```

5.6 Representation II.

Calculating the list of elements for each of the conjugate subgroups would have been a waste of energy, and negated their effect on the efficiency of the implementation. However, this meant that membership testing of an element in a subgroup, and checking whether a new subgroup is new, now had to be done using AXIOM's `member?` function, which is far slower than a mere bit check from a bitstring.

Another side-effect on the implementation, was that it rather called into doubt the choice of an association list for the representation of each layer of the lattice. This was still maintained, nonetheless.

For the entire lattice the following representations were used. The lattice itself was of type

```
PermutationSubgroupLattice S
```

which was internally represented by a

```
Record( elements  : Vector Permutation S,_
        order     : NonNegativeInteger,_
        cc        : List Bits,_
        ccsv      : Vector Union ( "start" , Permutation S ),_
        ccbp      : Vector Union ( "start" , Permutation S ),_
        lattice   : List AssociationList(Bits,_
                                    PermutationSubgroup S))
```

where: `elements` was the vector containing all the elements of the group; `order` was the order of the group; `cc` was the list of conjugacy classes of the group; `ccsv` was the conjugacy class Schreier vector of the group; `ccbp` was the vector of conjugacy class backwards pointers for the group; and `lattice` was the actual subgroup lattice, with each element of the list being a layer.

```
PermutationSubgroup S
```

was internally represented by,

```
Record( generators  : List Permutation S,_
        order       : Integer,_
        conjClass   : List List Permutation S,_
        overgroups  : List PermutationGroupPointer S,_
        subgroups   : List PermutationGroupPointer S)
```

where: `generators` was the list of generators of the subgroup; `order` was the order of the subgroup; `conjClass` was a list of lists of generators of subgroups conjugate (but not equal) to the subgroup; `overgroups` was a list of groups in the next layer up **only** which were overgroups of this particular subgroup; and `subgroups` was a list of groups in the next layer down **only** which were subgroups of this particular subgroup.

```
PermutationGroupPointer S
```

was internally represented by,

```
Record( bitstring   :  Bits,_
        generators  :  List Permutation S)
```

where S is any domain with SetCategory.

Chapter 6

Extending the Subgroup Lattice Algorithm.

6.1 Introduction.

In this chapter I shall discuss how the algorithms from chapter 5 were extended (and implemented) to deal with non-soluble groups.

6.2 The Method of Maximal Subgroups.

This is the first method I attempted, and relies on information from [10], in particular the maximal subgroups of each of the perfect groups. The implementation was never totally completed, as at a very early stage was seen to be very inefficient, due to the combinatorial problems in the algorithm.

The method involved calculating the whole of the lattice of soluble subgroups of the group, and then, if the order of a known perfect group divided the order of the whole group, subgroups which were of the correct order to be maximal subgroups of the perfect group were searched for inside the lattice. If such subgroups were present in the lattice, then (using AXIOM's built in `order` function,) the orders of join of two (or more) of the subgroups were checked to be of the desired order as the perfect subgroup being searched for, and if not, then the join of another pair (or tuple) was chosen. (The algorithm was only ever implemented to cope with the case that the join of *two* subgroups would be all that was needed.)

If all desired combinations failed, then the perfect subgroup in question was not a subgroup of the whole group. If, however, the subgroup had been found, then (if necessary) a similar process would look for extensions of this perfect group in the lattice, except that the join of the perfect group and some other subgroup would be used to search for the extension. This part however was never implemented.

The maximalSubgroupsTable is an association list (whose keys are the orders of the smallest member of each family of finite simple groups, and are the subgroups for which we are searching,) containing in each entry, a list of the orders of two of the maximal subgroups of the perfect group in question,

Algorithm 6.1 (Searching for Perfect Subgroups Algorithm.)

Input: $\mathcal{L}_{\text{soluble}}$, a partial subgroup lattice for the group G, containing all the soluble subgroups of G.
Output: $\mathcal{L}$, the complete lattice of subgroups for G.

