Computational Abstract Algebra:

Using Monomial Matrices to Represent Groups in GAP

By

Zachary Robert Rome

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Mathematics

THE UNIVERSITY OF ARIZONA

MAY 2012

Approved by:

Dr. Klaus Lux
Department of Mathematics

Computational Abstract Algebra : Using Monomial Matrices to Represent Groups in GAP

Zachary Robert Rome

May 2012

Abstract

A monomial matrix is a matrix with exactly one non-zero element in each row and column. We will utilize GAP to construct all (transitive) representations of a given group using monomial matrices. First, essential group theory definitions and theorems will be provided, as well as an in-depth look at table of marks and monomial matrices. After describing the necessary mathematics, we will explore the GAP programming needed to achieve this goal. Ultimately we want a table, where each row represents a subgroup of the given group and, within the row, the table will hold the linear characters fixed by the monomial matrices of that subgroup. We will furthermore explore how to represent monomial matrices computationally in different ways and how to create a data structure to represent them. Our final goal will require GAP functions for finding all homomorphisms from a subgroup to roots of unity, using these homomorphisms to create monomial matrix representations of the group, and iterating through the subgroups of the group (up to conjugacy) to find all (transitive) monomial matrix representations of the group.

Contents

1 Groups

1.1 Definitions

First we will set up some necessary definitions of group theory in order to introduce the syntax of the rest of the paper. These can be found in *A First Course In Abstract Algebra* [2] and *Contemporary Abstract Algebra* [3].

A group G is a set, along with a binary operation, call it $*$, such that:
(1) For all a,b, and c in G, $a * (b * c) = (a * b) * c$,
(2) There exists an identity element e such that for all a in G, $e*a = a = a*e$, and
(3) For all a in G, there exists an inverse a^{-1} such that $a*a^{-1} = e = a^{-1}*a$, where e is the identity element of G.

A subgroup, H, of a group, G, with binary operation $*$ is a non-empty subset of G that is a group with $*$ (or in other words H is closed under $*$ and for all $h \in H$ the inverse of h is also in H).

A coset of a subgroup H in its group G, gH, is the set of all elements $g * h$, where g is a given element of G and h is any element of H.

A coset representative for gH is any g' in G such that $g'H = gH$.

The order of a group G, $|G|$, is the number of elements in the group.

The index of a subgroup H in a group G, $[G:H]$, is the number of cosets of H in G.

A group action, $\cdot : G \times X \to X$, where G is a group (with binary operation $*$) and X is a set, is a binary operation such that:

(1) For all x in X, $e \cdot x = x$, where e is the identity element of G, and

(2) for all g, h in G and all x in X, $g \cdot (h \cdot x) = (g * h) \cdot x$.

This X is called a G-set.

Example: Suppose G is a group and H is a subgroup of G. Let X be the set of cosets of H ($\{gH | g \in G\} = G/H$). Then X is a G-set.

Proof: Consider arbitrary elements $a \in G$ and $gH \in X$. Then we define the action on the coset to be $a \cdot gH = (ag)H = g'H \in X$ since $g' = (ag) \in G$. Let e be the identity element of G. Then $e \cdot gH = (eg)H = gH$. Finally, consider $a, b \in G$. Then $a \cdot (b \cdot gH) = a \cdot ((bg)H) = (abg)H = (ab) \cdot gH$. Thus X is a G-set.

A transitive G-set, X, is a G-set such that for every $x_1, x_2 \in X$, there exists $g \in G$ such that $gx_1 = x_2$. Note that the cosets G/H is a transitive G-set.

The stabilizer of an element x in G-set X, G_x or $Fix_G(x)$, is the set of all elements g in group G such that $g \cdot x = x$.

A subgroup N of a group G is normal if for all $n \in N$, for all $g \in G$, $gng^{-1} \in N$.

The normalizer of a subgroup H of a group G, $N_G(H)$, is the set of all $a \in G$ such that $aHa^{-1} = H$.

The orbit of element x of a G-set X under a group G, $Orb_G(x)$, is the set of $g \cdot x$ for all g in G.

Subgroups H and K of a group G are conjugate if there exists some g in G such that $gHg^{-1} = K$.

A commutator of elements g and h of a group G, $[g, h]$, is $g^{-1}h^{-1}gh$. Note that $[g, h]^{-1} = [h, g]$.

The commutator subgroup of a group is the subgroup generated by all commutators of the group.

1.2 Theorems

Now we will provide some basic theorems that use the above definitions. These can also be found in *A First Course In Abstract Algebra* [2] and *Contemporary Abstract Algebra* [3].

Given a group G, a subgroup H of G, and a subgroup K of H, $[G : K] = [G : H][H : K]$.

(Lagrange's Theorem) Given a group G and a subgroup, H, of G, $|G| = [G : H]|H|$.

Given a group G, a subgroup H of G, and a subgroup K of H, if $g_1, ..., g_n$ are the coset representatives of H in G and $h_1, ..., h_m$ are the coset representatives of K in H, then the set of $g_i h_j$, where $1 \leq i \leq n$ and $1 \leq j \leq m$, are the coset representatives of K in G.

(Orbit-Stabilizer Theorem) Given a group G and a G-set X, for all $x \in X$, $|Orb_G(x)| = [G : Fix_G(x)] = |G|/|Fix_G(x)|$.

Any G-set is the disjoint union of transitive G-sets.

Any transitive G-set is equivalent to $G/Stab_G(x)$ for any $x \in X$ by the Orbit-Stabilizer Theorem.

(Fundamental Theorem of Finite Abelian Groups) Every finitely generated abelian group, G, is isomorphic to a direct sum of primary cyclic groups and infinite cyclic groups. In other words, $G = \mathbb{Z}^n \oplus \mathbb{Z}_{q_1} \oplus ... \oplus \mathbb{Z}_{q_k}$, where $q_1, ..., q_k$ are prime powers and $n = 0$ if G is finite.

Corollary: G can also be written as $G = \mathbb{Z}^n \oplus \mathbb{Z}_{c_1} \oplus ... \oplus \mathbb{Z}_{c_k}$ where k_i divides k_{i+1} for every $1 \leq i \leq k - 1$.

2 Table Of Marks

2.1 Creating A Table Of Marks

To create a table of marks for some group G we start off with a square matrix. Label the rows and columns with the subgroups of G up to conjugacy, $G_1, ..., G_n$, including the entire group and the identity group (the group containing only the identity of G). Each entry, $[i, j]$, in the table is then determined by how many cosets G/G_i are fixed by G_j. In other words, $[i, j] = |Fix_{G_j}(G/G_i)|$, where $Fix_{G_j}(X)$ is the set of all elements of X fixed by all elements of G_j (the intersection of $Fix_g(X)$ for all g in G_j).

