PARTITION BACKTRACK PROGRAMS: USER'S MANUAL

JEFFREY S. LEON

Mathmatics Dept, m/c 249
University of Illinois at Chicago
Box 4348
Chicago, IL 60680

I. INTRODUCTION

This document describes a collection of programs for permutation group computations employing the partition backtrack method, as described in a recent article by the author (Leon, 1991). At present, these programs perform the following computations:

set stabilizers,	set images (see below),
ordered partition stabilizers,	ordered partition images,
intersections,	
centralizers (of elements),	conjugacy (of elements),
centralizers (of subgroups),	
automorphism groups of designs,	isomorphism of designs,
automorphism groups of matrices,	isomorphism of matrices,
monomial automorphism groups of matrices over small fields,	monomial isomorphism of matrices over small fields,
automorphism groups of linear codes,	isomorphism of linear codes.

The term *set image* problem is used here to refer to the following problem: Given a permutation group G and subsets Λ and Φ of the domain, determine if there exists $g \in G$ such that $\Lambda^g = \Phi$. The ordered partition image problem is defined analogously. The term *design* is used here to refer to any collection of points and blocks. An automorphism of a matrix is any permutation of the rows and columns that leaves the matrix invariant; two matrices are isomorphic if one may be transformed to the other by permutation of rows and columns. For monomial automorphism groups and monomial isomorphism, the matrix entries must be taken from a finite field; in addition to permutation of rows and columns, we allow each row and each column to be multiplied by a nonzero field element.

Note that each of the problems in the first column above involves computation of a subgroup; these problems will be referred to as *subgroup computations.* For each problem in the second column, the set of permutations having the desired property either is empty or forms a right coset of an appropriate subgroup; we are seeking one coset representative (if it exists). These problems will be referred to as *coset representative computations.*

Some of the programs described here can be used to compute in groups of relatively high degree, considerably higher than those that can be handled by programs based on conventional algorithms. However, it should be kept mind that the programs are new. All appear to work correctly, but most have not been thoroughly tested, especially on intransitive groups. (The set stabilizer program has been tested the most thoroughly, and in general those for subgroup computations have received more testing than those for coset representative calculations.) The author would appreciate any reports of errors; they may be sent to leon@turing.math.uic.edu.

Work on programs for the following computations is in progress:

unordered partition stabilizers,
normalizers,
unordered partition images,
conjugacy (of subgroups),
coset intersections,

In the course of constructing test cases for the partition backtrack programs and verifying their output, the author has developed several other programs, not based on backtrack search. These programs are described briefly in Section IX. Many of these programs were put together quickly, with a view toward simplicity rather than efficiency and ease of use.

At present, the programs run on the following machines: The Sun/3, the Sun/4, the IBM 3090, and the IBM PC. Two versions are available for the IBM PC: a standard version, which is limited to groups of degree no more than 1000 (roughly) due to the 640K memory limitation, and a 386/486 version using a DOS extender, that can handle larger groups. The source code for all programs is written entirely in C and, with very minor exceptions, conforms to the ANSI standard. The programs should compile, with minimal changes, with any C compiler fully supporting the ANSI standard. The Sun/3 and Sun/4 versions have been compiled with the GNU C compiler, the IBM 3090 version with the Waterloo C compiler, and the IBM/PC versions with the Borland C++ and Zortech C++ compilers (both configured as C compilers).

II. OBJECTS AND FILE FORMATS

At present, the programs compute with objects of seven types: Permutation groups, permutations, point sets, partitions (ordered or unordered), block designs, matrices, and linear codes. Each object used as input by the programs is read from a file. Likewise, each object constructed by the programs is written to a file. All of these files are ordinary text files. The format of the files is designed for compatibility with Cayley (Cannon, 1984); it is that of a Cayley library, with certain restrictions added. Essentially, the restrictions say that the library may contain only statements defining the object, and (at present) that only certain attributes of the object may specified in the library. (Many of the permutation group libraries distributed with Cayley conform to these restrictions.) Thus objects constructed by these programs described here may be read into Cayley for further investigation. Likewise, objects defined in existing Cayley libraries may, in many cases, be used as input to the programs described here. An alternative format, compatible with Gap, may be added at a later date.

The examples which follow illustrate the correct format for these object files. Note that the contents of the files is case–insensitive; upper and lower case letters may be used interchangeably. (However, the names of the files may be case–sensitive, depending on the operating system.) Also, the use of white space (blanks, tabs, newline characters) is optional: Except within integers and identifiers, any number of whitespace characters may occur. Text enclosed by ampersands or quotation marks is treated as a comment.

a) Permutation groups: The format for permutation group files is illustrated by following file, named psp62, which defines $\mathrm{PSp}_6(2)$ as a permutation group on nonzero vectors (degree 63).

```
LIBRARY psp62;
" PSp(6,2) acting on nonzero vectors, degree 63."
psp62: permutation group(63);
psp62.forder: 2^9 * 3^4 * 5 * 7;
psp62.generators:
  a = (1,2)(3,5)(4,7)(8,12)(11,16)(13,19)(17,18)(20,26)(21,28)(23,30)(24,32)
      (25,34)(29,37)(31,40)(33,43)(36,46)(38,41)(39,49)(42,44)(45,52)(48,51)
      (53,58)(57,62)(59,61),
  b = (1,3,6,10,15,22)(2,4,8,13,20,27)(5,9,14,21,29,38)(7,11,17,23,31,41)
      (12,18,24,33,44,34)(16,19,25)(26,35,45,53,32,42)(28,36,47)(30,39,50,
      56,61,58)(37,48)(40,51,46,54,59,62)(49,55,60,63,52,57);
FINISH;
```

The line specifying the factored group order may be omitted; however, since the random Schreier method is currently used to construct a base and strong generating set for the group, there is a possibility (probably small) that the group may be generated incorrectly if this line is removed. When generators are written in cycle format, as above, inclusion of cycles of length one is optional. (For compatibility with Cayley, they should be omitted.) It is also possible to write the generators in image (rather than cycle) format; in this case, the file shown above would become:

```
LIBRARY psp62;
" PSp(6,2) acting on nonzero vectors, degree 63."
psp62: permutation group(63);
psp62.forder: 2^9 * 3^4 * 5 * 7;
psp62.generators:
   a = /2,1,5,7,3,6,4,12,9,10,16,8,19,14,15,11,18,17,13,26,28,22,30,
        32,34,20,27,21,37,23,40,24,43,25,35,46,29,41,49,31,38,44,33,42,
        52,36,47,51,39,50,48,45,58,54,55,56,62,53,61,60,59,57,63/,
   b = /3,4,6,8,9,10,11,13,14,15,17,18,20,21,22,19,23,24,25,27,29,1,
        31,33,16,35,2,36,38,39,41,42,44,12,45,47,48,5,50,51,7,26,43,34,
        53,54,28,37,55,56,46,57,32,59,60,61,49,30,62,63,58,40,52/;
FINISH;
```

Finally, if a base and strong generating set for the group are already known, they may be included in the file. This eliminates the need for the programs to first construct a base and strong generating set for the input group. The file format is then as follows.

```
LIBRARY psp62;
" PSp(6,2) acting on nonzero vectors, degree 63."
psp62: permutation group(63);
psp62.forder: 2^9 * 3^4 * 5 * 7;
psp62.base:  seq(1,3,6,2,4,5);
psp62.strong generators:  [
   x01 = (1,3)(4,27)(5,9)(7,45)(10,48)(11,17)(12,34)(13,20)(14,25)(15,
         39)(16,63)(19,60)(21,55)(22,58)(23,28)(24,37)(31,53)(32,47)(33,
         61)(36,42)(38,49)(44,50)(51,62)(54,59),
   x02 = (2,16)(3,6)(5,55)(8,21)(9,40)(13,51)(14,46)(15,47)(17,44)(19,
         59)(20,29)(22,36)(23,58)(24,61)(25,63)(26,37)(27,60)(31,50)(32,
         48)(33,41)(34,56)(38,57)(43,45)(52,62),
    .
    .

   x12 = (5,51)(7,12)(8,29)(9,62)(10,42)(13,55)(15,23)(18,35)(20,21)
         (22,32)(28,39)(34,45)(36,48)(40,52)(43,56)(47,58)];
FINISH;
```

(The vertical dots indicated that part of the file has been omitted.) When a base and strong generating set are given, inclusion of the factored order of the group is purely optional. At present it is *not* possible to specify both generators and a base and strong generating set for a group. (This obviously undesirable restriction will be removed eventually.)

b) Permutations: The format for permutation files is illustrated by following file, named g, which defines a permutation of g of degree 63 and order 4, which turns out to lie in the group $\mathrm{PSp}_6(2)$ given above.

```
LIBRARY g;
" An element of order 4 in the group psp62 above."
g = (1,40,50,6)(2,58,18,34)(3,8,44,30)(4,10,15,48)(5,11)(7,60,38,32)
    (12,46,22,56)(13,62,20,61)(16,42,36,63)(19,49,47,45)(21,53,31,
    55)(24,33,37,51)(25,28)(26,52)(27,59,39,54)(29,41);
FINISH;
```

As with generators for permutation groups, permutations may be written in image format, rather than cycle format. Note that, when cycle format is used, the file contains no explicit indication of the degree of the permutation. Thus, for example, the permutation g above could be used wherever a permutation of degree 63 (the largest point appearing explicitly) or greater is expected.

Given the files above, the centralizer in $\mathrm{PSp}_6(2)$ of g could be computed by the command

```
cent   psp62   g   C
```

which would save the centralizer (in the format described in part (a) above) in the file C.

c) Point sets: The format for point sets is illustrated by following file, named `lambda`, which defines a subset Λ of $\{1, \ldots, 63\}$.

```
LIBRARY lambda;
" A subset of 1,...,63 of size 31."
lambda = [10,16,44,3,5,33,48,63,56,50,6,52,55,19,34,25,2,35,17,40,21,
          58,49,36,39,12,60,30,15,29,37];
FINISH;
```

Note that there is no explicit indication of the size of the base set. Thus the set `lambda` above could be used wherever a subset of $\{1, \ldots, m\}$ is expected for any m with $m \geq 63$ (the largest point appearing explicitly).

The set stabilizer in $\mathrm{PSp}_6(2)$ of Λ could be computed by the command

```
setstab  psp62  lambda  S
```

which would save the stabilizer in the file `S`.

d) Partitions: The format for partitions (ordered or unordered) is illustrated by following file, named `pi`, which defines a partition Π of $\{1, \ldots, 63\}$.

```
LIBRARY pi;
" An (ordered or unordered) partition of 1,...,63 having four "
" cells of sizes 15, 20, 13, and 15. respectively."
pi = seq([1,34,28,48,37,41,13,54,57,51,4,38,8,46,16],[2,40,21,
          18,6,53,30,56,42,12,3,11,33,15,32,5,60,31,55,63],[7,
          36,25,29,35,9,26,49,14,47,10,24,43],[17,58,52,50,59,
          45,20,61,23,39,44,19,22,62,27]);
FINISH;
```

Note that the individual cells are delimited by square brackets. Note also that the file contains no indication whether the partition is ordered or unordered; rather each program operating on partitions interprets the partition as ordered or unordered, whichever is appropriate for the the program.

The stabilizer in $\mathrm{PSp}_6(2)$ of Π, interpreted as an ordered partition, could be computed by the command

```
parstab  psp62  pi  T
```

which saves the stabilizer in the file `T`.

e) Block designs: Here a block design refers to any collection of subsets of $\{1, \ldots, n\}$; it is even possible to have repeated blocks, i.e., two different blocks containing exactly the same points. (If repeated blocks occur, we require that an automorphism preserve multiplicities.) The file format for block designs is illustrated by following file, named `d17`, which defines a block design D_{17} with 17 points and with 34 blocks, each of size 5.

```
LIBRARY d17;
" The design with 17 points and 34 blocks, each containing "
" 5 points, obtained from the codewords of weight 5 in the "
" quadratic residue code of length 17 and dimension 9."
d17 = seq( 17, 34,
           [3,6,8,15,17], [1,4,7,9,16], [1,4,5,11,17],
           [2,5,8,10,17], [3,4,7,8,14], [1,2,5,6,12],
           [1,4,6,13,15], [1,3,6,9,11], [1,3,10,12,15],
           [3,5,8,11,13], [2,8,9,12,13], [1,7,13,14,17],
           [6,8,11,14,16], [4,5,8,9,15], [2,5,7,14,16],
           [2,3,6,7,13], [3,4,10,16,17], [3,5,12,14,17],
           [1,7,8,11,12], [5,7,10,13,15], [6,12,13,16,17],
           [2,4,7,10,12], [2,9,11,14,17], [2,3,9,15,16],
           [2,4,11,13,16], [6,7,10,11,17], [4,6,9,12,14],
           [5,11,12,15,16], [1,8,10,13,16], [1,2,8,14,15],
           [5,6,9,10,16], [4,10,11,14,15], [7,9,12,15,17],
           [3,9,10,13,14] );
FINISH;
```

Note that the file contains the number of points, followed by the number of blocks, followed by a listing of the blocks. Each block is delimited by square brackets.