```
For i in listOfOrdersOfSmallestGroupsInFamiliesOfFiniteSimpleGroups
  If i|order(G), then
    order1 := maximalSubgroupsTable.i.1
    order2 := maximalSubgroupsTable.i.2
    For H∈ L with order(H) = order1
      For K∈ L with order(K) = order2
        If order(join(H,K)) = i, then
          If newSubgroupCheck(join(H,K)), then
            placeInLattice(join(H,K))

            ;; Now look for extensions.
            ;; Unfortunately, not yet implemented.

return L
```

6.3 The Soluble Residuum Method.

This method rests heavily on data kindly supplied by V.Felsch of Aachen. Some of this information is contained in [5], and the method is described in [8]. The method relies on the fact that all perfect subgroups of a group are contained within the largest perfect subgroup of the group, which is also the last term in the derived series of the group. This subgroup is called the *soluble residuum* of the group.

The derived series of then group is easily calculated using algorithms found in [9], then if the last term is not the trivial group (or the smallest group in a family of finite simple groups), we then search for elements with certain properties, from which certain words can be formed, and these words generate all the perfect subgroups of this group.

Searching for these elements is nowhere near the combinatorial explosion that hampered the method of maximal subgroups.

These are the algorithms that were implemented.

Algorithm 6.2 (Soluble Residuum (Derived Series) Algorithm.)

Input: S, a list of generators for a group G.
Output: R, the soluble residuum of G.

T := S
T$'$:= commutatorSubgroup(T) ;; *see algorithm 6.3.*
While $\langle \mathrm{T} \rangle \neq \langle \mathrm{T}' \rangle$
  placeInLattice($\langle \mathrm{T}' \rangle$) ;; *In the subgroup lattice algorithm.*
  T := T$'$
  T$'$:= commutatorSubgroup(T) ;; *see algorithm 6.3.*
If in the subgroup lattice algorithm, then

  ;; *We must identify the soluble residuum, so as to include all the*
  ;; *other perfect subgroups.*

  identifySolubleResiduum($\langle \mathrm{T}' \rangle$) ;; *see algorithm 6.5.*
return T$'$

In the following algorithm $[g, h]$ means the commutator of g and h, not the list containing g and h.

Algorithm 6.3 (Commutator Subgroup Algorithm.)

Input: S, a list of generators for a group G
Output: G'.

```
T := []
For g in S
   For h in S such that h ≠ g
      If [g, h] ≠ id_G and [g, h] ∉ ⟨T⟩, then
         T := cons([g, h], T)
return normalClosure(⟨T⟩)    ;; see algorithm 6.4.
```

Algorithm 6.4 (Normal Closure Algorithm.)

Input: T, a list of generators for $H \leq G$;
L, a list of elements for H;
conjugacyClasses, the G-conjugacy classes.
Output: The normal closure of H in G.

```
answer := []
For g in T
   For h in conjugacyClass(g)∩H
      answer := cons(h,answer)
return ⟨answer⟩
```

The following algorithm is that used to identify the soluble residuum, and then calculate the perfect subgroups which it contains. perfectSubgroupTable is an association list whose keys are the orders of all perfect groups, (up to some maximum order,) and whose entries are a list of perfect subgroups of that order and some information about them.

The algorithm looks for all elements a and b in the soluble residuum, R, which are of a certain order, and have R-conjugacy classes of a certain size. Then amongst all pairs (a, b) of elements found, it checks whether certain relations, and anti-relations hold. If they all hold for the pair, then certain sets of words in a and b are formed which generate the perfect subgroups of R. All the information on the orders of a and b, their R-conjugacy class sizes, and the relations, anti-relations, generating words are contained in the entries of perfectSubgroupTable.

Algorithm 6.5 (Identifying the Soluble Residuum.)

Input: R, the soluble residuum of a group, G.
Output: None.

```
If order(R) ∉ {1, 60, ...}, then

  ;; R may have some perfect subgroups.
  ;; abFound is used to indicate whether a and b have been found.

  abFound := false

  ;; Now since there may be two non-isomorphic perfect groups of the same
  ;; order,we must traverse the list of all perfect groups of order equal to R's.
  ;; aClassList and bClassListCopy are lists of all elements
  ;; a and b, respectively, in R satisfying both the
  ;; order and R-conjugacy class conditions.

  For perfectSubgroup in perfectSubgroupTable.order(R)
    aClassList := A list of all the R-conjugacy classes of correct length, with
                        elements of the correct order.
    bClassListCopy := A list of all the R-conjugacy classes of correct length,
                        with elements of the correct order.

    ;; We now traverse the lists of R-conjugacy classes.