2.2 Example : The Symmetric Group S_3

Below we show the table of marks for the symmetric group on three elements.

$$\begin{array}{c|cccc} & <()> & <(12)> & <(123)> & S_3 \\ \hline <()> & 6 & 0 & 0 & 0 \\ <(12)> & 3 & 1 & 0 & 0 \\ <(123)> & 2 & 0 & 2 & 0 \\ S_3 & 1 & 1 & 1 & 1 \end{array}$$

Now we will calculate the entry at position $[2, 2]$ of the above table. Let us denote S_3 to be the symmetric group on three elements. Let H be the subgroup of S_3 generated by (123), or the label of the second row and column. Then the entry at position $[2, 2] = |Fix_H(S_3/H)|$, where $S_3/H = H, (12)H = \{\{(), (123), (132)\}, \{(12), (13), (23)\}\}$. Now we must check which cosets, S_3/H, are fixed by (), (123), and (132).

$$()H = ()(), ()(123), ()(132) = (), (123), (132) = H$$

$$()(12)H = ()(12), ()(13), ()(23) = (12), (13), (23) = (12)H$$

Thus () fixes S_3/H.

$$(123)H = (123)(), (123)(123), (123)(132) = (123), (132), (123) = H$$

$$(123)(12)H = (123)(12), (123)(13), (123)(23) = 13, (23), (12) = (12)H$$

Thus (123) fixes S_3/H.

$$(132)H = (132)(), (132)(123), (132)(132) = (132), (), (123) = H$$

$$(132)(12)H = (132)(12), (132)(13), (132)(23) = (23), (12), (13) = (12)H$$

Thus (132) fixes S_3/H.

Since all elements of H fix both cosets S_3/H, $[2, 2] = |Fix_H(S_3/H)| = 2$, as can be seen in the above table.

Note that $Fix_H(G/H) = \{gH | g \in N_G(H)\} = [N_G(H) : H] = |N_G(H)|/|H|$. And from this we can derive $|N_G(H)|$.

3 Monomial Matrices

3.1 Definition

A monomial matrix is a matrix with exactly one non-zero entry in each row and column. This is a generalization of a permutation matrix in which the non-zero entries can be scalars other than just 1.

Remember that any G-set X is a disjoint union of transitive G-sets. Furthermore, any transitive G-set X is congruent to the G-set $G/Stab_G(x)$ for any $x \in X$.

Theorem : Every monomial matrix can be written as the product of a permutation matrix (a matrix with exactly one entry 1 in each row and column and zeros elsewhere) and a diagonal matrix (a matrix whose only non-zero entries are on the diagonal).

Theorem : All monomial matrices of $GL_n(F)$, the group of $n \times n$ invertible matrices with entries in the field F, form a subgroup of $GL_n(F)$, denoted by $M_n(F)$.

For our purposes, we need only consider the invertible matrices with entries in the complex numbers.

Proof. We will prove that all monomial matrices, or matrices with exactly one non-zero entry in each row and column, of $GL_n(F)$ form a subgroup. In order to prove this, we must show that the identity matrix of $GL_n(F)$ is a monomial matrix, that monomial matrices are closed under matrix multiplication, and that the inverse of each monomial matrix is a monomial matrix.

First, it is true by definition that the identity matrix is a monomial matrix because there is exactly one non-zero entry in each row and column.

Secondly, we will prove that monomial matrices are closed under matrix multiplication. Suppose that X and Y are monomial matrices of $GL_n(F)$. We must prove that XY is a monomial matrix in $GL_n(F)$. Consider the fact that Z_{ij} of XY is given by $X_{i1}Y_{1j} + X_{i2}Y_{2j} + \ldots + X_{in}Y_{nj}$, where $X_{i1}, \ldots, X_{in}$ is row i of X and $Y_{1j}, \ldots, Y_{nj}$ is column j of Y (that is, $0 \leq i, j \leq n$).

Assume Z_{ia} and Z_{ib} (two elements of the same row in XY) are both non-zero, where $0 \leq a, b \leq n$. Because each row and column of X and Y has one element, $Z_{ia} = X_{im}Y_{mj}$ and $Z_{ib} = X_{ip}Y_{pj}$ (all other pairs would be zero). However, X and Y are monomial matrices, so the only way that Z_{ia} or Z_{ib} are non-zero is if $m = p$. Thus $a = b$ if Z_{ia} and Z_{ib} are both non-zero and there is only one element in each row that is non-zero.

Similarly, assume Z_{aj} and Z_{bj} (two elements of the same column of XY) are both non-zero, where $0 \leq a, b \leq n$. Because each row and column of X and Y has one element, $Z_{aj} = X_{mj}Y_{im}$ and $Z_{bj} = X_{pj}Y_{ip}$ (all other pairs would be zero). However, X and Y are monomial matrice, so the only way that Z_{aj}

or Z_{bj} are non-zero is if $m = p$. Thus $a = b$ if Z_{aj} and Z_{bj} are both non-zero and there is only one element in each column that is non-zero. Thus monomial matrices of $GL_n(F)$ are closed under matrix multiplication.

Finally, we need to prove that every monomial matrix has an inverse. We know that every such matrix can be written as the product of a permutation matrix and a diagonal matrix. Suppose we have monomial matrix $M = P \cdot D$, where P is a permutation matrix and D is a diagonal matrix. We want to find M^{-1} such that $M \cdot M^{-1} = I$, where I is the identity (monomial) matrix. However, since $M = P \cdot D$, $P \cdot D \cdot M^{-1} = I$ and thus $M^{-1} = P^{-1} \cdot D^{-1}$, where P^{-1} is the inverse of P, which exists because every permutation matrix has an inverse, and D^{-1} is the inverse of D, which exists because we are only looking at monomial matrices of $GL_n(F)$ and thus D has no non-zero entries on the diagonal and is invertible. Thus $P^{-1} \cdot D^{-1}$ exists so $M^{-1} = P^{-1} \cdot D^{-1}$ exists and $M \cdot M^{-1} = P \cdot D \cdot P^{-1} \cdot D^{-1} = I$ so M is invertible.

Therefore monomial matrices of $GL_n(F)$ are a subgroup of $GL_n(F)$.

$\square$

3.2 Representing A Group With Monomial Matrices

Now we represent a group, G, using monomial matrices. We do this by defining a homomorphism $\psi : G \to M_n(\mathbb{C})$, where each $\psi(g)$ is a monomial matrix. There will be different monomial matrix representations for G using each subgroup H, described as follows. First we choose coset representatives $S = \{g_1, ..., g_n\}$, where $n = [G : H]$. Thus for all elements g of G and $g_i \in S$, $gg_i = g_j h$ for some $g_j \in S$ and some $h \in H$. Next we choose a homomorphism, φ, from H to $R(\mathbb{C}) = \{e^{(2\pi ik)/n} | 0 \leq k \leq n, n > 1, n \in \mathbb{Z}, k \in \mathbb{Z}\}$. For each element of $g \in G$, it's monomial matrix will have rows and columns labelled $1, ..., n$ and row i will have an entry of $\varphi(h)$ at j, where $gg_i = g_j h$. In other words, $\psi(g)_{ij} = \varphi(h)\delta_{ij}$, where δ_{ij} is 1 if g_i and g_j hold in the previous equation and 0 otherwise. Note that we are creating monomial matrices using every subgroup, H, of G and every homomorphism from H to $R(\mathbb{C})$, the subgroup of $\mathbb{C}^*$.