The automorphism group of this block design could be computed by the command

```
desauto  d17  A
```

which saves the automorphism group in the file `A`.

f) Matrices: Here a matrix is merely a $k \times n$ array of integers, each in the set $\{0, \ldots, q-1\}$ for some k, n, and q. (At present, q cannot exceed 256; this limit could be raised, at the cost of additional space, by minor changes to the source code.) The file format for matrices is illustrated by following file, named `m17`, which represents incidence matrix M_{17} for the block design D_{17} in part (e) above. (In the incidence matrix, rows correspond to points and columns to blocks.)

```
LIBRARY m17;
" The incidence matrix of d17."
m17 = seq( 2, 17, 34, seq(
       0,1,1,0,0,1,1,1,1,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,
       0,0,0,1,0,1,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0,1,1,1,1,0,0,0,0,1,0,0,0,0,
       1,0,0,0,1,0,0,1,1,1,0,0,0,0,0,1,1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,
       0,1,1,0,1,0,1,0,0,0,0,0,0,1,0,0,1,0,0,0,0,1,0,0,1,0,1,0,0,0,0,1,0,0,
       0,0,1,1,0,1,0,0,0,1,0,0,0,1,1,0,0,1,0,1,0,0,0,0,0,0,0,1,0,0,1,0,0,0,
       1,0,0,0,0,1,1,1,0,0,0,0,1,0,0,1,0,0,0,0,1,0,0,0,0,1,1,0,0,0,1,0,0,0,
       0,1,0,0,1,0,0,0,0,0,0,1,0,0,1,1,0,0,1,1,0,1,0,0,0,1,0,0,0,0,0,0,1,0,
       1,0,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,
       0,1,0,0,0,0,0,1,0,0,1,0,0,1,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,1,0,1,1,
       0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,1,0,1,0,0,0,1,0,0,1,0,1,1,0,1,
       0,0,1,0,0,0,0,1,0,1,0,0,1,0,0,0,0,0,1,0,0,0,1,0,1,1,0,1,0,0,0,1,0,0,
       0,0,0,0,0,1,0,0,1,0,1,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,1,0,
       0,0,0,0,0,0,1,0,0,1,1,1,0,0,0,1,0,0,0,1,1,0,0,0,1,0,0,0,1,0,0,0,0,1,
       0,0,0,0,1,0,0,0,0,0,0,1,1,0,1,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1,0,1,0,1,
       1,0,0,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,1,0,1,1,0,
       0,1,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,1,1,0,0,1,1,0,1,0,0,0,
       1,0,1,1,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0,1,0,1,0,0,1,0,0,0,0,0,0,1,0
      ));
FINISH;
```

Note that the file contains the set size q, followed by the number k of rows, followed by the number n of columns, followed by the entries of the matrix listed in row–major order. Note that matrix entries are not, in general, limited to 0 and 1; if q is the set size, matrix entries may be integers in the range 0 through $q - 1$.

The automorphism group of this matrix could be computed by the command

```
matauto  m17  B
```

which saves the automorphism group in the file `B`.

Matrices may also be specified using an alternate format, which is not compatible with Cayley, but which saves considerable space for large matrices.[†] This alternate format is available only when the set size q is at most 9. Using the alternate format, the matrix M_{17} would be specified by a file as follows:

[†] At time of writing, the alternate format does not work correctly on some machines.

```
m17
2  17  34
0110011110010000001000000000110000
0001010000100011000001111000010000
1000100111000000111000001000000001
0110101000000100100001001010000100
0011010001000110010100000001001000
1000011100001001000010000110001000
0100100000001001100110100010000010
1001100001101100001000000000110000
0100000100100100000000110010001011
0001000010000000100101000100101101
0010000101001000001000101101000100
0000010010100000011011000011000010
0000001001110001000110001000100001
0000100000011010010000100010010101
1000001010000100000100010001010110
0100000000001010100010011001101000
1011000000001000011001010010000010
```

With the alternate format, blanks may occur between matrix entries, but are not required. The first line of the file is reserved for the matrix name; nothing else may be placed on this line.

g) Linear codes: The file format for lines codes is illustrated by following file, named `q17`, which represents the binary (17,9)–quadratic residue code Q_{17} mentioned in part (e) above. This file gives a generator matrix for the code, in the format of part (f) above.

```
LIBRARY q17;
" The (17,9) binary quadratic residue code."
q17 = seq( 2, 9, 17, seq(
           1,1,1,0,1,0,0,0,1,1,0,0,0,1,0,1,1,
           1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,0,1,
           1,1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,0,
           0,1,1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,
           1,0,1,1,1,1,1,0,1,0,0,0,1,1,0,0,0,
           0,1,0,1,1,1,1,1,0,1,0,0,0,1,1,0,0,
           0,0,1,0,1,1,1,1,1,0,1,0,0,0,1,1,0,
           0,0,0,1,0,1,1,1,1,1,0,1,0,0,0,1,1,
           1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
      ));
FINISH;
```

Note that the file contains the size of the field for the code, followed by the dimension of the code, followed by the length of the code, followed by a generator matrix whose entries are listed in row–major order. At present, the field size is restricted to a prime integer (less than 255) or to 4; automorphism group and isomorphism calculations are practical only when the field size is quite small. In the case of a prime, the field is taken as the integers modulo that prime.

The automorphism group of this code could be computed by the command

```
codeauto  q17  v17  H
```

which saves the automorphism group as the group `H`. (Here file `v17` defines a matrix V_{17} which is the transpose of the matrix M_{17} of part (f) above; the role of V_{17} will be explained later.)

As with matrices, an alternate (not Cayley–compatible) format is provided for codes over field of size at most 9.[†] With the alternate format, the file defining the code Q_{17} would be as follows:

```
q17
2  9  17
11101000110001011
11110100011000101
11111010001100010
01111101000110001
10111110100011000
01011111010001100
00101111101000110
00010111110100011
11111111111111111
```

In the examples above, it was assumed that the name of file matched the name of the Cayley library that it contained. Although this is recommended for simplicity, it need not be the case. When the names do not match, an object is specified using the format *fileName*::*libraryName*. For example, if the file `psp62` in part (a) above were renamed `psp` and the file `g` in part (b) were named `pspx4`, but if the contents of both files remained unchanged, the command to compute the centralizer in $\mathrm{PSp}_6(2)$ of g might be

```
cent  psp::psp62  pspx4::g  gCentr::C
```

where now the centralizer is saved in the file `gCentr`, but in a Cayley library named `C`. A path may also be specified, for example,

```
cent  ../groups/psp::psp62  ../groups/pspx4::g  gCentr::C
```

in Unix. (In MS DOS on the IBM PC, the forward slash must be replaced by a backslash. Under CMS on the IBM 370, the file name and file type must be separated by a period, rather than the blank normally used under CMS.) If however, the file name and the library name for input files differ only in that the file name contains path information, the `-p` option (discussed later) may be useful.

In the examples above, not only did the file name and the Cayley library name match, but both matched the actual name for the object appearing in the Cayley library. Actually, the name for the object need not be related to the name of the Cayley library. For example, the following is acceptable.

[†] As with matrices, this alternate format at present fails to work correctly on some machines.

```
LIBRARY psp62;
" PSp(6,2) acting on nonzero vectors, degree 63."
G: permutation group(63);
G.forder: 2^9 * 3^4 * 5 * 7;
G.generators:
  a = (1,2)(3,5)(4,7)(8,12)(11,16)(13,19)(17,18)(20,26)(21,28)(23,30)(24,32)
      (25,34)(29,37)(31,40)(33,43)(36,46)(38,41)(39,49)(42,44)(45,52)(48,51)
      (53,58)(57,62)(59,61),
  b = (1,3,6,10,15,22)(2,4,8,13,20,27)(5,9,14,21,29,38)(7,11,17,23,31,41)
      (12,18,24,33,44,34)(16,19,25)(26,35,45,53,32,42)(28,36,47)(30,39,50,
      56,61,58)(37,48)(40,51,46,54,59,62)(49,55,60,63,52,57);
FINISH;
```

In this situation, the command line must still specify the Cayley library name (and file name, if different), but informative messages printed as the command executes use the object name (`G`, in this case). For objects created by commands, an object name different from the Cayley library name may be specified by means of the `-n` option, discussed later.

III. FIELDS, MONOMIAL PERMUTATIONS, AND MATRICES

This section treats several topics that arise primarily in connection with computations involving combinatorial structures – designs, matrices, and codes.

i) Finite fields: Finite fields arise when computing with codes, or with matrices whose entries belong to the field. At present, only fields GF(q) whose order q is either 4 or a prime integer are supported; moreover, we must have $q \leq 255$. (For automorphism group and isomorphism calculations, time and space considerations generally dictate a practical limit on q that is far lower.) Field elements are numbered $0, 1, \ldots, q-1$. When q is prime, the field is taken as the integers modulo q. When $q = 4$, there is an essentially unique way to number the field elements.

We denote the set of nonzero elements of GF(q) by GF$(q)^{\#}$.

ii) Monomial permutations: Given a fixed field GF(q), a monomial permutation of monomial degree n over GF(q) is essentially a permutation s on GF$(q)^{\#} \times \{1, \ldots, n\}$ which satisfies the following property, henceforth referred to as the *monomial property:*

$$\begin{gathered}(\alpha, i)^s = (\beta, j) \quad \text{implies} \quad (\gamma\alpha, i)^s = (\gamma\beta, j) \\ \text{for all } \alpha, \beta, \gamma \in \mathrm{GF}(q)^{\#} \text{ and } i, j \in \{1, \ldots, n\}.\end{gathered}$$

Note that s is determined completely by its action on the points $(1, i)$, $1 \leq i \leq n$; note also that the actual degree of s is $(q-1)n$.

For purposes of actual computation, however, we want a representation of s as a permutation on $\{1, \ldots, (q-1)n\}$; to obtain this, we number the pair (α, i) by $(q-1)(i-1)+\overline{\alpha}$, where $\overline{\alpha}$ denotes the integer representing α. Then the monomial property becomes

$$\begin{aligned} &((q-1)(i-1)+\overline{\alpha})^s = (q-1)(j-1)+\overline{\beta} \quad \text{implies} \\ &\qquad ((q-1)(i-1)+\overline{\gamma\alpha})^s = (q-1)(j-1)+\overline{\gamma\beta}. \end{aligned}$$

For example, over the field GF(4), the monomial permutation s on $\mathrm{GF}(4) \times \{1,2,3,4\}$ determined by

$$(1,1)^s = (3,2), \quad (1,2)^s = (2,4), \quad (1,3)^s = (1,1), \quad (1,4)^s = (2,3)$$

is represented as a permutation on $\{1, \ldots, 12\}$ as follows:

$$(1,6,10,8,2,4,11,9,3,5,12,7).$$

iii) Permutations acting on matrices: Let $A = (a_{ij})$ be an $r \times c$ matrix with entries from an arbitrary set. A permutation s of degree $r+c$ which fixes $\{1, \ldots, r\}$ setwise induces an action on A as follows: Row i of A is moved to row position i^s $(1 \le i \le r)$ and column j of A is moved to column position $(r+j)^s - r$ $(1 \le j \le c)$. Thus

$$A^s = (b_{ij}), \quad \text{where} \quad b_{ij} = a_{i'j'}, \text{ with } i' = i^{s^{-1}} \text{ and } j' = (r+j)^{s^{-1}} - r.$$

If $A^s = B$, we say that s is an *isomorphism* of A to B. When $A^s = A$, s is called an *automorphism* of A. The group formed by the automorphisms is called the *automorphism group* of A, and denoted AUT(A).