    While aClassList ≠ [] and not abFound
      aClass := aClassList.1
      aClassList := aClassList−aClass
      bClassList := bClassListCopy
      While bClassList ≠ [] and not abFound
        bClass := bClassList.1
        bClassList := bClassList−bClass

        ;; We now traverse the R-conjugacy classes chosen.

        For a in aClass
          If abFound, then
            leave   ;; The For a in aClass loop.
          For b in bClass
            If abFound, then
              leave   ;; The For b in bClass loop.

            ;; Check to see if all the relations hold.
```

```
            abFound := true
            For relation in theRelations(pefectSubgroup)
              If relation(a,b) ≠ id_G then
                abFound := false
                leave    ;; The For relation... loop.

            ;; Check to see if all the anti-relations (don't) hold.

            If abFound, then
              For antiRelation in theAntiRelations(pefectSubgroup)
                If antiRelation(a,b) = id_G then
                leave    ;; The For antiRelation... loop.
    If abFound, then
      leave    ;; The For pefectSubgroup... loop.

  ;; a,b now found, so create all perfect subgroups.

  For subgroupGeneratingSet in subgroupGeneratingSets(perfectSubgroup)
    T := []
    For word in subgroupGeneratingSet
      T := cons(word(a,b),T)
    placeInLattice(⟨T⟩)
    identifySolubleResiduum(⟨T⟩)    ;; Recursive call.
return void()
```

6.4 Representation.

All the words, relations and anti-relations were stored as anonymous functions, since this is the only way (apart from writing a separate full-blown AXIOM function for each of the words, relations and anti-relations) that AXIOM can perform symbolic computation. The resulting code was rather inelegant.

Internally the table of all perfect groups was declared to be of type,

```
AssociationList(Integer,List PerfectSubgroupInfo S)
```

`PerfectSubgroupInfo S` was internally represented by,

```
Record( name           : String,_
        gen1Order      : Integer,_
        gen1CCS        : Integer,_
        gen2Order      : Integer,_
        gen2CCS        : Integer,_
        relations      : List WORD,_
        antirelations  : List WORD,_
        subgroupGens   : List List WORD)
```

where: `name` was an identifier for the group whilst coding (eg. `"A5"`); `gen1Order` was the required order of a; `gen1CCS` was the required length of the conjugacy class of a; similarly `gen1Order, gen1CCS` and b; `relations` was the list of relations to be satisfied; `antirelations` was the list of anti-relations to be satisfied; and `subgroupGens` was the list of lists of words which generated the perfect subgroups of the soluble residuum.

`WORD` was an abbreviation for the type of the anonymous functions, which was,

```
(Permutation S,Permutation S) -> Permutation S
```

and `S` is any domain with `SetCategory`.

6.5 Testing.

Testing the implementation of the algorithm has been very difficult, due to the length of time it takes to complete. It is known that the algorithm can find A_5 in S_5, but this takes roughly an hour! With half of the code "commented out", the algorithm did manage to find A_6 and two non-conjugate copies of A_5 (each in conjugacy classes of length six,) in S_6.

Unfortunately, AXIOM seems to suffer from a stack overflow, leaving the user back in their shell, if very large examples are attempted. Thus, at the moment, the perfect group table is only implemented to contain A_5, $SL(2,5)$, $PSL(2,7)$, $PSL(2,7)$ and A_6. It is a trivial exercise to enter any further groups, once they are needed.

Chapter 7

Conclusions.

From the times taken to perform the various algorithms, it appears that at present, AXIOM is not really well suited to performing permutation group theoretical tasks. Compare the times taken with Cayley or GAP, and you will see that AXIOM is roughly 10 to 100 times slower. (AXIOM running on an IBM RS6000; Cayley, GAP running on a SUN 4.)

Indeed, Cannon feared that this would be the case in [3] when he wrote of Scratchpad:

> "...there are fears that it [Scratchpad] is still too general to provide the efficiency of execution needed for serious algebraic computation."

However, AXIOM is intended to be a *general purpose* computer algebra system, with the capability to do many different types of mathematics, and in this it succeeds. The fact that AXIOM can handle integration, differentiation, numerical analysis, draw three-dimensional graphs, and perform complex operations on permutation groups at all is a feat in itself.

It is my opinion that the only reason AXIOM is so slow when dealing with permutation groups is the choice of implementation language, AKCL. Cayley (Magma) and GAP are C or C++ based, and thus the designers were far more "free" in their choice of data-structures, and their implementation. In AXIOM, the designers and developers are permanently tied to Lisp's basic construct of the list.

It will be interesting to see how much faster AXIOM code will run once the new compiler is written. It is also intended that the user will be able to compile AXIOM code to "stand alone" C programs, and then on to executables, only marginally slower than if the code had been hand-written in C! Maybe then, AXIOM will be the truly great system it deserves to be.

Appendix A

Functions Implemented in AXIOM.

A.1 Abbreviations used.

In this appendix the following abbreviations are used.