3.3 Example : The Symmetric Group S_3

Now let us look at an example of monomial matrix representations of the symmetric group on three elements, S_3:

3.3.1 Subgroup $< (123) >$

First we will look at the subgroup $< (123) >$. The cosets of $S_3/ < (123) >$ can be represented by () and (12). Then,

$$(123)() = ()(123)$$

$$(123)(12) = (13) = (12)(132)$$

$$(12)() = (12)()$$
$$(12)(12) = () = ()()$$

Assume that the first row and column is labelled by () and the second row and column is labelled by (12).

Thus if our homomorphism is $\varphi(h) = 1$ for all $h \in H$, then our monomial matrix representations for $g = (123)$ is:

$$\begin{array}{cc} & \begin{matrix} () & (12) \end{matrix} \\ \begin{matrix} () \\ (12) \end{matrix} & \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{array}$$

And our monomial matrix representation for $g = (12)$ is:

$$\begin{array}{cc} & \begin{matrix} () & (12) \end{matrix} \\ \begin{matrix} () \\ (12) \end{matrix} & \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \end{array}$$

We can also use the homomorphism $\varphi(()) = 1$, $\varphi((123)) = E(3)$, and $\varphi((132)) = E(3)^2$, where $E(3)$ is $e^{(2\pi i)/3}$. Then our monomial matrix for $g = (123)$ is:

$$\begin{array}{cc} & \begin{matrix} () & (12) \end{matrix} \\ \begin{matrix} () \\ (12) \end{matrix} & \begin{pmatrix} E(3) & 0 \\ 0 & E(3)^2 \end{pmatrix} \end{array}$$

And our monomial matrix for $g = (12)$ would be the same (because $\varphi(())$ is still 1).

Finally, we can use the homomorphism $\varphi(()) = 1$, $\varphi((123)) = E(3)^2$, and $\varphi((132)) = E(3)$. Then our monomial matrix representation for $g = (123)$ is:

$$\begin{array}{cc} & \begin{matrix} () & (12) \end{matrix} \\ \begin{matrix} () \\ (12) \end{matrix} & \begin{pmatrix} E(3)^2 & 0 \\ 0 & E(3) \end{pmatrix} \end{array}$$

And again our monomial matrix for $g = (12)$ is the same.

3.3.2 Subgroup $< (12) >$

Next we will look at the subgroup $< (12) >$. The cosets of $< (123), (12) > / < (12) >$ can be represented by (12), (23), and (13). Then,

$$(123)(12) = (13)()$$

$$(123)(23) = (12) = (12)()$$

$$(123)(13) = (23) = (23)()$$

$$(12)(12) = (12)(12)$$

$$(12)(23) = (123) = (13)(12)$$

$$(12)(13) = (132) = (23)(12)$$

Assume that the first row and column is labelled by (12), the second row and column is labelled by (23), and the third row and column is labelled by (13).

Thus if our homomorphism is $\varphi(h) = 1$ for all $h \in H$, then our monomial matrix representations for $g = (123)$ is:

$$\begin{array}{c c} & \begin{array}{ccc} (12) & (23) & (13) \end{array} \\ \begin{array}{c} (12) \\ (23) \\ (13) \end{array} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

And our monomial matrix representation for $g = (12)$ is:

$$\begin{array}{c c} & \begin{array}{ccc} (12) & (23) & (13) \end{array} \\ \begin{array}{c} (12) \\ (23) \\ (13) \end{array} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

We can also use the homomorphism $\varphi(()) = 1$ and $\varphi((12)) = -1$. Then our monomial matrix representation for $g = (123)$ is still:

$$\begin{array}{c c} & \begin{array}{ccc} (12) & (23) & (13) \end{array} \\ \begin{array}{c} (12) \\ (23) \\ (13) \end{array} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

And our monomial matrix representation for $g = (12)$ is now:

$$\begin{array}{c} \\ (12) \\ (23) \\ (13) \end{array} \begin{array}{c} \begin{array}{ccc} (12) & (23) & (13) \end{array} \\ \left(\begin{array}{ccc} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{array}\right) \end{array}$$

This method should be continued with the identity subgroup, $<()>$, and the full group as a subgroup, $<(123),(12)>$, using every subgroup, and every homomorphism from the subgroup to $R(\mathbb{C})$, in order to fully represent the group with monomial matrices. However, the above should provide enough of an example on how to create monomial matrix representations of a group, satisfying the homomorphism $\psi : G \to M_n(\mathbb{C})$, where $M_n(\mathbb{C})$ is the subgroup of $GL_n(\mathbb{C})$ containing all $n \times n$ monomial matrices.

4 GAP

4.1 Creating A Data Structure For Monomial Matrices

Now we will use GAP to represent groups using monomial matrices. First we must create a data structure to represent the monomial matrices in GAP. Among other things, it is important that we can multiply monomial matrices, find the inverse of a monomial matrix, and generate an identity monomial matrix in GAP (similar to the case of G-sets, only monomial matrix representations of a group can be written as a sum of the ones I have described).

4.1.1 Using A Full Matrix

The first thought would be to store the monomial matrix as a list of lists representing the matrix. This is implemented with the following functions:

```
# checkMonomial takes a list of lists and makes sure it is a
#    monomial matrix

checkMonomial := function(mono)

local size, seen, toFill, found, colnum, row, col;

size:= Length(mono);
seen:= [];
```

```
toFill := 1;

while toFill <= size do

    seen[toFill] := 0;
    toFill := toFill+1;

od;

for row in mono do

    # check that row is the correct length
    if not (Length(row) = size) then return false;
    fi;

    found := false;

    colnum := 0;

    #check that we have exactly one non-zero entry in each
    for col in row do

        colnum := colnum+1;

        # if we found the non-zero entry
        if not (col = 0) then
            if found then return false;
            else found := true;
            fi;

            if seen[colnum] = 1 then return false;
            else seen[colnum] := 1;
            fi;
        fi;
    od;
od;

return true;

end;
```

As is described in the code, $checkMonomial$ takes a list of lists and determines if it represents a monomial matrix. First we zero out the list *seen*, making it the same length as the given list (the number of rows in the matrix). Then we iterate through each row. First we check to see if the row is long enough. If not, we immediately know this is not a monomial matrix. Then we iterate through each element of the row, noting which column we are currently looking at. Once we find a non-zero element we enter the final loop. The first if statement here determines if we have already found a non-zero element in this row. If so, we know this is not a monomial matrix and return false. The second if statement determines if we have already found a non-zero element in this column by checking *seen*. If *seen*[*colnum*] is 0 then we have not and we mark *seen* accordingly by changing it to 1 and move on. Otherwise, this is the second non-zero element in this column and we return false, showing that what is given is not a monomial matrix.

```
# multiplyMonomials takes two monomial matrices and multiplies them

multiplyMonomials := function(mono1,mono2)

local r, c, result, entry, total;

if not checkMonomial(mono1) then return false;
elif not checkMonomial(mono2) then return false;
elif not (Length(mono1) = Length(mono2)) then return false;
else;
fi;

r := 1;
c := 1;
result := [];

#go through each row of the first monomial matrix
while r <= Length(mono1) do

    c := 1;

    while c <= Length(mono2) do

        entry := 1;

        total := 0;

        while entry <= Length(mono1) do

            total := total + (mono1[r][entry]*mono2[entry][c]);
            entry := entry+1;

        od;
        if c=1 then result[r] := [];
        fi;
        result[r][c] := total;

        c := c+1;

    od;

    r := r+1;

od;

return result;

end;
```