For example, the action of a permutation s of degree 7 on a 3×4 matrix A is illustrated by the following.

$$s = (1,3,2)(4,7,5), \qquad A = \begin{pmatrix} 8 & 0 & 4 & 3 \\ 2 & 9 & 3 & 0 \\ 0 & 1 & 7 & 5 \end{pmatrix}, \qquad A^s = \begin{pmatrix} 9 & 0 & 3 & 2 \\ 1 & 5 & 7 & 0 \\ 0 & 3 & 4 & 8 \end{pmatrix}.$$

iv) Monomial permutations acting on matrices: Now let $A = (a_{ij})$ be an $r \times c$ matrix with entries from a field GF(q). A monomial permutation s of monomial degree $r+c$ (actual degree $(q-1)(r+c)$) which fixes $\{1, \ldots, (q-1)r\}$ setwise induces an action on A. This action is most easily described if we think of s as a permutation on $\mathrm{GF}(q)^{\#} \times \{1, \ldots, n\}$, as in (ii) above; then s fixes $\{(\alpha, i) | \alpha \in \mathrm{GF}(q)^{\#}, 1 \le i \le r\}$ setwise. If $(1,i)^s = (\alpha, k)$ and $(1, r+j)^s = (\beta, r+m)$, then row i of A is multiplied by α and moved to row position k, and column j of A is multiplied by β and moved to column position m. Thus

$$A^s = (b_{ij}), \quad \text{where} \quad b_{ij} = \lambda^{-1}\mu^{-1}a_{i'j'},$$

with λ, μ, i', and j' determined by

$$(\lambda, i') = (1,i)^{s^{-1}} \quad \text{and} \quad (\mu, r+j') = (1, r+j)^{s^{-1}}.$$

If $A^s = B$, we say that s is an *monomial isomorphism* of A to B. When $A^s = A$, s is called a *monomial automorphism* of A. The group formed by the automorphisms is called the *monomial automorphism group* of A, and denoted AUT$^*(A)$.

For example, over the field GF(4), the action of a monomial permutation s of monomial degree 5 (actual degree 15) on a 2×3 matrix A is illustrated by the following.

$$(1,1)^s = (3,2), \quad (1,2)^s = (1,1), \quad (1,3)^s = (2,5), \quad (1,4)^s = (3,3), \quad (1,5)^s = (2,4),$$

$$s = (1,6,3,5,2,4)(7,14,12,8,15,10,9,13,11), \qquad A = \begin{pmatrix} 2 & 0 & 3 \\ 0 & 3 & 1 \end{pmatrix}, \qquad A^s = \begin{pmatrix} 2 & 2 & 0 \\ 0 & 3 & 2 \end{pmatrix}.$$

IV. PARTITION BACKTRACK COMMANDS

The commands employing the partition backtrack method that are currently available are described below. Note material in square brackets is optional. (The brackets themselves are not to be typed.) Discussion of most of the available options will be deferred to Section V; only those unique to a specific command will be mentioned here.

Options are never required, but they may prove useful in controlling the format of the output or the procedures used in the computation. For example, certain options allow for a time versus space tradeoff. For some "unusual" groups (e.g., very dense imprimitive groups), it may be necessary to specify nonstandard options in order to obtain acceptable performance.

The partition backtrack programs described here represent full implementations of the partition backtrack method, as set forth in (Leon, 1991), with two exceptions.

i) The criterion in Prop. 8(iii) is not checked.

ii) In coset–type computations, the refinement $\underline{\mathrm{R}}^+$ of Figure 8 is always taken as $\underline{\mathrm{R}}^{\dagger}$

Set stabilizers: Set stabilizers may be computed by the `setstab` command. The format is

`setstab` [*options*] *permGroup* *pointSet* *stabilizerSubgroup*

This command computes the set stabilizer in the permutation group *permGroup* of the set *pointSet* and saves the result (in Cayley library format) as the permutation group *stabilizerSubgroup.*

At present, the set stabilizer program sometimes run slowly in doubly transitive groups, and often runs very slowly in groups that are triply transitive or "almost" triply transitive (e.g., $\mathrm{SL}_n(2)$), especially when both the point set and its complement are large and when the set stabilizer turns out to be small. Imprimitive groups closely related to doubly transitive groups may also cause difficulty. Modifications to alleviate this difficulty, at least in part, will be added eventually.

† In order to allow this manual to be printed without special AMS TeX fonts, underlined letters (e.g., $\underline{\mathrm{R}}$) are used here as a substitute for letters appearing in the Euler Fraktur (German) font in (Leon, 1991).

Set images: Given a permutation group G on $\{1, \ldots, n\}$ and subsets Λ and Φ of $\{1, \ldots, n\}$, the `setimage` command may be used to determine if there exists an element g of G such that $\Lambda^g = \Phi$. The format is

```
setimage [options] permGroup pointSet1 pointSet2 groupElement
```

where *permGroup*, *pointSet1*, *pointSet2*, and *groupElement* play the role of G, Λ, Φ, and g, respectively. That is, the command determines whether there exists an element of *permGroup* mapping *pointSet1* to *pointSet2* and, if so, saves one such element as the permutation *groupElement*. Note that *groupElement* will not be created if $\Phi \notin \Lambda^G$. (Unless the `-q` option is specified, a message indicating whether $\Phi \in \Lambda^G$ will be written to the standard output.) The potential difficulties with doubly and triply transitive groups mentioned for set stabilizer computations apply here also.

Ordered partition stabilizers: Stabilizers of ordered partitions may be computed by the `parstab` command. The format is

```
parstab [options] permGroup ordPartition stabilizerSubgroup
```

This command computes the stabilizer in the permutation group *permGroup* of the ordered partition *ordPartition* and saves the result as the permutation group *stabilizerSubgroup*. The remarks about performance on doubly and triply transitive groups for set stabilizer computations apply here also.

Ordered partition images: Given a permutation group G on $\{1, \ldots, n\}$ and ordered partitions Π and Σ of $\{1, \ldots, n\}$, the `parimage` command may be used to determine if there exists an element g of G such that $\Pi^g = \Sigma$. The format is

```
parimage [options] permGroup ordPartition1 ordPartition2 groupElement
```

where *permGroup*, *ordPartition1*, *ordPartition2*, and *groupElement* play the role of G, Π, Σ, and g, respectively. That is, the command determines whether there exists an element of *permGroup* mapping *ordPartition1* to *ordPartition2* and, if so, saves one such element as the permutation *groupElement*. The permutation *groupElement* is created only if $\Sigma \in \Pi^G$. The remarks about performance on doubly and triply transitive groups given above for set stabilizer computations apply here also.

Group intersections: Given permutation groups G and H on $\{1, \ldots, n\}$, the `inter` command may be used compute the intersection $G \cap H$. The format is

```
inter [options] permGroup1 permGroup2 interGroup
```

This command computes the intersection of groups *permGroup1* and *permGroup2* and saves the result as the group *interGroup*. The potential difficulty with doubly and triply transitive groups discussed above for set stabilizer computations applies here also when both groups are doubly or triply transitive.

Centralizers of elements: Given a permutation group G and a permutation x (not necessarily contained in G), the `cent` command may be used compute $C_G(x)$, the centralizer in G of x. The format is

`cent` [*options*] *permGroup* *permutation* *centralizerSubgroup*

Here *permGroup*, *permutation*, and *centralizerSubgroup* play the role of G, x, and $C_G(x)$ above. That is, the command computes the centralizer in the group *permGroup* of the permutation *permutation* and saves the result as the group *centralizerSubgroup.*

For this command, it is permissible to specify *permGroup* as `#`*n*, where n is an integer at least 2, in which case *permGroup* is taken as the symmetric group of degree n; in this situation, the normal restrictions on base size (discussed later) do not apply to *permGroup*, although they do apply to *centralizerSubgroup.*

The `cent` command accepts an option `-np` which can have an effect (often small) on performance. If this option is specified, a refinement process based on the cycle structure of x will not be used. The effect is to reduce memory requirements a bit. In many cases, the running time does not change significantly, but in some cases it does increase a great deal.

It should be noted that in many cases, perhaps most cases arising in practice, centralizer computations are fairly easy even for conventional algorithms, and the partition backtrack program may perform no better than, and perhaps not even as well as, programs based on conventional techniques, such as those in Cayley. (Note, however, that, unlike Cayley, the program here does not require that the permutation to be centralized lie in the group.)

Conjugacy of elements: Given a permutation group G and permutations x and y (not necessarily contained in G), the `conj` command may be used to determine if x and y are conjugate under G and, if so, to find g in G with $x^g = y$. The format is

`conj` [*options*] *permGroup* *permutation1* *permutation2* *conjugatingElement*

Here *permGroup*, *permutation1*, *permutation2*, and *conjugatingElement* play the role of G, x, y, and g above. That is, the command determines if there exists an element of *permGroup* conjugating *permutation1* to *permutation2* and, if so, it saves one such element as the permutation *conjugatingElement.* If the two permutations are not conjugate in *permGroup*, then *conjugatingElement* is not created. In any case, a message indicating the result is written to the standard output (unless the `-q` option is specified).

As with the `cent` command, *permGroup* may be specified as `#`*n*, in which case conjugacy in the symmetric group of degree n is checked. (In this case, the program merely checks that the two permutations have the same cycle structure.) Also, the `-np` option is accepted, and it works as described above for the `cent` command.

As with centralizer computations, conjugacy calculations are usually easy with conventional algorithms, and the partition backtrack method may not yield an improvement.

Centralizers of groups: Given a permutation groups G and a second permutation group E (not necessarily contained in G), the `gcent` command may be used compute $C_G(E)$, the centralizer in G of E. The format is

`gcent` [*options*] *permGroup1* *permGroup2* *centralizerSubgroup*

Here *permGroup1*, *permGroup2*, and *centralizerSubgroup* play the role of G, E, and $C_G(E)$ above. That is, the command computes the centralizer in the group *permGroup1* of the group *permGroup2* and saves the result as the group *centralizerSubgroup*.

As with the element centralizer command (`cent`), it is permissible to specify *permGroup1* as `#`n, indicating the symmetric group of degree n.

To an even greater extent than element centralizer calculations, group centralizer calculations tend to be easy ones for conventional algorithms; the full power of the partition method is not needed, and perhaps not even desirable. For this reason, little effort has gone into development of the `gcent` command; its implementation is fairly crude, and it is included primarily for completeness. There are two options, `-cg:`m and `-cp:`p, which affect its performance; for some groups G, it may be necessary to assign them values different from the defaults (current 3 and 10, respectively). A full description of the significance of m and p will not be given here; however, we note that higher values (especially for m) increase memory requirements, and often increase execution time as well, but may be needed if the group E fails to have a small generating set (e.g., if E is a large elementary abelian group).

By specifying *permGroup1* and *permGroup2* as the same group, the `gcent` command may be used to compute the center of a group; note, however, that it represents an exceptionally inefficient algorithm for this purpose.

Automorphism groups of designs: The `desauto` command may be used to compute the automorphism group of a design. Here a *design* means any set of points (numbered $1, \ldots, n$ for some n) and any collection of subsets of the point set. The format of the design automorphism group command is:

`desauto` [*options*] *design* *autoGroup*

and the command sets *autoGroup* to the automorphism group of the design *design*.

The interpretation of the group *autoGroup* that is created depends on whether the `-pb` (points and blocks) option is specified. Let p and b denote the number of points and blocks, respectively, of the design.

i) If the option `-pb` is specified, then *autoGroup* is constructed as a group of degree $p+b$, in which the action on $1, \ldots, p$ is the action on points and in which the action on $p+1, \ldots, p+b$ is the action on blocks, the jth block being represented by $p+j$.

ii) If the `-pb` option is omitted, then *autoGroup* is constructed as a group of degree p, representing the action on points only. In this case, if there are repeated blocks, the group acting on points only has lower order than the group acting on points and blocks). When this situation arises, the group saved as *autoGroup* represents the group on points only, but the information written to the standard output during the computation refers to the group acting on points and blocks. (This occurs because the computation is carried out on points and blocks; restriction to points is performed only at the end; note also, for this reason, restriction to points only does not save time or memory.)

Isomorphism of designs: The `desiso` command may be used to check isomorphism of designs. The format is

`desiso` [*options*] *design1* *design2* *isoPerm*

and the command sets *isoPerm* to an isomorphism from design *design1* to design *design2*, provided the designs are isomorphic. (If not, the permutation *isoPerm* is not created. In any case, a message indicating the result is written to the standard output, unless the `-q` option is specified.)

As in the case of the `desauto` command, described above, the presence or absence of the `-pb` option determines whether *isoPerm* is constructed as a permutation on points and blocks, or on points only (the default). When the action on blocks is included, the jth block is represented by $p + j$.

Automorphism groups and monomial groups of matrices: The `matauto` command may be used to compute the automorphism group of a matrix. If the matrix elements are taken from a small finite field GF(q), then optionally the monomial automorphism group may be computed. (See Section III for definitions.) The command format is:

`matauto` [*options*] *matrix* *autoGroup*

and the command sets *autoGroup* to the automorphism group of the matrix *matrix* or, if the `-mm` option is specified, to the monomial automorphism group of *matrix*.

If the `-tr` option is specified, the matrix is transposed after it is read in, and all computations apply to the transposed matrix.