```
PSUBGRP   ==>  PermutationSubgroup
PSPTR     ==>  PermutationGroupPointer
PERFINFO  ==>  PerfectSubgroupInfo
PSLATT    ==>  PermutationGroupSubgroupLattice
ALIST     ==>  AssociationList
PERM      ==>  Permutation
USP       ==>  Union ( "start" , PERM S )
INT       ==>  Integer
NNI       ==>  NonNegativeInteger
WORD      ==>  ((PERM S,PERM S) -> PERM S)
PERMGRP   ==>  PermutationGroup
```

A.2 Functions Exported from Packages.

A.2.1 `Normalizer3Package(S:SetCategory)`

`normalizer: ( PERMGRP S, PERMGRP S ) -> PERMGRP S`

`normalizer(H,G)`, where `H` is a subgroup of `G`, returns the normalizer of `H` in `G`.

`normalizer: (PERMGRP S, List PERM S, List PERM S)-> PERMGRP S`

`normalizer(H,lsH,lsG)`, where `lsH` is a list of elements for `H`, and `lsG` is a list of elements for `G`, some overgroup of `H`, returns the normalizer of `H` in `G`.

`listOfElements : PERMGRP S -> List PERM S`

`listOfElements(G)` returns the list of all elements in the group `G`.

`listOfElements : List PERM S -> List PERM S`

`listOfElements(S)` returns the list of all elements in the group generated by `S`.

A.2.2 `Permutations4Package(S:SetCategory)`

`finestInvariantPartition : ( PERMGRP S, S, S ) -> List List S`

`finestInvariantPartition(G,w1,w2)` returns the finest invariant partition of the support of `G`, with the the two given points in the same block.

A.2.3 `Lattice8Package(S:SetCategory)`

`subgroupLattice: (PERMGRP S) -> PSLATT S`

`subgroupLattice(G)` returns the lattice of subgroups for `G`.

`conjugacyClasses: (PERMGRP S) -> PSLATT S` `conjugacyClasses(G)` currently returns the conjugacy classes of `G` in a `PSLATT S`, with empty lattice part. A more user friendly interface could be written in few lines.

`solubleResiduum: (PERMGRP S) -> PERMGRP S`

`solubleResiduum(G)` returns the soluble residuum of `G`.

`commutatorSubgroup: (PERMGRP S) -> PERMGRP S`

`commutatorSubgroup(G)` returns the commutator subgroup of `G`.

`normalClosure: (PERMGRP S,PERMGRP S) -> PERMGRP S`

`normalClosure(H,G)`, where `H` is a subgroup of `G`, returns the normal closure of `H` in `G`.

A.2.4 `HackedBitsPackage()`

Some of these functions were necessary "hacks" around AXIOM's inoperative code for the Domain `Bits`. Others merely make code look nicer.

`hackAnd : (Bits,Bits) -> Bits`

`hackAnd(a,b)` returns `a and b`.

`hackOr : (Bits,Bits) -> Bits`

`hackOr(a,b)` returns `a or b`.

`hackNot : (Bits) -> Bits`

`hackNot(a)` returns `not(a)`.

`hackXor : (Bits,Bits) -> Bits`

`hackXor(a,b)` returns `a xor b`.

`theBit : (Bits,Integer)-> Boolean`

`theBit(bits,i)` returns `bits.i`.

`setBit : (Bits,Integer,Boolean) -> Void`

`setBit(bits,i,bool)` sets `bits.i := bool`

`setBitTrue : (Bits,Integer)-> Void`

`setBitTrue(bits,i)` sets `bits.i := ''true''`

`setBitFalse : (Bits,Integer)-> Void`

`setBitFalse(bits,i)` sets `bits.i := ''false''`

`isSubset : (Bits,Bits) -> Boolean`

`isSubset(a,b)` returns true if `a and not(b) = ''000...000''`.

`isSubset : (Bits,Bits,NonNegativeInteger)->Boolean`

`isSubset(a,b,n)` returns true if `a and not(b) = ''000...000''`.

A.3 Functions Exported from Domains.

A.3.1 `PermutationGroupSubgroupLattice(S : SetCategory)`

In this section, `REC` is an abbreviation for