Here is how we multiply two monomial matrices represented in this way. The first thing that we do is check that both given monomial matrices are in fact monomial matrices. The above is the basic algorithm for multiplying two matrices, where r represents the current row of the first matrix we are looking at, c represents the current column of the second matrix, and *entry* iterates over every element of each of these. Of course, GAP supports straightforward multiplication of matrices, so $mono1 * mono2$ could have also sufficed.

```
# generateIdentity generates an identity matrix of size n x n

generateIdentity := function(n)

local row, col, result;

row := 1;
col := 1;
result := [];

while row <= n do

    col := 1;

    while col <= n do

        if col = 1 then result[row] := [];
        fi;
        if row = col then result[row][col] := 1;
        else result[row][col] := 0;
        fi;

        col := col+1;

    od;

    row := row+1;

od;

return result;

end;
```

Now we need to generate an identity matrix. The above goes through one row at a time and makes the element 1 if the column, *col*, we are at is equal to the row number, *row*, and 0 otherwise.

```
# monomialInverse gives us the inverse of a
#    monomial matrix

monomialInverse := function(mono)

if not checkMonomial(mono) then return false;
fi;

return mono^-1;

end;
```

Now we need to find the inverse of a monomial matrix. The above code checks that the given object is a monomial matrix and then uses the inverse function already supported by GAP to return the inverse of the matrix.

4.1.2 Storing Only Column And Scalar

The above function will correctly support a data structure for monomial matrices in GAP. However, each object takes up much more space than is necessary. Note that the majority of the matrices are 0's. In fact, as we know, each row has only one non-zero entry. Thus we need only store what that entry is and which column it is located in to retain all necessary information about the monomial matrix. The next set of functions again utilize a list of lists to represent the monomial matrix, but now each inner list only stores the column number of the non-zero entry and the entry itself. Note that each inner list has a constant two elements instead of the arbitrary amount of elements needed for the last representation, putting the space needed at $O(n)$ instead of $O(n^2)$ for each monomial matrix.

```
# checkMonomial takes a list of lists and makes sure it is a
#    monomial matrix

checkMonomial := function(mono);

local size, seen, toFill, row;

size := Length(mono);
seen := [];

toFill := 1;

while toFill <= size do

    seen[toFill] := 0;
    toFill := toFill+1;

od;

for row in mono do

    if row[1] > size then return false;
    elif seen[row[1]] = 1 then return false;
    elif row[2] = 0 then return false;
    else seen[row[1]] := 1;
    fi;

od;

return true;

end;
```

The above function checks if the given object is a monomial matrix. It makes all the same checks as the previous function, but quicker because of the new implementation. For each row we check if the column number is possible, if there has already been a non-zero element in that same column, and if the given scalar entry is non-zero. If all of these conditions are met, then we note that we have seen a non-zero entry in the given column and check the next row.

```
# multiplyMonomials takes two monomial matrices and multiplies them

multiplyMonomials := function(mono1,mono2);

local result, r, row;

#check both matrices
if not checkMonomial(mono1) then return false;
elif not checkMonomial(mono2) then return false;
elif not (Length(mono1) = Length(mono2)) then return false;
else;
fi;

# result will hold the matrix representation
result := [];
# r will hold the current row number
# (of first matrix and thus of resulting matrix)
r := 1;

# go through first matrix
for row in mono1 do

    result[r] := [];

    # this is the corresponding row in the second matrix
    row2 := mono2[row[1]];

    result[r][1] := row2[1];
    result[r][2] := (row2[2] * row[2]);

    r := r+1;

od;

return result;

end;
```

Again, while multiplying two monomial matrices, we first check if both objects are in fact monomial matrices. We need only multiply each row in the first monomial matrix with the entry of the row in the second monomial matrix that corresponds to the column of the non-zero entry in the row of the first monomial matrix because the rest of the entries in the row are 0. Knowing this, we get the corresponding column from the column in the row of the second monomial matrix and we get the entry by multiplying the entries of the two rows we are looking at. Clearly this *multiplyMonomials* function is more efficient than the previous one.

```
# generateIdentity generates an identity matrix of size n x n

generateIdentity := function(n)

local curr, result;

curr := 1;
result := [];

while curr <= n do

    result[curr] := [];
    result[curr][1] := curr;
    result[curr][2] := 1;

    curr := curr+1;

od;

return result;

end;
```

In order to generate the identity monomial matrix with this representation, we iterate from 1 to n, where n is the size of the matrix, and set the column number to this row number and the entry to 1.

```
# monomialInverse gives us the inverse of
#    a monomial

monomialInverse := function(mono)
```

```
local row, i, j, result;

if not checkMonomial(mono) then return false;
fi;

result := [];

# Create a full matrix
for i in [1..Length(mono)] do

row := [];

# Zero out the row
for j in [1..Length(mono)] do

row[j] := 0;

od;

# Set the non-zero entry
row[mono[i][1]] := mono[i][2];


# Append to result
Append(result, [row]);

od;

return result^-1;

end;
```

In adding to the previous function, the current implementation of $monomialInverse$ makes the monomial matrix into a full square matrix and uses GAP's inverse function.

4.1.3 Storing A Permutation And List Of Scalars

The final representation of monomial matrices is slightly more efficient and provides more utility. We will use the fact that all monomial matrices can be rewritten as the product of a permutation matrix and a diagonal matrix. Hence, we will store the permutation as one list and the scalars of the diagonal matrix as another list for each monomial matrix.

```
# checkMonomial takes a list of lists and makes sure it is a
#    monomial matrix

checkMonomial := function(mono)

local col, coeff, size, seen, toFill;

#check that the format is correct
if not (Length(mono[1]) = Length(mono[2])) then return false;
fi;

size := Length(mono[1]);
seen := [];
toFill := 1;

# fill seen with 0's
while toFill <= size do

    seen[toFill] := 0;
    toFill := toFill+1;

od;

#check that the permutation makes sense
for col in mono[1] do

    if col > size then return false;
    elif seen[col] = 1 then return false;
    else seen[col] := 1;
    fi;

od;

# check that the coefficients make sense
for coeff in mono[2] do

    if coeff = 0 then return false;
    fi;

od;

return true;
```

```
end;
```

In order to check if a given monomial matrix is correctly in the aforementioned representation, we first check that the length of the permutation list and scalar list are the same. If this were not the case, the matrices that these represent could not be multiplied. Then we check that the permutation list is accurate by checking that each element in the list is within the bounds of the size of the matrix and that we do not repeat anything in the permutation list. This is done in the same fashion as before. Finally, we check that the coefficients are all accurate by ensuring they are all non-zero.

```
# multiplyMonomials takes two monomial matrices and multiplies them

multiplyMonomials := function(mono1,mono2)

local perm, result, row, inverse;

# check monomial matrices
if not checkMonomial(mono1) then return false;
elif not checkMonomial(mono2) then return false;
elif not (Length(mono1) = Length(mono2)) then return false;
else;
fi;

result := [[],[]];
row := 1;

perm := PermList(mono1[1]) * PermList(mono2[1]);

while row <= Length(mono1[1]) do

    # get permutation entry
    Append(result[1], [row^perm]);

    # get coefficient entry
    Append(result[2],
        [mono1[2][row]*mono2[2][row^PermList(mono1[1])]]);

    row := row+1;

od;
```

```
return result;

end;
```