Let r and c denote the number of rows and columns, respectively, of the matrix $A = (a_{ij})$ whose group is to be constructed. Normally the automorphism group has degree $r + c$ and the monomial automorphism group has degree $(q - 1)(r + c)$; the interpretation of these groups is described in Section III. However, if the `-ro` (rows only) option is specified, the degree will be r or $(q - 1)r$, and the group will represent the action on rows only. Note that restriction to rows only may reduce the order of the group, just as in the case of designs restriction to points only may reduce the order of the group. When this occurs, the remarks above for design groups apply here also.

At present, the program for computing monomial groups of matrices is a very crude one. As a result, although it works reasonably for many matrices of fairly large size, it can fail to run in acceptable time even for very small matrices, e.g., matrices of all 0s. Sometimes use of the `-tr` option can get around this difficulty (which will be fixed eventually).

Isomorphism and monomial isomorphism of matrices: The `matiso` command may be used to check if two matrices are isomorphic or, if the matrix elements are from a finite field GF(q), monomially isomorphic. (See Section III for definitions.) The command format is

`matiso` [*options*] *matrix1* *matrix2* *isoPerm*

In the absence of the `-mm` option, the command sets *isoPerm* to an isomorphism from matrix *matrix1* to matrix *matrix2*, provided the matrices are isomorphic. (If not, the permutation *isoPerm* is not created). If the `-mm` option is specified, the command sets *isoPerm* to a monomial isomorphism from matrix *matrix1* to matrix *matrix2*, provided the matrices are monomially isomorphic. (In this case, the matrix entries should be field elements.) The effect of the `-ro` option is as described above for matrix automorphism group calculations.

Currently the monomial isomorphism program suffers from the same limitations as the monomial automorphism group program, as mentioned above.

Automorphism groups of linear codes: The `codeauto` command may be used to compute the automorphism group of a linear code over a small field GF(q). However, before the automorphism group of a code C may be computed, it is necessary to have a set V of vectors (not necessarily codewords) such that the following conditions hold. In these conditions, V^* denotes the set of all nonzero scalar multiples of vectors in V.

i) No vector in V is a scalar multiple of any other vector in V. (In particular, $|V^*| = (q-1)|V|$.)

ii) V is "reasonably small". (With a very large memory, "reasonably small" might mean 100,000 or more.)

iii) V^* is invariant under AUT(C) (the automorphism group of C),

iv) $|\mathrm{AUT}(V^*) : \mathrm{AUT}(C)|$ is very small. (The running time rises very rapidly as a function of this index. Note that, if V spans C, the index is 1.)

Often the set of minimal weight vectors of the code (scalar multiples removed if $q > 2$) make a suitable choice for V; minimum weight vectors of the dual code may also be used. This choice for V certainly satisfies (i) and (iii), may well satisfy (ii), and in many cases satisfies (iv). The author has available programs for computing the set of minimum weight vectors (or vectors of any specified weight.)

The format of the code automorphism group command is

`codeauto` [*options*] *code* *invarVectors* *autoGroup*

where *invarVectors* is the set V of vectors described above (in the format of a matrix, whose rows are the vectors). The command sets *autoGroup* to the automorphism group of the code *code*.

The `-cv` (coordinates and vectors) option for codes has essentially the same effect as the `-pb` option for designs. With this option, the automorphism group is saved in *autoGroup* as a permutation group of degree $(q-1)(n+|V|)$ (n = length of code), representing the action on (monomial) coordinates and invariant vectors; without the `-cv` option, it is saved as a permutation group acting of degree $(q-1)n$, representing the action on (monomial) coordinates only. (However, restriction to coordinates only can never lead to a reduction in the group order, as occurred with restriction to points or rows for designs or matrices.) For an explanation of the format of monomial permutations, see Section III.

At present, the program for computing groups of non–binary codes is a very crude one; sometimes it can fail to run in reasonable time even on small codes. Eventually this program will be improved.

Isomorphism of linear codes: The `codeiso` command may be used to check isomorphism of linear codes. However, before isomorphism of two codes C_1 and C_2 may be checked, it is necessary to have a sets V_1 and V_2 of vectors (not necessarily codewords of the two codes) such that V_1 and V_2 satisfy conditions (i), (ii), (iii), and (iv) above relative to C_1 and C_2, respectively, and in addition such that any isomorphism of C_1 to C_2 must map V_1^* to V_2^*. (As with code automorphism groups, V_1^* and V_2^* denote the sets of nonzero scalar multiples of vectors in V_1 and V_2, respectively. Often suitable choices for V_1 and V_2 are the minimal weight vectors of C_1 and C_2, respectively (scalar multiples removed.); minimal weight vectors of the duals of the two codes also could be used.

The format of the code isomorphism command is

`codeiso` [*options*] *code1* *code2* *invarVectors1* *invarVectors2* *isoPerm*

where *invarVectors1* and *invarVectors2* are the sets V_1 and V_2, respectively, of vectors described above (each in the format of a matrix, whose rows are the vectors). The command sets *isoPerm* to an isomorphism from *code1* to *code2*, if the codes are isomorphic; if not, *isoPerm* is not created.

As in the case of the `codeauto` command, described above, the presence or absence of the `-cv` option determines whether *isoPerm* is a permutation on (monomial) coordinates and invariant vectors, or on (monomial) coordinates only. The interpretation of monomial permutations is described in Section III..

Note that a number of the commands above are implemented as shell files (under Unix), batch files (under MS DOS), or exec files (under CMS). The commands that are implemented in this manner, and the contents of the Unix shell files, are as follows. (The list includes a few commands to be discussed in Section IX.)

command	*contents of shell file*
`setimage`	`setstab -image $*`
`parstab`	`setstab -partn $*`
`parimage`	`setstab -image -partn $*`
`conj`	`cent -conj $*`
`gcent`	`cent -group $*`
`desiso`	`desauto -iso $*`
`matauto`	`desauto -matrix $*`
`matiso`	`desauto -iso -matrix $*`
`codeauto`	`desauto -code $*`
`codeiso`	`desauto -iso -code $*`
`cjper`	`cjrndper -perm $*`
`ncl`	`commut -ncl $*`
`compper`	`compgrp -perm:$1 $2 $3`
`compset`	`compgrp -set:$1 $2 $3`
`chbase`	`orblist -chbase $*`
`ptstab`	`orblist -ptstab $*`

V. OPTIONS

A partial description of the options that are currently available follows. Most of the options are available with all of the commands described in Section IV. A few options apply only to subgroup computations, or only to coset–representative computations; these restrictions are noted below. Options applicable only to a single command are discussed with that command in Section IV.

In general, options may be specified in any order. However, if conflicting options are specified, the one specified last is the one that is used. (In some cases, conflicting options are treated as an error. Also, the `-l` and `-v` options, discussed later, are an exception to the general rule that options may be specified in any order; these options, if present, must come first, and the remainder of the command line is ignored.)

Entering any command with no options or arguments causes a brief summary of the command format to be displayed.

Options affecting file handling:

`-a` Normally, if a file name is specified for an object to be constructed, and if a file by that name already exists, the programs overwrite the existing file. With the `-a` option, they append to the existing file, rather than overwriting it.

`-p:`*path* Here *path* is a string. The string *path* is concatenated to the file name of every input file. This option can be useful if all the input files are in another directory. For example,

```
setstab  ../groups/psp62::psp62  ../groups/lambda::lambda  S
```

may be written more compactly as

```
setstab  -p:../groups/  psp62  lambda  S
```

(Note the final slash following `groups` is required.). The `-p` option has no effect on output files.

Options affecting output format:

`-i` This option applies to commands that construct and write out either a permutation or a permutation group. It causes permutations to be written in image format, rather than in cycle format (the default).

`-n:`*name* Here *name* is a string. The object created by the command will be named *name*. By default, the name assigned to the object will be the name of the Cayley library containing its definition. Note this option affects only the name of the object, not that of the file or the Cayley library.

`-q` Suppresses informative messages on the state of the computation, normally written to the standard output during the computation.

`-s` Causes statistics on the pruning of the backtrack search tree to be written out to the standard output. These statistics relate to the backtrack search tree defined in the author's paper (Leon, 1991), and are likely to be meaningful only to users familiar with that paper.

`-w:`*n* Here n should be a nonnegative integer. This option applies only to coset representative computations. If the degree is less than or equal to n, and if a coset representative is found, it will be included in the informative messages written to the standard output. (In any case, the coset representative will be written to a file in Cayley library format.) The default value of n is currently 300.

Options affecting performance of the algorithm:

`-b:`*k* Here k is a nonnegative integer which determines the extent to which base changes are performed in an attempt to improve pruning of the backtrack search tree using tests on double–coset minimality (Leon, 1991, Prop 8). When $k = 0$, base change is never performed (except during $\underline{\mathbf{R}}$–base construction, when it is used for a different purpose). As k increases, the number of base change operations performed increases; however, increasing k beyond the base size produces no further increase in the number of base change operations. Designating $k = 0$ reduces memory requirements and often produces the best running times as well. On the other hand, some high–density groups seem to require a higher value of k in order to obtain acceptable performance. By default, the program chooses a value of k based on the density and degree of transitivity of the group; quite often, this default value is 0.

Note: For coset representative computations, this option has no effect unless known subgroups of the two associated groups are specified; see discussion of the `-kL` and `-kR` options below.

`-g:`m

Here m should be a nonnegative integer. This is one of several parameters providing a time vs. space tradeoff. Small values of m, say 10 or less, minimize memory requirements, while large values of m, say 100 or greater, reduce the running time moderately for most difficult groups. Use of a high value is recommended for multiply transitive groups.

After $\underline{\mathbf{R}}$–base construction, the program attempts to reduce the height of the Schreier trees for the containing group by adding new strong generators. However, it will never add generators for this purpose if doing so would cause the total number of strong generators to exceed m. (It will also stop adding generators if the height falls below certain goals currently fixed in the program.)

`-k:`H

Here H specifies a permutation group (in the format *cayleyLibraryName* or *fileName*:: *cayleyLibraryName*). This option applies only to subgroup calculations. The group H must be a known subgroup of the group being computed. In principle, this option allows one to take advantage of any subgroup of the group being computed that happens to be known in advance. In practice, however, it seldom appears to speed up the computation by very much, and it increases memory requirements.

`-kL:`J
`-kR:`M

Here J and M must specify permutation groups (each in the format *cayleyLibraryName* or *fileName*:: *cayleyLibraryName*). These options apply only to coset representative calculations. Either or both may be specified. Associated with every coset representative computation, there are "left" and "right" groups, as explained in Section 2 of (Leon, 1991). The groups J and M must be known subgroups of these left and right groups, respectively. Specifying J and/or M, if known, increases memory requirements, but in some cases it may improve the running time. For some very dense groups, one or both of these options may be needed in order to allow the computation to finish in an acceptable amount of time.

`-mb:`k

Here k should be a nonnegative integer. This integer represents an upper bound on the size of the base for a permutation group. The default value of k is 62, which is more than adequate for many groups. For further discussion of the `-mb` option, see Section VII.

`-mw:`ℓ

Here ℓ should be a nonnegative integer whose value is at least several hundred. This integer represents an upper bound on the length of any word in the generators of any permutation group. For further discussion, see Section VII.

`-r:`*p* Here *p* should be a nonnegative integer, normally smaller than the integer *m* specified for the `-g` option described above. This is another option providing for a time versus space tradeoff. Small values of *p*, say less than 10, minimize memory requirements, while larger values, say 50 or higher, *may* reducing the running time, although usually not a great deal.

Whenever the number of strong generators for the containing group exceeds *p*, redundant strong generators are eliminated, using a procedure originally due to Sims (1971).

Special options:

`-l` This option, if present, must be the first option on the command line, and the remainder of the command line is ignored. (It may be omitted.) The `-l` option merely prints out limits on the default maximum base size, default maximum word length, degree, and other quantities with which this version of the program has been compiled. (See Section VII for discussion of these limits.)

`-v` This option, if present, must be the first option on the command line, and the remainder of the command line is ignored. (It may be omitted.) The `-v` option is intended to be used once following compilation of the program. It attempts to check that all the source files for the program were compiled with the same options and size limits. (See Section VII for discussion of size limits.)

VI. OUTPUT AND RETURN CODES

All programs for subgroup computations return a value of 0 if the computation is completed successfully and a nonzero value (currently 15) if the computation terminates due to an error (input file not found, incorrect format in input file, memory exhausted, size limit in program exceeded, etc.) All programs for coset representative computations return a value of 0 if the computation is completed successfully and a coset representative exists, 1 if it is completed but a coset representative does *not* exist, and a value different from 0 and 1 (currently 15) if the computation terminates due to an error.