```
Record( elements  :  Vector Permutation S,_
        order     :  NonNegativeInteger,_
        cc        :  List Bits,_
        ccsv      :  Vector Union ( "start" , Permutation S ),_
        ccbp      :  Vector Union ( "start" , Permutation S ),_
        lattice   :  List AssociationList(Bits,_
                                        PermutationSubgroup S))
```

`coerce : REC -> $`

`coerce(record)` creates a lattice from a `REC`, `record`.

`new : () -> $`

`new()` creates an empty lattice.

`theElements : $ -> Vector PERM S`

`theElements(subgrpLattice)` returns a vector containing all the elements of the group.

`theOrder : $ -> NonNegativeInteger`

`theOrder(subgrpLattice)` returns the order of the group.

`theLattice : $ -> List ALIST(Bits,PSUBGRP S)`

`theLattice(subgrpLattice)` returns the subgroup lattice of the group.

`theLayer : ($,Integer) -> ALIST(Bits,PSUBGRP S)`

`theLayer(sL,int)` returns the `int`'th layer of the lattice. WARNING: the layers are numbered from 1 upwards, not from 0, upwards.

`theSubgroup : ($,Integer,Bits) -> PSUBGRP S`

`theSubgroup(sL,int,bitstring)` returns the subgroup represented by `bitstring` in the `int`'th layer of the lattice.

`theConjugacyClasses : $ -> List Bits`

`theConjugacyClasses(sL)` returns a list of bitstrings, each of which represents a conjugacy class of the group.

`theConjugacyClass : ($,Integer) -> Bits`

`theConjugacyClass(sL,int)` returns `theConjugacyClasses(sL).int`.

`theCCSchreierVector : $ -> Vector USP`

`theCCSchreierVector(sL)` returns the conjugacy class Schreier vector for group.

`theCCBackwardsPtrs : $ -> Vector USP`

`theCCBackwardsPtrs(sL)` returns the conjugacy class backwards pointers of the group.

`setElements : ($,Vector PERM S) -> Void`

`setElements(sL,vec)` sets the vector of elements of `sL` to be`vec`.

`setOrder : ($,NonNegativeInteger) -> Void`

`setOrder(sL,int)` sets the order component of the lattice `sL` to be `int`.

`setLattice : ($,List ALIST(Bits,PSUBGRP S))-> Void`

`setLattice(sL,lls)` sets the lattice component of `sL` to be `lls`. `setLayer : ($,Integer,ALIST(Bits,PSUBGRP S))-> Void`

`setLayer(sL,int,ls)` sets the `int`'th layer of the lattice to be `ls`.

`setCClasses : ($,List Bits) -> Void`

`setCClasses(sL,ls)` sets the conjugacy classes of `sL` to be `ls`.

`setCCSV : ($,Vector USP) -> Void`

`setCCSV(sL,vec)` sets the conjugacy class Schreier vector of `sL` to be `vec`.

`setCCBP : ($,Vector USP) -> Void`

`setCCBP(sL,vec)` sets the conjugacy class backwards pointers of `sL` to be `vec`.

`addSubgroup : ($,Integer,Bits,PSUBGRP S) -> Void`

`addSubgroup(sL,int,bitstring,subgrp)` conses `subgrp` onto layer `int` of the lattice.

`addLayer : $ -> Void`

`addLayer(sL)` adds a new empty layer to the lattice component of `sL`.

`addSubgroupPtr: ($,Integer,Bits,Bits,List PERM S) -> Void`

`addSubgroupPtr(sL,int1,bits,ptr,gens)` add the subgroup pointer `ptr,gens` to the subgroup `bits` in the `int1`th layer of `sL`.

`addOvergroupPtr: ($,Integer,Bits,Bits,List PERM S)-> Void`

`addOvergroupPtr(sL,int1,bits,ptr,gens)` add the overgroup pointer `ptr,gens` to the subgroup `bits` in the `int1`th layer of `sL`.

`elementListFromBits: ($,Bits) -> List PERM S`

`elementListFromBits(sL,bits)` returns the list of elements of the group for which the component of `bits` is true. Note: `#bits` should equal the order of the group.

`element : ($,Integer) -> PERM S`

`element(sL,int)` returns the `int`'th element of the vector of elements.

`position : ($,PERM S) -> Integer`

`position(sL,elt)` returns the position of `elt` in the vector of elements.

`layerSize : ($,Integer) -> Integer`

`layerSize(sL,int)` returns the number of subgroups in the `int`'th layer.

`coerce : $ -> OutputForm`

`coerce(sL)` coerces `sL` to a printable format.

A.3.2 `PermutationSubgroup(S : SetCategory)`

In this section, `REC` is an abbreviation for