As with the previous *multiplyMonomials* functions, we first check that the given objects are actually monomial matrices. Next we find the permutation of the result by changing each of the given lists representing permutations into actual permutations and finding the product. Then we go through each row and do the following: find the column that the row goes to (it's permutation) using the previously acquired permutation, and find the coefficient of that row by multiplying the corresponding coefficient from the first monomial matrix and the coefficient from the correct row of the second monomial matrix (found by using the first monomial matrix's permutation and the current row).

```
# generateIdentity generates an identity matrix of size n x n

generateIdentity := function(n)

local curr, result;

curr := 1;
result := [[],[]];

while curr <= n do

    result[1][curr] := curr;
    result[2][curr] := 1;

    curr := curr+1;

od;

return result;

end;
```

The *generateIdentity* function is similar to the last implementation except now the row number is continuously added to the first inner list and 1 is always added to the second list.

```
# monomialInverse gives us the inverse of a monomial matrix

monomialInverse := function(mono)

local result, row;

if not checkMonomial(mono) then return false;
fi;

result := [[],[]];

# Get permutation matrix first
result[1] := ListPerm(PermList(mono[1])^-1);

# fill permutation if necessary
while Length(result[1]) < Length(mono[1]) do

    Append(result[1], [Length[result[1]]+1]);

od;

row := 1;

while row <= Length(mono[1]) do

    # get coefficient entry
    Append(result[2], [(mono[2][row^PermList(mono[1])])^-1]);

    row := row+1;

od;

return result;

end;
```

Lastly, we must implement how to find the inverse of a monomial matrix represented in this way. First, as always, we check that we are given a well-formed monomial matrix. Finding the permutation of the permutation portion of the monomial matrix is simple, we need only change the list to an actual permutation, find it's inverse using GAP, and then convert it back to a list. Note that we must also make sure that this resulting permutation list is long enough because if any of the biggest points are not permuted they will not appear in the permutation or the inverse of that permutation, and hence they will not show up in the resulting list either. Finally, we get the non-zero element in the scalar portion of the inverse of the monomial matrix by taking the inverse of the scalar at which that row is permuted to.

Here is a look at why the method used for finding the inverse works. Say we have monomial matrix M and we want to find it's inverse, call it M'. Then we know $MM' = I$, where I is the identity matrix. Furthermore, we can represent both M and M' as the product of a permutation matrix and diagonal matrix. Thus $PDP'D' = I$, where P and P' are permutation matrices and D and D' are diagonal matrices and $M = PD$ and $M' = P'D'$. Note that diagonal matrices cannot permute the column that the non-zero entry of each row is located. Thus the only way to get an identity matrix would be if $P' = P^{-1}$. Now we have $PDP^{-1}D' = I$ and, because all of these matrices are invertible (proven in the previous section), $D' = P^{-1}DP$. From here we can find that if the diagonal entries of D are $[d_1, ..., d_n]$, then the entries of D' should be $[{d_{1^Q}}^{-1}, ..., {d_{n^Q}}^{-1}]$. Thus we must use the permutation on the diagonal matrix to find the scalar entry, then take it's inverse, in order to find the scalars for the inverse of a given monomial matrix.

4.1.4 Creating The Data Structure

Now that we know how we want to represent monomial matrices, and how to find the product of two monomial matrices, the identity monomial matrix, and the inverse of a monomial matrix, the following will allow us to create objects of type $MonomialPermutation$:

```
DeclareCategory( "IsMonomialPermutation",
    IsMultiplicativeElementWithInverse and IsAssociativeElement and
        IsFiniteOrderElement );

DeclareCategoryCollections( "IsMonomialPermutation" );

InstallTrueMethod( IsGeneratorsOfMagmaWithInverses,
    IsMonomialPermutationCollection );

BindGlobal( "MonomialPermutationsFamily",
    NewFamily( "PermutationsFamily", IsMonomialPermutation,
               CanEasilySortElements, CanEasilySortElements ) );

BindGlobal( "MonomialPermutationDefaultType",
    NewType( MonomialPermutationsFamily,
             IsMonomialPermutation and IsComponentObjectRep ) );

BindGlobal( "MonomialPermutation", function( permpart, scalarpart )
    if not (IsPerm(permpart) and IsList(scalarpart)) then
        Print("Monomial must be a permutation and list of scalars\n");
        return false;
    elif LargestMovedPoint(permpart) > Length(scalarpart) then
        Print("Permutation does not match scalar list\n");
        return false;
    elif 0 in scalarpart then
        Print("Cannot have a scalar of 0\n");
        return false;
    else
        return Objectify( MonomialPermutationDefaultType,
        rec( perm:= permpart, scal:= scalarpart ) );
        fi;
    end);
```

Take note of the final *BindGlobal* function. A $MonomialPermutation$ is defined to have two parts, a permutation part and a scalar part. We have slightly altered our representation to include an actual permutation instead of a list representing a permutation. This will avoid the need to convert back and forth using the *PermList* and *ListPerm* functions. First we check that the permutation is in fact a permutation and that the scalar part is a list. Then we check that the largest moved point of the permutation is less than or equal to the length of the scalar part. Otherwise, we would be permuting cosets with the permutation that are not represented in the scalar part. Finally, we check that none of our scalars are 0. If all of these conditions are met then the $MonomialPermutation$ is stored as a record with *perm* as the permutation and *scal* as the list of scalars.

Next, we must utilize our identity, multiplication, and inverse functions in the data structure, along with other methods that all data structures are required to have.

```
InstallMethod( PrintObj,
    [ IsMonomialPermutation ],
    function( monperm )
    Print( "MonomialPermutation( ", monperm!.perm, ", ",
           monperm!.scal, " )" );
    end );

InstallMethod( \=,
    [ IsMonomialPermutation, IsMonomialPermutation ],
    function( monperm1, monperm2 )
    return monperm1!.perm = monperm2!.perm and
```

```
            monperm1!.scal = monperm2!.scal;
    end );

InstallMethod( \<,
    [ IsMonomialPermutation, IsMonomialPermutation ],
    function( monperm1, monperm2 )
# Perhaps change the definition of ‘<’.
    if monperm1!.perm < monperm2!.perm then
      return true;
    elif monperm1!.perm > monperm2!.perm then
      return false;
    elif monperm1!.scal < monperm2!.scal then
      return true;
    else
      return false;
    fi;
    end );
```

The above methods are somewhat unimportant for out purposes. The first allows a *MonomialPermutation* to be printed, printing out *MonomialPermutation*(*perm*, *scal*) where *perm* is the permutation and *scal* is the list of scalars of the monomial matrix. The second determines that two *MonomialPermutations* are equal if their permutation are the same and their list of scalars are the same. The third claims that a monomial matrix is < another monomial matrix if it's permutation is < the other's permutation, or if they are the same, if the list of scalars is < the other. Again, this is irrelevant for our uses, but creating a data structure in GAP requires this functionality.

```
InstallMethod( \*,
    [ IsMonomialPermutation, IsMonomialPermutation ],
    function( monperm1, monperm2 )
# Enter your code here!

    local perm, result, row, inverse;

result := [];
row := 1;

perm := monperm1!.perm * monperm2!.perm;

while row <= Length(monperm1!.scal) do

    # get coefficient entry
    Append(result,
        [monperm1!.scal[row]*monperm2!.scal[row^(monperm1!.perm)]]);

    row := row+1;

od;

    return MonomialPermutation( perm , result );
    end );
```