Unless the `-q` option is specified, all of the programs write information about the progress of the computation to the standard output. Some of this information, most notably that relating to the **R**–base and the backtrack search tree (the latter given only if `-s` is specified) will probably be meaningful primarily to users familiar with the author's paper (Leon, 1991). Information of more general interest includes:

i) The order of the containing group (unless it is the symmetric group). Note that this order is determined by computing a base and strong generating set for the containing group when it is read in, unless they are supplied in the input file.

ii) The new (changed) base and strong generating set for the containing group computed during **R**–base construction, and the corresponding basic orbit lengths. In the notation of (Leon, 1991), this is the base $(\alpha_1, \ldots, \alpha_k)$ associated with the **R**–base.

iii) A base for the subgroup to be computed (subgroup computations) or for the subgroup associated with the right coset whose representative is to be computed (coset representative computations). This is the subgroup base associated with the $\underline{\mathbf{R}}$–base; in the notation of (Leon, 1991), it is $(\hat{\alpha}_1, \ldots, \hat{\alpha}_\ell)$. Note that this base is a subsequence of the base for the containing group in (ii) above.

iv) The basic cell sizes corresponding to the subgroup base in (iii) above (for definitions, see (Leon, 1991)). Note that each basic cell size provides an upper bound for the corresponding basic orbit length of the subgroup to be computed (subgroup–type computations). (Usually the bound is not sharp).

v) The number of strong generators for the containing group and the mean node depths in the Schreier trees for the basic orbits of the containing group. Depending on the `-g` and `-r` options, following $\underline{\mathbf{R}}$–base construction, additional strong generators may be added in an attempt to reduce the height of the Schreier trees. Figures are provided both before and after additional strong generators are added.

vi) [subgroup computations only] A message for each strong generator that is found for the subgroup. The message gives the level and the basic orbit lengths for the subgroup constructed thus far. (A generator will be said to be at level i if it fixes the first $i-1$ base points but moves the ith.)

vii) [subgroup computations only] The order of the subgroup that was computed.

viii) [subgroup computations only] The base (same as in (iii) above) and basic orbit lengths for the subgroup that was computed.

ix) [coset representative computations only] A message indicating whether a coset representative exists.

x) [coset representative computations only] If a coset representative exists and the degree is sufficiently low (depending on the `-w` option), the coset representative that was found.

xi) The time required for the computation. Note that the time to read in the containing group from a file, construct the initial base and strong generating set for the containing group (if not present in the input file), and to write out the subgroup or coset representative to a file is not included in this time. All computations relating to calculation of the subgroup or coset representative (including base changes in the containing group) are included.

Note that, in subgroup computations, the actual strong generators for the subgroup are not written to standard output, and in coset computations, the actual coset representative found may not (depending on the degree and `-w` option) be written to the standard output. However, both may be found (in Cayley library format) in the output file that is created.

For design, matrix, or code isomorphism computations, the isomorphism that is constructed is written to the standard output (assuming that the degree is sufficiently low) in a more easily readable (but not Cayley compatible) format than that described in Sections II and III. For designs with the `-pb` option, the action on points and blocks is given separately. For matrices, the action on rows and columns is given separately. For monomial isomorphism of matrices for non–binary codes, the monomial isomorphism is written in the following format:

$$([\lambda_1]i_1, [\lambda_2]i_2, \ldots, [\lambda_k]i_k)$$

This denotes the monomial permutation mapping 1 to $\lambda_1 i_1$, 2 to $\lambda_2 i_2$, etc. For example, to apply this monomial permutation to the rows of an r by c matrix, row j is multiplied by λ_j and the result is moved to row position i_j.

VII. SIZE LIMITS

There are a few fixed limits on the sizes of objects that the programs can handle. These limits can be changed only by recompiling the programs. The order of any group may have at most 30 distinct prime divisors. The name of any file may have at most 60 characters (including path information supplied with the `-p` option). The name of any object may have at most 16 characters. Most importantly, if the program is compiled using 16–bit integers, the maximum degree of any permutation group is limited to slightly less than 2^{16} (about 65000). If it is compiled using 32–bit integers, there is, for practical purposes, no fixed limit. Note, however, that use of 16–bit integers reduces memory requirements substantially, and it is recommended unless groups of degree greater than 65000 (approx) are to be used. Only machines having at least 20 to 25 megabytes of memory are likely to be able to handle groups of degree high enough to require 32–bit integers. Currently both 16–bit and 32–bit compiled versions of the programs are available.

Although there is no fixed limit on the base size for a permutation group, a limit must be established at the time that the program is initiated, and this limit remains fixed during that run. This limit may be set at k by means of the `-mb:`k option, or it may be allowed to default to 62. Note that large values of k increase running times and memory requirements slightly even if the actual base size turns out to be much less than k.

For the most part, the amount of memory (real and virtual) available determines the sizes of objects that can be handled by the programs. Memory requirements depend heavily on the degree of the group, and to a somewhat lesser extent on the base size. The programs can use virtual memory to some extent; however, if virtual memory used exceeds real memory by a factor of more than 1.6 to 1.8, excessive paging is likely to occur. The following steps may be taken to reduce memory requirements; the steps are listed in order of decreasing benefit.

i) If the degree of the group is less than 65000 (approx), use a 16–bit version of the program rather than a 32–bit version. The 16–bit version is likely to run about as fast as the 32–bit version, and it requires a great deal less memory.

ii) Specify options of `-g:1` and `-r:1`. These options are likely to increase the execution time substantially, but they often save a good deal of memory. As a compromise, values greater than 1 but less than the defaults may be specified, e.g., `-g:20` and `-r:15`

iii) Specify the option `-b:0` if it is not already the default. In the majority of cases, this option will not increase execution time, and it reduces memory requirements considerably. However, in a great many cases, `-b:0` will already be the default. (The value for this and other options is displayed on the standard output when the program is is run.)

iv) For (element) centralizer and conjugacy calculations, specify the `-np` option. This saves a modest amount of memory. The effect on execution time is hard to predict; in some cases, it may lead to a major increase. For group centralizer calculations, options of `-cg:2` and `-cp:`*i*, where i is 3 to 5, may be tried, although on some groups they may raise the running time to unacceptable levels.

v) Specify the option `-mb:`*k* for a value of k less than the default of 62. The `-mb:`*k* options sets a limit of k on the base size for the group; often a value considerably less than 62 (e.g., 15 or 20) will be adequate. However, the amount of memory saved is relatively small.

In the author's experience, the programs can often handle groups of degree as high as $2000m$ to $3000m$, where m is the number of megabytes of *real* memory available. However, for groups lacking a relatively small base, the limit on the degree is much lower. Also, this limit applies only to memory requirements; depending on the type of computation and the specific groups, it may or may not be possible to perform computations in groups this large in an acceptable amount of time.

VIII. EXAMPLES

The author has prepared a number of sample objects that may be used to test the programs. In the Unix distribution, these objects appear in various subdirectories of the directory `partn/examples`.

The subdirectories `psp62`, `psp82`, `psu72`, `omg84`, `fi23`, `ahs2`, `rubik4`, and `syl128` of directory `partn/examples` contain examples for computation in the groups $\mathrm{PSp}_6(2)$ of degree 63, $\mathrm{PSp}_8(2)$ of degree 255, $\mathrm{PSU}_7(2)$ of degree 2709, $\Omega_8^+(4)$ of degree 5525, Fi_{23} of degree 31671, AUT(HS) × AUT(HS) of degree 200, the group of a 4 × 4 Rubik's cube (degree 96), and a Sylow 2–subgroup of the symmetric group $Sym(128)$ of degree 128, respectively. Note that, for the last two groups, any base will be large, and the `-mb` option (e.g., `-mb:75`) will need to be specified. Each of the directories contains files as follows, where *grp* is to be replaced by the actual name of the directory.

grp The permutation group mentioned above.

grp`x` A group permutation isomorphic to the group *grp* and having a small but nontrivial intersection with *grp*. The intersection of *grp* and *grp*`x` may be computed by the command

```
inter grp grpx int
```

which saves the intersection as the group `int`. The file *grp*`x` contains a comment giving the order of the intersection.

`set1` A random point set of size half the degree of *grp*. Except in the case of `rubik4` and `syl128`, the set stabilizer of `set1` in the group *grp* turns out to be trivial. This set stabilizer may be computed by the command

```
setstab  grp  set1  stab1
```

which saves the set stabilizer as the group `stab1`.

`set2` A point set of size approximately half the degree whose set stabilizer in *grp* is a dihedral group of low order, except in the case of `rubik4` and `syl128`. This set stabilizer may be computed by the command

```
setstab  grp  set2  stab2
```

which saves the set stabilizer as the group `stab2`. The file `set2` contains a comment indicating the order of the stabilizer.

`set3` A point set of size roughly half the degree (in most cases) whose set stabilizer in *grp* is a group of high order. This set stabilizer may be computed by the command

```
setstab  grp  set3  stab3
```

which saves the set stabilizer as the group `stab3`. The file `set3` contains a comment indicating the order of the stabilizer.

`set1x` A point set obtained by applying a randomly–chosen element of the group *grp* to `set1`. The command

```
setimage  grp  set1  set1x  g
```

may be used to find an element `g` of the group *grp* mapping `set1` to `set1x`.

`set1y` A point set having the same cardinality as `set1` but not equal to the image of `set1` under any element of *grp*. The command

```
setimage  grp  set1  set1y  h
```

may be used to determine that `set1y` is not in fact an image of `set1` under the group. (The permutation h will *not* be created.

`par1` A partition of the set $\{1,\ldots,n\}$, where n is the degree of group *grp*. (The file contains a comment indicating the number of cells and cell sizes.) The stabilizer in *grp* of `par1`, treated as an ordered partition, may be computed by the command

```
parstab  grp  par1  pstab1
```

which saves the ordered partition stabilizer as the group `pstab1`. The file `par1` contains a comment giving the order of the stabilizer.

`par1x` A partition obtained by applying a randomly–chosen element of the group *grp* to `par1`. The command

```
parimage  grp  par1  par1x  i
```

may be used to find an element `i` of the group *grp* mapping `par1` to `par1x`.

`par1y` A partition in which the sizes of the cells match those in `par1`, but which is not the image of `par1` under any element of the group *grp*. The command

```
parimage  grp  par1  par1y  j
```

may be used to demonstrate that `par1y` is not an image of `par1` under any group element.

`elt1` An element of the group *grp* having a fairly large centralizer in *grp*. This centralizer may be computed by the command

```
cent  grp  elt1  cent1
```

which saves the centralizer as the group `cent1`. The file `elt1` contains a comment stating the order of the centralizer.

`elt1x` An element conjugate under the group *grp* to `elt1`. An element of *grp* conjugating `elt1` to `elt1x` may be found by the command

```
conj  grp  elt1  elt1x  c1
```

which sets `c1` to such a conjugating element.

`elt2` An permutation *not* in the group *grp* having a nontrivial centralizer in *grp*. This centralizer may be computed by the command

```
cent  grp  elt2  cent2
```

which saves the centralizer as the group `cent2`. The file `cent2` contains a comment indicating the order of the centralizer.

`elt2x` A permutation (not in *grp*) conjugate under *grp* to `elt2`. An element of *grp* conjugating `elt2` to `elt2x` may be found by the command

```
conj  grp  elt2  elt2x  c2
```

which sets `c2` to such an element.

`elt2y` A permutation not in *grp* having the same cycle structure as `elt2` but not conjugate under *grp* to `elt2`. Non–conjugacy may be demonstrated by the command

```
conj  grp  elt2  elt2y  d
```

which does *not* create a permutation `d`.

Note that, in the case of the group `fi23`, about 16 megabytes of real memory may be needed to perform the calculations above.

The subdirectories `q17` and `q32` contain designs, (0,1)–matrices, and codes based on the quadratic residue code Q_{17} of length 17 and on the extended quadratic residue code Q_{32} of length 32, respectively. The contents of these directories are as follows, where i denotes either 17 or 32.

`q`*i* The quadratic or extended quadratic residue code.

`v`*i*

The matrix whose rows are the weight 5 ($i = 17$) or weight 8 ($i = 32$) codewords of the code `q`*i*. The automorphism group of the code `q`*i* may be computed by the command

```
codeauto  qi  vi  A
```

or

```
codeauto  -cv  qi  vi  A
```

which saves the automorphism group as the group `A`, either as a group of degree i or as a group of degree $i + k$, where k is the number of codewords in the set `v`*i*.

`q`*i*`x`

Another quadratic residue code obtained from `q`*i* by applying a random permutation to the coordinates..

`v`*i*`x`

The matrix whose rows are the weight 5 ($i = 17$) or weight 8 ($i = 32$) codewords of the code `q`*i*`x`. An isomorphism from `q`*i* to `q`*i*`x` may be found by the command

```
codeiso  qi  qix  vi  vix  s
```

which saves the isomorphism found as the permutation `s`.