```
Record( generators  :  List Permutation S,_
        order       :  Integer,_
        conjClass   :  List List Permutation S,_
        overgroups  :  List PermutationGroupPointer S,_
        subgroups   :  List PermutationGroupPointer S)
```

and `RECP` is an abbreviation for

```
Record( bitstring   :  Bits,_
        generators  :  List Permutation S)
```

`coerce : REC -> $`

`coerce(record)` creates a permutation subgroup.

`new : () -> $`

`new()` creates an empty permutation subgroup.

`theGenerators : $ -> List PERM S`

`theGenerators(subgrp)` returns the generators of the subgroup `subgrp`.

`theOrder : $ -> Integer`

`theOrder(subgrp)` returns the order of the subgroup `subgrp`.

`theCClassSize : $ -> Integer`

`theCClassSize(subgrp)` returns the size of the Conjugacy Class of `subgrp`.

`theCClassExcGen : $ -> List List PERM S`

`theCClass(subgrp)` returns a transversal of the set of all subgroups conjugate to, but not equal to `subgrp`.

`theCClass : $ -> List List PERM S`

`theCClass(subgrp)` returns a transversal of the set of all subgroups conjugate to `subgrp`.

`theOvergroups : $ -> List PSPTR S`

`theOvergroups(subgrp)` returns a list of the location of all conjugacy class representatives of overgroups of `subgrp` in the next layer up of the lattice.

`theSubgroups : $ -> List PSPTR S`

`theSubgroups(subgrp)` returns a list of the location of all conjugacy class representatives of subgroups of `subgrp` in the next layer down of the lattice.

`setGenerators : ($,List PERM S) -> Void` `setGenerators(subgrp,ls)` sets the generators of the subgroup `subgrp` to be the list of permutations `ls`.

`setOrder : ($,Integer) -> Void`

`setOrder(subgrp,int)` sets the order component of `subgrp` to be `int`.

`setCClassExcGen : ($,List List PERM S) -> Void`

`setCClass(subgrp,lls)` sets the conjugacy class of `subgrp` to be `lls`, and the size of the conjugacy class to be `#(lls)+1`. Note: `lls` should not contain `subgrp` itself.

`setOvergroups : ($,List PSPTR S) -> Void`

`setOvergroups(subgrp,ls)` sets the overgroups of the subgroup `subgrp` to be the list `ls`.

`setSubgroups : ($,List PSPTR S) -> Void`

`setSubgroups(subgrp,ls)` sets the subgroups of the subgroup `subgrp` to be the list `ls`.

`addOvergroup : ($,Bits,List PERM S) -> Void`

`addOvergroup(subgrp,bits,gens)` adds the true generators of an overgroup and a bitstring representation of an isomorphic overgroup in the lattice, to the list of overgroups of this subgroup.

`addSubgroup : ($,Bits,List PERM S) -> Void`

`addSubgroup(subgrp,bits,gens)` adds the true generators of an subgroup and a bitstring representation of an isomorphic subgroup in the lattice, to the list of subgroups of this subgroup.

`coerce : $ -> OutputForm`

`coerce(subgrp)` returns the output form of `subgrp`.

A.3.3 `PermutationGroupPointer(S : SetCategory)`

In this section `REC` is an abbreviation for