This method allows two *MonomialPermutations* to be multiplied together. This is taken almost directly from the previous *multiplyMonomials* function, the only differences being that, given a monomial matrix *monperm*, *monperm*!.*scal* is being used to get the list of scalars, *monperm*!.*perm* is begin used to get the permutation, and we can directly calculate the resulting permutation since our we are now utilizing a permutation instead of a list representing a permutation.

```
InstallMethod( InverseOp,
    [ IsMonomialPermutation ],
    function( monperm )
# Enter your code here!

local result, row, perm;

result := [];
row := 1;
perm := (monperm!.perm)^-1;

while row <= Length(monperm!.scal) do

    # get coefficient entry
    Append(result, [monperm!.scal[row^perm]^-1]);


    row := row+1;

od;

    return MonomialPermutation( perm , result );
    end );
```

Similarly to the function for multiplying two *MonomialPermutations*, this function for finding the inverse of a *MonomialPermutation* is taken almost directly from it's previous counterpart. Again, the only differences lie in how we are acquiring the permutation and list of scalars, and finding the permutation portion of the resulting inverse is much easier now that we are using permutations.

```
InstallMethod( OneOp,
    [ IsMonomialPermutation ],
    function( monperm )

    local i, result, entry;
    entry := monperm!.scal[1];
    result := [];

    result:= List ( [1..Length(monperm!.scal)],  x->entry^0 );

    return MonomialPermutation( (), result );
    end );
```

This method, for finding the identity element of a $MonomialPermutation$, is fairly different from it's counterpart above. Including the identity permutation is a simple change, but in order to fill the scalar part of the $MonomialPermutation$ with identity elements, we take the corresponding entry in the given $MonomialPermutation$ and raise it to the 0 power. We also use the length of the given $MonomialPermutation$'s list of scalars to ensure that our identity element is the correct size.

4.2 Representing A Given Group With Monomial Matrices

Now that we have a data structure for monomial matrices, we will tackle our original issue of representing a given group using monomial matrices. In order to do this, we will observe the necessary functions in the sequential order in which they would be called.

4.2.1 MonomialTable

The ultimate goal will be to create a table of monomial matrices similar to the table of marks.

```
# MonomialTable returns a table based on all the subgroups of a
#    given group, their MonomialPermutation representations,
#    and their fixed points

MonomialTable := function(group)

    local allMonos, table, gens, sub, fixed, i, sum, row, mono,
        imagegroup, phi, subgens, imgsubgens, tom, j, record;

    allMonos := AllMonomials(group);
    table := [];
    gens:=GeneratorsOfGroup(group);
    tom := TableOfMarks(group);

    # Go through each subgroup
    for j in [1..Length(allMonos[2])] do

    #for sub in allMonos[2] do

        sub := allMonos[2][j];

        for mono in sub[2] do

            # get fixed points
            row := [];

            imagegroup:=Group(mono);
            phi:=
              GroupHomomorphismByImages(group, imagegroup, gens,mono);

            for i in [1..Length(OrdersTom(tom))] do

                subgens:=
                  GeneratorsOfGroup(RepresentativeTom(tom,i));

                # Get the identity if necessary
                if (Length(subgens) = 0) then
                    subgens :=
                      [ GeneratorsOfGroup(RepresentativeTom(tom,2))[1]^0 ];
                fi;

                imgsubgens:=List(subgens,x->x^phi);

                record := allMonos[1][i];
                fixed := FixedPoints(imgsubgens, record);
                Append(row, [fixed]);
            od;
        od;

        Append(table, [row]);

    od;

    return table;

end;
```

As previously stated, we want to ultimately create a table that uniquely represents a given group using monomial matrices. To do this, we will need all of the monomial matrices using all of the subgroups of the group, which will be given to us from *AllMonomials* (see code below). We will also need the generators of the group (from *GeneratorsOfGroup*) and the table of marks for the group (from *TableOfMarks*). *AllMonomials* will return a list of records for each subgroup and a list containing separate lists for each subgroup of the given group. Within each of these inner lists will be two lists, the first containing the subgroup itself (like a label for the list) and the second containing all the *MonomialPermutations* for the subgroup in the given group. Each subgroup will have it's own row in our final table, so we first iterate through each of the lists provided by *AllMonomials*. Then we iterate through each of the monomial matrices within the subgroup. We must create a homomorphism from the given group to the group of *MonomialPermutations* in order to work with them. Now we pass information to *FixedPoints* (see code below) in order to find the linear characters that are fixed. This will be explained further in the description of *FixedPoints*. In the end, the list of these fixed linear characters will uniquely describe the given group.

4.2.2 FixedPoints

```
# FixedPoints takes a group (of MonomialPermutations) and
#    determines which points in the permutation are fixed.
#    It then finds the linear character corresponding to
#    fixed points and returns this.

FixedPoints := function(gens, record)

    local result, size, isFixed, gen, i, point, sum, lin, class, isLin;

    result := [];
    size := Length(gens[1]!.scal);
    isFixed := true;
```

```
# Go through each possible point
for point in [1..size] do

    # Reset this boolean
    isFixed := true;

    # Go through each generator until the point is not fixed
    for gen in gens do

        # If not fixed, set isFixed and break
        if not (point^(gen!.perm) = point) then
            isFixed := false;
            break;
        fi;

    od;

    # if fixed, find the linear character
    if isFixed then

        # go through all linear characters, find matching one
        for lin in record.subLin do

            # Reset isLin
            isLin := true;

            for class in [1..Length(gens)] do

                if not (lin[record.conj[class]] =
                    gens[class]!.scal[point]) then

                    isLin := false;

                fi;

            od;

            # Check if this is the correct linear character
            if isLin then

                Append(result, [lin]);
                break;

            fi;
            od;

            # Check that we found a linear character
            if not isLin then
                Append(result, ["NOT HERE"]);
                #Error("Linear character not found");
            fi;

        fi;

    od;

    return result;

end;
```

Given a list of generators of a group and a *rec* (record in GAP) of information, we want to determine the linear characters corresponding to fixed points in the group. First we determine the points that could be possibly fixed by the number of elements in the list of scalars and iterate from 1 to this size. Iterating through these points, first we determine if the point is fixed by checking that it is fixed by every generator. Then, if the point is fixed, we iterate through every linear character (a list of which is found in the record passed in) and determine which one corresponds to this fixed point. We do this by checking, for each linear character, if the conjugacy class that corresponds to each generator (stored previously, see *Homomorphism* function below) has the same value in the linear character as the scalar of this generator. If this is the case, then we append this linear character to our result and break out of the loop, as we have found the one and only linear character corresponding to this fixed point. Finally, we check that, for every fixed point, we have found the corresponding linear character. If we have not we can either include the string 'NOT HERE' in the result to indicate where the error has occurred or use GAP to throw an error.