`d`*i*

The design on $\{1, \ldots, i\}$ whose blocks correspond to the codewords of weight 5 ($i = 17$) or weight 8 ($i = 32$) in `q`*i*. The automorphism group of this design (which must contain the group of the corresponding code, and which in fact equals it) may be computed by the command

```
desauto  di  B
```

or

```
desauto  -pb  di  B
```

which saves the automorphism group as the group `B`, either as a group on points only, or as a group on points and blocks. Note that the incidence matrix of the design `d`*i* is the transpose of the matrix `v`*i*, so the same automorphism group could be computed by the command

```
matauto  -tr  vi  A
```

`d`*i*`x`

A design obtained from `d`*i* by applying a random permutation to the points. (The order of the blocks is also permuted randomly.) An isomorphism from `d`*i* to `d`*i*`x` may be found by the command

```
desiso  di  dix t
```

which sets `t` to one such isomorphism.

The subdirectory `dmcl` contains a design based on the sporadic simple group of McLaughlin (McL, degree 275). In this group, a point stabilizer has orbits of length 1, 112, and 162. The design `dmcl` on $\{1, \ldots, 275\}$ has 275 blocks, each of size 112; the blocks are the orbits of length 112 in the 275 point stabilizers. The group of this design, which must contain AUT(McL) and turns out to equal to AUT(McL), may be computed by the command

```
    desauto  dmcl  Y
```

which saves the group as `Y`. (Note that we are computing the group of the design, not the group of the graph associated with the orbit of length 112; in general, the design group is larger than the graph group, although in this case they are the equal.) The directory also contains a second design `dmclx`, isomorphic to `dmcl`. An isomorphism may be found by the command

```
    desiso  dmcl  dmclx  s
```

which sets `s` to an isomorphism from `dmcl` to `dmclx`.

Finally, the subdirectories `had32` and `had104` contains 32×32 and 104×104 matrices over GF(3), respectively, which are essentially the Paley–Hadamard matrices. (The entries of -1 have been changed to 2.) The (monomial) automorphism group of either of these Hadamard matrices may be computed by the command

```
    matauto  -mm  hadi  Z
```

(i = 32 or 104), which sets `Z` to the automorphism group. This subdirectories also contain matrices `had`*i*`x` obtained by applying random monomial permutations to the rows and columns of `had`*i*. Equivalence of `had`*i* and `had`*i*`x` may be established by the command

```
    matiso  -mm  hadi  hadix  w
```

which sets `w` to an monomial isomorphism from `had`*i* to `had`*i*`x`. Finally, the subdirectories contain Hadamard designs `dhad`*i* (i = 32 or 104) and equivalent designs `dhad`*i*`x`, whose groups may be computed using the `desauto` command, and whose equivalence may be established with the `desiso` command.

IX. OTHER COMMANDS

In the course of testing and benchmarking the partition backtrack algorithms described in Section IV, the author developed a number of other programs. Most of these programs were put together quickly, with a view toward simplicity rather than efficiency; in some cases, they are very inefficient. Also, some of them perform only minimal error checking. Nonetheless, they may prove useful since they operate on objects specified in the format described in Section II.

All of these programs accept the `-a`, `-i`, `-mb:`*k*, `-mw:`*w*, `-n:`*name*, `-p:`*path*, and `-q` options, as described in Section V, whenever they would be meaningful. (For example, the `-i` option is meaningful only if the command creates a permutation or permutation group.) Other options vary by command, and are discussed separately for each command below.

Base and strong generating set construction: The `generate` command may be used to construct a probable base and strong generating set for the permutation group generated by specified permutations. The random Schreier method (Leon, 1980) is used. If the group order is known in advance, this method always produces a correct base and strong generating set, although there is no bound on the time required to do so. Otherwise, there is no guarantee that the method will produce a correct result. However, in the author's experience, it nearly always does give a correct result, and it runs far more quickly than alternative methods, such as the Schreier–Sims or Schreier–Todd-Coxeter–Sims algorithms.

The format for the command is

`generate` *options* *inputGroup* *outputGroup*

where *options* denotes

[`-a`] [`-i`] [`-mb:`k] [`-mw:`w] [`-n:`*name*] [`-nro`] [`-p:`*path*] [`-q`] [`-s:`*seed*] [`-ti:`i] [`-tr:`m] [`-z`]

Here *inputGroup* denotes the original permutation group, for which a base and strong generating set are not yet available. The factored group order for *inputGroup* may or may not be present. The random Schreier method is used to construct a probable base and strong generating set for *inputGroup*,, and the result is saved as the permutation group *outputGroup.* If the factored group order is present, the computation will continue until a base and strong generating set has been found. Otherwise it continues until m consecutive quasi–random elements of the group factor in terms of the possible base and strong generating set, where m is the integer specified in the `-tr:`m option (default 40). High values of m may be specified to reduce the chance of an incorrect result, at the cost of slowing down the computation.

Normally, before a new strong generator is added to the strong generating set, an attempt is made to replace the new generator by a power of it, in order to obtain generators of low order. (This may be desirable later on if the Schreier–Todd–Coxeter–Sims method is used to verify the base and strong generating set; in addition, it saves space whenever a non–involutory generator is converted to an involution.) This attempt may be suppressed by the `-nro` option. Note, however, that replacement of generators by powers is relatively inexpensive, so the `-nro` option saves little time. The `-ti:`i option may be specified in order to have the program try harder to find involutory generators. Up to i consecutive generators that cannot be converted to involutory generators will be rejected. The default for i is 0; higher values often increase the execution time a good deal.

If the `-z` option is specified, the program will make some attempt to remove certain redundant strong generators from the strong generating set for *outputGroup.*

Base change: The `chbase` command may be used to change the base in a permutation group. The command format is

`chbase` *inputGroup* $p_1, p_2, \ldots, p_k$ *outputGroup*

where *options* denotes

[`-a`] [`-i`] [`-mb:`k] [`-mw:`w] [`-n:`*name*] [`-p:`*path*] [`-q`] [`-z`]

The base for permutation group *inputGroup* is changed, if necessary, so that it begins with $p_1, p_2, \ldots, p_k$, and the group with this new base is saved as *outputGroup.* Note that, in the list $p_1, p_2, \ldots, p_k$ of points, individual points are separated by commas but *not* by blanks. Note also that the points $p_1, p_2, \ldots, p_k$ are included in the new base even if they are redundant as base points. However, no other redundant base points will appear in the new base. If the `-z` option is specified, certain redundant strong generators will be removed following the base change.

Conjugation by a specified permutation: The `cjper` command may be used to conjugate an object (group, permutation, point set, partition, design, matrix, or code) by a specified permutation. The command format is

`cjper` *options type object conjugateObject conjugatingPerm*

where *options* denotes

[`-a`] [`-b`] [`-d:`*deg*] [`-i`] [`-mb:`*k*] [`-mm`] [`-mw:`*w*] [`-n:`*name*] [`-p:`*path*] [`-q`]

Here *type* must be one of the keywords `group`, `perm`, `set`, `partition`, `design`, `matrix`, or `code`, *object* must be an object of the type designated by *type*, and *conjugatingPerm* must be a permutation. In the event that *object* is a permutation, set, or partition, the `-d:`*deg* option *must* be used to specify the degree *deg*. The program sets *conjugateObject* to the object obtained by conjugating *object* by *conjugatingPerm*, in the case that *object* is a group or permutation, or to the object obtained by applying *conjugatingPerm* to *object*, if *object* is a point set, partition, design, matrix, or code. In the event that *object* is a group, the `-b` option forces the program to compute a base and strong generating set for *conjugateObject*; by default, *conjugateObject* will have a base and strong generating set only if *object* does.

In the event that *object* is a design or matrix, the degree is treated as the number of points plus the number of blocks, or the number of rows plus the number of columns, respectively. Blocks or columns are permuted as well as points or rows. However, since the input format for permutations permits a permutation of degree n to be treated as a permutation of any higher degree, this represents no real restriction.

If *object* is a matrix over a finite field GF(q), the `-mm` option may be specified. Then *conjugatingPerm* must have degree $(q-1)(r+c)$, where r and c are the number of rows and columns, and must satisfy the “monomial property”, as described in Section III.

Conjugation by a random permutation: The `cjrndper` command may be used to conjugate an object by a either a random permutation, or by a permutation chosen at random from a specified permutation group. The command format is

`cjrndper` *options type object conjugateObject* [*conjugatingPerm*]

where *options* denotes

[`-a`] [`-b`] [`-d:`*deg*] [`-g:`*grp*] [`-i`] [`-mb:`*k*] [`-mm`] [`-mw:`*w*] [`-n:`*name*] [`-p:`*path*] [`-s:`*seed*] [`-q`]

Here *type* must be one of the keywords `group`, `perm`, `set`, `partition`, `design`, `matrix`, or `code`, and *object* must be an object of the type designated by *type*. In the event that *object* is a permutation, set, or partition, the `-d:`*deg* option *must* be used to specify the degree *deg*. If *object* is a permutation or permutation group, the program sets *conjugateObject* to the object obtained by conjugating *object* by a certain permutation x; if *object* is a point set, partition, design, matrix, or code, it sets *conjugateObject* to the object obtained by applying a certain permutation x to *object*. In either case, the permutation x is chosen at random from the group *grp*, if the `-g:`*grp* option is specified, or at random from the symmetric group, if the `-g:`*grp* option is omitted. If *conjugatingPerm* is specified, the permutation x is saved as *conjugatingPerm.* The `-s:`*seed* option may be used to specify a specific seed for the random number generator that is used internally. In the event that *object* is a group, the `-b` option forces the program to compute a base and strong generating set for *conjObject*, even when one is not available for *object.*

In the event that *object* is a design (or matrix), both points and blocks (or both rows and columns) are permuted.

If *object* is a matrix over a finite field GF(q), the `-mm` option may be specified. Then a random monomial permutation is applied to the rows and columns of the input matrix.

Commutator groups and lower central series: The `commut` command may be used to compute commutator groups. Repeated application of the command may be used to compute lower central series. Specifically, given a group G and a (not necessarily normal) subgroup H of G, the command computes the commutator group $C = [G, H]$. The command format is

`commut` *options* *permGroup* [*subgroup*] *commutatorGroup*

where *options* denotes

[`-a`] [`-i`] [`-mb:`k] [`-mw:`w] [`-n:`*name*] [`-p:`*path*] [`-q`]

Here, *permGroup*, *subgroup*, and *commutatorGroup* play the role of G, H, and C above, respectively. If *subgroup* is specified, it must be a subgroup of *permGroup* (not checked); the command sets *commutatorGroup* to the commutator of *permGroup* and *subgroup.* If subgroup is omitted, the command sets *commutatorGroup* to the commutator of *permGroup* with itself (the derived group).

At present, the strong generating set for the commutator group is constructed only by the random Schreier method. Thus there is a (probably small) possibility that the base and strong generating set constructed for the commutator group will be incorrect. (If this occurs, the generators constructed will generate the commutator group, but not strongly. This undesirable feature will be fixed eventually.)

The return code from the `commut` command will 0 if the commutator group has order 1. Otherwise the return code will depend on whether the order of H (i.e., *subgroup*) is known in advance. If so, the return code will 1 if $|C| \neq |H|$ and 2 otherwise. (If H is normal in G, these correspond to the cases $H \subset G$ and $H = G$, respectively.) If not, the return code will be 3.

Comparison of groups, normality, and centralization: The `compgrp` command may be used to check if either of two permutation groups is contained in the other, or normalizes the other, or whether the two groups centralize each other. The command format is

`compgrp` [`-c`] [`-n`] [`-mb:`k] [`-mw:`w] [`-p:`*path*] *permGroup1* *permGroup2*

This command checks if either of *permGroup1* or *permGroup2* is contained in the other. If the `-n` option is specified, it also checks if either group normalizes the other. (Note here the `-n` option is used for a different purpose that that described in Section V.) If the `-c` option is given, it checks whether the two groups centralize each other. Messages indicating the result are written to the standard output. The return code is 0 if the two groups are equal, 1 if *permGroup1* is a proper subgroup of *permGroup2*, 2 if *permGroup2* is a proper subgroup of *permGroup1*, and 3 otherwise. Note: The procedure for checking normality is, at present, extremely inefficient in many cases.

Comparison of permutations: The `compper` command may be used to check if two permutations are equal, or if a permutation is the identity. The command format is

`compper` *degree* *permutation1* [*permutation2*]

This command checks if *permutation1* and *permutation2*, both of which must be permutations of degree *degree*, are equal. It prints a message indicating whether the permutations are equal, and gives a return code of 0 if they are equal and 1 otherwise. If *permutation2* is omitted, it is taken as the identity; thus the command checks if *permutation1* is the identity.