```
Record( bitstring   :  Bits,_
        generators  :  List Permutation S)
```

`coerce : REC -> $`

`coerce(record)` creates a new pointer.

`new : () -> $`

`new()` creates a new empty pointer.

`theGenerators : $ -> List PERM S`

`theGenerators(ptr)` returns `ptr.generators`.

`theBits : $ -> Bits`

`theBits(ptr)` returns `ptr.bitstring`

`setGenerators : ($,List PERM S) -> Void`

`setGenerators(ptr,ls)` sets `ptr.generators` to be `ls`.

`setBits : ($,Bits) -> Void`

`setBits(ptr,bits)` sets `ptr.bitstring` to be `bits`.

`coerce : $ -> OutputForm`

`coerce(ptr)` coerces `ptr` to output format.

A.3.4 `PerfectSubgroupInfo(S : SetCategory)`

In this section `REC` is an abbreviation for

```
Record( name          : String,_
        gen1Order     : Integer,_
        gen1CCS       : Integer,_
        gen2Order     : Integer,_
        gen2CCS       : Integer,_
        relations     : List WORD,_
        antirelations : List WORD,_
        subgroupGens  : List List WORD)
```

`coerce : REC -> $`

`coerce(record)` creates a new `PERFINFO S`.

`new : () -> $`

`new()` creates a new empty `PERFINFO S`.

`theName : $ -> String`

`theName(perf)` returns the name of the perfect subgroup.

`genOrders : $ -> List Integer`

`genOrders(perf)` returns a list of the orders of the two elements for which to search.

`genCClassSizes : $ -> List Integer`

`genCClassSizes(perf)` returns a list of the size of the conjugacy classes of the two elements for which to search.

`theRelations : $ -> List((PERM S,PERM S) -> PERM S)`

`theRelations(perf)` returns a list of anonymous functions which should map the elements to `1::PERM S`.

`theAntirelations : $ -> List((PERM S,PERM S) -> PERM S)`

`theAntielations(perf)` returns a list of anonymous functions which should NOT map the elements to `1::PERM S`.

`theSubgroupGens : $ -> List List((PERM S,PERM S) -> PERM S)` `theSubgroupGens(perf)` returns a list of lists of anonymous functions. Each sublist when applied to the elements found will generate new perfect subgroups.

```
coerce :  $ -> OutputForm
```

`coerce(perf)` coerces `perf` to output format.

Bibliography

[1] G. Butler:
Fundamental Algorithms For Permutation Groups.
Lecture Notes in Computer Science, 559, Springer-Verlag, Berlin Heidelberg. 1991.

[2] R. D. Jenks and R. S. Sutor:
Axiom: The Scientific Computation System.
Springer-Verlag, Berlin Heidelberg. 1992.

[3] J. Cannon:
A Language For Group Theory.
School of Mathematics, University of Sydney. 1982.

[4] J. B. Fraleigh:
A First Course in Abstract Algebra (Fourth Edition).
Addison-Wesley, Reading, Massachusetts. 1987.

[5] G. Sandlöbes: *Perfect Groups of order less than* 10^4.
Communications in Algebra, 9, pp. 477-490. 1981.

[6] J. H. Davenport and B. M. Trager:
Scratchpad's View of Algebra I: Basic Commutative Algebra.
Proc. DISCO '90, Lecture Notes in Computer Science, 429, pp. 40-54. Springer-Verlag, Berlin Heidelberg. 1990.

[7] J. H. Davenport, P. Gianni and B. M. Trager:
Scratchpad's View of Algebra II: A Categorical View of Factorization.
Proc. ISSAC 1991, pp. 32-48. ACM, New York. 1991

[8] V. Felsch and G. Sandlöbes:
An Interactive Program for Computing Subgroups.
Proc. LMS Symposium on Computational Group Theory, Durham 1982, Computational Group Theory, Ed. M. Atkinson, pp. 137-143. Academic Press, London. 1984.

[9] G. Butler and J. J. Cannon:
Computing in Permutation and Matrix Groups I: Normal Closure, Commutator Subgroups, Series.
Mathematics of Computation, 39, 160 (Oct. 1982), pp. 663-670. 1982.

[10] J. Conway, R. Curtis, S. Norton, R. Parker, R. Wilson.
Atlas of Finite Groups : maximal subgroups and ordinary characters for simple groups.
Oxford University Press, 1985.