4.2.3 AllMonomials

```
# AllMonomials takes a group and initializes a record for the
#    group and each of it's subgroups and then calls Monomial
#    for each of these records

AllMonomials := function(group)

local tom, subs, x, sub, output, record, records;
```

```
# Get the Table Of Marks
tom := TableOfMarks(group);
group := UnderlyingGroup(tom);
records := [];

# subs will hold all the subgroups
subs := List([1..Length(OrdersTom(tom))],
   x -> RepresentativeTom(tom,x));

output := [];

for sub in subs do

    record := MonomialInit(group,sub);
    Append(records, [record]);
    Append(output, [[sub, Monomial(record)]]);

od;

return [records,output];

end;
```

AllMonomials is used to take a group and provide a list of two lists: records of information for every subgroup and *MonomialPermutations* using every subgroup. This latter list contains a list for each subgroup which also has two inner lists, the first of which tells the subgroup and the second of which holds all the *MonomialPermutations* using that subgroup in the given group. First, we obtain the table of marks for the given group. From this we can get all the subgroups (up to conjugacy) of the group. For each of these subgroups, we get a record of information from *MonomialInit* (described below) and all the monomial matrices using the subgroup from *Monomial* (also described below). We then return all of this information as a list in the aforementioned way.

4.2.4 MonomialInit

```
# MonomialInit is passed a group and subgroup and initializes
#    a record that holds data for further use in the other functions

MonomialInit := function(group, subgroup)

local info, norm;
```

```
norm := Normalizer(group,subgroup);

info := rec(group := group, sub := subgroup,
    reps := RightTransversal(group,subgroup),
    gens := GeneratorsOfGroup(group), norm := norm,
    subConj := ConjugacyClasses(subgroup),
    subLin := LinearCharacters(subgroup),
    subGens := GeneratorsOfGroup(subgroup),
    normGens := GeneratorsOfGroup(norm));

return info;

end;
```

MonomialInit takes in a group and subgroup and returns a record of all information that may be useful for other functions throughout the process of representing a group with monomial matrices. This function is used so that other functions need only be passed the record, and thus all information in the record need be calculated only once. The following is a description of everything contained in the record returned by *MonomialInit*.

group is the group given to *MonomialInit*.
sub is the subgroup given to *MonomialInit*.
reps are coset representatives for the subgroup in the group.
gens are generators for the given group.
norm is the normalizer of the subgroup in the group.
subConj are the conjugacy classes of the subgroup.
subLin are the linear characters of the subgroup.
subGens are the generators of the subgroup.
normGens are the generators of the normalizer.

4.2.5 Monomial

```
# To Use: Call Monomial(info)
# where info is a record obtained by MonomialInit

Monomial := function(info)

local monos, homo, output, monperm, monomial, row, col, x,
    result, coeff, g, i, rep, elem, r;

#info should be a record created by MonomialInit
```

```
output := [];

for homo in Homomorphism(info) do

    monos := [];

    # go through each group element
    for g in info.gens do

        monomial := [[],[]];
        row := 1;
        col := 1;

        # Here we are filling the monomial matrix with 0's
        for i in info.reps do
            monomial[1][row] := 0;
            monomial[2][row] := 0;
            row := row+1;
        od;

        # Reset row, as we will use it later
        row := 1;

        # go through each coset representative
        for rep in info.reps do

            # x is the element times the representative
            x := rep*g;

            # check which coset it is in, coefficient it should have
            for elem in info.sub do
                result := elem*x;
                coeff := elem^homo[1];

                # Go through and set the coefficent for each col
                col := 1;
                for r in info.reps do

                    if result = r then
                        monomial[1][row] := col;
                        monomial[2][row] :=
                            CyclicToCyclotomic(coeff, homo[2]);
                    else;
                    fi;
                    col := col+1;
                od;
            od;

            #increase row
            row := row+1;
        od;

        monperm :=
            MonomialPermutation(PermList(monomial[1]),monomial[2]);

        Append(monos, [monperm]);

    od;

    Append(output, [monos]);

od;

return output;

end;
```

The above function is used to find all *MonomialPermutations* for a group and subgroup of that group. Note that *Monomial* should be passed a record created by *MonomialInit* that contains the group, subgroup, and other information. The first thing we do is iterate through homomorphisms from the subgroup to elements of a cyclic group (provided by the *Homomorphism* function, which is described below). Then, for each homomorphism, we look at each generator of the group and determine the corresponding *MonomialPermutation*. We calculate the monomial matrix as a list of columns and scalars (see section Storing Only Column And Scalar above) and then use *PermList* to provide the list of columns as a permutation. We do this using the same algorithm described in the Monomial Matrices section. For each coset representative we multiply the representative and the group generator and then use the subgroup generators to determine which column the result will be in (using the list of coset representatives) and the scalar (using the given homomorphism). Note that the homomorphism goes to an element of a cyclic group and must be converted to a root of unity. For further explanation of the process and correctness in creating the monomial matrix representation refer to the Monomial Matrices section above.

4.2.6 Homomorphism

```
# Homomorphism takes a record from MonomialInit
#    and returns all the homomorphisms of the group
#    into the smallest possible cyclic group using the group's
#    conjugacy classes, linear characters, and generators
#    (from info)

Homomorphism := function(info)

local redOrbits, nonRed, gen, classes, homos, homo, conj, class,
    elem, base, root, power, roots, cyclRoots, lcm, cyclic, i, r,
    mult, homomorphism;

# classes holds which class each generator is in
classes := [];

homos := [];
redOrbits := Redundants(info);
nonRed := [];

# utilize the Redundants function to use only necessary homomorphisms
for homo in [1..Length(redOrbits)] do

    # We will only use the first from each orbit
    Append(nonRed, [info.subLin[redOrbits[homo][1]]]);

od;

# Add the identity permutation if subgroup is the identity
if info.subGens = [] then info.subGens := [info.gens[1]^0];
fi;

# find which conjugacy class each generator is in
for gen in info.subGens do

    for i in [1..Length(info.subConj)] do

        if gen in info.subConj[i] then

            Append(classes, [i]);

        fi;

    od;

od;

info.conj := classes;
```

```
# go through each linearCharacter
# find the corresponding homomorphism
for homo in nonRed do

    # roots holds which root of unity
    # corresponds to each generator
    roots := [];

    # cyclRoots holds the corresponding
    # elements of the cyclic group
    cyclRoots := [];

    # go through each conjugacy class and find
    # the corresponding root of unity
    for class in classes do

        elem := homo[class];

        base := Order(elem);
        root := E(base);
        power := 1;
        while not (root^power = elem) do
            power := power+1;
        od;

        # each will be kept as a pair [base, power]
        Append(roots, [[base, power]]);

    od;

    # find the LCM of all the bases
    lcm := Lcm(List ( [1..Length(roots)],  x->roots[x][1] ));

    # cannot use cyclic group of just 1, so make at least 2
    if lcm = 1 then
        cyclic := CyclicGroup(2);
        # convert each pair in roots to an element of cyclic
        for r in roots do

            Append(cyclRoots, [ (cyclic.1)^(0) ] );

        od;

    else
        cyclic := CyclicGroup(lcm);
        # convert each pair in roots to an element of cyclic
        for r in roots do

            mult := lcm/(r[1]);
            Append(cyclRoots, [ (cyclic.1)^(mult*r[2]) ] );

        od;
    fi;

    homomorphism :=
        GroupHomomorphismByImages(info.sub,cyclic,info.subGens,cyclRoots);
    Append(homos, [[homomorphism, cyclic]]);

od;

return homos;

end;
```