Comparison of point sets: The `compset` command may be used to check if two point sets are equal. The command format is

`compset` *degree* *set1* [*set2*]

This command checks if *set1* and *set2*, both of which must be points sets of degree *degree*, are equal, or if either is contained in the other. if *set2* is omitted, it is taken to be the empty set, and the command checks if *set1* is empty. It prints a message indicating the result. If *set2* is specified, the return code is 0 if the two sets are equal, 1 if *set1* is properly contained in *set2*, 2 if *set1* properly contains *set2*, and 3 otherwise. If *set2* is omitted, the return code is 0 if *set1* is empty and 1 otherwise.

Coset weight distributions of codes: The `cwtdist` command† may be used to compute the coset weight distribution of a linear code. At present, the program is restricted to binary codes of codimension at most 32. The program is highly optimized; nonetheless, time and space requirements may restrict the codimension to values less than its maximum of 32. The command format is

`cwtdist` *options* *code* [*maxCosWeight* [*matrix*]]

where *options* denotes

[`-a`] [`-n:`*name*] [`-p:`*path*] [`-q`]

The program computes the coset weight distribution of the code *code*, that is, for each d, it determines the number of cosets of the code having minimum weight d. If *maxCosWeight* is specified (It should be an integer.), the computation is performed only for $d \leq maxCosWeight$. If *matrix* is given, a coset representative is saved for one coset whose minimum weight is the minimum of the covering radius and *maxCosetWeight*. *Note:* It is not possible to specify *matrix* without specifying *maxCosWeight*; however, an artificially large value of *maxCosWeight* may be given instead.

† At time of writing, this command was not complete. It should be available shortly.

Designs from groups: The `orbdes` command may be used to construct a block design D from a permutation group G, which normally should be transitive. The design D will have n points and n blocks, each having the same number of points, where n is the degree. The automorphism group $\mathrm{AUT}(D)$ will contain the group G (perhaps properly). For a specified point γ, let Γ denote the orbit of γ under the point stabilizer G_1. The blocks of D are exactly $\Gamma^{u_1}, \ldots, \Gamma^{u_k}$, where k is the size of the G–orbit of 1, and $u_1, \ldots, u_k$ map 1 to the different points of the G–orbit of 1. The command format is

`orbdes` [`-a`] [`-m`] [`-mb:`*k*] [`-mw:`*w*] [`-p:`*path*] *permGroup* *point* *design*

Here *permGroup*, *point*, and *design* play the role of G, γ, and D above, respectively. If the `-m` option is given, the incidence matrix of the design is written out as a matrix; otherwise the design is written in the standard format for designs.

Finding group elements of specified order: The `fndelt` command may be used to locate group elements of specified order, using a quasi-random search process. (Random group elements are examined in searching for ones whose order is a multiple of the specified order.) The command format is

`fndelt` *options* *permGroup* n k_1 k_2 ...

where *options* denotes

[`-a`] [`-f`] [`-i`] [`-m:`*trials*] [`-mb:`*k*] [`-mw:`*w*] [`-n:`*name*]
[`-o:`*ord*] [`-p:`*path*] [`-po`] [`-s:`*seed*] [`-wg:`*grp*] [`-wp:`*perm*]

By default, the command searches quasi–randomly until it finds n involutions in the group *permGroup*, and if k_1, k_2, ... are specified, it saves the k_1st, k_2nd, ... elements found. (See discussion of the `-wp` and `-wg` options below.) The value of n may be at most 99. The seed used in the random number generator may be specified by the `-s:`*seed* option; two invocations with the same (default or specified) seed should produce identical results. If the `-o:`*ord* option is given, the command searches for elements of order *ord*, rather than for involutions. If the `-f` option is specified, it prints the number of fixed points of each element found. If the `-po` option is specified, it prints the orders of all products of the elements found. If the `-wp:`*perm* option is given and k_1 is specified, the k_1st element found is saved as the permutation *perm*. If the `-wg:`*grp* is given and at least k_1 is specified, a group *grp* is created using the k_1st, k_2nd, ... elements found as generators.

In some groups, elements of a specified order may be very difficult to find using the quasi–random search technique employed by `fndelt`. For example, it is very difficult to find involutions in $\mathrm{PGL}_2(2^k)$ if k is fairly high (say 10 to 15). The `-m:`*trials* option may be used to terminate the command after *trials* random group elements have been generated, regardless of how many elements of the desired order have been found. By default, *trials* is essentially infinite.

Normal closures: The `ncl` command may be used to compute normal closures of subgroups. Specifically, given a group G and a subgroup H of G, the command computes the normal closure H^G of H in G. The command format is

`ncl` *options* *permGroup* *subgroup* *normal closure*

where *options* denotes

[`-a`] [`-i`] [`-mb:`*k*] [`-mw:`*w*] [`-n:`*name*] [`-p:`*path*] [`-q`]

The command sets *normalClosure* to the normal closure in *permGroup* of *subgroup*. (Note that *subgroup* must be a subgroup of *permGroup*; this is not checked.)

At present, the strong generating set for the normal closure is constructed only by the random Schreier method. Thus there is a (probably small) possibility that the strong generating set may be incorrect. (This will be fixed eventually; if it occurs, the generators obtained will generate the normal closure, but not strongly.)

Orbit structure: The `orblist` command performs various calculations relating to the orbit structure of a specified group, or to the orbit structure of point stabilizers within the group. The command format

`orblist` *options* *permGroup* [$p_1, p_2, \ldots, p_k$ [*ptStabGroup*]]

where *options* denotes

[`-a`] [`-i`] [`-len`] [`-lr`] [`-mb:`*k*] [`-mw:`*w*] [`-n:`*name*] [`-p:`*path*]
[`-ps:`*set*] [`-q`] [`-r`] [`-s:`*seed*] [`-wno:`*k*] [`-wo:`$p_1, p_2, \ldots$] [`-wp:`*partn*] [`-z`]

In the absence of any options, and with only one non–option parameter, the command writes (to standard output) the orbits of the group *permGroup*. For each orbit, the orbit representative, the orbit length, and the list of points in the orbit are written. If the `-r` option is given, the order in which the orbits are listed is randomized; the seed for the random number generator that is used may be specified by means of the `-s:`*seed* option. (Note that the `-r` option is ignored if the `-len` or `-lr` options are specified.)

If the `-len` option is specified, only the orbit lengths are written. If the `-lr` option is given, only the orbit representatives and orbit lengths are given; the output consists of a list of pairs *rep*`:`*len*, where *rep* is an orbit representative and *len* is the length of the corresponding orbit.

The `-wp:`*partn* may be used save the orbit partition of the group as the partition *partn*. The `-wo:`$p_1, p_2, \ldots$ option may be used to save the union of the orbits of points $p_1, p_2, \ldots$ as a point set; the name of the point set is the string *set* given by the `-ps:`*set* option. Alternatively, the `-wno:`*k* option causes the union of the first k orbits to be saved as a point set, whose name again is specified by the `-ps:`*set* option.

If a second non–option parameter $p_1, p_2, \ldots, p_k$ is specified, all orbit calculations are carried out in the stabilizer in *permGroup* of the point sequence $p_1, p_2, \ldots, p_k$, rather than in *permGroup* itself. Note that this entails a base change for the group *permGroup*. The `-z` option causes redundant strong generators for *permGroup* to be removed following this base change. Specifying a third non–option parameter *ptStabGroup* option causes point stabilizer of $p_1, p_2, \ldots, p_k$ to be saved as the permutation group *ptStab*.

Point stabilizers: The `ptstab` command may be used to compute point stabilizers in permutation groups. The command format is

`ptstab` *options* *permGroup* $p_1, p_2, \ldots, p_k$ *ptStabGroup*

where *options* denotes

[`-a`] [`-i`] [`-mb:`*k*] [`-mw:`*w*] [`-n:`*name*] [`-p:`*path*] [`-q`] [`-z`]

The point stabilizer in the group *permGroup* of the list $p_1, p_2, \ldots, p_k$ is computed and saved as the permutation group *ptStabGroup.* Note that, in the list $p_1, p_2, \ldots, p_k$, individual points are separated by commas but *not* by blanks. The `-z` option causes certain redundant strong generators for *ptStabGroup* to be removed before the group is saved.

Random point sets and partitions: The `randobj` command may be used to construct a random k–element subset of $\{1, \ldots, n\}$ for specified n and k, or to construct a partition of $\{1, \ldots, n\}$ that is random subject to the cells having specified sizes. The command format

`randobj` *options* *type* n $k_1, k_2, \ldots, k_p$ *newObject*

where *options* denotes

[`-a`] [`-e`] [`-n:`*name*] [`-s:`*seed*]

Here *type* must be either `set` or `partition`; it indicates whether a point set or partition is to be constructed. For a point set, p must be 1; a random k_1–element subset of $\{1, \ldots, n\}$ is constructed. For a partition, in the absence of the `-e` option, $k_1 + k_2 + \ldots + k_p$ must equal n; a partition of $\{1, \ldots, n\}$ random subject to having cell sizes k_1, k_2,..., k_p is constructed. However, if the `-e` option is specified, then p must equal 1, and a random partition having k_1 cells of equal size (as closely as possible) is constructed. In any case, the point set or partition constructed is saved as the object *newObject.*

The `-s:`*seed* option may be used to specify an initial seed for the random number generator that is used.

Weight distributions of codes: The `wtdist` command may be used to compute the weight distribution of a linear code. The program is highly optimized, both for binary and nonbinary codes. The command format is

`wtdist` *options* *code* [*saveWeight* *matrix*]

where *options* denotes

[`-a`] [`-b`] [`-g`] [`-n:`*name*] [`-p:`*path*] [`-pf:`*p*] [`-s:`*m*] [`-q`] [`-1`]

The weight distribution of the code *code* is computed and written to the standard output. If *saveWeight* (which should be a positive integer) and *matrix* are specified, the codewords of weight *saveWeight* are saved; specifically, a new matrix *matrix* is created whose rows are the vectors of weight *saveWeight* in the code *code.* However, for nonbinary fields, only one codeword from each one-dimensional subspace is saved. (Thus, for a code over GF(q) with k vectors of weight *saveWeight*, *matrix* will have $k/(q-1)$ rows.) Also, if the `-1` option is specified, only one codeword of weight *saveWeight* will be saved, and thus *matrix* will have only one row. In the event that the code has no codewords of the specified weight, the matrix is not created.

Four options, `-b`, `-g`, `pf:`p, and `s:`m, are never necessary but may be used to optimize performance. The `-s:`m option may be coded if codewords of weight *saveWeight* are to be saved, and if the number of such codewords is known in advance; the value of m should be the number of such codewords (excluding scalar multiples, for nonbinary fields). Use of this option saves some time and space. (If an incorrect value of m is specified, the savings in time and space may be lost, but the results will still be correct.)

To understand the other optimization options, it is necessary to understand that the `wtdist` command invokes one of two programs: a considerably optimized program for codes over any field, or an even more highly optimized one for binary codes meeting certain constraints on length and dimension. The binary code program requires a fixed amount of time (several seconds or more on a micro or workstation) to construct a certain table of size 65536, even if the dimension of the code is small. Accordingly, the `wtdist` command invokes the general code program for binary codes of low dimension; however, the criteria for choosing the cutoff point is relatively crude, and often a nonoptimal choice may be made. The `-g` option forces the general code program to be used, even if the binary one would be chosen by default. The `-b` option forces the binary code program to be used, provided the code is binary, provided its length is at most 128, and provided its dimension is at most 3. (If any of these conditions fail, the binary program cannot be used.)

The `-pf:`p option is applicable only if the general code program is employed. The value of p, which is referred to as the *packing factor*, should be a positive integer such that $q^p \leq 65535$. (Here q is the field size.) Internally, the program will pack p coordinates of each codeword into a single 16–bit word. Higher values of p improve performance on codes of high dimension. On the other hand, the size of the internal tables that must be allocated and initialized rise very rapidly as a function of p, and in particular, a value of p maximal subject to $q^p \leq 65535$ will be practical only with a rather large memory, and will be optimal with respect to time only for codes of quite high dimension, because of the large amount of time spent initializing the tables. (The size in bytes of the largest single table used internally is roughly $q^p ek\lceil n/p \rceil$, where n is the length, k is the dimension, and e is the exponent of the field.) The program will choose a packing factor by default, but at present the procedure for making this choice is crude. In particular, if the program runs out of memory, a smaller packing pactor should be specified.

X. THE UNIX DISTRIBUTION

On Unix, the programs, documentation, and examples will be available by anonymous ftp from `math.uic.edu`. The directory `pub/leon/partn` should contain the following files, where r denotes the release number, e.g. `1.00`:

```
doc- r.tar.Z
examples- r.tar.Z
src- r.tar.Z
bin16- r.sun4.tar.Z
bin16- r.sun3.tar.Z

bin32- r.sun4.tar.Z
bin32- r.sun3.tar.Z
```

(The last two files may be omitted to conserve disk space, but can be made available on request; contact the author at `leon@turing.math.uic.edu`.) Users with a Sun/4 should obtain the files `doc.tar.Z`, `examples.tar.Z`, and `bin16.sun4.tar.Z`, which contain, respectively, the documentation, the examples, and 16-bit executables for the Sun/4. Users desiring the C language source code (of limited use, due to inadequate documentation) should obtain the file `src.tar.Z`. Note that source code is not required since binary executables for the programs are supplied. Users needing to compute with groups of degree greater that 65000 (approximate) should obtain the file `bin32.sun4.tar.Z`, which contains 32-bit executables. Users with a Sun/3 merely substitute "`sun3`" for "`sun4`" in the preceding instructions. Other users should obtain only the files `doc.tar.Z`, `examples.tar.Z`, and `src.tar.Z`. It will be necessary to compile the source; a make file is provided for this purpose (See Section XI).

A directory, say `partn`, should be created, and all the files should be downloaded or moved to this directory. They should then be uncompressed and extracted by commands such as

```
uncompress -v doc.tar.Z
tar xvf doc.tar
```

(Repeat the above for each of the files.) The result should be subdirectories of `partn` as follows:

`doc` Documentation for the programs – primarily this manual.

`examples` The subdirectories of this directory contain examples. See Section VIII.

`test` Unix shell scripts to test the partition backtrack programs, using some of the examples provided in the `examples` subdirectory.

`src` Source code and a make file, to allow compilation of the programs on machines other than the Sun/3 and Sun/4.

`bin16` executable programs for the Sun/3 or Sun/4, using 16–bit integers.

`bin32` executable programs for the Sun/3 or Sun/4, using 32–bit integers.

For the Sun/3 or Sun/4, the appropriate directory `partn/bin16` or `partn/bin32` should be added to the path, or the files from one of those directories should be copied to a directory on the path. For other Unix machines, it will be necessary to recompile the source; see Section XI.

Once installation is complete, the programs may be tested using the shell scripts in the subdirectory `test`. Specifically, the following shell scripts, written for the Bourne shell, are provided:

```
test_setstab
test_cent
test_inter
test_desauto
test_setimage
test_conj
test_desiso
```

Each of the above shell files may be run, without options or parameters. When `test_setstab` is run, the output is collected in a file named `setstab.output`, as well as appearing on the screen. This file may be compared to the file `setstab.correct`, supplied in the subdirectory `tests`, which contains the correct output. The files should match exactly. The comparison may be performed, for example, with the command

```
diff setstab.output setstab.correct
```

The other shell files work in an analogous manner.

The tests above may take a number of hours, depending on the speed of the machine. In addition, they require several megabytes of memory. Each of the shell files accepts an option `-s`, which runs a much less time–taking series of tests, requiring less memory. When this option is used with `test_setstab`, the output in file `setstab.output` should be compared to the file `setstab-s.correct`, rather than to `setstab.correct`. A similar remark applies to the other tests.

The programs described in this manual are also available, in binary form, for the IBM PC and compatibles, under MS DOS, and for the IBM 370 family of machines, under CMS. For details, please contact the author.

XI. COMPILING THE SOURCE CODE

As mentioned earlier, compiled versions of the programs described here are available for the IBM PC (DOS), the Sun/3 and Sun/4 (Unix), and the IBM 370 (CMS). For other systems, it will be necessary to compile the C source code. The information in this section is intended for users intending to compile the source code.

To compile the C source code, a compiler that supports ANSI Standard C is required. With very minor exceptions, the source code conforms to the ANSI standard for C; it does, however, make use of a number of features not present in most pre–ANSI versions of C. The code was written under the assumption that it would be compiled with optimization turned on, so it is important to enable optimization, as the programs tend to run quite slowly if compiled with optimization disabled.[†] At present large sections of the source code are not commented or are commented incorrectly. Accordingly, the source is provided primarily so that users may compile the programs on other machines, rather than modify them. Eventually the author hopes to distribute adequately documented source code.

Prior to compiling the source, it is necessary to determine whether a special timing function needs to be supplied. By default the programs use the C standard library function `clock()` to measure execution times. This is adequate on many systems. However, on some systems (e.g., Sun/3 and Sun/4 Unix), a problem arises because the `clock()` function returns a result with a resolution of one microsecond and thus “wraps around” in about 36 minutes. Unless the user is content with timing statistics correct modulo 2^{32} microseconds (about 71.58 minutes), an alternate timing function must be supplied in a file which should be named `cputime.c`; this function must return a value of type `long` which represents the

[†] However, some compilers fail to optimize the code correctly. GNU C 1.39 and 2.1 (Sun/3, Sun/4) and Waterloo C 3.2 (IBM 370) optimize it correctly; at present, Borland C++ 3.0 and Microsoft C 5.1 do not. Microsoft C 6.0/7.0 and Borland C++ 3.1 have not been tested.

elapsed CPU time, and C macros CPU_TIME and TICK must be defined to specify the name of this special function (e.g., cpuTime) and the resolution of the function (in clock ticks per second), respectively. In addition, a make file macro CPUTIME must be defined to have the value cputime.o (assuming object code files have suffix o). These macros are discussed in more detail below. For the Sun/3 and Sun/4, source code for such a function cpuTime() is supplied in the file cputime.c; perhaps this function will work on other Unix systems as well. In the remainder of this section, it is assumed that such a function, if needed, has already been written.

A make file (named Makefile) is provided with the source. This make file is designed for the GNU C compiler, version 2, on the Sun/3 and Sun/4, but with some other compilers and systems, only simple editting of the first few lines of the make file should be needed; the purpose of these lines is to define appropriate make file macros. For some compilers, more extensive editting of the make file may be required. Note that the make file assumes that all source code and include files (except system include files) are in the currect directory, and that all object and executable files are to be placed in this directory.

The make file provided with the source begins as follows:

```
COMPILE = gcc
DEFINES = -DSUN_UNIX_GCC -DINT_SIZE=32 -DCPU_TIME=cpuTime -DTICK=1000
          -DLONG_EXTERNAL_NAMES -Dclock_t=long
INCLUDES =
COMPOPT = -c -O2 -Wall
LINKNAME = -o
LINKOPT = -v
OBJ = o
CPUTIME = cputime.o
BOUNDS =
```

The purpose of the make file macros defined here is as follows:

COMPILE: This macro specifies the command to invoke the C compiler; it may include path information.

DEFINES: This make file macro specifies C macros to be defined for the compilation; these are discussed below.

INCLUDES: This macro is used to tell the compiler where to search for include files, if they are located other than in the standard location.

COMPOPT: This macro is used to specify compile–time options, other than those given by means of the DEFINES and INCLUDES macros. These options should include code optimization and compile–only (no linking) if these are not defaults. For the IBM PC, an option specifying the large memory model should be given.

LINKNAME: This macro is used to specify the linker option used to give the name of the executable file to be created.

LINKOPT: This macro is used to specify linker options, other than that given by the LINKNAME macro above.

OBJ: This macro is used to specify the suffix (file type) for object files. Normally this would be o on Unix and obj on MS DOS.

CPUTIME: Unless a special timing function is to be supplied (see discussion above), this value of this macro should be the null string. If a special timing function is supplied, the value of the CPUTIME macro should be cputime.o, assuming object files have suffix o.

BOUNDS: This macro is present for use on MS DOS with Bounds Checker, a debugging product published by Nu-Mega Technologies and used heavily by the author in debugging the programs. Normally its value should be the null string.

It remains to discuss the C language macros that must, or may, be defined by the make macro DEFINES. One of these C macros, INT_SIZE, always must be defined; the others are optional, or are required only in certain circumstances. The C language macros are as follows:

INT_SIZE: This macro is always required. Its value must be the number of bits in the C data type int, usually 16 or 32. (The programs have never been adapted to a system in which the integer size is other than 16 or 32, although it should not be difficult to do so.) *Note:* This macro only specifies the size of the C data type int; it does *not* determine whether the programs are compiled to use 16 or 32 bit integers.

EXTRA_LARGE: If this macro is defined, and INT_SIZE above is 32, the programs will be compiled using 32–bit integers; that is, points will be represented using the C data type unsigned. Otherwise, if INT_SIZE is 32, the programs will be compiled using 16–bit integers (with most compilers), as points will be represented using the C data type unsigned short. Note EXTRA_LARGE should not be defined when INT_SIZE is 16. Care must be taken, of course, not to link object files compiled with EXTRA_LARGE defined with those compiled without it defined, as the make file cannot protect against this error. (However, running the programs with the -v option will reveal this error.)

LONG_EXTERNAL_NAMES: This macro should (but need not) be defined if the linker supports long external names (up to 31 characters). If defined, its value is irrelevent.

CPU_TIME: This macro must be defined if a special timing function is supplied, and its value must be the name of that function. If the standard C function clock() is to be used, the macro must not be defined (not even as the null string).

NOFLOAT: This macro probably should be defined on a system lacking floating point hardware support. (If defined, its value is irrelevant.) Normally, the programs perform a small amount of floating point arithmetic in attempting to produce a good $\underline{\mathbf{R}}$–base; with software emulation of floating point instructions, the cost of this use of floating point arithmetic probably exceeds its benefit. If the NOFLOAT macro is defined, no floating point arithmetic is used. (In this case, it may be advantageous to add a compiler option telling the compiler not to incorporate floating–point support into the executable program. For example, under Borland C++, the -f- option has this effect.)

TICK: This macro should be defined in two situations: (1) If a special CPU time function is supplied, TICK should be defined to be the resolution (in clock ticks per second) of that function, and (2) if the NOFLOAT macro is defined and if the compiler–supplied definition of the standard C macro CLK_TCK is a floating point constant (as occurs with Borland C++), TICK should be defined to be the nearest integer approximation to the value of CLK_TCK, e.g., 18 for Borland C++.

`HUGE`: This macro must be defined for the IBM PC (unless a DOS extender is being used). It simply causes a few pointers to be declared as `huge`. Only one program, the weight distribution program, uses huge pointers.

In addition, the author recommends defining a symbol unique to the system and compiler, e.g., `SUN_UNIX_GCC` above. Then any code modifications unique to this system and compiler can be bracketed as follows:

```
#ifdef SUN_UNIX_GCC
   /* Code modifications for Sun Unix, GNU C compiler. */
#endif
```

Once the make file has been editted appropriately, all of the programs may be compiled at once by the command

```
make all
```

or, alternatively, individual programs may be compiled by the commands

`make setstab`	for `setstab`, `setimage`, `parstab`, and `parimage`,
`make cent`	for `cent`, `conj`, `gcent`,
`make inter`	for `inter`,
`make desauto`	for `desauto`, `desiso`, `matauto`, `matiso`, `codeauto`, and `codeiso`,
`make cjrndper`	for `cjrndper` and `cjper`,
`make commut`	for `commut` and `ncl`,
`make compgrp`	for `compgrp`,
`make fndelt`	for `fndelt`,
`make generate`	for `generate`,
`make orblist`	for `orblist`, `chbase`, and `ptstab`,
`make randobj`	for `randobj`,
`make wtdist`	for `wtdist` and `cwtdist`.

Ideally, there should be no errors in compilation. However, with some compilers (e.g., Zortech C++), it is necessary to define `const` to be a null string in order to avoid compile–time errors. Typically compilers will generate a number of warnings; a number of them involve implicit conversion between pointers to constant and non–constant types.

The above commands place the executable programs in the current directory (that containing the source). The final step is to move the executables to the directory in which they should reside, preferably one on the current path. A shell file named `install` is provided for this purpose. The command

`sh install` *bindir*

should be issued, where *bindir* is the name of the directory in which the binary files should be placed, e.g., `../bin`. Note that this command also copies certain shell files from the source directory to the binary directory, renaming them by dropping the `sh` suffix in the process.

REFERENCES

Leon, J. (1980a), On an algorithm for finding a base and a strong generating set for a group given by generating permutations, *Math. Comp.* **35**, 941–974.

Leon, J. (1991), Permutation group algorithms based on partitions, I: Theory and algorithms, *J. Symbolic Comp.* **12**, 533–583.

Cannon, J. (1984), *A language for group theory,* Dept. of Pure Mathematics, University of Sydney, Australia.

Sims, C. C. (1971), Computation with permutation groups, In (Petrick, S. R., ed.), *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation,* New York: Assoc. for Computing Mach.