The *Homomorphism* function uses a record from *MonomialInit* to find all non-redundant homomorphisms from a subgroup to elements of a cyclic group, using linear characters and other information from the given record. First, we utilize the *Redundants* function (see below) to get only non-redundant linear characters from the linear characters of the subgroup in the given record. Next we find which conjugacy class each generator is in (storing only the integer representative of this conjugacy class because the order in which the classes are stored does not change) and store this in our record as well for further use. Then for each of these linear characters, we must determine the corresponding homomorphism. We will use the function *GroupHomomorphismByImages*, but GAP cannot create homomorphisms into roots of unity ($R(\mathbb{C})$), so we convert from roots of unity to elements of a cyclic group (for the homomorphism) and can convert to roots of unity later. In order to do this, we first take each root of unity from the linear character and convert it to a pair containing which root of unity we have and what power it is raised to. Then we use the least common multiple of the bases to find the cyclic group that we should utilize and convert these roots of unity to elements of this cyclic group. Finally, we use GAP's *GroupHomomorphismByImages* to get the homomorphism into the cyclic group and also pass along the cyclic group itself, so we can later convert these elements back into roots of unity.

4.2.7 Redundants

```
# Redundants takes record created by MonomialInit
# It returns the orbits of a group of linear characters

Redundants := function(info)

local gen, class, rep, new, class2, list, perm, permGroup, orbits,
    homo, newHomo, homPerms, homList, homo2;

homPerms := [];

for gen in info.normGens do

    list := [];
    homList := [];

    for class in [1..Length(info.subConj)] do

        rep := Representative(info.subConj[class]);
        new := (gen^-1) * rep * gen;

        for class2 in [1..Length(info.subConj)] do

            if new in info.subConj[class2] then list[class] := class2;
            fi;

        od;

    od;

    perm := PermList(list);

    # Find homomorphisms based on permutation of conj. classes
    for homo in info.subLin do

        newHomo := [];

        # Get the new Homomorphism
        for class in [1..Length(homo)] do

            Append(newHomo, [homo[class^perm]]);

        od;

        # Find which one matches the new Homomorphism
        for homo2 in [1..Length(info.subLin)] do
```

```
            if info.subLin[homo2] = newHomo
                then Append(homList, [homo2]);
            fi;

        od;

    od;

    Append(homPerms, [PermList(homList)]);

od;

permGroup := Group(homPerms);

orbits := Orbits(permGroup, [1..Length(info.subLin)]);

return orbits;

end;
```

Redundants is given a record (from *MonomialInit*) and finds the orbits of the linear characters in that record and returns them. This is important because we need only consider linear characters in different orbits, and we can discard the rest (as is done in the *Homomorphism* function). To do this, we iterate through the generators of the normalizer of the given subgroup in the group. For each conjugacy class of the subgroup, we take a representative and find it's conjugate with the generator. We then find what conjugacy class this conjugate is in and record that, creating a permutation of where the conjugacy classes go when they are conjugated by the generator. Now we iterate through the linear characters and use the permutation of the conjugacy classes to determine a permutation in the linear characters (which linear character goes to which other linear character when it's conjugacy classes are permuted in the given way). We then make this list into a permutation, and all of these permutations (for each generator) into a group. Then we can use GAP to find the orbits of this group and return these orbits, so we can only utilize one linear character from each.

4.2.8 CyclicToCyclotomic

```
# CyclicToCyclotomic takes an element of a cyclic group and
#    converts it to a Cyclotomic (root of unity)

CyclicToCyclotomic := function(elem, group)
```

```
local base, power, cyc;

base := Order(group);

# check if we have the identity
if base = 1 then
    return E(1);
fi;

power := 1;

# find the power
while not (((group.1)^power) = elem) do
    power := power+1;
od;

return E(base)^power;

end;
```

CyclicToCyclotomic converts an element of a cyclic group to a cyclotomic element (a root of unity, $R(\mathbb{C})$), given the element and the group it is in. Which root of unity to use is determined from the order of the cyclic group provided. If the order is 1 we return $E(1)$ which is just 1. Otherwise we continue to raise the root of unity to a higher power until we reach a root of unity that is equal to the element given. At this point, we are done and return the correct root of unity.

5 Results

Now we can use all of the functions above to run *MonomialTable* and get the table output of fixed monomial matrices. Here are some examples, providing only the number of fixed linear characters in each entry of the table, for simplicity of viewing. The runtime for the function is given as well.

The command $MonomialTable(SymmetricGroup(3))$ runs in 125 ms and gives us:

$$\begin{pmatrix} 6 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 2 & 0 & 2 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

The command $MonomialTable(SymmetricGroup(4))$ runs in 714 ms and gives us:

$$\begin{pmatrix}
24 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
12 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
12 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
8 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 6 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 2 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\
6 & 2 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\
4 & 0 & 2 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
3 & 3 & 1 & 0 & 3 & 1 & 1 & 0 & 1 & 0 & 0 \\
2 & 2 & 0 & 2 & 2 & 0 & 0 & 0 & 0 & 2 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}$$

The command $MonomialTable(SymmetricGroup(5))$ runs in 2656 ms and gives us:

$$\begin{pmatrix}
120 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
60 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
60 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
40 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
30 & 0 & 6 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
30 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
30 & 6 & 2 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
24 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
20 & 6 & 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
20 & 0 & 4 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
20 & 2 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
15 & 3 & 3 & 0 & 3 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
12 & 0 & 4 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
10 & 0 & 2 & 4 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\
10 & 4 & 2 & 1 & 0 & 0 & 2 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
6 & 0 & 2 & 0 & 0 & 2 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
5 & 3 & 1 & 2 & 1 & 1 & 1 & 0 & 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
2 & 0 & 2 & 2 & 2 & 0 & 0 & 2 & 0 & 2 & 0 & 0 & 2 & 2 & 0 & 0 & 0 & 2 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}$$

The command $MonomialTable(AlternatingGroup(4))$ runs in 209 ms and gives us:

$$\begin{pmatrix} 12 & 0 & 0 & 0 & 0 \\ 6 & 2 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 \\ 3 & 3 & 0 & 3 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The command $MonomialTable(AlternatingGroup(5))$ runs in 557 ms and gives us:

$$\begin{pmatrix} 60 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 30 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 20 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 15 & 3 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 12 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 10 & 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 6 & 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 5 & 1 & 2 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

References

[1] Curtis, Charles W. *Linear Algebra: An Introductory Approach.* Boston: Springer, 1974.

[2] Fraleigh, John B. *A First Course In Abstract Algebra.* Boston: Addison Wesley, 7th Edition, 2003.

[3] Gallian, Joseph A. *Contemporary Abstract Algebra.* Duluth: Brooks Cole, 7th Edition, 2009.

[4] Lux, K. and H. Pahlings. *Representations Of Groups: A Computational Approach.* Cambridge: Cambridge University Press, 2010.